Manual de JavaScript

Contenido

[Introducción a la programación web con javascript 3](#_Toc73167147)

[Una definición de alto nivel 3](#_Toc73167148)

[Entonces, ¿qué puede hacer realmente? 4](#_Toc73167149)

[¿Qué está haciendo Javascript en tu página? 5](#_Toc73167150)

[Seguridad del navegador 6](#_Toc73167151)

[Orden de ejecución de JavaScript 6](#_Toc73167152)

[Código interpretado versus compilado 7](#_Toc73167153)

[Código de lado del servidor versus de lado del cliente 7](#_Toc73167154)

[Código dinámico versus estático 7](#_Toc73167155)

[Historia de javascript 8](#_Toc73167156)

[Utilización de JavaScript 8](#_Toc73167157)

[¿Cómo agregas Javascript a tu página? 8](#_Toc73167158)

[JavaScript interno 9](#_Toc73167159)

[JavaScript externo 9](#_Toc73167160)

[Controladores de Javascript en línea 10](#_Toc73167161)

[Estrategias para la carga de scripts 10](#_Toc73167162)

[async y defer 11](#_Toc73167163)

[El Standard ECMAScript 12](#_Toc73167164)

[Especificaciones oficiales 12](#_Toc73167165)

[Un vistazo a ECMAScript 2020 12](#_Toc73167166)

[IMPORT 12](#_Toc73167167)

[OPTIONAL CHAINING 13](#_Toc73167168)

[ARRAY FLAT & FLATMAP 13](#_Toc73167169)

[PROMISE ALLSETTLED 13](#_Toc73167170)

[STRING MATCHALL 14](#_Toc73167171)

[BIG INT 14](#_Toc73167172)

[NULLISH COALESCING OPERATOR 14](#_Toc73167173)

[GLOBALTHIS 15](#_Toc73167174)

[FOR IN MECHANICS 15](#_Toc73167175)

[Conceptos sobre programación JS 15](#_Toc73167176)

[Glosario básico 15](#_Toc73167177)

[Glosario general 16](#_Toc73167178)

[Uso de consola 17](#_Toc73167179)

[**Firefox** 17](#_Toc73167180)

[**Google Chrome** 17](#_Toc73167181)

[**Internet Explorer / Edge** 17](#_Toc73167182)

[**Como crear mensajes para mostrar en la consola del navegador** 17](#_Toc73167183)

[**Métodos para mostrar mensajes en la consola del navegador** 18](#_Toc73167184)

[Uso inicial del lenguaje 19](#_Toc73167185)

[Declaración de variables 19](#_Toc73167186)

[Ámbito de variables 19](#_Toc73167187)

[Tipos de uso básico de Datos 19](#_Toc73167188)

[Conversión con Number, parseInt y parseFloat. 19](#_Toc73167189)

[Videos 19](#_Toc73167190)

Unidad 1:

# Introducción a la programación web con javascript

## [Una definición de alto nivel](https://developer.mozilla.org/es/docs/Learn/JavaScript/First_steps/What_is_JavaScript#una_definici%C3%B3n_de_alto_nivel)

JavaScript es un lenguaje de programación o de secuencias de comandos que te permite implementar funciones complejas en páginas web, cada vez que una página web hace algo más que sentarse allí y mostrar información estática para que la veas, muestra oportunas actualizaciones de contenido, mapas interactivos, animación de Gráficos 2D/3D, desplazamiento de máquinas reproductoras de vídeo, etc., puedes apostar que probablemente Javascript está involucrado. Es la tercera capa del pastel de las tecnologías web estándar, dos de las cuales ([HTML](https://developer.mozilla.org/es/docs/Learn/HTML) y [CSS](https://developer.mozilla.org/es/docs/Learn/CSS)) hemos cubierto con mucho más detalle en otras partes del Área de aprendizaje.

* [HTML](https://developer.mozilla.org/es/docs/Glossary/HTML) es el lenguaje de marcado que usamos para estructurar y dar significado a nuestro contenido web, por ejemplo, definiendo párrafos, encabezados y tablas de datos, o insertando imágenes y videos en la página.
* [CSS](https://developer.mozilla.org/es/docs/Glossary/CSS) es un lenguaje de reglas de estilo que usamos para aplicar estilo a nuestro contenido HTML, por ejemplo, establecer colores de fondo y tipos de letra, y distribuir nuestro contenido en múltiples columnas.
* [JavaScript](https://developer.mozilla.org/es/docs/Glossary/JavaScript) es un lenguaje de secuencias de comandos que te permite crear contenido de actualización dinámica, controlar multimedia, animar imágenes y prácticamente todo lo demás. (Está bien, no todo, pero es sorprendente lo que puedes lograr con unas pocas líneas de código JavaScript).

Las tres capas se superponen muy bien. Tomemos una etiqueta de texto simple como ejemplo. Podemos marcarla usando HTML para darle estructura y propósito:

<p>Player 1: Chris</p>



Luego, podemos agregar algo de CSS a la mezcla para que se vea bien:

p {

font-family: 'helvetica neue', helvetica, sans-serif;

letter-spacing: 1px;

text-transform: uppercase;

text-align: center;

border: 2px solid rgba(0,0,200,0.6);

background: rgba(0,0,200,0.3);

color: rgba(0,0,200,0.6);

box-shadow: 1px 1px 2px rgba(0,0,200,0.4);

border-radius: 10px;

padding: 3px 10px;

display: inline-block;

cursor: pointer;

}

Y finalmente, podemos agregar algo de Javascript para implementar un comportamiento dinámico:

const para = document.querySelector('p');

para.addEventListener('click', updateName);

function updateName() {

let name = prompt('Enter a new name');

para.textContent = 'Player 1: ' + name;

}

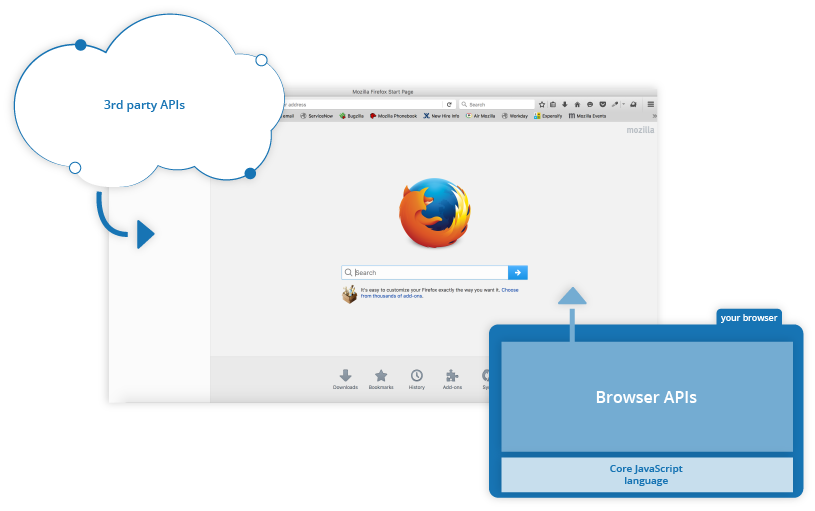
## [Entonces, ¿qué puede hacer realmente?](https://developer.mozilla.org/es/docs/Learn/JavaScript/First_steps/What_is_JavaScript#entonces_%C2%BFqu%C3%A9_puede_hacer_realmente)

El núcleo del lenguaje Javascript de lado del cliente consta de algunas características de programación comunes que te permiten hacer cosas como:

* Almacenar valores útiles dentro de variables. En el ejemplo anterior, por ejemplo, pedimos que ingreses un nuevo nombre y luego almacenamos ese nombre en una variable llamada name.
* Operaciones sobre fragmentos de texto (conocidas como "cadenas" (strings) en programación). En el ejemplo anterior, tomamos la cadena "Player1:" y la unimos a la variable name para crear la etiqueta de texto completa, p. ej. ''Player1: Chris".
* Y ejecuta código en respuesta a ciertos eventos que ocurren en una página web. Usamos un evento [click](https://developer.mozilla.org/es/docs/Web/API/Element/click_event) en nuestro ejemplo anterior para detectar cuándo se hace clic en el botón y luego ejecutar el código que actualiza la etiqueta de texto.
* ¡Y mucho más!

Las API son conjuntos de bloques de construcción de código listos para usar que permiten a un desarrollador implementar programas que de otro modo serían difíciles o imposibles de implementar. Hacen lo mismo para la programación que los kits de muebles prefabricados para la construcción de viviendas — es mucho más fácil tomar paneles precortados y atornillarlos para hacer una estantería que elaborar el diseño tú mismo, que ir y encontrar la madera correcta, cortar todos los paneles del tamaño y la forma correctos, buscar los tornillos del tamaño correcto y luego júntalos para hacer una estantería.

Generalmente se dividen en dos categorías.



Las APIs del navegador están integradas en tu navegador web y pueden exponer datos del entorno informático circundante o realizar tareas complejas y útiles. Por ejemplo:

* La [API del DOM (Document Object Model)](https://developer.mozilla.org/es/docs/Web/API/Document_Object_Model) te permite manipular HTML y CSS, crear, eliminar y cambiar el HTML, aplicar dinámicamente nuevos estilos a tu página, etc. Cada vez que ves aparecer una ventana emergente en una página, o se muestra algún nuevo contenido (como vimos anteriormente en nuestra sencilla demostración), por ejemplo, ese es el DOM en acción.
* La API de Geolocalización recupera información geográfica. Así es como [Google Maps](https://www.google.com/maps) puede encontrar tu ubicación y trazarla en un mapa.
* Las APIs [Canvas](https://developer.mozilla.org/es/docs/Web/API/Canvas_API) y [WebGL](https://developer.mozilla.org/es/docs/Web/API/WebGL_API) te permiten crear gráficos animados en 2D y 3D. Las personas están haciendo cosas increíbles con estas tecnologías web — consulta [Experimentos de Chrome](https://www.chromeexperiments.com/) y [webglsamples](https://webglsamples.org/).
* [APIs de audio y video](https://developer.mozilla.org/es/Apps/Fundamentals/Audio_and_video_delivery) como [HTMLMediaElement](https://developer.mozilla.org/es/docs/Web/API/HTMLMediaElement) y [WebRTC](https://developer.mozilla.org/es/docs/Web/API/WebRTC_API) te permiten hacer cosas realmente interesantes con multimedia, como reproducir audio y video directamente en una página web, o tomar video de tu cámara web y mostrarlo en la computadora de otra persona (prueba nuestra sencilla [demostración instantánea](http://chrisdavidmills.github.io/snapshot/) para hacerte una idea).

Nota: Muchas de las demostraciones anteriores no funcionarán en un navegador antiguo — al experimentar, es una buena idea utilizar un navegador moderno como Firefox, Chrome, Edge u Opera para ejecutar tu código. Deberás considerar las [pruebas en varios navegadores](https://developer.mozilla.org/es/docs/Learn/Tools_and_testing/Cross_browser_testing) con más detalle cuando estés más cerca de entregar el código de producción (es decir, código real que usarán los clientes reales).

Las APIs de terceros no están integradas en el navegador de forma predeterminada y, por lo general, debes obtener su código e información de algún lugar de la Web. Por ejemplo:

* La [API de Twitter](https://dev.twitter.com/overview/documentation) te permite hacer cosas como mostrar tus últimos tweets en tu sitio web.
* La [API de Google Maps](https://developers.google.com/maps/) y la [API de OpenStreetMap](https://wiki.openstreetmap.org/wiki/API) te permiten insertar mapas personalizados en tu sitio web y otras funciones similares.

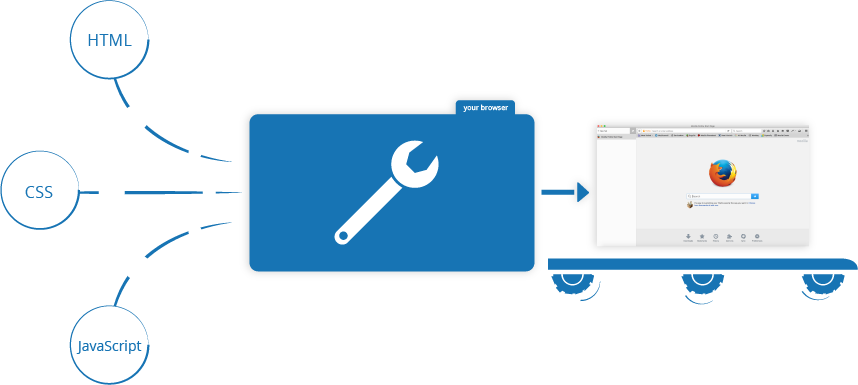
Nota: estas APIs son avanzadas y no cubriremos ninguna de ellas en este módulo. Puedes obtener más información sobre estas en nuestro [módulo de APIs web de lado del cliente](https://developer.mozilla.org/es/docs/Learn/JavaScript/Client-side_web_APIs).

¡También hay mucho más disponible! Sin embargo, no te emociones demasiado todavía. No podrás crear el próximo Facebook, Google Maps o Instagram después de estudiar Javascript durante 24 horas — hay muchos conceptos básicos que cubrir primero. Y es por eso que estás aquí — ¡sigamos adelante!

## [¿Qué está haciendo Javascript en tu página?](https://developer.mozilla.org/es/docs/Learn/JavaScript/First_steps/What_is_JavaScript#%C2%BFqu%C3%A9_est%C3%A1_haciendo_javascript_en_tu_p%C3%A1gina)

Aquí, de hecho, comenzaremos a ver algo de código y, mientras lo hacemos, exploraremos lo que realmente sucede cuando ejecutas Javascript en tu página.

Recapitulemos brevemente sobre la historia de lo que sucede cuando cargas una página web en un navegador (de lo que hablamos por primera vez en nuestro artículo [Cómo funciona CSS](https://developer.mozilla.org/es/docs/Learn/CSS/Introduction_to_CSS/How_CSS_works#how_does_css_actually_work)). Cuando cargas una página web en tu navegador, estás ejecutando tu código (HTML, CSS y JavaScript) dentro de un entorno de ejecución (la pestaña del navegador). Esto es como una fábrica que toma materias primas (el código) y genera un producto (la página web).



Un uso muy común de Javascript es modificar dinámicamente HTML y CSS para actualizar una interfaz de usuario, a través de la API del modelo de objetos del documento (como se mencionó anteriormente). Ten en cuenta que el código de tus documentos web generalmente se cargan y ejecutan en el orden en que aparece en la página. Si Javascript se carga e intenta ejecutarse antes de que se hayan cargado el HTML y el CSS al que afecta, pueden producirse errores. Aprenderás formas de evitar esto más adelante en el artículo, en la sección [Estrategias de carga de scripts](https://developer.mozilla.org/es/docs/Learn/JavaScript/First_steps/What_is_JavaScript#Script_loading_strategies).

### [Seguridad del navegador](https://developer.mozilla.org/es/docs/Learn/JavaScript/First_steps/What_is_JavaScript#seguridad_del_navegador)

Cada pestaña del navegador tiene su propio depósito separado para ejecutar código (estos depósitos se denominan "entornos de ejecución" en términos técnicos) — esto significa que, en la mayoría de los casos, el código de cada pestaña se ejecuta de forma completamente independiente y el código de una pestaña no puede afectar el código en otra pestaña, o en otro sitio web. Esta es una buena medida de seguridad — si este no fuera el caso, los piratas podrían comenzar a escribir código para robar información de otros sitios web y otras cosas muy malas.

Nota: Existen formas de enviar código y datos entre diferentes sitios web/pestañas de manera segura, pero estas son técnicas avanzadas que no cubriremos en este curso.

### [Orden de ejecución de JavaScript](https://developer.mozilla.org/es/docs/Learn/JavaScript/First_steps/What_is_JavaScript#orden_de_ejecuci%C3%B3n_de_javascript)

Cuando el navegador encuentra un bloque de JavaScript, generalmente lo ejecuta en orden, de arriba a abajo. Esto significa que debes tener cuidado con el orden en el que colocas las cosas. Por ejemplo, volvamos al bloque de Javascript que vimos en nuestro primer ejemplo:

const para = document.querySelector('p');

para.addEventListener('click', updateName);

function updateName() {

let name = prompt('Enter a new name');

para.textContent = 'Player 1: ' + name;

}

Aquí seleccionamos un párrafo de texto (línea 1), luego adjuntamos un detector de eventos (línea 3) de modo que cuando se hace clic en el párrafo, el bloque de código updateName() (líneas 5-8) se ejecuta. El bloque de código updateName() (estos tipos de bloques de código reutilizables se denominan "funciones") pide al usuario un nuevo nombre y luego inserta ese nombre en el párrafo para actualizar la pantalla.

Si cambiaras el orden de las dos primeras líneas de código, ya no funcionaría — en su lugar, obtendrías un error en la [consola del desarrollador del navegador](https://developer.mozilla.org/es/docs/Learn/Common_questions/What_are_browser_developer_tools) — TypeError: para is undefined. Esto significa que el objeto para aún no existe, por lo que no podemos agregarle un detector de eventos.

Nota: Este es un error muy común; debes tener cuidado de que los objetos a los que se hace referencia en tu código existan antes de intentar hacer algo con ellos.

### [Código interpretado versus compilado](https://developer.mozilla.org/es/docs/Learn/JavaScript/First_steps/What_is_JavaScript#c%C3%B3digo_interpretado_versus_compilado)

Es posible que escuches los términos interpretados y compilados en el contexto de la programación. En los lenguajes interpretados, el código se ejecuta de arriba a abajo y el resultado de ejecutar el código se devuelve inmediatamente. No tienes que transformar el código en una forma diferente antes de que el navegador lo ejecute. El código se recibe en su forma de texto amigable para el programador y se procesa directamente desde allí.

Los lenguajes compilados, por otro lado, se transforman (compilan) a código máquina antes de que sean ejecutados por la computadora. Por ejemplo, C/C++ se compila a código máquina que luego ejecuta la computadora. El programa se ejecuta desde un formato binario, que se generó a partir del código fuente del programa original.

JavaScript es un lenguaje de programación interpretado ligero. El navegador web recibe el código Javascript en su forma de texto original y ejecuta el script a partir de ahí. Desde un punto de vista técnico, la mayoría de los intérpretes de Javascript modernos utilizan una técnica llamada compilación en tiempo real para mejorar el rendimiento; el código fuente de Javascript se compila en un formato binario más rápido mientras se usa el script, de modo que se pueda ejecutar lo más rápido posible. Sin embargo, Javascript todavía se considera un lenguaje interpretado, ya que la compilación se maneja en el entorno de ejecución, en lugar de antes.

Ambos tipos de lenguaje tienen ventajas, pero no las abordaremos ahora.

### Código de lado del servidor versus de lado del cliente

También puedes escuchar los términos código de lado del servidor y de lado del cliente, especialmente en el contexto del desarrollo web. El código de lado del cliente es un código que se ejecuta en la computadora del usuario — cuando se ve una página web, el código de lado del cliente de la página se descarga, luego se ejecuta y se muestra en el navegador. En este módulo estamos hablando explícitamente de Javascript de lado del cliente.

El código de lado del servidor, por otro lado, se ejecuta en el servidor, luego sus resultados se descargan y se muestran en el navegador. Ejemplos de lenguajes web populares de lado del servidor incluyen a ¡PHP, Python, Ruby, ASP.NET y... JavaScript! Javascript también se puede utilizar como lenguaje de lado del servidor, por ejemplo, en el popular entorno Node.js — puedes obtener más información sobre Javascript de lado del servidor en nuestro tema [Sitios web dinámicos — Programación de lado del servidor](https://developer.mozilla.org/es/docs/Learn/Server-side).

### Código dinámico versus estático

La palabra dinámico se usa para describir tanto a Javascript de lado del cliente como a los lenguajes de lado del servidor — se refiere a la capacidad de actualizar la visualización de una página web/aplicación para mostrar diferentes cosas en diferentes circunstancias, generando contenido nuevo según sea necesario. El código de lado del servidor genera dinámicamente nuevo contenido en el servidor, p. ej. extraer datos de una base de datos, mientras que Javascript de lado del cliente genera dinámicamente nuevo contenido dentro del navegador del cliente, p. ej. creando una nueva tabla HTML, llenándola con los datos solicitados al servidor y luego mostrando la tabla en una página web que se muestra al usuario. El significado es ligeramente diferente en los dos contextos, pero relacionado, y ambos enfoques (de lado del servidor y de lado del cliente) generalmente funcionan juntos.

Una página web sin contenido que se actualiza dinámicamente se denomina estática — simplemente muestra el mismo contenido todo el tiempo.

# Historia de javascript

A principios de los años 90, la mayoría de usuarios que se conectaban a Internet lo hacían con módems a una velocidad máxima de 28.8 kbps. En esa época, empezaban a desarrollarse las primeras aplicaciones web y por tanto, las páginas web comenzaban a incluir formularios complejos.

Con unas aplicaciones web cada vez más complejas y una velocidad de navegación tan lenta, surgió la necesidad de un lenguaje de programación que se ejecutara en el navegador del usuario. De esta forma, si el usuario no rellenaba correctamente un formulario, no se le hacía esperar mucho tiempo hasta que el servidor volviera a mostrar el formulario indicando los errores existentes.

**Brendan Eich**, un programador que trabajaba en Netscape, pensó que podría solucionar este problema adaptando otras tecnologías existentes (como *ScriptEase*) al navegador Netscape Navigator 2.0, que iba a lanzarse en 1995. Inicialmente, Eich denominó a su lenguaje *LiveScript*.

Posteriormente, Netscape firmó una alianza con Sun Microsystems para el desarrollo del nuevo lenguaje de programación. Además, justo antes del lanzamiento Netscape decidió cambiar el nombre por el de JavaScript. La razón del cambio de nombre fue exclusivamente por marketing, ya que Java era la palabra de moda en el mundo informático y de Internet de la época.

La primera versión de Javascript fue un completo éxito y Netscape Navigator 3.0 ya incorporaba la siguiente versión del lenguaje, la versión 1.1. Al mismo tiempo, Microsoft lanzó JScript con su navegador Internet Explorer 3. JScript era una copia de Javascript al que le cambiaron el nombre para evitar problemas legales.

Para evitar una guerra de tecnologías, Netscape decidió que lo mejor sería estandarizar el lenguaje JavaScript. De esta forma, en 1997 se envió la especificación Javascript 1.1 al organismo ECMA *European Computer Manufacturers Association*).

ECMA creó el comité TC39 con el objetivo de *"estandarizar de un lenguaje de* script *multiplataforma e independiente de cualquier empresa"*. El primer estándar que creó el comité TC39 se denominó **ECMA-262**, en el que se definió por primera vez el lenguaje ECMAScript.

Por este motivo, algunos programadores prefieren la denominación *ECMAScript* para referirse al lenguaje JavaScript. De hecho, Javascript no es más que la implementación que realizó la empresa Netscape del estándar ECMAScript.

La organización internacional para la estandarización (ISO) adoptó el estándar ECMA-262 a través de su comisión IEC, dando lugar al estándar ISO/IEC-16262.

# Utilización de JavaScript

## [¿Cómo agregas Javascript a tu página?](https://developer.mozilla.org/es/docs/Learn/JavaScript/First_steps/What_is_JavaScript#%C2%BFc%C3%B3mo_agregas_javascript_a_tu_p%C3%A1gina)

JavaScript se aplica a tu página HTML de manera similar a CSS. Mientras que CSS usa elementos [<link>](https://developer.mozilla.org/es/docs/Web/HTML/Element/link) para aplicar hojas de estilo externas y elementos [<style>](https://developer.mozilla.org/es/docs/Web/HTML/Element/style) para aplicar hojas de estilo internas a HTML, Javascript solo necesita un amigo en el mundo de HTML: el elemento {htmlelement("script")}}. Aprendamos cómo funciona esto.

### [JavaScript interno](https://developer.mozilla.org/es/docs/Learn/JavaScript/First_steps/What_is_JavaScript#javascript_interno)

1. En primer lugar, haz una copia local de nuestro archivo de ejemplo [apply-javascript.html](https://github.com/mdn/learning-area/blob/master/javascript/introduction-to-js-1/what-is-js/apply-javascript.html). Guárdalo en un directorio en algún lugar accesible.
2. Abre el archivo en tu navegador web y en tu editor de texto. Verás que el HTML crea una página web simple que contiene un botón en el que se puede hacer clic.
3. A continuación, ve a tu editor de texto y agrega lo siguiente en tu head, justo antes de tu etiqueta de cierre </head>:

<script>

// Javascript va aquí

</script>

1. Ahora agregaremos algo de Javascript dentro de nuestro elemento [<script>](https://developer.mozilla.org/es/docs/Web/SVG/Element/script) para que la página haga algo más interesante — agrega el siguiente código justo debajo de la línea "// El código Javascript va aquí":

document.addEventListener("DOMContentLoaded", function() {

function createParagraph() {

let para = document.createElement('p');

para.textContent = 'You clicked the button!';

document.body.appendChild(para);

}

const buttons = document.querySelectorAll('button');

for(let i = 0; i < buttons.length ; i++) {

buttons[i].addEventListener('click', createParagraph);

}

});

1. Guarda tu archivo y actualiza el navegador — ahora deberías ver que cuando haces clic en el botón, se genera un nuevo párrafo y se coloca debajo.

Nota: Si tu ejemplo no parece funcionar, sigue los pasos nuevamente y verifica que hiciste todo bien. ¿Guardaste tu copia local del código de inicio como un archivo .html? ¿Agregaste tu elemento [<script>](https://developer.mozilla.org/es/docs/Web/SVG/Element/script) justo antes de la etiqueta </head>? ¿Ingresaste el Javascript exactamente como se muestra? Javascript distingue entre mayúsculas y minúsculas y es muy exigente, por lo que debes ingresar la sintaxis exactamente como se muestra; de lo contrario, es posible que no funcione.

Nota: Puedes ver esta versión en GitHub como [apply-javascript-internal.html](https://github.com/mdn/learning-area/blob/master/javascript/introduction-to-js-1/what-is-js/apply-javascript-internal.html) o ([verla en vivo también](https://mdn.github.io/learning-area/javascript/introduction-to-js-1/what-is-js/apply-javascript-internal.html)).

### [JavaScript externo](https://developer.mozilla.org/es/docs/Learn/JavaScript/First_steps/What_is_JavaScript#javascript_externo)

Esto funciona muy bien, pero ¿y si quisiéramos poner nuestro Javascript en un archivo externo? Exploremos esto ahora.

1. Primero, crea un nuevo archivo en el mismo directorio que tu archivo HTML del ejemplo. Como nombre ponle script.js; asegúrate de que el nombre tenga la extensión .js, ya que así es como se reconoce como JavaScript.
2. Reemplaza tu elemento [<script>](https://developer.mozilla.org/es/docs/Web/SVG/Element/script) actual con lo siguiente:

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

1. Dentro de script.js, agrega el siguiente script:

function createParagraph() {

let para = document.createElement('p');

para.textContent = 'You clicked the button!';

document.body.appendChild(para);

}

const buttons = document.querySelectorAll('button');

for(let i = 0; i < buttons.length ; i++) {

buttons[i].addEventListener('click', createParagraph);

}

1. Guarda y actualiza tu navegador, ¡y deberías ver lo mismo! Funciona igual, pero ahora tenemos nuestro Javascript en un archivo externo. Por lo general, esto es bueno en términos de organización de tu código y para hacerlo reutilizable en varios archivos HTML. Además, el HTML es más fácil de leer sin grandes trozos de script en él.

Nota: Puedes ver esta versión en GitHub como [apply-javascript-external.html](https://github.com/mdn/learning-area/blob/master/javascript/introduction-to-js-1/what-is-js/apply-javascript-external.html) y [script.js](https://github.com/mdn/learning-area/blob/master/javascript/introduction-to-js-1/what-is-js/script.js) ([verla en vivo también](https://mdn.github.io/learning-area/javascript/introduction-to-js-1/what-is-js/apply-javascript-external.html)).

### [Controladores de Javascript en línea](https://developer.mozilla.org/es/docs/Learn/JavaScript/First_steps/What_is_JavaScript#controladores_de_javascript_en_l%C3%ADnea)

Ten en cuenta que a veces te encontrarás con fragmentos de código Javascript real dentro de HTML. Podría verse algo similar a esto:

function createParagraph() {

let para = document.createElement('p');

para.textContent = 'You clicked the button!';

document.body.appendChild(para);

}

<button onclick="createParagraph()">Click me!</button>

Puedes probar esta versión de nuestra demostración a continuación.

Esta demostración tiene exactamente la misma funcionalidad que en las dos secciones anteriores, excepto que el elemento [<button>](https://developer.mozilla.org/es/docs/Web/HTML/Element/button) incluye un controlador onclick en línea para que la función se ejecute cuando se presiona el botón .

Sin embargo, no hagas esto. Es una mala práctica contaminar tu HTML con JavaScript, y es ineficiente; tendrías que incluir el atributo onclick="createParagraph()" en cada botón al que desees que se aplique JavaScript.

El uso de una construcción de Javascript pura te permite seleccionar todos los botones usando una instrucción. El código que usamos anteriormente para cumplir este propósito se ve así:

const buttons = document.querySelectorAll('button');

for(let i = 0; i < buttons.length ; i++) {

buttons[i].addEventListener('click', createParagraph);

}

Esto puede ser un poco más largo que el atributo onclick, pero funcionará para todos los botones, sin importar cuántos haya en la página, ni cuántos se agreguen o eliminen. No es necesario cambiar el JavaScript.

Nota: Intenta editar tu versión de apply-javascript.html y agrega algunos botones más en el archivo. Cuando la vuelvas a cargar, deberías encontrar que todos los botones al hacer clic crearán un párrafo. Limpio, ¿eh?

### [Estrategias para la carga de scripts](https://developer.mozilla.org/es/docs/Learn/JavaScript/First_steps/What_is_JavaScript#estrategias_para_la_carga_de_scripts)

Hay una serie de problemas relacionados con la carga de los scripts en el momento adecuado. ¡Nada es tan simple como parece! Un problema común es que todo el HTML de una página se carga en el orden en que aparece. Si estás utilizando Javascript para manipular elementos en la página (o exactamente, el [Modelo de objetos del documento (en-US)](https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Client-side_web_APIs/Manipulating_documents)), tu código no funcionará si el Javascript se carga y procesa antes que el HTML que estás intentando haga algo.

En los ejemplos de código anteriores, en los ejemplos internos y externos, Javascript se carga y se ejecuta en el encabezado del documento, antes de analizar el cuerpo HTML. Esto podría causar un error, por lo que hemos utilizado algunas construcciones para solucionarlo.

En el ejemplo interno, puedes ver esta estructura alrededor del código:

document.addEventListener("DOMContentLoaded", function() {

...

});

Este es un detector de eventos, que escucha el evento "DOMContentLoaded" del navegador, lo cual significa que el cuerpo HTML está completamente cargado y analizado. El Javascript dentro de este bloque no se ejecutará hasta que se active ese evento, por lo que se evita el error ([aprenderás sobre los eventos](https://developer.mozilla.org/es/docs/Learn/JavaScript/Building_blocks/Events) más adelante en el curso).

En el ejemplo externo, usamos una función de Javascript más moderno para resolver el problema, el atributo defer, que le dice al navegador que continúe descargando el contenido HTML una vez que se ha alcanzado la etiqueta del elemento <script>.

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

En este caso, tanto el script como el HTML se cargarán simultáneamente y el código funcionará.

Nota: En el caso externo, no necesitamos usar el evento DOMContentLoaded porque el atributo defer nos resolvió el problema. No usamos la solución defer para el ejemplo interno de Javascript porque defer solo funciona para scripts externos.

Una solución pasada de moda a este problema solía ser colocar tu elemento script justo en la parte inferior del cuerpo (por ejemplo, justo antes de la etiqueta </body>), para que se cargara después de haber procesado todo el HTML. El problema con esta solución es que la carga/procesamiento del script está completamente bloqueado hasta que se haya cargado el DOM HTML. En sitios muy grandes con mucho JavaScript, esto puede causar un importante problema de rendimiento y ralentizar tu sitio.

### async y defer

En realidad, hay dos modernas características que podemos usar para evitar el problema del bloqueo de script: async y defer (que vimos anteriormente). Veamos la diferencia entre estas dos.

Los scripts cargados con el atributo async (ve más abajo) descargarán el script sin bloquear el renderizado de la página y lo ejecutará tan pronto como el script se termine de descargar. No tienes garantía de que los scripts se ejecuten en un orden específico, solo que no detendrán la visualización del resto de la página. Es mejor usar async cuando los scripts de la página se ejecutan de forma independiente y no dependen de ningún otro script de la página.

Por ejemplo, si tienes los siguientes elementos script:

<script async src="js/vendor/jquery.js"></script>

<script async src="js/script2.js"></script>

<script async src="js/script3.js"></script>

No puedes confiar en el orden en que se cargarán los scripts. jquery.js se puede cargar antes o después de script2.js y script3.js y si este es el caso, cualquier función en esos scripts dependiendo de jquery producirá un error porque jquery no se definirá en el momento en que se ejecute el script.

async se debe usar cuando tienes un montón de scripts en segundo plano para cargar, y solo deseas ponerlos en su lugar lo antes posible. Por ejemplo, tal vez tengas algunos archivos de datos del juego para cargar, que serán necesarios cuando el juego realmente comience, pero por ahora solo deseas continuar mostrando la introducción del juego, los títulos y el lobby, sin que se bloqueen al cargar el script.

Los scripts cargados con el atributo defer (ve a continuación) se ejecutarán en el orden en que aparecen en la página y los ejecutará tan pronto como se descarguen el script y el contenido:

<script defer src="js/vendor/jquery.js"></script>

<script defer src="js/script2.js"></script>

<script defer src="js/script3.js"></script>

Todos los scripts con el atributo defer se cargarán en el orden en que aparecen en la página. Entonces, en el segundo ejemplo, podemos estar seguros de que jquery.js se cargará antes que script2.js y script3.js y que script2.js se cargará antes de script3.js. No se ejecutarán hasta que se haya cargado todo el contenido de la página, lo cual es útil si tus scripts dependen de que el DOM esté en su lugar (por ejemplo, modifican uno o más elementos de la página).

Para resumir:

* async y defer indican al navegador que descargue los scripts en un hilo separado, mientras que el resto de la página (el DOM, etc.) se descarga, por lo que los scripts no bloquean la carga de la página.
* Si tus scripts se deben ejecutar inmediatamente y no tienen ninguna dependencia, utiliza async.
* Si tus scripts necesitan esperar a ser procesados y dependen de otros scripts y/o del DOM en su lugar, cárgalos usando defery coloca tus elementos <script> correspondientes en el orden que desees que el navegador los ejecute.

## El Standard ECMAScript

### Especificaciones oficiales

ECMA ha publicado varios estándares relacionados con ECMAScript. En Junio de 1997 se publicó la primera edición del estándar ECMA-262. Un año después, en Junio de 1998 se realizaron pequeñas modificaciones para adaptarlo al estandar ISO/IEC-16262 y se creó la segunda edición.

La tercera edición del estándar ECMA-262 (publicada en Diciembre de 1999) es la versión que utilizan los navegadores actuales y se puede consultar gratuitamente en http://www.ecma-international.org/publications/standards/Ecma-262.htm

Actualmente se encuentra en desarrollo la cuarta versión de ECMA-262, que podría incluir novedades como paquetes, namespaces, definición explícita de clases, etc.

ECMA también ha definido varios estándares relacionados con ECMAScript, como el estándar ECMA-357, que define una extensión conocida como E4X y que permite la integración de Javascript y XML.

### Un vistazo a ECMAScript 2020

Aunque la próxima especificación del standard ECMAScript aún no está completamente cerrada, ya podemos comenzar a probar muchas de las funcionalidades más interesantes que van a ser incluidas en la siguiente especificación de ECMAScript y que llevará por nombre ECMAScript 2020.

Como hemos comentado en algún [post](https://pablomagaz.com/blog/las-novedades-de-ecmascript2018) previo sobre las especificaciones de ECMASCript, el [TC39](https://ecma-international.org/memento/TC39.htm), es el comité que propone nuevas funcionalidades al stándar mediante un proceso que cuenta con 5 [stages](https://tc39.github.io/process-document/) por las que toda propuesta de nueva funcionalidad tiene que pasar hasta alcanzar el [stage 4](https://github.com/tc39/proposals/blob/master/finished-proposals.md), que es el stage final. Adicionalmente las funcionalidades seleccionadas pasan, en la mayoría de los casos, a formar parte de la especificación ECMAScript seguida del nombre del año, ya que la nomenclatura de ECMAScript 5, ECMAScript 6, etc y que hemos usado durante años previos ya no volverá a ser usada. Aunque aún no está totalmente cerrada la lista completa de funcionalidades que serán incluídas como parte del posible ECMAScript 2020 o ES2020, ya hay una lista bastante interesante de funcionalidades [finalistas](https://github.com/tc39/proposals/blob/master/finished-proposals.md) que merece la pena examinar.

¿Podemos probar ya estas nuevas funcionalidades? Sí, y para probar la gran mayoría de ellas no necesitamos nada más que un navegador Chrome actualizado, pues Chrome suele dar soporte a la mayoría de funcionalidades en cuanto llegan a stage 3, por lo que cuando llegan a stage 4 y son parte del estándar Chrome ya suele soportarlas, aunque existe algún caso en el que puede ser necesario un [plugin](https://babeljs.io/docs/en/babel-plugin-proposal-nullish-coalescing-operator) de Babel.

### IMPORT

Sin lugar a dudas una de las funcionalidades más esperadas e interesantes. Los módulos que importamos habitualmente en nuestras aplicaciones son totalmente estáticos, es decir, no tenemos la capacidad de que el nombre de dicho módulo sea dinámico, no pudiendo realizar cargas condicionales. Con [import](https://github.com/tc39/proposal-dynamic-import) esto va a cambiar ya que finalmente, vamos a poder tener nombres de módulo totalmente dinámicos, pero no solo eso si no que además import nos devuelve una promesa:

import(`./my-modules/${ dynamicModuleName }.js`)

.then(module => {

module.doStuff();

})

.catch(err => console.log(err));

Como vemos con import podemos controlar el resultado de la importación de dicho módulo y además dado que retorna una promesa podemos utilizar [async/await](https://developer.mozilla.org/es/docs/Web/JavaScript/Referencia/Sentencias/funcion_asincrona) para la carga de los módulos.

const getModule = (module) => `./my-modules/${ module }.js`;

const asyncModuleLoad = async () => {

const { fn1, fn2 } = await import(getModule('module1'));

const module2 = await import(getModule('module2'));

module2.doStuff();

};

### OPTIONAL CHAINING

Si siempre has encontrado sumamente tedioso la tarea de comprobar la existencia de una propiedad antes de poder acceder a sus subpropiedades para evitar el [TypeError](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Errors/Cant_access_property) "can't access x of undefined" estás de suerte. Con [optional chaining](https://github.com/tc39/proposal-optional-chaining) esto es historia ¡al fin!. Con el operador "?" podemos indicar que una propiedad puede existir o no sin tener que validar su existencia previamente y asi evitar el TypeError en caso de que no exista, dejando un codigo mucho más limpio:

//Tedioso...

const departmentName = employee.department && employee.department.name;

//Optional chaining!

const departmentName = employee.department?.name;

### ARRAY FLAT & FLATMAP

[Flat](https://developer.mozilla.org/es/docs/Web/JavaScript/Referencia/Objetos_globales/Array/flat) nos permite coger un Array multidimensional, es decir, un Array de Arrays y convertirlo (aplanarlo) en un Array unidemensional de forma rápida y sencilla, pudiendo además, elegir cuantos niveles de profundidad queremos aplanar:

const nestedArray = [1, 2, [3, 4]];

const flattened = nestedArray.flat();

console.log(flattened);

// [1, 2, 3, 4]

const nestedArray2 = [1, 2, [3, 4, [6, 7]]];

const flattened2 = nestedArray2.flat(1);//Solo queremos aplanar un nivel de profundidad

console.log(flattened2);

// [1, 2, 3, 4, [6,7]] // [6,7] queda intacto

[Flatmap](https://developer.mozilla.org/es/docs/Web/JavaScript/Referencia/Objetos_globales/Array/flatmap) funciona de forma muy parecida a flat pero como su propio nombre indica, antes de aplanar el array aplica una función de mapeo a cada uno de los elementos del array y devuelve el resultado aplanado:

const myArray = [1, 2, 3, 4];

myArray.map(x => [x \* 2]);

// [[2], [4], [6], [8]]

nestedArray.flatMap(x => [x \* 2]);

// [2, 4, 6, 8]

### PROMISE ALLSETTLED

El nombre de esta funcionalidad es bastante parecido a [Promise.all](https://developer.mozilla.org/es/docs/Web/JavaScript/Referencia/Objetos_globales/Promise/all) que nos devuelve una promesa cuando todas las promesas pasadas han sido ejecutadas con éxito, o al menos, una de ellas ha fallado:

const p1 = new Promise(resolve => resolve('Promise one'));

const p2 = new Promise((\_ ,reject) => reject('Promise rejected!'));

const p3 = new Promise(resolve => resolve('Promise two'));

Promise.all([p1, p2, p3])

.then(res => console.log(res))

.catch(err => console.log(err));

// Promise rejected!

Esto impone una limitación y es que en el momento que haya una sola promesa rejected el resto no se ejecutara y esto es precisamente lo que podemos evitar con [Promise.allSettled](https://github.com/tc39/proposal-promise-allSettled) que nos devolverá una promesa con el status final de todas y cada una de las promesas y su correspondiente valor de retorno aunque una o mas fallen:

const p1 = new Promise(resolve => resolve('Promise one'));

const p2 = new Promise((\_ ,reject) => reject('Promise rejected!'));

const p3 = new Promise(resolve => resolve('Promise two'));

Promise.allSettled([p1, p2, p3])

.then(res => console.log(res))

.catch(err => console.log(err));

/\*

0: {status: "fulfilled", value: "Promise one"}

1: {status: "rejected", reason: "Promise rejected!"}

2: {status: "fulfilled", value: "Promise two"}

\*/

### STRING MATCHALL

El método [match](https://developer.mozilla.org/es/docs/Web/JavaScript/Referencia/Objetos_globales/String/match) se utiliza para obtener todas las ocurrencias de una expresión regular en una cadena. [MatchAll](https://github.com/tc39/proposal-string-matchall) hace lo mismo pero en lugar de devolvernos un Array, nos devuelve un [Iterador](https://developer.mozilla.org/es/docs/Web/JavaScript/Guide/Iterators_and_Generators) cuyos valores podemos obtener mediante el método next que tienes todos los iteradores o iterarando el resultado mediante el bucle [for of](https://developer.mozilla.org/es/docs/Web/JavaScript/Referencia/Sentencias/for...of):

const str = "hello-hello-hello!";

const regex = /hello/g;

const matches = str.matchAll(regex);

//Podemos llamar al método next del iterador

console.log(matches.next().value);

// ["hello", index: 0, input: "hello-hello-hello!", groups: undefined]

// O usar un bucle for of

for (const match of matches) {

console.log(match);

}

// ["hello", index: 6, input: "hello-hello-hello!", groups: undefined]

// ["hello", index: 12, input: "hello-hello-hello!", groups: undefined

### BIG INT

JavaScript tiene algunas limitaciones con respecto a los números, siendo 9007199254740991 (2 elevado a 53), el numero más grande posible que podíamos manejar en Javascript con el tipo primitivo [Number](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Number). Esto supone algunos problemas que afectan incluso al manejo de ciertas APIS cuando los ids numéricos son [muy grandes](https://developer.twitter.com/en/docs/basics/twitter-ids) como ha sido el caso de los ids usados por [Twitter](https://developer.twitter.com/en/docs/basics/twitter-ids), y cuyo valor superaba el valor de Number.MAX\_SAFE\_INTEGER, obligando a tener que tratarlos como strings. Con BigInt esa limitación queda atrás y podemos crear números enteros más grandes añadiendo una "n" al final del entero o mediante la función [BigInt](https://developer.mozilla.org/en-US/docs/Glossary/BigInt):

const bigint = BigInt(Number.MAX\_SAFE\_INTEGER) + 2n;

// 9007199254740996n

console.log(bigint > Number.MAX\_SAFE\_INTEGER);

// true

### NULLISH COALESCING OPERATOR

Este operador (??) es un operador lógico similar al operador [OR](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Logical_Operators#Logical_OR_2) con la diferencia de que el operador "??" solo tiene en cuenta los valores undefined o nulos, pero no los false como sucede con el operador ||:

console.log(undefined || "default!"); // default!

console.log(null || "default!"); // default!

console.log(0 || "default!"); // default!

console.log(false || "default!"); // default

console.log(undefined ?? "default!"); // default!

console.log(null ?? "default!"); // default!

console.log(0 ?? "default!"); // 0

console.log(false ?? "default!"); // false

Este operador resulta muy útil cuando es usado con el operador de optional chaining que hemos visto anteriormente:

const departmentName = employee.department?.name ?? "no department!";

### GLOBALTHIS

El acceso al objeto global (this) en Javascript siempre ha dependido del entorno, usando [window](https://developer.mozilla.org/es/docs/Web/API/Window) o [self](https://developer.mozilla.org/en-US/docs/Web/API/Window/self) en el browser, o [global](https://nodejs.org/api/globals.html#globals_global) en NodeJs pero no había una forma estandarizada de acceder al objeto global hasta la llegada de [globalthis](https://github.com/tc39/proposal-global):

console.log(typeof globalThis);

// Object

console.log(globalThis);

// Window {parent: Window, opener: null, top: Window, length: 2, frames: Window, …}

### FOR IN MECHANICS

Un problema, con el que todos hemos tenido que lidiar alguna vez es que cuando accedemos a las propiedades de un objeto usando [Object.keys](https://developer.mozilla.org/es/docs/Web/JavaScript/Referencia/Objetos_globales/Object/keys), el bucle [for in](https://developer.mozilla.org/es/docs/Web/JavaScript/Referencia/Sentencias/for...in) o [JSON.stringify](https://developer.mozilla.org/es/docs/Web/JavaScript/Referencia/Objetos_globales/JSON/stringify) el orden de las propiedades no está garantizado sencillamente porque los objetos son una collection no ordenada de propiedades.

const myObject = {

a: 1,

b: 2,

c: 3

}

Object.keys(myObject); // ["a", "b", "c"] orden NO garantizado!

for(let k in myObject) {

console.log(k);

} // a, b, c orden NO garantizado!

Con la inclusión de esta [for-in-order](https://github.com/tc39/proposal-for-in-order), podemos decir adiós a estos problemas cuando iteremos las propiedades de los objetos.

Pues estas son, todas las funcionalidades que en el momento de escribir este post, ya son propuesas finales e incluídas en el stándar ECMAScript 2020 pero es posible que se incluyan algunas más, que actualmente están en [stage 3](https://github.com/tc39/proposals#stage-3) antes de cerrar la específica final. Estaremos atentos :)

# Conceptos sobre programación JS

## Glosario básico

**Script:** cada uno de los programas, aplicaciones o trozos de código creados con el lenguaje de programación JavaScript. Unas pocas líneas de código forman un script y un archivo de miles de líneas de Javascript también se considera un script. A veces se traduce al español directamente como "guión", aunque script es una palabra más adecuada y comúnmente aceptada.

**Sentencia:** cada una de las instrucciones que forman un script.

**Palabras reservadas:** son las palabras (en inglés) que se utilizan para construir las sentencias de Javascript y que por tanto no pueden ser utilizadas libremente. Las palabras actualmente reservadas por Javascript son: break, case, catch, continue, default, delete, do, else, finally, for, function, if, in, instanceof, new, return, switch, this, throw, try, typeof, var, void, while, with.

## [Glosario general](https://lenguajejs.com/javascript/introduccion/conceptos-basicos/#glosario-general)

**Programa:** En programación se suele llamar «programa» a el conjunto total de código que desarrollas. En Javascript, quizás el término más utilizado es aplicación web (cuando es un desarrollo con mucha cantidad de Javascript). También se suelen generalizar utilizando términos como «script» o «código Javascript».

**Algoritmo:** Un algoritmo es un conjunto de pasos conocidos, en un determinado orden, para conseguir realizar una tarea satisfactoriamente y lograr un objetivo.

**Comentarios:** Los comentarios en nuestro código son fragmentos de texto o anotaciones que el navegador ignora y no repercuten en el programa. Sirven para dejar por escrito detalles importantes para el programador. De esta forma cuando volvamos al código, nos será más rápido comprenderlo. Es una buena costumbre comentar en la medida de lo posible nuestro código.

**Indentación:** Se llama indentar a la acción de colocar espacios o tabuladores antes del código, para indicar si nos encontramos dentro de un if, de un bucle, etc... Esta práctica es muy importante y necesaria, y más adelante profundizaremos en ella.

**Variables:** Es el nombre genérico que se le da a pequeños espacios de memoria donde guardas una información determinada, de forma muy similar a las incógnitas en matemáticas. Un programa puede tener muchas variables, y cada una de ellas tendrá un nombre, un valor y un tipo de dato. El nombre se utiliza para diferenciarlas unas de otras y hacer referencia a ellas, el valor es la información que contienen y el tipo de dato es la naturaleza de ese valor. Se llaman variables porque podemos cambiarle su valor a lo largo del programa, según necesitemos.

x = 5; // nombre: x, valor: 5, tipo de dato: número

y = "Hola"; // nombre: y, valor: Hola, tipo de dato: texto

Manz = "me"; // nombre: Manz, valor: me, tipo de dato: texto

**Constantes:** Es el mismo concepto de una variable, salvo que en este caso, la información que contiene es siempre la misma (no puede variar).

**Funciones:** Cuando comenzamos a programar, nuestro código se va haciendo cada vez más y más grande, por lo que hay que buscar formas de organizarlo y mantenerlo lo más simple posible. Las funciones son agrupaciones de código que, entre otras cosas, evitan que tengamos que escribir varias veces lo mismo en nuestro código. Una función contendrá una o mas acciones a realizar y cada vez que ejecutemos una función, se realizarán todas ellas.

**Parámetros:** Es el nombre que reciben las variables que se le pasan a las funciones. Muchas veces también se les denomina argumentos.

**Bucles:** Cuando estamos programando, muchas veces necesitaremos realizar tareas repetitivas. Una de las ventajas de la programación es que permite automatizar acciones y no es necesario hacerlas varias veces. Los bucles permiten indicar el número de veces que se repetirá una acción. De esta forma, sólo la escribimos una vez en nuestro código, y simplemente indicamos el número de veces que queremos que se repita.

**Iteración:** Cuando el programa está en un bucle repitiendo varias veces la misma tarea, cada una de esas repeticiones se denomina iteración.

**Librería:** Muchas veces, desarrollamos código que resuelve tareas o problemas que, posteriormente, querremos reutilizar en otros programas. Cuando eso ocurre, en Javascript se suele empaquetar el código en lo que se llaman librerías, que no es más que código listo para que otros programadores puedan utilizarlo fácilmente en sus programas y beneficiarse de las tareas que resuelven de forma muy sencilla.

## Uso de consola

Todos los navegadores web modernos incluyen una consola, una herramienta creada para desarrolladores, usuarios avanzados o cualquier persona.

Tiene como finalidad mostrar mensajes de información, error o alerta que se reciben al hacer las peticiones para cargar desde la red los elementos incluidos en las páginas.

Además incluye inspectores y verdaderos depuradores de código.

También permite interactuar con la página, ejecutando expresiones o comandos de JavaScript.

El propósito es probar el funcionamiento de las páginas o aplicaciones y descubrir posibles errores en el código.

Esta consola la utilizan los desarrolladores, pero cualquier usuario puede probarla y así estar al tanto del contenido de las páginas de cualquier sitio web.

La consola de Javascript es un panel que se abre en alguna porción pequeña del navegador para no interferir con el contenido principal, aunque se puede mover y cambiar su tamaño.

En cada navegador web posee un aspecto y se abre de forma diferente.

### Firefox

En el navegador Firefox podemos usar las teclas Control + Mayus + J para abrir el panel de la consola.

También podemos emplear el menú Herramientas y en Desarrollador web escoger: "Consola del navegador".

La consola muestra varias pestañas en las que se pueden encontrar información de las peticiones HTTP (los elementos de la página que se cargan de la red), los errores y análisis de CSS, JavaScript, errores y advertencias de seguridad y los Logs (mensajes).

El resultado de las peticiones HTTP se muestra de color negro, CSS de color azul, Javascript amarillo y los errores o advertencias de seguridad de color rojo.

### Google Chrome

En el navegador Google Chrome de forma similar a Firefox podemos usar las teclas Control + Mayus + I.

También podemos dar un clic en el icono del menú en la esquina superior derecha y en Herramientas escoger "Herramientas para desarrolladores".

### Internet Explorer / Edge

En el navegador Internet Explorer basta con presionar la tecla F12.

También podemos dar en clic en el menú Herramientas y escoger: “Mas Herramientas” y después seleccionar "Herramientas de desarrollo".

### Como crear mensajes para mostrar en la consola del navegador

La consola se puede emplear en conjunto con Javascript en muchas operaciones y para mostrar un sencillo mensaje en una página web, solo es necesario incluir en cualquier parte de la página o de un archivo HTML el siguiente script:

<script>

console.log("Texto del mensaje")

</script>

Como podrás observar, el código Javascript posee las etiquetas <script> y </script> para indicarle al navegador que el código existente en esas etiquetas debe ser procesado como lenguaje javascript.

En este caso “console.log()” con un mensaje de texto dentro de los paréntesis, indican el mensaje a mostrar en la consola.

También es posible utilizar el caracter \n para indicar un salto de línea entre mensajes.

### Métodos para mostrar mensajes en la consola del navegador

➔ El método anterior "console.log()" muestra solo un mensaje, sin clasificarlo.

➔ El método "console.info()" muestra un icono azul de información y a continuación el texto del mensaje. En algunos exploradores, se muestra simplemente como el .log().

➔ El método "console.error()" muestra un icono rojo de error con el mensaje coloreado en rojo.

➔ El método "console.warn()" muestra un icono amarillo de advertencia.

Por ejemplo podés utilizar el siguiente código y ver el resultado del mismo en la consola de tu navegador escribiendo el siguiente script:

<script>

if (window.console){

console.log("Ejemplo de mensaje o log predeterminado");

console.error("Ejemplo de mensaje de error");

console.warn("Ejemplo de mensaje de advertencia");

console.info("Ejemplo de mensaje de información");

}

</script>

#### Otros métodos de la consola

➔ console.dir(object). Imprime una representación del objeto con JavaScript.

Por ejemplo: "console.dir(document.links);" muestra todos los enlaces en la página.

➔ console.dirxml(object). Imprime una representación en XML de de los elementos descendientes de objeto.

➔ console.clear(). Limpia la ventana de la consola.

### Personalizar el formato del mensaje en la consola del navegador

Es posible aplicar formato al mensaje de texto usando CSS, aunque algunas configuraciones solo se verán correctamente en los navegadores Google Chrome y Edge.

Por ejemplo, para darle el color verde y usar una fuente grande usa:

console.log("%cMensaje de color verde y grande", "color: green; font-size: x-large");

## Uso inicial del lenguaje

## Declaración de variables

## Ámbito de variables

## Tipos de uso básico de Datos

## Conversión con Number, parseInt y parseFloat.

## Videos

* **Configurar área de trabajo en JavaScript** <https://youtu.be/6pLMiqMqBfo>
* **JavaScript inicial** <https://drive.google.com/file/d/139XlkwFzzphJKmxgaZbxmck4VP4CNn2t/view?usp=sharing>