



# Manual de Javascript

JS

Miguel Angel Alvarez  
Manu Gutierrez  
Eduard Tomàs



desarrolloweb.com

[desarrolloweb.com/manuales/manual-javascript.html](https://desarrolloweb.com/manuales/manual-javascript.html)

# Introducción: Manual de JavaScript

Manual de Javascript desde cero. Adéntrate en el lenguaje de programación más popular de la web y conoce todas las características de la programación del lado del cliente.

Javascript es el lenguaje de programación usado para las páginas web, compatible con todos los navegadores y que forma un estándar de desarrollo que ahora también se extiende a dispositivos o programas de propósito general multiplataforma.

Este manual de Javascript está dividido en dos grandes partes. En la primera parte veremos las características fundamentales del lenguaje, como su sintaxis, variables, estructuras de control, funciones, arrays, etc. Toda la información que encontrarás en la primera parte te sirve para programar Javascript a nivel general, sea donde sea el entorno de ejecución de tus programas.

En la segunda parte, que también encontrarás en esta misma página, ahondaremos en el uso de Javascript en el ámbito del navegador, es decir, aprenderemos a manejar los recursos que nos ofrece el navegador para hacer páginas interactivas, capaces de interactuar con el usuario de manera avanzada.

Encuentras este manual online en:

<https://desarrolloweb.com/manuales/manual-javascript.html>

## Autores del manual

Las siguientes personas han participado como autores escribiendo artículos de este manual.

### Miguel Angel Alvarez

Fundador de DesarrolloWeb.com y la plataforma de formación online EscuelaIT. Comenzó en el mundo del desarrollo web en el año 1997, transformando su hobby en su trabajo.



### Manu Gutierrez

### Eduard Tomàs

Apasionado de la informática, los videojuegos, rol y... la cerveza. Key Consultant en Pasiona y MVP en #aspnet



# Introducción a Javascript

Artículos que nos servirán para introducirnos en este lenguaje, aprendiendo los conceptos más básicos de Javascript y la programación del lado del cliente.

## Cómo y por qué aprender programación con Javascript

**Una descripción completa sobre el ecosistema Javascript, qué se puede hacer con el lenguajes, como comenzar el estudio y cuáles son las tecnologías que rodean el desarrollo frontend y backend con Javascript.**



En este artículo pretendemos orientar a las personas que **comienzan a estudiar Javascript** y a aquellas que inician el estudio de la disciplina de programación y desean acercarse a este mundo usando un lenguaje sencillo para aprender y **útil en el ambiente profesional**.

Comenzar con Javascript es sencillo, pero hay un largo camino por delante si se pretende llegar a exprimir todas las posibilidades del lenguaje. **Javascript es uno de los lenguajes estándares de la web** y por tanto ideal para muchos de los profesionales que piensan dedicarse a este medio. Pero incluso, aunque no quieras desarrollar específicamente para la web, Javascript es una excelente alternativa para hacer aplicaciones móviles o aplicaciones de escritorio.

Quizás no todos los desarrolladores necesiten llegar a un nivel avanzado de Javascript, pero sin duda un conocimiento básico será de gran ayuda en muchos momentos de su carrera profesional. En cualquier caso, completado el aprendizaje, tendrás a tu alcance un sin fin de oportunidades.

En este artículo que da inicio al [Manual de Javascript](#) queremos hacer una especie de prólogo que nos ofrezca una vista de pájaro sobre todo el ecosistema Javascript. Comenzaremos analizando el estado de Javascript en la actualidad (actualizado a 2022) y los motivos por los que merece la pena usar Javascript en general. Luego nos ocuparemos de explicar cómo se puede aprender Javascript y llegar a cualquier nivel que nos propongamos.



## Javascript está en todas partes

Javascript se ha convertido en un lenguaje idóneo y muy recomendable para aprender a todos los niveles, por disponer de **muchas y variadas aplicaciones**, además de aportar sencillez para las personas que comienzan. Para ejecutarlo necesitamos tan solo un navegador, aunque en la actualidad Javascript ha sobrepasado el ámbito de los clientes web, para situarse en casi cualquier parte.

### Javascript en la web

El entorno donde Javascript apareció en primer lugar fue la web. Su ejecución se centraba en el ámbito de un documento HTML y permitía a los desarrolladores aportar interactividad, manipular el documento o la ventana del navegador, realizar cálculos, etc. Netscape Navigator, un navegador desaparecido a día de hoy, tiene el honor de haber introducido Javascript como lenguaje, aunque hoy lo soportan todos los clientes web con los que un usuario común pueda llegar a navegar.

Usar Javascript para la web, en el entorno del navegador, también se conoce como **programación del lado del cliente** y es una de las actividades englobadas en el término "frontend". Hoy sigue siendo el ambiente más habitual de ejecución de Javascript, pero realmente es solo una más entre sus posibilidades.

### Javascript en el backend

A ciertos desarrolladores se les ocurrió que podrían extraer el motor de ejecución de Javascript, que hasta entonces sólo se disponía dentro del ámbito del navegador, para usarlo en cualquier otro propósito fuera del cliente web. Así es como nació [NodeJS](#), que no es más que una **plataforma para la ejecución de Javascript fuera del navegador**.

Con NodeJS podemos programar con Javascript aplicaciones que se ejecutan directamente sobre el sistema operativo y que son capaces de resolver cualquier tipo de problema. Con esta tecnología Javascript se convirtió en un lenguaje de programación de propósito general.

Uno de los usos más habituales de NodeJS es la programación "backend", que permite programar aplicaciones que son capaces de ejecutarse en el servidor, proporcionando acceso a bases de datos, el sistema de archivos y cualquier otro recurso del lado del servidor. Sin embargo, NodeJS es tan amplio que se puede usar para muchas otras tareas, como la automatización, optimización o despliegue de aplicaciones, entre otras operaciones.

### Javascript como lenguaje para apps de dispositivos

Desde hace años también es posible usar Javascript como lenguaje para la creación de aplicaciones para dispositivos (apps para móviles, tablets, TVs...). Las aplicaciones desarrolladas se instalan desde los correspondientes app stores de los principales sistemas móviles y el usuario en principio no percibe ninguna diferencia de éstas con respecto a las aplicaciones desarrolladas con los lenguajes nativos. Pero, por el hecho de ser programadas con Javascript y HTML5, abren un **nuevo**



**campo de actuación para las personas experimentadas en el desarrollo para la web y dispensa del aprendizaje de lenguajes nativos para cada plataforma móvil.**

Al principio, para ejecutar aplicaciones realizadas en Javascript y HTML5 se necesitaba de un "Web View". Básicamente la función del web view consiste en ofrecer un marco para la ejecución de la app, de modo que ésta se ejecuta dentro de un navegador, aunque el usuario no lo perciba. Esta situación tiene diversas ventajas e inconvenientes en los que no vamos a entrar, pero hoy también existe la posibilidad de usar Javascript como lenguaje de desarrollo de aplicaciones que se compilan a los lenguajes nativos, que no requieren un web view para funcionar.

Ejemplos de frameworks para el desarrollo de aplicaciones basadas en web view tenemos a Apache Cordova, [Phone Gap](#) o [Ionic](#). Ejemplos de frameworks para desarrollo de aplicaciones nativas usando Javascript tenemos a Native Script y React Native. Todas las alternativas tienen la importante ventaja de producir apps para Android e iOS con la misma base de código, así como para otros sistemas minoritarios.

### Javascript como lenguaje para aplicaciones de escritorio

Otro de los ámbitos en los que Javascript ha penetrado con fuerza es en el desarrollo de aplicaciones para ordenadores personales. Con Javascript somos capaces de crear aplicaciones avanzadas de **interfaz gráfica por ventanas**, capaces de usar todos los recursos de un ordenador y además ejecutarse en cualquier sistema operativo que necesitemos.

Usar Javascript para aplicaciones de escritorio es sencillo gracias a proyectos como Electron, que nos permite producir aplicaciones multiplataforma, es decir, que se podrán instalar en Windows, Mac OS X y Linux. Hay bastantes aplicaciones conocidas desarrolladas con Electron, como Atom, Visual Studio Code, Slack, Hyper, etc.

El secreto detrás de ElectronJS es que levanta un navegador con el motor de Chrome para ejecutar las aplicaciones. Por un lado podemos ejecutar el Javascript convencional en el propio navegador que nos proporciona, pero por otro lado permite la ejecución de NodeJS para los procesos críticos del sistema operativo, como el acceso al sistema de archivos o las bases de datos.

### Javascript en la web

Hasta este punto hemos visto los campos principales de actuación de Javascript. Hemos podido comprobar que con este lenguaje es posible hacer prácticamente cualquier cosa que nos propongamos, ya que proporciona alternativas para casi cualquier entorno de ejecución. Ahora vamos a analizar el ecosistema "JS" en la web.

El campo de actuación de Javascript más adecuado si queremos aprender el lenguaje es la web, de modo que es donde nos vamos a centrar a lo largo de todo el [Manual de Javascript](#). Si quieres



manuales sobre cómo se usa Javascript en otros ámbitos te recomendamos navegar por la [categoría de Javascript](#).

Dentro de la disciplina de desarrollo del lado del cliente (Javascript ejecutado dentro del entorno del navegador) podemos aplicarlo para desarrollar distintos tipos de proyectos, que requieren también distintos enfoques y conocimientos.

## Sitios web

Cuando nos referimos aquí a sitios web queremos indicar sitios donde la parte más importante es el contenido, ya sean blogs, páginas de noticias, e incluso comercio electrónico.

Javascript en estos casos se dedica a aportar funcionalidad e interacción, permitiendo disponer de interfaces de usuario dinámicas, respuesta a acciones del usuario, validación de formularios, etc.

## Single Page Applications

En los últimos años se ha popularizado la web como plataforma para aplicaciones de negocio. Aplicaciones denominadas "de gestión", que antes se ejecutaban con programas de escritorio, hoy tienen frontales web que nos permiten usarlas desde la nube, es decir desde cualquier navegador conectado a Internet y sin la necesidad de instalar un software en la máquina. En este tipo de aplicación es habitual portar mucha parte de la carga de procesamiento desde el lado del servidor, al ámbito del cliente. En este nuevo paradigma el navegador se encarga de hacer muchas más cosas que en sitios web tradicionales, como la creación del código HTML para visualizar los datos o la navegación entre pantallas o rutas de la aplicación.

En las [Single Page Applications](#), conocidas también con sus siglas SPA, es normal que el servidor entregue solamente los datos de negocio en crudo y que el navegador haga todo el trabajo de presentar esos datos en un formato adecuado (en el navegador se produce el HTML para representar esos datos). Pero lo que más caracteriza a una SPA es que la navegación se realiza siempre dentro de la misma página y Javascript se encarga de presentar una u otra pantalla al usuario sin tener que recargar todo el conjunto de la página.

Los dos principales factores que caracterizan las SPA son los siguientes:

1. El hecho de traer del servidor los datos en crudo (más ligeros, con menor consumo de transferencia y mayor velocidad)
2. Toda la navegación se realiza dentro del mismo documento, producen aplicaciones web de una respuesta muy rápida, aportando una experiencia de uso cercana a la de una aplicación de escritorio.

## Progressive Web Apps

Con un enfoque similar a las SPA tenemos las aplicaciones progresivas o [Progressive Web Apps](#) (PWA). Este tipo de aplicaciones permite el acceso a nuevas características de los navegadores para



conseguir desarrollar funcionalidades que anteriormente solamente estaban disponibles en aplicaciones nativas.

Las PWA pueden desarrollarse en una misma página como las SPA, pero también pueden ser sitios web tradicionales a los que se les hayan incorporado funcionalidades como la navegación offline, la posibilidad de instalarse en el dispositivo, recibir notificaciones push y mucho más.

## Librerías, frameworks y desarrollo "Vanilla Javascript" para la web

La creación de una SPA es una tarea bastante más avanzada que el desarrollo de un sitio web y para poder realizar ese trabajo es importante que el equipo de desarrollo se base en un framework Javascript, como podría ser [Angular](#), [React](#), VueJS, Ember o [Lit](#), entre otros muchos ejemplos.

### Nota:

React se considera más como una librería, pero con una serie de añadidos, que ellos mismos proporcionan en muchos casos, ofrecen tantas prestaciones como los que encontramos en un framework. Por si alguien no lo sabe todavía, **una librería es un conjunto de funciones, o clases y objetos, que nos permite realizar un abanico de tareas habituales** para el desarrollo de ciertas necesidades de aplicaciones. Un framework se distingue principalmente de una librería porque, además de proveer código para resolver problemas comunes, **ofrece una arquitectura que los desarrolladores deben seguir para producir las aplicaciones** y asegurarse una mejor calidad del código y mayor facilidad de mantenimiento. Dicho de otro modo, el framework además de ofrecer utilidades diversas, te marca un estilo y flujo de trabajo a la hora de desarrollar aplicaciones.

Por su parte, Lit (antes conocido como Lit-HTML) es una microlibrería que ocupa poco más de 5KB y que permite mayores utilidades para el desarrollo con Web Components, los cuales aclararemos un poco más abajo.

Para el desarrollo de sitios web generalmente sería suficiente con el uso de Javascript puro, sin necesidad de basarnos en ninguna librería adicional. Ese desarrollo de Javascript "puro" se conoce generalmente como "Vanilla Javascript".

**Nota:** Debe quedar claro que "Vanilla" no es ninguna marca comercial o ningún sabor de Javascript más allá que el del propio lenguaje. Es como una broma para indicar que con Javascript (y nada más) se pueden resolver todas las cosas que te ofrecen librerías y frameworks ya hechas.

Sin embargo, es también habitual que en el desarrollo de sitios web se usen librerías como [jQuery](#). jQuery es un conjunto de objetos y funciones (código de utilidad general) que tiene el objetivo



ayudarte a **manipular la página dinámicamente**, salvando las diferencias entre los distintos navegadores y permitiendo escribir un único código que se ejecuta correctamente en cualquier cliente web. Además, jQuery te ofrece muchas funciones que realmente resultan útiles para el desarrollo de muchas tareas habituales de los sitios web, que puedes usar de una manera más rápida que si trabajajes solo con Javascript.

**Nota:** Con respecto a jQuery es conveniente mencionar que hay una corriente de desarrolladores que advierten que usar jQuery no es absolutamente necesario en la actualidad, para la mayoría de los casos. Usar jQuery no está mal, pero muchas personas lo implementan para resolver necesidades que un poco de Javascript "Vanilla" es capaz de realizar. A veces se carga jQuery de manera predeterminada, quizás por comodidad, por dejarse llevar o simplemente por desconocimiento del propio lenguaje Javascript, y sin embargo se usan muy pocas de sus funciones. Hay librerías más especializadas para resolver todas y cada una de las cosas que una librería generalista como jQuery ofrece y que ocupan mucho menos peso para la descarga y tiempo de procesamiento para los navegadores.

Pero más allá de librerías y frameworks, si estamos aprendiendo **nos debemos concentrar primero en dominar Javascript**. Muchas personas desean ir muy rápido y se lanzan de cabeza a aprender algo como jQuery o React sin tener las bases necesarias de Javascript, lo encuentran difícil y se frustran. Es importante no empezar la casa por el tejado y no querer quemar etapas, lo que nos ayudará en el aprendizaje a todos los niveles.

## APIs HTML5

Además del propio lenguaje debemos saber que hoy existen en los navegadores muchas otras tecnologías basadas en Javascript para resolver una amplia gama de necesidades.

Con la llegada de HTML5 se produjo una estandarización mayor de los navegadores, llegando a un compromiso por los fabricantes de apoyar los lenguajes de la web (HTML + CSS + Javascript) tal como dictaban sus especificaciones. Pero además produjo una corriente abundante de nuevas especificaciones para trabajar con la más variada gama de recursos del navegador, ordenador o dispositivo.

**HTML5 ofrece APIs para el trabajo con una variada gama de recursos del navegador** como la cámara, geolocalización, almacenamiento, dibujo bitmap o vectorial, audio, vídeo, etc. Todo lo que ofrece HTML5 está disponible en todos los navegadores y forma parte del kit de herramientas del desarrollador Javascript. Podemos usar las API HTML 5 sin necesidad de cargar ninguna librería o framework.

## Web Components

La última revolución del desarrollo para la web, que ya en 2022 está totalmente extendido en los navegadores, se llama Web Components. Se basa en un nuevo API (con varias especificaciones en



conjunto) encaminado a crear componentes personalizados. Los componentes personalizados o "**Custom Elements**" son como nuevas etiquetas del HTML que cualquier desarrollador puede crear para resolver problemas comunes o particulares de las aplicaciones.

Con los web components los desarrolladores podrán extender el HTML creando nuevos componentes capaces de hacer cualquier cosa y con avanzadas capacidades de encapsulación, para respetar su autonomía y que sean capaces de usarse en cualquier proyecto, maximizando la reutilización del código y sin la necesidad de basarse en cualquier tipo de librería o framework.

Al igual que HTML5, los Web Components forman parte de las posibilidades que ofrecen los navegadores de manera predeterminada, sin necesidad de librerías o frameworks.

**Nota:** Actualmente todos los navegadores dan soporte a Web Components V1. Desde Chrome a Safari, pasando por Firefox y Edge. Por ello, podemos usarlos sin ninguna restricción. Para navegadores antiguos que no se actualizan ya, como Internet Explorer existe un Polyfill que permite ampliar el soporte a Web Components. Sin embargo, por suerte la cuota de uso de estos navegadores es prácticamente despreciable, por lo que no es necesario ya usar los polyfill en 2022. Para quien no lo sepaa, los **polyfill** son literalmente "**rellenadores de huecos**", que permiten suplir las carencias de navegadores antiguos con respecto a los estándares. En las APIs HTML5 ya se comenzaron a usar intensivamente para permitir usar características nuevas de los lenguajes de la web en clientes antiguos o poco actualizados.

## Javascript como primer lenguaje

Si estás aprendiendo programación, Javascript es una apuesta excelente para comenzar. Básicamente por tres motivos principales:

- **Facilidad de uso:**

No necesitamos más que un navegador para poder ejecutar Javascript. No requiere ningún tipo de compilación (proceso para crear un archivo ejecutable binario para un sistema operativo en particular) sino que es interpretado. Esto implica un flujo de trabajo más simplificado, lo que facilita los primeros pasos. Además Javascript es de tipado dinámico y su sintaxis es menos rebuscada que la de otros lenguajes y permite realizar las cosas de distintos modos, según la habilidad, preferencias o costumbres de cada programador.

- **Amplias áreas de aplicación:**

Como hemos visto, Javascript dispone de usos prácticamente ilimitados. Significa que puedes usar el lenguaje prácticamente para lo que necesites. Aprendiendo un único lenguaje serás capaz de llegar a cualquier propósito que te propongas.

- **Es un lenguaje abierto y estándar para web:**

En algún momento de su actividad profesional la mayoría de los desarrolladores trabajará el entorno web. Aprender programación con Javascript nos asegura que el conocimiento va a



ser aplicado de manera directa en varios momentos de la carrera procesional de los estudiantes.

Estos son los motivos por los que nosotros venimos enseñando a programar con Javascript desde hace más de 15 años, tanto en DesarrolloWeb.com como en EscuelaIT.

Actualmente universidades del prestigio de Stanford han abandonado su curso de introducción a la programación con Java para introducir a sus estudiantes al mundo de la programación con Javascript.

## Múltiples profesiones comienzan con Javascript

Otro de los motivos por los que merece la pena comenzar con Javascript es porque el lenguaje es uno de los pilares fundamentales para desempeñar múltiples profesiones demandadas dentro del ámbito de la web.

- **Frontend developer:**

Es el profesional que se ocupa del desarrollo del lado del cliente, aunque puede tener diversas actividades además de la programación, como diseño, maquetación, desarrollo de interfaces de usuario, aplicaciones SPA del lado del cliente, etc. El conocimiento Javascript es fundamental en todas las áreas de actuación de un frontend developer.

- **Backend developer:**

Como backend entendemos la programación del lado del servidor. Para backend lo cierto es que Javascript (con NodeJS) es solo una de las muchas posibilidades. Sin embargo, NodeJS ofrece muchas ventajas como su asincronía, rapidez y optimización, lo que lo hace idóneo para muchos tipos de proyectos.

- **Fullstack developer:**

Fullstack developer es aquel que es capaz de trabajar tanto del lado del cliente como del lado del servidor. En realidad es una raza rara, tanto que se dice muchas veces que no existe realmente, porque requiere muchos conocimientos del desarrollador y por tanto es muy complicado, o imposible, que un único perfil los reúna todos. Sin embargo, en mi opinión es una realidad en muchos trabajos ya que hay profesionales que deben tener un conocimiento global en varias áreas (freelances autónomos son el principal ejemplo) y por tanto es una figura que realmente sí existe. Hoy, el hecho de Javascript servir tanto para la parte del cliente como para la del servidor, hace mucho más sencilla la figura del fullstack developer y facilita la vida a miles de desarrolladores, ya que les evita la fatiga mental de pasar de un lenguaje a otro constantemente.

**Nota:** La especialización también está presente en estas profesiones. Es verdaderamente asombroso ver cómo se desarrollan los perfiles en el mundo de la web y cómo año tras año se van generando nuevas profesiones especializadas. A lo que antes llamábamos frontend, a secas, hoy



podemos subdividirlo en decenas de perfiles o profesiones determinados como "frontend engineer", "frontend web designer", "CSS architect", "mobile frontend developer", "frontend devops"...

## Cómo aprender Javascript

Aprender Javascript a nivel avanzado requiere meses de estudio, pero comenzar es muy sencillo y en poco tiempo serás capaz de hacer cosas asombrosas con poco esfuerzo. [En el manual de Javascript](#) podrás conocer el lenguaje, pero para comenzar queremos darte un par de consejos.

### Aprende Javascript y no un derivado

Nuestro primer consejo si estás empezando es que **aprendas Javascript y no quieras comenzar por una librería o framework**. Al final todos los proyectos en la web usan Javascript y todo lo que puedes hacer con una librería o framework lo puedes hacer también con Javascript, por lo que lo correcto es comenzar dominando el lenguaje, para luego plantearse nuevos objetivos a medio o largo plazo.

En resumen, no aprendas jQuery: aprende Javascript. No aprendas Angular: aprende Javascript, no aprendas React: aprende Javascript... Con una **base sólida de Javascript te resultará mucho más sencillo aprender más adelante cualquier librería o framework** en la que te quieras basar. Asimismo, no tendrás problemas en el futuro si te cambian de framework en un proyecto o cuando encuentres necesidades para las que esa librería no está pensada, o necesites personalizar cualquier detalle de tu aplicación.

Lo que sí debe acompañar al conocimiento y uso de Javascript son tus habilidades en todo aquello que el navegador (la plataforma web) te ofrece de manera predeterminada: **HTML5 y Web Components**. Todo lo que realices basándote en Javascript "Vanilla", HTML5 y Web Components tienes la certeza que se podrá usar en **cualquier proyecto** donde puedas llegar a trabajar, independientemente de la librería o framework que se esté usando en cada caso.

### Conocimiento de la plataforma web

Nuestro segundo consejo es que, si vas a trabajar en este medio, debes tener presente cada una de las particularidades de la plataforma web. Hay mucho conocimiento general que debes adquirir y que te dará una base sobre la que asentar tus habilidades en el mundo del desarrollo para la web.

**Conocimiento del medio:** Debes saber qué es Internet, la Web, el protocolo HTTP, el sistema de nombres de dominio y por supuesto los lenguajes fundamentales para especificar el contenido y la forma: [HTML](#) y [CSS](#).

**Conocimiento de diseño:** Aunque tu perfil pueda ser más de programador, es ideal tener una visión, al menos técnica, de las características del [diseño para la web](#). Experiencia de usuario, usabilidad, diseño gráfico en general o accesibilidad son puntos importantes.



**Conocimiento de programación:** Si vas a dedicarte a la profesión de programador no solo vale con tener un conocimiento básico del código y la programación estructurada, es ideal interesarte por la [programación orientada a objetos](#), el análisis y diseño de software, patrones de diseño, bases de datos, etc.

**Conocimiento de herramientas:** El profesional además deberá conocer un nutrido grupo de herramientas para el trabajo del día a día, desde el terminal de línea de comandos y administración básica de servidores, hasta la automatización de tareas, pasando por las herramientas de control de versiones ([Git](#) de preferencia) y la optimización.

Afortunadamente todos estos lenguajes, tecnologías y herramientas las puedes conocer o aprender de manera gratuita con los manuales de DesarrolloWeb.com, pero si además quieras un aprendizaje guiado y paso a paso, te recomendamos [suscribirte a los cursos de EscuelaIT](#), donde podrás disponer del material necesario para hacerte un gran profesional del medio.

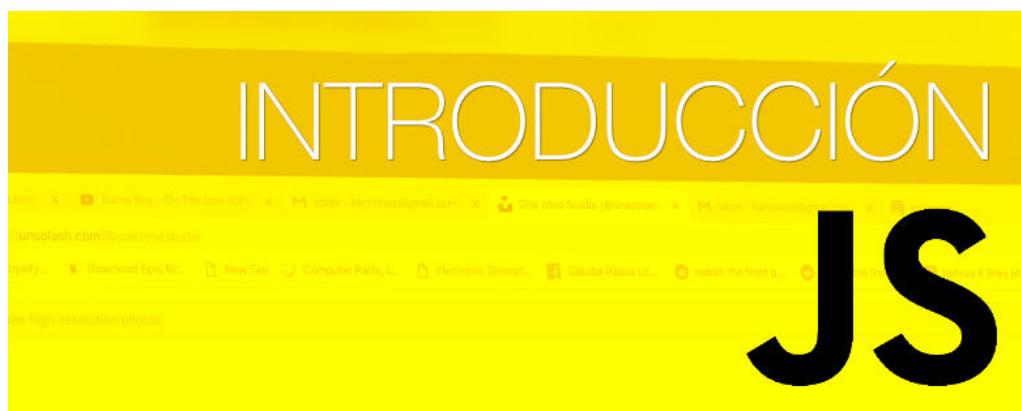
Este artículo es obra de *Miguel Angel Alvarez*

Fue publicado / actualizado en 01/03/2022

Disponible online en <https://desarrolloweb.com/articulos/enfoque-aprender-programacion-javascript.html>

## Introducción a Javascript en el navegador

**Qué es JavaScript qué posibilidades nos ofrece a la hora de desarrollar páginas web del lado del cliente. Comenzamos con algo de conocimiento general del lenguaje y un poco de su evolución.**



En este artículo del [Manual de Javascript](#) queremos abordar las particularidades del uso de Javascript dentro del contexto del navegador, que es básicamente lo que vamos a aprender a lo largo de todo el manual. Será un punto de partida para las personas que quieren introducirse en el mundo de la programación web del lado del cliente.

## Cómo está estructurado el Manual de Javascript

El curso de Javascript que hemos preparado en DesarrolloWeb.com está dividido en dos partes fundamentales que te vamos a resumir para que entiendas cuáles son sus objetivos particulares.

1. En este primer manual pretendemos explicar [el lenguaje Javascript de manera general](#), ofreciendo información sobre cómo incluir scripts y lidiar con los elementos más básicos de cualquier lenguaje de programación, como son las variables, operadores, estructuras de control, funciones, etc.
2. La segunda parte del manual la dedicaremos a explorar temas más específicos sobre [cómo Javascript nos puede ayudar a aplicar dinamismo a una página web](#), a través del control de los elementos de la página y la interacción con el usuario.

**Nota:** En este primer manual encontraréis que se ofrece mucha más información teórica y práctica típica sobre lenguajes de programación. Será esencial para saber cómo dar los primeros pasos en la programación, pero quizás resulte un poco más aburrida que la segunda parte, donde aprenderemos a alterar dinámicamente la página web, responder a acciones del usuario, etc.

Nosotros hemos querido explicar las cosas con detenimiento, para que aprender Javascript con este manual esté al alcance de personas incluso sin conocimientos de programación. No obstante, en DesarrolloWeb.com existen diversos manuales más básicos todavía para aprender a programar, como puede ser el [curso de programación en vídeo](#), o más específicos para la web con el manual de [Páginas dinámicas](#) o la [introducción a la programación](#).

Sin embargo, quizás personas más experimentadas puedan preferir pasar directamente a la [segunda parte del manual de Javascript](#), donde explicaremos cosas más prácticas y volver sobre artículos puntuales de esta primera parte, para utilizarlos como referencia a medida que vayan necesitando conocer la sintaxis de determinadas estructuras de control, operadores del lenguaje, construcción de funciones, etc.

En este artículo pretendemos explicar **qué es Javascript en el contexto del navegador y para qué sirve este lenguaje**, al menos en líneas generales, sin entrar todavía en la parte práctica. En esta primera serie de artículos podrás encontrar información básica sobre el lenguaje, sus principales posibilidades, usos más comunes y los modos de trabajo que podemos emplear para desarrollar nuestros propios scripts.

**Nota:** Otro recurso que queremos recomendar para aprender Javascript, especialmente indicado para las personas con menos experiencia, es el [Videotutorial de Javascript](#). Por supuesto, tampoco nos queremos olvidar de los [Talleres de Javascript](#), así como otros manuales más específicos que encontrarás en la categoría de [Javascript](#).



## Qué es Javascript

Javascript es un lenguaje de programación utilizado para crear comportamientos dinámicos en las páginas web. Con Javascript, al menos en un primer paso, podrás crear pequeños programas encargados de realizar acciones dentro del ámbito de una página web, que generalmente incluye efectos especiales en las páginas e implementar interacción con el usuario. A estos pequeños programas los llamamos scripts, porque a menudo son porciones de código de tamaño limitado, que se encargan de hacer comportamientos muy específicos con los que mejorar la experiencia de usuario al visitar un sitio web.

El navegador del usuario se le conoce como el "cliente web" y es el encargado de interpretar las instrucciones Javascript y ejecutarlas para realizar estos efectos e interactividades, de modo que el mayor recurso, y a menudo el único, con que cuenta este lenguaje es el propio navegador.

Javascript es el siguiente paso, después del [HTML](#) y [CSS](#), que puede dar un programador de la web que decida mejorar sus páginas y la potencia de sus proyectos. Es un lenguaje de programación bastante sencillo y pensado para hacer las cosas con rapidez, a veces con demasiada ligereza. Incluso las personas que no tengan una experiencia previa en la programación podrán aprender este lenguaje con facilidad y utilizarlo en toda su potencia con sólo un poco de práctica.

Entre las acciones típicas que se pueden realizar en Javascript tenemos dos vertientes.

- Por un lado los efectos especiales sobre páginas web, para crear contenidos dinámicos y elementos de la página que tengan movimiento, cambio de color o cualquier otro dinamismo.
- Por el otro, javascript nos permite ejecutar instrucciones como respuesta a las acciones del usuario, con lo que podemos crear páginas interactivas con programas como calculadoras, agendas, o tablas de cálculo.

Javascript es un lenguaje con muchas posibilidades, aunque al principio comenzaremos desarrollando pequeños scripts, también nos permite crear programas más grandes, orientados a objetos, con funciones, estructuras de datos complejas, etc. De hecho, los sitios web más impactantes que te puedas imaginar usan cantidad de Javascript para conseguir que la experiencia de uso sea tan espectacular y personalizada. Hoy, las aplicaciones web modernas ofrecen gracias a Javascript una experiencia de uso casi más parecida a lo que sería una aplicación de escritorio. Prácticamente no hay fronteras a las que no podamos llegar con Javascript.

En este manual vamos a tratar de acercarnos a este lenguaje en profundidad y conocer todos sus secretos y métodos de trabajo. Al final del manual seremos capaces de controlar el flujo en nuestros programas Javascript y saber cómo colocar scripts para resolver distintas necesidades que podamos tener. Todo lo que veremos a continuación nos servirá de base para adentrarnos más adelante en el desarrollo de páginas enriquecidas del lado del cliente.

## Algo de la historia de Javascript

En Internet se han creado multitud de servicios para realizar muchos tipos de comunicaciones, como correo, charlas, transferencias, búsquedas de información, etc. Pero ninguno de estos servicios se ha desarrollado tanto como la Web. Si estamos leyendo estas líneas no vamos a necesitar ninguna explicación de lo que es la web, pero sí podemos hablar un poco sobre cómo se ha ido desarrollando con el paso de los años.

La Web es un sistema Hipertexto, una cantidad de enorme de textos interrelacionados por medio de enlaces. Cada una de las unidades básicas donde podemos encontrar información son las páginas web. En un principio, para diseñar este sistema de páginas con enlaces se pensó en un lenguaje que permitiese presentar cada una de estas informaciones junto con unos pequeños estilos, este lenguaje fue el HTML.

Conforme fue creciendo la Web y sus distintos usos, se fueron complicando las páginas y las acciones que se querían realizar a través de ellas. Al poco tiempo quedó patente que HTML no era suficiente para realizar todas las acciones que se pueden llegar a necesitar en una página web. En otras palabras, HTML se había quedado corto ya que sólo sirve para presentar el texto en un página y poco más.

Al complicarse los sitios web, una de las primeras necesidades fue que las páginas respondiesen a algunas acciones del usuario, para desarrollar pequeñas funcionalidades más allá de los propios enlaces. El primer ayudante para cubrir las necesidades que estaban surgiendo fue Java, que es un lenguaje de propósito general. Java ofrecía una manera de incrustar programas en páginas web, a través de la [tecnología de los Applets](#), con los que se podía crear pequeños programas que se ejecutaban en el navegador dentro de las propias páginas web, pero que tenían posibilidades similares a los programas de propósito general. La programación de Applets fue un gran avance y Netscape, por aquel entonces el navegador más popular, había roto la primera barrera del HTML, al hacer posible la programación dentro de las páginas web. No cabe duda que la aparición de los Applets supuso un gran avance en la historia del web, pero no ha sido una tecnología definitiva y muchas otras han seguido implementando el camino que comenzó con ellos.

## Llega Javascript

Netscape, después de hacer sus navegadores compatibles con los applets, comenzó a desarrollar un lenguaje de programación al que llamó LiveScript que permitiese crear pequeños programas en las páginas y que fuese mucho más sencillo de utilizar que Java. De modo que el primer Javascript se llamo LiveScript, pero no duró mucho ese nombre, pues antes de lanzar la primera versión del producto se forjó una alianza con Sun Microsystems, creador de Java, para desarrollar en conjunto ese nuevo lenguaje.

La alianza hizo que Javascript se diseñara como un hermano pequeño de Java, solamente útil dentro de las páginas web y mucho más fácil de utilizar, de modo que cualquier persona, sin conocimientos de programación pudiese adentrarse en el lenguaje y utilizarlo sin mayores dificultades. Además, para programar Javascript no es necesario un kit de desarrollo, ni compilar los scripts, ni realizarlos en ficheros externos al código HTML, como ocurría con los applets.



Netscape 2.0 fue el primer navegador que entendía Javascript y su estela fue seguida por otros clientes web como Internet Explorer a partir de la versión 3.0. Al motor de ejecución de Javascript dentro de Internet Explorer Microsoft lo bautizó como JScript y tenía ligeras diferencias con respecto a Javascript, lo que hacía que no fuera al 100% compatible con el motor que se ejecutaba en Netscape.

### Diferencias entre distintos navegadores

Como hemos dicho el Javascript de Netscape y el de Microsoft Internet Explorer tenían ligeras diferencias de partida. Incluso, a medida que el propio lenguaje fue evolucionando en las distintas versiones de navegadores y a la par que las páginas web se hacían más dinámicas y exigentes, las diferencias se fueron acentuando.

Esta situación tardó años en corregirse y durante este tiempo **las diferencias de funcionamiento de Javascript entre navegadores marcaron la historia del lenguaje** y el modo en el que los desarrolladores se relacionan con él. Durante mucho tiempo los desarrolladores estábamos obligados a crear código que funcionase correctamente en diferentes plataformas y diferentes versiones de las mismas. A día de hoy, siguen habiendo algunas diferencias, aunque desde la aparición de HTML 5 los fabricantes llegaron a un acuerdo para ser **fieles a los estándares abiertos** y se fueron corrigiendo los problemas de compatibilidad.

Sin embargo, no todos los navegadores se actualizaban por igual y las versiones viejas de Internet Explorer hacían que siguiera siendo necesario un esfuerzo extra para poder crear Javascript compatible con todos los clientes web. Para solucionar todos estos problemas han surgido muchos productos como los [Frameworks Javascript](#), que ayudan a realizar funcionalidades avanzadas de [DHTML](#) sin tener que preocuparse en hacer versiones distintas de los scripts para cada uno de los navegadores posibles del mercado. En este sentido [jQuery](#) fue el claro dominador del mercado.

En la actualidad hemos llegado a un punto en el que, afortunadamente, **Javascript funciona de manera prácticamente idéntica en todos los navegadores**, al menos las partes fundamentales del lenguaje. Sin embargo, el estándar de Javascript sigue evolucionando y por ese motivo la siempre tenemos que llevar en cuenta que no todo navegador puede disponer de todos los avances más nuevos del lenguaje.

A lo largo de este manual **cubrimos principalmente la versión de Javascript más tradicional**, que tiene el nombre de ES5 y está **disponible de manera completa en cualquier cliente web**. Existe una versión de Javascript más moderna, que incluye todo lo tradicional y muchas funcionalidades extra llamada ES6. Esta versión es muy recomendada, porque tiene soporte prácticamente en la totalidad de navegadores, menos en Internet Explorer, y la vemos más a fondo en el [Manual de ES6](#).

**Nota:** Solo a modo de apunte queremos que sepas que, debido a las diferencias entre versiones de Javascript y los navegadores a los que queremos ampliar el soporte, generalmente en el desarrollo frontend (del lado del cliente) se tienen que usar herramientas extra como Babel, que



permiten traducir el código de modo que se adapte a los navegadores objetivo. Babel generalmente se usa a través herramientas como [Webpack](#), que permiten además empaquetar el código Javascript e incluso producir distintas versiones del mismo, que podemos distribuir de manera específica para cada grupo de navegadores.

A continuación seguiremos aprendiendo curiosidades del lenguaje y aclararemos que Java y Javascript son dos cosas distintas, en el artículo sobre [las diferencias de Java y Javascript](#).

Este artículo es obra de *Miguel Angel Alvarez*  
Fue publicado / actualizado en *01/03/2022*  
Disponible online en <https://desarrolloweb.com/articulos/introduccion-javascript.html>

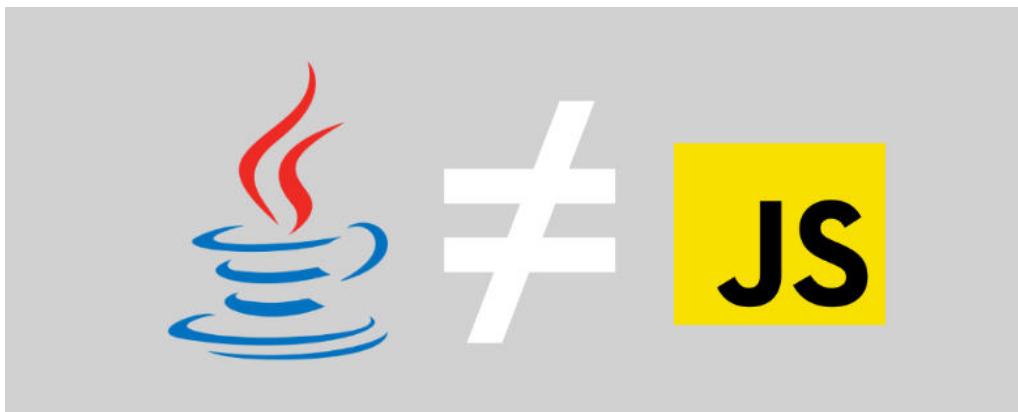
## Diferencias entre Java y Javascript

**Java y Javascript son dos lenguajes de programación distintos. Comparten parte de su nombre y la base de su sintaxis, sin embargo en realidad sus diferencias no notables.**

Estamos contando diversos asuntos interesantes y curiosidades que sirven de introducción para el [Manual de Javascript](#) y queremos tratar una de las más típicas asociaciones que se hacen al oír hablar de Javascript. Nos referimos a relacionarlo con otro lenguaje de programación, llamado Java, que no tiene mucho que ver.

Realmente Javascript se llamó así porque Netscape (el navegador que lanzó Javascript), que estaba aliado a los creadores de Java en la época (Sun Microsystems), quiso aprovechar la imagen de marca y la popularidad de Java, como maniobra para la expansión de su nuevo lenguaje. Con todo, se creó un producto que tenía ciertas similitudes, como la sintaxis del lenguaje o el nombre. Se hizo entender que era un hermano pequeño y orientado específicamente para hacer cosas en las páginas web, pero también se hizo caer a muchas personas en el error de pensar que son lo mismo.

Queremos que quede claro que **Javascript no tiene nada que ver con Java**, salvo el acuerdo de colaboración de los creadores de ambos lenguajes, como se ha podido leer. Siempre han sido productos totalmente distintos que no guardan entre sí más relación que la sintaxis idéntica, aunque a decir verdad, la sintaxis de ambos está heredada de otro popular lenguaje llamado C.



## Diferencias más notables entre Java y Javascript

Algunas de las diferencias más representativas estos dos lenguajes son las siguientes:

- **Compilador.** Para programar en Java necesitamos un Kit de desarrollo y un compilador. Sin embargo, Javascript no es un lenguaje que necesite que sus programas se compilen, sino que éstos se interpretan por parte del navegador cuando éste lee la página.
- **Orientado a objetos.** Java es un lenguaje de programación orientado a objetos. (Más tarde veremos que quiere decir orientado a objetos, para el que no lo sepa todavía). Javascript es un lenguaje "multiparadigma" no requiere programar orientado a objetos, aunque sí lo permite. Esto quiere decir que podremos programar en Javascript sin necesidad de crear clases, tal como se realiza en los lenguajes de programación estructurada como C o Pascal.
- **Propósito.** Java en principio es mucho más potente que Javascript, debido a que es un lenguaje de propósito general. Con Java se pueden hacer aplicaciones de lo más variado, sin embargo, con Javascript sólo podemos escribir programas para que se ejecuten en páginas web. Aunque a decir verdad, desde la última década Javascript se ha expandido tanto que hoy podemos usar el lenguaje para hacer aplicaciones de todo tipo, como programas de consola, aplicaciones para móviles, de escritorio, etc. Por tanto, a día de hoy es Javascript podríamos decir que es casi tan potente, o más, de lo que es Java.
- **Tipado estático.** Java es un lenguaje de programación fuertemente tipado (también llamado de tipado estático). Esto quiere decir que al declarar una variable en Java tendremos que indicar su tipo y no podrá cambiar de un tipo a otro a lo largo de la ejecución del programa. Por su parte Javascript no tiene esta característica, sino que es un lenguaje de tipado dinámico (o levemente tipado) y podemos meter en una variable la información que deseemos, independientemente del tipo de ésta. Además, podremos cambiar el tipo de dato de una variable cuando queramos.
- **Lenguaje abierto / lenguaje propietario .** Otra diferencia importante es que Javascript está basado en un estándar abierto, que no tiene un dueño en particular y por tanto cualquier fabricante puede implementarlo en sus sistemas libremente. Sin embargo Java es un lenguaje propiedad de una empresa (actualmente Oracle), por lo que está dirigido con un enfoque particular y comercial.- **Otras características.** Como vemos, Java es mucho más complejo, aunque también más potente y robusto. De entrada Java tiene más funcionalidades que Javascript y requiere de un aprendizaje mucho más intenso para poder dominarlo. Javascript



permite aprender con facilidad, incluso para personas sin experiencia en la programación, y permite hacer programas rápidamente, obteniendo resultados bastante atractivos con poco código y esfuerzo.

Las diferencias que separan a Java de Javascript por tanto son notables. También se usan para cosas muy diferentes. Java está más pensado para hacer aplicaciones complejas, orientadas al mundo empresarial, o aplicaciones para teléfonos Android. Javascript está pensado para hacer aplicaciones menos pesadas, principalmente orientadas a la web.

Queremos remarcar que actualmente el enfoque web de Javascript no tiene por qué considerarse único. Como hemos dicho, es posible hacer casi cualquier tipo de programa mediante Javascript y en los últimos años ha ido ocupando más y más parcelas. También, a medida que se han ido presentando nuevas versiones del estándar de Javascript, se ha ido haciendo más robusto y adecuado también para aplicaciones grandes.

En definitiva, hay tantas cosas que separan a Java y Javascript que podríamos decir que comparten poco más que su nombre y una base común en su sintaxis, heredada de C. Sus diferencias son lo suficientemente importantes como para distinguirlos fácilmente.

Este artículo es obra de *Miguel Angel Alvarez*  
Fue publicado / actualizado en 01/02/2020  
Disponible online en <https://desarrolloweb.com/articulos/492.php>

## Antes de empezar

**Mostramos ejemplos de páginas que emplean JavaScript en su desarrollo y comentamos las aplicaciones necesarias para empezar a programar.**

Hay varios puntos que queremos comentar como introducción en el [Manual de Javascript](#) y que podrás querer conocer antes de comenzar a programar. Primero sería bueno hacernos una idea más concreta de las posibles aplicaciones que podría tener el lenguaje y que se pueden encontrar en innumerables sitios web. Además también queremos comentar las herramientas y conocimientos previos que necesitamos para ponernos manos a la obra.

## Usos de Javascript

Quizás a día de hoy sobra decir para qué sirve Javascript, pero veamos brevemente algunos usos de este lenguaje que podemos encontrar en el web para hacernos una idea de las posibilidades que tiene.



Sin ir más lejos, DesarrolloWeb.com utiliza Javascript para el menú superior, que muestra diferentes enlaces dentro de cada opción principal. Vamos cambiando la página cada cierto tiempo, pero en el diseño actual de este sitio web, elementos como el recuadro de "Login" también tienen su dinamismo con Javascript.

Actualmente casi todas las páginas un poco avanzadas utilizan Javascript, pues se ha vuelto una de las insignias de lo que se denomina la Web 2.0 y la experiencia enriquecida de usuario. Por ejemplo, webs tan populares como Facebook, Twitter o Youtube usan Javascript a raudales. Para ser más concretos, cuando en la red social apretamos un enlace para comentar algo, se muestra en la página un pequeño formulario que aparece como por arte de magia y luego se envía sin salirse de la propia página. También cuando votamos por un vídeo en Youtube o cuando se cuentan los caracteres que llevamos escritos en los mini-post de Twitter, se utiliza Javascript para realizar pequeñas funcionalidades que no es posible realizar con HTML sólo. En realidad se pueden ver ejemplos de Javascript dentro de cualquier página un poco compleja. Algunos que habremos visto en innumerables ocasiones son calendarios dinámicos para seleccionar fechas, calculadoras o convertidores de divisas, editores de texto enriquecido, navegadores dinámicos, etc.

Es mucho más habitual encontrar Javascript para realizar efectos simples sobre páginas web, o no tan simples, como pueden ser navegadores dinámicos, apertura de ventanas secundarias, validación de formularios, etc. Nos atrevemos a decir que este lenguaje es realmente útil para estos casos, pues estos típicos efectos tienen la complejidad justa para ser implementados en cuestión de minutos sin posibilidad de errores. Sin embargo, aparte de esos unos simples ejemplos, podemos encontrar dentro de Internet muchas aplicaciones que basan parte de su funcionamiento en Javascript, que hacen que una página web se convierta en un verdadero programa interactivo de gestión de cualquier recurso. Ejemplos claros son las aplicaciones de ofimática online, como Google Docs, Office Online o Google Calendar.

## Qué necesitas para trabajar con Javascript

Para programar en Javascript necesitamos básicamente lo mismo que para desarrollar páginas web con HTML. Un editor de textos y un navegador compatible con Javascript. Cualquier ordenador mínimamente actual posee de salida todo lo necesario para poder programar en Javascript. Por ejemplo, un usuario de Windows dispone dentro de su instalación típica de sistema operativo, de un editor de textos, el Bloc de notas, y de un navegador: Internet Explorer.

**Nota:** Usuarios que deseen herramientas más avanzadas pueden encontrar en Internet fácilmente programas similares en la sección de [programas para desarrolladores](#).

Permitidme una recomendación con respecto al editor de textos. Se trata de que, aunque el Bloc de Notas es suficiente para empezar, tal vez sea muy útil contar con otros programas que nos ofrecen mejores prestaciones a la hora de escribir las líneas de código. Estos editores avanzados tienen algunas ventajas como que colorean los códigos de nuestros scripts, nos permiten trabajar con



varios documentos simultáneamente, tienen ayudas, etc. Entre otros queremos destacar Visual Studio Code o Atom. Puedes ver varias [alternativas de programas de edición de código](#).

## Conocimientos previos recomendables

Lo cierto es que no hace falta tener mucha base de conocimientos para ponerse a programar en Javascript. Lo más seguro es que si lees estas líneas ya sepas todo lo necesario para trabajar, puesto que ya habrás tenido alguna relación con el desarrollo de sitios web y habrás detectado que para hacer ciertas cosas te viene bien conocer un poco de Javascript.

No obstante, sería bueno tener un dominio avanzado de HTML, al menos el suficiente para escribir código en ese lenguaje sin tener que pensar qué es lo que estás haciendo. También será útil un conocimiento medio sobre CSS y quizás alguna experiencia previa sobre algún lenguaje de programación, aunque en este manual de DesarrolloWeb.com vamos a tratar de explicar Javascript incluso para personas que no hayan programado nunca.

En el siguiente artículo seguiremos con temas que sirven de introducción al lenguaje de scripting del lado del cliente viendo las [algunas diferencias de Javascript que existen en las versiones de navegadores que han ido apareciendo](#).

Este artículo es obra de *Miguel Angel Alvarez*

Fue publicado / actualizado en *16/07/2001*

Disponible online en <https://desarrolloweb.com/articulos/493.php>

## Versiones de navegadores y de Javascript

**Javascript ha ido evolucionando a lo largo del tiempo, surgiendo varias versiones del estándar ECMAScript. Los navegadores pueden ser compatibles solamente con ciertas versiones.**

Para continuar con la introducción a Javascript, también resulta apropiado introducir las distintas versiones de Javascript que existen y que han evolucionado a la par que las versiones de navegadores. El lenguaje ha ido avanzando durante sus años de vida e incrementando sus capacidades.

Este es un conocimiento interesante, dado que cuando desarrollamos con Javascript dependemos directamente de la plataforma de ejecución y la compatibilidad o no con alternativas modernas del lenguaje.

Ya para advertirlo de entrada, en este [Manual de Javascript](#) vamos a trabajar con versiones del lenguaje completamente extendidas en todos los navegadores, por lo que cualquier cosa que expliquemos podrás aplicarla sin problema alguno.



## La evolución de Javascript

En el artículo de la [introducción a Javascript](#) ya explicamos algo de la historia del lenguaje. Vimos que en un principio sus objetivos eran sencillos y que en la actualidad con Javascript podemos llegar a realizar páginas, interfaces de usuario y efectos realmente complejos. Por tanto, a medida que las exigencias de los desarrolladores crecían, también lo tenía que realizar el propio lenguaje.

Es importante mencionar que el lenguaje ha ido evolucionando en dos vertientes:

- Por un lado el propio lenguaje ha ido incorporando operadores, estructuras de control, reglas de sintaxis para hacer cosas repetitivas con menos código. Estas mejoras en el lenguaje las debemos al estándar ECMAScript, del que vamos a hablar en seguida.
- Por otro lado, los navegadores han ido incorporando instrucciones nuevas, para poder manipular elementos modernos de la página, como por ejemplo las divisiones, los estilos CSS y sistemas como almacenamiento local, trabajo a pantalla completa, geolocalización y un largo etc. Todas estas mejoras son definidas bajo los estándares de HTML 5 y sus APIs Javascript.

En cualquier caso, lo que debe quedar claro es que Javascript ha ido mejorando sus características como lenguaje, a la vez que los navegadores han ido ofreciendo un mayor soporte a funcionalidades avanzadas para el control de los elementos de la página. Es por ello que en Javascript tenemos generalmente que tener cuidado con el mercado de los navegadores y la compatibilidad a las funcionalidades que queremos utilizar.

## Estándar ECMAScript

Javascript como lenguaje es estandarizado por la organización "ECMA International". Esta empresa se dedica a la creación de estándares para la comunicación y el tratamiento de información. Uno de sus más conocidos estándares es ECMAScript.

ECMAScript es el estándar de definición del lenguaje Javascript, que cualquier cliente que soporte Javascript debe de soportar. A lo largo del tiempo se han ido publicando distintas versiones de ECMAScript, siendo la última más extendida ECMAScript 2015, también conocido por ES6.



ES6 es soportado por todos los navegadores actuales, excepto Internet Explorer. Por lo tanto, si programás con soporte a IE, no podrías usar en principio ES6.

Esto es una verdad a medias, ya que existen sistemas que permiten traducir el código de una versión a otra del estándar, aunque de momento no queremos meternos en estos detalles. Si te interesa, te recomendamos la lectura del [Manual de Webpack](#), que es una de las muchas herramientas que puedes usar para realizar esa tarea.

## APIs HTML 5

Un [API](#) es una interfaz de programación de aplicaciones. Básicamente indica cómo se va a acceder a las funcionalidades o servicios de un sistema, por medio de qué instrucciones y qué procedimientos.

Los navegadores definen un API para acceder a los recursos con los que contamos para manipular el estado de la página y acceder a periféricos, almacenamiento, etc.

Los APIs del navegador son generalmente especificados por la W3C y todos los clientes web los tienen que implementar de manera fiel.

Ocurre un poco lo mismo que con las versiones del lenguaje. Si el navegador es muy viejo puede no tener soporte a algunos API, por lo que tenemos que ser cautelosos. En la práctica más que nada con Internet Explorer y algún navegador para móviles antiguos.

En esta primera parte del manual no vamos a tratar las API HTML 5 ni el acceso a los recursos del navegador. Ese área es materia de estudio de la segunda parte, dedicada al [Desarrollo en Javascript del lado del cliente](#).

## Versiones del lenguaje

En la primera parte de este manual el objetivo es aprender bien Javascript, por lo que nos limitaremos al estudio del lenguaje en particular. En este sentido usaremos ES5, que es la versión del lenguaje que funciona en todos los navegadores actuales.

Como información general, vamos a resumir vamos a comentar las distintas versiones de Javascript:

Realmente cualquier navegador medianamente moderno tendrá ahora todas las funcionalidades de Javascript que vayamos a necesitar y sobre todo, las que podamos utilizar en nuestros primeros pasos con el lenguaje. No obstante puede venir bien conocer las primeras versiones de Javascript que comentamos en este artículo, a modo de curiosidad.



- **Javascript 1:** nació con el Netscape 2.0 y soportaba gran cantidad de instrucciones y funciones, casi todas las que existen ahora ya se introdujeron en el primer estandar.
- **Javascript 1.1:** Es la versión de Javascript que se diseñó con la llegada de los navegadores 3.0. Implementaba poco más que su anterior versión, como por ejemplo el tratamiento de imágenes dinámicamente y la creación de arrays.
- **Javascript 1.2:** La versión de los navegadores 4.0. Esta tiene como desventaja que es un poco distinta en plataformas Microsoft y Netscape, ya que ambos navegadores crecieron de distinto modo y estaban en plena lucha por el mercado.
- **Javascript 1.3:** Versión que implementan los navegadores 5.0. En esta versión se han limado algunas diferencias y asperezas entre los dos navegadores.
- **Javascript 1.5:** Versión actual, en el momento de escribir estas líneas, que implementa Netscape 6.
- Por su parte, **Microsoft** también ha evolucionado hasta presentar su **versión 5.5 de JScript** (así llaman al javascript utilizado por los navegadores de Microsoft).
- En 2009 se publica la versión de **ECMAScript 5**. Esta es la versión que vamos a tratar principalmente en el manual, que funciona en todos los navegadores del mercado.
- En 2015 se lanza la **sexta versión de ECMAScript**, con una cantidad muy importante de novedades. Esta versión se puede usar completamente, excepto en Internet Explorer. Sus mejoras son tan importantes que todos los desarrolladores profesionales las usan y, si fuera necesario, se traduce el código para soporte a navegadores antiguos. Esta versión se explica en el [Manual de ES6](#).
- Actualmente se ha adquirido un compromiso de liberar una versión del estándar ECMAScript por año. Por ello existen diversos estándares con pequeños cambios incrementales, que se han ido incorporando a distintos ritmos entre los navegadores.

## Conclusión a las versiones de Javascript y compatibilidad

Es obvio que todavía, después de escribir estas líneas, se presentarán o habrán presentado muchas otras versiones de Javascript, pues, a medida que se van mejorando los navegadores y van saliendo versiones de HTML, surgen nuevas necesidades para programación de elementos dinámicos. No obstante, todo lo que vamos a aprender en este manual, incluso otros usos mucho más avanzados, ya está implementado en cualquier Javascript que existan en la actualidad.

En el siguiente artículo comenzaremos ya a mostrar [pequeños códigos Javascript que servirán para hacer efectos simples en páginas web](#).

Este artículo es obra de *Miguel Angel Alvarez*

Fue publicado / actualizado en *10/02/2020*

Disponible online en <https://desarrolloweb.com/articulos/494.php>

## Efectos rápidos con Javascript

**En este último artículo de introducción a Javascript veremos algunos ejemplos de códigos sencillos de gran utilidad.**

Antes de meternos de lleno en materia podemos ver una serie de efectos rápidos que se pueden programar con Javascript, lo que nos puede hacer una idea más clara de las capacidades y potencia del lenguaje. A continuación veremos varios ejemplos, que hemos destacado para esta introducción en el [Manual de Javascript](#), por tener un mínimo de complejidad y aunque sean muy básicos, nos vendrán bien para tener una idea más exacta de lo que es Javascript a la hora de recorrer los siguientes capítulos.

### Abrir una ventana secundaria

Primero vamos a ver que con una línea de Javascript podemos hacer cosas bastante atractivas. Por ejemplo podemos ver cómo abrir una ventana secundaria sin barras de menús que muestre el buscador Google. El código sería el siguiente.

```
<script>
window.open("http://www.google.com","", "width=550,height=420,menubar=no")
</script>
```

Podemos [ver el ejemplo en marcha aquí](#).

### Un mensaje de bienvenida

Podemos mostrar una caja de texto emergente al terminarse de cargar la portada de nuestro sitio web, que podría dar la bienvenida a los visitantes.

```
<script>
window.alert("Bienvenido a mi sitio web. Gracias...")
</script>
```

Puedes [ver el ejemplo en una página a parte](#).

### Fecha actual

Veamos ahora un sencillo script para mostrar la fecha de hoy. A veces es muy interesante mostrarla en las webs para dar un efecto de que la página está al "al día", es decir, está actualizada.

```
<script> document.write(new Date()) </script>
```

Estas líneas deberían introducirse dentro del cuerpo de la página en el lugar donde queramos que aparezca la fecha de última actualización. Podemos [ver el ejemplo en marcha aquí](#).



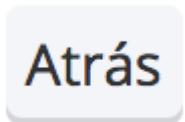
**Nota:** Un detalle a destacar es que la fecha aparece en un formato un poco raro, indicando también la hora y otros atributos de la misma, pero ya aprenderemos a obtener exactamente lo que deseemos en el formato correcto.

## Botón de volver

Otro ejemplo rápido se puede ver a continuación. Se trata de un botón para volver hacia atrás, como el que tenemos en la barra de herramientas del navegador. Ahora veremos una línea de código que mezcla HTML y Javascript para crear este botón que muestra la página anterior en el historial, si es que la hubiera.

```
<input type=button value=Atrás onclick="history.go(-1)">
```

El botón sería parecido al siguiente, un botón normal con el aspecto predeterminado que el navegador y sistema operativo que usas otorgue a los botones. A continuación tienes una imagen sobre cómo se vería el botón en mi sistema.



Atrás

Como diferencia con los ejemplos anteriores, hay que destacar que en este caso la instrucción Javascript se encuentra dentro de un atributo de HTML, onclick, que indica que esa instrucción se tiene que ejecutar como respuesta a la pulsación del botón.

Se ha podido comprobar la facilidad con la que se pueden realizar algunas acciones interesantes. Como podréis imaginar, existirían muchas otras muestras sencillas de Javascript que nos reservamos para capítulos posteriores.

Si lo deseas, puedes ver cómo hemos desarrollado algunos de estos [efectos rápidos Javascript paso por paso y en vídeo](#). En el siguiente artículo empezaremos ya a hablar del propio [lenguaje de programación Javascript](#).

Este artículo es obra de *Miguel Angel Alvarez*  
Fue publicado / actualizado en *16/07/2001*  
Disponible online en <https://desarrolloweb.com/articulos/495.php>

# Primeros pasos con el lenguaje Javascript

Comenzamos a aprender cosas que tienen que ver directamente con la programación en el lenguaje Javascript y la manera con la que se integra éste en una página web.

## El lenguaje Javascript

**Veamos cómo insertar el código en Javascript dentro de un documento HTML. En este artículo te explicaremos algunas reglas importantes y las maneras más básicas de ejecutar Javascript en el contexto de una página.**

En esta parte del [manual sobre Javascript](#) vamos a conocer la manera más básica de trabajar con el lenguaje. En este artículo daremos las primeras informaciones sobre cómo incluir scripts, mezclando el propio código Javascript con el HTML.

Luego veremos también cómo se debe [colocar código para que nuestra web sea compatible con todos los navegadores, incluso aquellos que no soportan Javascript](#). Muchas ideas del funcionamiento de Javascript ya se han descrito en capítulos anteriores, pero con el objetivo de no dejarnos nada en el tintero vamos a tratar de acaparar a partir de aquí todos los datos importantes de este lenguaje.



### Javascript se escribe en el documento HTML

Lo más importante y básico que podemos destacar en este momento es que la programación de Javascript se realiza dentro del propio documento HTML. Es decir, el código Javascript, en la mayoría de los casos, se mezcla con el propio código HTML para generar la página.

Esto quiere decir que debemos aprender a mezclar los dos lenguajes de programación y rápidamente veremos que, para que estos dos lenguajes puedan convivir sin problemas entre ellos, se han de incluir unos delimitadores que separan las etiquetas HTML de las instrucciones



Javascript. Estos delimitadores son las etiquetas `<SCRIPT>` y `</SCRIPT>`. Todo el código Javascript que pongamos en la página ha de ser introducido entre estas dos etiquetas.

## La colocación de los scripts sí que importa

En una misma página podemos introducir varios scripts, cada uno que podría introducirse dentro de unas etiquetas `<SCRIPT>` distintas. La colocación de estos scripts no es indiferente. En un principio, con lo que sabemos hasta el momento y los [scripts que hemos realizado de prueba](#), nos da un poco igual donde colocarlos, pero en determinados casos esta colocación sí que será muy importante. En cada caso, y llegado el momento, se informará de ello convenientemente.

También se puede escribir Javascript dentro de determinados atributos de la página, como el atributo `onclick`. Estos atributos están relacionados con las acciones del usuario y se llaman manejadores de eventos.

A continuación vamos a ver más detenidamente estas dos maneras de escribir scripts, que tienen como diferencia principal el momento en que se ejecutan las sentencias.

## Maneras de escribir scripts Javascript

Hasta ahora en el [Manual de Javascript](#) ya hemos tenido la ocasión de probar algunos scripts sencillos, no obstante, todavía tenemos que aprender una de las bases para poder trabajar con el lenguaje y es aprender las dos maneras de ejecutar código Javascript. Existen **dos maneras fundamentales de ejecutar scripts** en la página. La primera de estas maneras se trata de ejecución directa de scripts, la segunda es una ejecución como respuesta a la acción de un usuario.

Explicaremos ahora cada una de estas formas de ejecución disponibles, pero para el que lo deseé, recomendamos también ver el [vídeo sobre Maneras de incluir y ejecutar scripts](#).

## Ejecución directa del código Javascript

Es el método de ejecutar scripts más básico. En este caso se incluyen las instrucciones dentro de la etiqueta `<SCRIPT>`, tal como hemos comentado anteriormente. Cuando el navegador lee la página y encuentra un script va interpretando las líneas de código y las va ejecutando una después de otra. Llamamos a esta manera ejecución directa pues cuando se lee la página se ejecutan directamente los scripts.

```
<!DOCTYPE html>
<html lang="es">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <meta http-equiv="X-UA-Compatible" content="ie=edge">
  <title>Ejemplo ejecución directa</title>
</head>
<body>
  <h1>Página con Javascript</h1>
  <p>Esta página tiene un cuadro de diálogo, que se mostrará nada más el navegador la procese.</p>

  <script>
    var personas = 4;
```



```
var importeEntradas = 9.50;
alert('Necesitas ' + personas * importeEntradas + ' euros para que entren todos al cine');
</script>

<p>Cuando el usuario pulse aceptar en el cuadro de diálogo, el navegador mostrará la página completa.</p>
</body>
</html>
```

Ese código Javascript se ejecutará según se abra la página. Cuando el navegador al procesar la página se encuentre este código, parará la lectura de la página para ejecutar el script de Javascript. Como resultado mostrará en una caja de diálogo "Necesitas 38 euros para que entren todos al cine". Cuando el usuario pulse el botón "aceptar", continuará leyendo el resto de la página y mostrando el contenido de la página completa en la ventana del navegador. Por supuesto veremos muchos otros ejemplos a lo largo del manual.

Este método será el que utilicemos preferentemente en la mayoría de los ejemplos de [esta parte del Manual de Javascript](#). En la [segunda parte del Manual de Javascript](#) podremos aprender muchas cosas y entre ellas veremos con detalle el segundo modo de ejecución de scripts que vamos a relatar a continuación.

## Ejecución de Javascript como respuesta a un evento

Es la otra manera de ejecutar scripts, pero antes de verla debemos hablar sobre los eventos. Los eventos son acciones que realiza el usuario. Los programas como Javascript están preparados para atrapar determinadas acciones realizadas, en este caso sobre la página, y realizar acciones como respuesta. De este modo se pueden realizar programas interactivos, ya que controlamos los movimientos del usuario y respondemos a ellos. Existen muchos tipos de eventos distintos, por ejemplo la pulsación de un botón, el movimiento del ratón o la selección de texto de la página.

Las acciones que queremos realizar como respuesta a un evento se pueden indicar de muchas maneras distintas, por ejemplo dentro del mismo código HTML, en atributos que se colocan dentro de la etiqueta que queremos que responda a las acciones del usuario. En el [capítulo donde vimos algún ejemplo rápido](#) ya comprobamos que si queríamos que un botón realizase acciones cuando se pulsase sobre él, debíamos indicarlas dentro del atributo onclick del botón.

Comprobamos pues que se puede introducir código Javascript dentro de determinados atributos de las etiquetas HTML. Sin embargo, no es el único método posible. También podemos seleccionar elementos de la página directamente con Javascript y asociar funciones que se ejecutarán como respuesta a eventos. Aunque es un poco pronto para que puedas entender todo este proceso con detalle, vamos a ver un ejemplo sencillo.

```
<!DOCTYPE html>
<html lang="es">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <meta http-equiv="X-UA-Compatible" content="ie=edge">
  <title>Ejemplo pasar ratón por encima</title>
</head>
<body>
  <h1>Ejemplo Javascript</h1>
```



```
<span id="mieldimento">Pasa el ratón por aquí</span>
<script>
var pasadas = 0;
function anunciarPasadas() {
    pasadas = pasadas + 1;
    alert('Has pasado el ratón encima ' + pasadas + ' veces');
}
document.getElementById('mieldimento').addEventListener('mouseenter', anunciarPasadas);
</script>
</body>
</html>
```

En esta ocasión tienes el código de una página entera. En el puedes encontrar el código Javascript embutido dentro del cuerpo de la página. A diferencia del código anterior, cuando el navegador lee la página no ejecuta nada. Simplemente memoriza el script y asocia la función al elemento que se ha definido. En concreto hemos definido un evento de tipo "mouseenter" sobre un elemento de la página que tiene identificador "mieldimento". Esto quiere decir que, cada vez que el usuario coloque el puntero del ratón encima del elemento "mieldimento" ejecutará la función "anunciarPasadas".

Como decimos, quizás es un poco temprano para ver ejemplos de este estilo, por lo que no te preocupes si no lo has entendido todo. Estamos solamente comenzando el manual de Javascript y podrás aprender todo esto y mucho más poco a poco. Así pues, Veremos más adelante este tipo de ejecución en profundidad y los tipos de eventos que existen. Para llegar a ello aun tenemos que aprender muchas otras cosas de Javascript. En el próximo artículo mostraremos cómo podemos [ocultar el código Javascript para navegadores antiguos](#).

Este artículo es obra de *Miguel Angel Alvarez*

Fue publicado / actualizado en 04/12/2019

Disponible online en <https://desarrolloweb.com/articulos/501.php>

## Ocultar scripts Javascript en navegadores antiguos

**No todos los navegadores son compatibles con Javascript, así que tenemos que aprender cómo hacer que los scripts no molesten en navegadores que no los entienden.**

A lo largo de los anteriores capítulos del [Manual de Javascript](#) ya hemos visto que el lenguaje se implementó a partir de Netscape 2.0 e Internet Explorer 3.0. Incluso, para los que no lo sepan, está bien decir que hay navegadores que funcionan en sistemas operativos, donde sólo se puede visualizar texto y donde determinadas tecnologías y aplicaciones no están disponibles, como el uso de imágenes, fuentes tipográficas distintas o el propio Javascript.

Así pues, no todos los navegadores que puedan utilizar los usuarios que visiten nuestra página comprenden Javascript. En los casos en los que no se interpretan los scripts, los navegadores asumen que el código de éstos es texto de la propia página y como consecuencia, presentan los



scripts en el cuerpo del documento, como si de texto normal se tratara. Para evitar que el texto de los scripts se escriba en la página cuando los navegadores no los entienden se tienen que ocultar los con comentarios HTML (<!-comentario HTML -->). Además, en este artículo veremos también cómo mostrar un mensaje que se vea sólo en los navegadores que no son compatibles con Javascript.

**Actualizado:** En el momento actual podemos decir que casi el 100% de los navegadores disponibles soportan Javascript, o por lo menos reconocen las etiquetas de script, por lo que, aunque esté desactivado, no mostrarán el texto de nuestros programas Javascript. Por ello, actualmente ya no es fundamental realizar la operación de ocultar el código de los scripts de la página para navegadores antiguos. No obstante, si queremos hacer una página totalmente correcta, convendrá aprender cómo se puede ocultar un script para que en ningún caso se muestre como texto en la página.

## Ocultar el código Javascript con comentarios HTML

Veamos con un ejemplo de código donde se han utilizado comentarios HTML para ocultar Javascript, o mejor dicho, el código los scripts Javascript.

```
<SCRIPT>
<!--
Código Javascript
//-->
</SCRIPT>
```

Vemos que el inicio del comentario HTML es idéntico a cómo lo conocemos en el HTML, pero el cierre del comentario presenta una particularidad, que empieza por doble barra inclinada. Esto es debido a que el final del comentario contiene varios caracteres que Javascript reconoce como operadores y al tratar de analizarlos lanza un mensaje de error de sintaxis. Para que Javascript no lance un mensaje de error se coloca antes del comentario HTML esa doble barra, que no es más que un comentario Javascript, que conoceremos más adelante cuando hablemos de sintaxis.

El inicio del comentario HTML no es necesario comentarlo con la doble barra, dado que Javascript entiende bien que simplemente se pretende ocultar el código. Una aclaración a este punto: si pusiésemos las dos barras en esta línea, se verían en navegadores antiguos por estar fuera de los comentarios HTML. Las etiquetas `<SCRIPT>` no las entienden los navegadores antiguos, por lo tanto no las interpretan, tal como hacen con cualquier etiqueta que desconocen.

## Mostrar un mensaje para navegadores antiguos con `<NOSCRIPT>`

Existe la posibilidad de indicar un texto alternativo para los navegadores que no entienden Javascript, para informarles de que en ese lugar debería ejecutarse un script y que la página no está funcionando al 100% de sus capacidades. También podemos sugerir a los visitantes que actualicen



su navegador a una versión compatible con el lenguaje. Para ello utilizamos la etiqueta <NOSCRIPT> y entre esta etiqueta y su correspondiente de cierre podemos colocar el texto alternativo al script.

```
<SCRIPT>
código javascript
</SCRIPT>
<NOSCRIPT>
Este navegador no comprende los scripts que se están ejecutando, debes actualizar tu versión de navegador a una más
reciente.
<br><br>
<a href="http://netscape.com">Netscape</a>.<br>
<a href="http://microsoft.com">Microsoft</a>.
</NOSCRIPT>
```

En el siguiente artículo veremos algunos [otros detalles sobre colocar scripts Javascript](#) que se han quedado en el tintero.

Este artículo es obra de *Miguel Angel Alvarez*

Fue publicado / actualizado en *29/07/2001*

Disponible online en <https://desarrolloweb.com/articulos/503.php>

## Más sobre colocar scripts

**Últimas notas sobre cómo colocar scripts Javascript en una página web. Nos centramos en distintos atributos que puedes usar en la etiqueta script y en la posibilidad de usar código que viene de archivos externos al propio documento HTML.**

Seguimos el [Manual de Javascript](#), abordando temas de información relevante para saber cómo utilizar el lenguaje en el contexto de una página web. Así que ahora veremos un par de notas adicionales sobre cómo colocar scripts en páginas web.

En este artículo mostraremos uno de los atributos que se pueden indicar en la etiqueta de SCRIPT, que indica el lenguaje que vamos a utilizar. Además, mostraremos otro modo muy útil de asociar código Javascript en la página, por medio de un fichero externo. Este punto es algo fundamental y tendremos que prestar especial atención, ya que es sin duda la manera de trabajar que se utiliza más a menudo.



## Indicar el lenguaje que estamos utilizando

**Actualizado:** Hoy ya no es necesario especificar el lenguaje de script que estamos utilizando. El motivo es que Javascript es el único lenguaje para crear scripts en páginas web aceptado por la industria. Por tanto, el atributo "language" es realmente innecesario.

La etiqueta <SCRIPT> tiene un atributo que sirve para indicar el lenguaje que estamos utilizando, así como la versión de este. Por ejemplo, podemos indicar que estamos programando en Javascript 1.2 o Visual Basic Script, que es otro lenguaje para programar scripts en el navegador cliente que sólo es compatible con Internet Explorer.

El atributo en cuestión es "language" y lo más habitual es indicar simplemente el lenguaje con el que se han programado los scripts. El lenguaje por defecto es Javascript, por lo que si no utilizamos este atributo, el navegador entenderá que el lenguaje con el que se está programando es Javascript. Un detalle donde se suele equivocar la gente sin darse cuenta es que language se escribe con dos -g- y no con -g- y con -j- como en castellano.

```
<script language="javascript">
```

### Uso del atributo "type":

Cuando colocamos una etiqueta SCRIPT debemos usar el atributo "type" para indicar que tipo de codificación de script estamos haciendo y el lenguaje utilizado.

```
<script type="text/javascript">
```

El atributo "type" es necesario para que valide correctamente tu documento en las versiones más actuales del HTML.

**Nota sobre las versiones de HTML:** Con la llegada de HTML5 se retiró la necesidad de especificar el lenguaje ni el tipo (atributos language y type), por lo tanto, con la etiqueta script es más que suficiente para abrir un bloque de código Javascript. Sin embargo, en versiones anteriores de HTML sí que estamos obligados a definir el atributo "type". A pesar que en HTML 4.01 transicional nos valide correctamente el atributo language, no validará si estamos haciendo



HTML strict, con lo que no recomendamos usar "language" en ningún caso. En los ejemplos de DesarrolloWeb.com donde se utilizaba language, por favor ignorarlo.

## Incluir ficheros externos de Javascript

Otra manera de incluir scripts en páginas web, implementada a partir de Javascript 1.1, es incluir archivos externos donde se pueden colocar muchas funciones que se utilicen en la página. Los ficheros suelen tener extensión .js y se incluyen de esta manera.

```
<script src="archivo_externo.js"></script>
```

Dentro de las etiquetas <SCRIPT> se puede escribir cualquier texto y será ignorado por el navegador, sin embargo, los navegadores que no entienden el atributo SRC tendrán a este texto por instrucciones. Es algo muy raro que esto ocurra en el panorama de navegadores. Así que no es realmente aconsejable poner ningún código Javascript dentro de un bloque script que estamos usando para incluir un fichero externo.

El archivo que incluimos (en este caso "archivo\_externo.js") debe contener tan solo sentencias Javascript. No debemos incluir código HTML de ningún tipo, ni tan siquiera las etiquetas </SCRIPT> y </SCRIPT>.

### La importancia de usar archivos de código Javascript externos

A nivel didáctico en este manual usamos mucho la práctica de incluir el código Javascript dentro del propio documento HTML. Es algo que viene muy cómodo para expresar pequeños ejemplos Javascript. Sin embargo, a nivel profesional no es una práctica aconsejable, sino que la recomendación es colocar todo, o la mayoría de tu código Javascript en archivos ".js" que incluyas de manera externa. Existen varios motivos para recomendar esta práctica, entre los cuales podemos destacar:

- Desde el punto de vista de la separación del código, cada archivo debería tener solo un lenguaje. Los documentos HTML se colocan en archivos ".html", los documentos CSS en archivos ".css" y el Javascript también separado en sus archivos ".js".
- El archivo externo permite que el navegador lo cachee, de modo que cuando la página se vuelve a visitar, el código ya se encuentra en el navegador y no lo tiene que volver a descargar. Este punto es especialmente importante cuando tenemos muchas páginas que cargan el mismo código Javascript, ya que el navegador realmente descargará una vez el archivo ".js" y las demás veces lo consumirá desde la caché, ahorrando transferencia y aumentando la velocidad de carga.
- Cuando tengas más nociones avanzadas sobre Javascript verás que en la etiqueta script puedes usar atributos como "defer" o "async". Ambos atributos provocan que el navegador no se detenga a ejecutar un script externo que se encuentre en medio del contenido HTML, sino que siga analizando la página y renderizando su contenido mientras que el archivo se

descarga. Por tanto, ambos atributos optimizan la carga de la página y mejoran la velocidad con la que el navegador presenta el contenido al usuario.

**Nota:** Cuando usas "defer" la carga del archivo se hace en paralelo con el análisis del HTML y otros archivos externos, pero la ejecución se difiere hasta que el navegador ha terminado de analizar y renderizar la página. Cuando usas "async" quieres decir que el script se descargue a la vez que se analiza la página y que, una vez descargado se ejecute, aunque no haya tardado de analizarse la página completamente.

Un ejemplo de carga y ejecución del Javascript deferida lo conseguimos así.

```
<script src="archivo_externo_deferido.js" defer></script>
```

Vistos estos otros usos interesantes que existen en Javascript y que debemos conocer para poder aprovechar las posibilidades de la tecnología, debemos haber aprendido todo lo esencial para empezar a trabajar con Javascript en el contexto de una página web.

Así que, por fin, en el próximo artículo empezaremos a introducirnos más a fondo con la programación. Para comenzar os hablaremos de la [sintaxis del lenguaje Javascript](#).

Este artículo es obra de *Miguel Angel Alvarez*  
Fue publicado / actualizado en *04/12/2019*  
Disponible online en <https://desarrolloweb.com/articulos/504.php>

## Sintaxis Javascript

**Empezamos a estudiar la sintaxis del lenguaje Javascript, deteniéndonos en sus principales características, como el uso de mayúsculas y minúsculas, los comentarios, el punto y coma y muchas otras cosas.**

Por fin empezamos a ver código fuente de Javascript! Esperamos que se hayan asimilado todas las informaciones previas del [Manual de Javascript](#), en las que hemos aprendido básicamente diversos modos de incluir scripts en páginas web. Hasta ahora todo lo que hemos visto en este manual puede haber parecido muy teórico, pero de aquí en adelante esperamos que os parezca más ameno por empezar a ver cosas más prácticas y relacionadas directamente con la programación.

# SINTAXIS JS de Javascript

El lenguaje **Javascript tiene una sintaxis muy parecida a la de Java** por estar basado en él. También es muy parecida a la del lenguaje C, de modo que si el lector conoce alguno de estos dos lenguajes se podrá manejar con facilidad con el código. De todos modos, en los siguientes capítulos vamos a describir toda la sintaxis con detenimiento, por lo que los novatos no tendrán ningún problema con ella.

## Comentarios en el código

Un comentario es una parte de código que no es interpretada por el navegador y cuya utilidad radica en facilitar la lectura al programador. El programador, a medida que desarrolla el script, va dejando frases o palabras sueltas, llamadas comentarios, que le ayudan a él o a cualquier otro a leer más fácilmente el script a la hora de modificarlo o depurarlo.

Ya se vio anteriormente algún comentario Javascript, pero ahora vamos a contarlos de nuevo. Existen dos tipos de comentarios en el lenguaje. Uno de ellos, la doble barra, sirve para comentar una línea de código. El otro comentario lo podemos utilizar para comentar varias líneas y se indica con los signos /\* para empezar el comentario y \*/ para terminarlo. Veamos unos ejemplos.

```
<SCRIPT>
//Este es un comentario de una línea
/*Este comentario se puede extender
por varias líneas.
Las que quieras*/
</SCRIPT>
```

## Mayúsculas y minúsculas

**En Javascript se han de respetar las mayúsculas y las minúsculas.** Si nos equivocamos al utilizarlas el navegador responderá con un mensaje de error, ya sea de sintaxis o de referencia indefinida.

Por poner un ejemplo, no es lo mismo la función alert() que la función Alert(). La primera muestra un texto en una caja de diálogo y la segunda (con la primera A mayúscula) simplemente no existe, a no ser que la definamos nosotros. Como se puede comprobar, para que la función la reconozca Javascript, se tiene que escribir toda en minúscula. Otro claro ejemplo lo veremos cuando tratemos

con variables, puesto que los nombres que damos a las variables también son sensibles a las mayúsculas y minúsculas.

Por regla general, los nombres de las cosas en Javascript se escriben siempre en minúsculas, salvo que se utilice un nombre con más de una palabra, pues en ese caso se escribirán con mayúsculas las iniciales de las palabras siguientes a la primera. Por ejemplo document.bgColor (que es un lugar donde se guarda el color de fondo de la página web), se escribe con la "C" de color en mayúscula, por ser la primera letra de la segunda palabra. También se puede utilizar mayúsculas en las iniciales de las primeras palabras en algunos casos, como los nombres de las clases, aunque ya veremos más adelante cuáles son estos casos y qué son las clases.

## Separación de instrucciones

Las distintas instrucciones que contienen nuestros scripts se han de separar convenientemente para que el navegador no indique los correspondientes errores de sintaxis. Javascript tiene dos maneras de separar instrucciones. La primera es a través del carácter punto y coma (;) y la segunda es a través de un salto de línea.

Por esta razón Las sentencias Javascript no necesitan acabar en punto y coma a no ser que coloquemos dos instrucciones en la misma línea.

No es una mala idea, de todos modos, acostumbrarse a utilizar el punto y coma después de cada instrucción pues otros lenguajes como Java o C obligan a utilizarlas y nos estaremos acostumbrando a realizar una sintaxis más parecida a la habitual en entornos de programación avanzados.

En el próximo artículo comenzaremos a hablaros sobre la creación de [variables en Javascript](#). Además, si te gusta aprender en vídeo en DesarrolloWeb también tienes un [Videotutorial dedicado a la sintaxis en Javascript](#).

Este artículo es obra de *Miguel Angel Alvarez*

Fue publicado / actualizado en *24/02/2020*

Disponible online en <https://desarrolloweb.com/articulos/507.php>

# Trabajo con variables y tipos de datos en Javascript

Una de las cosas más fundamentales en cualquier lenguaje de programación son las variables y los tipos de datos. Veremos qué son y cómo se trabaja con ellos en Javascript.

## Variables en Javascript

**Abordamos con detalle las variables en Javascript. Veremos en términos generales qué es una variable en los lenguajes de programación. Luego veremos cómo funcionan las variables específicamente en el lenguaje Javascript.**

Este es el primero de los artículos que vamos a dedicar a las variables en Javascript dentro del [Manual de Javascript](#). Veremos, si no lo sabemos ya, que las variables son uno de los elementos fundamentales a la hora de realizar los programas, en Javascript así como en la mayoría de los lenguajes de programación existentes.

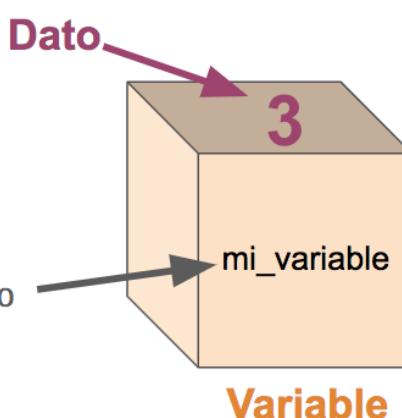
Comencemos entonces conociendo el concepto de variable y aprenderemos a declararlas en Javascript, junto con explicaciones detalladas sobre su uso en el lenguaje.

### Concepto de variable

**Una variable es un espacio en memoria donde se almacena un dato**, un espacio donde podemos guardar cualquier tipo de información que necesitemos para realizar las acciones de nuestros programas. Podemos pensar en ella como una caja, donde almacenamos un dato. Esa caja tiene un nombre, para que más adelante podamos referirnos a la variable, recuperar el dato así como asignar un valor a la variable siempre que deseemos.

## Variabls JS

Cada variable tiene un nombre, de modo que podamos acceder a ese dato siempre que necesitemos.





Por ejemplo, si nuestro programa realiza sumas, será muy normal que guardemos en variables los distintos sumandos que participan en la operación y el resultado de la suma. El efecto sería algo parecido a esto.

```
sumando1 = 23
sumando2 = 33
suma = sumando1 + sumando2
```

En este ejemplo tenemos tres variables, sumando1, sumando2 y suma, donde guardamos el resultado. Vemos que su uso para nosotros es como si tuviésemos un apartado donde guardar un dato y que se pueden acceder a ellos con sólo poner su nombre.

### Reglas para el nombrado de variables en Javascript

Los nombres de las variables han de construirse con caracteres alfanuméricos (números y letras), el carácter subrayado o guión bajo (\_) y el carácter dólar \$. Aparte de esta, hay una serie de reglas adicionales para construir nombres para variables. La más importante es que no pueden comenzar por un carácter numérico. No podemos utilizar caracteres raros como el signo +, un espacio o un signo -. Nombres admitidos para las variables podrían ser:

Edad  
paisDeNacimiento  
\_nombre  
\$elemento  
Otro\$\_Nombres

También hay que evitar utilizar nombres reservados como variables, por ejemplo no podremos llamar a nuestra variable palabras como return o for, que ya veremos que son utilizadas para estructuras del propio lenguaje. Veamos ahora algunos nombres de variables que no está permitido utilizar:

12meses  
tu nombre  
return  
for  
mas-o-menos  
pe%pe

### Los nombres de variables en Javascript son sensibles a mayúsculas y minúsculas

Recuerda que Javascript es un lenguaje sensible a mayúsculas y minúsculas, por lo que las variables también se afectan por esa distinción. Por lo tanto, no es lo mismo la variable de nombre "minombre" que la variable "miNombre". No es lo mismo "Edad" que "edad".

Ten muy en cuenta este detalle, ya que es una habitual fuente de problemas en el código que a veces son difíciles de detectar. Esto es porque a veces piensas que estás usando una variable, que debería



tener un dato determinado, pero si te equivocas al escribirla y pones mayúsculas o minúsculas donde no debería, entonces será otra variable diferente, que no tendrá el dato que se espera. Como Javascript no te obliga a declarar las variables el programa se ejecutará sin producir un error, sin embargo, la ejecución no producirá los efectos deseados.

## Declaración de variables en Javascript

Declarar variables consiste en definir, y de paso informar al sistema, que vas a utilizar una variable. Es una costumbre habitual en los lenguajes de programación el especificar explícitamente las variables que se van a usar en los programas. En muchos lenguajes de programación hay unas reglas estrictas a la hora de declarar las variables, pero lo cierto es que Javascript es bastante permisivo.

Javascript se salta muchas reglas por ser un lenguaje un tanto libre a la hora de programar y uno de los casos en los que otorga un poco de libertad es a la hora de declarar las variables, ya que no estamos obligados a hacerlo, al contrario de lo que pasa en otros lenguajes de programación como Java, C, C# y muchos otros.

### Declaración de variables con var

Javascript cuenta con la palabra "var" que utilizaremos cuando queramos declarar una o varias variables. Como es lógico, se utiliza esa palabra para definir la variable antes de utilizarla.

**Nota:** Aunque Javascript no nos obligue a declarar explícitamente las variables, es aconsejable declararlas antes de utilizarlas y veremos en adelante que se trata también de una buena costumbre. Además, en sucesivos artículos veremos que en algunos casos especiales, no producirá exactamente los mismos resultados un script en el que hemos declarado una variable y otro en el que no lo hayamos hecho, ya que la declaración o no afecta al [ámbito de las variables](#).

```
var operando1  
var operando2
```

También se puede asignar un valor a la variable cuando se está declarando

```
var operando1 = 23  
var operando2 = 33
```

También se permite declarar varias variables en la misma línea, siempre que se separen por comas.

```
var operando1,operando2
```

## Declaración de variables Javascript con let y const



Desde Javascript en versiones modernas (recuerda que Javascript es un estándar y que como tal va evolucionando en el tiempo), en concreto en Javascript en su versión ES6, existen otros modos de declarar variables:

Declaración let: Esta nueva manera de declarar las variables afecta a su ámbito, ya que son locales al bloque donde se están declarando.

Declaración const: En realidad "const" no declara una variable sino una constante, que no puede variar su valor a lo largo de la ejecución de un programa.

Quizás en este punto del manual de Javascript no es necesario profundizar mucho en estos modelos de declaración y a las personas que están aprendiendo les recomendamos centrarse en el uso de las declaraciones con "var". Sin embargo, si quieras tener más información sobre estos nuevos tipos de variables los explicamos en el artículo [Let y const: variables en ECMAScript 2015](#). Si quieras ver otras novedades del estándar de Javascript que vinieron en 2015 te recomendamos la lectura del [Manual de ES6](#).

En el siguiente artículo seguiremos aprendiendo cosas de variables en Javascript y veremos uno de los conceptos más importantes que deberemos aprender sobre ellas, el [ámbito de las variables](#).

Este artículo es obra de *Miguel Angel Alvarez*

Fue publicado / actualizado en 14/01/2020

Disponible online en <https://desarrolloweb.com/articulos/508.php>

## Ambito de las variables en Javascript

**El ámbito de las variables en Javascript: qué son las variables locales y variables globales. Cómo declarar variables o no afecta a su ámbito en Javascript.**

El ámbito de las variables es uno de los conceptos más importantes que deberemos conocer cuando trabajamos con variables, no sólo en Javascript, sino en la mayoría de los lenguajes de programación.

En el artículo anterior ya comenzamos a explicar [qué son las variables y cómo declararlas](#). En este artículo del [Manual de Javascript](#) pretendemos explicar con detenimiento qué es este ámbito de las variables y ofrecer ejemplos para que se pueda entender bien.

Al principio del artículo haremos referencia al ámbito de las variables declaradas con "var". Sin embargo hay una forma más reciente de declarar variables, con "let", que afecta directamente a su ámbito. Al final también ofreceremos algunas notas y referencias para poder entenderlo.

# ÁMBITO JS de las variables

## Concepto de ámbito de variables

**Se le llama ámbito de las variables al lugar donde estas están disponibles.** Por lo general, cuando declaramos una variable hacemos que esté disponible en el lugar donde se ha declarado, esto ocurre en todos los lenguajes de programación y como Javascript se define dentro de una página web, **las variables que declaremos en la página estarán accesibles dentro de ella.**

En Javascript no podremos acceder a variables que hayan sido definidas en otra página. Por tanto, la propia página donde se define es el ámbito más habitual de una variable y le llamaremos a este tipo de **variables globales** a la página. Veremos también se pueden hacer variables con ámbitos distintos del global, es decir, variables que declararemos y tendrán validez en lugares más acotados.

## Variables globales

Como hemos dicho, las variables globales son las que están declaradas en el ámbito más amplio posible, que en Javascript es una página web. Para declarar una variable global a la página simplemente lo haremos en un script, con la palabra *var*.

```
<SCRIPT>
var variableGlobal
</SCRIPT>
```

Las variables globales son accesibles desde cualquier lugar de la página, es decir, desde el script donde se han declarado y todos los demás scripts de la página, incluidos los manejadores de eventos, como el onclick, que ya vimos que se podía incluir dentro de determinadas etiquetas HTML.

## Variables locales

También podremos declarar variables en lugares más acotados, como por ejemplo una función. A estas variables les llamaremos locales. Cuando se declaren variables locales sólo podremos acceder a ellas dentro del lugar donde se ha declarado, es decir, si la habíamos declarado en una función solo podremos acceder a ella cuando estemos en esa función.



Las variables pueden ser locales a una función, pero también pueden ser locales a otros ámbitos, como por ejemplo un bucle. En general, son ámbitos locales cualquier lugar acotado por llaves.

```
<SCRIPT>
function miFuncion (){
    var variableLocal
}
</SCRIPT>
```

En el script anterior hemos declarado una variable dentro de una función, por lo que esa variable sólo tendrá validez dentro de la función. Se pueden ver cómo se utilizan las llaves para acotar el lugar donde está definida esa función o su ámbito.

No hay problema en declarar una variable local con el mismo nombre que una global, en este caso la variable global será visible desde toda la página, excepto en el ámbito donde está declarada la variable local ya que en este sitio ese nombre de variable está ocupado por la local y es ella quien tiene validez. En resumen, la variable que tendrá validez en cualquier sitio de la página es la global. Menos en el ámbito donde está declarada la variable local, que será ella quien tenga validez.

```
<SCRIPT>
var numero = 2
function miFuncion (){
    var numero = 19
    document.write(numero) //imprime 19
}
document.write(numero) //imprime 2
</SCRIPT>
```

Nota: Para entender este código seguramente te vendrá bien consultar el capítulo sobre la [creación de funciones en Javascript](#).

Un consejo para los principiantes podría ser no declarar variables con los mismos nombres, para que nunca haya lugar a confusión sobre qué variable es la que tiene validez en cada momento.

### Diferencias entre declarar variables con var, o no declararlas

Como hemos dicho, en Javascript tenemos libertad para declarar o no las variables con la palabra var, pero los efectos que conseguiremos en cada caso serán distintos. En concreto, cuando utilizamos var estamos haciendo que la variable que estamos declarando sea local al ámbito donde se declara. Por otro lado, si no utilizamos la palabra var para declarar una variable, ésta será global a toda la página, sea cual sea el ámbito en el que haya sido declarada.

En el caso de una variable declarada en la página web, fuera de una función o cualquier otro ámbito más reducido, nos es indiferente si se declara o no con var, desde un punto de vista funcional. Esto es debido a que cualquier variable declarada fuera de un ámbito es global a toda la página. La diferencia se puede apreciar en una función por ejemplo, ya que si utilizamos var la variable será local a la función y si no lo utilizamos, la variable será global a la página. Esta diferencia es



fundamental a la hora de controlar correctamente el uso de las variables en la página, ya que si no lo hacemos en una función podríamos sobreescribir el valor de una variable, perdiendo el dato que pudiera contener previamente.

```
<SCRIPT>
var numero = 2
function miFuncion (){
    numero = 19
    document.write(numero) //imprime 19
}
document.write(numero) //imprime 2
//llamamos a la función
miFuncion()
document.write(numero) //imprime 19
</SCRIPT>
```

En este ejemplo, tenemos una variable global a la página llamada numero, que contiene un 2. También tenemos una función que utiliza la variable numero sin haberla declarado con var, por lo que la variable numero de la función será la misma variable global numero declarada fuera de la función. En una situación como esta, al ejecutar la función se sobreescribirá la variable numero y el dato que había antes de ejecutar la función se perderá.

## Declaración de variables con let

Desde ECMAScript 2015 existe la declaración let. La sintaxis es la misma que var a la hora de declarar las variables, pero en el caso de let la declaración afecta al bloque.

Bloque significa cualquier espacio acotado por unas llaves, como podría ser las sentencias que hay dentro de las llaves de un bucle for.

```
for(let i=0; i<3; i++) {
    // en este caso, la variable i sólo existe dentro del bucle for
    alert(i);
}
// fuera del bloque for no existe la variable i
```

Si esa variable "i" hubiera sido declarada en la cabecera del bucle for mediante "var", sí que existiría fuera del bloque de código del for.

Para encontrar más información acerca de esta nueva forma de declarar variables te recomendamos leer el artículo sobre [Let y const](#). Además, si lo deseas puedes encontrar en este manual las explicaciones sobre el [bucle for](#).

En el próximo artículo continuaremos hablando de variables y mostraremos que en ellas [se pueden guardar distintos tipos de información](#).

Este artículo es obra de *Miguel Angel Alvarez*

Fue publicado / actualizado en 15/01/2020



Disponible online en <https://desarrolloweb.com/articulos/517.php>

## Qué podemos guardar en variables

**Vemos el concepto de tipos de datos para el lenguaje Javascript y por qué es importante manejarlos bien.**

En el [Manual de Javascript](#) ya hemos hablado sobre las variables en varios artículos. Pero todavía nos quedan cosas por ver y en concreto mostraremos en este artículo que en una variable podemos guardar distintos tipos de datos.

En una variable podemos introducir varios tipos de información. Por ejemplo podríamos introducir simple texto, números enteros o reales, etc. A estas distintas clases de información se les conoce como tipos de datos. Cada uno tiene características y usos distintos.

Veamos cuáles son los tipos de datos más habituales de Javascript.

### Números

Para empezar tenemos el tipo numérico, para guardar números como 9 o 23.6

### Cadenas

El tipo cadena de carácter guarda un texto. Siempre que escribamos una cadena de caracteres debemos utilizar las comillas ("").

### Booleanos

También contamos con el tipo booleano, que guarda una información que puede valer si (true) o no (false).

Por último sería relevante señalar aquí que nuestras variables pueden contener cosas más complicadas, como podría ser un objeto, una función, o vacío (null) pero ya lo veremos más adelante.

En realidad nuestras variables no están forzadas a guardar un tipo de datos en concreto y por lo tanto no especificamos ningún tipo de datos para una variable cuando la estamos declarando.

Podemos introducir cualquier información en una variable de cualquier tipo, incluso podemos ir cambiando el contenido de una variable de un tipo a otro sin ningún problema. Vamos a ver esto con un ejemplo.

```
var nombre_ciudad = "Valencia"
var revisado = true
nombre_ciudad = 32
revisado = "no"
```

Esta ligereza a la hora de asignar tipos a las variables puede ser una ventaja en un principio, sobretodo para personas inexpertas, pero a la larga puede ser fuente de errores ya que dependiendo del tipo que son las variables se comportarán de un modo u otro y si no controlamos con exactitud el tipo de las variables podemos encontrarnos sumando un texto a un número. Javascript operará perfectamente, y devolverá un dato, pero en algunos casos puede que no sea lo que estábamos esperando. Así pues, aunque tenemos libertad con los tipos, esta misma libertad nos hace estar más atentos a posibles desajustes difíciles de detectar a lo largo de los programas. Veamos lo que ocurriría en caso de sumar letras y números.

```
var sumando1 = 23
var sumando2 = "33"
var suma = sumando1 + sumando2
document.write(suma)
```

Este script nos mostraría en la página el texto 2333, que no se corresponde con la suma de los dos números, sino con su concatenación, uno detrás del otro.

Veremos [algunas cosas más referentes a los tipos de datos](#) en el próximo artículo.

Este artículo es obra de *Miguel Angel Alvarez*  
Fue publicado / actualizado en *24/08/2001*  
Disponible online en <https://desarrolloweb.com/articulos/518.php>

## Tipos de datos en Javascript

**Tipos en Javascript:** qué son los tipos de datos, qué tipos soporta Javascript, desde los más básicos como los numéricos, booleanos y las cadenas de texto, hasta otros más complejos como los objetos.



En el artículo anterior del Manual de Javascript ya empezamos a mostrar que [en una variable podemos almacenar distintos tipos de información](#). No obstante, todavía hay algunas cosas que queremos explicar sobre los distintos **tipos de datos disponibles en Javascript**.



En nuestros scripts vamos a manejar variables diversas clases de información, como textos o números. Cada una de estas clases de información son los **tipos de datos**. Todos los datos que se puedan guardar en variables van a estar encajadas en uno de los tipos de datos disponibles en el lenguaje.

## Qué son tipos de datos

Antes de comenzar a ver los tipos de datos en Javascript vamos a pararnos para definir mejor qué es un tipo de datos.

**Un tipo de datos es un conjunto de valores que comparten unos determinados atributos, sobre los que podemos hacer unas determinadas operaciones.** Los lenguajes de programación usan los tipos de datos para definir y acotar los valores que podrían contener las variables y definir qué tipos de operaciones que podríamos realizar con ellos.

Existen tipos de datos clásicos que se encuentran en la mayoría de los lenguajes de programación, como los tipos de datos numéricos, las cadenas de caracteres o los tipos de datos booleanos. A estos tipos de datos se les conoce como **tipos de datos primitivos**. Pero también existen tipos de datos un poco más avanzados, como los arrays u objetos, a los que se les suele llamar **tipos de datos compuestos**.

## Tipos de datos primitivos en Javascript

Los tipos de datos simples o tipos de datos primivivos que distingue Javascript son tres. Veamos detenidamente cuáles son estos tres tipos de datos.

### Tipo de datos numérico

En este lenguaje **sólo existe un tipo de datos numérico**, al contrario que ocurre en la mayoría de los lenguajes más conocidos. Todos los números son por tanto del tipo numérico, independientemente de si son números reales (con decimales) o números enteros (sin decimales), incluso independientemente de la precisión que tengan.

Los números enteros son números que no tienen coma, como 3 o 339. Los números reales son números fraccionarios, como 2.69 o 0.25, que también se pueden escribir en notación científica, por ejemplo 2.482e12.

Vamos a ver algunos ejemplos de variables creadas con tipos de datos numéricos:

```
let entero = 22;
let real = 4.9;
let realNotacionCientifica = 1.23e+22;
```

Con Javascript también podemos escribir números en otras bases, como la hexadecimal. Las bases son sistemas de numeración que utilizan más o menos dígitos para escribir los números. Existen

tres bases con las que podemos trabajar

- Base 10, es el sistema que utilizamos habitualmente, el sistema decimal. Cualquier número, por defecto, se entiende que está escrito en base 10.
- Base 8, también llamado sistema octal, que utiliza dígitos del 0 al 7. Para escribir un número en octal basta con escribir ese número precedido de un 0, por ejemplo 045.
- Base 16 o sistema hexadecimal, es el sistema de numeración que utiliza 16 dígitos, los comprendidos entre el 0 y el 9 y las letras de la A a la F, para los dígitos que faltan. Para escribir un número en hexadecimal debemos escribirlo precedido de un cero y una equis, por ejemplo ox3EF.

```
let enteroBase10 = 598; let enteroBase8 = 077; let enteroBase16 = ox6F8A;
```

### Tipo booleano

El tipo boolean, boolean en inglés, sirve para guardar un **valor lógico**: *si* o *no*. Dicho de otro modo los tipos booleanos permiten solamente dos tipos de valores, verdadero o falso. Se utiliza para realizar operaciones lógicas, generalmente para realizar acciones si el contenido de una variable es verdadero o falso.

Este tipo de datos generalmente lo usamos para tomar decisiones en un programa. Por ejemplo:

Si una variable es verdadero ----- Ejecuto unas instrucciones Si no ----- Ejecuto otras instrucciones

Los dos valores que pueden tener las variables booleanas son true o false.

```
let miBoleana = true  
let miBoleana = false
```

### Tipo de datos cadena de caracteres

El último tipo de datos primitivo que existe en Javascript de manera tradicional es el que sirve para guardar un texto, lo que solemos llamar **cadena de caracteres** o string en inglés.

Javascript sólo tiene un tipo de datos para guardar texto y en el se pueden introducir cualquier número de caracteres. Un texto puede estar compuesto de números, letras y cualquier otro tipo de caracteres y signos. Los textos se escriben entre comillas, dobles o simples.

```
let miTexto = "Pepe se va a pescar";  
miTexto = '23%$ Letras & *---';
```

Todo lo que se coloca entre comillas, como en los ejemplos anteriores, es tratado como una cadena de caracteres independientemente de lo que coloquemos en el interior de las comillas. Por ejemplo, en una variable de texto podemos guardar números y en ese caso tenemos que tener en cuenta que las variables de tipo texto y las numéricas no son la misma cosa y mientras que las de numéricas nos sirven para hacer cálculos matemáticos las de texto no.

## Caracteres de escape en cadenas de texto

Ya que hablamos de cadenas de texto queremos explicar qué son los caracteres de escape. Hay una serie de caracteres especiales que sirven para expresar en una cadena de texto determinados controles como puede ser un salto de línea o un tabulador. Estos son los caracteres de escape y se escriben con una notación especial que comienza por una contra barra (una barra inclinada al revés de la normal '\') y luego se coloca el código del carácter a mostrar.

Un carácter muy común es el salto de línea, que se consigue escribiendo \n. Otro carácter muy habitual es colocar unas comillas, pues si colocamos unas comillas sin su carácter especial nos cerrarían las comillas que colocamos para iniciar la cadena de caracteres. Las comillas las tenemos que introducir entonces con \" o \' (comillas dobles o simples). Existen otros caracteres de escape, que veremos en la tabla de abajo más resumidos, aunque también hay que destacar como carácter habitual el que se utiliza para escribir una contrabarra, para no confundirla con el inicio de un carácter de escape, que es la doble contrabarra \\.

Veamos ahora la tabla con todos los caracteres de escape

Salto de línea: \n

Comilla simple: \'

Comilla doble: \"

Tabulador: \t

Retorno de carro: \r

Avance de página: \f

Retroceder espacio: \b

Contrabarra: \\

Algunos de estos caracteres probablemente no los llegarás a utilizar nunca, pues su función es un poco rara y a veces poco clara.

## Otros tipos de datos primitivos en Javascript

En Javascript existen otros tipos de datos primitivos, que no son tan importantes como los que hemos visto anteriormente, pero que no podemos dejar de comentar para que esta información esté suficientemente completa.

Son los siguientes:

### Undefined

Este es el tipo de datos que tiene Javascript cuando una variable ha sido declarada pero no ha sido inicializada.

Por ejemplo, tenemos esta variable no inicializada:

```
let noInicializada;
```

Como no hemos asignado todavía ningún valor a la variable, no tiene un tipo de datos definido. En este caso Javascript asume que tendrá el tipo de datos "undefined".

### BigInt

Este es un tipo de datos agregado más recientemente al lenguaje Javascript. Simplemente permite almacenar números mayores en él.

Tienes que tener en cuenta que los números en Javascript o en cualquier otro lenguaje tienen una capacidad, que la define el propio lenguaje o el tipo de hardware sobre el que se ejecuta un programa. Los números enteros en Javascript no son ajenos a estas limitaciones. Permiten números grandes, pero no llegan a todas partes.

El tipo BigInt se creó para ampliar el rango de valores posibles en los tipos enteros soportados tradicionalmente por Javascript. Con BigInt podemos operar con números enormes de manera segura. Para poder definir valores del tipo de datos bigint tenemos que agregar una "n" al final del valor.

```
let valorBigInt = 4n;
```

Como ves, solamente hemos colocado una "n" al final. Aunque el valor es pequeño, esta variable podrá contener si fuera necesario valores gigantescos.

### Symbol

Este tipo de datos es bastante raro y no demasiado usado. También es un tipo de datos agregado a Javascript de manera reciente. Su función consiste en permitir tener un valor inmutable y que se garantiza que será diferente a cualquier otro valor que pueda crearse en cualquier otro momento en un programa.

Como puedes deducir, es algo un poco avanzado y, ahora que estamos comenzando con Javascript, nos interesa poquito. Aún así vamos a poner un ejemplo sobre cómo crear un symbol.

```
let miSymbol = Symbol('hohoho');
```

### Tipos de datos compuestos

Aunque todavía es un poco pronto para dar muchos detalles (llegará el momento en este mismo [manual de Javascript](#)), queremos mencionar algunas cosas con respecto a los tipos de datos compuestos.

Estos tipos de datos son aquellos un poco más complejos, que no están compuestos generalmente por un único valor. Es decir, cuando guardamos un número en una variable tenemos un único valor, pero el lenguaje es capaz de guardar en variables valores que tienen distintos datos. Es el caso de los objetos o los arrays.

## Objetos

Los objetos son un tema un poco más avanzado y los vamos a tratar más adelante con todo detalle. No obstante te adelantamos que son valores que se colocan entre llaves y donde tenemos distintos elementos. Cada uno de los elementos de un objeto tiene una clave y un valor. Se dice que son compuestos porque en un objeto puedo tener distintas claves con distintos valores. No hay límite, es decir, puedo tener un objeto con una única clave y un único valor y objetos con miles de claves.

Veamos un ejemplo sencillo de objeto en Javascript:

```
let persona = {  
    nombre: 'Miguel Angel',  
    apellidos: 'Alvarez Sanchez',  
    profesion: 'Informático'  
}
```

Además, si lo consideramos necesario, los valores de algunas claves pueden ser otros objetos, en cualquier nivel de anidación que sea oportuno.

```
let empresa = {  
    nombre: 'EscuelaIT',  
    CIF: 'B12345678',  
    direccion: {  
        calle: 'C/ Valencia',  
        numero: 2,  
        codigoPostal: '12345'  
    },  
    empresaActiva: true  
}
```

Puedes obtener más información en el artículo de [Introducción general a los objetos en Javascript](#).

## ¿Qué pasa con los Arrays?

También tenemos que hablar sobre una estructura de datos compuesta que existe en todos los lenguajes de programación: los arrays, también llamados arreglos, tablas o vectores. Un ejemplo de array lo tienes a continuación:

```
let miArray = [1, 2, 3];
```

Esta estructura de datos nos permite guardar varios elementos en una única variable. Puedes encontrar más información sobre ellos más adelante cuando lleguemos al [capítulo de arrays](#). Lo que queremos decir por ahora es que, aunque parezca raro, **los arrays no son un tipo de datos específico de Javascript**. En este lenguaje los trata como si fueran objetos.

## Tipo null

En Javascript también existe un tipo de datos específico cuando tenemos una referencia a un lugar donde no hay nada. Eso se especifica con el valor `null`.



Podríamos discutir sobre si null es o no es un tipo de datos, porque lo cierto es que, cuando le preguntas a Javascript de qué tipo es el valor `null` el lenguaje te responde que es un objeto. Por tanto, podríamos decir que null es un tipo de objeto particular que está vacío. Ya hablaremos de él más adelante también.

## Funciones

Las funciones son otro de los tipos de datos que maneja Javascript. Las funciones son una de las piezas fundamentales para estructurar el código de las aplicaciones y poder reutilizarlo.

También es un poco pronto para hablar de funciones. Lo veremos más adelante pero si te quieres adelantar te dejamos el enlace al capítulo inicial sobre las [funciones en Javascript](#).

Las funciones en Javascript también se pueden meter en variables. Es un poco raro en algunos lenguajes de programación, pero en Javascript es muy común.

```
let miFuncion = function() {  
    // Hacer alguna cosa...  
}
```

## Cómo averiguar el tipo de una variable en Javascript

Ya que hablamos de tipos debemos mencionar que en Javascript tenemos un operador que usamos para saber el tipo particular de una variable. Es el operador `typeof`. Los operadores los vamos a estudiar en el capítulo siguiente de este manual, pero vamos a ver un ejemplo que nos permitiría saber el tipo de una variable.

```
let y = 'gol';  
alert(typeof y);
```

Hemos creado una variable `y` con el valor de una cadena de caracteres. A continuación hemos mandado a una caja de diálogo `alert` el valor de su tipo, por lo que Javascript nos mostrará que es un "string".

```
let num = 444;  
alert(typeof num);
```

En este caso la caja de diálogo nos dirá que el tipo es "number".

Puedes experimentar a usar `typeof` con diversos tipos de datos de los que hemos hablado en este artículo para practicar por tu cuenta.

## Qué son los literales



Por último queremos hablar de una jerga muy común en el mundo de la programación que es la denominación "literal".

**Un literal es un valor escrito tal cual en el código de un programa.** Al asignar valores en las variables usamos muy frecuentemente literales con aquello que queremos introducir en ellas. Tenemos literales de todos los tipos, por ejemplo literales de cadena, literales numéricos, literales de objeto.

Por ejemplo:

```
let x = 'literal de cadena';
let y = 902;
let z = { key: 'val' };
let k = [4, 'hola', 5];
```

En el código anterior tenemos asignaciones en variables de literales de distintos tipos. Por ejemplo 'literal de cadena' es un literal de tipo string. 902 es un literal de tipo number. { key: 'val' } es un literal de tipo object. [4, 'hola', 5] es un literal de array, aunque en Javascript los arrays ya se dijeron considerados de tipo object.

El operador typeof lo puedes usar para comprobar el tipo de cualquier literal que quieras, directamente, sin que lo necesites almacenar en una variable.

```
typeof '1234' // devuelve 'string'
typeof 03 // devuelve 'number'
typeof true // devuelve 'boolean'
typeof [1] // devuelve 'object'
typeof undefined // devuelve 'undefined'
```

## Valores de tipo number especiales

Aunque quizás sea un poco pronto para entrar en detalles como lo que vamos a relatar ahora, por ser un poco aburridos, queremos dejar en algún lugar unas notas sobre cosas que también son de tipo number en Javascript.

Dentro de los valores del tipo de datos "number" encontramos algunos especiales que te podrás encontrar en el día a día.

### Infinity

El primero de ellos es el valor `Infinity`, que como su nombre indica es el equivalente al número o concepto infinito.

Fíjate que `Infinity` lleva la primera I en mayúscula. Esto es importante ya que Javascript distingue entre mayúsculas y minúsculas.



El valor `Infinity` puede aparecer en Javascript como por ejemplo cuando hacemos una división por cero.

```
let x = 1 / 0; // El valor de x será Infinity
```

Si usamos el operador `typeof` con el valor `Infinity` nos dirá que es de tipo "number".

```
typeof Infinity // devuelve number
```

## NaN

El valor `NaN` (cuyas letras están compuestas por las iniciales de Not a Number) significa que ese valor no es un número.

Curiosamente, si le preguntamos a Javascript el tipo de `NaN` nos dirá que es un número.

```
typeof NaN // devuelve number
```

El valor `NaN` se produce en Javascript cuando hacemos una operación en la que el resultado no sea un número. Por ejemplo podemos usar el operador unario `+` que es un operador que solo recibe un dato e intenta convertir ese dato en un número entero.

```
+"33" // devuelve el número 33
```

Sin embargo, si lo que le damos es una cadena no numérica, nos devolverá `NaN`.

```
+"a" // devuelve NaN
```

También obtenemos `NaN` con operaciones matemáticas con tipos de datos numéricos y cadenas no numéricas.

```
5 / "hola" // devuelve NaN
```

Sin embargo, tienes que tener cuidado con los tipos porque hay operaciones que a simple vista parece matemáticas pero que no dan este mismo comportamiento.

```
5 + "hola" // devuelve la cadena "5hola"
```

Esto último ocurre porque el operador `+` es el operador de la concatenación entre cadenas de texto. Lo que está haciendo Javascript es considerar la concatenación en lugar de la suma, debido a que uno de los dos operadores es una cadena de texto.



Este trabajo que hace Javascript al realizar operaciones en las que intervienen dos tipos distintos, o realizar operaciones con tipos que realmente no son los adecuados para tales operaciones, es un detalle importante, que a veces trae bastantes quebraderos de cabeza. Si quieres profundizar un poco más a fondo sobre este asunto te recomendamos leer el artículo sobre [Coerción de tipos en Javascript](#).

### Conclusión sobre las variables y los tipos de datos

Con esto ya hemos terminado de explicar todo lo que se debe conocer sobre las variables y sus tipos de datos en Javascript. A continuación podemos comenzar con un tema nuevo que será el de [operadores en Javascript](#).

Este artículo es obra de *Miguel Angel Alvarez*

Fue publicado / actualizado en *02/05/2023*

Disponible online en <https://desarrolloweb.com/articulos/519.php>

# Operadores en Javascript

Tratamos en diversos artículos los operadores. Ofreceremos explicaciones de todos los operadores que podremos encontrarnos en Javascript.

## Operadores Javascript

**Estudiamos lo que es un operador y para qué sirve. Vemos los operadores de Javascript, en diversas clasificaciones, aritméticos, asignación, comparación, condicionales, a nivel de bit y preferencia de operadores.**

En el presente artículo del [Manual de Javascript](#) vamos a comenzar una serie de textos enfocados en explicar los diferentes operadores disponibles en este lenguaje de programación.

Al desarrollar programas en cualquier lenguaje se utilizan los operadores, que sirven para hacer los cálculos y operaciones necesarios para llevar a cabo tus objetivos. Hasta el menor de los programas imaginables necesita de los operadores para realizar cosas, ya que un programa que no realizase operaciones, sólo se limitaría a hacer siempre lo mismo.

Es el resultado de las operaciones lo que hace que un programa varíe su comportamiento según los datos que tenga para trabajar y nos ofrezca resultados que sean relevantes para el usuario que lo utilice. Existen operaciones más sencillas o complejas, que se pueden realizar con operandos de distintos tipos, como números o textos, veremos en este capítulo, y los siguientes, de manera detallada todos estos operadores disponibles en Javascript.

### Ejemplos de uso de operadores

Antes de entrar a enumerar los distintos tipos de operadores vamos a ver un par de ejemplos de éstos para que nos ayuden a hacernos una idea más exacta de lo que son. En el primer ejemplo vamos a realizar una suma utilizando el operador suma.

3 + 5

Esta es una expresión muy básica que no tiene mucho sentido ella sola. Hace la suma entre los dos operandos número 3 y 5, pero no sirve de mucho porque no se hace nada con el resultado.

Normalmente se combinan más de un operador para crear expresiones más útiles. La expresión siguiente es una combinación entre dos operadores, uno realiza una operación matemática y el otro sirve para guardar el resultado.

```
miVariable = 23 * 5
```



En el ejemplo anterior, el operador \* se utiliza para realizar una multiplicación y el operador = se utiliza para asignar el resultado en una variable, de modo que guardemos el valor para su posterior uso.

Los operadores se pueden clasificar según el tipo de acciones que realizan. A continuación vamos a ver cada uno de estos grupos de operadores y describiremos la función de cada uno.

## Operadores aritméticos

Son los utilizados para la realización de operaciones matemáticas simples como la suma, resta o multiplicación. En javascript son los siguientes:

- + Suma de dos valores
- Resta de dos valores, también puede utilizarse para cambiar el signo de un número si lo utilizamos con un solo operando -23
- \* Multiplicación de dos valores
- / División de dos valores
- % El resto de la división de dos números (3%2 devolvería 1, el resto de dividir 3 entre 2)
- ++ Incremento en una unidad, se utiliza con un solo operando
- Decremento en una unidad, utilizado con un solo operando

## Ejemplos

```
precio = 128 //introduzco un 128 en la variable precio
unidades = 10 //otra asignación, luego veremos operadores de asignación
factura = precio * unidades //multiplico precio por unidades, obtengo el valor factura
resto = factura % 3 //obtengo el resto de dividir la variable factura por 3
precio++ //incrementa en una unidad el precio (ahora vale 129)
```

## Operadores de asignación

Sirven para asignar valores a las variables, ya hemos utilizado en ejemplos anteriores el operador de asignación =, pero hay otros operadores de este tipo, que provienen del lenguaje C y que muchos de los lectores ya conocerán.

- = Asignación. Asigna la parte de la derecha del igual a la parte de la izquierda. A al derecha se colocan los valores finales y a la izquierda generalmente se coloca una variable donde queremos guardar el dato.
- += Asignación con suma. Realiza la suma de la parte de la derecha con la de la izquierda y guarda el resultado en la parte de la izquierda.
- = Asignación con resta
- \*= Asignación de la multiplicación
- /= Asignación de la división
- %= Se obtiene el resto y se asigna

## Ejemplos



```
ahorros = 7000 //asigna un 7000 a la variable ahorros  
ahorros += 3500 //incrementa en 3500 la variable ahorros, ahora vale 10500  
ahorros /= 2 //divide entre 2 mis ahorros, ahora quedan 5250
```

En el siguiente artículo seguiremos conociendo otros de los operadores de Javascript: [Operadores de cadenas, operadores lógicos y operadores condicionales](#).

## Operadores de cadenas

Las cadenas de caracteres, o variables de texto, también tienen sus propios operadores para realizar acciones típicas sobre cadenas. Aunque javascript sólo tiene un operador para cadenas se pueden realizar otras acciones con una serie de funciones predefinidas en el lenguaje que veremos más adelante.

+ Concatena dos cadenas, pega la segunda cadena a continuación de la primera.

### Ejemplo

```
cadena1 = "holo"  
cadena2 = "mundo"  
cadenaConcatenada = cadena1 + cadena2 //cadena concatenada vale "holamundo"
```

Un detalle importante que se puede ver en este caso es que el operador + sirve para dos usos distintos, si sus operandos son números los suma, pero si se trata de cadenas las concatena. Esto pasa en general con todos los operadores que se repiten en el lenguaje, javascript es suficientemente listo para entender que tipo de operación realizar mediante una comprobación de los tipos que están implicados en ella.

Un caso que resultaría confuso es el uso del operador + cuando se realiza la operación con operadores texto y numéricos entremezclados. En este caso javascript asume que se desea realizar una concatenación y trata a los dos operandos como si de cadenas de caracteres se trataran, incluso si la cadena de texto que tenemos fuese un número. Esto lo veremos más fácilmente con el siguiente ejemplo.

```
miNumero = 23  
miCadena1 = "pepe"  
miCadena2 = "456"  
resultado1 = miNumero + miCadena1 //resultado1 vale "23pepe"  
resultado2 = miNumero + miCadena2 //resultado2 vale "23456"  
miCadena2 += miNumero //miCadena2 ahora vale "45623"
```

Como hemos podido ver, también en el caso del operador +=, si estamos tratando con cadenas de texto y números entremezclados, tratará a los dos operadores como si fuesen cadenas.

**Nota:** Como se puede haber imaginado, faltan muchas operaciones típicas a realizar con cadenas, para las cuales no existen operadores. Es porque esas funcionalidades se obtienen a



través de la [clase String de Javascript](#), que veremos más adelante.

## Operadores lógicos

Estos operadores sirven para realizar operaciones lógicas, que son aquellas que dan como resultado un verdadero o un falso, y se utilizan para tomar decisiones en nuestros scripts. En vez de trabajar con números, para realizar este tipo de operaciones se utilizan operandos booleanos, que conocimos anteriormente, que son el verdadero (true) y el falso (false). Los operadores lógicos relacionan los operandos booleanos para dar como resultado otro operando booleano, tal como podemos ver en el siguiente ejemplo.

Si tengo hambre y tengo comida entonces me pongo a comer

Nuestro programa Javascript utilizaría en este ejemplo un operando booleano para tomar una decisión. Primero mirará si tengo hambre, si es cierto (true) mirará si dispongo de comida. Si son los dos ciertos, se puede poner a comer. En caso de que no tenga comida o que no tenga hambre no comería, al igual que si no tengo hambre ni comida. El operando en cuestión es el operando Y, que valdrá verdadero (true) en caso de que los dos operandos valgan verdadero.

**Nota:** Para no llevarnos a engaño, cabe decir que los operadores lógicos pueden utilizarse en combinación con tipos de datos distintos de los booleanos, pero en este caso debemos utilizarlos en expresiones que los conviertan en booleanos. En el siguiente grupo de operadores que vamos a tratar en este artículo hablaremos sobre los operadores condicionales, que se pueden utilizar junto con los operadores lógicos para realizar sentencias todo lo complejas que necesitemos. Por ejemplo:

```
if (x==2 && y!=3){  
    //la variable x vale 2 y la variable y es distinta de tres  
}
```

En la expresión condicional anterior estamos evaluando dos comprobaciones que se relacionan con un operador lógico. Por una parte `x==2` devolverá un true en caso que la variable `x` valga 2 y por otra, `y!=3` devolverá un true cuando la variable `y` tenga un valor distinto de 3. Ambas comprobaciones devuelven un booleano cada una, que luego se le aplica el operador lógico `&&` para comprobar si ambas comprobaciones se cumplieron al mismo tiempo.

Sobra decir que, para ver ejemplos de operadores condicionales, necesitamos aprender estructuras de control como `if`, a las que no hemos llegado todavía.

`!` Operador NO o negación. Si era true pasa a false y viceversa.

`&&` Operador Y, si son los dos verdaderos vale verdadero.

`||` Operador O, vale verdadero si por lo menos uno de ellos es verdadero.



## Ejemplo

```
miBooleano = true  
miBooleano = !miBooleano //miBooleano ahora vale false  
tengoHambre = true  
tengoComida = true  
comoComida = tengoHambre && tengoComida
```

## Operadores condicionales

Sirven para realizar expresiones condicionales todo lo complejas que deseemos. Estas expresiones se utilizan para tomar decisiones en función de la comparación de varios elementos, por ejemplo si un numero es mayor que otro o si son iguales.

**Nota:** Por supuesto, los operadores condicionales sirven también para realizar expresiones en las que se comparan otros tipos de datos. Nada impide comparar dos cadenas, para ver si son iguales o distintas, por ejemplo. Incluso podríamos comparar booleanos.

Los operadores condicionales se utilizan en las expresiones condicionales para tomas de decisiones. Como estas expresiones condicionales serán objeto de estudio más adelante será mejor describir los operadores condicionales más adelante. De todos modos aquí podemos ver la tabla de operadores condicionales.

== Comprueba si dos valores son iguales  
!= Comprueba si dos valores son distintos  
> Mayor que, devuelve true si el primer operando es mayor que el segundo  
< Menor que, es true cuando el elemento de la izquierda es menor que el de la derecha  
>= Mayor igual  
<= Menor igual

Veremos ejemplos de operadores condicionales cuando expliquemos estructuras de control, como la condicional if.

De manera adicional, en este texto veremos un asunto de bastante importancia en la programación en general, que es la precedencia de operadores, que debemos tener en cuenta siempre que utilicemos diversos operadores en una misma expresión, para que se relacionen entre ellos y se resuelvan de la manera habíamos planeado.

## Operadores a nivel de bit

Estos son muy poco corrientes y es posible que nunca los llegues a utilizar. Su uso se realiza para efectuar operaciones con ceros y unos. Todo lo que maneja un ordenador son ceros y unos, aunque nosotros utilicemos números y letras para nuestras variables en realidad estos valores están escritos internamente en forma de ceros y unos. En algunos caso podremos necesitar realizar operaciones tratando las variables como ceros y unos y para ello utilizaremos estos operandos. En este manual



se nos queda un poco grande realizar una discusión sobre este tipo de operadores, pero aquí podréis ver estos operadores por si algún día os hacen falta.

& Y de bits

^ Xor de bits

| O de bits

<< >> >>> >>>= >>= <<= Varias clases de cambios

## Precedencia de los operadores

La evaluación de una sentencia de las que hemos visto en los ejemplos anteriores es bastante sencilla y fácil de interpretar, pero cuando en una sentencia entran en juego multitud de operadores distintos puede haber una confusión a la hora de interpretarla y dilucidar qué operadores son los que se ejecutan antes que otros. Para marcar unas pautas en la evaluación de las sentencias y que estas se ejecuten siempre igual y con sentido común existe la precedencia de operadores, que no es más que el orden por el que se irán ejecutando las operaciones que ellos representan. En un principio todos los operadores se evalúan de izquierda a derecha, pero existen unas normas adicionales, por las que determinados operadores se evalúan antes que otros. Muchas de estas reglas de precedencia están sacadas de las matemáticas y son comunes a otros lenguajes, las podemos ver a continuación.

0 [] . Paréntesis, corchetes y el operador punto que sirve para los objetos

! - ++ -- negación, negativo e incrementos

\* / % Multiplicación división y módulo

+- Suma y resta

<< >> >>> Cambios a nivel de bit

< <= > >= Operadores condicionales

== != Operadores condicionales de igualdad y desigualdad

& ^ | Lógicos a nivel de bit

&& || Lógicos booleanos

= += -= \*= /= %= <<= >>= >>>= &= ^= != Asignación

En los siguientes ejemplos podemos ver cómo las expresiones podrían llegar a ser confusas, pero con la tabla de precedencia de operadores podremos entender sin errores cuál es el orden por el que se ejecutan.

12 \* 3 + 4 - 8 / 2 % 3

En este caso primero se ejecutan los operadores \* / y %, de izquierda a derecha, con lo que se realizarían estas operaciones. Primero la multiplicación y luego la división por estar más a la izquierda del módulo.

36 + 4 - 4 % 3

Ahora el módulo.

36 + 4 - 1

Por último las sumas y las restas de izquierda a derecha.

40 - 1

Lo que nos da como resultado el valor siguiente.

39

De todos modos, es importante darse cuenta que el uso de los paréntesis puede ahorrarnos muchos quebraderos de cabeza y sobretodo la necesidad de sabernos de memoria la tabla de precedencia de los operadores. Cuando veamos poco claro el orden con el que se ejecutarán las sentencias podemos utilizarlos y así forzar que se evalúe antes el trozo de expresión que se encuentra dentro de los paréntesis.

Para acabar con el tema de operadores de Javascript básicos nos quedan por ver el [Operador ternario en Javascript](#) y también [otro operador un tanto especial, llamado typeof](#). En versiones del lenguaje más modernas añadieron más operadores que podrás conocer en el [Manual de Javascript ES6](#).

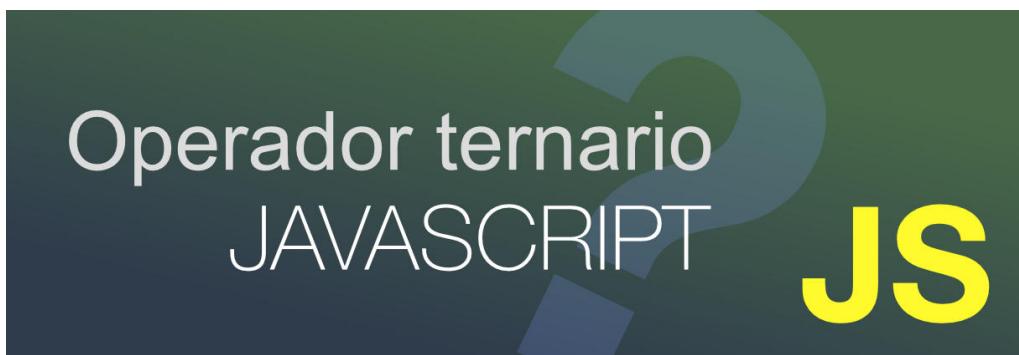
Este artículo es obra de *Miguel Angel Alvarez*

Fue publicado / actualizado en *20/09/2001*

Disponible online en <https://desarrolloweb.com/articulos/529.php>

## Operador ternario en Javascript

**En qué consiste el operador ternario de Javascript, también llamado operador condicional, con ejemplos de uso y recomendaciones.**



Después de haber conocido los [operadores de Javascript](#) más importantes, vamos a abordar otro operador relevante, aunque no tan usado. Se trata del operador ternario de Javascript, el cual



resulta de mucha utilidad en el día a día cuando queremos sacar un valor posible dependiendo de una expresión condicional.

## Qué es el operador ternario

El operador ternario lo podemos entender como una **estructura compacta para hacer condicionales**. Consiste en una expresión que se evaluará y, dependiendo de si dicha evaluación fue positiva o negativa devolverá una u otra cosa.

Su sintaxis es la siguiente:

```
expresión_condicional ? expresion1 : expresion2;
```

La "expresión\_condicional" será la que se evalue, positiva o negativamente. "expresion1" se ejecutará en caso que sea positiva y "expresion2" se ejecutará si era negativa. Adicionalmente, como resultado de la ejecución del operador ternario, se retornará aquello que devuelva la expresión finalmente ejecutada (la 1 o la 2).

Este operador condicional se conoce como ternario porque tiene tres operadores, tanto la expresión condicional a evaluar como dos expresiones a ejecutar para el caso positivo o el negativo.

Veamos un ejemplo de uso:

```
(x < 1000) ? x : 1000;
```

Ese código entregará el valor de x si la variable x era un número menor de 1000. En caso contrario (x no era menor de 1000) devolverá 1000.

Como esta sentencia termina devolviendo el valor resultado de una expresión u otra, es común que se use junto con una sentencia de asignación.

```
var valor_x_limitado_a_1000 = (x < 1000) ? x : 1000;
```

Al ejecutarse la anterior línea de código tendremos una variable llamada "valor\_x\_limitado\_a\_1000" que contendrá el valor de x o, si x sobrepasa 1000, contendrá el valor 1000.

Otro ejemplo, podríamos sacar el nombre de un día de la semana, asegurándonos que nos den un número de día correcto. Para construir este ejemplo necesitamos tener un array con los números de la semana;

```
var weekDays = ['lunes', 'martes', 'miércoles', 'jueves', 'viernes', 'sábado', 'domingo'];
```



Luego podríamos usar el operador ternario de esta manera:

```
var dia = (numeroDeDia < weekDays.length) ? weekDays[numeroDeDia - 1] : 'dia incorrecto';
```

El código completo de un script que podría valernos para probar esta sentencia podría ser el siguiente:

```
var numeroDeDia = 3;
var weekDays = ['lunes', 'martes', 'miércoles', 'jueves', 'viernes', 'sábado', 'domingo'];
var dia = (numeroDeDia < weekDays.length) ? weekDays[numeroDeDia - 1] : 'dia incorrecto';
console.log(dia);
```

Como has visto, hemos conseguido un comportamiento muy compacto. Sin usar el operador ternario también podríamos haber obtenido un resultado válido pero con más líneas de código. Veamos este mismo comportamiento con un `if` de toda la vida.

```
var numeroDeDia = 3;
var weekDays = ['lunes', 'martes', 'miércoles', 'jueves', 'viernes', 'sábado', 'domingo'];
if (numeroDeDia < weekDays.length) {
    var dia = weekDays[numeroDeDia - 1];
} else {
    var dia = 'dia incorrecto';
}
console.log(dia);
```

## Recomendaciones de uso del operador condicional

Este operador es muy potente, dado que consigue un comportamiento bastante amplio para el **poco código usado**. Sin embargo, en ocasiones, el resultado no es tan claro como podría ser un simple "if".

Por tanto, debemos usarlo con precaución, analizando si verdaderamente es necesario o no, así como si facilita realmente la lectura del código y por tanto su mantenimiento, o por contra, lo empeora.

Por ejemplo, vamos a volver para el ejemplo del número limitado a 1000. ¿Cómo haríamos para controlar el caso especial en el que se entregue algo que no sea un número? Queremos que se controle:

- Si no es un número, queremos el valor 0
- Si es un número, queremos que lo limite a 1000 como máximo

Esa lógica la podríamos expresar así:

```
var limitado = (isNaN(x)) ? 0 : (x < 1000) ? x : 1000;
```

No sé si estarás de acuerdo que la sentencia comienza a resultar poco clara. Quizás habría que cambiar la estrategia por algo más sencillo como:



```
var limitado
if(isNaN(x)) {
    limitado = 0;
} else {
    limitado = (x < 1000) ? x : 1000;
}
```

Estos dos códigos son equivalentes. La segunda alternativa tiene claramente más código, pero quizás para muchas personas sea más clara que la primera. A la vista del if, queda patente que primero deseamos comprobar si no es un número, en cuyo caso el valor limitado será 0. Luego, si era un número, entonces se entrega el valor limitado a 1000.

Por poner otro tipo de ejemplo. Queremos una función que nos indique si un número es par o impar. El código podría ser este:

```
function esPar(valor) {
    return valor % 2 == 0 ? true : false;
}
```

¿Qué opinas de ese código? ¿es suficientemente claro?

Es una pregunta con trampa, porque en este caso el problema no es la claridad, es que realmente es innecesario el operador condicional. Podríamos haber escrito simplemente esto:

```
function esPar(valor) {
    return valor % 2 == 0;
}
```

## Conclusión

Hemos aprendido a usar el **operador ternario**, también llamado **operador condicional**, de Javascript. Es una herramienta útil por su potencia y la capacidad de implementar una operación condicional con poco código.

Este operador lo encontrarás en muchos otros lenguajes de programación, por lo que resulta muy extendido. Es importante que lo conozcas y lo tengas a mano para usarlo cuando realmente ayude a crear un código más resumido, siempre con cuidado para no perder su facilidad de lectura y comprensión.

Este artículo es obra de *Miguel Angel Alvarez*

Fue publicado / actualizado en 14/03/2022

Disponible online en <https://desarrolloweb.com/articulos/operador-ternario-javascript>

## Operador `typeof` de Javascript para control de tipos



## Conoce el operador **typeof** de Javascript, el operador de control de tipo, que permite informar del tipo de una variable existente.

El listado de operadores que venimos ofreciendo para el [Manual de Javascript](#) se completa con el operador **typeof**, que veremos a continuación.

Hemos podido comprobar que, para determinados operadores, es importante el tipo de datos que están manejando, puesto que si los datos son de un tipo se realizarán operaciones distintas que si son de otro.

Por ejemplo, cuando utilizábamos el operador **+**, si se trataba de números los sumaba, pero si se trataba de cadenas de caracteres las concatenaba. Vemos pues que el tipo de los datos que estamos utilizando sí que importa y que tendremos que estar pendientes este detalle si queremos que nuestras operaciones se realicen tal como esperábamos.

Para comprobar el tipo de un dato se puede utilizar otro operador que está disponible a partir de javascript 1.1, el **operador typeof**, que devuelve una cadena de texto que describe el tipo del operador que estamos comprobando.

**Nota:** a lo largo de nuestra experiencia con Javascript veremos que muchas veces es más útil cambiar el tipo de dato de una variable antes de hacer una comprobación con **typeof** para ver si la podemos utilizar como operando. Existen diversas funciones para intentar cambiar el tipo de una variable, como por ejemplo [parseInt\(\)](#), que veremos más adelante en la [Segunda Parte del Manual de Javascript](#).

```
var booleano = true
var numerico = 22
var numerico_flotante = 13.56
var texto = "mi texto"
var fecha = new Date()
document.write("<br>El tipo de booleano es: " + typeof booleano)
document.write("<br>El tipo de numerico es: " + typeof numerico)
document.write("<br>El tipo de numerico_flotante es: " + typeof numerico_flotante)
document.write("<br>El tipo de texto es: " + typeof texto)
document.write("<br>El tipo de fecha es: " + typeof fecha)
```

Si ejecutamos este script obtendremos que en la página se escribirá el siguiente texto:

El tipo de booleano es: boolean

El tipo de numerico es: number

El tipo de numerico\_flotante es: number

El tipo de texto es: string

El tipo de fecha es: object

En este ejemplo podemos ver que ser imprimen en la página los distintos tipos de las variables.

Estos pueden ser los siguientes:



- **boolean**, para los datos booleanos. (True o false)
- **number**, para los numéricos.
- **string**, para las cadenas de caracteres.
- **object**, para los objetos.
- **function**, para las funciones.
- **undefined**, para variables declaradas a las que no se les ha asignado valores.

Queremos destacar tan sólo dos detalles más:

1. Los números, ya tengan o no parte decimal, son siempre del tipo de datos numérico.
2. Una de las variables es un poco más compleja, es la variable fecha que es un objeto de la clase Date(), que se utiliza para el manejo de fechas en los scripts. La veremos más adelante, así como los objetos.
3. Ten en cuenta también que las funciones en Javascript son tratadas como el

### Recomendaciones con typeof

En nuestra opinión no es muy adecuado preguntar constantemente el tipo de una variable, para hacer cosas distintas cuando tiene un tipo u otro. Lo normal es que tengas claro qué tipo de operación deseas realizar en un momento dado y que, si no estás seguro si te ha llegado la variable de un tipo o de otro, la conviertas con alguna función como parseInt() o parseFloat(), que veremos más adelante.

El caso más útil que ahora consigo recordar es preguntar si el tipo de una variable es undefined, lo que te puede dar la respuesta sobre si la variable ha sido inicializada o no.

```
let variable;
if(typeof variable == 'undefined') {
    console.log('La variable vale undefined, no tiene valor definido');
}
```

Otro caso que se suele usar bastante es a la hora de hacer debug, cuando necesitas encontrar errores en tu código, ante funcionamientos erráticos e inesperados. En esos casos es útil consultar el tipo de las variable, porque a veces en Javascript te traen sorpresas que hacen que tus programas no acaben de funcionar bien.

Con esto ya hemos terminado de ver la lista de operadores que podemos utilizar en Javascript y en el próximo artículo pasaremos a conocer las [estructuras de control](#).

Este artículo es obra de *Miguel Angel Alvarez*  
Fue publicado / actualizado en 26/09/2019  
Disponible online en <https://desarrolloweb.com/articulos/542.php>

# Estructuras de control en Javascript

Las estructuras de control nos permitirán controlar el flujo de nuestros programas. Por supuesto, también forman parte de los asuntos más básicos de Javascript y de cualquier lenguaje de programación, por lo que las veremos con detenimiento.

## Estructuras de control en Javascript

### **Introducción a las estructuras de control. Enumeramos las que tenemos disponibles en Javascript.**

Los scripts vistos hasta ahora en el [Manual de Javascript](#) han sido tremadamente sencillos y lineales: se iban ejecutando las sentencias simples una detrás de la otra desde el principio hasta el fin. Sin embargo, esto no tiene porque ser siempre así y de hecho, en la mayoría de los casos las cosas son bastante más complejas.

Si tenemos alguna experiencia en la programación sabremos que en los programas generalmente se necesitará hacer cosas distintas dependiendo del estado de nuestras variables o realizar un mismo proceso muchas veces sin escribir las mismas líneas de código una y otra vez. Para realizar cosas más complejas en nuestros scripts se utilizan las estructuras de control. Con ellas podemos realizar tomas de decisiones y bucles. En los siguientes capítulos vamos a conocer las distintas estructuras de control que existen en Javascript.

### Toma de decisiones

Nos sirven para realizar unas acciones u otras en función del estado de las variables. Es decir, tomar decisiones para ejecutar unas instrucciones u otras dependiendo de lo que esté ocurriendo en ese instante en nuestros programas.

Por ejemplo, dependiendo si el usuario que entra en nuestra página es mayor de edad o no lo es, podemos permitirle o no ver los contenidos de nuestra página.

Si edad es mayor que 18 entonces

    Te dejo ver el contenido para adultos

Si no

    Te mando fuera de la página

En Javascript podemos tomar decisiones utilizando dos enunciados distintos.

- [IF](#)
- [SWITCH](#)



## Bucles

Los bucles se utilizan para realizar ciertas acciones repetidamente. Son muy utilizados a todos los niveles en la programación. Con un bucle podemos por ejemplo imprimir en una página los números del 1 al 100 sin necesidad de escribir cien veces el la instrucción imprimir.

Desde el 1 hasta el 100

Imprimir el número actual

En javascript existen varios tipos de bucles, cada uno está indicado para un tipo de iteración distinto y son los siguientes:

- [FOR](#)
- [WHILE](#)
- [DO WHILE](#)

Como hemos señalado ya, las estructuras de control son muy importantes en Javascript y en cualquier lenguaje de programación. Es por ello que en los siguientes capítulos veremos cada una de estas estructuras detenidamente, describiendo su uso y ofreciendo algunos ejemplos.

Comenzaremos con la [estructura de control if](#).

Este artículo es obra de *Miguel Angel Alvarez*

Fue publicado / actualizado en *03/10/2001*

Disponible online en [\*https://desarrolloweb.com/articulos/543.php\*](https://desarrolloweb.com/articulos/543.php)

## Estructura IF en Javascript

**Vemos cómo trabajar con la estructura de control IF en Javascript.**

En el [Manual de Javascript](#) de DesarrolloWeb.com ya [empezamos a explicar lo que son las estructuras de control](#). En el presente artículo vamos a dedicarnos a mostrar cómo funciona la sentencia if, que es la estructura más habitual de las utilizadas para tomar decisiones en programas informáticos.

IF es una estructura de control utilizada para **tomar decisiones**. Es un condicional que sirve para realizar unas u otras operaciones en función de una expresión. Funciona de la siguiente manera, primero se evalúa una expresión, si da resultado positivo se realizan las acciones relacionadas con el caso positivo.

La sintaxis de la estructura IF es la siguiente.



**Nota:** Todas las estructuras de control se escriben en minúsculas en Javascript. Aunque algunas veces para destacar el nombre de la estructura la podemos escribir en el texto del manual con letras mayúsculas, en el código de nuestros scripts siempre tenemos que ponerlo en minúsculas. En caso contrario recibiremos un mensaje de error.

```
if (expresión) {  
    //acciones a realizar en caso positivo  
    //...  
}
```

Opcionalmente se pueden indicar acciones a realizar en caso de que la evaluación de la sentencia devuelva resultados negativos.

```
if (expresión) {  
    //acciones a realizar en caso positivo  
    //...  
} else {  
    //acciones a realizar en caso negativo  
    //...  
}
```

Fijémonos en varias cosas. Para empezar vemos como con unas llaves engloban las acciones que queremos realizar en caso de que se cumplan o no las expresiones. Estas llaves han de colocarse siempre, excepto en el caso de que sólo haya una instrucción como acciones a realizar, que son opcionales.

**Nota:** Aunque las llaves para englobar las sentencias a ejecutar tanto en el caso positivo como negativo sean opcionales cuando queremos ejecutar una única sentencia, la recomendación es colocarlas siempre, porque obtendremos así un código fuente más claro. Por ejemplo:

```
if (llueve)  
    alert("Cae agua");
```

Sería exactamente igual que este código:

```
if (llueve){  
    alert("Cae agua");  
}
```

O incluso, igual a este otro:

```
if (llueve) alert("Cae agua");
```

Sin embargo, cuando utilizamos las llaves, el código queda bastante más claro, porque se puede apreciar en un rápido vistazo qué instrucciones están dependiendo del caso positivo del if. Esto es un detalle que ahora quizás no tenga mucha importancia, pero que se agradecerá cuando el



programa sea más complejo o cuando varios programadores se encarguen de tocar un mismo código.

Otro detalle que salta a la vista es el sangrado (margen) que hemos colocado en cada uno de los bloques de instrucciones a ejecutar en los casos positivos y negativos. Este sangrado es totalmente opcional, sólo lo hemos hecho así para que la estructura IF se comprenda de una manera más visual. Los saltos de línea tampoco son necesarios y se han colocado también para que se vea mejor la estructura. Perfectamente podríamos colocar toda la instrucción IF en la misma línea de código, pero eso no ayudará a que las cosas estén claras.

**Nota:** Nosotros, así como lo haría cualquier persona con cierta experiencia en el área de programación, aconsejamos que se utilicen los sangrados y saltos de línea necesarios para que las instrucciones se puedan entender mejor. Quizás el día que realizas un código tengas claro qué has hecho y por qué es así, pero dentro de un mes, cuando tengas que releer ese código, quizás te acuerdes menos de lo que hiciste en tus scripts y agradecerás que tengan un formato amigable para que se puedan leer con facilidad por las personas. Si trabajas en equipo estas recomendaciones serán todavía más importantes, puesto que todavía es más complicado leer código fuente que han realizado otras personas.

Veamos algún ejemplo de condicionales IF.

```
if (dia == "lunes")
    document.write ("Que tengas un feliz comienzo de semana")
```

Si es lunes nos deseará una feliz semana. No hará nada en caso contrario. Como en este ejemplo sólo indicamos una instrucción para el caso positivo, no hará falta utilizar las llaves (aunque sí sería recomendable haberlas puesto). Fíjate también en el operador condicional que consta de dos signos igual.

Vamos a ver ahora otro ejemplo, un poco más largo.

```
if (credito >= precio) {
    document.write("has comprado el artículo " + nuevoArtículo) //enseño compra
    carrito += nuevoArtículo //introduzco el artículo en el carrito de la compra
    credito -= precio //disminuyo el crédito según el precio del artículo
} else {
    document.write("se te ha acabado el crédito") //informo que te falta dinero
    window.location = "carritodelacompra.html" //voy a la página del carrito
}
```

Este ejemplo es un poco más complejo, y también un poco ficticio. Lo que hago es comprobar si tengo crédito para realizar una supuesta compra. Para ello miro si el crédito es mayor o igual que el precio del artículo, si es así informo de la compra, introduzco el artículo en el carrito y resto el



precio al crédito acumulado. Si el precio del artículo es superior al dinero disponible informo de la situación y mando al navegador a la página donde se muestra su carrito de la compra.

## Expresiones condicionales

La expresión a evaluar se coloca siempre entre paréntesis y está compuesta por variables que se combinan entre si mediante operadores condicionales. Recordamos que los operadores condicionales relacionaban dos variables y devolvían siempre un resultado booleano. Por ejemplo un operador condicional es el operador "es igual" (==), que devuelve true en caso de que los dos operandos sean iguales o false en caso de que sean distintos.

```
if (edad > 18)
    document.write("puedes ver esta página para adultos")
```

En este ejemplo utilizamos en operador condicional "es mayor" (>). En este caso, devuelve true si la variable edad es mayor que 18, con lo que se ejecutaría la línea siguiente que nos informa de que se puede ver el contenido para adultos.

Las expresiones condicionales se pueden combinar con las expresiones lógicas para crear expresiones más complejas. Recordamos que las expresiones lógicas son las que tienen como operandos a los booleanos y que devuelvan otro valor booleano. Son los operadores negación lógica, Y lógico y O lógico.

```
if (bateria < 0.5 && redElectrica == 0)
    document.write("tu ordenador portatil se va a apagar en segundos")
```

Lo que hacemos es comprobar si la batería de nuestro supuesto ordenador es menor que 0.5 (está casi acabada) y también comprobamos si el ordenador no tiene red eléctrica (está desenchufado). Luego, el operador lógico los relaciona con un Y, de modo que si está casi sin batería Y sin red eléctrica, informo que el ordenador se va a apagar.

**Nota:** La lista de operadores que se pueden utilizar con las estructuras IF se pueden ver en el

[capítulo de operadores condicionales y operadores lógicos](#).

La estructura if es de las más utilizadas en lenguajes de programación, para tomar decisiones en función de la evaluación de una sentencia. En el artículo anterior del [Manual de Javascript](#) ya [comenzamos a explicar la estructura if](#) y ahora vamos a ver algunos usos un poquito más avanzados.

## Sentencias IF anidadas

Para hacer estructuras condicionales más complejas podemos anidar sentencias IF, es decir, colocar estructuras IF dentro de otras estructuras IF. Con un solo IF podemos evaluar y realizar una acción u otra según dos posibilidades, pero si tenemos más posibilidades que evaluar debemos anidar IFs para crear el flujo de código necesario para decidir correctamente.



Por ejemplo, si deseo comprobar si un número es mayor menor o igual que otro, tengo que evaluar tres posibilidades distintas. Primero puedo comprobar si los dos números son iguales, si lo son, ya he resuelto el problema, pero si no son iguales todavía tendré que ver cuál de los dos es mayor. Veamos este ejemplo en código Javascript.

```
var numero1=23
var numero2=63
if (numero1 == numero2){
    document.write("Los dos números son iguales")
} else{
    if (numero1 > numero2) {
        document.write("El primer número es mayor que el segundo")
    } else{
        document.write("El primer número es menor que el segundo")
    }
}
```

El flujo del programa es como comentábamos antes, primero se evalúa si los dos números son iguales. En caso positivo se muestra un mensaje informando de ello. En caso contrario ya sabemos que son distintos, pero aun debemos averiguar cuál de los dos es mayor. Para eso se hace otra comparación para saber si el primero es mayor que el segundo. Si esta comparación da resultados positivos mostramos un mensaje diciendo que el primero es mayor que el segundo, en caso contrario indicaremos que el primero es menor que el segundo.

Volvemos a remarcar que las llaves son en este caso opcionales, pues sólo se ejecuta una sentencia para cada caso. Además, los saltos de línea y los sangrados también opcionales en todo caso y nos sirven sólo para ver el código de una manera más ordenada. Mantener el código bien estructurado y escrito de una manera comprensible es muy importante, ya que nos hará la vida más agradable a la hora de programar y más adelante cuando tengamos que revisar los programas.

**Nota:** En este manual utilizaré una notación como la que has podido ver en las líneas anteriores. Además mantendré esa notación en todo momento. Esto sin lugar a dudas hará que los códigos con ejemplos sean comprensibles más rápidamente, si no lo hicieramos así sería un verdadero incordio leerlos. Esta misma receta es aplicable a los códigos que has de crear tú y el principal beneficiado serás tú mismo y los compañeros que lleguen a leer tu código.

## Operador IF

Hay un operador que no hemos visto todavía y es una forma más esquemática de realizar algunos IF sencillos. Proviene del lenguaje C, donde se escriben muy pocas líneas de código y donde cuanto menos escribamos más elegantes seremos. Este operador es un claro ejemplo de ahorro de líneas y caracteres al escribir los scripts. Lo veremos rápidamente, pues la única razón por la que lo incluyo es para que sepas que existe y si lo encuentras en alguna ocasión por ahí sepas identificarlo y cómo funciona.

Un ejemplo de uso del operador IF se puede ver a continuación.



```
Variable = (condición) ? valor1 : valor2
```

Este ejemplo no sólo realiza una comparación de valores, además asigna un valor a una variable. Lo que hace es evaluar la condición (colocada entre paréntesis) y si es positiva asigna el valor1 a la variable y en caso contrario le asigna el valor2. Veamos un ejemplo:

```
momento = (hora_actual < 12) ? "Antes del mediodía" : "Después del mediodía"
```

Este ejemplo mira si la hora actual es menor que 12. Si es así, es que ahora es antes del mediodía, así que asigna "Antes del mediodía" a la variable momento. Si la hora es mayor o igual a 12 es que ya es después de mediodía, con lo que se asigna el texto "Después del mediodía" a la variable momento.

Para ampliar la información recomendamos ver también el [Videotutorial de Javascript](#), en el [vídeo donde tratamos la estructura IF](#).

Este artículo es obra de *Miguel Angel Alvarez*

Fue publicado / actualizado en *03/10/2001*

Disponible online en <https://desarrolloweb.com/articulos/544.php>

## Estructura SWITCH de Javascript

**La estructura de control switch de Javascript es utilizada para tomar decisiones en función de distintos estados o valores de una variable.**

Las [estructuras de control](#) son la manera con la que se puede dominar el flujo de los programas, para hacer cosas distintas en función de los estados de las variables. En el [Manual de Javascript](#) ya empezamos a ver las estructuras de control y ahora le ha tocado el turno a SWITCH, una estructura un poco más compleja que permite hacer múltiples operaciones dependiendo del estado de una variable.

En este artículo veremos que switch nos sirve para tomar decisiones en función de distintos estados de las variables. Esta expresión se utiliza cuando tenemos múltiples posibilidades como resultado de la evaluación de una sentencia.

La estructura SWITCH se incorporó a partir de la versión 1.2 de Javascript (Netscape 4 e Internet Explorer 4). Su sintaxis es la siguiente.

```
switch (expresión) {
    case valor1:
        Sentencias a ejecutar si la expresión tiene como valor a valor1
        break
    case valor2:
        Sentencias a ejecutar si la expresión tiene como valor a valor2
        break
```



```
case valor3:  
    Sentencias a ejecutar si la expresión tiene como valor a valor3  
    break  
default:  
    Sentencias a ejecutar si el valor no es ninguno de los anteriores  
}
```

La expresión se evalúa, si vale valor1 se ejecutan las sentencias relacionadas con ese caso. Si la expresión vale valor2 se ejecutan las instrucciones relacionadas con ese valor y así sucesivamente, por tantas opciones como deseemos. Finalmente, para todos los casos no contemplados anteriormente se ejecuta el caso por defecto.

La palabra break es opcional, pero si no la ponemos a partir de que se encuentre coincidencia con un valor se ejecutarán todas las sentencias relacionadas con este y todas las siguientes. Es decir, si en nuestro esquema anterior no hubiese ningún break y la expresión valiese valor1, se ejecutarían las sentencias relacionadas con valor1 y también las relacionadas con valor2, valor3 y default.

También es opcional la opción default u opción por defecto.

Veamos un ejemplo de uso de esta estructura. Supongamos que queremos indicar que día de la semana es. Si el día es 1 (lunes) sacar un mensaje indicándolo, si el día es 2 (martes) debemos sacar un mensaje distinto y así sucesivamente para cada día de la semana, menos en el 6 (sábado) y 7 (domingo) que queremos mostrar el mensaje "es fin de semana". Para días mayores que 7 indicaremos que ese día no existe.

```
switch (dia_de_la_semana) {  
    case 1:  
        document.write("Es Lunes")  
        break  
    case 2:  
        document.write("Es Martes")  
        break  
    case 3:  
        document.write("Es Miércoles")  
        break  
    case 4:  
        document.write("Es Jueves")  
        break  
    case 5:  
        document.write("Es viernes")  
        break  
    case 6:  
    case 7:  
        document.write("Es fin de semana")  
        break  
    default:  
        document.write("Ese día no existe")  
}
```

El ejemplo es relativamente sencillo, solamente puede tener una pequeña dificultad, consistente en interpretar lo que pasa en el caso 6 y 7, que habíamos dicho que teníamos que mostrar el mismo mensaje. En el caso 6 en realidad no indicamos ninguna instrucción, pero como tampoco colocamos un break se ejecutará la sentencia o sentencias del caso siguiente, que corresponden con la sentencia indicada en el caso 7 que es el mensaje que informa que es fin de semana. Si el caso es 7 simplemente se indica que es fin de semana, tal como se pretendía.



**Nota:** Además contamos con un [videotutorial sobre el SWITCH en Javascript](#) que os puede ser de mucha ayuda para entenderlo todo mucho mejor.

En el siguiente artículo comenzaremos a explorar las estructuras de control para hacer bucles o repeticiones en Javascript, comenzando por el [bucle for](#).

Este artículo es obra de *Miguel Angel Alvarez*

Fue publicado / actualizado en *03/10/2001*

Disponible online en <https://desarrolloweb.com/articulos/546.php>

## Bucle FOR en Javascript

### Descripción y ejemplos de funcionamiento del bucle FOR.

Comenzamos con este artículo del [Manual de Javascript](#) a explorar las estructuras de control para producir bucles o repeticiones, comenzando por el bucle for, no por ser el más simple, pero sí el más utilizado en la programación.

**El bucle FOR se utiliza para repetir una o más instrucciones un determinado número de veces.** De entre todos los bucles, el FOR se suele utilizar cuando sabemos seguro el número de veces que queremos que se ejecute. La sintaxis del bucle for se muestra a continuación.

```
for (inicialización; condición; actualización) {
    //sentencias a ejecutar en cada iteración
}
```

El bucle FOR tiene tres partes incluidas entre los paréntesis, que nos sirven para definir cómo deseamos que se realicen las repeticiones. La primera parte es la inicialización, que se ejecuta solamente al comenzar la primera iteración del bucle. En esta parte se suele colocar la variable que utilizaremos para llevar la cuenta de las veces que se ejecuta el bucle.

La segunda parte es la condición, que se evaluará cada vez que comience una iteración del bucle. Contiene una expresión para decidir cuándo se ha de detener el bucle, o mejor dicho, la condición que se debe cumplir para que continúe la ejecución del bucle.

Por último tenemos la actualización, que sirve para indicar los cambios que queramos ejecutar en las variables cada vez que termina la iteración del bucle, antes de comprobar si se debe seguir ejecutando.



Después del for se colocan las sentencias que queremos que se ejecuten en cada iteración, acotadas entre llaves.

Un ejemplo de utilización de este bucle lo podemos ver a continuación, donde se imprimirán los números del 0 al 10.

```
var i
for (i=0;i<=10;i++) {
    document.write(i)
    document.write("<br>")
}
```

En este caso se inicializa la variable i a 0. Como condición para realizar una iteración, se tiene que cumplir que la variable i sea menor o igual que 10. Como actualización se incrementará en 1 la variable i.

Como se puede comprobar, **este bucle es muy potente, ya que en una sola línea podemos indicar muchas cosas distintas y muy variadas**, lo que permite una rápida configuración del bucle y una versatilidad enorme.

Por ejemplo si queremos escribir los número del 1 al 1.000 de dos en dos se escribirá el siguiente bucle.

```
for (i=1;i<=1000;i+=2)
    document.write(i)
```

Si nos fijamos, en cada iteración actualizamos el valor de i incrementándolo en 2 unidades.

**Nota:** Otro detalle, no utilizamos las llaves englobando las instrucciones del bucle FOR porque sólo tiene una sentencia y en este caso no es obligado, tal como pasaba con las instrucciones del IF.

Si queremos contar descendentemente del 343 al 10 utilizaríamos este bucle.

```
for (i=343;i>=10;i--)
    document.write(i)
```

En este caso decrementamos en una unidad la variable i en cada iteración, comenzando en el valor 343 y siempre que la variable tenga un valor mayor o igual que 10.

### Ejercicio de ejemplo del bucle for

Vamos a hacer una pausa para asimilar el bucle for con un ejercicio que no encierra ninguna dificultad si hemos entendido el funcionamiento del bucle.



Se trata de hacer un bucle que escriba en una página web los encabezamientos desde <H1> hasta <H6> con un texto que ponga "Encabezado de nivel x".

Lo que deseamos escribir en una página web mediante Javascript es lo siguiente:

```
<H1>Encabezado de nivel 1</H1>
<H2>Encabezado de nivel 2</H2>
<H3>Encabezado de nivel 3</H3>
<H4>Encabezado de nivel 4</H4>
<H5>Encabezado de nivel 5</H5>
<H6>Encabezado de nivel 6</H6>
```

Para ello tenemos que hacer un bucle que empiece en 1 y termine en 6 y en cada iteración escribiremos el encabezado que toca.

```
for (i=1;i<=6;i++) {
    document.write("<H" + i + ">Encabezado de nivel " + i + "</H" + i + ">")
}
```

Este script se puede [ver en funcionamiento aquí](#).

Ahora que ya conocemos el bucle for, estamos en condiciones de aprender a manejar otras estructuras de control para realizar repeticiones, como los [bucles while y do...while](#).

Este artículo es obra de *Miguel Angel Alvarez*  
Fue publicado / actualizado en 18/10/2001  
Disponible online en <https://desarrolloweb.com/articulos/bucle-for-javascript.html>

## Bucles WHILE y DO WHILE

**Descripción y diferentes usos de los dos tipos de bucles WHILE que se encuentran disponibles en Javascript, con algunos ejemplos prácticos.**

Estamos tratando acerca de las distintas estructuras de control que existen en el lenguaje Javascript y en concreto viendo los distintos tipos de bucles que podemos implementar en este lenguaje de programación. En artículos anteriores del [Manual de Javascript](#) vimos ya el primero de los bucles que debemos conocer, el [bucle for](#) y ahora vamos a tratar sobre los otros dos tipos de estructuras de control para hacer repeticiones. Así pues, veamos ahora los dos tipos de bucles WHILE que podemos utilizar en Javascript y los usos de cada uno.

### Bucle WHILE

Estos bucles se utilizan cuando queremos repetir la ejecución de unas sentencias un número indefinido de veces, siempre que se cumpla una condición. Se más sencillo de comprender que el



bucle FOR, pues no incorpora en la misma línea la inicialización de las variables su condición para seguir ejecutándose y su actualización. Sólo se indica, como veremos a continuación, la condición que se tiene que cumplir para que se realice una iteración.

```
while (condición){  
    //sentencias a ejecutar  
}
```

Un ejemplo de código donde se utiliza este bucle se puede ver a continuación.

```
var color = ""  
while (color != "rojo"){  
    color = prompt("dame un color (escribe rojo para salir)", "")  
}
```

Este es un ejemplo de lo más sencillo que se puede hacer con un bucle while. Lo que hace es pedir que el usuario introduzca un color y lo hace repetidas veces, mientras que el color introducido no sea rojo. Para ejecutar un bucle como este primero tenemos que inicializar la variable que vamos utilizar en la condición de iteración del bucle. Con la variable inicializada podemos escribir el bucle, que comprobará para ejecutarse que la variable color sea distinto de "rojo". En cada iteración del bucle se pide un nuevo color al usuario para actualizar la variable color y se termina la iteración, con lo que retornamos al principio del bucle, donde tenemos que volver a evaluar si lo que hay en la variable color es "rojo" y así sucesivamente mientras que no se haya introducido como color el texto "rojo".

**Nota:** Hemos utilizado en este ejemplo la función prompt de Javascript, que no hemos visto todavía en este manual. Esta función sirve para que mostrar una caja de diálogo donde el usuario debe escribir un texto. Esta función pertenece al objeto window de Javascript y la comentamos en el artículo [Métodos de window en Javascript](#).

## Bucle DO...WHILE

El bucle do...while es la última de las estructuras para implementar repeticiones de las que dispone en Javascript y es una variación del bucle while visto anteriormente. Se utiliza generalmente cuando no sabemos cuantas veces se habrá de ejecutar el bucle, igual que el bucle WHILE, con la diferencia de que sabemos seguro que el bucle por lo menos se ejecutará una vez.

Este tipo de bucle se introdujo en Javascript 1.2, por lo que no todos los navegadores los soportan, sólo los de versión 4 o superior. En cualquier caso, cualquier código que quieras escribir con DO...WHILE se puede escribir también utilizando un bucle WHILE, con lo que en navegadores antiguos deberás traducir tu bucle DO...WHILE por un bucle WHILE.

La sintaxis es la siguiente:



```
do {  
    //sentencias del bucle  
} while (condición)
```

El bucle se ejecuta siempre una vez y al final se evalúa la condición para decir si se ejecuta otra vez el bucle o se termina su ejecución.

Veamos el ejemplo que escribimos para un bucle WHILE en este otro tipo de bucle.

```
var color  
do {  
    color = prompt("dame un color (escribe rojo para salir)", "")  
} while (color != "rojo")
```

Este ejemplo funciona exactamente igual que el anterior, excepto que no tuvimos que inicializar la variable color antes de introducirnos en el bucle. Pide un color mientras que el color introducido es distinto que "rojo".

### Ejemplo de uso de los bucles while

Vamos a ver a continuación un ejemplo más práctico sobre cómo trabajar con un bucle WHILE. Como resulta muy difícil hacer ejemplos prácticos con lo poco que sabemos sobre Javascript, vamos a adelantar una instrucción que aun no conocemos.

En este ejemplo vamos a declarar una variable e inicializarla a 0. Luego iremos sumando a esa variable un número aleatorio del 1 al 100 hasta que sumemos 1.000 o más, imprimiendo el valor de la variable suma después de cada operación. Será necesario utilizar el bucle WHILE porque no sabemos exactamente el número de iteraciones que tendremos que realizar (dependerá de los valores aleatorios que se vayan obteniendo).

```
var suma = 0  
while (suma < 1000){  
    suma += parseInt(Math.random() * 100)  
    document.write (suma + "<br>")  
}
```

Suponemos que por lo que respecta al bucle WHILE no habrá problemas, pero donde si que puede haberlos es en la sentencia utilizada para tomar un número aleatorio. Sin embargo, no es necesario explicar aquí la sentencia porque lo tenemos planeado hacer más adelante. De todos modos, si lo deseas, puedes ver este artículo que habla sobre [números aleatorios en Javascript](#).

Podemos [ver una página con el ejemplo en funcionamiento](#).

Con esto ya hemos conocido todos los tipos de bucles que existen en Javascript, no obstante aun vamos a dedicar un artículo para explicar las [sentencias break y continue](#) que nos sirven para alterar el funcionamiento normal de los bucles en dos sentidos.



Este artículo es obra de *Miguel Angel Alvarez*

Fue publicado / actualizado en 18/10/2001

Disponible online en <https://desarrolloweb.com/articulos/567.php>

## Break y continue

**Dos instrucciones que aumentan el control sobre los bucles en Javascript. Sirven para parar y continuar con la siguiente iteración del bucle respectivamente.**

Javascript tiene diferentes estructuras de control para implementar bucles, como FOR, WHILE y DO...WHILE, que ya hemos podido explicar en capítulos anteriores del [Manual de Javascript](#). Como hemos podido comprobar, con estos bucles podemos abarcar gran cantidad de necesidades, pero quizás con el tiempo encuentres que te faltan algunas posibilidades de control de las repeticiones de los bucles.

Imagina por ejemplo que estas haciendo un bucle muy largo para encontrar algo en cientos o miles de sitios. Pero ponte en el caso que durante las primeras iteraciones encuentres ese valor que buscabas. Entonces no tendría sentido continuar con el resto del bucle para buscar ese elemento, pues ya lo habías encontrado. En estas situaciones nos conviene saber para el bucle cancelar el resto de iteraciones. Obviamente, ésto es solo un ejemplo de cómo podríamos vernos en la necesidad de controlar un poco más el bucle. En la vida real como programador encontrarás muchas otras ocasiones en las que te interesará hacer esto u otras cosas con ellos.

Así pues, existen dos instrucciones que se pueden usar en de las distintas estructuras de control y principalmente en los bucles, que te servirán para controlar dos tipos de situaciones. **Son las instrucciones break y continue.:**

- **break:** Significa detener la ejecución de un bucle y salirse de él.
- **continue:** Sirve para detener la iteración actual y volver al principio del bucle para realizar otra iteración, si corresponde.

### Break

Se detiene un bucle utilizando la palabra break. Detener un bucle significa salirse de él y dejarlo todo como está para continuar con el flujo del programa inmediatamente después del bucle.

```
for (i=0;i<10;i++){  
    document.write (i)  
    escribe = prompt("dime si continuo preguntando...", "si")  
    if (escribe == "no")  
        break  
}
```



Este ejemplo escribe los números del 0 al 9 y en cada iteración del bucle pregunta al usuario si desea continuar. Si el usuario dice cualquier cosa continua, excepto cuando dice "no", situación en la cual se sale del bucle y deja la cuenta por donde se había quedado.

## Continue

Sirve para volver al principio del bucle en cualquier momento, sin ejecutar las líneas que haya por debajo de la palabra continue.

```
var i=0
while (i<7){
    incrementar = prompt("La cuenta está en " + i + ", dime si incremento", "si")
    if (incrementar == "no")
        continue
    i++
}
```

Este ejemplo, en condiciones normales contaría hasta desde i=0 hasta i=7, pero cada vez que se ejecuta el bucle pregunta al usuario si desea incrementar la variable o no. Si introduce "no" se ejecuta la sentencia continue, con lo que se vuelve al principio del bucle sin llegar a incrementar en 1 la variable i, ya que se ignorarían las sentencias que hayan por debajo del continue.

## Ejemplo adicional de la sentencia break

Un ejemplo más práctico sobre estas instrucciones se puede ver a continuación. Se trata de un bucle FOR planeado para llegar hasta 1.000 pero que lo vamos a parar con break cuando lleguemos a 333.

```
for (i=0;i<=1000;i++){
    document.write(i + "<br>")
    if (i==333)
        break;
}
```

Podemos [ver una página con el ejemplo en funcionamiento](#).

Con la descripción de las sentencias break y continue hemos podido abarcar todo lo que se debe saber sobre la creación de bucles en Javascript. Ahora bien, en el siguiente artículo todavía vamos a seguir en el tema de las estructuras de control, porque queremos ofrecer un ejemplo un poco más avanzado donde [aprenderemos a anidar bucles](#).

Este artículo es obra de *Miguel Angel Alvarez*  
Fue publicado / actualizado en *18/10/2001*  
Disponible online en <https://desarrolloweb.com/articulos/568.php>

## Bucles anidados en Javascript



**Qué es un bucle anidado. Qué tipo de operaciones se pueden hacer con los bucles anidados. Ejemplos de anidación de bucles en Javascript.**



En el [Manual de Javascript](#) hemos recorrido ya diversos artículos para hablar de bucles. En este momento no debería haber ningún problema para poder crear los distintos tipos de bucles sin problemas, no obstante, queremos dedicar un artículo completo a tratar acerca de uno de los **usos más habituales de los bucles**, que podremos encontrar cuando estemos haciendo programas más complejos: la **anidación de bucles**.

**Anidar un bucle consiste en meter ese bucle dentro de otro.** La anidación de bucles es necesaria para hacer determinados procesamientos un poco más complejos que los que hemos visto en los ejemplos anteriores. Si en vuestra experiencia como programadores los habéis anidado un bucle todavía, tener certeza que más tarde o temprano os encontraréis con esa necesidad.

### Operaciones típicas para las que podríamos necesitar un bucle anidado

Los bucles anidados se necesitan para hacer muchas cosas. Algunos algoritmos que requerirían bucles anidados serían:

- Recorrer una estructura de datos compleja
- Recorrer todos los elementos de un array multidimensional
- Realizar ordenación de elementos de un array

Son algunas ideas así de manera genérica, pero la naturaleza de tu algoritmo podría requerir la realización de bucles anidados cuando consistan en hacer repeticiones dentro de repeticiones. Por ejemplo, si quieres sacar todas las tablas de multiplicar del uno al diez, necesitarías un bucle para cada tabla de multiplicar, y otro bucle para iterar del 1 al 10.

### Ejemplo de un bucle anidado

Un bucle anidado tiene una estructura como la que sigue. Vamos a tratar de explicarlo a la vista de estas líneas:

```
for (let i=0; i < 10; i++){
    for (let j=0; j < 10; j++) {
        document.write(i + "-" + j)
    }
}
```



La ejecución funcionará de la siguiente manera. Para empezar se inicializa el primer bucle, con lo que la variable i valdrá 0 y a continuación se inicializa el segundo bucle, con lo que la variable j valdrá también 0. En cada iteración se imprime el valor de la variable i, un guión ("") y el valor de la variable j, como las dos variables valen 0, lo primero que se imprimirá será el texto "0-0" en la página web.

Debido al flujo del programa en esquemas de anidación como el que hemos visto, el bucle que está anidado (más hacia dentro) es el que más veces se ejecuta. En este ejemplo, para cada iteración del bucle más externo el bucle anidado se ejecutará por completo una vez, es decir, hará sus 10 iteraciones. En la página web se escribirían estos valores, en la primera iteración del bucle externo y desde el principio:

0-0  
0-1  
0-2  
0-3  
0-4  
0-5  
0-6  
0-7  
0-8  
0-9

Para cada iteración del bucle externo se ejecutarán las 10 iteraciones del bucle interno o anidado. Hemos visto la primera iteración, ahora vamos a ver las siguientes iteraciones del bucle externo. En cada una acumula una unidad en la variable i, con lo que saldrían estos valores.

1-0  
1-1  
1-2  
1-3  
1-4  
1-5  
1-6  
1-7  
1-8  
1-9

Y luego estos:

2-0  
2-1  
2-2  
2-3  
2-4



2-5  
2-6  
2-7  
2-8  
2-9

Así hasta que se terminen los dos bucles, que sería cuando se alcanzase el valor 9-9.

Veamos un ejemplo muy parecido al anterior, aunque un poco más útil. Se trata de imprimir en la página las todas las tablas de multiplicar. Del 1 al 9, es decir, la tabla del 1, la del 2, del 3...

```
for (i=1;i<10;i++){
    document.write("<br><b>La tabla del " + i + ":"</b><br>")
    for (j=1;j<10;j++) {
        document.write(i + " x " + j + ": ")
        document.write(i*j)
        document.write("<br>")
    }
}
```

Con el primer bucle controlamos la tabla actual y con el segundo bucle la desarrollamos. En el primer bucle escribimos una cabecera, en negrita, indicando la tabla que estamos escribiendo, primero la del 1 y luego las demás en orden ascendente hasta el 9. Con el segundo bucle escribo cada uno de los valores de cada tabla.

### Ejemplo de recorrido de un array multidimensional en Javascript

Creo que sería una buena práctica mostrar un ejemplo de bucle anidado para recorrer un array multidimensional en Javascript.

Un array multidimensional es aquel que tiene otros arrays en cada una de sus casillas. Por ejemplo, podríamos definir un array multidimensional de esta manera:

```
let arrayMultidimensional = [
    [4, 6, 7],
    [6, 1, 6],
    [81, 0],
    [3, 9, 64, 7],
];
```

Ahora podemos ver un bucle sencillo para hacer el recorrido con el bucle anidado:

```
for(let i = 0; i < arrayMultidimensional.length; i++) {
    for(let j = 0; j < arrayMultidimensional[i].length; j++) {
        console.log(arrayMultidimensional[i][j]);
    }
}
```

Como podemos ver, se realizan dos bucles. El bucle más general se encarga de recorrer las casillas del `arrayMultidimensional` y luego, como cada una de sus casillas es a la vez otro array, necesitamos un bucle anidado para recorrerlas también.



La salida que mostrará este programa está en la consola, por lo que tendrás que abrir las herramientas de desarrollo y activar la consola de Javascript para poder encontrarla.

Lo que verás en la consola es esta secuencia de números:

```
4  
6  
7  
6  
1  
6  
81  
0  
3  
9  
64  
7
```

Ahora, por su quieres verlo y estudiarlo con calma, te pongo aquí un código que permite mostrar este array multidimensional de una manera más legible, mostrando su contenido tal como está organizado, en filas.

```
for(let i = 0; i < arrayMultidimensional.length; i++) {  
    let numeros = '[';  
    for(let j = 0; j < arrayMultidimensional[i].length; j++) {  
        numeros += arrayMultidimensional[i][j];  
        if(arrayMultidimensional[i].length-1 > j) {  
            numeros += ', ';  
        }  
    }  
    numeros += "]";  
    console.log(numeros);  
}
```

Espero que te sirva de ejemplo un poco más elaborado de recorridos a arrays multidimensionales.

### Ejemplo de bucle anidado en Javascript con do while

En todos los ejemplos anteriores hemos realizado recorridos con bucles anidados donde se han usado bucles for. Eso no es una norma!! puedes hacer bucles anidados con todo tipo de bucles, for, while, do while.

Para que lo veas vamos a realizar ahora una práctica en la que solicitamos al usuario un número entero y le mostramos todos los números enteros impares que sean menores al número entero proporcionado.

```
do {  
    let numero = prompt('Dame un número entero', '');  
    let entero = parseInt(numero);  
    if(entero != numero) {  
        alert('No has escrito un número entero');  
    } else {  
        let imparesMenores = [];  
        for(let i = entero -1; i > 0; i--) {  
            if(i % 2 != 0) {  
                imparesMenores.push(i);  
            }  
        }  
        console.log(imparesMenores);  
    }  
}
```



```
        imparesMenores.push(i);
    }
}
alert('Los impares menores que ' + entero + ' son ' + imparesMenores.join(', '));
}
} while(confirm('Quieres continuar?'));
```

En el bucle do while realizamos iteraciones hasta que el usuario nos diga que no quiere continuar.

Dentro del bucle while:

- Se pide un número
- Se comprueba que sea un entero con la función parseInt() de Javascript.
- Si no era un entero, simplemente se muestra un mensaje al usuario y no se hace nada más
- Si era un entero, entonces hacemos un bucle anidado en el que se obtienen todos los impares menores, que se van metiendo en un array.
- Por último se muestran los impares menores en una caja de diálogo

La función `parseInt()` simplemente convierte cualquier cosa que le pases en un entero. Si no lo puede convertir te pasa una cosa que se llama `NaN` (not a number). Se usa en muchos ejemplos. Si no la conoces anteriormente puedes verla en el artículo de [funciones integradas en Javascript](#).

**Nota:** Veremos más cosas con bucles anidados en capítulos posteriores, aunque si queremos adelantarnos un poco para ver un nuevo ejemplo que afiance estos conocimientos podemos ir viendo un [ejemplo en el Taller de Javascript sobre bucles anidados](#), donde se construye la tabla con todos los colores puros en definiciones de 256 colores.

Con este artículo más bien práctico sobre anidación de bucles, terminamos el tema de las estructuras de control. Ahora pasaremos a otra sección de este [Manual de Javascript](#), en la que [trataremos sobre las funciones](#).

Este artículo es obra de *Miguel Angel Alvarez*

Fue publicado / actualizado en *23/03/2023*

Disponible online en <https://desarrolloweb.com/articulos/619.php>



# Funciones en Javascript

Las funciones nos permitirán hacer programas y scripts más optimizados y de fácil mantenimiento. También son básicas en cualquier lenguaje de programación y les dedicaremos varios artículos.

## Funciones en Javascript

**Comenzamos con las funciones en Javascript. Definimos el concepto de función y aprendemos a crearlas y a llamarlas.**

Seguimos trabajando y ampliando nuestros conocimientos sobre Javascript. Con lo visto hasta ahora en el [Manual de Javascript](#) ya tenemos una cierta soltura para trabajar en este interesante lenguaje de programación. Pero todavía nos queda mucho por delante.

Ahora vamos a ver un tema muy importante, sobretodo para los que no han programado nunca y con Javascript están dando sus primeros pasos en el mundo de la programación ya que veremos un concepto nuevo, el de función, y los usos que tiene. Para los que ya conozcan el concepto de función también será un capítulo útil, pues también veremos la sintaxis y funcionamiento de las funciones en Javascript.

### Qué es una función

A la hora de hacer un programa ligeramente grande existen determinados procesos que se pueden concebir de forma independiente, y que son más sencillos de resolver que el problema entero.

Además, estos suelen ser realizados repetidas veces a lo largo de la ejecución del programa. Estos procesos se pueden agrupar en una función, definida para que no tengamos que repetir una y otra vez ese código en nuestros scripts, sino que simplemente llamamos a la función y ella se encarga de hacer todo lo que debe.

Así que podemos ver una función como una serie de instrucciones que englobamos dentro de un mismo proceso. Este proceso se podrá luego ejecutar desde cualquier otro sitio con solo llamarlo. Por ejemplo, en una página web puede haber una función para cambiar el color del fondo y desde cualquier punto de la página podríamos llamarla para que nos cambie el color cuando lo deseemos.

**Nota:** Si queremos, podemos ampliar esta descripción de las funciones en el artículo [Concepto de función](#).



Las funciones se utilizan constantemente, no sólo las que escribes tú, sino también las que ya están definidas en el sistema, pues todos los lenguajes de programación suelen tener un montón de funciones para realizar procesos habituales, como por ejemplo obtener la hora, imprimir un mensaje en la pantalla o convertir variables de un tipo a otro. Ya hemos visto alguna función en nuestros sencillos ejemplos anteriores. Por ejemplo, cuando hacíamos un `document.write()` en realidad estábamos llamando a la función `write()` asociada al documento de la página, que escribe un texto en la página.

En los capítulos de funciones vamos primero a ver cómo realizar nuestras propias funciones y cómo llamarlas luego. A lo largo del manual veremos muchas de las funciones definidas en Javascript que debemos utilizar para realizar distintos tipos de acciones habituales.

## Cómo se escribe una función

Una función se debe definir con una sintaxis especial que vamos a conocer a continuación.

```
function nombrefuncion (){
    instrucciones de la función
    ...
}
```

Primero se escribe la palabra **function**, reservada para este uso. Seguidamente se escribe el nombre de la función, que como los nombres de variables puede tener números, letras y algún carácter adicional como en guión bajo. A continuación se colocan entre llaves las distintas instrucciones de la función. Las llaves en el caso de las funciones no son opcionales, además es útil colocarlas siempre como se ve en el ejemplo, para que se reconozca fácilmente la estructura de instrucciones que engloba la función.

Veamos un ejemplo de función para escribir en la página un mensaje de bienvenida dentro de etiquetas `<H1>` para que quede más resaltado.

```
function escribirBienvenida(){
    document.write("<H1>Hola a todos</H1>")
}
```

Simplemente escribe en la página un texto. Admitimos que es una función tan sencilla, que el ejemplo no expresa suficientemente el concepto de función, pero ya veremos otras más complejas. Las etiquetas H1 no se muestran en la página, sino que son interpretadas como el significado de la misma, en este caso que escribimos un encabezado de nivel 1. Como estamos escribiendo en una página web, al poner etiquetas HTML se interpretan como lo que son.

## Cómo llamar a una función

Para ejecutar una función la tenemos que invocar en cualquier parte de la página. Con eso conseguiremos que se ejecuten todas las instrucciones que tiene la función entre las dos llaves.



Para ejecutar la función utilizamos su nombre seguido de los paréntesis. Por ejemplo, así llamaríamos a la función escribirBienvenida() que acabamos de crear.

```
escribirBienvenida()
```

Luego veremos que existen muchas cosas adicionales que debemos conocer de las funciones, como el paso de parámetros o los valores de retorno. Pero antes vamos a explicar [dónde debemos colocar las funciones Javascript](#).

Este artículo es obra de *Miguel Angel Alvarez*

Fue publicado / actualizado en *02/11/2001*

Disponible online en <https://desarrolloweb.com/articulos/583.php>

## Dónde colocamos las funciones Javascript

**Vemos la manera de incluir funciones Javascript, de cliente, dentro de las páginas web.**

Las funciones son uno de los principales componentes de los programas, en la mayoría de los lenguajes de programación. En el [Manual de Javascript](#) ya hemos comenzado a explicar [qué es una función y cómo podemos crearla e invocarla](#) en este lenguaje. Ahora vamos a tratar un tema que no es tanto de sintaxis y programación, sino que tiene más que ver con el uso correcto y habitual que se hace de las funciones en Javascript, que no es otro que la colocación del código de las funciones en la página web.

En principio, podemos colocar las funciones en cualquier parte de la página, siempre entre etiquetas <SCRIPT>, claro está. No obstante existe una limitación a la hora de colocarla con relación a los lugares desde donde se la llame. Te adelantamos que lo más fácil es colocar la función antes de cualquier llamada a la misma y así seguro que nunca nos equivocaremos.

Existen dos opciones posibles para colocar el código de una función:

**a) Colocar la función en el mismo bloque de script:** En concreto, la función se puede definir en el bloque <SCRIPT> donde esté la llamada a la función, aunque es indiferente si la llamada se encuentra antes o después del código de la función, dentro del mismo bloque <SCRIPT>.

```
<SCRIPT>
miFuncion()
function miFuncion(){
    //hago algo...
    document.write("Esto va bien")
}
</SCRIPT>
```



Este ejemplo funciona correctamente porque la función está declarada en el mismo bloque que su llamada.

**b) Colocar la función en otro bloque de script:** También es válido que la función se encuentre en un bloque <SCRIPT> anterior al bloque donde está la llamada.

```
<HTML>
<HEAD>
    <TITLE>MI PÁGINA</TITLE>
<SCRIPT>
function miFuncion(){
    //hago algo...
    document.write("Esto va bien")
}
</SCRIPT>
</HEAD>
<BODY>

<SCRIPT>
miFuncion()
</SCRIPT>

</BODY>
</HTML>
```

Vemos un código completo sobre cómo podría ser una página web donde tenemos funciones Javascript. Como se puede comprobar, las funciones están en la cabecera de la página (dentro del HEAD). Éste es un lugar excelente donde colocarlas, porque se supone que en la cabecera no se van a utilizar todavía y siempre podremos disfrutar de ellas en el cuerpo porque sabemos seguro que ya han sido declaradas.

Para que quede claro este asunto de la colocación de funciones veamos el siguiente ejemplo, **que daría un error**. Examina atentamente el código siguiente, que lanzará un error, debido a que hacemos una llamada a una función que se encuentra declarada en un bloque <SCRIPT> posterior.

```
<SCRIPT>
miFuncion()
</SCRIPT>

<SCRIPT>
function miFuncion(){
    //hago algo...
    document.write("Esto va bien")
}
</SCRIPT>
```

Con esto esperamos haber resuelto todas las dudas sobre la colocación del código de las funciones Javascript. En siguientes artículos veremos otros temas interesantes como los [parámetros de las funciones](#).

Este artículo es obra de *Miguel Angel Alvarez*

Fue publicado / actualizado en 02/11/2001



Disponible online en <https://desarrolloweb.com/articulos/584.php>

## Parámetros de las funciones

**Vemos lo que son los parámetros en las funciones. Vemos como definir funciones que reciben parámetros en el lenguaje Javascript y como hacer llamadas a funciones pasando parámetros.**

En el [Manual de Javascript](#) hemos hablado anteriormente sobre funciones. En concreto este es el tercer artículo que abordamos sobre el tema.

Las ideas que hemos explicado anteriormente sobre funciones no son las únicas que debemos aprender para manejarlas en toda su potencia. Las funciones también tienen una entrada y una salida de datos. En este artículo veremos cómo podemos enviar datos a las funciones Javascript.

### Parámetros

Los parámetros se usan para mandar valores a las funciones. Una función trabajará con los parámetros para realizar las acciones. Por decirlo de otra manera, los parámetros son los valores de entrada que recibe una función.

Por poner un ejemplo sencillo de entender, una función que realizase una suma de dos números tendría como parámetros a esos dos números. Los dos números son la entrada, así como la salida sería el resultado de la suma, pero eso lo veremos más tarde.

Veamos un ejemplo anterior en el que creábamos una función para mostrar un mensaje de bienvenida en la página web, pero al que ahora le vamos a pasar un parámetro que contendrá el nombre de la persona a la que hay que saludar.

```
function escribirBienvenida(nombre){  
    document.write("<H1>Hola " + nombre + "</H1>")  
}
```

Como podemos ver en el ejemplo, para definir en la función un parámetro tenemos que poner el nombre de la variable que va a almacenar el dato que le pasemos. Esta variable, que en este caso se llama nombre, tendrá como valor el dato que le pasemos a la función cuando la llamemos. Además, la variable donde recibimos el parámetro tendrá vida durante la ejecución de la función y dejará de existir cuando la función termine su ejecución.

Para llamar a una función que tiene parámetros se coloca entre paréntesis el valor del parámetro. Para llamar a la función del ejemplo habría que escribir:

```
escribirBienvenida("Alberto García")
```



Al llamar a la función así, el parámetro nombre toma como valor "Alberto García" y al escribir el saludo por pantalla escribirá "Hola Alberto García" entre etiquetas <H1>.

Los parámetros pueden recibir cualquier tipo de datos, numérico, textual, booleano o un objeto. Realmente no especificamos el tipo del parámetro, por eso debemos tener un cuidado especial al definir las acciones que realizamos dentro de la función y al pasarle valores, para asegurarnos que todo es consecuente con los tipos de datos que esperamos tengan nuestras variables o parámetros.

## Múltiples parámetros

Una función puede recibir tantos parámetros como queramos y para expresarlo se colocan los nombres de los parámetros separados por comas, dentro de los paréntesis. Veamos rápidamente la sintaxis para que la función de antes, pero hecha para que reciba dos parámetros, el primero el nombre al que saludar y el segundo el color del texto.

```
function escribirBienvenida(nombre,colorTexto){  
    document.write("<FONT color=' " + colorTexto + "'>")  
    document.write("<H1>Hola " + nombre + "</H1>")  
    document.write("</FONT>")  
}
```

Llamaríamos a la función con esta sintaxis. Entre los paréntesis colocaremos los valores de los parámetros.

```
var miNombre = "Pepe"  
var miColor = "red"  
escribirBienvenida(miNombre,miColor)
```

He colocado entre los paréntesis dos variables en lugar de dos textos entrecomillados. Cuando colocamos variables entre los parámetros en realidad lo que estamos pasando a la función son los valores que contienen las variables y no las mismas variables.

## Los parámetros se pasan por valor

Al hilo del uso de parámetros en nuestros programas Javascript, tenemos que saber que los parámetros de las funciones se pasan por valor. Esto quiere decir que estamos pasando valores y no variables. En la práctica, aunque modifiquemos un parámetro en una función, la variable original que habíamos pasado no cambiará su valor. Se puede ver fácilmente con un ejemplo.

```
function pasoPorValor(miParametro){  
    miParametro = 32  
    document.write("he cambiado el valor a 32")  
}  
var miVariable = 5  
pasoPorValor(miVariable)  
document.write ("el valor de la variable es: " + miVariable)
```

En el ejemplo tenemos una función que recibe un parámetro y que modifica el valor del parámetro asignándole el valor 32. También tenemos una variable, que inicializamos a 5 y posteriormente



llamamos a la función pasándole esta variable como parámetro. Como dentro de la función modificamos el valor del parámetro podría pasar que la variable original cambiase de valor, pero como los parámetros no modifican el valor original de las variables, ésta no cambia de valor.

De este modo, una vez ejecutada la función, al imprimir en pantalla el valor de miVariable se imprimirá el número 5, que es el valor original de la variable, en lugar de 32 que era el valor con el que habíamos actualizado el parámetro.

En Javascript sólo se pueden pasar las variables por valor.

Ahora que hemos aprendido a enviar datos a las funciones, por medio de los parámetros, podemos [aprender a hacer funciones que devuelven valores](#).

Este artículo es obra de *Miguel Angel Alvarez*

Fue publicado / actualizado en *02/11/2001*

Disponible online en <https://desarrolloweb.com/articulos/585.php>

## Valores de retorno en funciones Javascript

**Las funciones pueden devolver valores, a través de la sentencia return. También vemos un apunte sobre el ámbito de variables en funciones en Javascript.**

Estamos aprendiendo acerca del uso de funciones en Javascript y en estos momentos quizás ya nos hayamos dado cuenta de la gran importancia que tienen para hacer programas más o menos avanzados. En este artículo del [Manual de Javascript](#) seguiremos aprendiendo cosas sobre funciones y en concreto que con ellas también se puede devolver valores. Además, veremos algún caso de uso interesante sobre las funciones que nos puede aclarar un poco el ámbito de variables locales y globales.

### Devolución de valores en las funciones

Las funciones en Javascript también pueden retornar valores. De hecho, ésta es una de las utilidades más esenciales de las funciones, que debemos conocer, no sólo en Javascript sino en general en cualquier lenguaje de programación. De modo que, al invocar una función, se podrá realizar acciones y ofrecer un valor como salida.

Por ejemplo, una función que calcula el cuadrado de un número tendrá como entrada a ese número y como salida tendrá el valor resultante de hallar el cuadrado de ese número. La entrada de datos en las funciones la vimos anteriormente en el artículo sobre [parámetros de las funciones](#). Ahora tenemos que aprender acerca de la salida.



Veamos un ejemplo de función que calcula la media de dos números. La función recibirá los dos números y retornará el valor de la media.

```
function media(valor1,valor2){  
    var resultado  
    resultado = (valor1 + valor2) / 2  
    return resultado  
}
```

Para especificar el valor que retornará la función se utiliza la palabra **return** seguida de el valor que se desea devolver. En este caso se devuelve el contenido de la variable resultado, que contiene la media calculada de los dos números.

Quizás nos preguntemos ahora cómo recibir un dato que devuelve una función. Realmente en el código fuente de nuestros programas podemos invocar a las funciones en el lugar que deseemos. Cuando una función devuelve un valor simplemente se sustituye la llamada a la función por ese valor que devuelve. Así pues, para almacenar un valor de devolución de una función, tenemos que asignar la llamada a esa función como contenido en una variable, y eso lo haríamos con el operador de asignación `=`.

Para ilustrar esto se puede ver este ejemplo, que llamará a la función `media()` y guardará el resultado de la media en una variable para luego imprimirla en la página.

```
var miMedia  
miMedia = media(12,8)  
document.write (miMedia)
```

## Múltiples return

En realidad en Javascript las funciones sólo pueden devolver un valor, por lo que en principio no podemos hacer funciones que devuelvan dos datos distintos.

**Nota:** en la práctica nada nos impide que una función devuelva más de un valor, pero como sólo podemos devolver una cosa, tendríamos que meter todos los valores que queremos devolver en una estructura de datos, como por ejemplo un [array](#). No obstante, eso sería un uso más o menos avanzado que no vamos a ver en estos momentos.

Ahora bien, aunque sólo podamos devolver un dato, en una misma función podemos colocar más de un `return`. Como decimos, sólo vamos a poder retornar una cosa, pero dependiendo de lo que haya sucedido en la función podrá ser de un tipo u otro, con unos datos u otros.

En esta función podemos ver un ejemplo de utilización de múltiples `return`. Se trata de una función que devuelve un `0` si el parámetro recibido era par y el valor del parámetro si este era impar.

```
function multipleReturn(numero){  
    var resto = numero % 2
```



```
if (resto == 0)
    return 0
else
    return numero
}
```

Para averiguar si un número es par hallamos el resto de la división al dividirlo entre 2. Si el resto es cero es que era par y devolvemos un 0, en caso contrario -el número es impar- devolvemos el parámetro recibido.

## Ámbito de las variables en funciones

Dentro de las funciones podemos declarar variables. Sobre este asunto debemos de saber que todas las variables declaradas en una función son locales a esa función, es decir, sólo tendrán validez durante la ejecución de la función.

**Nota:** Incluso, si lo pensamos, nos podremos dar cuenta que los parámetros son como variables que se declaran en la cabecera de la función y que se inicializan al llamar a la función. Los parámetros también son locales a la función y tendrán validez sólo cuando ésta se está ejecutando.

Podría darse el caso de que podemos declarar variables en funciones que tengan el mismo nombre que una variable global a la página. Entonces, dentro de la función, la variable que tendrá validez es la variable local y fuera de la función tendrá validez la variable global a la página.

En cambio, si no declaramos las variables en las funciones se entenderá por javascript que estamos haciendo referencia a una variable global a la página, de modo que si no está creada la variable la crea, pero siempre global a la página en lugar de local a la función.

Veamos el siguiente código.

```
function variables_globales_y_locales(){
    var variableLocal = 23
    variableGlobal = "qwerty"
}
```

En este caso variableLocal es una variable que se ha declarado en la función, por lo que será local a la función y sólo tendrá validez durante su ejecución. Por otra parte variableGlobal no se ha llegado a declarar (porque antes de usarla no se ha utilizado la palabra var para declararla). En este caso la variable variableGlobal es global a toda la página y seguirá existiendo aunque la función finalice su ejecución. Además, si antes de llamar a la función existiese la variable variableGlobal, como resultado de la ejecución de esta función, se machacaría un hipotético valor de esa variable y se sustituiría por "qwerty".

**Nota:** Podemos encontrar más información sobre [ámbito de variables](#) en un artículo anterior.

Con esto hemos terminado el tema de las funciones, así que en adelante nos dedicaremos a otros asuntos también interesantes, como son los [Arrays en Javascript](#).

Este artículo es obra de *Miguel Angel Alvarez*

Fue publicado / actualizado en *02/11/2001*

Disponible online en <https://desarrolloweb.com/articulos/586.php>

## Sobrecarga de funciones en Javascript

**Cómo realizar lo que se conoce como Sobre carga de funciones en el lenguaje Javascript, un concepto usado en la Programación Orientada a Objetos que no es exactamente igual en Javascript.**



Comenzamos con este artículo una serie de entregas dedicadas a "**Buenas prácticas en Javascript**" que nos servirán para entender un poco mejor algunas de las cosas que se pueden hacer en el lenguaje para mejorar nuestra programación.

El desarrollo web ha cambiado mucho en los últimos años y ahora es casi imprescindible tener un relativo **dominio de Javascript** para hacer un buen trabajo, nos dediquemos a lo que nos dediquemos dentro del desarrollo. Javascript ya no es el lenguaje que utilizamos para comprobar un formulario y ver si está todo correcto, sino que ahora, con una serie de frameworks como Angular, Backbone, Vue.js, etc. se nos abren muchas posibilidades nuevas.

Javascript es un lenguaje [orientado a objetos](#) serio que nos sirve para hacer grandes cosas, con el que realmente se pueden crear aplicaciones de todo tipo. Tanto es así que Javascript ha entrado en



el mundo backend de la mano de [NodeJS](#), un terreno vedado hasta entonces, ya que no existía método de usar Javascript del lado del servidor.

Este texto es una transcripción, no literal, de un interesante hangout impartido por Eduard Tomàs, que nos dejó muchas claves para entender Javascript, un lenguaje que tiene muchas cosas malas, pero también muchas cosas buenas. Para saber valorar Javascript y "amarlo", necesitamos entenderlo y en este programa #jsIO sobre buenas prácticas en Javascript que ahora estamos transcribiendo nos ayudará sin duda a ello.

El programa también está motivado por una serie de conceptos que los programadores en lenguajes más tradicionales entienden, pero no tratados exactamente igual en Javascript. Por eso, quien viene de lenguajes como PHP, C#, Java, quizás se encuentra con algunas confusiones cuando se aproxima a Javascript. Es el caso de clases, objetos, la variable "this" y el contexto, etc.

Durante el hangout se habló de muchas cosas interesantes. En este artículo abordaremos la sobrecarga de funciones en Javascript y veremos que en Javascript también se puede realizar, si conocemos algunos de los mecanismos del lenguaje.

Al final de este artículo puedes ver el hangout embebido.

## Sobrecarga de funciones

Antes que nada cabe aclarar: **como tal no existe la sobrecarga de funciones en Javascript**. Esto puede percibirse como una carencia en el lenguaje, pero es que realmente Javascript tiene otro mecanismo que, si quieras entenderlo así, puede incluso considerarse **más potente**. En resumen, se trata de lo siguiente:\*\*\* una función en Javascript puede ser llamada con cualquier número de argumentos, con independencia de los que se defina\*\*\*.

Es que Javascript puede invocar una función con cualquier juego de argumentos. Es decir, puedo **declararla con un número de parámetros dado, pero puedo invocarla con cualquier otro conjunto de argumentos que quiera o necesite**.

Si no coinciden los parámetros con los argumentos utilizados, no es considerado ningún error en el lenguaje, simplemente **el intérprete hará lo que pueda para adaptarse a esa manera de invocarla**. La situación será la siguiente:

- Si faltan parámetros, su valor será "undefined". Por ejemplo, si tengo una función definida con dos parámetros y al invocarla sólo defino el valor de uno de ellos, el segundo parámetro tendrá este valor.
- Si le sobran, puedo acceder a los parámetros restantes a través de "arguments". El objeto arguments está siempre disponible dentro de la función y contiene todos esos parámetros que se le han pasado a la función cuando fue invocada.

## Valores por defecto de los parámetros en funciones



En Javascript tradicional no existía el típico mecanismo que conocemos en otros lenguajes más extendidos, en el que las funciones pueden tener parámetros con valores por defecto, pero realmente podemos asignar esos valores por defecto de otra manera. Se hace dentro del código de las funciones y está basado en esta sintaxis:

Actualmente Javascript sí que admite valores por defecto declarados en los parámetros de las funciones. Si te interesa puedes leer el artículo [Parámetros con valores predeterminados en funciones Javascript ES6](#).

```
undefined || {algo} === {algo}
```

El anterior sería más bien un pedazo de "*pseudocódigo*" nos dice que `undefined` o `{algo}` es exactamente igual a `{algo}`. Pero lo vamos a ver mucho mejor con un código real para que se entienda mejor.

```
var mivariable = undefined || 45  
console.log(mivariable);
```

Como se puede ver, estamos definiendo una variable y el valor que le asignamos es `undefined` o `45`. Pues entonces, el valor de mi variable será siempre `45`. Lógicamente, aquí sigue sin tener mucho sentido, porque el valor de `undefined` es siempre indefinido y `45` es otro literal, pero lo podemos aplicar para definir valores por defecto en los parámetros de la siguiente manera:

```
function f(param){  
    var variable = param || 10;  
    console.log(variable);  
}
```

En esta función tenemos un parámetro "`param`". En la primera línea definimos una variable llamada "`variable`" a la que le asignamos el valor de parámetro "`param`" o el literal `10`.

Recordemos que a la función podemos invocarla enviando o no el valor de ese parámetro con el que fue declarada. Si no lo enviamos, simplemente el parámetro tiene como valor `undefined`. De este modo, al crear la variable "`variable`", si "`param`" era `undefined`, pues simplemente se le asignará el valor `10`.

Si llamamos la función con distintos juegos de argumentos:

```
f(80);  
f();
```

En el primer caso nos mostraría `80` por consola y en el segundo nos mostraría `10`, que es el valor por defecto, ya que no se le ha pasado ningún parámetro.



## Objeto arguments o "array" de arguments

Dentro de una función, sin que tengas que definir nada, existe un objeto llamado `arguments` que te servirá para acceder a todos los parámetros que puedan invocarse en la función que estás definiendo. En la variable `arguments` encontrarás un número variable de casillas, dependiendo del número de parámetros con el que se invoque cada vez a la función.

Ten en cuenta que `arguments` es realmente un objeto, pero muchos programadores lo tratan como si fuera un "array" y eso no está mal. De hecho, a veces por simplicidad, podemos referirnos a `arguments` como array, a pesar que realmente en Javascript es un objeto. En verdad, es un objeto donde todos sus índices son numéricos y en nuestra programación lo podemos tratar tranquilamente como si fuera un simple array, abstrayéndonos de que internamente Javascript lo define como un objeto. Insistimos: a pesar que accedemos a `arguments` con índices numéricos y tenemos `arguments.length` que nos dice el número de argumentos pasados a la función, no es un array. Por ejemplo, no tendremos en `arguments` algunos métodos de los arrays como `forEach()` o `map()`.

```
function args(){
    console.log(arguments);
    console.log(arguments.length);
}
```

Como ves, lanzamos un par de mensajes a la consola, sencillos, simplemente para mostrar el contenido de `arguments` y luego para saber el número de parámetros en la invocación de la función.

Luego podemos invocar a la función con varios juegos de parámetros:

```
args();
args(2);
args("gol", "sol", "remo", 445);
```

Como verás, si pones ese código en ejecución, en `arguments` encontrarás cada uno de los valores de esos parámetros y en "`arguments.length`" el número de parámetros de cada invocación.

## Ejemplo de sobrecarga de funciones

Con todo lo que has aprendido, puedes **crear funciones que acepten cualquier número de parámetros e implementar los valores por defecto que necesites**.

Librerías como jQuery también te permiten mediante sus funcionalidades hacer este tipo de cosas con un código de nivel un poco más alto, como invocar a las funciones pasando como parámetro distintos juegos de valores. Lo podemos combinar con el operador "`typeof`" si lo necesitamos para saber qué tipos son los parámetros que estamos recibiendo, por si queremos hacer cosas diferentes con los parámetros dependiendo del tipo.



```
typeof(33); //nos devolverá number  
typeof("hola"); //nos devolverá string
```

Solo para probar lo aprendido en este artículo, vamos a ver un ejemplo de sobrecarga que nos permite sumar un número indeterminado de parámetros. Lógicamente, solo puedo sumar los parámetros que tengan valores numéricos.

```
function sumar() {  
    var suma = 0;  
    for(var i=0; i<arguments.length; i++){  
        if(typeof(arguments[i])=="number"){  
            suma += arguments[i];  
        }else{  
            console.log("no puedo sumar ", arguments[i], " porque es un ", typeof(arguments[i]));  
        }  
    }  
    return suma;  
}
```

A esta función podríamos llamarla con cualquier número de parámetros, de un modo como este:

```
sumar(34, ["kk", "jj"], 3, "hola", 3.5); //nos devolverá 40.5
```

Con esto acabamos este artículo sobre **sobrecarga de funciones en Javascript**. Hemos llegado tan solo al minuto 17 del Hangout de Buenas prácticas en Javascript #jsIO, que os recomendamos ver para seguir aprendiendo. En futuras entregas iremos cubriendo otros de los conceptos que se explicaron aquel día.

Te dejamos el vídeo de la charla que hemos mencionado de buenas prácticas en JS para que puedas verlo si te interesa.

Para ver este vídeo es necesario visitar el artículo original en:

<https://desarrolloweb.com/articulos/sobrecarga-funciones-javascript.html>

Este artículo es obra de *Miguel Angel Alvarez*

Fue publicado / actualizado en *25/07/2023*

Disponible online en <https://desarrolloweb.com/articulos/sobrecarga-funciones-javascript.html>

## Funciones integradas en el lenguaje Javascript

## Vamos a ver algunas funciones que el lenguaje Javascript incorpora de manera nativa para realizar diversos trabajos de utilidad variada.

En todos los lenguajes de programación existen librerías de funciones que sirven para hacer cosas diversas y muy repetitivas a la hora de programar. Las librerías de los lenguajes de programación ahoran la tarea de escribir las funciones comunes que por lo general pueden necesitar los programadores. Un lenguaje de programación bien desarrollado tendrá una buena cantidad de ellas. En ocasiones es más complicado conocer bien todas las librerías que aprender a programar en el lenguaje.

Javascript contiene una buena cantidad de funciones en sus librerías. Como se trata de un lenguaje que trabaja con objetos muchas de las librerías se implementan a través de objetos. Por ejemplo, las funciones matemáticas o las de manejo de strings se implementan mediante los objetos Math y String. Sin embargo, existen algunas funciones que no están asociadas a ningún objeto y son las que veremos en este capítulo, ya que todavía no conocemos los objetos y no los necesitaremos para estudiarlas.

### Funciones incorporadas en Javascript

Estas son las funciones que Javascript pone a disposición de los programadores.

#### `eval(string)`

Esta función recibe una cadena de caracteres y la ejecuta como si fuera una sentencia de Javascript.

#### `parseInt(cadena,base)`

Recibe una cadena y una base. Devuelve un valor numérico resultante de convertir la cadena en un número en la base indicada.

#### `parseFloat(cadena)`

Convierte la cadena en un número y lo devuelve.

#### `escape(carácter)`

Devuelve un el carácter que recibe por parámetro en una codificación ISO Latin 1.

#### `unescape(carácter)`

Hace exatamente lo opuesto a la función escape.

#### `isNaN(número)`

Devuelve un boleano dependiendo de lo que recibe por parámetro. Si no es un número devuelve un true, si es un numero devuelve false.

Las librerías que se implementan mediante objetos y las del manejo del explorador, que también se manejan con objetos, las veremos más adelante.



**Nota:** No queremos llevar a engaño a las personas con esta corta lista de funciones nativas de Javascript. Realmente existen muchas otras funciones que vamos a ver a lo largo del presente manual, lo que ocurre es que están asociadas a objetos. Por ejemplo, como habíamos señalado, existen funciones de cadenas de caracteres, que están asociadas a objetos string, funciones para trabajo con cálculos matemáticos avanzados, que están asociadas a la clase Math, funciones para trabajo con el objeto de la ventana del navegador, con el documento, etc.

## Ejemplos de uso de las funciones incorporadas en Javascript

Hasta el momento hemos conocido simplemente un listado de las funciones nativas del lenguaje Javascript. Ahora podemos ver varios ejemplos de utilización de funciones nativas de Javascript, que tenemos disponibles en cualquier navegador y en cualquier versión de Javascript.

Veremos tres funciones de diverso ámbito que resultan bastante fundamentales en el trabajo habitual con este lenguaje, explicadas a través de ejemplos.

### Función eval

Esta función es muy importante, tanto que hay algunas aplicaciones de Javascript que no se podrían realizar si no la utilizamos. Su utilización es muy simple, pero puede que resulte un poco más complejo entender en qué casos utilizarla porque a veces resulta un poco sutil su aplicación.

Con los conocimientos actuales no podemos hacer un ejemplo muy complicado, pero por lo menos podemos ver en marcha la función. Vamos a utilizarla en una sentencia un poco rara y bastante inservible, pero si la conseguimos entender conseguiremos entender también la función eval.

```
var miTexto = "3 + 5"
eval("document.write(" + miTexto + ")")
```

Primero creamos una variable con un texto, en la siguiente línea utilizamos la función eval y como parámetro le pasamos una instrucción javascript para escribir en pantalla. Si concatenamos los strings que hay dentro de los paréntesis de la función eval nos queda esto.

```
document.write(3 + 5)
```

La función eval ejecuta la instrucción que se le pasa por parámetro, así que ejecutará esta sentencia, lo que dará como resultado que se escriba un 8 en la página web. Primero se resuelve la suma que hay entre paréntesis, con lo que obtenemos el 8 y luego se ejecuta la instrucción de escribir en pantalla.

### Función parseInt

Esta función recibe un número, escrito como una cadena de caracteres, y un número que indica una base. En realidad puede recibir otros tipos de variables, dado que las variables no tienen tipo en



Javascript, pero se suele utilizar pasándole un string para convertir la variable de texto en un número.

Las distintas bases que puede recibir la función son 2, 8, 10 y 16. Si no le pasamos ningún valor como base la función interpreta que la base es decimal. El valor que devuelve la función siempre tiene base 10, de modo que si la base no es 10 convierte el número a esa base antes de devolverlo.

Veamos una serie de llamadas a la función parseInt para ver lo que devuelve y entender un poco más la función.

```
document.write (parseInt("34"))
```

Devuelve el numero 34

```
document.write (parseInt("101011",2))
```

Devuelve el numero 43

```
document.write (parseInt("34",8))
```

Devuelve el numero 28

```
document.write (parseInt("3F",16))
```

Devuelve el numero 63

Esta función se utiliza en la práctica para un montón de cosas distintas en el manejo con números, por ejemplo obtener la parte entera de un decimal.

```
document.write (parseInt("3.38"))
```

Devuelve el numero 3

También es muy habitual su uso para saber si una variable es numérica, pues si le pasamos un texto a la función que no sea numérico nos devolverá NaN (Not a Number) lo que quiere decir que No es un Número.

```
document.write (parseInt("desarrolloweb.com"))
```

Devuelve el numero NaN

Este mismo ejemplo es interesante con una modificación, pues si le pasamos una combinación de letras y números nos dará lo siguiente.

```
document.write (parseInt("16XX3U"))
```



Devuelve el numero 16

```
document.write (parseInt("TG45"))
```

Devuelve el numero NaN

Como se puede ver, la función intenta convertir el string en número y si no puede devolver NaN.

Todos estos ejemplos, un tanto inconexos, sobre cómo trabaja parseInt los revisaremos más adelante en ejemplos más prácticos cuando tratemos el trabajo con formularios.

### Función isNaN

Esta función devuelve un booleano dependiendo de si lo que recibe es un número o no. Lo único que puede recibir es un número o la expresión NaN. Si recibe un NaN devuelve true y si recibe un número devuelve false. Es una función muy sencilla de entender y de utilizar.

La función suele trabajar en combinación con la función parseInt o parseFloat, para saber si lo que devuelven estas dos funciones es un número o no.

```
miInteger = parseInt("A3.6")
isNaN(miInteger)
```

En la primera línea asignamos a la variable miInteger el resultado de intentar convertir a entero el texto A3.6. Como este texto no se puede convertir a número la función parseInt devuelve NaN. La segunda línea comprueba si la variable anterior es NaN y como si lo es devuelve un true.

```
miFloat = parseFloat("4.7")
isNaN(miFloat)
```

En este ejemplo convertimos un texto a número con decimales. El texto se convierte perfectamente porque corresponde con un número. Al recibir un número la función isNaN devuelve un false.

#### Referencia: [Validar entero en campo de formulario](#)

Tenemos un Taller de Javascript muy interesante que ha sido realizado para afianzar los conocimientos de estos capítulos. Se trata de un script para validar un campo de formulario de manera que sepamos seguro que dentro del campo hay siempre un número entero. Puede ser muy interesante leerlo ahora, ya que utilizamos las funciones isNaN() y parseInt(). [Ver el taller](#)

Esperamos que los ejemplos vistos en este artículo hayan resultado interesantes. No obstante, como habíamos señalado anteriormente, existen bastantes otras funciones nativas en Javascript que debemos conocer, pero que están asociadas a [clases y objetos nativos Javascript](#). Pero antes de pasar a ese punto queremos ofrecer una pequeña [guía básica para el trabajo con programación orientada a objetos en Javascript](#).



Este artículo es obra de *Miguel Angel Alvarez*

Fue publicado / actualizado en *11/03/2002*

Disponible online en <https://desarrolloweb.com/articulos/705.php>

## IIFE, closures o simplemente envoltura de función en Javascript

**Explicamos un concepto ampliamente usado en Javascript para crear una envoltura para una función, llamada como closure o las siglas IIFE.**



Vamos a ponerle nombre a algo que quizás ya conozcas, ya habías usado en Javascript o al menos visto en algún código por ahí. Se trata de las envolturas de funciones que podemos crear para aislar un código fuente y evitar colisiones de variables o espacios de nombres en general.

Yo siempre le he llamado simplemente "envoltura" y en inglés "closure", pero es un patrón que tiene nombre con las siglas IIFE que vienen de "Immediately-invoked function expression". Explicaremos de donde viene y cuál es su utilidad en el mundo de la programación en Javascript.

Este artículo quizás sea un poco avanzado, teniendo en cuenta el momento en el que estamos en el [Manual de Javascript](#), si estás aprendiendo Javascript en estos momentos. No te preocupes si no lo entiendes porque es un conocimiento que puedes dejar para más adelante. Permite aplicar prácticas aconsejables para el desarrollo de aplicaciones, pero no son para nada imprescindibles en el desarrollo con Javascript.

### Problemática

Todo el mundo conoce ya el riesgo de las variables globales y sabe que las colisiones de nombres a veces ocurren, trayendo consecuencias desastrosas para la ejecución de los programas. Lo peor es que a menudo los motivos de los problemas son difíciles de detectar. Eso unido a que en Javascript acostumbramos a usar librerías con códigos de distintos lugares, produce que en ocasiones esa problemática sea todavía más frecuente.



Por otra parte, en la programación orientada a objetos sabemos que la encapsulación es una de las mecánicas que aportan diversos beneficios al código. Javascript no tiene clases y tampoco puedes crear dentro de objetos variables que sean privadas, por lo que se deben implementar mecanismos alternativos.

Por lo tanto tenemos que buscar un método de crear módulos que sean lo más independientes posibles y donde tengamos un ámbito de las variables propio y restringido.

\*\*Nota:\*\* En las próximas versiones de Javascript (dentro de poco tendremos ECMA 6) se irán aportando soluciones "nativas" (provistas por el propio lenguaje) a estas situaciones. De momento tenemos que hacerlo un poco "a mano".

## Solución

Todo pasa por crear un ámbito propio para tu código, donde las variables que crees existan solo dentro de ese ámbito reducido, y no se puedan acceder desde fuera. Esto en código no es nada difícil de conseguir, pues con Javascript podemos crear ámbitos simplemente con funciones.

Las variables declaradas con "var" dentro de una función son locales a esa función. Sin embargo, estamos exponiendo hacia fuera el propio nombre de la función.

```
function miFuncion(){
    var x = "hola DesarrolloWeb.com";
}
```

Esto no es desastroso, pues solo estamos reclamando para nosotros el nombre "miFuncion" al que le estamos asignando una función. Pero se puede mejorar.

Lo conseguimos con funciones anónimas que se auto-invocan. Existen varias maneras de conseguir esto, pero el patrón comúnmente usado es este:

```
(function(){
    //código de tu función
})();
```

Eso es lo que significa IIFE y lo que se llama habitualmente como "closure" o envoltura.

## Punto y coma "defensivo"

Solo un detalle interesante que encontrarás por ahí. Generalmente colocamos un punto y coma antes del paréntesis inicial. Ésto se hace para conseguir que los paréntesis donde estamos encerrando la función no se tomen como una llamada a una función.

```
var y = 3;
var x = y
(function(){
```



```
//esto sintácticamente es correcto
})();
```

Ese código es sintácticamente correcto, a pesar que en la segunda línea no se haya finalizado con un punto y coma, pues sabemos que ";" como final de las sentencias es opcional en Javascript. El problema es que la envoltura de la función se va a tomar como si fueran parámetros enviados a una supuesta función "y". Esto provoca que tu código arroje un error diciendo que un número ( $y = 3$ ) no es una función.

La solución es colocar un punto y coma antes del closure, lo que se llama un "punto y coma defensivo".

```
var y = 3;
var x = y
;(function(){
    //Ahora es mejor todavía!
})();
```

Listo! ahora ya no da error. Podrías pensar que nuestro código podría ser más correcto si le colocamos el punto y coma al final de la segunda línea y tendrías razón. Si fuera tu propio código así lo harías, pero el caso es que no siempre va a ser tu código lo que tengas detrás, pues puedes haber incluido un script creado por otro desarrollador o una librería donde se hayan olvidado de colocar los ";".

## Paso de variables

A veces necesitamos pasar variables a nuestro código, para realizar cualquier tipo de operación donde las necesites. Por ejemplo es el caso de la librería jQuery, si es que en ese código vas a usarla. \$ es una variable global que te crea la librería, pero lo cierto es que no siempre \$ puede significar jQuery, pues esa variable puede haber sido ocupada por otras librerías o códigos Javascript.

En las buenas prácticas de creación de plugins jQuery aprendimos que es ideal pasar la variable jQuery a la función que tenemos en la envoltura y recogerla con el nombre \$, para asegurarnos que dentro de esa función \$ siempre equivale a jQuery.

```
;(function($){
    //Ahora $ siempre es jQuery
})(jQuery);
```

No me extiendo aquí en más explicaciones porque ya las vimos en el [artículo Alias personalizado en plugins jQuery](#). Pero sí quiero comentar algo que comúnmente se usa y que se aprende si lees el código fuente de la propia librería jQuery.

Ese sistema para crear alias de variables lo puedes usar para cualquier variable u objeto que piensas usar dentro de la función, por ejemplo, window o document. Aunque éstos sean objetos globales escribir "document" es muy largo (y no digamos algo como document.forms[0].campo) y si dentro de tu función vas a usarlo varias veces, puedes asignarle un alias personalizado de manera similar a como hiciste con jQuery.



```
; (function(w,d,o){  
    //Ahora w es un alias (shortcut) para window  
    //d es un alias de document  
    //o es un alias de otraVariableMuyLarga  
    (window, document, otraVariableMuyLarga));
```

Simplemente escribirás menos y tu código será menos pesado en bytes.

## Variables privadas

Ahora quiero ahondar sobre la posibilidad de hacer variables encapsuladas en objetos. Lo que se conoce en programación orientada a objetos como variables privadas. Javascript no las implementa.

Para aclararnos veamos este código:

```
var cuadrado = {  
    altura: 2,  
    anchura: 3,  
    area: function(){  
        return this.altura * this.anchura;  
    }  
}
```

Tenemos definido un objeto cuadrado (debería haberle llamado rectángulo), con valores en sus propiedades asignados de manera literal. Desde fuera de mi objeto podré acceder a todos sus elementos, tanto propiedades como métodos.

```
cuadrado.altura=10;  
console.log(cuadrado.area());
```

Ahora veamos este segundo código. Ya usa nuestro método de envoltura para crear un cuadrado y además podemos asignarle los valores de altura y anchura a través de un par de parámetros.

```
var cuadrado2 = (function(al, an){  
    return {  
        altura: al,  
        anchura: an,  
        area: function(){  
            return this.altura * this.anchura;  
        }  
    };  
}(5,4));
```

Lo que pasa es que no hemos adelantado mucho, pues seguimos pudiendo acceder a esas propiedades y métodos.

```
console.log(cuadrado2.area());  
cuadrado2.altura=10;  
console.log(cuadrado2.area());
```

Recuerda que queríamos que esas propiedades fueran privadas, solo accesibles desde mi función. Lo conseguimos con variables locales a la función.



```
var cuadrado3 = (function(al, an){
    var altura = al;
    var anchura = an;
    return {
        area: function(){
            return altura * anchura;
        }
    };
})(4,6);
```

En este último caso tenemos un ejemplo bastante ilustrativo de lo que nos permite este patrón IIFE, pues ahora desde fuera no vamos a poder acceder a las propiedades (realmente es que ya no son propiedades del objeto, sino variables locales a la función, sin embargo, dentro del código de los métodos de mi objeto las puedo acceder como si fueran propiedades normales).

```
console.log(cuadrado3.area());
console.log(cuadrado3.altura); // me dice que undefined
```

Seguiremos pudiendo acceder al método `area()`, que nos devolverá el dato correcto. Pero si intentamos acceder a la propiedad "privada" `altura`, observaremos que Javascript nos dice que es "undefined".

\*\*Nota:\*\* Para la redacción de partes de este artículo he tomado como referencia la Wikipedia, en la entrada sobre [IIFE](#). Si sigues ese enlace encontrarás a su vez las referencias de la propia Wikipedia que te darán acceso a artículos todavía más técnicos sobre este patrón de programación.

Este artículo es obra de *Miguel Angel Alvarez*  
Fue publicado / actualizado en *09/12/2014*  
Disponible online en <https://desarrolloweb.com/articulos/iife-closures-envolutra-funcion-javascript.html>

# Arrays Javascript

Los arrays, también llamados tablas o matrices, son la primera estructura de datos que podemos aprender en Javascript y en otros lenguajes de programación. Sin duda serán imprescindibles para desarrollar programas medianamente avanzados.

## Arrays en Javascript

**Vemos que son los arrays en Javascript, para qué sirven y cómo utilizarlos. Veremos diversas formas de crearlos, así como definir y acceder a sus valores.**

Pasamos a un nuevo tema en el [Manual de Javascript](#), en el que vamos a conocer nuestra primera estructura de datos.

En los lenguajes de programación existen estructuras de datos especiales que nos sirven para guardar información más compleja que simples variables. Una estructura típica en todos los lenguajes es el Array, que es como una variable donde podemos introducir varios valores, en lugar de solamente uno como ocurre con las variables normales.

Los arrays nos permiten guardar varias variables y acceder a ellas de manera independiente, es como tener una variable con distintos compartimentos donde podemos introducir datos distintos. Para ello utilizamos un índice que nos permite especificar el compartimiento o posición a la que nos estamos refiriendo.

**Nota:** Los arrays se introdujeron en versiones Javascript 1.1 o superiores, es decir, solo los podemos utilizar a partir de los navegadores 3.0. Para navegadores antiguos se puede simular el array utilizando sintaxis de programación orientada a objetos, pero la verdad es que actualmente esta limitación no debe preocuparnos. Además, dada la complejidad de la tarea de simular un array por medio de objetos, por lo menos en el momento en que nos encontramos y las pocas ocasiones en que lo necesitaremos, opinamos que es mejor olvidarnos de ese asunto y trabajar simplemente con los arrays normalmente. Así que en este artículo y los siguientes vamos a ver cómo utilizar el auténtico array de Javascript.

## Creación de Arrays javascript

El primer paso para utilizar un array es crearlo. Para ello utilizamos un objeto Javascript ya implementado en el navegador. Veremos en adelante un tema para explicar lo que es la orientación



a objetos, aunque no será necesario para poder entender el uso de los arrays. Esta es la sentencia para crear un objeto array:

```
var miArray = new Array()
```

Esto crea un array en la página que está ejecutándose. El array se crea sin ningún contenido, es decir, no tendrá ninguna casilla o compartimiento creado. También podemos crear el array Javascript especificando el número de compartimentos que va a tener.

```
var miArray = new Array(10)
```

En este caso indicamos que el array va a tener 10 posiciones, es decir, 10 casillas donde guardar datos.

Es importante que nos fijemos que la palabra Array en código Javascript se escribe con la primera letra en mayúscula. Como en Javascript las mayúsculas y minúsculas sí que importan, si lo escribimos en minúscula no funcionará.

Tanto se indique o no el número de casillas del **array javascript**, podemos introducir en el array cualquier dato. Si la casilla está creada se introduce simplemente y si la casilla no estaba creada se crea y luego se introduce el dato, con lo que el resultado final es el mismo. Esta creación de casillas es dinámica y se produce al mismo tiempo que los scripts se ejecutan. Veamos a continuación cómo introducir valores en nuestros arrays.

```
miArray[0] = 290  
miArray[1] = 97  
miArray[2] = 127
```

Se introducen indicando entre corchetes el índice de la posición donde queríamos guardar el dato. En este caso introducimos 290 en la posición 0, 97 en la posición 1 y 127 en la 2.

**Los arrays en Javascript empiezan siempre en la posición 0**, así que un array que tenga por ejemplo 10 posiciones, tendrá casillas de la 0 a la 9. Para recoger datos de un array lo hacemos igual: poniendo entre corchetes el índice de la posición a la que queremos acceder. Veamos cómo se imprimiría en la pantalla el contenido de un array.

```
var miArray = new Array(3)  
  
miArray[0] = 155  
miArray[1] = 4  
miArray[2] = 499  
  
for (i=0;i<3;i++){  
    document.write("Posición " + i + " del array: " + miArray[i])  
    document.write("<br>")  
}
```

Hemos creado un array con tres posiciones, luego hemos introducido un valor en cada una de las posiciones del array y finalmente las hemos impreso. En general, el recorrido por arrays para



imprimir sus posiciones, o cualquier otra cosa, se hace utilizando bucles. En este caso utilizamos un bucle FOR que va desde el 0 hasta el 2.

Podemos [ver el ejemplo en marcha en otra página](#).

## Tipos de datos en los arrays

En las casillas de los arrays podemos guardar datos de cualquier tipo. Podemos ver un array donde introducimos datos de tipo carácter.

```
miArray[0] = "Hola"  
miArray[1] = "a"  
miArray[2] = "todos"
```

Incluso, en Javascript podemos guardar distintos tipos de datos en las casillas de un mismo array. Es decir, podemos introducir números en unas casillas, textos en otras, booleanos o cualquier otra cosa que deseemos.

```
miArray[0] = "desarrolloweb.com"  
miArray[1] = 1275  
miArray[1] = 0.78  
miArray[2] = true
```

## Declaración e inicialización resumida de Arrays

En Javascript tenemos a nuestra disposición una manera resumida de declarar un array y cargar valores en un mismo paso. Fijémonos en el código siguiente:

```
var arrayRapido = [12,45,"array inicializado en su declaración"]
```

Como se puede ver, se está definiendo una variable llamada arrayRapido y estamos indicando en los corchetes varios valores separados por comas. Esto es lo mismo que haber declarado el array con la función Array() y luego haberle cargado los valores uno a uno.

En el próximo artículo seguiremos viendo cosas relacionadas con los arrays, en concreto aprenderemos a [acceder a la longitud de un array](#).

Este artículo es obra de *Miguel Angel Alvarez*  
Fue publicado / actualizado en *22/12/2001*  
Disponible online en <https://desarrolloweb.com/articulos/630.php>

## Longitud de los arrays



## Aprendemos más cosas sobre el funcionamiento de los arrays y en concreto vemos como utilizar su propiedad length para acceder al número de casillas que tiene.

En el artículo anterior del [Manual de Javascript](#) empezamos a explicar el [concepto de array y su utilización en Javascript](#). En este artículo vamos a continuar con el tema, mostrando el uso de su propiedad length.

Todos los arrays en javascript, aparte de almacenar el valor de cada una de sus casillas, también almacenan el número de posiciones que tienen. Para ello utilizan una propiedad del objeto array, la propiedad length. Ya veremos en objetos qué es una propiedad, pero para nuestro caso podemos imaginarnos que es como una variable, adicional a las posiciones, que almacena un número igual al número de casillas que tiene el array.

Para acceder a una propiedad de un objeto se ha de utilizar el operador punto. Se escribe el nombre del array que queremos acceder al número de posiciones que tiene, sin corchetes ni paréntesis, seguido de un punto y la palabra length.

```
var miArray = new Array()  
  
miArray[0] = 155  
miArray[1] = 499  
miArray[2] = 65  
  
document.write("Longitud del array: " + miArray.length)
```

Este código imprimiría en pantalla el número de posiciones del array, que en este caso es 3. Recordamos que un array con 3 posiciones abarca desde la posición 0 a la 2.

Es muy habitual que se utilice la propiedad length para poder recorrer un array por todas sus posiciones. Para ilustrarlo vamos a ver un ejemplo de recorrido por este array para mostrar sus valores.

```
for (i=0;i<miArray.length;i++){  
    document.write(miArray[i])  
}
```

Hay que fijarse que el bucle for se ejecuta siempre que i valga menos que la longitud del array, extraída de su propiedad length.

El siguiente ejemplo nos servirá para conocer mejor los recorridos por los arrays, el funcionamiento de la propiedad length y la creación dinámica de nuevas posiciones. Vamos a crear un array con 2 posiciones y llenar su valor. Posteriormente introduciremos un valor en la posición 5 del array. Finalmente imprimiremos todas las posiciones del array para ver lo que pasa.

```
var miArray = new Array(2)  
  
miArray[0] = "Colombia"  
miArray[1] = "Estados Unidos"  
  
miArray[5] = "Brasil"
```



```
for (i=0;i<miArray.length;i++){
    document.write("Posición " + i + " del array: " + miArray[i])
    document.write("<br>")
}
```

El ejemplo es sencillo. Se puede apreciar que hacemos un recorrido por el array desde 0 hasta el número de posiciones del array (indicado por la propiedad length). En el recorrido vamos imprimiendo el número de la posición seguido del contenido del array en esa posición. Pero podemos tener una duda al preguntarnos cuál será el número de elementos de este array, ya que lo habíamos declarado con 2 y luego le hemos introducido un tercero en la posición 5. Al ver la salida del programa podremos contestar nuestras preguntas. Será algo parecido a esto:

Posición 0 del array: Colombia

Posición 1 del array: Estados Unidos

Posición 2 del array: null

Posición 3 del array: null

Posición 4 del array: null

Posición 5 del array: Brasil

Se puede ver claramente que el número de posiciones es 6, de la 0 a la 5. Lo que ha ocurrido es que al introducir un dato en la posición 5, todas las casillas que no estaban creadas hasta la quinta se crean también.

Las posiciones de la 2 a la 4 están sin inicializar. En este caso nuestro navegador ha escrito la palabra *null* para expresar esto, pero otros navegadores podrán utilizar la palabra *undefined*. Ya veremos más adelante qué es este *null* y dónde lo podemos utilizar, lo importante ahora es que comprendas cómo trabajan los arrays y los utilices correctamente.

Podemos [ver el efecto de este script en tu navegador en una página a parte](#).

Continuaremos el tema de arrays en la siguiente entrega de este manual: [Arrays multidimensionales](#).

Este artículo es obra de *Miguel Angel Alvarez*

Fue publicado / actualizado en *22/12/2001*

Disponible online en <https://desarrolloweb.com/articulos/631.php>

## Arrays multidimensionales en Javascript

**Vemos qué son los arrays multidimensionales (arrays de más de una dimensión) y cómo utilizarlos. Además explicamos cómo inicializar arrays en su declaración.**

# Arrays multidimensionales



Como estamos viendo, los arrays son bastante importantes en Javascript y también en la mayoría de los lenguajes de programación. En concreto ya hemos aprendido a crear arrays y utilizarlos en artículos anteriores del [Manual de Javascript](#). Pero aun nos quedan algunas cosas importantes que explicar, como son los arrays de varias dimensiones.

Los arrays multidimensionales son un estructuras de datos que almacenan los valores en más de una dimensión. Los arrays que hemos visto hasta ahora almacenan valores en una dimensión, por eso para acceder a las posiciones utilizamos tan solo un índice. Los arrays de 2 dimensiones guardan sus valores, por decirlo de alguna manera, en filas y columnas y por ello necesitaremos dos índices para acceder a cada una de sus posiciones.

Dicho de otro modo, un array multidimensional es como un contenedor que guardara más valores para cada posición, es decir, como si los elementos del array fueran a su vez otros arrays.

En Javascript no existe un auténtico objeto array-multidimensional. Para utilizar estas estructuras podremos definir arrays que donde en cada una de sus posiciones habrá otro array. En nuestros programas podremos utilizar arrays de cualquier dimensión, veremos a continuación cómo trabajar con arrays de dos dimensiones, que serán los más comunes.

En este ejemplo vamos a crear un array de dos dimensiones donde tendremos por un lado ciudades y por el otro la temperatura media que hace en cada una durante de los meses de invierno.

```
var temperaturas_medias_ciudad0 = new Array(3)
temperaturas_medias_ciudad0[0] = 12
temperaturas_medias_ciudad0[1] = 10
temperaturas_medias_ciudad0[2] = 11

var temperaturas_medias_ciudad1 = new Array (3)
temperaturas_medias_ciudad1[0] = 5
temperaturas_medias_ciudad1[1] = 0
temperaturas_medias_ciudad1[2] = 2

var temperaturas_medias_ciudad2 = new Array (3)
temperaturas_medias_ciudad2[0] = 10
temperaturas_medias_ciudad2[1] = 8
temperaturas_medias_ciudad2[2] = 10
```

Con las anteriores líneas hemos creado tres arrays de 1 dimensión y tres elementos, como los que ya conocíamos. Ahora crearemos un nuevo array de tres elementos e introduciremos dentro de cada



una de sus casillas los arrays creados anteriormente, con lo que tendremos un array de arrays, es decir, un array de 2 dimensiones.

```
var temperaturas_cuidades = new Array (3)
temperaturas_cuidades[0] = temperaturas_medias_ciudad0
temperaturas_cuidades[1] = temperaturas_medias_ciudad1
temperaturas_cuidades[2] = temperaturas_medias_ciudad2
```

Vemos que para introducir el array entero hacemos referencia al mismo sin paréntesis ni corchetes, sino sólo con su nombre. El array temperaturas\_cuidades es nuestro array bidimensional.

También es interesante ver cómo se realiza un recorrido por un array de dos dimensiones. Para ello tenemos que hacer un bucle que pase por cada una de las casillas del array bidimensional y dentro de éstas hacer un nuevo recorrido para cada una de sus casillas internas. Es decir, un recorrido por un array dentro de otro.

El método para hacer un recorrido dentro de otro es colocar un bucle dentro de otro, lo que se llama un bucle anidado. En este ejemplo vamos a meter un bucle FOR dentro de otro. Además, vamos a escribir los resultados en una tabla, lo que complicará un poco el script, pero así podremos ver cómo construir una tabla desde Javascript a medida que realizamos el recorrido anidado al bucle.

```
document.write("<table width=200 border=1 cellpadding=1 cellspacing=1>");
for (i=0;i<temperaturas_cuidades.length;i++){
    document.write("<tr>")
    document.write("<td><b>Ciudad " + i + "</b></td>")
    for (j=0;j<temperaturas_cuidades[i].length;j++){
        document.write("<td>" + temperaturas_cuidades[i][j] + "</td>")
    }
    document.write("</tr>")
}
document.write("</table>")
```

Este script resulta un poco más complejo que los vistos anteriormente. La primera acción consiste en escribir la cabecera de la tabla, es decir, la etiqueta `<TABLE>` junto con sus atributos. Con el primer bucle realizamos un recorrido a la primera dimensión del array y utilizamos la variable `i` para llevar la cuenta de la posición actual. Por cada iteración de este bucle escribimos una fila y para empezar la fila abrimos la etiqueta `<TR>`. Además, escribimos en una casilla el numero de la ciudad que estamos recorriendo en ese momento. Posteriormente ponemos otro bucle que va recorriendo cada una de las casillas del array en su segunda dimensión y escribimos la temperatura de la ciudad actual en cada uno de los meses, dentro de su etiqueta `<TD>`. Una vez que acaba el segundo bucle se han impreso las tres temperaturas y por lo tanto la fila está terminada. El primer bucle continúa repitiéndose hasta que todas las ciudades están impresas y una vez terminado cerramos la tabla.

**Nota:** Habrás podido observar que en ocasiones generar código HTML desde Javascript se hace complejo. Pero el problema no es solo que el código sea difícil de producir, sino lo peor es que creas un código difícil de mantener, en el que se mezcla tanto la parte de la programación en Javascript con la parte de la presentación en HTML. Lo que has visto además es solo un código



bien simple, con una tabla realmente elemental, imagina qué pasaría cuando la tabla o los datos fueran más complejos. Afortunadamente, hay maneras de generar código HTML de salida mejores que las que hemos visto ahora, aunque resulta un poco avanzado para el momento en el que estamos. De todos modos, te dejamos un enlace al [manual del sistema de templates Javascript Handlebars](#), que es una alternativa de librería sencilla para generar salida en HTML desde Javascript.

Podemos [ver el ejemplo en marcha](#) y examinar el código del script entero.

## Inicialización de arrays

Para terminar con el tema de los arrays vamos a ver una manera de inicializar sus valores a la vez que lo declaramos, así podemos realizar de una manera más rápida el proceso de introducir valores en cada una de las posiciones del array.

El método normal de crear un array vimos que era a través del objeto Array, poniendo entre paréntesis el número de casillas del array o no poniendo nada, de modo que el array se crea sin ninguna posición. Para introducir valores a un array se hace igual, pero poniendo entre los paréntesis los valores con los que deseamos llenar las casillas separados por coma. Veámoslo con un ejemplo que crea un array con los nombres de los días de la semana.

```
var diasSemana = new Array("Lunes", "Martes", "Miércoles", "Jueves", "Viernes", "Sábado", "Domingo")
```

El array se crea con 7 casillas, de la 0 a la 6 y en cada casilla se escribe el dia de la semana correspondiente (Entre comillas porque es un texto).

Ahora vamos a ver algo más complicado, se trata de declarar el array bidimensional que utilizamos antes para las temperaturas de las ciudades en los meses en una sola línea, introduciendo los valores a la vez.

```
var temperaturas_cuidades = new Array(new Array (12,10,11), new Array(5,0,2),new Array(10,8,10))
```

En el ejemplo introducimos en cada casilla del array otro array que tiene como valores las temperaturas de una ciudad en cada mes.

Javascript todavía tiene una manera más resumida que la que acabamos de ver, que explicamos en el [primer artículo donde tratamos los arrays](#). Para ello simplemente escribimos entre corchetes los datos del array que estamos creando. Para acabar vamos a mostrar un ejemplo sobre cómo utilizar esta sintaxis para declarar arrays de más de una dimensión.

```
var arrayMuchasDimensiones = [1, ["hola", "que", "tal", ["estas", "estamos", "estoy"], ["bien", "mal"], "acabo"], 2, 5];
```



En este ejemplo hemos creado un array muy poco uniforme, porque tiene casillas con contenido de simples enteros y otras con contenido de cadena y otras que son otros arrays. Podríamos acceder a algunas de sus casillas y mostrar sus valores de esta manera:

```
alert (arrayMuchasDimensiones[0])
alert (arrayMuchasDimensiones[1][2])
alert (arrayMuchasDimensiones[1][3][1])
```

Con esto hemos cubierto lo básico que necesitas saber sobre Arrays, pero en el manual vamos a seguir viendo algunas cosas prácticas que puedes realizar con ellos, con nuevos ejemplos que creemos que te resultarán de utilidad para trabajar con ellos. Por ejemplo, en el siguiente artículo aprenderás a [recorrer los arrays con forEach](#).

Este artículo es obra de *Miguel Angel Alvarez*

Fue publicado / actualizado en 22/12/2001

Disponible online en <https://desarrolloweb.com/articulos/632.php>

## Recorridos forEach sobre Arrays de JavaScript

**Explicaciones y ejemplos del método forEach de Javascript que ofrecen una posibilidad de alto nivel para realizar o recorridos a los elementos de un array.**



En pasados artículos del [Manual de Javascript](#) hemos conocido la [estructura de array](#) y hemos visto que nos permite almacenar múltiples datos a los que accedemos mediante su índice numérico.

A lo largo de los diversos artículos se han planteado numerosos ejemplos para iterar sobre los elementos de los arrays con bucles de los típicos de toda la vida, como el bucle for. Sin embargo, existe una manera de realizar recorridos sobre arrays en Javascript que nos permite otras ventajas, como una mejor legibilidad del código. Así que vamos a ponernos manos a la obra y **conocer el método forEach de los arrays en Javascript para realizar recorridos a sus elementos.**



## Qué es `forEach`

JavaScript ofrece una variedad de métodos incorporados sobre los arrays que facilitan tareas frecuentes que podemos realizar con ellos, como encontrar un elemento, filtrar el array y mucho más.

El método `forEach` de los arrays se encarga de ejecutar una función específica para cada elemento de un array. Por tanto, **no podemos decir que sea exactamente un bucle**, sino un mecanismo de alto nivel para **realizar un comportamiento por cada uno de los elementos de un array**.

Al final, podemos entenderlo como un medio de recorrer los elementos del array y hacer algo para cada uno de ellos. Ese algo lo expresaremos mediante una función que enviaremos al bucle `forEach` como parámetro.

La sintaxis del método `forEach()` es la siguiente:

```
array.forEach(function(valorActual, indice, esteArray), thisValue)
```

### Parámetros admitidos por `forEach()`

La función `forEach()` acepta dos parámetros:

- Una función Callback: con el comportamiento que se ejecutará para cada elemento del array.
- Un segundo parámetro, opcional, que permite indicar el valor que tendrá la variable "this" dentro de la función.

Además, la función callback, que debemos enviar como primer parámetro de manera obligatoria, toma tres argumentos:

- `valorActual`: es el valor actual del elemento en el array en el recorrido
- `indice`: el índice del array que tiene el valor actual
- `esteArray`: el array completo sobre el que se está iterando

**La función callback se ejecutará una vez por cada elemento del array.** En ella no se requiere que se indiquen todos los parámetros, sino solamente los que necesitemos. Lo general es usar al menos el primer parámetro porque usualmente querremos hacer algo con el elemento actual, pero muchas veces necesitamos otros datos como por ejemplo el índice, que también nos proporciona.

### Ejemplos de recorrido `forEach` en Javascript

Vamos a ver un ejemplo sobre cómo se usa el método `forEach` sobre un array. Supongamos que tenemos un array de números y queremos imprimir cada número en la consola, algo muy sencillo. Aquí es cómo lo haríamos con `forEach`:



```
let numeros = [10, 20, 30, 40, 50];
numeros.forEach(function(numero) {
    console.log(numero);
});
```

En este caso, `forEach` está recorriendo cada elemento del array `numeros` y está pasando cada uno de los valores a la función callback. La función callback se ejecutará tantas veces como las necesarias hasta acabar el array, haciendo la impresión de su valor en la consola.

Ahora vamos a ver un ejemplo un poco más elaborado en el que queremos mostrar un listado de opciones en un supuesto menú con el índice que ocupa cada una en el array. Pero no queremos la opción empezando en cero como hacen los índices normales, sino empezando en uno.

Además, para este ejemplo vamos a usar una [función flecha de Javascript](#), para que veamos que también podemos alimentar el método `forEach` con este tipo de funciones, lo que nos dará como resultado un código bastante más compacto:

```
let opciones = ['Abrir', 'Guardar', 'Cerrar'];
console.log('Elige una opción');
opciones.forEach( (opcion, index) => console.log(`#${index + 1}: ${opcion}`));
```

Por supuesto, podemos tener arrays más complejos, donde cada uno de sus elementos sea un objeto y los podemos recorrer fácilmente haciendo cosas con cada uno de esos objetos.

Supongamos que tenemos un array de objetos, donde cada objeto representa un país con las propiedades: `nombre`, `capital` y `poblacion`, para ver un recorrido `forEach` que nos permita imprimir la información de cada país en la consola:

```
let paises = [
    { nombre: 'España', capital: 'Madrid', poblacion: 48000000 },
    { nombre: 'Francia', capital: 'París', poblacion: 70000000 },
    { nombre: 'Alemania', capital: 'Berlín', poblacion: 40000000 },
];

paises.forEach(function(country) {
    console.log(`El país ${country.nombre} tiene como capital ${country.capital} y población es de ${country.poblacion}`);
});
```

Este código imprimirá una línea para cada uno de los países en la consola. En esa funciona mostraremos el nombre del país, su capital y su población. La función `forEach` llama a la función callback una vez para cada objeto que hay en el array, que encontrarnos en el parámetro `país`.

## Beneficios de usar `forEach`

Ahora vamos a ver algunas de las ventajas que encuentras al usar `forEach` sobre otros métodos para recorrer un array:

- **Claridad y legibilidad:** El uso de `forEach` puede hacer que el código sea más claro y fácil de leer. Es un poco más semántico, ya que se expresa con el nombre de un método y nos



dispensa de especificar variables locales para tener que usarlas de índices para poder acceder a los elementos del array.

- **Evita errores comunes:** `forEach` maneja automáticamente la iteración sobre el array, por lo que es imposible equivocarse al manejar los índices o cometer fallos que nos lleven a meternos en un bucle infinito.
- **Encapsulamiento de lógica:** `forEach` permite encapsular la lógica de procesamiento de los elementos del array en una función. Habitualmente esa función solo la queremos expresar una única vez, por lo que usamos muy comúnmente funciones anónimas, pero nada nos impide que esa lógica la tengamos en una función con nombre que podemos usar en repetidas ocasiones, o incluso suministrar esa función con un método de un objeto.

### Algunas limitaciones del método `forEach` para iteraciones sobre arrays

Aunque `forEach` es muy útil y práctico, también queremos señalar algunas limitaciones a la hora de usarlo en los programas.

- Este modelo de bucle **no se puede interrumpir**: como sabes, existe la instrucción `break` que podemos ejecutar dentro de un bucle tradicional, como `for` o `while`, sin embargo, esta instrucción no la puedes usar para interrumpir un bucle `forEach`.
- **No encaja con funciones asíncronas:** `forEach` no espera a que las promesas se resuelvan antes de pasar al siguiente elemento del array, por lo que no podemos usarlo si para cada uno de los elementos necesitamos esperar a que se realice un comportamiento asíncrono. En este caso sería más adecuado pensar en usar el método `map` creando todos los comportamientos en un [Promise.all](#), o usar un bucle `for...of` con `async / await`.

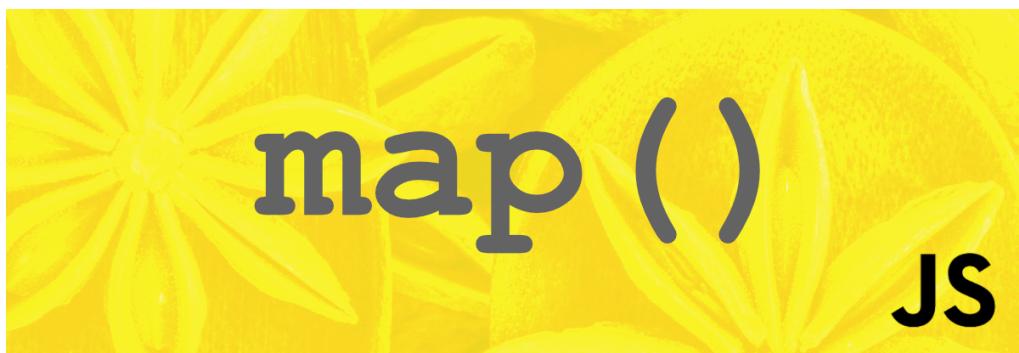
Este artículo es obra de *Miguel Angel Alvarez*

Fue publicado / actualizado en *06/07/2023*

Disponible online en <https://desarrolloweb.com/articulos/recorridos-foreach-array-javascript>

## Método map de los arrays en Javascript

Qué es el método `map()` de Javascript que permite generar un array a partir de otro array. Ejemplos de uso de la función `map` de los arrays.



En Javascript existen varios **métodos de arrays** muy útiles para realizar diversas operaciones repetitivas con los elementos del array. En este artículo vamos a explicar la función `map()` que nos sirve para **construir un array de elementos a partir de otro array**.

### Para qué utilizamos `map()` sobre los arrays

La función `map()`, aunque deberíamos llamarla **método** porque pertenecen a los arrays que en Javascript se consideran objetos, sirve para generar arrays a partir de los valores de otro array.

Al usar `map()` se realiza una **iteración por cada uno de los elementos de un array**. Para cada iteración se ejecuta una función, que devolverá un valor. **Con los valores devueltos se crea un nuevo array**, que es el que devuelve el método `map()`. Ahora lo veremos con código y será más claro.

Este método lo puedes usar para construir un array realizando una transformación de los elementos que tenemos en otro array. Esta transformación puede ser todo lo complicada que quieras o necesites, por lo que realmente los casos de uso pueden ser tan variados como la imaginación, o la necesidad, nos lleve. En este artículo veremos varios ejemplos para que puedas entender hasta qué punto las posibilidades pueden variar.

### Ejemplo sencillo de `map()`

Vamos a comenzar por un ejemplo sencillo con `map()` para ver cómo funciona.

Tenemos un array con valores numéricos y queremos construir un segundo array transformado que contenga el doble de los valores numéricos del array original. Para ello lo más cómodo es usar `map()`.

```
const numeros = [2, 7, 4, 6];
const doble = numeros.map( valor => valor * 2);
```

Como puedes ver, `numeros.map()` se alimenta con una función (hemos usado una función flecha que es lo más habitual en este tipo de códigos). La función que entregamos a `map()` se ejecutará una vez por cada uno de los elementos del array. En cada iteración recibirá el valor del elemento actual y devolverá el valor transformado (en este caso la multiplicación por 2 del valor original).



Si no conoces la sintaxis de este tipo de funciones resumidas y sus posibles variantes, te recomendamos el artículo dedicado a las [funciones flecha](#).

El array resultante que tendríamos en la variable doble sería el siguiente:

```
[ 4, 14, 8, 12 ]
```

Hemos visto que en una línea de código hemos realizado muchas cosas y esta es una de las ventajas de map. Si no tuviéramos `map()` el código tendría que haber sido algo como esto:

```
const numeros = [2, 7, 4, 6];
const doble = [];
for(let i = 0; i < numeros.length; i++) {
    doble.push(numeros[i] * 2)
}
```

A la vista de los dos códigos anteriores, y sabiendo que son equivalentes, espero que se pueda apreciar la potencia de este método de Javascript.

El método `map()` forma parte de una nueva hornada de métodos Javascript que nos permiten realizar prácticas que vienen de la llamada "programación funcional", que tiene diversas ventajas a la hora de realizar ciertos procesos de una manera ágil y elegante.

## Ejemplo de transformación de URLs en enlaces

Como decíamos, el método `map()` lo podemos usar para todo tipo de transformaciones, tan obvias o imaginativas como lo que se te pueda llegar a ocurrir.

En este segundo ejemplo vamos a partir de un array de cadenas que serán URLs y por medio de ese array de cadenas queremos producir el HTML de varios enlaces.

Comenzamos con este array de origen:

```
const urls = [
    'https://google.com',
    'https://intel.com',
    'https://oracle.com'
];
```

Ahora vamos a crear un array de enlaces HTML, gracias a `map()`.

```
const enlacesHTML = urls.map((url) => `<a href="${url}">${url}</a>`);
```



Una vez ejecutado este método tendríamos como resultado el siguiente array:

```
[  
  '<a href="https://google.com">https://google.com</a>',  
  '<a href="https://intel.com">https://intel.com</a>',  
  '<a href="https://oracle.com">https://oracle.com</a>'  
]
```

## Recibir el índice del elemento actual con map()

No lo habíamos dicho todavía, pero el método `map()` también puede recibir el índice del elemento actual, por si lo necesitas para alguna cosa. Para ello podemos usar un segundo parámetro que colocamos en la función entregada al método `map()`.

Este ejemplo no es demasiado útil pero podríamos generar un array con los números del mes, a partir de un array de meses, usando el índice de cada mes.

```
const meses = ['Enero', 'Febrero', 'Marzo', 'Abril', 'Mayo', 'Junio', 'Julio', 'Agosto', 'Septiembre', 'Octubre',  
  'Noviembre', 'Diciembre'];  
const numerosMes = meses.map((mes, index) => index + 1);
```

Al ejecutarse ese código el array `numerosMes` contendrá el siguiente conjunto de valores:

```
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]
```

Y por supuesto, si fuera necesario, la función que se ejecuta en cada iteración podría ser mucho más compleja, incluso requerir procesamientos que se realicen en varias líneas de código.

Solo para que sirva de ejemplo, vamos a ver ahora cómo crear un array de meses donde los meses pares aparezcan en mayúsculas.

```
const mesesParesEnMayusculas = meses.map((mes, index) => {  
  if(index % 2 != 0) {  
    return mes.toUpperCase();  
  }  
  return mes;  
});
```

El array resultante tendría este contenido:

```
[  
  'Enero',      'FEBRERO',  
  'Marzo',      'ABRIL',  
  'Mayo',       'JUNIO',  
  'Julio',       'AGOSTO',  
  'Septiembre', 'OCTUBRE',  
  'Noviembre',  'DICIEMBRE'  
]
```

Ahora vamos a ver otro ejemplo, por hacer algo que pudiera ser más útil y a la vez un poco más sofisticado. En este caso, entre todos los meses del array queremos colocar solamente en

mayúsculas el mes actual.

```
const mesesActualEnMayusculas = meses.map((mes, index) => {
    let numeroDeMes = new Date().getMonth();
    if(index == numeroDeMes) {
        return mes.toUpperCase();
    }
    return mes;
});
```

En este caso hemos usado unas sencillas funciones para el control de fechas. Si no las conoces te sugiero consultar el artículo sobre la [clase Date de Javascript](#).

## Conclusión

Hemos aprendido a trabajar con el método `map()` de los arrays. Hemos visto lo potente que resulta para hacer todo tipo de transformaciones, para generar arrays a partir del contenido de otros arrays.

Numerosas herramientas y frameworks usan este método map para producir HTML a partir del contenido de arrays en sistemas de templates. Esto es algo bastante más avanzado, pero te dejo un enlace por si quieras ver `map()` en uso para realizar [repeticiones en los templates de la librería Lit](#).

Este artículo es obra de *Miguel Angel Alvarez*

Fue publicado / actualizado en *10/03/2023*

Disponible online en <https://desarrolloweb.com/articulos/metodo-map-arrays-javascript>

## Ordenación de arrays en Javascript con array.sort()

Aprende a ordenar arrays en Javascript, con varios ejemplos de ordenación en los que usamos el método `sort()` disponible de manera nativa en los arrays.





En Javascript existe un mecanismo de ordenación de arrays muy potente y totalmente personalizable. Usa un método llamado `sort()`, disponible en los arrays, que recibe una función que implementa la lógica de ordenación de los elementos.

Es un método muy potente, ya que mediante él podemos realizar todo tipo de órdenes, clásicos como el alfabético, mayor a menor, menor a mayor, pero también cualquier orden arbitrario sobre elementos de cualquier tipo.

En este artículo vamos a aprender a usar el método `sort()` para ordenar los arrays con varios ejemplos prácticos que te aclararán cualquier posible duda.

### Método sort de los arrays

El método `sort()` está disponible en los arrays a partir de ECMAScript 5, es decir, está disponible en absolutamente todos los navegadores, incluso los viejos Internet Explorer. Por lo tanto lo puedes usar con total confianza.

Básicamente es un método que recibe una función callback y ejecuta esa función con todos los elementos del array, un número indeterminado de veces, con combinaciones entre ellos, de modo que pueda ponderar el orden, hasta que queden ordenados completamente sus elementos.

Esa función callback que se ejecuta es invocada con dos parámetros, que toman los valores de los elementos que se están comparando en cada ocasión.

```
miarray.sort( function(a, b) {  
    // Hacer lo que se necesite para comparar "a" y "b" y devolver el resultado de la comparación  
});
```

Esos parámetros se tienen que comparar dentro de la función para saber si uno va delante del otro, o viceversa, o si simplemente son iguales. Esa comparación puede ser totalmente arbitraria, es decir, los puedes comparar por orden natural, pero también por cualquier criterio que necesites. Dependiendo del resultado de esa comparación se deben devolver valores numéricos, atendiendo a estas reglas:

- Si son iguales, devolveremos 0.
- Si "a" debe ir ordenado antes que "b", entonces devolvemos un número menor que 0.
- Si "a" debe ir ordenado después que "b", entonces devolvemos un número mayor que 0.

Es decir, nosotros somos dueños de escoger el criterio de comparación entre los elementos del array. Esos elementos aparecerán antes o después en el array ordenado dependiendo del valor que hemos devuelto al compararlos.

### Ejemplo de ordenación ascendente de números

Para entender bien cómo funciona `array.sort()` lo mejor es verlo con un ejemplo. En este primer caso veremos un ejemplo con un array de elementos numéricos, que deben ordenarse en orden ascendente.



```
let numbers = [7, 6, 44, 101, 55, 60, 82, 1, 57, 6];
numbers.sort((a, b) => {
  if(a == b) {
    return 0;
  }
  if(a < b) {
    return -1;
  }
  return 1;
});
```

Como ves, usamos sort() enviando la función de ordenación.

- Si a y b eran iguales, devolvemos 0.
- Si a es menor que b, devolvemos -1, que es un valor por debajo de cero.
- Al final devolvemos 1, porque querrá decir que no eran ni iguales ni a era menor que b.

Ten en cuenta que una vez se ejecuta el método sort() sobre el array, el propio array es el que cambia de orden. Es decir, después de llamar a esta función, "numbers" habrá cambiado y ahora sus elementos estarán ordenados.

## Ordenación descendente de números

Si quisieramos un orden descendente la lógica sería prácticamente la misma, simplemente cambiando la comparación del segundo if.

```
numbers.sort((a, b) => {
  if(a == b) {
    return 0;
  }
  if(a > b) {
    return -1;
  }
  return 1;
});
```

Ahora hemos dicho que, si a es mayor que b, entonces devolvemos -1, por lo que pondrá a a después de b.

## Ejemplo de orden alfabético de un array Javascript

Ahora vamos a ver cómo se ordenaría por orden alfabético. Prácticamente es el mismo ejemplo que el de ordenación de números, como puedes ver en el código.

```
let words = ["hola", "adiós", "gusta", "quiero", "ordenar", "arrays"];
words.sort((a, b) => {
  if (a == b) {
    return 0;
  }
  if (a < b) {
    return -1;
  }
  return 1;
});
```



Podríamos ejecutar luego un console.log() para ver el contenido del array words una vez ordenado.

```
console.log(words);
```

### Orden sin tener en cuenta mayúsculas y minúsculas en los elementos del array

Ten en cuenta que al hacer comparaciones en cadenas no es lo mismo que estén en mayúsculas o minúsculas. El orden hace que primero se colocasen todas las palabras en mayúsculas y luego las que están en minúsculas, por lo que, si quieras que el orden no tenga en cuenta si están en mayúsculas o minúsculas, tendrás que hacer un paso extra.

El paso extra consiste en convertir la palabra a todo en minúsculas, o mayúsculas, para luego hacer las comparaciones. El código podría ser como este:

```
let wordsUppercased = ["hola", "Adiós", "gusta", "Quiero", "ordenar", "arrays"];
wordsUppercased.sort((a, b) => {
  a = a.toLowerCase();
  b = b.toLowerCase();
  if (a == b) {
    return 0;
  }
  if (a < b) {
    return -1;
  }
  return 1;
});
console.log(wordsUppercased);
```

### Ordenes totalmente arbitrarios con sort()

Ten en cuenta que las comparaciones de orden pueden ser totalmente arbitrarias, es decir, eres dueño y señor de cómo se van a comparar. Por ello array.sort() es tremadamente potente, porque podrías conseguir casi cualquier tipo de orden.

En este ejemplo me da igual cómo queden ordenados los valores del array. Simplemente quiero que los números que sean igual o superiores a 1000 se pongan delante y los que sean menores que 1000, que aparezcan detrás.

```
let numbersGreaterThan1000 = [700, 6000, 44, 1031, 55, 60, 8200, 1001, 57, 6];
numbersGreaterThan1000.sort((a, b) => {
  if (a <= 1000) {
    return 1;
  }
  return -1;
});
console.log(numbersGreaterThan1000);
```

Simplemente estoy viendo si el elemento "a" de la comparación es menor que 1000. Entonces ese elemento irá después, por lo que devuelvo 1. En caso devuelvo -1 para que se ponga delante.



Realmente no sé si será el mecanismo más inteligente para hacer este tipo de orden, porque no he llegado a analizar el número de pasadas que se hace para conseguir ordenar el array. Con un algoritmo creado a mano seguramente se consiga hacer más rápido, porque en principio con una pasada para verificar los elementos de uno en uno sería suficiente. Tómalo como un ejemplo tonto para que veamos que las posibilidades de ordenación son muy amplias.

## Ordenar arrays de objetos por los valores de las propiedades de los objetos

Lo bueno de usar `array.sort()` es que somos capaces de ordenar elementos de arrays que sean de cualquier tipo, incluso de tipos complejos como podrían ser los objetos.

En este ejemplo vamos a realizar un orden de elementos del array que son objetos de tipo "persona". Cada una de estas personas tiene una edad y un nombre. Nuestro algoritmo quiere conseguir un orden de los elementos en el array que tome en cuenta la edad. Si dos personas tienen la misma edad, entonces los ordenará por orden alfabético de nombres.

```
let people = [
  {
    name: 'Julia',
    age: 10,
  },
  {
    name: 'Miguel',
    age: 45,
  },
  {
    name: 'Juan',
    age: 24,
  },
  {
    name: 'Maria',
    age: 60,
  },
  {
    name: 'Alfredo',
    age: 45,
  },
  {
    name: 'Alba',
    age: 10,
  },
];
people.sort( (a, b) => {
  if(a.age < b.age) {
    return -1;
  }
  if(a.age > b.age) {
    return 1;
  }
  if (a.name.toLowerCase() < b.name.toLowerCase()) {
    return -1;
  }
  if (a.name.toLowerCase() > b.name.toLowerCase()) {
    return 1;
  }
  return 0;
}
```



```
});  
console.log(people);
```

Como puedes ver, vamos devolviendo 0, 1 o -1 dependiendo primero de la edad. Si la edad no es ni mayor ni menor (es igual) entonces lo que hacemos es comprar el nombre de las personas.

Con este ejemplo habrás podido comprobar que podemos ordenar cualquier cosa y el algoritmo puede tener en cuenta el criterio que se considere oportuno en cada situación.

Espero que hayas podido aprender bastante y hayan quedado claras todas las posibilidades de ordenación nativa de elementos de arrays en Javascript.

## Videotutorial de ordenación de arrays Javascript

Acabamos este artículo con un videotutorial en el que vemos en vivo cómo se realizan estos ejemplos prácticos para ordenar arrays.

Para ver este vídeo es necesario visitar el artículo original en:

<https://desarrolloweb.com/articulos/ordenacion-arrays-javascript-sort>

Este artículo es obra de *Miguel Angel Alvarez*

Fue publicado / actualizado en *14/05/2021*

Disponible online en <https://desarrolloweb.com/articulos/ordenacion-arrays-javascript-sort>



# Objetos en Javascript

En los siguientes artículos vamos a abordar el mundo de los objetos en Javascript. Serán esenciales para cualquier persona que está comenzando en la programación en general, ya que se tratarán conceptos muy recurrentes en cualquier lenguaje. Además trataremos con detalle las particularidades de los objetos en el lenguaje Javascript, ya que son bastante distintas en relación a otros lenguajes más tradicionales.

## Introducción general a los objetos en Javascript

### Breve introducción al mundo de los objetos, en programación en general, y a las particularidades del uso de objetos en el lenguaje Javascript.

Vamos a introducirnos en un tema muy importante de Javascript como son los objetos. Es un tema que aun no hemos visto y sobre el que en adelante vamos a tratar constantemente pues la mayoría de las cosas en Javascript, incluso las más sencillas, las vamos a realizar a través del manejo de objetos. De hecho, en los ejemplos realizados hasta ahora hemos hecho grandes esfuerzos para no utilizar objetos y aun así los hemos utilizado en alguna ocasión, pues es muy difícil encontrar ejemplos en Javascript que, aunque sean simples, no hagan uso de ellos.

La programación orientada a objetos (POO) representa una nueva manera de pensar a la hora de hacer un programa. Javascript no es un lenguaje de programación orientado a objetos puro porque, aunque utiliza objetos en muchas ocasiones, no necesitamos programar todos nuestros programas en base a ellos. De hecho, lo que vamos a hacer generalmente con Javascript es **usar objetos** y no tanto programar orientado a objetos. Por ello, la manera de programar no va a cambiar mucho con respecto a lo que hemos visto hasta ahora en el [Manual de Javascript](#). En resumen, lo que hemos visto hasta aquí relativo a sintaxis, funciones, etc. sigue siendo perfectamente válido y puede ser utilizado igual que se ha indicado. Solo vamos a aprender una especie de estructura nueva como son los objetos.

**Nota:** Para empezar a empaparnos un poco sobre los objetos tenemos un [pequeño artículo publicado en DesarrolloWeb sobre la programación orientada a objetos](#). Sería muy recomendable que lo leyeras, porque se explican varios conceptos en los cuales no vamos a entrar con tanto detalle. Si conoces ya la POO continúa leyendo sin pausa, pero si deseas profundizar recuerda que tenemos también un [Manual completo de Orientación a objetos](#). Si te gusta ver vídeos te recomendamos también la clase [Qué son los objetos](#) impartida en el [Curso de Programación en vídeo](#).

## Qué es un objeto

Aunque no vamos a entrar en detalle con los concetos, pues se encuentran muy bien explicados en referencias que ya hemos indicado, los objetos son una herramienta de lenguajes de programación en la que se unen dos cosas fundamentales: los datos y la funcionalidad. Todo programa informático trata básicamente esas dos cosas de alguna manera. Con lo que hemos visto hasta ahora los datos los teníamos en variables y la funcionalidad en funciones ¿no es así? pues en el mundo de los objetos, tanto datos como funcionalidad están en la misma estructura, el objeto.

El asunto es que ahora necesitas aprender nuevos nombres con los que referirte a los datos y funcionalidad agrupados en un objeto:

- **Propiedades:** En los objetos las propiedades se refieren a los datos
- **Métodos:** En objetos, los métodos se refieren a la funcionalidad

Imagina que tienes un objeto botón (un botón del navegador, algo que puedes pulsar para realizar una acción). El botón tiene un texto escrito, pues ese texto sería un dato y por lo tanto le llamaríamos propiedad. Otra propiedad de un botón sería si está o no activado. Por otra parte, un botón podría tener funcionalidad asociada, que estaría en un método, como procesar la acción de un clic. Imagina algo más genérico como un teléfono. El teléfono puede tener propiedades como la marca, modelo, sistema operativo y métodos como encender, apagar, llamar a un número, etc.

En lenguajes de programación orientados a objetos puros, como puede ser Java, tienes que programar siempre en base a objetos. Para programar tendrías que crear "clases", que son una especie de "moldes" a partir de los cuales se crean objetos. El programa resolvería cualquier necesidad mediante la creación de objetos en base a esos moldes (clases), existiendo varios (decenas, cientos o miles) de objetos de diversas clases. Los objetos tendrían que colaborar entre sí para resolver cualquier tipo de acción, igual que en sistemas como un avión existen diversos objetos (el motor, hélices, mandos...) que colaboran entre sí para resolver la necesidad de llevar pasajeros o mercancía en viajes aéreos.

Sin embargo, como veníamos diciendo, en Javascript no es tanto programar orientado a objetos, sino usar objetos. Muchas veces serán objetos ya creados por el propio navegador (la ventana del navegador, un documento HTML que se está visualizando, una imagen o un formulario dentro de ese documento HTML, etc), y otras veces serán objetos creados por ti mismo o por otros desarrolladores que te sirven para hacer cosas específicas. Por tanto, lo que nos interesa saber para comenzar es la sintaxis que necesitas para usar los objetos, básicamente acceder a sus propiedades y ejecutar sus métodos.

**Nota:** Para conocer cuáles son los objetos del navegador, que tenemos a disposición en Javascript para resolver las necesidades de las páginas web, tienes que leer el manual sobre [trabajo con Javascript para uso y manipulación de los recursos del navegador](#).



## Cómo acceder a propiedades y métodos de los objetos

En Javascript podemos acceder a las propiedades y métodos de objetos de forma similar a como se hace en otros lenguajes de programación, con el operador punto (".").

Las propiedades se acceden colocando el nombre del objeto seguido de un punto y el nombre de la propiedad que se desea acceder. De esta manera:

```
miObjeto.miPropiedad
```

Para llamar a los métodos utilizamos una sintaxis similar, pero poniendo al final entre paréntesis los parámetros que pasamos a los métodos. Del siguiente modo:

```
miObjeto.miMetodo(parametro1,parametro2)
```

Si el método no recibe parámetros colocamos los paréntesis también, pero sin nada dentro.

```
miObjeto.miMetodo()
```

## Cómo crear objetos

Como hemos dicho, la mayoría de los objetos con los que vas a trabajar en Javascript para poder crear interacción, efectos y comportamientos diversos en páginas web, te los dan ya hechos. El propio navegador te los ofrece para que tú simplemente los tengas que usar. Eso es material de estudio del [Manual de Javascript y los objetos del navegador](#). Aclarado ese punto hay que advertir que Javascript es un tanto particular a la hora de crear objetos, básicamente porque tradicionalmente no existe el concepto de "clase".

Para ser más exactos, en Javascript, ES5, las clases se crean por medio de funciones y con el operador new creas objetos a partir de esas funciones, pero no existen las clases como las que conocemos en otros lenguajes más tradicionales.

**Nota:** Ahora en ES6 ya existen las clases y Javascript es capaz de generar clases y a partir de ellas producir objetos, como otros lenguajes. Obtienes más información en el [Manual de ES6](#).

La otra alternativa para crear objetos en Javascript es por medio de literales de objeto, que no son más que la definición del objeto por medio de código encerrado entre llaves, indicando sus propiedades o métodos tal cual.

```
{
  nombre: 'Miguel Angel Alvarez',
  sitioWeb: 'DesarrolloWeb.com'
}
```



En el código anterior solo hemos definido propiedades, pero tenemos otros artículos donde podrás ver cómo definir métodos también. Para saber más sobre los literales de objetos te recomendamos la lectura del artículo sobre [Literales de objetos Javascript](#), donde vamos a explicarte más detenidamente esta sintaxis habitual de creación de objetos en este lenguaje.

En Javascript tradicional hemos dicho que no existen las clases, pero podremos crear instancias de objetos a partir de funciones, como veremos en el siguiente punto.

### Crear e instanciar objetos a partir de funciones

Para quien no lo sepa, instanciar un objeto es la acción de crear un ejemplar de una clase, para poder trabajar con él luego. La clase es la definición de las características y funcionalidades de un objeto. Con las clases no se trabaja directamente, éstas sólo son definiciones. Para trabajar con una clase debemos tener un objeto instanciado de esa clase. Recordamos que en Javascript no existen clases, pero podemos usar funciones.

Esta simple función podríamos usarla como molde para construir objetos de la clase Persona:

```
function Persona(nombre) {  
    this.nombre = nombre;  
}
```

Observarás que estamos usando dentro la palabra "this". Esa palabra es una referencia al objeto que se va a crear con esta función. En javascript para crear un objeto a partir de una función se utiliza la instrucción new, de esta manera.

```
var miguel = new Persona('Miguel Angel Alvarez');
```

En una variable que llamamos "miguel" asigno un nuevo (new) ejemplar de la clase Persona. Los paréntesis se rellenan con los datos que necesite la clase para inicializar el objeto, si no hay que meter ningún parámetro los paréntesis se colocan vacíos. En realidad lo que se hace cuando se crea un objeto es llamar a la función que lo construye y la propia función se encargará de inicializar las propiedades del objeto (para lo que usa la referencia "this").

Ciertamente, si los conceptos de programación orientada a objetos son nuevos para ti, quizás muchos puntos de este artículo se quedarán un poco complejos. No queremos que te asustes, puesto que volveremos sobre todo esto en futuros artículos y lo podrás ir entendiendo mejor. En concreto, esta parte de la creación de objetos a partir de funciones está explicada en el artículo [Creación de clases en Javascript con ES5](#). Recuerda también aclarar tus dudas más teóricas sobre clases y objetos en el [Manual de la teoría de la programación orientada a objetos](#).

Este artículo es obra de *Miguel Angel Alvarez*

Fue publicado / actualizado en 19/12/2016



Disponible online en <https://desarrolloweb.com/articulos/introduccion-objetos-javascript.html>

## Literales de objeto en Javascript

**Cómo podemos crear objetos en Javascript a partir de un literal, operaciones típicas que puedes realizar con esos objetos.**

Voy a comenzar una serie de artículos que me sirven como apuntes de programación orientada a objetos en Javascript. Este lenguaje es bastante particular en este sentido y en ocasiones te permite hacer cosas que en otros lenguajes serían impensables. También al contrario, algunas cosas básicas de lenguajes tradicionales en Programación Orientada a Objetos no existen en Javascript.

Javascript no tiene clases como tal, por lo menos hasta que llegue la nueva actualización Ecma6 prevista para 2015, pero sí que tiene objetos. Además tiene algo que no todos los lenguajes poseen, un mecanismo para la creación de objetos a partir de lo que conocemos como "literal".



**Nota:** Para los que no conozcan el término "literal", cabe decir que es una palabra que indica algo escrito de manera "literal". Aunque sea muy feo usar la misma palabra para definir algo, me lo vas a permitir porque es la mejor que existe. Un literal es algo extremadamente sencillo de entender con un ejemplo.

```
var x = "hola";
```

En esa línea de código "hola" es un literal. En concreto decimos que es un "literal de cadena".

Ahora mira esta otra línea:

```
y +=5;
```

Cuando ves un número escrito tal cual en tu código decimos que es un literal numérico.

Podemos crear un objeto en Javascript asignando un valor literal de objeto en una variable. Eso se consigue colocando dicho literal entre llaves y dentro de ellas tantas propiedades o métodos con



pares "clave/valor", por medio de una sintaxis como esta:

```
var dimensiones = {  
    altura: 34,  
    anchura: 455  
}
```

Como estás viendo, tenemos una variable dimensiones. Al asignarle un literal objeto estamos realmente asignando una referencia a un objeto en la memoria creado con las propiedades que acabamos de asignar.

Las propiedades se separan por comas y se coloca siempre el nombre de la propiedad, el carácter ":" y luego el valor de la propiedad.

Por supuesto, también podemos asignar métodos a nuestros objetos literales.

```
var dimensiones = {  
    altura: 34,  
    anchura: 455,  
    area: function(){  
        return this.altura * this.anchura;  
    }  
}
```

Como ves, los métodos en Javascript simplemente son propiedades a los que les asignas una función. Dentro del código de tus métodos puedes acceder a las propiedades del objeto a través de la variable this.

Una vez creado ese objeto, puedes usar la notación punto para acceder a sus propiedades (o métodos).

```
dimensiones.altura = 90 //accede a la propiedad altura y le asigna el valor 90
```

**Nota:** Los literales de objeto no son la única manera de crear objetos en Javascript. Además existe un manera de definir algo parecido a una clase, pero no es exactamente lo que conocemos en la programación orientada a objetos tradicional y no podríamos llegar a considerarla tal. En lugar de ello podemos crear funciones que, al invocarlas con la palabra "new" te crean nuevos objetos inicializados con esa función. Técnicamente, en vez de definir clases, en Javascript definimos funciones constructoras de objetos. En definitiva, una implementación muy particular que a veces nos puede liar si estamos familiarizados con lenguajes de enfoque más tradicional (en lo que respecta a OOP "Object Oriented Programming") como Java o PHP. Esta es una discusión interesante, que podrías complementar con el artículo dedicado a crear [clases en Javascript](#), pero que no es la que nos ocupa en esta ocasión.

## ¿Esto no es JSON?



El formato de intercambio de datos JSON se ha popularizado mucho y quizás lo conozcas antes de conocer estos detalles sobre los literales de objeto Javascript. En ese caso puede que te hayas dado cuenta que los literales de objeto Javascript no son más que estructuras JSON. Si es así permíteme apuntar que realmente tendríamos que darle la vuelta a esa frase y decir que la notación JSON utiliza la sintaxis de los literales de objeto Javascript como formato.

Así pues, lo que conozcas de JSON lo puedes aplicar al mundo de Javascript inmediatamente. De hecho, si has tenido ocasión de trabajar con este formato desde Javascript, habrás comprobado que en la mayoría de los casos puedes volcar un objeto JSON en una variable Javascript e inmediatamente trabajar con él como si fuera un objeto que tienes en la memoria.

**Nota:** Puedes saber más de JSON en el artículo [Qué es JSON](#).

## Crear nuevas propiedades y métodos sobre objetos creados

Javascript es muy permisivo y nos deja hacer cosas que producirían errores en otros lenguajes. Es el caso de la asignación de valores a propiedades que no han sido creadas previamente.

Tengo mi objeto coche:

```
var coche = {  
    color: "rojo",  
    marca: "Opel"  
}
```

Ahora podría crear nuevas propiedades en ese objeto asignando valores a las propiedades que no existían previamente.

```
coche.anoFabricacion = 2014;
```

Los métodos los creas asignando funciones:

```
coche.arrancar = function(){  
    alert("rum rum");  
}
```

Los métodos y funciones que acabamos de crear son tan válidos como los que ya estuvieran en mi objeto cuando fue definido por medio de su literal. Podré acceder a sus elementos con la notación punto, que ya conoces.

```
console.log(coche.color);  
console.log(coche.anoFabricacion);  
coche.arrancar();
```

## Ejemplo para practicar con literales de objeto



En el siguiente código ponemos en marcha los nuevos conocimientos que has adquirido sobre literales de objeto en Javascript.

```
var miObjeto = {  
    propiedad1: "Algo",  
    propiedad2: "Otra cosa",  
    propiedad3: true,  
    propiedad4: 344,  
  
    metodo1: function(){  
        alert("Ejecutaste metodo1");  
    },  
    metodo2: funcionMetodo2  
}  
  
function funcionMetodo2(){  
    //puedo usar la variable this para acceder a mis propias propiedades o métodos  
    this.propiedad2 = "Esto lo he modificado desde el método metodo2";  
}  
  
//Veamos el valor de esas propiedades  
console.log(miObjeto.propiedad1);  
console.log(miObjeto.propiedad2);  
  
//ejecutemos algún método  
miObjeto.metodo1();  
miObjeto.metodo2();  
  
//creamos nuevas propiedades  
miObjeto.otraPropiedad = "Esto está creado a posteriori";  
  
//creamos nuevos métodos  
miObjeto.otroMetodo = function(){  
    console.log("ejecutaste otro método");  
}  
  
//veamos el contenido de todo el objeto  
console.log(miObjeto);
```

Eso es todo por el momento, espero que te hayamos aclarado algo.

Este artículo es obra de *Miguel Angel Alvarez*  
Fue publicado / actualizado en *31/10/2014*  
Disponible online en <https://desarrolloweb.com/articulos/literales-objeto-javascript.html>

## for in en Javascript

**Bucle for in Javascript. Una alternativa al bucle for, para recorridos a propiedades de objetos en Javascript. Cómo iterar por las propiedades y los valores de las propiedades de un objeto, de manera genérica en Javascript con el bucle for .....**

# Bucle for...in JS

Bucle for in Javascript. Una alternativa al bucle for, para recorridos a propiedades de objetos en Javascript. Cómo iterar por las propiedades y los valores de las propiedades de un objeto, de manera genérica en Javascript con el bucle for ... in.

En Javascript no existen arrays asociativos y como sabes, éstos son siempre buenos aliados como recursos para la programación. Si queremos usar algo parecido a un array asociativo tendremos que utilizar las construcciones de objetos. De todos modos, con lo que nos ofrece el lenguaje somos capaces de realizar todas las cosas que en otros lenguajes haces con los arrays asociativos.

En este artículo pretendemos explicarte cómo realizar un recorrido genérico a todas las propiedades presentes en un objeto, con el bucle for in Javascript. Mediante esta estructura de control podremos obtener los nombres de las propiedades junto con sus valores. Ese recorrido es "genérico", es decir, es independiente del número de propiedades que se encuentre en el objeto que estamos recorriendo y es independiente también de sus nombres de propiedad o sus valores.

**Nota:** Por si no lo sabes, los arrays asociativos son aquellos que no tienen índices numéricos sino alfabéticos. Nos sirven para muchas cosas en los lenguajes de programación. Si no lo sabías, ya te darás cuenta. De momento entonces olvida la mención a los arrays asociativos y piensa solo en objetos y sus propiedades.

## Bucle for ... in

En Javascript hay una construcción especial del bucle for de toda la vida, que te permite recorrer todas las propiedades de un objeto. Es parecido a lo que en otros lenguajes tenemos en el bucle foreach (Javascript también tiene el forEach pero es solo para arrays y no es un bucle sino un método de arrays, que sirve para iterar, pero no es una estructura de control como tal). Su sintaxis es la siguiente:

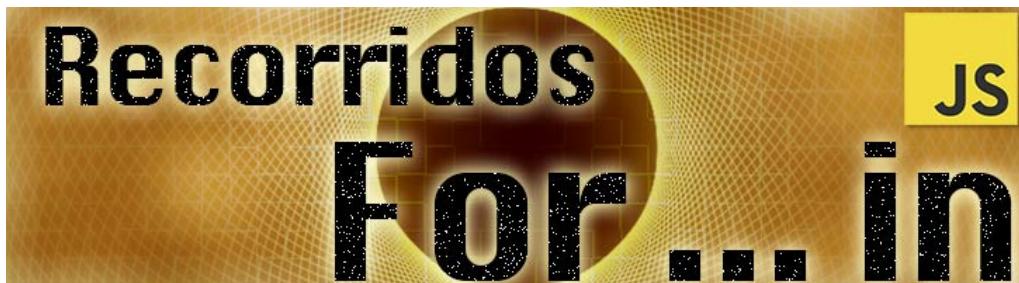
```
for (var propiedad in objeto){  
    // código a repetir por el bucle.  
    // dentro de este código la variable "propiedad" contiene la propiedad actual  
    // actual en cada uno de los pasos de la iteración.  
}
```



No hay mucho más que explicar sobre esta estructura del bucle for in, solo decir que de esta manera tal cual puedes acceder a los nombres de las propiedades del objeto. Gracias al bucle, el código que se incluye dentro del for se ejecutará una vez por cada una de las propiedades del objeto. Luego veremos algunos ejemplos de código para que quede suficientemente claro.

Habrás notado que hemos dicho que así podrás acceder a las propiedades. No te preocupes que para acceder a sus valores tendrás que usar un truquillo. Lo veremos también en este artículo, pero no nos adelantemos.

### Bucle para acceder a las propiedades de un objeto



Veamos una aplicación de este bucle for in con un ejemplo sencillo que nos arroje algo de luz.

Tenemos un objeto creado a partir de un literal de objeto.

```
var diasMes = {  
    enero: 31,  
    febrero: 28,  
    marzo: 31,  
    abril: 30,  
    mayo: 31  
}
```

Son los meses del año y los días que tiene ese mes. Es verdad que febrero puede tener otros días y que me faltan meses, pero a fines didácticos es suficiente. Observarás que esta estructura en resumen es muy parecida a lo que sería un array asociativo. Si los necesitamos en nuestro programa la podemos usar como si lo fuera.

**Nota:** Si no entiendes la definición de un objeto por medio de un literal, simplemente lee el [artículo sobre literales de objeto](#).

Pues para acceder a las propiedades de ese objeto (en este caso sería para acceder a los nombres de cada uno de los meses) usarías este bucle.

```
for (var mes in diasMes){  
    console.log(mes);  
}
```



Con esto conseguirás en la consola que aparezcan todos los nombres de los meses del año que tenemos en ese objeto diasMes.

## Acceder a los valores de las propiedades

Una situación común es que no quieras acceder a los nombres de las propiedades, sino a los valores. Para ello podemos usar un pequeño truco que nos permite el lenguaje. Esto no es algo en concreto del bucle for in Javascript, sino del tratamiento que hace este lenguaje a los objetos. Se trata de que podemos acceder a las propiedades de los objetos como si fueran arrays, indicando como índice del array una cadena con el nombre de la propiedad.

```
alert(diasMes["mayo"]);
```

Eso nos mostrará en una caja de alerta el valor 31 que es el que está asociado a la propiedad "mayo" del objeto "diasMes".

Te habrás dado cuenta que esta sintaxis para acceder a los valores de las propiedades es justamente igual a la sintaxis que se usa para acceder a valores de casillas de un array asociativo.

Bueno, pues sabiendo esto podrás acceder a los valores del objeto, uno por uno, con un bucle for ... in muy parecido al que teníamos antes.

```
for (var mes in diasMes){  
    console.log(diasMes[mes]);  
}
```

Seguro que ahora te parece sencillo. Es que realmente lo es.

## Todo junto, acceso a la propiedad y su valor

Esta parte ya seguro que ni te hace falta leerla. Simplemente quiero mostrar un mensaje en la consola más legible, accediendo en el mismo bucle a la propiedad y su valor.

```
for (var mes in diasMes){  
    console.log("El mes " + mes + " tiene " + diasMes[mes] + " días.");  
}
```

Otro ejemplo puedes verlo en el código a continuación. Vamos a hacer un recorrido a este objeto, que tiene en sus propiedades otros objetos, es decir, es un objeto de objetos. Es algo como un array, pero realmente no es un array, sino una colección. Cada uno de los objetos tiene un identificador, que es su nombre y como valor tiene un objeto con otras propiedades. Esta construcción es muy común en Firebase, donde no podemos guardar arrays en la base de datos.

```
{  
  "Miguel Angel Alvarez" : {  
    "isFav" : true  
  },  
  "DesarrolloWeb.com" : {  
    "isFav" : false  
}
```



```
},
"EscuelaIT" : {
  "isFav" : true
},
}
```

Ahora puedes ver el bucle for in Javascript, con el que realizamos el recorrido, comprobando para cada uno de los objetos de la colección si es favorito.

```
for(var usuario in usuarios) {
  if(usuarios[usuario].isFav) {
    console.log('El usuario con índice (usamos su nombre como índice) ' + usuario + ' es uno de tus favoritos');
  }
}
```

Eso es todo, con estos conocimientos ya no se te pueden escapar las posibilidades de recorridos a objetos y la construcción del útil bucle for in Javascript. Además, si necesitas arrays asociativos para construir tus programas sabrás la manera alternativa de implementarlos y usarlos, por medio de objetos, en Javascript. Seguro que te servirá de ayuda, incluso aunque ahora no sepas para qué.

### Videotutorial de recorridos a las propiedades de objetos

Acabamos este artículo recomendando este videotutorial, en el que verás un par de ejemplos de recorridos a las propiedades de objetos con el bucle for...in. Espero que te guste y pueda complementar bien lo que acabas de aprender.

Para ver este vídeo es necesario visitar el artículo original en:

<https://desarrolloweb.com/articulos/recorridos-propiedades-objetos-javascript-forin.html>

Este artículo es obra de *Miguel Angel Alvarez*

Fue publicado / actualizado en *17/05/2021*

Disponible online en <https://desarrolloweb.com/articulos/recorridos-propiedades-objetos-javascript-forin.html>

## Creación de clases en Javascript tradicional

**Ahora que ya sabemos lo que son los objetos, vamos a ver cómo podemos crear y usar nuestros propios objetos en Javascript tradicional, ES5, abordando diversos ejemplos prácticos.**

En capítulos anteriores de esta [segunda parte del Manual de Javascript](#) vimos las [generalidades sobre la programación orientada a objetos](#). Además, estuvimos dando un repaso bastante completo



a los distintos [objetos nativos de Javascript](#).

Ahora que ya hemos conocido un poco los objetos y hemos aprendido a manejarlos podemos pasar a un tema más avanzado, como es el de construir nuestros propios objetos, que puede sernos útil en ciertas ocasiones para temas avanzados.

Así que vamos a ver cómo podemos definir nuestros propios objetos, con sus propiedades y métodos, de manera que aprendamos el mecanismo, pero sin entrar demasiado en aspectos prácticos que los dejamos para ejemplos del futuro.

Para crear nuestros propios objetos debemos crear una clase, que recordamos que es algo así como la definición de un objeto con sus propiedades y métodos. Para crear la clase en Javascript debemos escribir una función especial, que se encargará de construir el objeto en memoria e inicializarlo. Esta función se le llama constructor en la terminología de la programación orientada a objetos.

```
function MiClase (valor_inicializacion){  
    //Inicializo las propiedades y métodos  
    this.miPropiedad = valor_inicializacion  
    this.miMetodo = nombre_de_una_funcion_definida  
}
```

Eso era un constructor. Utiliza la palabra `this` para declarar las propiedades y métodos del objeto que se está construyendo. This hace referencia al objeto que se está construyendo, pues recordemos que a esta función la llamaremos para construir un objeto. A ese objeto que se está construyendo le vamos asignando valores en sus propiedades y también le vamos asignando nombres de funciones definidas para sus métodos. Al construir un objeto técnicamente es lo mismo declarar una propiedad o un método, solo difiere en que a una propiedad le asignamos un valor y a un método le asignamos una función.

## La clase AlumnoUniversitario

Lo veremos todo más detenidamente si hacemos un ejemplo. En este ejemplo vamos a crear un objeto estudiante universitario. Como estudiante tendrá unas características como el nombre, la edad o el número de matrícula. Además podrá tener algún método como por ejemplo matricular al alumno.

### Constructor: Colocamos propiedades

Veamos cómo definir el constructor de la clase `Alumnouniversitario`, pero solamente vamos a colocar por ahora las propiedades de la clase.

```
function AlumnoUniversitario(nombre, edad){  
    this.nombre = nombre  
    this.edad = edad  
    this.numMatricula = null  
}
```



Los valores de inicialización los recibe el constructor como parámetros, en este caso es sólo el nombre y la edad, porque el número de matrícula no lo recibe un alumno hasta que es matriculado. Es por ello que asignamos null a la propiedad numMatrícula.

## Escribir métodos

Ahora vamos a continuar con esa definición de la clase para incorporar métodos.

Para construir un método debemos crear una función. Una función que se construye con intención de que sea un método para una clase puede utilizar también la variable this, que hace referencia al objeto sobre el que invocamos el método. Pues debemos recordar que para llamar a un método debemos tener un objeto y this hace referencia a ese objeto.

```
function matriculate(num_matricula){  
    this.numMatricula = num_matricula  
}
```

La función matricular recibe un número de matrícula por parámetro y lo asigna a la propiedad numMatricula del objeto que recibe este método. Así rellenamos el la propiedad del objeto que nos faltaba.

Vamos a construir otro método que imprime los datos del alumno.

```
function imprimete(){  
    document.write("Nombre: " + this.nombre)  
    document.write("<br>Edad: " + this.edad)  
    document.write("<br>Número de matrícula: " + this.numMatricula)  
}
```

Esta función va imprimiendo todas las propiedades del objeto que recibe el método.

## Constructor: Colocamos métodos

Para colocar un método en una clase debemos asignar la función que queremos que sea el método al objeto que se está creando. Veamos cómo quedaría el constructor de la clase AlumnoUniversitario con el método matricular.

```
function AlumnoUniversitario(nombre, edad){  
    this.nombre = nombre  
    this.edad = edad  
    this.numMatricula = null  
    this.matriculate = matriculate  
    this.imprimete = imprimete  
}
```

Vemos que en las últimas líneas asignamos a los métodos los nombres de las funciones que contienen su código.

## Para instanciar un objeto



Para instanciar objetos de la clase AlumnoUniversitario utilizamos la sentencia new, que ya hemos tenido ocasión de ver en otras ocasiones.

```
miAlumno = new AlumnoUniversitario("José Díaz",23)
```

## Trabajando con la clase, creando instancias y usando objetos

Para ilustrar el trabajo con objetos y terminar con el ejemplo del AlumnoUniversitario, vamos a ver todo este proceso en un solo script en el que definiremos la clase y luego la utilizaremos un poquito.

```
//definimos el método matricularte para la clase AlumnoUniversitario
function matriculate(num_matricula){
    this.numMatricula = num_matricula
}

//definimos el método imprimete para la clase AlumnoUniversitario
function imprimete(){
    document.write("<br>Nombre: " + this.nombre)
    document.write("<br>Edad: " + this.edad)
    document.write("<br>Número de matrícula: " + this.numMatricula)
}

//definimos el constructor para la clase
function AlumnoUniversitario(nombre, edad){
    this.nombre = nombre
    this.edad = edad
    this.numMatricula = null
    this.matriculate = matriculate
    this.imprimete = imprimete
}

//Creamos un alumno
miAlumno = new AlumnoUniversitario("José Díaz",23)

//Le pedimos que se imprima
miAlumno.imprimete()

//Le pedimos que se matricule
miAlumno.matriculate(305)

//Le pedimos que se imprima de nuevo (con el número de matrícula relleno)
miAlumno.imprimete()
```

Si lo deseamos, podemos [ver el script en funcionamiento en una página a parte](#).

No vamos a hablar más por el momento sobre cómo crear y utilizar nuestros propios objetos, pero en el futuro trataremos este tema con más profundidad y haremos algún ejemplo adicional.

Este artículo es obra de *Miguel Ángel Álvarez*

Fue publicado / actualizado en 21/05/2002

Disponible online en <https://desarrolloweb.com/articulos/clases-javascript-es5.html>



## Qué significa que Javascript es Orientado a Prototipos

Por qué se dice que Javascript es orientado a prototipos. Cómo funciona la orientación a prototipos de Javascript. Cómo cambiar el prototype de los objetos de Javascript para cambiar las propiedades o métodos existentes en los objetos de un tipo.



A veces se dice que Javascript es un **lenguaje "orientado a prototipos"**, en lugar de **orientado a objetos**. ¿Qué quiere decir? ¿Cómo afecta eso a mi día a día en el desarrollo con Javascript? ¿Qué debo saber sobre los prototipos en Javascript? Todas esas preguntas las vamos a responder en este artículo.

### Qué es la programación orientada a objetos

Bueno, para explicar la orientación a prototipos es ideal que conozcas la orientación a objetos tradicional, pues **la orientación a prototipos es un pequeño cambio** y lo vamos a estudiar en contraste con lo que ya conocemos.

La orientación a objetos es un paradigma de la programación que se basa en la creación de clases que son definiciones de los objetos de un tipo en particular. Para programar defines una "clase" que es como un plano o plantilla o definición, y luego creas **objetos basados en esas clases**.

Cada objeto se denomina una "*instancia*" de esa clase. Es decir, es un ejemplar concreto de una clase que tiene un conjunto de propiedades y funcionalidades que se ha definido por medio de una clase. Una clase define las propiedades y métodos que tendrán las instancias o ejemplares creados mediante esa clase. Cada instancia puede tener valores específicos y concretos de las propiedades definidas en la clase.

Como no es nuestro objetivo explicar con detalle este paradigma te vamos a recomendar que obtengas más información por medio del artículo sobre [qué es la programación orientada a objetos](#).

### Qué es la orientación a prototipos

Tradicionalmente la orientación a objetos se realiza por clases. Sin embargo, en los lenguajes orientados a prototipos tenemos un modelo de trabajo un poco distinto a la hora de crear los objetos



y a la hora de definir cómo unos objetos pueden heredar sus características.

En la **orientación a prototipos**, en lugar de tener clases, tienes objetos individuales que pueden ser clonados o extendidos para crear nuevos objetos. **Los objetos pueden heredar propiedades y métodos de otros objetos**. Es como si en lugar de crear casas mediante un plano se creasen mediante la copia de otra casa existente. Este proceso es llamado "cadena de prototipos".

### Cómo es la orientación a prototipos en Javascript

En JavaScript, cada objeto tiene un **enlace a un objeto prototipo** que es esa especie de molde que se usó para crear el objeto. En otras palabras, el prototipo es todo aquello en común que tienen los objetos de un tipo. Cuando se crean objetos de un tipo se hacen en base a un prototipo, el cual permite **heredar todas las características de ese tipo de objetos**. Pero incluso, cuando nosotros modificamos el prototipo de un objeto, automáticamente modificamos todos los objetos existentes de ese tipo.

En resumen, un "prototipo" es simplemente un **objeto del que otros objetos heredarán las propiedades y las funcionalidades al momento de crearse**. Pero además, si una vez creados los objetos modificamos el prototipo, los cambios se harán patentes en todos los objetos existentes que se hayan creado en base a ese prototipo. Esto es una característica muy potente.

Como en Javascript existen muchas maneras de crear objetos, podemos encontrar distintas maneras de trabajo que vamos a revisar.

### Prototipo con la función constructora

El mecanismo más común para crear objetos en Javascript, y el más tradicional que existe desde el inicio del lenguaje de programación, es mediante lo que llamamos función constructora.

Por ejemplo tenemos esta función que nos permitirá crear objetos Perro que son capaces de ladear.

```
function Perro(nombre) {
  this.nombre = nombre;
  this.ladear = function() {
    console.log('guau guau');
  }
}

const miPerro = new Perro('Jade');
miPerro.ladear();
```

Ahora podemos ver cómo modificar el prototipo del perro para asignar un nuevo método a esta clase.

```
Perro.prototype.saludar = function() {
  this.ladear()
  console.log(`Soy ${this.nombre}`);
}
```



Simplemente hemos asignado una nueva propiedad con el valor de una función a `Perro.prototype`. A partir de ahora todos los perros son capaces de saludar también. Incluso aunque se trate de perros que ya habían sido instanciados.

```
miPerro.saludar();
```

La magia que ha ocurrido aquí consiste simplemente en el comportamiento basado en prototipos de Javascript, que funciona así: Cuando intentas acceder a una propiedad que no está en el objeto actual, el intérprete buscará en la cadena de prototipos hasta que encuentre esa propiedad que se intenta acceder (o se llegue al final de la cadena de prototipos y no se encuentre nada).

### Prototipos con un objeto prototipo

Quizás es más fácil todavía de ver cómo funciona el prototipo con un ejemplo de creación de objetos basado en el clonado de otros objetos existentes.

Vamos a tener un objeto común de Javascript que luego usaremos como prototipo para crear otro objeto:

```
const CochePrototype = {
  modelo: 'BMW',
  arrancar: function() {
    console.log('rum rum');
  }
}

const renault = Object.create(CochePrototype);
renault.modelo = 'Renault';
```

Ahora podemos ajustar el prototipo `CochePrototype` para asignar nuevos métodos de esta manera:

```
CochePrototype.parar = function() {
  console.log(`pof pof, tu ${this.modelo} ha parado`);
}
```

Con ello conseguimos que todos los coches creados mediante este prototipo tengan el método nuevo, incluso si estaban creados antes de haber modificado el prototipo.

```
renault.arrancar();
renault.parar();
```

### Ejemplo de utilidad de prototipos en Javascript

Quizás no hemos visto que el prototipo sirva para gran cosa, pero sí que hay ejemplos clave que nos pueden ayudar a hacer cosas interesantes en el lenguaje.

Por ejemplo, sabemos que en Javascript las variables de tipo cadena tienen una serie de métodos de utilidad que podemos hacer con ellas. Ahora imaginemos que deseamos ampliar la funcionalidad de



Javascript para las cadenas y queremos crear un nuevo método que aplique a todas las cadenas nuevas y todas las variables de cadena que hayan sido creadas ya.

Esto lo podemos ver en este pedazo de código.

```
let x = "Javascript";

String.prototype.alertar = function() {
    alert(this);
}
```

Ahora la cadena x tendrá un nuevo método llamado "alertar" el cual mostrará la cadena en una caja de diálogo de tipo alert.

```
x.alertar();
```

Además, todas las cadenas creadas a partir de ahora tendrán también ese nuevo método.

```
let y = "Prototipo";
y.alertar();
```

En este ejemplo hemos hecho uso de la función `alert()` que está solamente disponible en el navegador, por lo que este último ejemplo no funcionará en NodeJS.

## Prototipos en las clases de ECMAScript 2015

Como quizás sabrás, en la versión de Javascript ES6 (ECMAScript 2015) se introdujo una **nueva sintaxis para crear clases**. Si no la conoces puedes estudiarla en el artículo [Clases en ES6](#).

Lo que queremos mencionar es que JavaScript esta utilidad no es más que un "azúcar sintáctico". Por ello, aunque usemos clases con una sintaxis más tradicional para luego crear objetos en base a ellas, no difiere en nada lo que ya sabes sobre los prototipos de Javascript. Es decir, estos objetos **seguirán basándose en prototipos**.

Por tanto la herencia de propiedades y métodos que te ofrecen los prototipos sigue aplicándose a este tipo de clases. Veamos un ejemplo con este tipo de declaración de clases y sus prototipos.

```
class Computador {
    constructor(marca, precio) {
        this.marca = marca;
        this.precio = precio;
    }

    comprar(importe) {
        if(importe >= this.precio) {
            console.log('Ahora el ordenador es tuyo!!');
        } else {
            console.log('Uppsss... debes seguir ahorrando');
        }
    }
}
```

{  
}

Ahora creamos un objeto de la clase Computador.

```
const miOrdenador = new Computador("Toshiba", 300);
```

Es interesante ver que si inspeccionamos este objeto (o cualquier otro objeto de Javascript) en la consola, podemos ver que se menciona el hecho de tener un prototipo. Quizás lo habías visto más de una vez y no sabías a qué se refería exactamente.

```
> miOrdenador
```

```
< ▼ Computador {marca: 'Toshiba', precio: 300} ⓘ  
  marca: "Toshiba"  
  precio: 300  
  ▼ [[Prototype]]: Object  
    ► comprar: f comprar(importe)  
    ► constructor: class Computador  
    ► [[Prototype]]: Object
```

Ahora vamos a crear un nuevo método en el prototipo de los computadores, usando la propiedad `prototype` de la clase `Computador`.

```
Computador.prototype.rebajar = function(rebaja) {  
  this.precio -= rebaja;  
}
```

Simplemente ese método ahora existirá en todos los computadores creados a partir de ahora e incluso en todos los computadores que se hayan creado anteriormente.

```
miOrdenador.rebajar(100);  
miOrdenador.comprar(200);
```

## Conclusión

Hemos conocido una de las características más marcantes del lenguaje Javascript. Este sistema de prototipos permite una gran flexibilidad en la estructura de los objetos y un dinamismo impactante, permitiendo cosas increíbles a la hora de heredar y compartir propiedades y funcionalidades de los objetos.

Sin embargo, también puede ser un punto oscuro y difícil de entender para los programadores que están acostumbrados a la orientación a objetos en lenguajes más tradicionales como Java, C# o PHP.



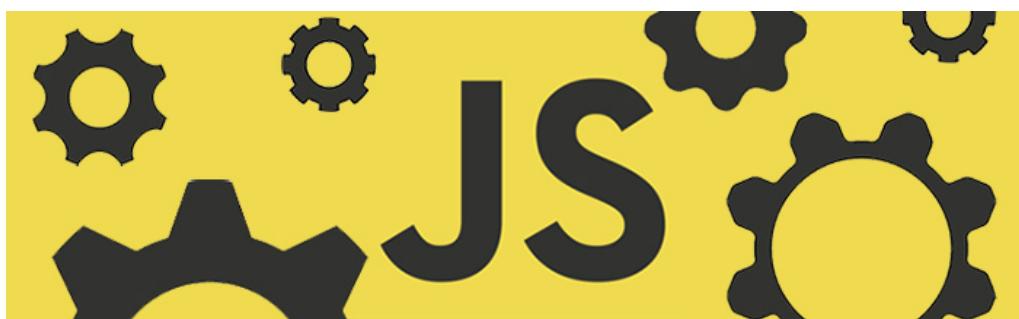
Este artículo es obra de *Miguel Angel Alvarez*

Fue publicado / actualizado en *19/07/2023*

Disponible online en <https://desarrolloweb.com/articulos/javascript-orientado-prototipos>

## Propiedades con get y set en Javascript

**Conoce los getters y setters en Javascript, llamados accessors, que permiten crear lo que se conoce propiedades computadas cuyo valor se extrae de métodos declarados con get y set.**



En este artículo vamos a conocer los **getters y los setters de Javascript**, una herramienta disponible en objetos de Javascript, que existe en el lenguaje desde hace bastante tiempo y que muchos desconocen. Son los getters y los setters que produces con get y set en los objetos o clases de Javascript.

En principio, esta utilidad se encuentra disponible en navegadores muy antiguos, pues forma parte de ES5, así que la puedes usar con tranquilidad. Si conoces la programación orientada a objetos, y conceptos como encapsulación, tendrá más sentido para ti, pero si no es así tampoco te viene mal echarle un vistazo, para cuando encuentres código que haga uso de esta construcción del lenguaje, que sepas de qué se trata.

### Qué es un getter o un setter

Para los que no han oído hablar sobre éstos, los getters y los setters son construcciones habituales de los objetos que permiten acceder a valores o propiedades, sin revelar la forma de implementación de las clases. En pocas palabras, permiten encapsular los objetos y evitar mantenimiento de las aplicaciones cuando la forma de implementar esos objetos cambia.

Utilidad aparte, en la práctica son simplemente métodos que te permite acceder a datos de los objetos, para leerlos o asignar nuevos valores. El setter lo que hace es asignar un valor y el getter se encarga de recibir un valor.

Sin embargo, los getters y los setters de los que vamos a hablar en Javascript son un poco distintos de los tradicionales.



## También llamados accessors

Los get y los set de Javascript se denominan técnicamente **accessors** o también "Object Accessors". Seguramente encontrarás esta terminología en documentación diversa de Javascript, por lo que deberías conocer el término.

Otra manera de denominar este tipo de miembros de objetos es con el nombre de propiedades computadas o "computed properties" en inglés.

## Propiedades computadas de objetos Javascript

Los get y los set de los que nos ocupamos en este artículo son útiles básicamente para realizar el acceso a propiedades de los objetos que son resultado de computar otras propiedades existentes.

El ejemplo más claro de setters y getters en Javascript lo tenemos con el objeto persona. Este objeto podría tener un par de propiedades como el "nombre" y los "apellidos". Además podríamos desear conocer el nombre completo, pero realmente éste no sería realmente una propiedad nueva del objeto, sino el resultado de un cómputo de concatenación entre el nombre y los apellidos que tenemos en las dos propiedades anteriores.

Es decir, "nombreCompleto" podría ser una propiedad computada. En los lenguajes tradicionales, incluso en Javascript si lo deseamos, podríamos implementar ese valor computado con un método, como este:

```
var persona = {
  nombre: 'Miguel Angel',
  apellidos: 'Alvarez Sánchez',
  getNombreCompleto: function() {
    return this.nombre + ' ' + this.apellidos;
  }
}
```

**Nota:** El código anterior, y el que verás en el resto de este artículo, está basado en [objetos literales de Javascript](#). Lo normal sería usar clases, pero mantendremos este estilo de codificación ya que las clases se agregaron en el lenguaje en la versión [ES6](#) y en este manual estamos tratando la versión tradicional de Javascript (ES5).

Podríamos acceder al nombre completo con este código:

```
var nombreCompletoPersona = persona.getNombreCompleto();
```

Sin embargo, con las construcciones get y set de Javascript la forma de definir este código cambia un poco. Mediante la palabra reservada "get" puedo definir una función que se encarga de realizar el cómputo. Aquello que devuelve la función será el valor de la propiedad computada. El código nos quería más o menos así:



```
var persona = {
    nombre: 'Miguel Angel',
    apellidos: 'Alvarez Sánchez',
    get nombreCompleto() {
        return this.nombre + ' ' + this.apellidos;
    }
}
```

La novedad es el uso de la construcción "get" en el código anterior, que nos sirve para definir una especie de método. Sin embargo, no es un método tradicional, sino más bien una propiedad computada, es decir, una propiedad de un objeto que para resolver su valor tiene que ejecutar una función. Nos debería quedar claro este punto con el código siguiente, en el que usamos esa propiedad computada "nombreCompleto".

```
var nombreCompletoPersona = persona.nombreCompleto;
```

Ahora en la variable "nombreCompletoPersona" tenemos el valor del nombre y apellidos concatenados. Fíjate que para acceder a ese "getter" usamos "persona.nombreCompleto", que es como si estuviéramos accediendo a una propiedad convencional, aunque internamente en nuestro objeto no existe tal propiedad, sino que se realiza un cómputo para poder evaluarla.

### Ejemplo completo con get y set

En el siguiente ejemplo vamos a profundizar un poco en los getters y los setters de Javascript, implementando un nuevo caso de uso que seguro que nos despejará posibles dudas. Además veremos cómo funciona set, para asignar un valor "computado".

En nuestro ejemplo tenemos un objeto intervalo que para su implementación define un valor máximo y un valor mínimo. Pero queremos disponer de un mecanismo que nos ofrezca los valores, enteros, comprendidos en ese intervalo. Para practicar con los get y set vamos a hacerlo mediante una propiedad computada.

Aunque primero vamos a ver cómo se resolvería de manera tradicional, con un método de toda la vida.

```
var intervalo = {
    valorMinimo: 3,
    valorMaximo: 4,
    valoresContenidos: function() {
        var contenidos = [];
        for(var i=this.valorMinimo; i<=this.valorMaximo; i++) {
            contenidos.push(i);
        }
        return contenidos;
    }
}
```

Como ves, para acceder a los valores contenidos tendríamos que ejecutar el método correspondiente:



```
var valores = intervalo.valoresContenidos();
```

Ahora vamos a ver cómo realizar esto mismo pero con una propiedad "get", computada.

```
var intervalo = {
    valorMinimo: 3,
    valorMaximo: 7,
    get valoresContenidos() {
        var contenidos = [];
        for(var i=this.valorMinimo; i<=this.valorMaximo; i++) {
            contenidos.push(i);
        }
        return contenidos;
    },
}
```

**Nota:** No lo hemos mencionado todavía, pero las propiedades generadas con get se implementan con funciones no pueden recibir ningún parámetro.

Ahora llamaríamos al método de esta manera.

```
var valores = intervalo.valoresContenidos;
```

Nos queda por ver un ejemplo con la construcción "set". En este caso lo que vamos a hacer es recibir un parámetro con aquello que se tenga que asignar y realizar los cambios en las propiedades de los objetos que sea pertinente.

```
var intervalo = {
    valorMinimo: 3,
    valorMaximo: 7,
    get valoresContenidos() {
        var contenidos = [];
        for(var i=this.valorMinimo; i<=this.valorMaximo; i++) {
            contenidos.push(i);
        }
        return contenidos;
    },
    set valoresContenidos(arrayValores) {
        arrayValores.sort();
        this.valorMinimo = arrayValores[0];
        this.valorMaximo = arrayValores[arrayValores.length - 1];
    }
}
```

El setter se define de manera similar al getter. En este caso como hemos dicho se usará una función que recibe un dato que haya que setear. La gracia está en el modo de uso de un setter, que no es un método tradicional en el que se enviarían parámetros, sino que se invoca mediante una asignación.

```
intervalo.valoresContenidos = [5, 6, 2, 9, 3];
```



El valor que asignamos en la propiedad definida como un setter es el que se recibe como valor del parámetro.

Eso es todo, espero que puedas sacarle partido a los getters y los setters de Javascript, una interesante funcionalidad del lenguaje, que cada vez se utiliza más en diferentes áreas de las aplicaciones.

Este artículo es obra de *Miguel Angel Alvarez*

Fue publicado / actualizado en *11/03/2022*

Disponible online en <https://desarrolloweb.com/articulos/propiedades-get-set-javascript.html>

## La palabra this y el contexto en Javascript

**Aprende qué tiene de particular el uso de this en Javascript con respecto a otros lenguajes de programación. Entiende los distintos valores de this dependiendo del contexto donde se use.**



Hemos realizado una charla recientemente en nuestro canal de Youtube dedicada a las "buenas prácticas en el lenguaje Javascript". Allí vimos varios puntos interesantes que ahora estamos transcribiendo en artículos. Éste es el segundo que publicamos, siendo que el primero se dedicó a tratar la [sobrecarga de funciones](#). Nos lo ofreció nuestro compañero Eduard Tomàs, que siempre nos obsequia con grandes joyas de conocimiento.

En los lenguajes de programación más "tradicionales", la variable "this" siempre tiene un valor clarísimo. Sin lugar a dudas, "this" siempre contiene la referencia al objeto sobre el que se está ejecutando un método. Todos los métodos se ejecutan en el contexto de un objeto (esto exceptuando los métodos estáticos, en los que no puede accederse a this) y "this" apunta a este objeto. Sin embargo, en Javascript no siempre es así.

Consulta el [Manual de Programación Orientada a Objetos](#) saber más de this y otros conceptos



sobre este paradigma.

En Javascript, buscando por muchos sitios realmente se encuentran definiciones de "this" un poco complicadas de entender. Te dicen que es "el propietario de la función"... y en caso que no exista propietario, te dicen que "this" es el "objeto global".

**Nota:** Cuando ejecutas Javascript dentro de una página web, o sea, en un navegador, el objeto global es el objeto "window", a partir del que cuelgan todos los objetos de la jerarquía. Sin embargo, en NodeJS que se ejecuta en una máquina como un lenguaje de programación de propósito general, el objeto global es otra cosa. También existe un objeto global en NodeJS, que tiene una serie de funciones, que hace el rol del objeto "window". Obviamente, en ese objeto global de NodeJS tengo cosas muy distintas de las que encuentras en "window" en un navegador, pero también existe un "contexto global".

Dicho con otras palabras, que podemos encontrar en literatura Javascript, "**this es el contexto de una función**". Una de las cosas interesantes de Javascript y muy potentes es que el contexto se puede modificar y depende de como se invoca esa función. Repetimos: el valor de "this" dentro de una función no depende de cómo se define esa función, sino de cómo se invoca.



## this como el objeto global

Si yo coloco código fuera de cualquier función y trato de ver el contenido de this, encontraré el contexto global. Puedes probarlo con una línea como esta:

```
console.log(this);
```

Si estás en un navegador encontrarás que te manda a la consola el contenido del objeto "window".

## This como el propietario de la función Javascript

Si ahora escribes el código de una función y dentro accedes a "this", lo que encontrarás es el "proprietario de la función". Pero esto del "proprietario" es un poco complicado de entender, porque básicamente pueden ocurrir dos cosas:



1. Si la función se ejecuta como global, "this" será el propio objeto global.
2. Si la función se ejecuta como método de un objeto, entonces "this" es el objeto que está recibiendo este método.

Por ejemplo, si tenemos este código tal cual en una página web, fuera de cualquier función u objeto:

```
function miFuncion() {
    console.log(this);
}
miFuncion();
```

Veremos que el valor de `this` en este caso resulta ser el objeto `window`.

Tiene sentido que sea el objeto `window` porque en el javascript del navegador, toda declaración global (como esta función) se adjunta como propiedad del objeto `window`. Por tanto, el propietario de una función global viene a ser el objeto global, o sea, `window`. Esto quiere decir, que a la función podríamos haberla invocado también como `window.miFuncion()`, dado que pertenece al objeto `window`. Son esas cosas raras de Javascript que cuando las entiendes no te parecen tan extrañas.

Ahora vamos a ver cómo `this` cambia en el caso que el propietario de una función sea un objeto. Tenemos el siguiente código:

```
let miObjeto = {
    prop: 'valor',
    metodo: function() {
        console.log(this);
    }
}
miObjeto.metodo();
```

Al ejecutarse podrás ver que en la consola el valor de `this` que se muestra es el propio objeto "`miObjeto`".

Pero luego existen mecanismos para cambiarlo, como veremos también.

## Error típico en Javascript

Ahora veamos un código que podemos encontrarnos por ahí que podría resultar en otros lenguajes correcto, pero que en Javascript denota que el programador no entiende bien que "this" puede apuntar a varias cosas.

```
//esto estaría conceptualmente mal
var obj = {
    nombre: "Edu",
    apellidos: "Tomas",
    completo: this.nombre + " " + this.apellidos
}
```



```
console.log(obj);
//nos mostrará en consola
//Object { nombre="Edu", apellidos="Tomas", completo="undefined undefined"}
```

Como ves, hemos definido un objeto con la "notación de objeto Javascript" conocida como JSON habitualmente. La propiedad completo, a la que aparentemente se habría asignado el nombre concatenado con los apellidos, en realidad nos queda como "undefined undefined". Esto es porque en ese lugar, "this" es el objeto global, ya que no se ha ejecutado dentro de una función como método de un objeto.

**Nota:** Te devuelve *undefined* porque en el contexto global (objeto "window" teniendo en cuenta que eso se ejecutase en un navegador) "window.nombre" no existe y tampoco "window.apellidos" y si accedemos a una propiedad inexistente de un objeto, Javascript nos devuelve el valor "undefined".

Eso se podría haber "arreglado" convirtiendo a "completo" en una función que devuelve esa concatenación:

```
var obj = {
  nombre: "Edu",
  apellidos: "Tomas",
  completo: function(){
    return this.nombre + " " + this.apellidos;
  }
}
console.log(obj.completo());
//nos mostrará en consola "Edu Tomas"
```

Luego si invocamos a la función completo como método del objeto, "this" sí que será la instancia del objeto sobre el que se llamó al método.

### this en los manejadores de eventos

Otro caso interesante de la referencia `this` en Javascript es cuando definimos manejadores de eventos. En ellos la palabra `this` hace referencia al objeto sobre el cual se ha configurado el manejador de evento.

Esto lo podemos ver muy claramente con un ejemplo de manejador de evento, como en el código siguiente.

```
document.addEventListener('mousemove', function() {
  console.log(this);
});
```

Hemos definido un manejador para el evento 'mousemove' sobre el objeto document. Esto hará que cada vez que se mueva el ratón por encima del documento mostrará el valor de `this` en la consola. Si lo ejecutas podrás apreciar que el valor que aparece es el objeto `document` de la página.



Nuevamente, tiene sentido porque en este caso podemos pensar que el propietario de una función definida como manejador de eventos sobre un objeto dado, es el propio objeto. Es decir, en este caso el manejador definido sobre el objeto document, tendría como propietario al objeto document. Por tanto, `this` contendrá una referencia al objeto document.

Si nuestro manejador de eventos se hubiera colocado en un botón de la página, `this` haría referencia a ese botón de la página, dado que sería el propietario de la función manejadora.

## Funciones call / apply / bind

Para terminar de liarlo, vienen tres funciones "muy divertidas" que son "call()", "apply()" y "bind()". Estas funciones las veremos con detalle más adelante, cuando veamos otros aspectos de Javascript, pero básicamente, "call()" y "apply()" sirven para invocar una función pero en la que se suministra un valor de "this" que sustituirá al valor que hubiera tomado en otra situación.

Dicho de otra manera, en la ejecución de una función, el valor de "this" no será el que tocaría, sino el que será el valor que tú le indiques en la llamada a "call()" o "apply()".

Por su parte, "bind()" es un método muy curioso que para empezar, no devuelve un valor, sino que devuelve otra función. Cuando invoco esta otra función que devuelve, el valor de "this" dentro de ella, es el valor original que yo había pasado al invocar a "bind()".

**Nota:** Tener siempre en cuenta que Javascript es un lenguaje tipo "funcional" (por decirlo de alguna manera), en el que las funciones son ciudadanos de primer orden. El tipo de datos "función" es tan valido como puede ser "string", "object", "number"... Eso significa que puedo pasar funciones como parámetro y puedo devolver funciones como valores de retorno.

Esto es difícil verlo sin un pedazo de código y aunque más adelante queremos ver "bind()" con mayor detalle, creo que estaría bien echar un vistazo:

```
function bindeando(){
  console.log(this);
}
bindeando.bind("hola!!")();
```

## Conclusiones

- El valor de `this` no es el clásico que todos tenemos en mente en otros lenguajes de programación como podrían ser PHP, Java, C#, C++...
- No puedes saber el valor que tendrá la variable "`this`" al ver el código de una función, porque depende de cómo te hayan invocado a esa función.

Más adelante veremos más detalles de "this" y las funciones "call()", "apply()" y "bind()". y en este artículo nos hemos quedado en el minuto 27 de esa presentación.

## Vídeo de Buenas prácticas en Javascript

Acabamos compartiendo el vídeo del que hemos extractado la información contenida en este artículo sobre la palabra `this` en Javascript. Esperamos que os guste esta charla.

Para ver este vídeo es necesario visitar el artículo original en:

<https://desarrolloweb.com/articulos/palabra-this-contexto-javascript.html>

Este artículo es obra de *Eduard Tomàs*

Fue publicado / actualizado en *21/01/2023*

Disponible online en <https://desarrolloweb.com/articulos/palabra-this-contexto-javascript.html>

## Mixin en Javascript

**Qué es un mixin en Javascript, ejemplo de mixin para conseguir los efectos de herencia múltiple en un lenguaje de programación que no la soporta.**



[Javascript](#) es un lenguaje de programación que no soporta **herencia múltiple**, igual que ocurre en la mayoría de los lenguajes de programación orientados a objetos. Eso no es problema, gracias a las posibilidades de Javascript y lo sencillo que es retorcer el lenguaje para llegar a conseguir cosas para las que inicialmente no está preparado.

En este artículo vamos a explicar lo **que es un mixin en Javascript**. Como tal, no se trata de ninguna estructura o utilidad nueva del lenguaje, que se ofrezca de manera "oficial", como podrían ser las clases, sino más bien una técnica para poder llegar a soluciones similares a la herencia múltiple, sin disponer de ella realmente.



Este artículo lo hemos englobado en el [Manual de Javascript](#), pero por tratarse más de una técnica que de algo que aporta directamente el lenguaje, podría haberse agregado al taller. Lo hemos decidido colocar en el manual de Javascript finalmente porque es el lugar donde se explican muchas de las cosas que tienen que ver con los objetos y las clases de Javascript, aunque en realidad es un tema que resulta un poco más avanzado, en comparación con otros artículos.

Si eres principiante en Javascript quizás es un poco pronto para ti entender los mixins y tardes un poco más de tiempo en sacarles partido. En este caso casi te recomendamos continuar con otros artículos del manual. Si has llegado aquí justamente para saber qué son los mixin y para y cómo los puedes usar, estás en el lugar correcto!

## Qué es un mixin en Javascript

**Un mixin en Javascript es una función que recibe una clase base y devuelve esa misma clase base, a la que se han añadido nuevos métodos.**

Es decir, un mixin es una función que devuelve una clase. El mixin para funcionar recibe como parámetro una clase base. Crea una clase anónima, que extiende de la clase base recibida por parámetro, y devuelve la clase anónima creada.

Quizás explicado con palabras es más complicado que ver que si mostramos un ejemplo de mixin:

```
let miMixin = (claseBase) => class extends claseBase {
    constructor() {
        super();
        this.name = "Soy c2";
    }

    saludar() {
        console.log("hola, " + this.name);
    }
}
```

Como puedes ver, hemos definido una función que devuelve una clase anónima. La función recibe por parámetro "claseBase" y la clase que devuelve extiende de "claseBase".

Podemos usar el mixin enviando cualquier clase base, a la que el mixin agrega un método constructor, que nos permite crear nuevas propiedades y un método extra, para agregar a la clase base.

Si has entendido esto, ya sabes todo lo que necesitas saber sobre los mixin y entenderás por qué son más que nada una técnica.

## Cómo usar un mixin en Javascript

Ahora vamos a ver cómo podríamos usar ese mixin en Javascript. Como se trata simplemente de una función podemos invocarla con cualquier clase base y habremos conseguido una clase



extendida a la que se han incorporado los métodos del mixin.

```
let ClaseNuevaExtendida = miMixin(ClaseBaseQueSeExtendera);
```

A continuación puedes hacer new de la ClaseNuevaExtendida, como estás acostumbrado a hacer con cualquier otra clase.

```
let objeto = new ClaseNuevaExtendida;
```

El objeto contendrá todo lo que hereda de ClaseBaseQueSeExtendera junto con todo lo que se ha incorporado por medio del mixin.

### Ejemplo completo de creación y uso de mixins en Javascript

Ahora vamos a ver un ejemplo para ilustrar un poco mejor todo lo que hemos visto, para que no queden dudas sobre la creación de mixins en Javascript y su uso.

```
class Base1 {
  constructor() {
    this.prop = 1;
  }
  foo() {
    this.prop++;
    console.log(this.prop);
  }
}

let miMixin = (claseBase) => class extends claseBase {
  constructor() {
    super();
    this.name = "Soy c2";
  }

  saludar() {
    console.log("hola, " + this.name);
  }
}

class Extendida extends miMixin(Base1) {
  resetea() {
    this.prop = 0;
  }
}

let miObjeto = new Extendida();
miObjeto.saludar();
miObjeto.foo();
miObjeto.foo();
miObjeto.resetea()
miObjeto.foo();
```

En este ejemplo "miObjeto", que es un objeto de la clase Extendida, contendrá:

- Todas las propiedades y métodos heredados de la clase Base1
- Todas las propiedades y métodos heredados que ha incorporado el mixin miMixin
- Todas las propiedades y métodos que se ha declarado en la definición de la clase Extendida

La salida de este programa sería la siguiente:

```
hola, soy c2  
2  
3  
1
```

## Conclusión sobre los mixins en Javascript

Con esto has aprendido lo que es un mixin de Javascript. Es una técnica que te conviene tener a mano, bastante típica en el día a día de la programación con el lenguaje, en lenguajes que no soportan la herencia múltiple. Para quien conozca PHP es algo como lo que nos permiten los traits.

Personalmente lo uso bastante a la hora de hacer custom elements del estándar de [Web Components](#), ya que nos permite separar métodos que podemos llegar a utilizar desde varios componentes. Todos los métodos que quieras reutilizar en varios componentes los puedes meter en un mixin que usarás todas las veces que te haga falta, aumentando las posibilidades de reutilización del código.

Este artículo es obra de *Miguel Angel Alvarez*

Fue publicado / actualizado en *10/10/2022*

Disponible online en <https://desarrolloweb.com/articulos/mixin-javascript>

# Librerías de clases y objetos en Javascript

En Javascript, como en cualquier lenguaje, existen muchas funcionalidades básicas ya desarrolladas para el resolver necesidades habituales en los programas, como cálculos matemáticos, o control de fechas. En los siguientes artículos podrás conocer cómo esas librerías están disponibles en Javascript por medio de diversas clases y objetos de utilidad general. Aprenderemos y practicaremos con diversos recursos del lenguaje para realizar todo tipo de operaciones matemáticas avanzadas, trabajo con cadenas de caracteres, trabajo con fechas, tipos primitivos, etc.

## Objetos incorporados en Javascript

### **Lista de los objetos que tenemos a nuestra disposición a la hora de trabajar con Javascript.**

Llegamos a un punto interesante dentro de la segunda parte del [Manual de programación en Javascript](#). En estos momentos sabemos ya lo que es la [programación orientada a objetos](#), así que vamos a introducirnos en el manejo de objetos en Javascript y para ello vamos a empezar con el estudio de las clases predefinidas que implementan las librerías para el trabajo con strings, matemáticas, fechas etc.

Las clases que se encuentran disponibles de manera nativa en Javascript, y que vamos a ver a continuación, son las siguientes:

- [\*\*String\*\*](#), para el trabajo con cadenas de caracteres.
- [\*\*Date\*\*](#), para el trabajo con fechas.
- [\*\*Math\*\*](#), para realizar funciones matemáticas.
- [\*\*Number\*\*](#), para realizar algunas cosas con números
- [\*\*Boolean\*\*](#), trabajo con booleanos.

**Nota: Las clases se escriben con la primera letra en mayúsculas.** Tiene que quedar claro que una clase es una especie de "declaración de características y funcionalidades" de los objetos. Dicho de otro modo, las clases son descripciones de la forma y funcionamiento de los objetos. Con las clases generalmente no se realiza ningún trabajo, sino que se hace con los objetos, que creamos a partir de las clases.

Una vez comprendida la diferencia entre objetos y clases, cabe señalar que String es una clase, lo mismo que Date. Si queremos trabajar con cadenas de caracteres o fechas necesitamos crear objetos de las clases String y Date respectivamente. Como sabemos, Javascript es un lenguaje sensible a las mayúsculas y las minúsculas y en este caso, **las clases, por convención,**



**siempre se escriben con la primera letra en mayúscula y también la primera letra de las siguientes palabras**, si es que el nombre de la clase está compuesto de varias palabras.

Por ejemplo si tuvieramos la clase de "Alumnos universitarios" se escribiría con la -A- de alumnos y la -U- de universitarios en mayúscula. AlumnosUniversitarios sería el nombre de nuestra supuesta clase.

Hay otros que pertenecen a este mismo conjunto, los enumeramos aquí para que quede constancia de ellos, pero no los vamos a tocar en capítulos siguientes.

- **Array**, ya los hemos visto en [capítulos correspondientes al primer manual de Javascript](#).
- También la clase **Function**, lo hemos visto parcialmente al [estudiar las funciones](#).
- Hay otra clase **RegExp** que sirve para construir patrones que veremos tal vez junto con Function cuando tratemos temas más avanzados todavía.

**Nota: Uso de mayúsculas y minúsculas.** Ya que nos hemos parado anteriormente a hablar sobre el uso de mayúsculas en ciertas letras de los nombre de las clases, vamos a terminar de explicar la convención que se lleva a cabo en Javascript para nombrar a los elementos.

Para empezar, cualquier variable se suele escribir en minúsculas, aunque si este nombre de variable se compone de varias palabras, se suele escribir en mayúscula la primera letra de las siguientes palabras a la primera. Esto se hace así en cualquier tipo de variable o nombre menos en los nombres de las clases, donde se escribe también en mayúscula el primer carácter de la primera palabra.

Ejemplos:

**Number**, que es una clase se escribe con la primera en mayúscula.

**RegExp**, que es el nombre de otra clase compuesto por dos palabras, tiene la primera letras de las dos palabras en mayúscula.

**miCadena**, que supongamos que es un objeto de la clase String, se escribe con la primera letra en minúscula y la primera letra de la segunda palabra en mayúscula.

**fecha**, que supongamos que es un objeto de la clase Date, se escribe en minúsculas por ser un objeto.

**miFunciónO**, que es una función definida por nosotros, se acostumbra a escribir con minúscula la primera.

Como decimos, esta es la norma general para dar nombres a las variables, clases, objetos, funciones, etc. en Javascript. Si la seguimos así, no tendremos problemas a la hora de saber si un nombre en Javascript tiene o no alguna mayúscula y además todo nuestro trabajo será más claro y fácil de seguir por otros programadores o nosotros mismos en caso de retomar un código una vez pasado un tiempo.

Este artículo es obra de *Miguel Angel Alvarez*

Fue publicado / actualizado en *21/03/2002*

Disponible online en <https://desarrolloweb.com/articulos/728.php>

## Tratamiento de String en Javascript

**Cadenas en Javascript:** En este artículo encontrarás toda la información y ejemplos el tratamiento de strings en Javascript o cadenas de caracteres. Clase y objetos String, funciones de utilidad de cadenas, etc.

En javascript las variables de tipo texto son objetos de la clase String. Esto quiere decir que cada una de las variables que creamos de tipo texto tienen una serie de propiedades y métodos.

Recordamos que las propiedades son las características, como por ejemplo longitud en caracteres del string y los métodos son funcionalidades, como pueden ser extraer un substring o poner el texto en mayúsculas.

Para crear un objeto de la clase String lo único que hay que hacer es asignar un texto a una variable. El texto va entre comillas, como ya hemos visto en los capítulos de sintaxis. También se puede crear un objeto string con el operador new, que veremos más adelante. La única diferencia es que en versiones de Javascript 1.0 no funcionará new para crear los Strings.



### Propiedades de String

#### Length

La clase String sólo tiene una propiedad: length, que guarda el número de caracteres del String.

### Métodos de String

Los objetos de la clase String tienen una buena cantidad de métodos para realizar muchas cosas interesantes. Primero vamos a ver una lista de los métodos más interesantes y luego vamos a ver otra lista de métodos menos útiles.

### charAt(indice)

Devuelve el carácter que hay en la posición indicada como índice. Las posiciones de un string empiezan en 0.

### indexOf(carácter,desde)

Devuelve la posición de la primera vez que aparece el carácter indicado por parámetro en un string. Si no encuentra el carácter en el string devuelve -1. El segundo parámetro es opcional y sirve para indicar a partir de qué posición se desea que empiece la búsqueda.

### lastIndexOf(carácter,desde)

Busca la posición de un carácter exactamente igual a como lo hace la función indexOf pero desde el final en lugar del principio. El segundo parámetro indica el número de caracteres desde donde se busca, igual que en indexOf.

### replace(substring\_a\_buscar,nuevoStr)

Implementado en Javascript 1.2, sirve para reemplazar porciones del texto de un string por otro texto, por ejemplo, podríamos utilizarlo para reemplazar todas las apariciones del substring "xxx" por "yyy". El método no reemplaza en el string, sino que devuelve un resultante de hacer ese reemplazo. Acepta expresiones regulares como substring a buscar.

### split(separador)

Este método sólo es compatible con javascript 1.1 en adelante. Sirve para crear un vector a partir de un String en el que cada elemento es la parte del String que está separada por el separador indicado por parámetro.

### substring(inicio,fin)

Devuelve el substring que empieza en el carácter de inicio y termina en el carácter de fin. Si intercambiamos los parámetros de inicio y fin también funciona. Simplemente nos da el substring que hay entre el carácter menor y el mayor.

### toLowerCase()

Pone todas los caracteres de un string en minúsculas.

### toUpperCase()

Pone todas los caracteres de un string en mayúsculas.

### toString()

Este método lo tienen todos los objetos y se usa para convertirlos en cadenas.

Hasta aquí hemos visto los métodos que nos ayudarán a tratar cadenas. Ahora vamos a ver otros métodos que son menos útiles, pero hay que indicarlos para que quede constancia de ellos. Todos



---

sirven para aplicar estilos a un texto y es como si utilizásemos etiquetas HTML. Veamos cómo.

### anchor(name)

Convierte en un ancla (sitio a donde dirigir un enlace) una cadena de caracteres usando como el atributo name de la etiqueta `<A>` lo que recibe por parámetro.

### big()

Aumenta el tamaño de letra del string. Es como si colocásemos en un string al principio la etiqueta `<BIG>` y al final `</BIG>`.

### blink()

Para que parpadee el texto del string, es como utilizar la etiqueta `<BLINK>`. Solo vale para Netscape.

### bold()

Como si utilizásemos la etiqueta `<B>`.

### fixed()

Para utilizar una fuente monoespaciada, etiqueta `<TT>`.

### fontColor(color)

Pone la fuente a ese color. Como utilizar la etiqueta `<FONT color=el_color_indicado>`.

### fontSize(tamaño)

Pone la fuente al tamaño indicado. Como si utilizásemos la etiqueta `<FONT>` con el atributo size.

### italics()

Pone la fuente en cursiva. Etiqueta `<I>`.

### link(url)

Pone el texto como un enlace a la URL indicada. Es como si utilizásemos la etiqueta `<A>` con el atributo href indicado como parámetro.

### small()

Es como utilizar la etiqueta `<SMALL>`

### strike()

Como utilizar la etiqueta `<STRIKE>`, que sirve para que el texto aparezca tachado.

### sub()



Actualiza el texto como si se estuviera utilizando la etiqueta <SUB>, de subíndice.

### sup()

Como si utilizásemos la etiqueta <SUP>, de superíndice.

## Ejemplos de funcionamiento de los objetos de clase String

Hasta ahora hemos visto muchos métodos interesantes de elementos de tipo String, pero no estará de más pasar a la práctica para entender todo mucho mejor. Así pues, veamos unos ejemplos sobre cómo se utilizan los métodos y propiedades del objeto String.

### Ejemplo de strings 1

Vamos a escribir el contenido de un string con un carácter separador ("") entre cada uno de los caracteres del string.

```
var miString = "Hola Amigos"
var result = ""

for (i=0;i<miString.length-1;i++) {
    result += miString.charAt(i)
    result += "-"
}
result += miString.charAt(miString.length - 1)

document.write(result)
```

Primero creamos dos variables, una con el string a recorrer y otra con un string vacío, donde guardaremos el resultado. Luego hacemos un bucle que recorre desde el primer hasta el penúltimo carácter del string -utilzamos la propiedad length para conocer el número de caracteres del string- y en cada iteración colocamos un carácter del string seguido de un carácter separador "-". Como aun nos queda el último carácter por colocar lo ponemos en la siguiente línea después del bucle. Utilizamos la función charAt para acceder a las posiciones del string. Finalmente imprimimos en la página el resultado.

### Ejemplo de strings 2

Vamos a hacer un script que rompa un string en dos mitades y las imprima por pantalla. Las mitades serán iguales, siempre que el string tenga un número de caracteres par. En caso de que el número de caracteres sea impar no se podrá hacer la mitad exacta, pero partiremos el string lo más aproximado a la mitad.

```
var miString = "0123456789"
var mitad1,mitad2

posicion_mitad = miString.length / 2

mitad1 = miString.substring(0,posicion_mitad)
mitad2 = miString.substring(posicion_mitad,miString.length)

document.write(mitad1 + "<br>" + mitad2)
```

Las dos primeras líneas sirven para declarar las variables que vamos a utilizar e inicializar el string a partir. En la siguiente línea hallamos la posición de la mitad del string.

En las dos siguientes líneas es donde realizamos el trabajo de poner en una variable la primera mitad del string y en la otra la segunda. Para ello utilizamos el método substring pasándole como inicio y fin en el primer caso desde 0 hasta la mitad y en el segundo desde la mitad hasta el final. Para finalizar imprimimos las dos mitades con un salto de línea entre ellas.

Una vez sabemos manejar los objetos de la clase string, podemos pasar a ver otras de las clases nativas de Javascript, como la [clase Date](#).

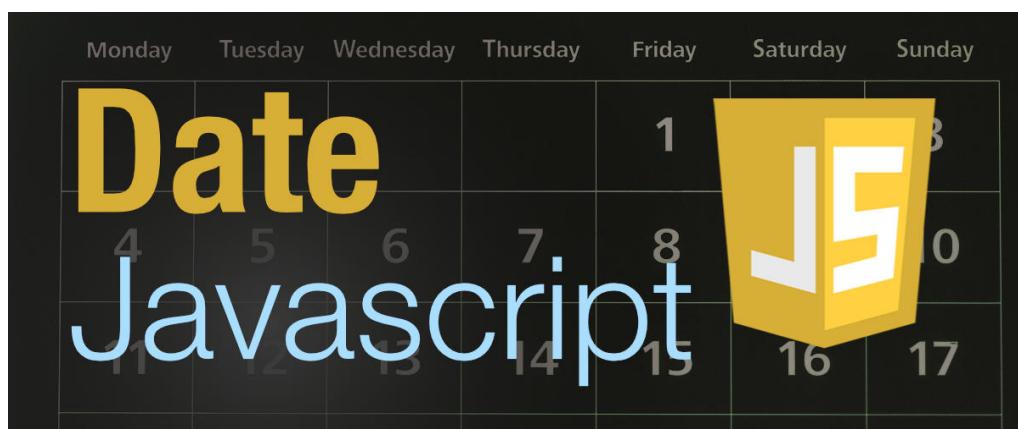
Este artículo es obra de *Miguel Angel Alvarez*

Fue publicado / actualizado en 05/12/2019

Disponible online en <https://desarrolloweb.com/articulos/objetos-string-javascript.html>

## Clase Date en Javascript

**Explicamos la clase Date de Javascript, que se utiliza para el manejo de fechas y horas. Exploramos sus diversos métodos y propiedades de los objetos Date, con los que realizar cálculos en el tiempo.**



Vamos a empezar a relatar todas las cosas que debes saber sobre otro de los [objetos nativos de Javascript](#), el que implementa la clase Date.

Sobre la clase Date recae todo el trabajo con fechas y horas en Javascript, como obtener una fecha, el día la hora actuales y otras cosas. Para trabajar con fechas necesitamos instanciar un objeto de la clase `Date` y con él ya podemos realizar las operaciones que necesitemos.

Un objeto de la clase `Date` se puede crear de dos maneras distintas. Por un lado podemos crear el objeto con el día y hora actuales y por otro podemos crearlo con un día y hora distintos a los actuales. Esto depende de los parámetros que pasemos al construir los objetos.



Para crear un objeto fecha con el día y hora actuales colocamos los paréntesis vacíos al llamar al constructor de la clase `Date`.

```
let miFecha = new Date()
```

Para crear un objeto fecha con un día y hora distintos de los actuales tenemos que indicar entre paréntesis el momento con que inicializar el objeto. Hay varias maneras de expresar un día y hora válidas, por eso podemos construir una fecha guiándonos por varios esquemas. Estos son dos de ellos, suficientes para crear todo tipo de fechas y horas.

```
let miFecha = new Date(año,mes,dia,hora,minutos,segundos)
let miFecha2 = new Date(año,mes,dia)
```

Los valores que debe recibir el constructor son siempre numéricos. Un detalle: el mes comienza por 0, es decir, enero es el mes 0. Si no indicamos la hora, el objeto fecha se crea con hora 00:00:00.

Los objetos de la clase `Date` no tienen propiedades pero si un montón de métodos, vamos a verlos ahora.

### `getDate()`

Devuelve el día del mes, un valor que puede ir desde 1 a 31. Depende del mes que sea, claro.

### `getDay()`

Devuelve el día de la semana. Por ejemplo, si la fecha corresponde con el martes te devolvería 2.

### `getHours()`

Retorna el valor de la hora.

### `getMinutes()`

Devuelve los minutos.

### `getMonth()`

Devuelve el mes, comenzando por cero (atención a este valor porque el mes de enero correspondería con el valor 0, o diciembre con el 11).

### `getSeconds()`

Devuelve los segundos.

### `getTime()`

Devuelve los milisegundos transcurridos entre el día 1 de enero de 1970 y la fecha correspondiente al objeto al que se le pasa el mensaje.



### getYear()

Retorna el año, al que se le ha restado 1900. Por ejemplo, para el 1995 retorna 95, para el 2005 retorna 105. Este método está obsoleto en Netscape a partir de la versión 1.3 de Javascript y ahora se utiliza `getFullYear()`.

### getFullYear()

Retorna el año con todos los dígitos. Usar este método para estar seguros de que funcionará todo bien en fechas posteriores al año 2000.

### setDate()

Actualiza el día del mes.

### setHours()

Actualiza la hora.

### setMinutes()

Cambia los minutos.

### setMonth()

Cambia el mes (atención al mes que empieza por 0).

### setSeconds()

Cambia los segundos.

### setTime()

Actualiza la fecha completa. Recibe un número de milisegundos desde el 1 de enero de 1970.

### setYear()

Cambia el año recibe un número, al que le suma 1900 antes de colocarlo como año de la fecha. Por ejemplo, si recibe 95 colocará el año 1995. Este método está obsoleto a partir de Javascript 1.3 en Netscape. Ahora se utiliza `getFullYear()`, indicando el año con todos los dígitos.

### setFullYear()

Cambia el año de la fecha al número que recibe por parámetro. El número se indica completo ej: 2005 o 1995. Utilizar este método para estar seguros que todo funciona para fechas posteriores a 2000.

Como habréis podido apreciar, hay algún método obsoleto por cuestiones relativas al año 2000: `setYear()` y `getYear()`. Estos métodos se comportan bien en Internet Explorer y no nos dará ningún problema el utilizarlos. Sin embargo, no funcionarán correctamente en Netscape, por lo que es



interesante que utilicemos en su lugar los métodos `getFullYear()` y `setFullYear()`, que funcionan bien en Netscape e Internet Explorer.

## Ejemplos de trabajo con fechas en Javascript

Visto lo anterior, ahora podemos poner los conocimientos en la práctica en un ejemplo completo de trabajo con objetos de la clase Date.

En este ejemplo vamos a crear dos fechas, una con el instante actual y otra con fecha del pasado. Luego las imprimiremos las dos y extraeremos su año para imprimirlo también. Luego actualizaremos el año de una de las fechas y la volveremos a escribir con un formato más legible.

```
//en estas líneas creamos las fechas
let miFechaActual = new Date()
let miFechaPasada = new Date(1998,4,23)

//en estas líneas imprimimos las fechas.
document.write (miFechaActual)
document.write ("<br>")
document.write (miFechaPasada)

//extraemos el año de las dos fechas
let añoActual = miFechaActual.getFullYear()
let añoPasado = miFechaPasada.getFullYear()

//Escribimos en año en la página
document.write("<br>El año actual es: " + añoActual)
document.write("<br>El año pasado es: " + añoPasado)

//cambiamos el año en la fecha actual
miFechaActual.setFullYear(2005)

//extraemos el día mes y año
let dia = miFechaActual.getDate()
let mes = parseInt(miFechaActual.getMonth()) + 1
let año = miFechaActual.getFullYear()

//escribimos la fecha en un formato legible
document.write ("<br>")
document.write (dia + "/" + mes + "/" + año)
```

Hay que destacar un detalle antes de acabar y es que el número del mes puede empieza desde 0. Por esta razón sumamos uno al mes que devuelve el método `getMonth()`.

Hay más detalles a destacar, pues resulta que en Javascript el método `getYear()` devuelve los años transcurridos desde 1900, con lo que al obtener el año de una fecha de, por ejemplo, 2005, indica que es el año 105. Si por ejemplo la fecha está en 2023 `getYear()` te devolvería 123.

Este es un comportamiento poco intuitivo pero que viene desde el navegador Netscape (quien introdujo Javascript en el pasado). En realidad el valor del año es el resultado de restarle 1900 porque esa fórmula se considera como "fecha universal". El lenguaje lo ha heredado hasta el día de hoy y seguirá así.



Dado lo anterior, debes recordar siempre que, para obtener el año completo tenemos a nuestra disposición el método `getFullYear()` que devolvería justo lo que estamos esperando, por ejemplo para una fecha de `2023` devolvería `2023`, lo que resulta más adecuado en la mayoría de los casos.

Mucha atención, pues, al trabajo con fechas en distintas plataformas, puesto que podría ser problemático el hecho de que nos ofrezcan distintas salidas los métodos de manejo de fechas, com siempre dependiendo de la marca y versión de nuestro navegador.

Se puede ver otros ejemplos de trabajo con la clase Date en el taller de Javascript. Te dejamos algunos ejemplos para complementar esta información.

- [Calcular la edad en Javascript](#)
- [5 formas de mostrar la fecha actual con Javascript](#)
- [Cómo hacer un reloj en Javascript](#)

Este artículo es obra de *Miguel Angel Alvarez*

Fue publicado / actualizado en *25/07/2023*

Disponible online en <https://desarrolloweb.com/articulos/clase-date-javascript.html>

## Clase Math en Javascript

### La clase que utilizamos para realizar cálculos matemáticos de todo tipo.

La clase Math es una de las [clases nativas de Javascript](#). Proporciona los mecanismos para realizar operaciones matemáticas en Javascript. Algunas operaciones se resuelven rápidamente con los operadores aritméticos que ya conocemos, como la multiplicación o la suma, pero hay una serie de operaciones matemáticas adicionales que se tienen que realizar usando la clase Math como pueden ser calcular un seno o hacer una raíz cuadrada.

De modo que para cualquier cálculo matemático complejo utilizaremos la clase Math, con una particularidad. Hasta ahora cada vez que queríamos hacer algo con una clase debíamos instanciar un objeto de esa clase y trabajar con el objeto y en el caso de la clase Math se trabaja directamente con la clase. Esto se permite porque las propiedades y métodos de la clase Math son lo que se llama propiedades y métodos de clase y para utilizarlos se opera a través de la clase en lugar de los objetos. Dicho de otra forma, para trabajar con la clase Math no deberemos utilizar la instrucción `new` y utilizaremos el nombre de la clase para acceder a sus propiedades y métodos.

### Propiedades de Math



---

Las propiedades guardan valores que probablemente necesitemos en algún momento si estamos haciendo cálculos matemáticos. Es probable que estas propiedades resulten un poco raras a las personas que desconocen las matemáticas avanzadas, pero los que las conozcan sabrán de su utilidad.

## E

Número E o constante de Euler, la base de los logaritmos neperianos.

## LN2

Logaritmo neperiano de 2.

## LN10

Logaritmo neperiano de 10.

## LOG2E

Logaritmo en base 2 de E.

## LOG10E

Logaritmo en base 10 de E.

## PI

Conocido número para cálculo con círculos.

## SQRT1\_2

Raiz cuadrada de un medio.

## SQRT2

Raiz cuadrada de 2.

## Métodos de Math

Así mismo, tenemos una serie de métodos para realizar operaciones matemáticas típicas, aunque un poco complejas. Todos los que conozcan las matemáticas a un buen nivel conocerán el significado de estas operaciones.

### abs()

Devuelve el valor absoluto de un número. El valor después de quitarle el signo.

### acos()

Devuelve el arcocoseno de un número en radianes.

**asin()**

Devuelve el arcoseno de un numero en radianes.

**atan()**

Devuelve un arcotangente de un numero.

**ceil()**

Devuelve el entero igual o inmediatamente siguiente de un número. Por ejemplo, ceil(3) vale 3, ceil(3.4) es 4.

**cos()**

Retorna el coseno de un número.

**exp()**

Retorna el resultado de elevar el número E por un número.

**floor()**

Lo contrario de ceil(), pues devuelve un número igual o inmediatamente inferior.

**log()**

Devuelve el logaritmo neperiano de un número.

**max()**

Retorna el mayor de 2 números.

**min()**

Retorna el menor de 2 números.

**pow()**

Recibe dos números como parámetros y devuelve el primer número elevado al segundo número.

**random()**

Devuelve un número aleatorio entre 0 y 1. Método creado a partir de Javascript 1.1.

**round()**

Redondea al entero más próximo.

**sin()**

Devuelve el seno de un número con un ángulo en radianes.

### sqrt()

Retorna la raíz cuadrada de un número.

### tan()

Calcula y devuelve la tangente de un número en radianes.

## Ejemplo de utilización de la clase Math

Vamos a ver un sencillo ejemplo sobre cómo utilizar métodos y propiedades de la clase Math para calcular el seno y el coseno de 2 PI radianes (una vuelta completa). Como algunos de vosotros sabréis, el coseno de 2 PI radianes debe dar como resultado 1 y el seno 0.

```
document.write (Math.cos(2 * Math.PI))

document.write ("<br>")

document.write (Math.sin(2 * Math.PI))
```

2 PI radianes es el resultado de multiplicar 2 por el número PI. Ese resultado es lo que recibe el método cos, y da como resultado 1. En el caso del seno el resultado no es exactamente 0 pero está aproximado con una exactitud de más de una millonésima de fracción. Se escriben los el seno y coseno con un salto de línea en medio para que quede más claro.

Podemos [ver el resultado de ejecutar estas líneas de código](#).

Además, si deseamos profundizar en la clase Math os recomiendo leer el [artículo de crear un número aleatorio](#). También se hace uso de la clase Math en el artículo [enlace aleatorio](#). Todos en el [Taller de Javascript](#).

Este artículo es obra de *Miguel Angel Alvarez*  
Fue publicado / actualizado en *25/04/2002*  
Disponible online en <https://desarrolloweb.com/articulos/762.php>

## Clase Number en Javascript

Clase que modeliza el tipo de datos numérico.



Vamos a ver a continuación otra de las [clases nativas de Javascript](#) para trabajo con datos numéricos. Se trata de la clase Number, que modeliza el tipo de datos numérico y fue añadida en la versión 1.1 de Javascript (con Netscape Navigator 3).

### Qué ofrece la clase Number

Como veremos en este artículo, la clase Number nos sirve para crear objetos que tienen datos numéricos como valor y realizar algunas operaciones sencillas a partir de sus métodos.

Lo cierto es que para trabajar con números generalmente usamos los operadores, por lo que es probable que no llegues a utilizar la clase Number, por lo menos en la mayoría de los scripts. Pero en resumen, con la clase Number tendremos algunas utilidades generales para trabajar con datos que tienen el tipo de datos numérico, como podrías imaginar. Viene bien conocerla para tener una noción más clara sobre lo que puedes hacer con el lenguaje.

Conocimos el [tipo de datos numérico en el primer manual de javascript](#). Este nos servía para guardar un valores numéricos sin más. Este objeto modeliza este tipo de datos y la clase en si, ofrece algún método que puede ser útil. Para los cálculos matemáticos y el uso de números en general vamos a utilizar siempre las variables numéricas vistas anteriormente.

### Constructor de la clase Number

La utilidad principal de la clase Number, al menos la que podrías usar con más frecuencia, es la de crear valores numéricos a través de valores que puedes tener en otros tipos de datos.

Para ello se usa el constructor de la clase Number, al que le enviamos por parámetro aquello que queremos convertir en un dato de tipo Number. El valor del objeto Number que se crea depende de lo que reciba el constructor de la clase Number. Con estas reglas:

- Si el constructor recibe un número, entonces inicializa el objeto con el número que recibe. Si recibe un número entrecomillado (lo que sería una cadena de caracteres con algo que se corresponda con un número) lo convierte a valor numérico, devolviendo también dicho valor, con el tipo de datos number.
- Devuelve `NaN` en caso de que no reciba nada.



- En caso de que reciba un valor no numérico devuelve NaN, que significa "Not a Number" (lo que se traduce por "No es un número" en español).
- Si recibe false se inicializa a 0 y si recibe true se inicializa a 1.

## Ejemplos de construcción de valores numéricos

Ahora veremos cómo podemos usar el constructor de la clase Number para conseguir producir datos donde podremos estar seguros que serán de tipo number. Su funcionamiento se puede resumir en estos ejemplos.

```
var n1 = new Number()  
document.write(n1 + "<br>")  
//muestra un 0  
  
var n2 = new Number("hola")  
document.write(n2 + "<br>")  
//muestra NaN  
  
var n3 = new Number("123")  
document.write(n3 + "<br>")  
//muestra 123  
  
var n4 = new Number("123asdfQWERTY")  
document.write(n4 + "<br>")  
//muestra NaN  
  
var n5 = new Number(123456)  
document.write(n5 + "<br>")  
//muestra 123456  
  
var n6 = new Number(false)  
document.write(n6 + "<br>")  
//muestra 0  
  
var n7 = new Number(true)  
document.write(n7 + "<br>")  
//muestra 1
```

## Propiedades de la clase Number

Esta clase también nos ofrece varias propiedades que contienen los siguientes valores:

### NaN

Como hemos visto, significa Not a Number, o en español, no es un número.

### MAX\_VALUE y MIN\_VALUE

Guardan el valor del máximo y el mínimo valor que se puede representar en Javascript

### NEGATIVE\_INFINITY y POSITIVE\_INFINITY

Representan los valores, negativos y positivos respectivamente, a partir de los cuales hay desbordamiento.

### Number.MAX\_SAFE\_INTEGER Number.MIN\_SAFE\_INTEGER



Son el máximo valor representable y el mínimo valor representable (negativo)

Podrás apreciar que estas constantes son muy similares a MIN\_VALUE y MAX\_VALUE. La diferencia es que MIN\_VALUE y MAX\_VALUE usan notación científica. En el caso de MIN\_SAFE\_INTEGER y MAX\_SAFE\_INTEGER este valor se expresa con un entero común.

Para una muestra más gráfica, los valores de estas constantes son los siguientes:

```
Number.MAX_SAFE_INTEGER  
9007199254740991
```

```
Number.MIN_SAFE_INTEGER  
-9007199254740991
```

```
Number.MAX_VALUE  
1.7976931348623157e+308
```

```
Number.MIN_VALUE  
5e-324
```

## Number.EPSILON

La distancia más pequeña dados dos números representables en Javascript.

Estas propiedades son de clase, así que accederemos a ellas a partir del nombre de la clase, tal como podemos ver en este ejemplo en el que se muestra cada uno de sus valores.

```
document.write("Propiedad NaN: " + Number.NaN)  
document.write("<br>")  
document.write("Propiedad MAX_VALUE: " + Number.MAX_VALUE)  
document.write("<br>")  
document.write("Propiedad MIN_VALUE: " + Number.MIN_VALUE)  
document.write("<br>")  
document.write("Propiedad NEGATIVE_INFINITY: " + Number.NEGATIVE_INFINITY)  
document.write("<br>")  
document.write("Propiedad POSITIVE_INFINITY: " + Number.POSITIVE_INFINITY)
```

## Métodos de la clase Number

Además la clase Number tiene algunos métodos, los cuales ya conoces prácticamente todos, porque hemos hablado de ellos en capítulos anteriores del manual. Los métodos más usados de esta clase los encuentras como funciones sueltas del lenguaje, por lo que no se suele usar mucho sus correspondientes en la clase Number.

### Number.isNaN()

Sirve para saber si lo que le envíes por parámetro es un "no número", es decir, devuelve `true` si el parámetro enviado es `NaN`.

```
Number.isNaN('hola') // devuelve false  
Number.isNaN(parseInt('hola')) // devuelve true
```

### Number.isFinite()



Sirve para saber si el parámetro enviado es un numérico y además es finito. Devuelve false si el valor es "infinity".

```
let x = 1/0;  
Number.isFinite(x); // devuelve false  
Number.isFinite(5); // devuelve true
```

### Number.isInteger()

Devuelve true si el número enviado es un entero.

```
Number.isInteger('9') // devuelve false  
Number.isInteger(9) // devuelve true
```

### Number.isSafeInteger()

Te dice si es un entero seguro.

### Number.parseFloat()

Esta función convierte cualquier cosa en un número en coma flotante, con decimales, igual que la función parseFloat().

```
Number.parseFloat('9.9')
```

### Number.parseInt()

La conocida función parseInt() dentro de la clase Number.

```
Number.parseInt('9.9') // devuelve 9  
Number.parseInt('hohoho') // devuelve NaN
```

## Conclusión

Hemos conocido una de las clases disponibles en el lenguaje, útil para el tratamiento de números, tanto enteros como reales.

En el siguiente artículo vamos a conocer otra clase similar, aunque en este caso para trabajar con los tipos de datos booleanos: [Clase Boolean en Javascript](#).

Este artículo es obra de *Miguel Angel Alvarez*  
Fue publicado / actualizado en 02/05/2023  
Disponible online en <https://desarrolloweb.com/articulos/778.php>

## Clase Boolean en Javascript

**Otra de las clases incorporadas en Javascript, en este caso para crear valores booleanos a partir de valores no booleanos.**

En este artículo presentaremos otra de las [clases nativas de Javascript](#), que es la clase Boolean. Esta clase nos sirve para crear valores booleanos y fue añadida en la versión 1.1 de Javascript (con Netscape Navigator 3).

Una de sus posibles utilidades es la de conseguir valores booleanos a partir de datos de cualquier otro tipo. No obstante, al igual que ocurría con la [clase Number](#), es muy probable que no la llegues a utilizar nunca.

**Nota:** conocimos el [tipo de datos boolean en el primer manual de Javascript](#). Este nos servía para guardar un valor verdadero (true) o falso (false). Esta clase modeliza ese tipo de datos para crear objetos booleanos.

Dependiendo de lo que reciba el constructor de la clase Boolean el valor del objeto booleano que se crea será verdadero o falso, de la siguiente manera

### Se inicializa a false

Cuando no pasas ningún valor al constructor, o si pasas una cadena vacía, el número 0 o la palabra false sin comillas.

### Se inicializa a true

Cuando recibe cualquier valor entrecomillado o cualquier número distinto de 0.

Se puede comprender el funcionamiento de este objeto fácilmente si examinamos unos ejemplos.

```
var b1 = new Boolean()  
document.write(b1 + "<br>")  
//muestra false  
  
var b2 = new Boolean("")  
document.write(b2 + "<br>")  
//muestra false  
  
var b25 = new Boolean(false)  
document.write(b25 + "<br>")  
//muestra false  
  
var b3 = new Boolean(0)  
document.write(b3 + "<br>")  
//muestra false  
  
var b35 = new Boolean("0")  
document.write(b35 + "<br>")  
//muestra true  
  
var b4 = new Boolean(3)
```



```
document.write(b4 + "<br>")  
//muestra true  
  
var b5 = new Boolean("Hola")  
document.write(b5 + "<br>")  
//muestra true
```

Se puede [ver en funcionamiento el ejemplo en una página a parte](#).

Este artículo es obra de *Miguel Angel Alvarez*

Fue publicado / actualizado en 14/05/2002

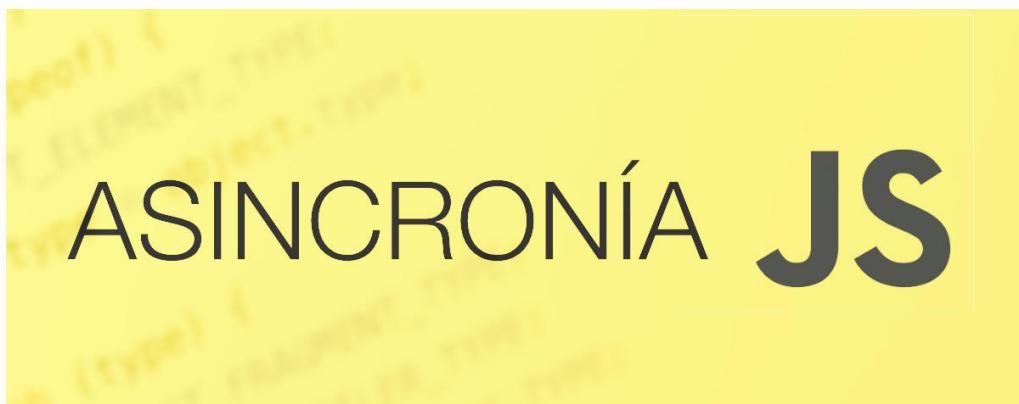
Disponible online en <https://desarrolloweb.com/articulos/777.php>

# Javascript y su comportamiento asíncrono

Hay algo que caracteriza a Javascript y que resulta complejo de entender para las personas que empiezan a programar en este lenguaje: su comportamiento asíncrono. Esta es una de las cuestiones que más quebraderos de cabeza dan a los programadores inexpertos que empiezan a desarrollar en Javascript y no tienen un suficiente conocimiento del lenguaje. Vamos a ver qué significa que Javascript sea asíncrono y cómo se gestiona la asincronía en Javascript.

## Javascript asíncrono

**Qué es Javascript asíncrono. Por qué es tan importante la asincronía en Javascript. Cómo es el código asíncrono y cuáles son los mecanismos para poder controlar los procesos asíncronos.**



Este va a ser un artículo de cultura general sobre Javascript y su **modelo de trabajo asíncrono**, algo esencial para dominar el lenguaje de programación por excelencia para la web. Veremos qué es la asincronía y esperamos ofrecer explicaciones claras y sencillas para que puedas entender las bases y despejar dudas.

### Qué es síncrono

Antes de estudiar qué es la asincronía vamos a entender **qué significa síncrono**, lo que sería el concepto contrario. Es útil empezar por aquí porque **nuestro modelo de pensamiento a la hora de programar es principalmente síncrono**. Tener claro este concepto nos ayudará a entender mejor su opuesto.

Algo síncrono según la RAE es algo que *coincide en el tiempo*. Desde nuestro punto de vista como programadores en Javascript nos aclara algunas cosas, pero debemos dejar claros algunos matices adicionales.



Como sabes, al escribir programas tenemos una serie de sentencias de código para definir los procesos o algoritmos. **Síncrono quiere decir que esas sentencias se ejecutarán una detrás de otra**, de manera continua en el tiempo, sin que existan pausas entre la ejecución de una y otra sentencia.

Esta es la manera común que tenemos de entender los programas y la manera como hemos aprendido generalmente un lenguaje de programación.

```
let x = 1;  
x++;  
console.log(x);
```

Todas estas sentencias se ejecutarán una detrás de otra, de manera prácticamente instantánea, pues el hilo de procesamiento de Javascript es muy rápido, pero siempre respetando el orden con el que se han escrito en el código. De modo que, al acabar de ejecutarse, aparecerá un mensaje en la consola con el valor de `x`, tal como ha quedado después de las tres sentencias.

## Qué es asíncrono

Por contra, **asíncrono significa que no se tiene por qué respetar el orden de las sentencias durante su ejecución en el tiempo**. Es decir, pueden aparecer en el código fuente en un orden y sin embargo se pueden llegar a ejecutar en otro orden distinto.

Esto quizás puede resultar un poco extraño, porque generalmente en programación tendemos a escribir algoritmos en los cuales se van realizando distintas cosas, paso a paso, para la consecución de un objetivo. Si las sentencias no se ejecutan en el orden en el que las hemos escrito lo normal sería pensar que los programas no producirán los resultados deseados. Sin embargo, en la práctica y en ciertas situaciones, la asíncronía de Javascript puede requerir que nosotros escribamos las sentencias de los programas en orden distinto a como se van a ejecutar.

El ejemplo más sencillo de asíncronía lo encontramos con la función `setTimeout()` de Javascript, que puede retrasar la ejecución de una función suministrada hasta pasados unos milisegundos.

## Entendiendo `setTimeout`

Por si acaso alguien no conoce la función `setTimeout()` vamos a ver un rápido ejemplo:

```
setTimeout( function() {  
    console.log('pasaron 2 segundos');  
, 2000);
```

En la invocación de la función `setTimeout()` suministramos dos argumentos.

- Una función con código a ejecutar
- Los milisegundos que deben de pasar hasta que se ejecute la función del primer parámetro.



Por lo tanto, en el código anterior veremos que el mensaje del console.log() se mostrará pasados dos segundos (2000 milisegundos).

Tienes decenas de ejemplos de setTimeout() a lo largo de todo desarolloweb, puedes usar el buscador para encontrar mucha más información.

## Ejemplos de código asíncrono con setTimeout

Ahora vamos a ver este código donde tenemos un ejemplo de asincronía, donde podemos comenzar a experimentar con esta característica de Javascript:

```
let x = 1;
setTimeout( function() {
  x++;
}, 1);
console.log(x);
```

En este ejemplo, que es muy similar al que hemos visto en el primer código expuesto en este artículo, por lo que imagino que lo entenderás. **Tenemos asincronía por el hecho de usar la función setTimeout()**. Por tanto, ¿Qué piensas que va a producirse de salida por consola al ejecutar el programa?

En realidad por consola nos aparecerá el valor 1, porque el incremento de x se realizará más tarde de mostrarse en consola, aunque sea solamente 1 milisecondo más tarde.

- Primero se crea la variable x con el valor 1
- Luego se define una función que se ejecutará más adelante. Ojo! se define, pero no se ejecuta todavía! aquí está la clave
- A continuación se muestra el valor de x, que no se ha modificado todavía desde su declaración
- Por último, se ejecuta la función que hace el incremento de la variable x

Realmente es muy fácil interpretar este código si llegaste a entender la función setTimeout(). Podríamos acudir a ejemplos un poco más complejos.

```
let x = 1;
setTimeout( function() {
  x = 3;
  console.log(x);
  x = 20;
}, 100);
setTimeout( function() {
  x +=10;
  console.log(x);
}, 10);
setTimeout( function() {
  console.log(x);
}, 1);
```



La salida de este programa será el valor 1, 11, 3. Tampoco tiene mucho misterio pero para deducirlo tienes que ir viendo el código por partes, mirando los milisegundos de cada setTimeout() y viendo el valor que tendrá la variable x en cada paso.

## Por qué lo asíncrono es tan importante en Javascript

Con Javascript nos encontramos ante un lenguaje que, desde su construcción, aporta unas **características muy relevantes y relacionadas con la asíncronía**. Básicamente todo parte del hecho de que en Javascript tenemos únicamente **un único hilo de ejecución de los programas**.

Hay lenguajes que, como PHP, son capaces de levantar diversos procesos para mantener hilos de ejecución paralelos. Por ejemplo, cada vez que un usuario consulta una página se levanta un proceso de ejecución de PHP para procesar la página, componer el HTML y devolverlo al usuario. Eso lo hace el servidor web sin que nosotros tengamos que programar nada, de modo que en la práctica el servidor podrá tener varios procesos de PHP ejecutándose al mismo tiempo, donde cada uno de ellos prepara el HTML que debería enviar a cada cliente que se conecta con la web.

Javascript, sin embargo, solamente tiene un único proceso funcionando, lo que en principio podría parecer una desventaja, pero en la práctica su gestión asíncrona hace que no sea un inconveniente. Incluso a veces puede significar una mejora significativa en el rendimiento de las aplicaciones!

## Comportamiento bloqueante del modelo síncrono

En lenguajes síncronos como PHP generalmente tenemos un comportamiento bloqueante. Vamos a entender qué significa esto con un ejemplo.

Cuando accedemos con PHP a una base de datos tendremos que realizar una consulta contra el sistema gestor de la base de datos. Este servidor **tardará en responder**, aunque sea unos milisegundos. Durante ese tiempo PHP estará esperando simplemente la respuesta del servidor de la base de datos y mientras que eso ocurre el hilo de ejecución del programa simplemente estará esperando ocioso. Dicho de otro modo, el proceso de ejecución del código estará bloqueado en esa sentencia y no hará nada (no ejecutará nada más) hasta que el sistema gestor de la base de datos le responda con el resultado de la operación solicitada.

En PHP estas esperas no impactan negativamente porque cada proceso de ejecución estará atendiendo a un usuario y no importa tanto que se tenga que esperar a la base de datos u otros trabajos que pueden hacer que el programa sea más lento. Sin embargo, si PHP tuviera un único hilo de ejecución para atender a todos los visitantes que pueda tener una web ocurriría que las esperas en las consultas a la base de datos bloquearán a todos los usuarios, porque hasta que no



se reciban los resultados de una consulta para atender a un usuario no podrían procesarse las páginas solicitadas por otros usuarios de la web.

## Lenguajes bloqueantes vs lenguajes no bloqueantes

Como hemos explicado **lenguajes como PHP son bloqueantes**. Es decir, que al realizar procesos pesados, que requieran un tiempo en ejecutarse (acceso a una base de datos, al sistema de archivos, a un API, etc) el proceso de ejecución se queda en estado bloqueado, esperando que ese proceso termine antes de continuar con la siguiente línea de código.

Sin embargo, existen los **lenguajes no bloqueantes, como Javascript**, que tienen otro comportamiento. En ellos, el tiempo en el que se espera a los procesos lentos, o no inmediatos, que a menudo no dependen del propio lenguaje sino de la respuesta de un sistema distinto (como el motor de la base de datos), **el lenguaje libera el hilo de ejecución**, de modo que es capaz de atender otras cuestiones.

**Gracias a su característica no bloqueante Javascript puede atender a muchas cosas a la vez**, aunque solo tenga un hilo de ejecución. Simplemente porque cualquier cosa que necesite hacer que no sea inmediata, libera el hilo de ejecución para que otras cosas se puedan ir realizando al mismo tiempo.

Aquí está la clave de Javascript y por qué consigue funcionar de manera fluida, incluso en máquinas con poca capacidad de procesamiento. Simplemente tiene un único proceso, por lo que no exige demasiado. Si hay cosas que no dependen de Javascript, no congela su funcionamiento y puede dedicarse a otras cosas.

Sin embargo, como programadores, para poder componer el código de aplicaciones en lenguajes no bloqueantes, debemos **entender muy bien la asincronía y acostumbrarnos a escribir el código de las aplicaciones de una manera diferente**, a menudo **más compleja** de lo que nos requieren los lenguajes bloqueantes.

Tendrás que gestionar la asincronía de Javascript para realizar tareas tan comunes como el acceso a las bases de datos, o el acceso al sistema de archivos en el lenguaje NodeJS o para realizar consultas a servicios web o APIs desde el navegador con Ajax, por poner algunos ejemplos.

Aunque Javascript sea un buen lenguaje para aprender a programar, porque tiene cantidad de aplicaciones directas, temas como la asincronía lo hacen más complejo. Muchas veces se dejan pasar en los tutoriales o cursos básicos, pero son importantes de dominar para poder sacarle partido en situaciones diversas.



## Mecanismos para la gestión de los procesos asíncronos

Ahora vamos a comentar brevemente los distintos mecanismos que nos ofrece Javascript para gestionar los procesos asíncronos, desde los más sencillos hasta los de más alto nivel.

### Funciones callback

Las funciones callback son funciones que se suministran a los procesos asíncronos, que contienen el código que debe ejecutarse una vez que acaban. Las funciones callback se envían como argumento a las funciones que realizan procesos asíncronos y el propio motor de Javascript se encargará de procesar esos callbacks cuando corresponda.

La función `setTimeout()` que hemos visto antes tenía una función callback, que es aquella que debe procesarse pasados un número de milisegundos. A menudo expresamos las funciones callbacks mediante funciones anónimas.

```
setTimeout( function() {
  console.log('Se está ejecutando la función callback');
}, 1000);
```

Si te interesa, en este artículo puedes profundizar un poco más sobre las [funciones callback](#) de Javascript.

### Promesas

Las promesas son herramientas un poco más avanzadas para organizar el código de las funciones asíncronas. Nos permiten de manera centralizada especificar qué queremos hacer cuando un proceso asíncrono se ha ejecutado correctamente y lo que se debería hacer cuando el proceso asíncrono ha resultado en una situación de error.

Las promesas puedes verlas como objetos que te devuelven los procesos asíncronos. Cuando un proceso asíncrono funciona devolviendo promesas, en lugar de alimentarlo con una función callback usaremos la promesa de retorno para definir qué queremos hacer en cada caso.

El flujo principal del código definirá qué es lo que se debe hacer cuando la promesa se resuelva, mediante los métodos `then()` y `catch()`.

Aquí podemos ver un código que usa el API [Fetch](#) para la realización de una solicitud Ajax. Fetch tiene la característica de funcionar devolviendo promesas.

```
fetch('https://jsonplaceholder.typicode.com/photos')
  .then(res => console.log(res))
  .catch(err => console.log(err));
```



Por supuesto, sabemos que para entender este código hacen falta las debidas explicaciones. Por eso, si te interesa, en este artículo puedes profundizar sobre las [promesas en Javascript](#).

## Async Await

Es el último modelo para gestionar los procesos asíncronos (por lo menos el último modelo de gestión asíncrona que ha aportado Javascript). Este modelo en realidad es lo que conocemos como un "azúcar sintáctico" que nos permite resolver los procesos asíncronos con un estilo de programación similar al que usaríamos en la programación síncrona.

Por tanto, las personas comúnmente preferirán implementar comportamientos usando Async / Await porque el código resultante resultará más claro en la mayoría de las ocasiones.

Para usar Async Await simplemente debemos marcar una función como asíncrona (con `async`) y entonces en ella podemos usar la palabra "await" cada vez que necesitemos esperar que se resuelva un proceso asíncrono.

Aquí te dejamos un ejemplo de cómo se usa Async / Await en una función que hace uso de [fetch](#), que es un proceso asíncrono para obtener datos por Ajax. La función `fetch` en realidad trabaja con promesas, pero gracias a Async / Await nos podemos ahorrar tratarlas con el bloque "then".

```
async function funcionAsincrona() {
  const response = await fetch('https://jsonplaceholder.typicode.com/todos/6')
  const json = await response.json()
  console.log(json)
}
```

No te preocupes si todavía no entiendes Async / Await. Tenemos un artículo para explicarlo con todo detalle: [async await en Javascript](#).

## Event loop de Javascript

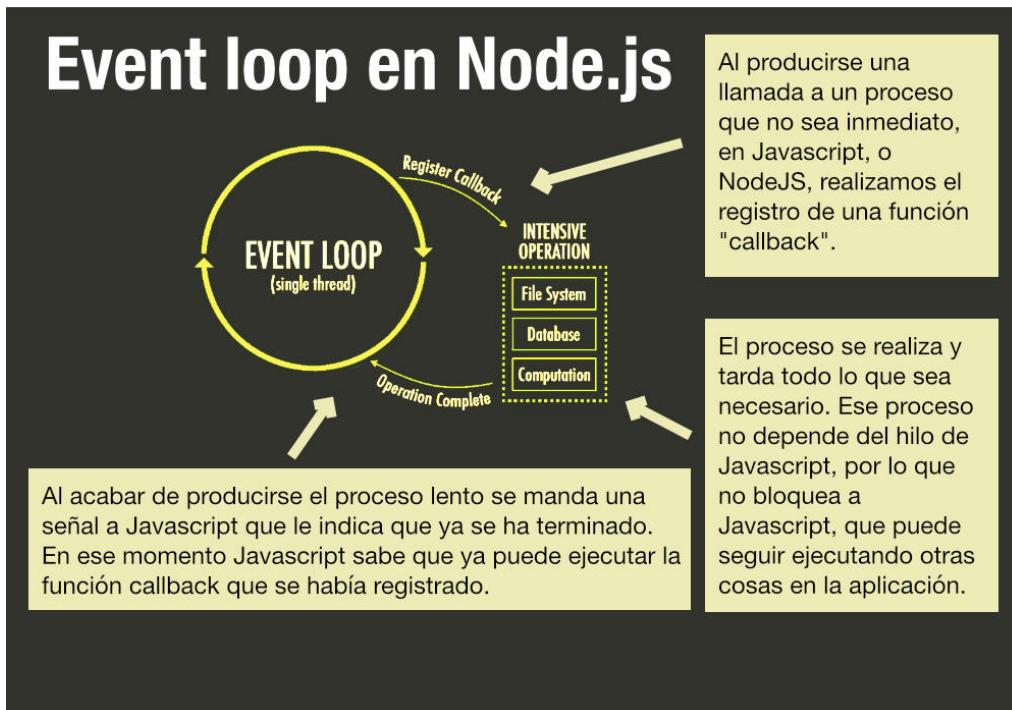
Para acabar este artículo vamos a tocar otro tema interesante relacionado con la gestión de la asíncronía realizada internamente por el lenguaje Javascript. Se trata del denominado Event Loop, que es quien atiende todos los procesos asíncronos, registra callbacks y las ejecuta cuando se necesita.



Este es un concepto que vemos con más detalle en el [Manual de NodeJS](#), ya que afecta mucho al tipo de programas que se hacen en esta plataforma.

El Event Loop no es más que un bucle que Javascript implementa de manera transparente para el desarrollador, que está siempre ejecutándose y atento a cualquier demanda de ejecución.

Aquí tenemos una imagen simplificada que se usa para explicar el event loop.



Simplemente queremos que notes en el diagrama que el event loop **se está ejecutando en un ciclo infinito**. Este es el hilo único de ejecución de Javascript, que se encarga de hacer todas las operaciones demandadas en el código de las aplicaciones.

Cuando una operación depende de un agente externo, como el acceso a la base de datos, simplemente se guarda la función callback, que debe ejecutarse al terminar. Este proceso puede durar un tiempo indeterminado, durante el cual el event loop se ocupa de hacer cualquier otra cosa, como por ejemplo atender otro request de otro usuario, o atender la interacción del usuario con la página, si está en un entorno de cliente.

Finalmente, cuando el proceso termina, el callback configurado para ejecución a su fin se procesa.

### Conclusión sobre la asincronía en Javascript

Esperamos que este artículo te haya aclarado algunas ideas sobre cómo es el lenguaje Javascript y cómo funciona. Es verdad que no hemos ahondado mucho en el código y te queda pendiente profundizar un poco más sobre los distintos modos de procesar la asincronía, pero el objetivo del



artículo era ofrecer algo de cultura general sobre Javascript. Espero que te haya resultado interesante.

Ahora, siguiendo los enlaces del propio artículo podrás encontrar mucha más información práctica.

Este artículo es obra de *Miguel Angel Alvarez*

Fue publicado / actualizado en 01/12/2023

Disponible online en <https://desarrolloweb.com/articulos/javascript-asincrono>

## Funciones Callback en Javascript

**Qué son las funciones callback, cómo especificar funciones callback y funciones anónimas para resolver procesos asíncronos de Javascript.**



Estamos haciendo una serie de artículos para explicar los procesos asíncronos en Javascript, una de las características más potentes que condicionan el desarrollo con este lenguaje de programación.

En este artículo vamos a ver las **funciones callback**, que constituyen un primer paso para trabajar y experimentar con la asincronía en el lenguaje.

### Qué es un callback

Un callback es una **función que se invoca cuando un proceso asíncrono ha terminado**. El propio nombre puede facilitar su comprensión: una función que se llama (call) de vuelta (back), para retornar el control de flujo al código asíncrono que estamos programando.

Date cuenta que los procesos asíncronos de Javascript tienen la particularidad de ejecutarse **liberando el proceso principal**. Por tanto, cuando el proceso asíncrono termina queremos que el proceso principal se entere y para ello se usan las funciones callback: Para devolver el control al iniciador del proceso asíncrono, enviando generalmente el dato resultante de la ejecución de ese proceso asíncrono.

### Qué es un proceso asíncrono

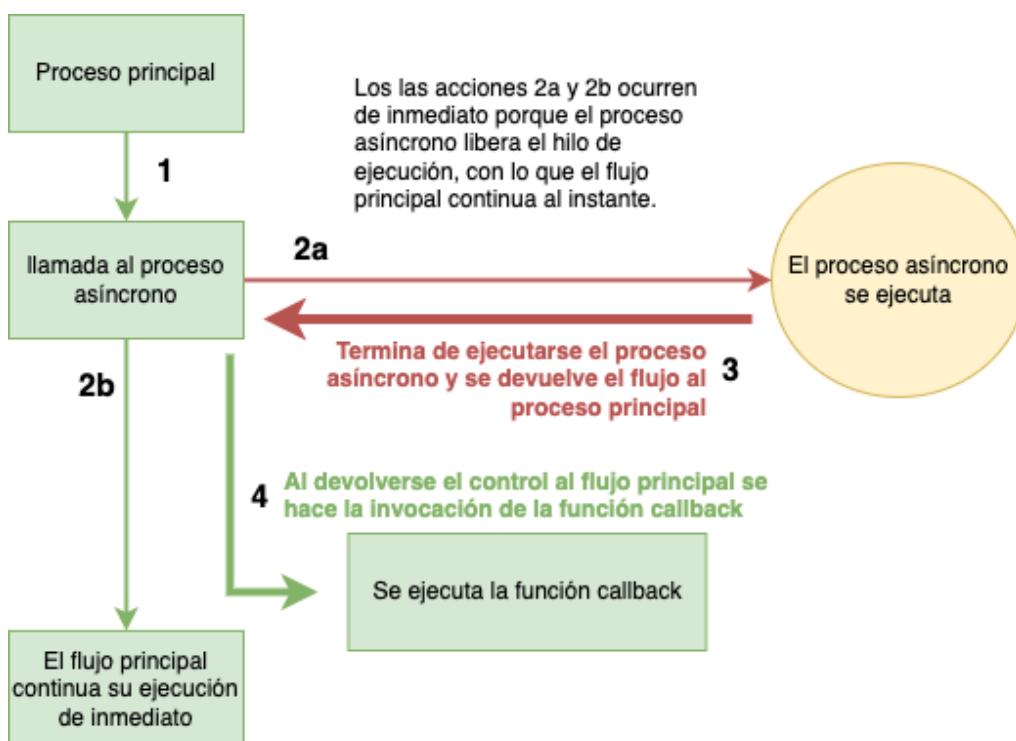


Por supuesto, para entender qué es un callback necesitamos saber qué es un proceso asíncrono. No es más que una operación realizada durante la programación de Javascript que tardará en ejecutarse. En realidad, todo proceso en general lleva un tiempo para su ejecución. Lo que caracteriza al proceso asíncrono es que **durante este tiempo de procesamiento libera el hilo de ejecución, devolviendo inmediatamente el flujo al programa principal**.

No vamos a dar más detalles en este momento porque esto ya lo hemos explicado anteriormente. Así que, para entenderlo mejor te recomendamos el artículo que explica [concepto de Javascript asíncrono](#).

### Cómo funcionan los callbacks de un proceso asíncrono

En el siguiente diagrama vamos a expresar el flujo de ejecución de un proceso asíncrono con una imagen, que esperamos sirva de guía.



Dentro del anterior diagrama tenemos etiquetados varias etapas del proceso, que describimos a continuación con más detalle.

- 1. Se está ejecutando el flujo principal hasta que llega un momento que se hace una llamada a un proceso asíncrono.
- 2a) Se hace una llamada a un proceso asíncrono
- 2b) Continúa la ejecución del proceso principal.
- 2a) y 2b) Estas acciones se ejecutan en secuencia, pero para nuestra percepción como humanos ocurren de inmediato, ya que el proceso asíncrono arranca pero toda la demora que podría producirse en él no se nota, ya que el flujo de ejecución se libera y con ello el flujo principal puede continuar su ejecución instantáneamente



- 3. El proceso asíncrono lleva su tiempo y una vez termina devuelve de nuevo el flujo de ejecución al proceso principal.
- 4. El proceso principal tiene entonces la opción de invocar la función callback, a la que generalmente se le envía el dato que el proceso asíncrono ha producido como respuesta. Por ejemplo, si fuera una llamada a la base de datos, se le entregaría el resultado de la ejecución de la consulta.

## Cuándo usamos las funciones callback

Las funciones callback son una de las alternativas existentes en Javascript para lidiar con los procesos asíncronos. Otras vías de trabajo las encontramos en las Promesas.

En Javascript encontramos numerosos ejemplos de uso de funciones callback, como por ejemplo en las solicitudes [Ajax](#). Al solicitar al servidor un recurso con Ajax el servidor puede tardar en responder. Durante ese tiempo el hilo de ejecución de Javascript puede dedicarse a otras cosas, como por ejemplo mostrar la animación del típico "loading". Una vez termina la solicitud y el navegador nos envía la respuesta, el navegador podrá usar la función callback que se haya definido, con el código que necesite ejecutarse en ese momento.

Así pues, el proceso asíncrono libera el hilo de ejecución y **la función callback es llamada para recuperar ese proceso**, realizando acciones una vez ya se tiene el dato que se necesitaba.

Ajax es el ejemplo más típico de asincronía en el navegador, pero en NodeJS los ejemplos son todavía más frecuentes, por ejemplo cuando se accede a las bases de datos, al sistema de archivos, etc.

Otro ejemplo muy básico de función callback lo encontramos en el uso del `setTimeout()` de Javascript. Enseguida veremos ejemplos precisamente de este caso.

## Cómo se definen las funciones callback

Habitualmente las funciones callback se definen mediante **funciones anónimas**, que son aquellas a las que simplemente nadie les ha puesto un nombre.

Para la realización de un proceso asíncrono generalmente tendremos que invocar una función que se encargará de producir ese proceso. A esa función le enviaremos como argumento la función anónima que hará de callback para retomar el control en el flujo principal de ejecución.

## Qué es una función anónima

Puedes haber visto en innumerables ocasiones el uso de funciones anónimas, por ejemplo al definir manejadores de eventos.

```
elemento.addEventListener('click', function() {
  // Esta función anónima se ejecutará cuando se hace clic sobre el elemento
```



});

Como has podido comprobar, esa función no tiene nombre. Se ejecutará cuando se haga clic, pero **nadie la ha declarado en un espacio de nombres**.

### Ejemplo de código asíncrono y su función callback

El ejemplo más sencillo de código asíncrono lo tenemos con la función `setTimeout()`, que se ejecuta pasados un número de milisegundos.

```
let x = 2;
setTimeout( function() {
  console.log("x vale", x);
}, 1000);
```

Este código indicará en la consola "x vale 2" pasado 1 segundo (lo equivalente a 1000 milisegundos).

Explicamos la función timeout en el artículo [qué es asíncrono en Javascript](#). Si no entiendes este código te sugiero que repases ese artículo.

### Callback con una función con nombre

Aunque lo normal sea usar funciones anónimas al registrar callbacks no quiere decir que no podamos usar también una función con nombre. De hecho, aquí vemos un código diferente al anterior pero que tiene como resultado una salida idéntica.

```
function miCallback() {
  console.log("x vale", x);
}

let x = 2;
setTimeout(miCallback, 1000);
```

En este punto lo que tienes que fijarte es que al invocar `setTimeout()` le estamos pasando la función completa como argumento. Al enviarla como argumento no se invoca la función, ya que no hemos colocado los paréntesis después de "`miCallback`".

Si le colocases los paréntesis lo que harías sería invocar la función inmediatamente, por lo que no se conseguiría ningún retardo en el mensaje.

### Definir un callback con una función flecha

Podemos usar arrow functions, también llamadas funciones flecha, para especificar el callback, lo que nos permitirá tener un código bastante más compacto.



```
let x = 2;
setTimeout( () => console.log("x vale", x), 3000);
```

Además de un código más compacto, las funciones flecha tienen una diferencia fundamental en el tratamiento de la palabra "this" dentro de la función. Si no lo conoces te recomendamos leer el artículo sobre las [arrow functions en Javascript](#).

## Otros ejemplos de callbacks en Javascript para el navegador

NodeJS está repleto de funciones que usan callbacks, porque el lenguaje se usa para muchos procesos asíncronos, que no dependen directamente de Javascript sino de sistemas externos, como el sistema de archivos o sistemas gestores de bases de datos. En el navegador hay menos ejemplos de funciones que requieran callbacks, comparativamente. Algunos de ellos los vamos a ver a continuación.

### Función setInterval()

La función `setInterval()` es muy parecida a `setTimeout()` con la diferencia de que el callback se invocará repetidas veces, cada vez que pase un intervalo de tiempo.

Por ejemplo, con el siguiente código conseguimos que haya un mensaje en la consola que nos diga el número de segundos que han pasado desde que empezó la ejecución.

```
let x = 0;
setInterval( function() {
  x++;
  console.log(`Ha pasado ${x} segundo/s`);
}, 1000);
```

En este ejercicio estamos usando otra cosa más o menos nueva de Javascript que son las [template string](#). Con ellas podemos interpolar el valor de variables de una manera cómoda en las cadenas.

## Objetos XMLHttpRequest para Ajax

Otro de los ejemplos típicos donde funcionamos con callbacks en Javascript es Ajax. El objeto `XMLHttpRequest` es la interfaz más antigua disponible en Javascript para realizar comunicaciones asíncronas contra servicios web, o sea: Ajax.

Con **Ajax** encontramos un "ejemplo de libro" en lo que respecta a la **asincronía en Javascript**, ya que las comunicaciones con otros servidores son el ejemplo más típico de procesos que van a tardar un rato. Durante ese tiempo de espera antes de recibir la respuesta del servidor, algo no depende del propio lenguaje Javascript, el hilo de ejecución debe liberarse para que el navegador pueda atender otras cuestiones.



El caso es que el objeto `XMLHttpRequest` tiene una manera diferente de definir las funciones callback, básicamente porque lo podemos configurar con múltiples funciones, para cada uno de los distintos casos posibles de respuesta. Por ejemplo, cuando una llamada al servidor recibió una respuesta correcta o cuando hubo un error.

Básicamente lo que se hace con `XMLHttpRequest` es definir una serie de eventos que nos permiten ejecutar código cuando pasan cosas en el objeto `XMLHttpRequest`. Cada uno de esos eventos los puedes asociar con manejadores de eventos, que se ejecutarán cuando éstos ocurran.

Podríamos decir que los manejadores de eventos son una especie de callbacks. Aunque sería justo aclarar que **son cosas distintas**. Sin embargo, en el fondo son funciones que se registran para ejecutarse cuando ocurren cosas, por lo que las vamos a ver en este artículo dedicado a callbacks. Es verdad que técnicamente `XMLHttpRequest` funciona con eventos, pero en el fondo los podemos entender como callbacks. Solo habría una **diferencia entre eventos y callbacks**. Mientras que los callbacks a menudo reciben de vuelta el dato que se ha conseguido por medio del proceso asíncrono, los manejadores de eventos reciben por parámetro el objeto evento.

En el siguiente código encuentras un ejemplo de trabajo con `XMLHttpRequest` y la definición de una función anónima como manejador de evento `onload`. Como decía, no es exactamente un callback pero funciona de manera muy similar.

```
// creación del objeto XMLHttpRequest
let xhr = new XMLHttpRequest();
// definición de la consulta Ajax, asíncrona, que vamos a realizar
xhr.open("GET", 'https://jsonplaceholder.typicode.com/todos/1');

// definición de un manejador de eventos al recibir el resultado de la llamada ajax
xhr.onload = function(e) {
    // Miro si la llamada está finalizada y el estado es 404
    if(xhr.readyState == 4 && xhr.status == 404) {
        console.log('Error de recurso no encontrado');
    }
    // muestro el cuerpo de la respuesta
    console.log(xhr.responseText);
}

// ejecuto la llamada
// nota que por el hecho de hacer la llamada a open() solo se configura la solicitud
// para que realmente se realice hay que llamar a send()
xhr.send();
```

## Otros mecanismos de trabajo con los procesos asíncronos

Durante mucho tiempo los procesos asíncronos se gestionaron únicamente mediante callbacks. Actualmente se siguen usando mucho, pero existen otras maneras de gestionarlos en Javascript, un poco más avanzadas y útiles, ya que nos permiten una mejor organización del código.

El siguiente punto que te recomiendo estudiar son las [promesas de Javascript](#), pero también existe el `async await` para gestionar los procesos asíncronos.

Este artículo es obra de *Miguel Angel Alvarez*

Fue publicado / actualizado en *20/03/2023*

Disponible online en <https://desarrolloweb.com/articulos/funciones-callback-javascript>

## Promesas en Javascript

**Explicaciones detalladas sobre las promesas en Javascript. Qué son las promesas, cómo usarlas en Javascript. Cómo hacer funciones que devuelven promesas y cómo resolver promesas en secuencia y en paralelo.**



**Las promesas son herramientas de los lenguajes de programación que nos sirven para gestionar situaciones futuras en el flujo de ejecución de un programa.** Aunque es un concepto que usamos en Javascript desde hace un relativamente corto espacio de tiempo, ya se viene implementando en el mundo de la programación desde la década de los 70.

Las promesas **se originaron en el ámbito de la programación funcional**, aunque diversos paradigmas las han incorporado, generalmente para gestionar la programación asíncrona. En resumen, **nos permiten definir cómo se tratará un dato que sólo estará disponible en un futuro**. Es decir, mediante las promesas podemos indicar qué se realizará con ese dato más adelante cuando haya sido devuelto por un proceso asíncrono.

Ahora con ES6 podemos beneficiarnos de las ventajas de las promesas a la hora de escribir un código más limpio y claro. De hecho son una de las [novedades más destacadas de ES6](#).





Aunque en Javascript las promesas se introdujeron en ES6 (estándar ECMAScript 2015), lo cierto es que se vienen usando desde hace tiempo, ya que varias librerías las habían implementado para solucionar sus necesidades de una manera más elegante. Aunque ahora estas bibliotecas independientes para implementar promesas en Javascript están en desuso, durante un tiempo se utilizaron bastante, como único propósito facilitar la mejora del código asíncrono, antes que el propio Javascript "Vanilla" las incorporarse.

Para abordar las **Promesas en Javascript** con todo el detalle que se merece hemos estructurado este artículo en los siguientes apartados de interés.

- [Cómo usar promesas](#)
- [Pirámide de callbacks](#)
- [Escribir funciones que devuelven promesas](#)
- [Encadenar promesas](#)
- [Conclusión](#)
- [Encadenar promesas transmitiendo datos de unas a otras](#)
- [Desarrollar funciones que devuelven promesas](#)
- [Funciones resolve y reject](#)
- [Cómo implementar una conexión Ajax con fetch que devuelve un texto](#)
- [Promesas Javascript en secuencia o en paralelo](#)
- [Promesas en secuencia](#)
- [Encadenar más promesas en secuencia](#)
- [Ejecutar promesas en paralelo](#)
- [Conclusión](#)

## Cómo usar promesas

Creo que para comenzar a entender las promesas nos viene muy bien comenzar viendo cómo podemos gestionarlas, es decir, qué se debe hacer cuando ejecutamos funciones que nos devuelven promesas. Luego aprenderemos a crear funciones que implementan promesas y veremos que es también bastante fácil.

Por ejemplo, la función set() de Firebase, para guardar datos en la base de datos en tiempo real, devuelve una promesa cuando se realiza una operación de escritura.

**Nota:** Da igual que no conozcas Firebase, y aunque puedes aprender en el [Manual de Firebase](#), no es necesario para entender las promesas. Simplemente quiero que se entienda cómo gestionar una promesa. Olvida que esta función pertenece al API de Firebase, concéntrate en el esquema de trabajo de promesas, que es siempre el mismo.



Tiene sentido que se use una promesa porque, aunque Firebase es realmente rápido, siempre va a existir un espacio de tiempo entre que solicitamos realizar una escritura de un dato y que ese dato se escribe realmente en la base de datos. Además, la escritura podría dar algún tipo de problema y por tanto producirse un error de ejecución, que también deberíamos gestionar. Todas esas situaciones se pueden implementar por medio de dos métodos:

- **then:** usado para indicar qué hacer en caso que la promesa se haya ejecutado con éxito.
- **catch:** usado para indicar qué hacer en caso que durante la ejecución de la operación se ha producido un error.

Ambos métodos debemos usarlos pasándoles la función callback a ejecutar en cada una de esas posibilidades.

```
referenciaFirebase.set(data)
  .then(function(){
    console.log('el dato se ha escrito correctamente');
  })
  .catch(function(err) {
    console.log('hemos detectado un error', err);
  });
});
```

Fíjate que "referenciaFirebase.set(data)" nos devuelve una promesa. Sobre esa promesa encadenamos dos métodos, then() y catch(). Esos dos métodos son para gestionar el futuro estado de la escritura en Firebase y están encadenados a la promesa. Por si acaso no se entiende eso, podríamos leer este mismo código de esta manera.

```
referenciaFirebase.set(data).then(function(){
  console.log('el dato se ha escrito correctamente');
}).catch(function(err) {
  console.log('hemos detectado un error', err);
});
```

Igual así se ve mejor qué es lo que me refiero cuando digo que están encadenados. Insisto en esto para que nos demos cuenta que **los then() y catch() forman parte de la misma cosa** (la promesa devuelta por el método set() de Firebase). Además, porque encadenar promesas es algo bastante normal y así nos vamos familiarizando mejor con cosas que usaremos en un futuro próximo.

**Nota:** Estoy escribiendo código más parecido a ES5, porque así nos quedamos solo con la parte nueva, las promesas y no nos despistamos con sintaxis que quizás todavía no tienes perfectamente asimilada como las [arrow functions](#). Pero obviamente, esas funciones callback enviadas a then() y catch() podrían ser perfectamente expresadas con arrow functions de ES6.

## Datos devueltos por las promesas



Otro detalle que no debe pasar desapercibido es que la promesa puede devolver datos. Es muy normal que esto ocurra. Por ejemplo queremos recibir algo de una base de datos y cuando la promesa se ejecuta correctamente querremos que nos lleve ese dato buscado. No es el caso en este método set() de Firebase, porque una operación de escritura no te devuelve nada en esta base de datos, pero insisto que es algo bastante común.

En el caso que la promesa te devuelva un dato, lo podrás recibir como parámetro en la función callback que estás adjuntando al then().

```
funcionQueDevuelvePromesa()
  .then( function(datoProcesado){
    //hacer algo con el datoProcesado
  })
```

**Nota:** Como estás viendo, al usar una promesa no estoy obligado a escribir la parte del catch, para procesar un posible error, ni tan siquiera la parte del then, para el caso positivo.

En el caso negativo implementado mediante el catch() siempre vamos a recibir un dato, que es el error que se ha producido al ejecutar la promesa y el causante de estar procesándose el correspondiente catch. Volviendo al ejemplo de antes, método set(), observa que el error lo hemos recibido en el parámetro de la función callback indicada en el catch().

## Pirámide de callbacks

Seguro que habrás oído hablar del código spaghetti. Uno de los síntomas en Javascript de ello es lo que se conoce como pirámide de callbacks o "callback hell". Hay mucha literatura y ejemplos en Javascript sobre ello. Ocurre cuando quieres hacer una operación asíncrona, a la que le colocas un callback para continuar tu ejecución. Luego quieres encadenar una nueva operación cuando acaba la anterior y otra nueva cuando acaba ésta.

El método setTimeout() de toda la vida en Javascript nos sirve para escribir algo de código spaghetti y ver la temida pirámide de callbacks.

```
setTimeout(function() {
  console.log('hago algo');
  setTimeout(function() {
    console.log('hago algo 2');
    setTimeout(function() {
      console.log('hago algo 3');
      setTimeout(function() {
        console.log('hago algo 4');
      }, 1000)
    }, 1000)
  }, 1000)
}, 1000);
```

En resumen lo que hacemos es encadenar una serie de tareas, para realizarlas secuencialmente, una cuando acaba la otra. Funciona, pero ese código es un infierno para mantener, pues tiene difícil



lectura y cuesta meterle mano para implementar nuevas funcionalidades. Las promesas nos pueden ayudar a mejorarlo, pero primero vamos a tener que aprender a implementarlas nosotros mismos.

## Escribir funciones que devuelven promesas

Ahora viene otra parte interesante, en la que aprendemos a crear nuestras propias funciones que devuelven promesas. Esto se consigue mediante la creación de un nuevo objeto "Promise", como veremos a continuación. Pero antes de ponernos con ello debes tener bien claro el objetivo de una promesa: "hacer algo que dura un tiempo y luego tener la capacidad de informar sobre posibles casos de éxito y de fracaso"

Ahora verás el código y aunque pueda parecer confuso al principio, la experiencia usando promesas te lo irá clarificando naturalmente. Ten en cuenta que para crear un objeto "Promise" voy a tener que entregarle una función, la encargada de realizar ese procesamiento que va a tardar algo de tiempo. En esa función debo ser capaz de procesar casos de éxito y fracaso y para ello recibo como parámetros dos funciones:

- **La función "resolve":** la ejecutamos cuando queremos finalizar la promesa con éxito.
- **La función "reject":** la ejecutamos cuando queremos finalizar una promesa informando de un caso de fracaso.

```
function hacerAlgoPromesa() { return new Promise( function(resolve, reject){  
    console.log('hacer algo que ocupa un tiempo...'); setTimeout(resolve, 1000); }) }
```

Como puedes ver, nuestra función hacerAlgoPromesa() devolverá siempre una promesa (return new Promise). Se encarga de hacer alguna cosa, y luego ejecutará el método resolve (1 segundo después, gracias al serTimeout).

**Nota:** Aun no estamos controlando posibles casos de fracaso, pero de momento está bien para no liarnos demasiado.

Esa misma función algunos programadores la preferirían ver escrita de este otro modo.

```
function hacerAlgoPromesa(tarea) {  
    function haciendoalgo(resolve, reject) {  
        console.log('hacer algo que ocupa un tiempo...');  
        setTimeout(resolve, 1000);  
    }  
    return new Promise( haciendoalgo );  
}
```

Es exactamente lo mismo que teníamos antes, solo que se ha ordenado el código de otra manera. Usa la alternativa que veas más clara.



Ahora vamos a ver cómo ejecutar esta función que nos devuelve una promesa, aunque si entendiste el principio del artículo ya lo tendrás bastante claro.

```
hacerAlgoPromesa()
  .then( function() {
    console.log('la promesa terminó.');
  })
```

Un poco más adelante en este mismo artículo profundizaremos sobre este asunto y mostraremos nuevos ejemplos sobre cómo crear tus propias funciones que devuelven promesas y veremos cómo tratar los casos negativos y rechazar promesas con `reject()`.

## Encadenar promesas

Como colofón a esta introducción a las promesas de ES6 queremos ver cómo nos facilitan la vida, creando un código mucho más limpio y entendible que la famosa pirámide de callbacks que hemos visto en un punto anterior. Si no estás familiarizado con el tema estoy seguro que te sorprenderás. Para que sea así, vamos directamente con el código fuente:

Imagina que quieres hacer algo y repetirlo por cuatro veces, ejecutando la función `hacerAlgoPromesa()` repetidas veces, de manera secuencial, una después de la otra.

```
hacerAlgoPromesa()
  .then( hacerAlgoPromesa )
  .then( hacerAlgoPromesa )
  .then( hacerAlgoPromesa )
```

Eso, comparado con el spaghetti code de antes, tiene su diferencia ¿no? y es básicamente lo mismo, ejecutar una acción 4 veces con un retardo entre ellas.

**Nota:** Obviamente en la realidad generalmente no repites lo mismo cuatro veces la misma operación, aunque podría ser, sino que puedes encadenar cuatro promesas distintas, una detrás de la otra, para realizar varias tareas diferentes.

## Encadenar promesas transmitiendo datos de unas a otras

A partir de aquí queda todavía por abordar diferentes puntos interesantes y útiles, como controlar posibles casos de error e informar de ellos en nuestras promesas, o poder ejecutar varias promesas en paralelo, en vez de secuencialmente. Todo eso lo iremos tratando, aunque espero que con este artículo se te abra un poco de luz y que puedas apreciar algunas de las ventajas de usar promesas ES6.

Si quieres investigar algo más sobre encadenar promesas, piensa que a veces a las funciones que devuelven promesas les tienes que pasar parámetros. ¿Cómo escribirías el chaining de promises? Para ser más claros, echa un vistazo a esta promesa.



```
function hacerAlgoPromesa2(tarea) {
  function haciendoalgo(resolve, reject) {
    console.log('Hacer ' + tarea + ' que ocupa un tiempo...');
    setTimeout(resolve, 1000);
  }
  return new Promise( haciendoalgo );
}
```

Es casi casi lo mismo que teníamos antes, solo que ahora le podemos pasar la tarea que quieras realizar. Lo que queremos es encadenar cuatro tareas diferentes, para ejecutar en secuencial, igual que antes. Pero tienes que pasarles parámetros distintos.

La solución la encuentras en el siguiente pedazo de código.

```
hacerAlgoPromesa('documentar un tema')
  .then(function() {
    return hacerAlgoPromesa('escribir el artículo')
  })
  .then(function() {
    return hacerAlgoPromesa('publicar en desarroloweb.com')
  })
  .then(function() {
    return hacerAlgoPromesa('recibir vuestro apoyo cuando compartís en vuestras redes sociales')
  })
```

Échale un vistazo y trata de entenderlo. La clave es que para encadenar promesas la función a ejecutar como callback debe devolver también una nueva promesa.

Nos hemos quitado de en medio la pirámide de callbacks, ero tampoco creas que sería el mejor código para resolver este problema. Ya que estamos en un Manual de ES6, no queremos perder la oportunidad de mostrar un ejemplo del azúcar sintáctico que nos ofrecen las Arrow Functions.

Este código sería equivalente al anterior:

```
hacerAlgoPromesa('documentar un tema')
  .then(() => hacerAlgoPromesa('escribir el artículo'))
  .then(() => hacerAlgoPromesa('publicar en desarroloweb.com'))
  .then(() => hacerAlgoPromesa('...compartís en vuestras redes sociales'))
```

Mucho más limpio, no?

Ahora ya conoces una de las mejoras más interesantes de la versión de ECMAScript 2015 (ES6), que es la posibilidad de gestionar el código asíncrono por medio de promesas. Este estilo de programación facilita mucho la legibilidad y el mantenimiento del código.

Hemos podido aprender lo básico de las promesas, centrándonos en cómo usar promesas y por qué son tan útiles en Javascript. De todos modos, para sacarles de verdad partido y resolver muchas de las necesidades de tus aplicaciones tendrás que aprender a crear tus propias funciones asíncronas que devuelven promesas y usar Resolve / Reject.

## Desarrollar funciones que devuelven promesas



Hasta este punto hemos aprendido lo que son las promesas y cómo usar funciones que devuelven promesas con `then` y `catch` para organizar nuestro código. Vimos que en el lenguaje Javascript es fácil caer en lo que se conoce como "código spaghetti" y que las promesas nos ofrecen una vía excelente para organizar nuestro código evitando la pirámide de callbacks. También vimos algunos ejemplos sobre cómo crear funciones que devuelven promesas, pero nos quedamos en lo elemental.

Ahora vamos a ampliar la información en lo referente a la implementación de una función que devuelve una promesa, tratando tanto los casos positivos (éxito) como los casos negativos (fracaso). Además ahondaremos en otra información importante de lenguaje Javascript y la nueva API de acceso a recursos Ajax: `fetch`.

**Nota:** Si no conoces `fetch` también te recomendamos estudiarlo antes de abordar este artículo, en el [artículo de introducción a `fetch`](#).



## Funciones `resolve` y `reject`

Cuando implementamos una función que devuelve una promesa tenemos a nuestra disposición dos funciones que permiten devolver el control al código que invocó la promesa. Esas funciones devuelven datos cuando la promesa se ejecutó normalmente y produjo los resultados esperados y cuando la promesa produjo un error o no pudo alcanzar los resultados deseados.

**Nota:** En el [artículo de Introducción a las promesas de ES6](#) ya vimos cómo usar el `resolve` para devolver el control en el caso positivo, pero no hemos visto cómo usar el `reject` para informar de un caso negativo.

El código de una función que devuelve una promesa tiene una forma inicial como esta:

```
function devuelvePromesa() {
    return new Promise( (resolve, reject) => {
        //realizamos nuestra operativa...
    })
}
```

Como puedes ver, este código devuelve una nueva promesa. Para crear la promesa usamos "new Promise". El constructor de la promesa lo alimentamos con una función que recibe dos parámetros:



resolve y reject, que son a su vez funciones que podremos usar para devolver valores tanto en el caso positivo como en el caso negativo.

**Nota:** Obviamente esos parámetros los puedes llamar como te apeteza, ya que son solo parámetros, pero los nombres indicados ya nos indican para qué sirve cada uno.

Ahora veamos nuestra operativa, que podría tener una forma como esta:

```
function devuelvePromesa() {
  return new Promise( (resolve, reject) => {
    setTimeout(() => {
      let todoCorrecto = true;
      if (todoCorrecto) {
        resolve('Todo ha ido bien');
      } else {
        reject('Algo ha fallado')
      }
    }, 2000)
  })
}
```

En nuestro código realizaremos cualquier tipo de proceso, generalmente asíncrono, por lo que tendrá un tiempo de ejecución durante el cual se devolverá el control por medio de una función callback. En la función callback podremos saber si aquel proceso se produjo de manera correcta o no. Si fue correcto usaremos la función resolve(), mandando de vuelta como parámetro el valor que se haya conseguido como resultado. Si algo falló usaremos la función reject(), enviando el motivo del error.

Podremos usar esa función que devuelve la promesa con un código como este:

```
devuelvePromesa()
  .then( respuesta => console.log(respuesta) )
  .catch( error => console.log(error) )
```

Como ya supondrás, then() recibe la respuesta indicada en el resolve() y catch() el error indicado en el reject.

### Cómo implementar una conexión Ajax con fetch que devuelve un texto

En el [artículo de fetch](#) vimos que es un nuevo modelo de trabajo con Ajax, pero usando promesas. Vimos que un fetch() te devuelve una respuesta del servidor, con datos sobre la solicitud HTTP, pero si lo que queríamos es acceder al texto de la respuesta, necesitábamos encadenar una segunda promesa, llamando al método text() sobre la respuesta. Esa segunda promesa nos complicó un poco el código de algo tan sencillo como es: dada una URL recibir el texto que hay en el recurso.

A continuación vamos a poner un código de alternativa, que realiza ambas promesas y directamente nos devuelve el texto de respuesta. Como es un código asíncrono, que tardará un poco en ejecutarse



y después de ello debe devolver el control al script original, lo implementaremos por medio de una promesa.

```
function obtenerTexto(url) {
  return new Promise( (resolve, reject) => {
    fetch(url)
      .then(response => {
        if(response.ok) {
          return response.text();
        }
        reject('No se ha podido acceder a ese recurso. Status: ' + response.status);
      })
      .then( texto => resolve(texto) )
      .catch (err => reject(err));
  });
}
```

Este código usa el modelo de encadenado de promesas, ejecutando operaciones asíncronas, una cuando termina la anterior.

Comenzamos haciendo el `fetch()` a una URL que recibimos por parámetro. Ese `fetch` devuelve una promesa, que cuando se ejecuta correctamente nos entrega la respuesta del servidor.

Si la respuesta estuvo bien (`response.ok` es `true`) entonces devuelve una nueva promesa entregada por la ejecución de la función `response.text()`. Si la respuesta no estuvo bien, entonces rechazamos la promesa con `reject()`, indicando el motivo por el que estamos rechazando con un error.

A su vez el código de `response.text()`, que devolvía otra promesa, puede dar un caso de éxito o uno de error. El caso de éxito lo tratamos con el segundo `then()`, en el que aceptamos la promesa con el `resolve`.

Tanto para el caso de error de `fetch()` como para un caso de error en el método `text()`, como para cualquier otro error detectado (incluso un código mal escrito) realizamos el correspondiente `catch()`, rechazando nuestra promesa original.

**Nota:** Observa que un solo `catch()` te sirve para detectar cualquier tipo de error, tanto en la primera promesa (`fetch`) como en la segunda (`text`).

Este código lo puedes usar de la siguiente manera. Verás que tenemos un único método que nos devuelve directamente el texto.

```
obtenerTexto('test.txt')
  .then( texto => console.log(texto) )
  .catch( err => console.log('ERROR', err) )
```

## Procesamiento de múltiples Promesas Javascript

# Promesas

## secuencia y paralelo

 JS

Ya sabes usar promesas e implementar funciones que devuelven promesas. Es un gran paso para mejorar el código asíncrono de las aplicaciones Javascript. Sin embargo todavía hay más que deberías saber para sacarles todo el partido.

A continuación vamos a ir un paso más allá, viendo ejemplos de trabajo con promesas un poco más complejos, combinando varias ejecuciones. Y es que muchas veces las promesas no vienen por separado, sino que tienes que ejecutar varias promesas una detrás de otra, porque unas dependan entre sí, o ejecutarlas todas a la vez y recibir una señal cuando se ha completado el conjunto entero.

### Promesas en secuencia

En muchos casos unas promesas dependen de otras, es decir, tienes que esperar que una promesa haya terminado y, con su resultado, ejecutar otra. En estos casos decimos que las promesas deben ejecutarse en secuencia.

El ejemplo más típico de realización de una promesa en secuencia es el que hacemos para resolver una solicitud [Ajax mediante Fetch](#), dado que para obtener el JSON de un servicio web necesitamos encadenar dos promesas. La primera para recibir una respuesta del servidor y la segunda para procesar esa respuesta y quedarnos con el cuerpo en un objeto.

La manera de ejecutar promesas en secuencia es encadenar varios "then", como puedes ver aquí.

```
fetch("https://jsonplaceholder.typicode.com/todos/")
  .then( response => response.json() )
  .then( json => console.log(json) )
```

La función `fetch()` devuelve una promesa, que se resuelve cuando se recibe la respuesta del servidor. El primer "then" recibe la respuesta del servidor. Puedes examinar la respuesta para ver si es válida o verificar cualquier otra cosa. Sin embargo aquí solamente ejecutamos `response.json()` para obtener el objeto de respuesta. El método `response.json()` devuelve otra promesa que se encadena. El segundo "then" se ejecuta cuando se resuelve la segunda promesa y en él recibimos ya el json procesado y podemos hacer cualquier cosa con él.

Nota: Si te interesa explorar más esta modalidad de trabajo con Ajax te recomendamos el [artículo sobre fetch](#).



Date cuenta que, siempre que devolvamos una promesa nueva, podremos enganchar los "then", que esperarán a que la promesa se resuelva para continuar.

## Encadenar más promesas en secuencia

El ejemplo se puede complicar todo lo que queramos. Ahora vamos a acceder a dos servicios web y esperar 5 segundos entre uno y otro.

Para el acceso a los servicios web usaremos fetch, igual que antes. Entre medias haremos una pausa para poder espaciar estos accesos. Para poder ver todo esto con distintas promesas hemos creado una función que devuelve una promesa, que simplemente se resuelve después de una espera.

```
const esperar = tiempo => {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      resolve('ok');
    }, tiempo);
  })
}
```

Ahora veamos el código que ejecuta toda la secuencia de promesas de la que hemos hablado. Las dos promesas del acceso al primer servicio web, la espera de 5 segundos y finalmente las dos promesas de acceso al segundo servicio web.

```
fetch("https://jsonplaceholder.typicode.com/todos/")
  .then( response => response.json())
  .then( json => console.log(json))
  .then(() => esperar(5000))
  .then( res => console.log(res))
  .then(() => fetch("https://pokeapi.co/api/v2/pokemon/1"))
  .then( response => response.json())
  .then( json => console.log(json));
```

¿Qué te parece? normalmente no tienes tantas promesas que ejecutar en secuencia, pero si fuera el caso puedes hacer un código bastante compacto y evitar lo que se llama la pirámide de callbacks.

Generalmente, cada una de las funciones de los "then" devuelven nuevas promesas que se irán ejecutando cuando acabe la anterior. Sin embargo, esto no tiene por qué ser siempre así. Por ejemplo, por en medio hemos colocado una función de un "then" que no cumple esta norma. ¿La encuentras?

Seguramente la hayas visto, pero si no, es la siguiente:

```
.then( res => console.log(res))
```

Esa función callback del "then" no devuelve una promesa. De hecho no devuelve ningún valor. Esa situación no es un problema en realidad, simplemente el próximo "then" se ejecutará inmediatamente después del anterior, sin tener que esperar nada, puesto que no se ha devuelto una promesa que tarde en resolverse.



## Ejecutar promesas en paralelo

No siempre se necesitan ejecutar todas las promesas una detrás de otra. Muchas veces no son dependientes entre sí y podemos ejecutarlas a la vez, para ahorrar tiempo.

Podríamos simplemente colocar el código de todas las promesas suelto e independiente entre sí. Algo como:

```
esperar(5000).then((res) => console.log("Uno"));
esperar(1000).then((res) => console.log("Dos"));
esperar(3000).then((res) => console.log("Tres"));
console.log("Fin!");
```

¿En qué orden piensas que se mostrarán los distintos mensajes a la consola?

Seguramente lo has adivinado.... será: "Fin! Dos Tres Uno".

Pero a veces tienes que esperar a que se ejecuten todas las promesas para luego hacer alguna acción con todos los valores de respuesta. Por ejemplo, dado el código anterior nos gustaría que el mensaje "Fin!" fuera el último que se mostrase.

Quizás en este ejemplo está claro porque sabemos que la promesa que más tiempo va a tardar en ejecutarse es la primera, que espera 5 segundos. Podríamos escribir el mensaje "Fin!" cuando ella termine. Sin embargo, en la mayoría de las ocasiones no sabemos qué promesa va a terminar más tarde y queremos asegurarnos que todas hayan acabado antes de hacer nada. Para estos casos tenemos "Promise.all".

La clase Promise de Javascript tiene un método llamado all() que recibe un array de promesas. Las ejecuta todas y, cuando acaba la última, se devuelve un array con todos los valores devueltos por las promesas.

```
Promise.all([
  esperar(5000),
  esperar(1000),
  esperar(3000)
]).then( respuestas => {
  console.log("Fin!");
});
```

Ahora el mensaje "Fin" se mostrará por último, una vez acaben todas las promesas. Por tanto, y dado que se ejecutan en paralelo, tardará 5 segundos en total en aparecer "Fin".

Si quisieramos trabajar con los valores de devolución de las promesas, podemos obtenerlos con el array de respuestas que nos devuelve Promise.all(). Por ejemplo así mostraríamos todas las respuestas haciendo un recorrido al array.

```
Promise.all([
  esperar(5000),
  esperar(1000),
  esperar(3000)
]).then( respuestas => {
```

```
for (let i in respuestas) {  
    console.log(respuestas[i]);  
}  
};
```

## Conclusión

Las promesas son ideales para la programación asíncrona, porque mantienen el orden en el código y la sencillez, con lo que también tus programas serán más claros y fáciles de extender o depurar. En este artículo has podido aprender a trabajar con promesas, además de crear tus propias implementaciones de funciones que devuelven promesas. Con todo ello serás capaz de sacarles un gran partido a estas herramientas para la mejora del código.

Sin embargo, para manejar promesas de una manera más avanzada hemos podido aprender a gestionar varias a la vez, tanto en secuencia (útil cuando la ejecución de una promesa depende del resultado producido por la anterior) como en paralelo (de manera que se realicen todas a una y ahorremos tiempo), algo que seguramente te será de utilidad más de una vez en tus aplicaciones Javascript.

Este artículo es obra de *Miguel Angel Alvarez*

Fue publicado / actualizado en *01/12/2023*

Disponible online en <https://desarrolloweb.com/articulos/introduccion-promesas-es6.html>

## async await en Javascript

**Explicaciones y ejemplos de las sentencias async await en Javascript. Qué es async y await, para qué sirven, qué utilidad tienen y cómo se usan para mejorar el código asíncrono.**



Una de las características más importantes de JavaScript es su **comportamiento asíncrono**. Actualmente la mayor parte de los lenguajes tienen características asíncronas, porque se van



copiando los unos a los otros, sin embargo JavaScript nació con la asincronía como una de sus características iniciales.

Decimos generalmente que JavaScript es un lenguaje fácil, pero justamente por su carácter asíncrono a veces resulta un poco difícil llegar a dominarlo. En otros artículos anteriores de desarrollo web hemos hablado bastante sobre la [asincronía](#) y explicado con detalle de qué se trata y cómo se debe de gestionar. En este artículo vamos a tratar un tema un poquito más avanzado que es el **async await de JavaScript**.

## Qué es async await

En JavaScript existen varios modos de tratarla a sincronía. **El modo de más alto nivel es async / await**, que nos permite un **código más claro** y más **parecido al código síncrono**.

Se trata básicamente de una manera especial de tratar las promesas en Javascript, evitando tener que crearlas manualmente y escribir los bloques `then` y `catch`. Por extensión, nos ahorra trabajar con funciones anónimas. Lo más notable de `async await` es que permiten escribir un código asíncrono que se lee más como si fuera un código síncrono tradicional.

Los bloques `async` y `await` son independientes pero los usamos de manera conjunta. Básicamente sirven para lo siguiente:

- **Con `async` podemos declarar una función asíncrona** que básicamente tendrá la **habilidad de poder tratar comportamientos asíncronos usando await**.
- **Con `await`** por su parte conseguimos llamar a una función que devuelve una promesa con una facilidad extra. La ventaja o facilidad de `await` es que el **propio JavaScript esperará a la resolución de la promesa**, sin ejecutar las siguientes líneas de código del programa, y una vez tenga la respuesta la devolverá y continuará con el flujo de ejecución de las sentencias.

Por supuesto todo esto se ve bien con ejemplos que vamos a abordar a continuación.

## Ejemplos sencillos con async await

Vamos a comenzar con ejemplos sencillos que nos permitan ver cómo funcionan las sentencias `async / await`.

### Creamos una función con `async`

Comenzamos con **`async`**. Esta palabra clave se utiliza antes de declarar una función asíncrona, que devuelve una promesa. En lugar de devolver los valores directamente, una función `async` devuelve una promesa que se resolverá con el valor devuelto por la función.

```
async function devuelvoPromesa() {
    return "Resultado devuelto al resolverse";
}
```



En el código anterior la función `devuelvoPromesa()` devuelve una promesa, aunque si la leemos aparentemente está devolviendo un string. Sin embargo es una promesa que se resuelve inmediatamente y entrega el valor: "Resultado devuelto al resolverse".

Podemos comprobar que verdaderamente esta función devuelve una promesa si escribimos un código como este:

```
console.log(typeof devuelvoPromesa());
```

Al ejecutarse el código anterior nos mostrará en la consola el tipo del valor devuelto por la función que es un "object".

Incluso podríamos hacer uso de `instanceof` para averiguar si lo que te devuelve la función es una instancia de la clase `Promise`:

```
console.log(devuelvoPromesa() instanceof Promise);
```

En la consola aparecerá `true` al ejecutarse el código anterior.

En cualquier caso, tal como hemos escrito este código no hemos sacado mucho partido a la función declarada con `async`, ya que la gracia de hacerlo es ejecutar el `await` para resolver promesas con un estilo de código similar al código síncrono secuencial.

### Hacemos uso de `await`

Ahora vamos a usar la palabra clave `await` dentro de funciones declaradas como `async`. La utilidad de esta construcción será para esperar que se resuelva una promesa. Hace que el código asíncrono se parezca y se comporte más como el código común, síncrono, lo que facilita la lectura y comprensión de los programas.

```
async function devuelvoPromesa() {
  let valor = await otraFuncionQueDevuelvePromesa();
  console.log(valor); // Este código espera hasta que la promesa se resuelva
  return valor;
}
```

En el código anterior tenemos ahora un uso más útil de `async`, ya que hemos colocado un `await` dentro. Básicamente lo que hace el `await` es pausar la ejecución de `devuelvoPromesa()` hasta que `otraFuncionQueDevuelvePromesa()` se resuelva. Esto significa que el código después de `await` se ejecutará como si fuera código síncrono, es decir, la línea siguiente se ejecutará después que la promesa que devuelve `otraFuncionQueDevuelvePromesa()` se haya resuelto.

Te mostramos el código de ambas funciones a la vez por si quieras copiar y pegar y así probarlo en tu entorno.

```
async function devuelvoPromesa() {
  let valor = await otraFuncionQueDevuelvePromesa();
```



```
        console.log(valor); // Este código espera hasta que la promesa se resuelva
        return valor;
    }

    function otraFuncionQueDevuelvePromesa() {
        return new Promise( (resolve, reject) =>
            setTimeout(() => resolve('valor de respuesta'), 1000)
        );
    }
}
```

## Dónde le vamos a sacar partido a `async await`

Estas utilidades de Javascript son meramente lo que llamamos un "azúcar sintáctico", ya que en el fondo todo funciona igual que con las promesas ya conocidas anteriormente.

Puedes ver este artículo siquieres repasar sobre las [promesas en Javascript](#).

Donde resulta especialmente útil el `async await` es para la realización de operaciones como solicitudes de red o consultas a bases de datos, donde se requiere esperar a que se complete una operación antes de continuar con la ejecución del código siguiente.

Su principal ventaja es reducir la complejidad del código y hacerlo más legible, comparado con el encadenamiento de promesas tradicionales que resolvemos con los bloques `then` y `catch`.

Vamos a suponer que queremos hacer una función que recibe las tareas que hay que realizar entregadas por un API REST ("ToDos"). Sin `async await` podríamos llegar a un código como este:

```
function recibeTodosSinAsyncAwait() {
    return new Promise( resolve => {
        fetch('https://jsonplaceholder.typicode.com/todos')
            .then(response => response.json())
            .then(json => resolve(json))
    })
}
```

Como se trata de una funcionalidad asíncrona lo máximo que podemos hacer es devolver una promesa que se resolverá cuando se haya recibido respuesta del API REST. En este estilo de codificación somos nosotros mismos los que tenemos que crear un nuevo objeto de la instancia `Promise` y resolverla cuando tengamos el dato.

Ahora vamos a ver el código que tendríamos si usamos a `async await`. Como verás se simplifica bastante.

```
async function recibeTodos() {
    const response = await fetch('https://jsonplaceholder.typicode.com/todos');
    return await response.json();
}
```



En este ejemplo en particular hemos utilizado dos veces await, ya que el método fetch() requiere que resolvamos dos promesas hasta obtener el valor del JSON que nos devuelve el API.

Aquí tienes un ejemplo de uso de la función recibirTodos():

```
recibirTodos().then(todos => console.log(todos));
```

### Cómo tratar casos de promesas rechazadas con async await

La parte más complicada de esta estructura de funcionamiento con `async await` consiste en la resolución de promesas rechazadas, ya que nos obliga a incorporar un bloque `try catch`.

Ahora vamos a ver un código de un acceso a un API REST en el que también vamos a validar que los datos se reciban correctamente, una comprobación importante a la hora de hacer programas robustos.

```
async function obtenerDatosDeApi() {
  try {
    // Realizar la solicitud HTTP con fetch y esperar la respuesta
    let respuesta = await fetch('https://jsonplaceholder.typicode.com/todos');

    // Verificar si la respuesta es exitosa
    if (respuesta.ok) {
      // Parsear la respuesta como JSON y esperar el resultado
      let datos = await respuesta.json();
      return datos;
    } else {
      // Manejar errores de respuesta HTTP (ej. 404, 500)
      throw new Error('Error en la solicitud: ' + respuesta.status);
    }
  } catch (error) {
    // Manejar errores de red
    console.error('Hubo un problema con la operación fetch: ' + error.message);
  }
}
```

En este ejemplo, la función `obtenerDatosDeApi()` es una función asíncrona que usa `fetch` para realizar una solicitud HTTP. Es un código similar en el fondo al de antes, solo que realiza comprobaciones de posibles respuestas incorrectas, tanto a nivel de red como a nivel de API.

En la función se utiliza `await` para esperar a que se complete la solicitud y para procesar la respuesta. La respuesta se verifica para asegurarse de que la solicitud fue exitosa (`respuesta.ok`) y luego se convierte el cuerpo de la respuesta en JSON.

Todo se tiene que incorporar dentro de un bloque `try...catch`, pues es la manera que tenemos en Javascript para controlar las excepciones.



Esta es realmente la parte compleja de usar await, que nos obliga a usar excepciones. No obstante, nos ahorra generar las promesas a mano, que ya está bastante bien.

Si ocurre un error durante la solicitud o el procesamiento de la respuesta, se levanta un error, que al final se captura y se trata con el `catch` del bloque `try`.

## Conclusión

Trabajar con la asincronía en el código JavaScript siempre a resultado un poco laborioso. La parte más compleja es organizar el código de manera que sea muy claro y por lo tanto se pueda mantener con facilidad. Es en este sentido donde las mejoras incorporadas en el lenguaje, como el tratamiento de promesas y los bloques `async await`, han venido a ayudar de manera muy notable.

Con este artículo terminamos la serie de entregas dedicada a hablar de asincronía en Javascript. Esperamos que te haya resultado interesante y que hayas aprendido lo suficiente para abordar los problemas complejos que se plantean en las aplicaciones reales. Ya sabes que si tienes cualquier duda nos la puedes formular en la sección de [faqs](#).

Este artículo es obra de *Miguel Angel Alvarez*

Fue publicado / actualizado en *01/12/2023*

Disponible online en <https://desarrolloweb.com/articulos/async-await-javascript>

# Epílogos a esta primera parte del Manual de Javascript

Con estos artículos terminaremos la primera parte del manual de Javascript de DesarrolloWeb.com. Aquí veremos por dónde continuar estas explicaciones y algunos temas de interés como el control de errores en los programas.

## Pausa y consejos Javascript

### Hacemos una pausa en el manual de Javascript para ofrecer una serie de consejos útiles.

Hasta aquí hemos visto la mayor parte de la sintaxis y forma de funcionar de el lenguaje Javascript. Ahora podemos escribir scripts simples que hagan uso de variables, funciones, arrays, estructuras de control y toda clase de operadores. Con todo esto conocemos el lenguaje, pero aun queda mucho camino por delante para dominar Javascript y saber hacer todos los efectos posibles en páginas web, que en definitiva es lo que interesa.

De todos modos, este manual nos lo hemos tomado con mucha calma, con intención de que las personas que no estén familiarizadas con el mundo de la programación puedan también tener acceso al lenguaje y aprendan las técnicas básicas que permitirán [afrontar futuros retos](#) en la programación. Esperamos entonces que la lectura del manual haya sido provechosa para los más inexpertos y que ahora puedan entender con facilidad las siguientes lecciones o ejemplos, ya que conocen las bases sobre las que están implementados.

Antes de acabar, vamos a dar una serie de consejos a seguir a la hora de programar nuestros scripts, que nos pueden ayudar a hacernos la vida más fácil. Algunos consejos son nuevos e importantes, otros se han señalado con anterioridad, pero sin duda viene bien recordar.

### Distintos navegadores

Lo primero importante en señalar es que Javascript es un lenguaje muy dinámico y que ha sido implementado poco a poco, a medida que salían nuevos navegadores. Si pensamos en el Netscape 2, el primer navegador que incluía Javascript, y el Netscape 6 o Internet Explorer 6 nos daremos cuenta que han pasado un mundo de novedades entre ellos. Javascript ha cambiado por lo menos tanto como los navegadores y eso representa un problema para los programadores, porque tienen que estar al tanto de las distintas versiones y la manera de funcionar que tienen.



**Actualizado:** a día de hoy todavía las diferencias entre los navegadores antiguos y los nuevos es todavía más patente. Incluso, ahora que aparece el HTML 5, existen muchos otros usos donde Javascript tiene validez.

Las bases de javascript, sobre las que hemos hablado hasta ahora, no han cambiado prácticamente nada. Sólo en algunas ocasiones, que hemos señalado según las conocíamos -como los arrays por ejemplo-, el lenguaje ha evolucionado un poco. Sin embargo, a medida que nuevas tecnologías como el DHTML se desarrollaban, los navegadores han variado su manera de manejarlas.

Nuestro consejo es que esteis al tanto de las cosas que funcionan en unos u otros sistemas y que programéis para que vuestras páginas sean compatibles en el mayor número de navegadores. Para procurar esto último es muy aconsejable probar las páginas en varias plataformas distintas.

También es muy útil dejar de lado los últimos avances, es decir, no ir a la última, sino ser un poco conservadores, para que las personas que han actualizado menos el browser puedan también visualizar los contenidos.

### Consejos para hacer un código sencillo y fácil de mantener

Ahora vamos a dar una serie de consejos para que el código de nuestros scripts sea más claro y libre de errores. Muchos de ellos ya los hemos señalado y somos libres de aplicarlos o no, pero si lo hacemos nuestra tarea como programadores puede ser mucho más agradable, no sólo hoy, sino también el día que tengamos que revisar los scripts en su mantenimiento.

- Utiliza comentarios habitualmente para que lo que estás haciendo en los scripts pueda ser recordado por ti y cualquier persona que tenga que leerlos más adelante.
- Ten cuidado con el ámbito de las variables, recuerda que existen variables globales y locales, que incluso pueden tener los mismos nombres y conoce en cada momento la variable que tiene validez.
- Escribe un código con suficiente claridad, que se consigue con saltos de línea después de cada instrucción, un sangrado adecuado (poner márgenes a cada línea para indicar en qué bloque están incluidas), utilizar las llaves {} siempre, aunque no sean obligatorias y en general mantener siempre el mismo estilo a la hora de programar.
- Aplica un poco de consistencia al manejo de variables. Declara las variables con var. No cambies el tipo del dato que contiene (si era numérica no pongas luego texto, por ejemplo). Dales nombres comprensibles para saber en todo momento qué contienen. Incluso a la hora de darles nombre puedes aplicar una norma, que se trata de que indiquen en sus nombres el tipo de dato que contienen. Por ejemplo, las variables de texto pueden empezar por una s (de String), las variables numéricas pueden empezar por una n o las booleanas por una b.
- Prueba tus scripts cada poco a medida que los vas programando. Puedes escribir un trozo de código y probarlo antes de continuar para ver que todo funciona correctamente. Es más fácil encontrar los errores de código en bloques pequeños que en bloques grandes.



En el resto de esta primera parte del [manual de Javascript](#) vamos a ver un par de cosas también fundamentales en el trabajo del día a día con este lenguaje, como son la [detección de errores](#). Además, os dejamos unas referencias importantes para continuar con el aprendizaje:

- [Manual de Javascript II](#): la segunda parte de este manual abordará usos de Javascript para control de los elementos de las páginas web.
- [Videotutorial de Javascript](#): estamos publicando una serie de videotutoriales de Javascript que te encantarán si quieras aprender por la práctica.
- [Javascript a fondo](#): todo lo que tiene que ver con Javascript lo encontrarás en esta sección de DesarrolloWeb.com.

Este artículo es obra de *Miguel Angel Alvarez*

Fue publicado / actualizado en *10/01/2002*

Disponible online en <https://desarrolloweb.com/articulos/642.php>

## Tratamiento de errores en javascript

**Vamos a explicar los errores comunes que podemos cometer y cómo evitarlos y depurarlos. Además veremos una pequeña conclusión a la primera parte del manual.**

Hemos terminado ya de explicar todo el contenido de la primera parte del [manual de javascript](#), pero aun tenemos algunas cosas importantes que tratar. En concreto en este artículo vamos a explicar los errores comunes que podemos cometer y cómo evitarlos y depurarlos. Además veremos una pequeña conclusión a esta parte del manual.

### Errores comunes en Javascript

Cuando ejecutamos los scripts pueden ocurrir dos tipos de errores de sintaxis o de ejecución, los vemos a continuación.

Errores de sintaxis ocurren por escribir de manera errónea las líneas de código, equivocarse a la hora de escribir el nombre de una estructura, utilizar incorrectamente las llaves o los paréntesis o cualquier cosa similar. Estos errores los indica javascript a medida que está cargando los scripts en memoria, lo que hace siempre antes de ejecutarlos, como se indicó en los primeros capítulos.

Cuando el analizador sintáctico de javascript detecta un error de estos se presenta el mensaje de error.

Errores de ejecución ocurren cuando se están ejecutando los scripts. Por ejemplo pueden ocurrir cuando llamamos a una función que no ha sido definida. javascript no indica estos errores hasta que no se realiza la llamada a la función.



La manera que tiene javascript de mostrar un error puede variar de un navegador a otro. En versiones antiguas se mostraba una ventanita con el error y un botón de aceptar, tanto en Internet Explorer como en Netscape. En la actualidad los errores de javascript permanecen un poco más ocultos al usuario. Esto viene bien, porque si nuestras páginas tienen algún error en alguna plataforma no será muy molesto para el usuario que en muchas ocasiones ni se dará cuenta. Sin embargo para el programador puede ser un poco más molesto de revisar y se necesitará conocer la manera de que se muestren los errores para que puedan ser reparados.

En versiones de Internet Explorer mayores que la 4 se muestra el error en la barra de estado del navegador y se puede ver una descripción más grande del error si le damos un doble click al icono de alerta amarillo que aparece en la barra de estado. En Netscape aparece también un mensaje en la barra de estado que además nos indica que para mostrar más información debemos teclear "javascript:" en la barra de direcciones (donde escribimos las URL para acceder a las páginas). Con eso conseguimos que aparezca la Consola javascript, que nos muestra todos los errores que se encuentran en las páginas.

También podemos cometer fallos en la programación que hagan que los scripts no funcionen tal y como deseábamos y que javascript no detecta como errores y por lo tanto no muestra ningún mensaje de error.

Por dejarlo claro vamos a ver un ejemplo en el que nuestro programa puede no funcionar como deseamos sin que javascript ofrezca ningún mensaje de error. En este ejemplo trataríamos de sumar dos cifras pero si una de las variables es de tipo texto podríamos encontrarnos con un error.

```
var numero1 = 23
var numero2 = "42"
var suma = numero1 + numero2
```

¿Cuánto vale la variable suma? Como muchos ya sabéis, la variable suma vale "2342" porque al intentar sumar una variable numérica y otra textual, se tratan a las dos como si fueran de tipo texto y por lo tanto, el operador + se aplica como una concatenación de cadenas de caracteres. Si no estamos al tanto de esta cuestión podríamos tener un error en nuestro script ya que el resultado no es el esperado y además el tipo de la variable suma no es numérico sino cadena de caracteres.

## Evitar errores comunes

Vamos a ver ahora una lista de los errores típicos cometidos por inexpertos en la programación en general y en javascript en particular, y por no tan inexpertos.

Utilizar = en expresiones condicionales en lugar de == es un error difícil de detectar cuando se comete, si utilizamos un solo igual lo que estamos haciendo es una asignación y no una comparación para ver si dos valores son iguales.

No conocerse la precedencia de operadores y no utilizar paréntesis para agrupar las operaciones que se desea realizar. En este caso nuestras operaciones pueden dar resultados no deseados.



Usar comillas dobles y simples erróneamente. Recuerda que se pueden utilizar comillas dobles o simples indistintamente, con la norma siguiente: dentro de comillas dobles se deben utilizar comillas simples y viceversa.

Olivarse de cerrar una llave o cerrar una llave de más.

Colocar varias sentencias en la misma línea sin separarlas de punto y coma. Esto suele ocurrir en los manejadores de eventos, como onclick, que veremos más adelante.

Utilizar una variable antes de inicializarla o no declarar las variables con var antes de utilizarlas también son errores comunes. Recuerda que si no la declaras puedes estar haciendo referencia a una variable global en lugar de una local.

Contar las posiciones de los arrays a partir de 1. Recuerda que los arrays empiezan por la posición 0.

## Depurar errores javascript

Cualquier programa es susceptible de contener errores. javascript nos informará de muchos de los errores de la página: los que tienen relación con la sintaxis y los que tienen lugar en el momento de la ejecución de los scripts a causa de equivocarnos al escribir el nombre de una función o una variable. Sin embargo, no son los únicos errores que nos podemos encontrar, también están los errores que ocurren sin que javascript muestre el correspondiente mensaje de error, como vimos anteriormente, pero que hacen que los programas no funcionen como esperábamos.

**Nota:** Para aprender a utilizar las herramientas de detección de errores Javascript más populares, recomendamos especialmente ver el [videotutorial sobre detectar errores Javascript en páginas web](#).

Para todo tipo de errores, unos más fáciles de detectar que otros, debemos utilizar alguna técnica de depuración que nos ayude a encontrarlos. Lenguajes de programación más potentes que el que tratamos ahora incluyen importantes herramientas de depuración, pero en javascript debemos contentarnos con una rudimentaria técnica. Se trata de utilizar una función predefinida, la función alert() que recibe entre paréntesis un texto y lo muestra en una pequeña ventana que tiene un botón de aceptar.

Con la función alert() podemos mostrar en cualquier momento el contenido de las variables que estamos utilizando en nuestros scripts. Para ello ponemos entre paréntesis la variable que deseamos ver su contenido. Cuando se muestra el contenido de la variable el navegador espera a que apretemos el botón de aceptar y cuando lo hacemos continúa con las siguientes instrucciones del script.

Este es un sencillo ejemplo sobre cómo se puede utilizar la función alert() para mostrar el contenido de las variables.



```
var n_actual = 0
var suma = 0
while (suma<300){
    n_actual ++
    suma += suma + n_actual
    alert("n_actual vale " + n_actual + " y suma vale " + suma)
}
```

Con la función alert() se muestra el contenido de las dos variables que utilizamos en el script. Algo similar a esto es lo que tendremos que hacer para mostrar el contenido de las variables y saber cómo están funcionando nuestros scripts, si todo va bien o hay algún error.

## Conclusión

Hasta aquí hemos conocido la sintaxis javascript en profundidad. Aunque aun nos quedan cosas importantes de sintaxis, la visión que has podido tener del lenguaje será suficiente para enfrentarte a los problemas más fundamentales. En adelante presentaremos otros reportajes para aprender a utilizar los recursos con los que contamos a la hora de hacer efectos en páginas web.

Veremos la jerarquía de objetos del navegador, cómo construir nuestros propios objetos, las funciones predefinidas de javascript, características del HTML Dinámico, trabajo con formularios y otras cosas importantes para dominar todas las posibilidades del lenguaje.

Todo ello en nuestro [manual de Javascript II](#) y en el [Taller de Javascript](#).

Este artículo es obra de *Miguel Angel Alvarez*

Fue publicado / actualizado en *11/02/2002*

Disponible online en <https://desarrolloweb.com/articulos/668.php>

## Consejos para escribir código Javascript

**En este artículo puedes encontrar varios consejos bastante interesantes a la hora de programar código Javascript.**

Si estas dando tus primeros pasos en Javascript y estas empezando ya a ser sucio y desordenado... No tienes excusa da un giro para escribir el código ordenado y todo será más sencillo.

Los foros estan llenos de peticiones de información sobre Ajax, DOM y como se usan algunas librerías o efectos. Hay una extraordinaria cantidad de información, scripts, librerías que estan siendo desarrollados, blogs y nuevos sitios especializados en esta temática, sólo necesitas un poco de tiempo y echarle un vistazo... es muy fácil los mejores los encuentras en [Digg.com](#) o en [del.icio.us](#), se acabaron aquellos días en el que Javascript y el DHTML se convirtieron en persona non grata como habilidad principal en tu CV.



La gran mayoría de código Javascript es hoy en dia mucho más limpio que en la "era" DHTML.

Ahora es un buen momento para convertirte en un entusiasta de Javascript. Aunque algunos defectos que ocurrieron tiempo atras se repiten sin embargo.

Aquí os dejo una series de consejos que os hará más sencillo mantener tu código Javascript ordenado, algunos consejos son demasiado obvios pero todos sabemos que el hombre es el único animal que...

### Conserva la sintaxis y estructura de tu código limpia y ordenada

Esto significa que guardes por ejemplo un límite de longitud de línea (80 caracteres) y que programes funciones razonablemente pequeñas. Un fallo es pensar que en una función larga lo podemos hacer todo.

Tener un tamaño razonable para tus funciones significa que las podrás reutilizar cuando amplies el código, tampoco seas extremista y hagas funciones de una o dos líneas esto puede llegar a ser más confuso que usar una única función.

Este es un ejemplo que muestra cual es la justa medida en cuanto al tamaño de las funciones y la división de las tareas:

```
function toolLinks(){
    var tools = document.createElement('ul');
    var item = document.createElement('li');
    var itemlink = document.createElement('a');
    itemlink.setAttribute('href', '#');
    itemlink.appendChild(document.createTextNode('close'));
    itemlink.onclick = function(){window.close();}
    item.appendChild(itemlink);
    tools.appendChild(item);
    var item2 = document.createElement('li');
    var itemlink2 = document.createElement('a');
    itemlink2.setAttribute('href', '#');
    itemlink2.appendChild(document.createTextNode('print'));
    itemlink2.onclick = function(){window.print();}
    item2.appendChild(itemlink2);
    tools.appendChild(item2);
    document.body.appendChild(tools);
}
```

Puedes optimizar esta función separando cada tarea con su propia función:

```
function toolLinks(){
    var tools = document.createElement('ul');
    var item = document.createElement('li');
    var itemlink = createLink('#', 'close', closeWindow);
    item.appendChild(itemlink);
    tools.appendChild(item);
    var item2 = document.createElement('li');
    var itemlink2 = createLink('#', 'print', printWindow);
    item2.appendChild(itemlink2);
    tools.appendChild(item2);
    document.body.appendChild(tools);
}

function printWindow(){}
```



```
window.print();
}

function closeWindow() {
window.close();
}

function createLink(url,text,func){
var temp = document.createElement('a');
temp.setAttribute('href', url);
temp.appendChild(document.createTextNode(text));
temp.onclick = func;
return temp;
}
```

## Utiliza inteligentemente los nombres de variables y funciones

Esta es un técnica esencial de programación, utiliza nombres de variables y funciones que sean totalmente descriptivos e incluso otra persona pueda llegar a plantearse que función realizan antes de ver el código.

Recuerda que es correcto el uso de guiones o mayúsculas para concatenar diferentes palabras, en este caso concreto de es más típico el uso de mayúsculas debido a la sintaxis del lenguaje, (ej. `getElementsByTagName()`).

```
CambioFormatoFecha();
cambio_formato_fecha();
```

## Comenta el código

Gracias a los comentarios puedes librarte de más de un quebradero de cabeza, es mejor resolver el problema una única vez.

Cómo puedes comprobar en cualquier libro de programación cada línea tiene comentarios explicando su objetivo.

## Diferencia las variables dependiendo de su importancia

Este paso es simple: Coloca aquellas variables que son usadas durante todo el script en la cabecera del código, de esta maneras siempre sabrás donde encontrar estas variables que son las que determinan el resultado de nuestro código.

```
function toolLinks(){
var tools, closeWinItem, closeWinLink, printWinItem, printWinLink;

// variables temporales
var printLinkLabel = ?print?;
var closeLinkLabel = ?close?;#

tools = document.createElement(?ul?);
closeWinItem = document.createElement(?li?);
closeWinLink = createLink(?#, closeLinkLabel, closeWindow);
closeWinItem.appendChild(closeWinLink);
tools.appendChild(closeWinItem);
printWinItem = document.createElement(?li?);
```



```
printWinLink = createLink('#', printLinkLabel, printWindow);
printWinItem.appendChild(printWinLink);
tools.appendChild(printWinItem);
document.body.appendChild(tools);
}
```

## Separa el texto del código

Puedes separar el texto del código, utilizando un documento llamado texto.js en formato JSON.

Un ejemplo que funciona muy bien podría ser:

```
var locales = {
'en': {
'homePageAnswerLink':'Answer a question now',
'homePageQuestionLink':'Ask a question now',
'contactHoverMessage':'Click to send this info as a message',
'loadingMessage' : 'Loading your data...',
'noQAMessage' : 'You have no Questions or Answers yet',
'questionsDefault': 'You have not asked any questions yet',
'answersDefault': 'You have not answered any questions yet.',
'questionHeading' : 'My Questions',
'answersHeading' : 'My Answers',
'seeAllAnswers' : 'See all your Answers',
'seeAllQuestions' : 'See all your Questions',
'refresh': 'refresh'
},
'es': {
'homePageAnswerLink':'Responde una pregunta',
'homePageQuestionLink':'Haz una pregunta',
'contactHove' : 'Cargando datos...',
'noQAMessage' : 'No quedan preguntas',
'questionsDefault': 'Quedan preguntas por responder',
'answersDefault': 'No quedan preguntas pendientes',
'questionHeading' : 'Mis preguntas',
'answersHeading' : 'Mis respuestas',
'seeAllAnswers' : 'Ver todas las respuestas',
'seeAllQuestions' : 'Ver todas las preguntas',
'refresh': 'Refrescar'
},
'fr': {
}
'de': {
}
};
```

Esto permitiría a cualquiera que no es programador traducir el texto del script, cambiando únicamente las etiquetas sin necesidad de acceder al código.

## Documenta el código

Escribe una buena documentación de tu script / librería o efecto. Una buena documentación da calidad al código, sino pregúntate porque existe la clásica documentación en cualquier API con todas las posibles propiedades y parámetros, pero sin duda lo mejor de todo es explicar con ejemplos que contienen una lista de posibilidades.



Este artículo es obra de *Manu Gutierrez*

Fue publicado / actualizado en *20/12/2007*

Disponible online en <https://desarrolloweb.com/articulos/consejos-para-escribir-codigo-javascript.html>

## Coerción de tipos en Javascript

**Qué es la coerción de tipos en Javascript y por qué es importante que la conozcas si estás trabajando con este lenguaje de programación.**



Javascript es un lenguaje de programación sencillo para comenzar. Al poco tiempo de empezar a programar verás que haces cosas interesantes en el contexto de la web y le tomas el gusto rápidamente a la programación, sobre todo si no tenías antes conocimientos previos.

Paralelamente, gracias a que Javascript permite hacer desarrollos en muchos entornos, no solamente en el navegador, sientes que estás adquiriendo habilidades que realmente merecen la pena.

Sin embargo, no nos debemos llevar a engaño y debemos saber que el camino para dominar Javascript no es tan sencillo como podría parecer. En este artículo vamos a analizar **uno de los puntos básicos del lenguaje** que a menudo uno deja escapar durante su formación pero que es **relevante para adquirir dotes profesionales**. En otras palabras, puedes vivir durante muchos años sin saber lo que es la **coerción de tipos y cómo funciona en Javascript**, pero es importante que la conozcas para no andar perdido y frustrado cuando las cosas no funcionan a la primera.

### Tipos en Javascript

Primero queremos señalar que a veces se le califica a Javascript como un "*lenguaje no tipado*". Lejos de la realidad. En realidad claro que **existen tipos** en el lenguaje, como ya hemos explicado en el artículo dedicado a los [tipos en Javascript](#). Lo que tenemos en realidad es un **lenguaje de tipado dinámico**, donde una variable puede tener diversos tipos.



En los lenguajes de tipado estático como Java cada operador se debe usar sobre variables de un determinado tipo. Sin embargo en lenguajes de tipado dinámico es posible usar **operadores con variables de distintos tipos**. Por ejemplo, podemos sumar una cadena a un número sin problemas, algo que en Java daría un error de compilación.

## Coerción de tipos

Al relacionar operandos de distintos tipos para realizar diversas operaciones, el intérprete de Javascript, tiene que hacer una **transformación implícita de los valores de un tipo a otro**. Esa transformación se llama **coerción**.

Por ejemplo, al usar el operador `+` con dos valores de tipos distintos, como una cadena de texto y un valor numérico, lo que ocurre por debajo es que el valor numérico se convierte en una cadena y se realiza la concatenación.

```
let x = 5 + '5'; // la variable x valdrá 55
```

La **coerción de tipos es algo que ocurre constantemente en las aplicaciones Javascript**. El intérprete de Javascript la mayoría de las veces funciona de manera bastante lógica, pero en ocasiones podemos tener sorpresas!

## Comportamientos extraños de Javascript

Muchas veces podremos suponer cómo se va a comportar el lenguaje cuando se realiza la coerción de tipos para poder realizar operaciones en las que participan distintos tipos de datos, pero algunas veces aplicar simplemente el sentido común no será suficiente! De hecho, Javascript es tan laxo en su sistema de tipos que podemos hacer cosas tan extrañas como esta:

```
[] + []
```

En la anterior sentencia no hay sentido común que sirva para imaginarse qué es lo que va a pasar. No hay otra solución que ejecutarlo para ver ¿no?. No hace falta que lo ejecutes, ya te informo yo que evaluar esa sentencia te devolverá un string vacío.

De todos modos, sin irnos a cosas tan raras, vamos a ver ahora algunos casos extraños en Javascript que ocurren cuando se realiza la coerción implícita de tipos al usar operadores distintos. Veremos que la mayoría de las sorpresas ocurrirán porque hacemos cosas que quizás no tengan mucho sentido, pero a veces sin darnos cuenta este tipo de operaciones son bastante frecuentes cuando descuidamos el tipo que tienen las variables que estamos usando.

Veamos un ejemplo bastante extraño.

```
console.log(0 == "0"); //muestra true en la consola  
console.log(0 == []); // muestra true en la consola
```



Pero ahora a simple vista, y por deducción lógica, qué piensas que mostrará esta sentencia en la consola:

```
console.log("0" == []);
```

La lógica nos llevaría a pensar que esta comparación daría `true`, pero sin embargo Javascript lo evalúa como `false`. Estas son las cosas que llaman la atención y ante las cuales es mejor ser precavidos, evitando hacer este tipo de comparaciones extrañas.

Veamos otro caso difícil de entender.

```
console.log('0' == false); // muestra true en la consola
```

Al ejecutar la sentencia anterior veremos que el resultado de comparar la cadena '0' con `false` nos devuelve `true`. Sin embargo, dado el código siguiente:

```
if('0') {  
    console.log("'0' evalúa como true");  
} else {  
    console.log("'0' evalúa como false");  
}
```

¿Qué salida piensas que veremos? Pues bien, la cadena '0' evaluada tal cual, nos llevaría por el caso positivo del if. Quizás no era lo que esperabas!

## Coerción explícita de los datos

Si queremos realizar determinadas operaciones, asegurándonos que **los tipos de datos sean los que nosotros queremos**, podemos realizar una **coerción explícita**. En muchos casos es más conveniente que de dejar al intérprete que haga la coerción de tipos que tenga definida de manera predeterminada.

Para ello en Javascript podemos usar unas clases incorporadas que nos pueden ayudar a convertir los datos en los tipos deseados. Las principales serían:

- [Clase Number](#)
- [Clase Boolean](#)
- [Clase String](#)

Cuando usamos el constructor de estas clases obtenemos un valor del tipo particular de cada clase. Veamos unos ejemplos de uso de los constructores de estas clases.

```
let x = Number('999') // devuelve el valor 999 de tipo number  
let y = String(true) // devuelve la cadena "true"  
let z = Boolean(0); // devuelve el boleano false
```



Por ejemplo, en el caso de la suma que hemos visto antes de una cadena con un número, si deseamos que se produzca verdaderamente una suma y no una concatenación, podemos hacer lo siguiente.

```
let sum1 = 5;
let sum2 = '5';
sum2 = Number(sum2);
let suma = sum1 + sum2; // obtenemos el valor 10 en vez del valor 55 como el ejemplo primero del artículo
```

En este caso, también podemos usar el operador unario + que convierte de manera implícita el valor al tipo number.

```
let val1 = +'66' // asigna el valor numérico 66
let val2 = +'hola' // asigna el valor NaN (Not a Number)
let val3 = +true // asigna el valor numérico 1
let val4 = +false // asigna el valor numérico 0
```

Al final se trata de usar las herramientas que te ofrece el lenguaje para asegurarnos que la variable guarda el tipo que deseamos. En este caso particular tienes esta alternativa y quedaría a tu juicio saber qué te ofrece un código más claro, usar el operador unario o bien la clase Number.

## Conclusión

Seguramente hemos aprendido alguna cosa más sobre el lenguaje Javascript. Es sin duda bastante básico todo lo que hemos visto, pero muchas veces no se reflexiona tanto sobre las implicaciones que tienen las operaciones con datos en los lenguajes de tipado dinámico.

Si estamos desarrollando aplicaciones y queremos evitar sorpresas es importante no dejar pasar por alto estos detalles. Espero que este artículo sirva para ayudar a ser más consciente de las implicaciones que tiene el trabajar con Javascript.

Este artículo es obra de *Miguel Angel Alvarez*

Fue publicado / actualizado en *21/04/2023*

Disponible online en <https://desarrolloweb.com/articulos/coercion-de-tipos-javascript>

## Hoisting y otras particularidades del intérprete de Javascript

**Qué es el concepto de hoisting que tenemos en Javascript y cómo funciona el mecanismo de doble pasada del intérprete con respecto a las variables.**



Javascript es un lenguaje bastante particular y tiene algunos mecanismos que no existen en otros lenguajes de programación, como es el caso del **hoisting**. Debemos entender cómo funciona el intérprete de Javascript y su doble pasada para entender el hoisting y, en definitiva, para poder dominar este apasionante lenguaje.

En este artículo vamos a **resumir algunos mecanismos interesantes del intérprete de Javascript** que nos servirán para ser un poco más conscientes de algunos funcionamientos del lenguaje.

### Intérprete de Javascript

Javascript es un **lenguaje interpretado**. Esto quiere decir que no se compila como otros lenguajes del estilo de C o Java, sino que existe un proceso mediante el cual el código se interpreta cada vez que un módulo debe de ejecutarse.

El intérprete simplemente lee el código, realiza los procesos que tiene que hacer para entenderlo y luego se encarga de ejecutar el código, realizando aquellas cosas que se solicitan en el script.

### Doble pasada del intérprete

El intérprete trabaja mediante un proceso de doble pasada, es decir, realiza dos recorridos al código para poder ejecutar y realizar el trabajo solicitado en las líneas del programa. Durante estas dos pasadas realiza las siguientes operaciones.

**1.- En la primera pasada lee todas las variables y funciones** y reserva espacio en memoria para ellas.

**2.- En la segunda pasada se encarga de realizar la ejecución del código.**

En la primera pasada, a pesar de que lea todas las variables, no les asigna valores. Solo es en la segunda pasada en la que, al ejecutarse el código, las variables toman valores.

### Ejemplo de ejecución de un código Javascript y la doble pasada del intérprete

Ahora veamos de manera más práctica qué es lo que pasa cuando se produce esa doble pasada con las variables declaradas en Javascript.

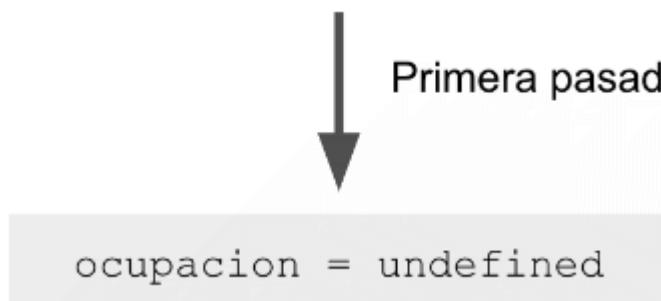
```
alert(ocupacion);
var ocupacion = 'desarrollador';
```



```
alert(ocupacion);
```

Este código tiene una particularidad importante. **Se utiliza una variable antes de su declaración.** En la mayoría de los lenguajes esto provocaría un error, dado que no puedes generalmente hacer uso de una variable que no ha sido creada. Sin embargo, el intérprete de doble pasada de Javascript permite que se ejecute sin problemas.

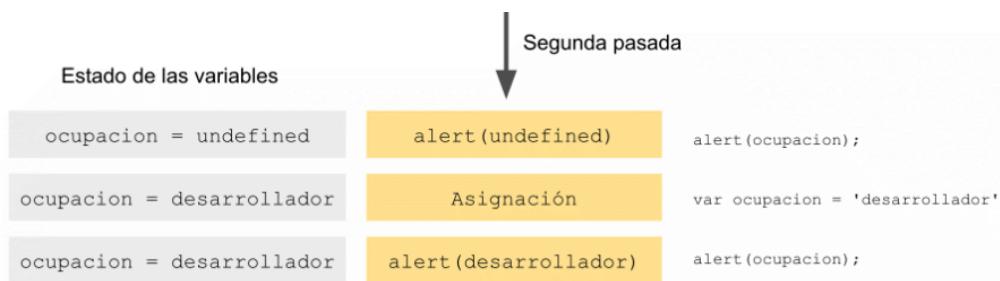
En la primera pasada el intérprete de Javascript reserva un espacio de memoria para la variable "ocupacion". Sin embargo, esa **variable queda indefinida** (valor undefined de Javascript).



En la segunda pasada comienza con una variable "ocupacion" que se encuentra con el valor "undefined", y entonces se encarga de ejecutar el código.

Al ejecutarse la primera línea de código el alert muestra el valor actual de la variable "ocupacion", por lo tanto mostrará "undefined". Esa variable ya existe por la primera pasada! por eso en lugar de mostrar un error, saca el valor indefinido.

En la segunda línea de código se asigna un valor a la variable, por lo que al realizarse la ejecución de la tercera línea de código se muestra el valor asignado a la variable: "desarrollador"



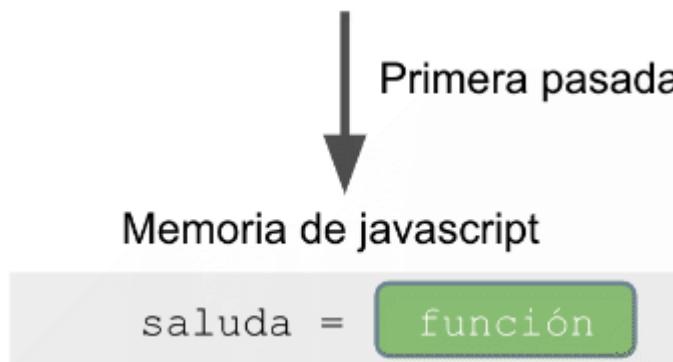
## Funciones y su contexto de ejecución

Ahora tenemos que analizar qué pasa con las funciones y cuando se produce la doble pasada del intérprete, porque ellas también generan un espacio de variables y por tanto\*\* se realiza también una doble pasada\*\*, la primera para la creación de los espacios de memoria y luego la segunda para su ejecución.

En la primera pasada del contexto actual, cada vez que el intérprete de Javascript ve una función, produce un espacio en memoria donde guarda tal función. Reserva simplemente ese espacio en



memoria, donde queda almacenada la función, sin interpretarla todavía y por supuesto sin ejecutarla.



A la vista de ese procedimiento, un código como este no resulta problemático para Javascript, a pesar que estemos invocando la función antes de definirla.

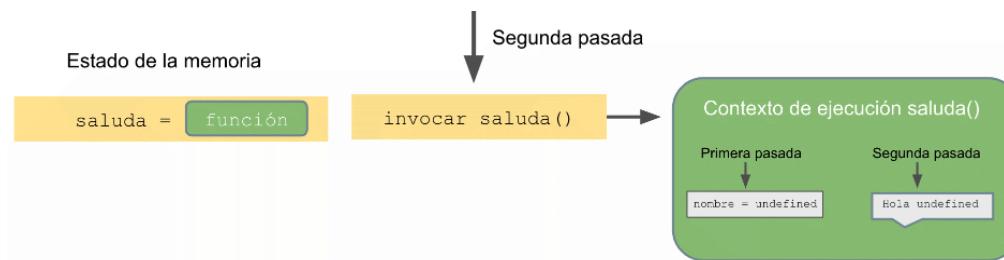
```
saluda();
function saluda() {
    console.log('Hola ' + nombre);
    var nombre = 'Miguel';
}
```

Esto es porque en la primera pasada la función se almacena en la memoria. Por lo que, antes de comenzar la ejecución del código, la función ya realmente existe.

### Doble pasada al ejecutarse una función

La doble pasada también se produce en el tiempo de ejecución de una función. Por lo tanto, al invocarse la función se crea un nuevo contexto de ejecución, que tiene el mismo proceso de interpretación que el contexto global.

- En la **primera pasada del contexto de ejecución de la función** se crean las variables en la memoria, aunque con el valor "undefined".
- En la **segunda pasada del contexto de ejecución de la función** se ejecuta el código, igual que ocurría con el contexto global.



### Pila de ejecución de las funciones

Este mismo proceso del intérprete de Javascript ocurre cada vez que una función se invoca. Se crea un nuevo contexto de ejecución para cada invocación, siendo que **esos contextos pueden estar**

**unos dentro de otros**, en el caso de que unas funciones invoquen a otras.

Los contextos por tanto se van introduciendo de manera que ocupan una pila de ejecución, produciéndose esa doble pasada del intérprete para cada uno de ellos. A medida que las ejecuciones de las funciones finalizan la pila se va vaciando, hasta llegar de nuevo al contexto global.

## Hoisting

Existe un término en Javascript llamado "hoisting", que hace referencia justamente a este procedimiento por el cual **las variables son creadas antes de su uso**.

Dado el concepto de doble pasada que acabamos de aprender, **toda variable creada en un script Javascript es tenida en cuenta, reservando espacio para ella en memoria, antes de su utilización durante el tiempo de ejecución** y esto ocurre tanto en el contexto global como en el contexto de ejecución de una función.

Veamos la función siguiente:

```
function hoisting() {  
    console.log(lenguaje);  
    var lenguaje = "Javascript";  
    console.log(lenguaje);  
}
```

En esta función se define una variable lenguaje, a la que se asigna el valor después de usarla. Dada la regla de hoisting este código sería equivalente a crear la variable en la primera línea de código de la función, tal como aparece aquí.

```
function hoisting() {  
    var lenguaje;  
    console.log(lenguaje);  
    lenguaje = "Javascript";  
    console.log(lenguaje);  
}
```

Este segundo código mostraría cómo realmente Javascript realiza las cosas cuando se invoca la función. Como puedes ver, primero crea las variables en memoria y luego ejecuta el código. Por tanto, ese es el motivo por el cual en el primer console.log() la variable "lenguaje" vale undefined y en el segundo console.log() la variable ya tiene el valor "Javascript".

## Buenas prácticas en la declaración de variables Javascript

Finalmente, y como conclusión, dado el mecanismo de hoisting de Javascript que acabamos de explicar, es una buena práctica crear las variables antes de usarlas, en las primeras líneas de código del contexto global, o bien al comienzo de las funciones.

No es que Javascript lo necesite, pero sí ayuda a que las personas entiendan qué es lo que va a pasar realmente a la hora de ejecutarse el código, evitando posibles sorpresas. Tú, que ya sabes lo que es



el hoisting y cómo funciona, no vas a tener dudas sobre qué ocurre en tu script, pero otras personas que lean el código lo agradecerán.

Este artículo es obra de *Miguel Angel Alvarez*

Fue publicado / actualizado en *08/11/2021*

Disponible online en <https://desarrolloweb.com/articulos/hoisting-interprete-javascript>