

# Una introducción a JavaScript

Veamos qué tiene de especial JavaScript, qué podemos lograr con este lenguaje y qué otras tecnologías se integran bien con él.

## ¿Qué es JavaScript?

*JavaScript* fue creado para “dar vida a las páginas web”.

Los programas en este lenguaje se llaman *scripts*. Se pueden escribir directamente en el HTML de una página web y ejecutarse automáticamente a medida que se carga la página.

Los scripts se proporcionan y ejecutan como texto plano. No necesitan preparación especial o compilación para correr.

En este aspecto, JavaScript es muy diferente a otro lenguaje llamado [Java](#).

### ¿Por qué se llama [JavaScript](#)?

Cuando JavaScript fue creado, inicialmente tenía otro nombre: “LiveScript”. Pero Java era muy popular en ese momento, así que se decidió que el posicionamiento de un nuevo lenguaje como un “Hermano menor” de Java ayudaría.

Pero a medida que evolucionaba, JavaScript se convirtió en un lenguaje completamente independiente con su propia especificación llamada [ECMAScript](#), y ahora no tiene ninguna relación con Java.

Hoy, JavaScript puede ejecutarse no solo en los navegadores, sino también en servidores o incluso en cualquier dispositivo que cuente con un programa especial llamado [El motor o intérprete de JavaScript](#).

El navegador tiene un motor embebido a veces llamado una “Máquina virtual de JavaScript”.

Diferentes motores tienen diferentes “nombres en clave”. Por ejemplo:

- [V8](#) – en Chrome, Opera y Edge.
- [SpiderMonkey](#) – en Firefox.
- ...Existen otros nombres en clave como “Chakra” para IE, “JavaScriptCore”, “Nitro” y “SquirrelFish” para Safari, etc.

Es bueno recordar estos términos porque son usados en artículos para desarrolladores en internet. También los usaremos. Por ejemplo, si “la característica X es soportada por V8”, entonces probablemente funciona en Chrome, Opera y Edge.

## ¿Cómo trabajan los motores?

Los motores son complicados, pero los fundamentos son fáciles.

- 1 El motor (embebido si es un navegador) lee ("analiza") el script.
- 2 Luego convierte ("compila") el script a lenguaje de máquina.
- 3 Por último, el código máquina se ejecuta, muy rápido.

El motor aplica optimizaciones en cada paso del proceso. Incluso observa como el script compilado se ejecuta, analiza los datos que fluyen a través de él y aplica optimizaciones al código máquina basadas en ese conocimiento.

## ¿Qué puede hacer JavaScript en el navegador?

El JavaScript moderno es un lenguaje de programación "seguro". No proporciona acceso de bajo nivel a la memoria ni a la CPU (UCP); ya que se creó inicialmente para los navegadores, los cuales no lo requieren.

Las capacidades de JavaScript dependen en gran medida en el entorno en que se ejecuta. Por ejemplo, [Node.JS](#) soporta funciones que permiten a JavaScript leer y escribir archivos arbitrariamente, realizar solicitudes de red, etc.

En el navegador JavaScript puede realizar cualquier cosa relacionada con la manipulación de una página web, interacción con el usuario y el servidor web.

Por ejemplo, en el navegador JavaScript es capaz de:

- Agregar nuevo HTML a la página, cambiar el contenido existente y modificar estilos.
- Reaccionar a las acciones del usuario, ejecutarse con los clics del ratón, movimientos del puntero y al oprimir teclas.
- Enviar solicitudes de red a servidores remotos, descargar y cargar archivos (Tecnologías llamadas [AJAX](#) y [COMET](#)).
- Obtener y configurar cookies, hacer preguntas al visitante y mostrar mensajes.
- Recordar datos en el lado del cliente con el almacenamiento local ("local storage").

## ¿Qué NO PUEDE hacer JavaScript en el navegador?

Las capacidades de JavaScript en el navegador están limitadas para proteger la seguridad de usuario. El objetivo es evitar que una página maliciosa acceda a información privada o dañe los datos de usuario.

Ejemplos de tales restricciones incluyen:

- JavaScript en el navegador no puede leer y escribir arbitrariamente archivos en el disco duro, copiarlos o ejecutar programas. No tiene acceso directo a funciones del Sistema operativo (OS).

Los navegadores más modernos le permiten trabajar con archivos, pero el acceso es limitado y solo permitido si el usuario realiza ciertas acciones, como "arrastrar" un archivo a la ventana del navegador o seleccionarlo por medio de una etiqueta `<input>`.

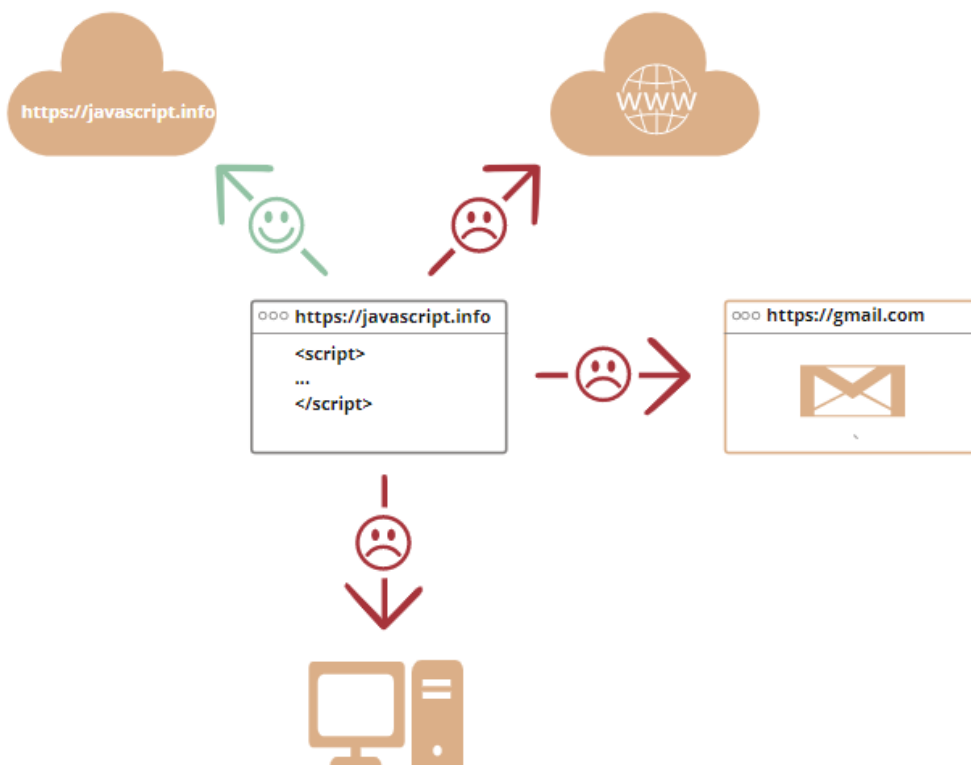
Existen maneras de interactuar con la cámara, micrófono y otros dispositivos, pero eso requiere el permiso explícito del usuario. Por lo tanto, una página habilitada para JavaScript no puede habilitar una cámara web para observar el entorno y enviar la información a la NSA.

- Diferentes pestañas y ventanas generalmente no se conocen entre sí. A veces sí lo hacen: por ejemplo, cuando una ventana usa JavaScript para abrir otra. Pero incluso en este caso, JavaScript no puede acceder a la otra si provienen de diferentes sitios (de diferente dominio, protocolo o puerto).

Esta restricción es conocida como "política del mismo origen" ("Same Origin Policy"). Es posible la comunicación, pero ambas páginas deben acordar el intercambio de datos y también deben contener el código especial de JavaScript que permite controlarlo. Cubriremos esto en el tutorial.

De nuevo: esta limitación es para la seguridad del usuario. Una página de <http://algunsitio.com>, que el usuario haya abierto, no debe ser capaz de acceder a otra pestaña del navegador con la URL <http://gmail.com> y robar la información de esta otra página.

- JavaScript puede fácilmente comunicarse a través de la red con el servidor de donde la página actual proviene. Pero su capacidad para recibir información de otros sitios y dominios está bloqueada. Aunque sea posible, esto requiere un acuerdo explícito (expresado en los encabezados HTTP) desde el sitio remoto. Una vez más: esto es una limitación de seguridad.



Tales limitaciones no existen si JavaScript es usado fuera del navegador; por ejemplo, en un servidor. Los navegadores modernos también permiten complementos y extensiones que pueden solicitar permisos extendidos.

## ¿Qué hace a JavaScript único?

Existen al menos *tres* cosas geniales sobre JavaScript:

- Completa integración con HTML y CSS.
- Las cosas simples se hacen de manera simple.
- Soportado por la mayoría de los navegadores y habilitado de forma predeterminada.

JavaScript es la única tecnología de los navegadores que combina estas tres cosas.

Eso es lo que hace a JavaScript único. Por esto es la herramienta mas extendida para crear interfaces de navegador.

Dicho esto, JavaScript también permite crear servidores, aplicaciones móviles, etc.

## Lenguajes “por arriba de” JavaScript

La sintaxis de JavaScript no se adapta a las necesidades de todos. Personas diferentes querrán diferentes características.

Esto es algo obvio, porque los proyectos y requerimientos son diferentes para cada persona.

Así que recientemente han aparecido una gran cantidad de nuevos lenguajes, los cuales son *transpilados* (convertidos) a JavaScript antes de ser ejecutados en el navegador.

Las herramientas modernas hacen la conversión (Transpilación) muy rápida y transparente, permitiendo a los desarrolladores codificar en otros lenguajes y convertirlo automáticamente detrás de escena.

Ejemplos de tales lenguajes:

- [CoffeeScript](#) Es una “sintaxis azucarada” para JavaScript. Introduce una sintaxis corta, permitiéndonos escribir un código más claro y preciso. Usualmente desarrolladores de Ruby prefieren este lenguaje.
- [TypeScript](#) se concentra en agregar “tipado estricto” (“strict data typing”) para simplificar el desarrollo y soporte de sistemas complejos. Es desarrollado por Microsoft.
- [Flow](#) también agrega la escritura de datos, pero de una manera diferente. Desarrollado por Facebook.
- [Dart](#) es un lenguaje independiente, tiene su propio motor que se ejecuta en entornos que no son de navegador (como aplicaciones móviles), pero que también se puede convertir/transpilar a JavaScript. Desarrollado por Google.
- [Brython](#) es un transpilador de Python a JavaScript que permite escribir aplicaciones en Python puro sin JavaScript.
- [Kotlin](#) es un lenguaje moderno, seguro y conciso que puede apuntar al navegador o a Node.

Hay más. Por supuesto, incluso si nosotros usamos alguno de estos lenguajes transpilados, deberíamos conocer también JavaScript para realmente entender qué estamos haciendo.

## ↳ Editores de Código

Un editor de código es el lugar donde los programadores pasan la mayor parte de su tiempo.

Hay dos principales tipos de editores de código: IDEs y editores livianos. Muchas personas usan una herramienta de cada tipo.

### IDE

El término **IDE** (siglas en inglés para Integrated Development Environment, Ambiente Integrado de Desarrollo) se refiere a un poderoso editor con varias características que operan usualmente sobre un "proyecto completo". Como el nombre sugiere, no sólo es un editor, sino un completo "ambiente de desarrollo".

Un IDE carga el proyecto (el cual puede ser de varios archivos), permite navegar entre archivos, provee autocompletado basado en el proyecto completo (no sólo el archivo abierto), e integra un sistema de control de versiones (como [git](#)), un ambiente de pruebas, entre otras cosas a "nivel de proyecto".

Si aún no has seleccionado un IDE, considera las siguientes opciones:

- [Visual Studio Code](#) (Multiplataforma, gratuito).
- [WebStorm](#) (Multiplataforma, de pago).

Para Windows, también está "Visual Studio", no lo confundamos con "Visual Studio Code". "Visual Studio" es un poderoso editor de pago sólo para Windows, idóneo para la plataforma .NET. Una versión gratuita es de este editor se llama [Visual Studio Community](#).

Muchos IDEs son de paga, pero tienen un periodo de prueba. Su costo usualmente es pequeño si lo comparamos al salario de un desarrollador calificado, así que sólo escoge el mejor para ti.

### Editores livianos

Los "editores livianos" no son tan poderosos como los IDEs, pero son rápidos, elegantes y simples.

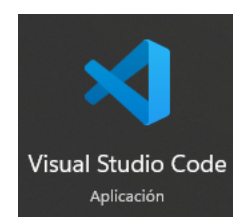
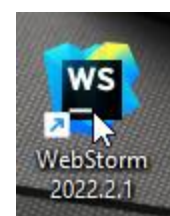
Son usados principalmente para abrir y editar un archivo al instante.

La diferencia principal entre un "editor liviano" y un "IDE" es que un IDE trabaja a nivel de proyecto, por lo que carga mucha más información desde el inicio, analiza la estructura del proyecto si así lo requiere y continua. Un editor liviano es mucho más rápido si solo necesitamos un archivo.

En la práctica, los editores livianos pueden tener montones de plugins incluyendo analizadores de sintaxis a nivel de directorio y autocompletado, por lo que no hay un límite estricto entre un editor liviano y un IDE.

Las siguientes opciones merecen tu atención:

- [Sublime Text](#) (multiplataforma, shareware).
- [Notepad++](#) (Windows, gratuito).
- [Vim](#) y [Emacs](#) son también interesantes si sabes cómo usarlos.





# La etiqueta "script"

Los programas de JavaScript se pueden insertar en casi cualquier parte de un documento HTML con el uso de la etiqueta `<script>`.

Por ejemplo:

```
ejercicioEjemplo.html x
1  <!DOCTYPE html>
2  <html lang="en">
3  <head>
4      <meta charset="UTF-8">
5      <title>Ejemplo 1 asignatura JavaScript</title>
6  </head>
7  <body>
8  <script>
9      alert("Hola Estudiantes");
10 </script>
11 </body>
12 </html>
```

Etiqueta para JavaScript

## Scripts externos

Si tenemos un montón de código JavaScript, podemos ponerlo en un archivo separado.

Los archivos de script se adjuntan a HTML con el atributo `src`:

```
1_ejercicioHolaMundo.html x
1_ejercicioHolaMundo.html > ...
1  <!DOCTYPE html>
2  <html lang="en">
3  <head>
4      <meta charset="UTF-8" />
5      <title>Ejemplo1</title>
6  </head>
7  <body>
8      <script src="2_ejemploExterno.js"></script>
9  </body>
10 </html>
```

```
2_ejemploExterno.js x
2_ejemploExterno.js
1  alert("hola desde un archivo externo");
```

Para adjuntar varios scripts, usa varias etiquetas:

```
1 <script src="/js/script1.js"></script>
2 <script src="/js/script2.js"></script>
3 ...
```

### **i** Por favor tome nota:

Como regla general, solo los scripts más simples se colocan en el HTML. Los más complejos residen en archivos separados.

La ventaja de un archivo separado es que el navegador lo descargará y lo almacenará en [caché](#).

Otras páginas que hacen referencia al mismo script lo tomarán del caché en lugar de descargarlo, por lo que el archivo solo se descarga una vez.

Eso reduce el tráfico y hace que las páginas sean más rápidas.

### **⚠** Si se establece `src`, el contenido del script se ignora.

Una sola etiqueta `<script>` no puede tener el atributo `src` y código dentro.

Esto no funcionará:

```
1 <script src="file.js">
2   alert(1); // el contenido se ignora porque se estableció
3 </script>
```

Debemos elegir un `<script src="...">` externo o un `<script>` normal con código.

El ejemplo anterior se puede dividir en dos scripts para que funcione:

```
1 <script src="file.js"></script>
2 <script>
3   alert(1);
4 </script>
```



Descargar Node.js

Node.js® es un entorno de ejecución para JavaScript construido con V8, motor de JavaScript de Chrome.

New security releases now available for 18.x, 16.x, and 14.x release lines

Descargar para Windows (x64)

16.17.1 LTS

Recomendado para la mayoría

18.9.1 Actual

Últimas características

[Otras Descargas](#) | [Cambios](#) | [Documentación de la API](#)

[Otras Descargas](#) | [Cambios](#) | [Documentación de la API](#)

O eche un vistazo al Programa de soporte a largo plazo (LTS)



```
JS holaMundo.js
1 console.log("Hola, buenos dias");
```



con Node.js

```
P$ D:\2022\universidad\programacion II\ejemplo> node holaMundo.js
Hola, buenos dias
```

## Estructura del código

Lo primero que estudiaremos son los bloques de construcción del código.

### Sentencias

```
<script>
  alert("Bienvenidos a la asignatura de JavaScript");
  alert("Hola Estudiantes");
</script>
```

Las sentencias son construcciones sintácticas y comandos que realizan acciones.

Ya hemos visto una sentencia, `alert('¡Hola mundo!')`, que muestra el mensaje "¡Hola mundo!".

Podemos tener tantas sentencias en nuestro código como queramos, las cuales se pueden separar con un punto y coma.

Por ejemplo, aquí separamos "Hello World" en dos alerts:



Aquí, JavaScript interpreta el salto de línea como un punto y coma “implícito”. Esto se denomina [inserción automática de punto y coma](#).

**En la mayoría de los casos, una nueva línea implica un punto y coma. Pero “en la mayoría de los casos” no significa “siempre”!**

```
<script>
    alert("Hola");
    ["primer mensaje", "segundo mensaje", "tercer mensaje"].forEach(alert);
</script>
```

Recomendamos colocar puntos y coma entre las sentencias, incluso si están separadas por saltos de línea. Esta regla está ampliamente adoptada por la comunidad. Notemos una vez más que es posible omitir los puntos y coma la mayoría del tiempo. Pero es más seguro, especialmente para un principiante, usarlos.

## Comentarios

A medida que pasa el tiempo, los programas se vuelven cada vez más complejos. Se hace necesario agregar *comentarios* que describan lo que hace el código y por qué.

Los comentarios se pueden poner en cualquier lugar de un script. No afectan su ejecución porque el motor simplemente los ignora.

**Los comentarios de una línea comienzan con dos caracteres de barra diagonal `//`.**

El resto de la línea es un comentario. Puede ocupar una línea completa propia o seguir una sentencia.

Como aquí:

```
1 // Este comentario ocupa una línea propia.
2 alert('Hello');
3
4 alert('World'); // Este comentario sigue a la sentencia.
```

**Los comentarios de varias líneas comienzan con una barra inclinada y un asterisco `/*` y terminan con un asterisco y una barra inclinada `*/`.**

# Variables

La mayoría del tiempo, una aplicación de JavaScript necesita trabajar con información. Aquí hay 2 ejemplos:

1. Una tienda en línea – La información puede incluir los bienes a la venta y un “carrito de compras”.
2. Una aplicación de chat – La información puede incluir los usuarios, mensajes, y mucho más.

Utilizamos las variables para almacenar esta información.

## Una variable

Una **variable** es un “almacén con un nombre” para guardar datos. Podemos usar variables para almacenar golosinas, visitantes, y otros datos.

Para generar una variable en JavaScript, se usa la palabra clave `let`.

La siguiente declaración genera (en otras palabras: *declara* o *define*) una variable con el nombre “message”:

```
<script>
  let message;
  message = "Este es un mensaje";
  alert(message);
</script>
```

También podemos declarar variables múltiples en una sola línea:

Esto puede parecer más corto, pero no lo recomendamos. Por el bien de la legibilidad, por favor utiliza una línea por variable.

La versión de líneas múltiples es un poco más larga, pero se lee más fácil:

```
<script>
  let user = 'John', age = 25, message = 'Hola';
  alert(age);
</script>
```

```
<script>
  let user = "John";
  let age = 25;
  let message = "Hola";
  alert(age);
</script>
```

## **var en vez de let**

En scripts más viejos, a veces se encuentra otra palabra clave: `var` en lugar de `let`:

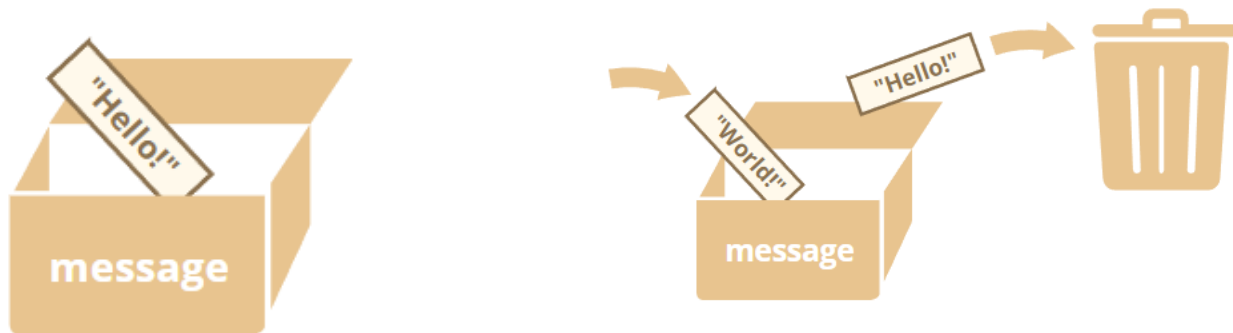
```
1 var mensaje = 'Hola';
```

La palabra clave `var` es *casi* lo mismo que `let`. También hace la declaración de una variable, aunque de un modo ligeramente distinto, y más antiguo.

## **Una analogía de la vida real**

Podemos comprender fácilmente el concepto de una “variable” si nos la imaginamos como una “caja” con una etiqueta de nombre único pegada en ella.

Por ejemplo, podemos imaginar la variable `message` como una caja etiquetada “`message`” con el valor “`Hola!`” adentro:



También podemos declarar dos variables y copiar datos de una a la otra.

Podemos introducir cualquier valor a la caja.

### **Nombramiento de variables**

Existen dos limitaciones de nombre de variables en JavaScript:

1. El nombre únicamente puede incluir letras, dígitos, o los símbolos `$` y `_`.
2. El primer carácter no puede ser un dígito.

Ejemplos de nombres válidos:

```
1 let userName;  
2 let test123;
```

Cuando el nombre contiene varias palabras, comúnmente se utiliza [camelCase](#). Es decir: palabras van una detrás de otra, con cada palabra iniciando con letra mayúscula: `miNombreMuyLargo`.

Es interesante notar – el símbolo del dólar `'$'` y el guión bajo `'_'` también se utilizan en nombres. Son símbolos comunes, tal como las letras, sin ningún significado especial.

## La Capitalización es Importante

Variables con el nombre `manzana` y `manzana` son distintas.

## Letras que no son del alfabeto inglés están permitidas, pero no se recomiendan

Es posible utilizar letras de cualquier alfabeto, incluyendo el cirílico e incluso jeroglíficos, por ejemplo:

```
1 let имя = '...';  
2 let 我 = '...';
```

### Nombres reservados

Hay una [lista de palabras reservadas](#), las cuales no pueden ser utilizadas como nombre de variable porque el lenguaje en sí las utiliza.

Por ejemplo: `let`, `class`, `return`, y `function` están reservadas.


## Constantes

Para declarar una variable constante (inmutable) use `const` en vez de `let`:

```
1 const fechaNacimiento = "27-03-1981";  
2 console.log(fechaNacimiento);
```

Las variables declaradas utilizando `const` se llaman "constantes". No pueden ser alteradas. Al intentarlo causaría un error:

```
1 const fechaNacimiento = "27-03-1981";  
2 fechaNacimiento = "04-07-1992"  
3 console.log(fechaNacimiento);
```

 Uncaught TypeError: Assignment to constant variable.

## Constantes mayúsculas

Existe una práctica utilizada ampliamente de utilizar constantes como alias de valores difíciles-de-recordar y que se conocen previo a la ejecución.

Tales constantes se nombran utilizando letras mayúsculas y guiones bajos.

Por ejemplo, creemos constantes para los colores en el formato "web" (hexadecimal):

```
1  const COLOR_RED = "#F00";
2  const COLOR_GREEN = "#0F0";
3  const COLOR_BLUE = "#00F";
4  const COLOR_ORANGE = "#FF7F00";
5
6  // ...cuando debemos elegir un color
7  let color = COLOR_ORANGE;
8  alert(color); // #FF7F00
```

## Nombrar cosas correctamente

Estando en el tema de las variables, existe una cosa de mucha importancia.

Una variable debe tener un nombre claro, de significado evidente, que describa el dato que almacena.

Nombrar variables es una de las habilidades más importantes y complejas en la programación. Un vistazo rápido a el nombre de las variables nos revela cuál código fue escrito por un principiante o por un desarrollador experimentado.

En un proyecto real, la mayor parte de el tiempo se pasa modificando y extendiendo una base de código en vez de empezar a escribir algo desde cero. Cuando regresamos a algún código después de hacer algo distinto por un rato, es mucho más fácil encontrar información que está bien etiquetada. O, en otras palabras, cuando las variables tienen nombres adecuados.

Por favor pasa tiempo pensando en el nombre adecuado para una variable antes de declararla. Hacer esto te da un retorno muy sustancial.

Algunas reglas buenas para seguir:

- Use términos legibles para humanos como `userName` p `shoppingCart`.
- Evite abreviaciones o nombres cortos `a`, `b`, `c`, al menos que en serio sepa lo que está haciendo.
- Cree nombres que describen al máximo lo que son y sean concisos. Ejemplos que no son adecuados son `data` y `value`. Estos nombres no nos dicen nada. Estos solo está bien usarlos en el contexto de un código que deje excepcionalmente obvio cuál valor o cuales datos está referenciando la variable.
- Acuerda en tu propia mente y con tu equipo cuáles términos se utilizarán. Si a un visitante se le llamara "user", debemos llamar las variables relacionadas `currentUser` o `newUser` en vez de `currentVisitor` o `newManInTown`.

# Tipos de datos

Un valor en JavaScript siempre pertenece a un tipo de dato determinado. Por ejemplo, un string o un número.

Hay ocho tipos de datos básicos en JavaScript. En este capítulo los cubriremos en general y en los próximos hablaremos de cada uno de ellos en detalle.

Podemos almacenar un valor de cualquier tipo dentro de una variable. Por ejemplo, una variable puede contener en un momento un string y luego almacenar un número:

```
1 // no hay error
2 let message = "hola";
3 message = 123456;
```

Los lenguajes de programación que permiten estas cosas, como JavaScript, se denominan “dinámicamente tipados”, lo que significa que allí hay tipos de datos, pero las variables no están vinculadas rígidamente a ninguno de ellos.

## Number

```
1 let n = 123;
2 n = 12.345;
```

El tipo *number* representa tanto números enteros como de punto flotante.

Hay muchas operaciones para números. Por ejemplo, multiplicación `*`, división `/`, suma `+`, resta `-`, y demás.

Además de los números comunes, existen los llamados “valores numéricos especiales” que también pertenecen a este tipo de datos: `Infinity`, `-Infinity` y `NaN`.

- `Infinity` representa el **Infinito** matemático  $\infty$ . Es un valor especial que es mayor que cualquier número.

Podemos obtenerlo como resultado de la división por cero:

```
1 alert( 1 / 0 ); // Infinity
```





☞ simplemente hacer referencia a él directamente:

```
1 alert( Infinity ); // Infinity
```

- `NaN` representa un error de cálculo. Es el resultado de una operación matemática incorrecta o indefinida, por ejemplo:

```
1 alert( "no es un número" / 2 ); // NaN, tal división es errónea
```

`NaN` es "pegajoso". Cualquier otra operación sobre `NaN` devuelve `NaN`:

```
1 alert( NaN + 1 ); // NaN
2 alert( 3 * NaN ); // NaN
3 alert( "not a number" / 2 - 1 ); // NaN
```

## BigInt

En JavaScript, el tipo "number" no puede representar valores enteros mayores que  $(2^{53}-1)$  (eso es `9007199254740991`), o menor que  $-(2^{53}-1)$  para negativos. Es una limitación técnica causada por su representación interna.

Para la mayoría de los propósitos es suficiente, pero a veces necesitamos números realmente grandes. Por ejemplo, para criptografía o marcas de tiempo de precisión de microsegundos.

`BigInt` se agregó recientemente al lenguaje para representar enteros de longitud arbitraria.

Un valor `BigInt` se crea agregando `n` al final de un entero:

```
1 // la "n" al final significa que es un BigInt
2 const enteroLargo = 451465464646474178789971184n;
```

Como los números `BigInt` rara vez se necesitan, no los cubrimos aquí sino que les dedicamos un capítulo separado <info: bigint>. Léelo cuando necesites números tan grandes.

# String

Un *string* en JavaScript es una cadena de caracteres y debe colocarse entre comillas.

```
1 let str = "Hola, Bienvenidos";
2 let str2 = 'las comillas simples se pueden';
3 let frase = `aqui vamos a encrustar el texto: ${str}`;
4 console.log(frase);
```

En JavaScript, hay 3 tipos de comillas.

1. Comillas dobles: "Hola".
2. Comillas simples: 'Hola'.
3. Backticks (comillas invertidas): `Hola`.

Las comillas dobles y simples son comillas "sencillas" (es decir, funcionan igual). No hay diferencia entre ellas en JavaScript.

Los backticks son comillas de "funcionalidad extendida". Nos permiten incrustar variables y expresiones en una cadena de caracteres encerrándolas en `${...}`, por ejemplo:

## String

Un *string* en JavaScript es una cadena de caracteres y debe colocarse entre comillas.

```
1 let str = "Hola, Bienvenidos";
2 let str2 = 'las comillas simples se pueden';
3 let frase = `aqui vamos a encrustar el texto: ${str}`;
4 console.log(frase);
```

En JavaScript, hay 3 tipos de comillas.

1. Comillas dobles: "Hola".
2. Comillas simples: 'Hola'.
3. Backticks (comillas invertidas): `Hola`.

Las comillas dobles y simples son comillas "sencillas" (es decir, funcionan igual). No hay diferencia entre ellas en JavaScript.

Los backticks son comillas de "funcionalidad extendida". Nos permiten incrustar variables y expresiones en una cadena de caracteres encerrándolas en `${...}`, por ejemplo:

```
1 let name = "Arle";
2
3 // incrustar una variable
4 console.log(`Hola, ${name}`);
5 // incrustar una expresión
6 console.log(`el resultado de esta operación es ${3 + 7}`);
7
```

La expresión dentro de `${...}` se evalúa y el resultado pasa a formar parte de la cadena. Podemos poner cualquier cosa ahí dentro: una variable como `name`, una expresión aritmética como `1 + 2`, o algo más complejo.

Toma en cuenta que esto sólo se puede hacer con los backticks. ¡Las otras comillas no tienen esta capacidad de incrustación!

```
1 // el resultado es 9
2 console.log(`el resultado de esta operación es ${3 + 7} - 1`);
3
```

## Boolean (tipo lógico)

El tipo *boolean* tiene sólo dos valores posibles: `true` y `false`.

Este tipo se utiliza comúnmente para almacenar valores de sí/no: `true` significa "sí, correcto, verdadero", y `false` significa "no, incorrecto, falso".

Por ejemplo:

```
1 let campoNombreMarcado = true;
2 let campoEdadMarcado = false;
3 console.log(campoNombreMarcado);
```

Los valores booleanos también son el resultado de comparaciones:

```
1 let campoNombreMarcado = true;
2 let campoEdadMarcado = false;
3 console.log(campoNombreMarcado);
```

## El valor “null” (nulo).

El valor especial `null` no pertenece a ninguno de los tipos descritos anteriormente.

Forma un tipo propio separado que contiene sólo el valor `null`:

```
1 let age = null;
2 let objeto = null;
3
4 console.log(age);
```

En JavaScript, `null` no es una “referencia a un objeto inexistente” o un “puntero nulo” como en otros lenguajes.

Es sólo un valor especial que representa “nada”, “vacío” o “valor desconocido”.

El código anterior indica que el valor de `age` es desconocido o está vacío por alguna razón.

## El valor “undefined” (indefinido)

El valor especial `undefined` también se distingue. Hace un tipo propio, igual que `null`.

El significado de `undefined` es “valor no asignado”.

Si una variable es declarada pero no asignada, entonces su valor es `undefined`:

```
1 let name;
2 let objeto = null;
3
4 console.log(name);
```

Técnicamente, es posible asignar `undefined` a cualquier variable:

```
1 let age = 27;
2
3 age = undefined;
4
5 console.log(age);
```

# Object y Symbol

El tipo `object` (objeto) es especial.

Todos los demás tipos se llaman "primitivos" porque sus valores pueden contener una sola cosa (ya sea una cadena, un número o lo que sea). Por el contrario, los objetos se utilizan para almacenar colecciones de datos y entidades más complejas.

Siendo así de importantes, los objetos merecen un trato especial. Nos ocuparemos de ellos más adelante en el capítulo [Objetos](#) después de aprender más sobre los primitivos.

El tipo `symbol` (símbolo) se utiliza para crear identificadores únicos para los objetos. Tenemos que mencionarlo aquí para una mayor integridad, pero es mejor estudiar este tipo después de los objetos.

## El operador typeof

El operador `typeof` devuelve el tipo del argumento. Es útil cuando queremos procesar valores de diferentes tipos de forma diferente o simplemente queremos hacer una comprobación rápida.

Soporta dos formas de sintaxis:

1. Como operador: `typeof x`.
2. Como una función: `typeof(x)`.

En otras palabras, funciona con paréntesis o sin ellos. El resultado es el mismo.

La llamada a `typeof x` devuelve una cadena con el nombre del tipo:

```
1 let a = typeof undefined; // indefinido
2
3 let b = typeof 0; //number
4
5 let c = typeof 10n; //bigint
6
7 let d = typeof true; // boolean
8
9 console.log(d);
```

# Interacción: alert, prompt, confirm

Como usaremos el navegador como nuestro entorno de demostración, veamos un par de funciones para interactuar con el usuario: `alert`, `prompt` y `confirm`.

## alert

Este ya lo hemos visto. Muestra un mensaje y espera a que el usuario presione "Aceptar".

Por ejemplo:

```
1 alert("Hello");
```

La mini ventana con el mensaje se llama \* ventana modal \*. La palabra "modal" significa que el visitante no puede interactuar con el resto de la página, presionar otros botones, etc., hasta que se haya ocupado de la ventana. En este caso, hasta que presionen "OK".

El usuario puede escribir algo en el campo de entrada de solicitud y presionar OK. Así obtenemos ese texto en `result`. O puede cancelar la entrada presionando Cancelar o presionando la tecla `Esc` obteniendo `null` en `result`.

La llamada a `prompt` retorna el texto del campo de entrada o `null` si la entrada fue cancelada.

Por ejemplo:

```
1 <!DOCTYPE html>
2 <html lang="en">
3 <head>
4     <meta charset="UTF-8">
5     <title>Ejemplos</title>
6 </head>
7 <body>
8 <script>
9     let age = prompt("¿cuántos años tiene?", "18");
10    console.log(`usted tiene ${age} años!`);
11 </script>
12 </body>
13 </html>
```



## confirm

La sintaxis:

```
1 result = confirm(pregunta);
```

La función `confirm` muestra una ventana modal con una `pregunta` y dos botones: OK y CANCELAR.

El resultado es `true` si se pulsa OK y `false` en caso contrario.

Por ejemplo:

```
<script>
  let isBoss = confirm("¿Es usted un jefe?");
  alert(isBoss);
</script>
```

## Conversiones de Tipos

La mayoría de las veces, los operadores y funciones convierten automáticamente los valores que se les pasan al tipo correcto. Esto es llamado "conversión de tipo".

Por ejemplo, `alert` convierte automáticamente cualquier valor a string para mostrarlo. Las operaciones matemáticas convierten los valores a números.

También hay casos donde necesitamos convertir de manera explícita un valor al tipo esperado.

### ToString

La conversión a string ocurre cuando necesitamos la representación en forma de texto de un valor.

Por ejemplo, `alert(value)` lo hace para mostrar el valor como texto.

También podemos llamar a la función `String(value)` para convertir un valor a string:

```
1 let value = true;
2 alert(typeof value); // boolean
3
4 value = String(value); // ahora value es el string "true"
5 alert(typeof value); // string
```

La conversión a string es bastante obvia. El boolean `false` se convierte en `"false"`, `null` en `"null"`, etc.

# ToNumber

La conversión numérica ocurre automáticamente en funciones matemáticas y expresiones.

Por ejemplo, cuando se dividen valores no numéricos usando `/`:

```
1 alert( "6" / "2" ); // 3, los strings son convertidos a números
```

Podemos usar la función `Number(value)` para convertir de forma explícita un valor a un número:

```
1 let str = "123";
2 alert(typeof str); // string
3
4 let num = Number(str); // se convierte en 123
5
6 alert(typeof num); // number
```

La conversión explícita es requerida usualmente cuando leemos un valor desde una fuente basada en texto, como lo son los campos de texto en los formularios, pero que esperamos que contengan un valor numérico.

Si el string no es un número válido, el resultado de la conversión será `NaN`. Por ejemplo:

```
1 let age = Number("un texto arbitrario en vez de un número");
2
3 alert(age); // NaN, conversión fallida
```

Reglas de conversión numérica:

Valor	Se convierte en...
undefined	NaN
null	0
true and false	1 y 0
string	Se eliminan los espacios al inicio y final del texto. Si el string resultante es vacío, el resultado es 0, en caso contrario el número es "leído" del string. Un error devuelve NaN.

```
1 alert( Number(" 123 ") ); // 123
2 alert( Number("123z") ); // NaN (error al leer un número en
3 alert( Number(true) ); // 1
4 alert( Number(false) ); // 0
```

Ten en cuenta que `null` y `undefined` se comportan de distinta manera aquí: `null` se convierte en `0` mientras que `undefined` se convierte en `NaN`.

### **i** Adición '+' concatena strings

Casi todas las operaciones matemáticas convierten valores a números. Una excepción notable es la suma `+`. Si uno de los valores sumados es un string, el otro valor es convertido a string.

Luego, los concatena (une):

```
1 alert( 1 + '2' ); // '12' (string a la derecha)
2 alert( '1' + 2 ); // '12' (string a la izquierda)
```

## ToBoolean

La conversión a boolean es la más simple.

Ocurre en operaciones lógicas (más adelante veremos test condicionales y otras cosas similares) pero también puede realizarse de forma explícita llamando a la función `Boolean(value)`.

Las reglas de conversión:

- Los valores que son intuitivamente "vacíos", como `0`, `""`, `null`, `undefined`, y `NaN`, se convierten en `false`.
- Otros valores se convierten en `true`.

Por ejemplo:

```
1 alert( Boolean(1) ); // true
2 alert( Boolean(0) ); // false
3
4 alert( Boolean("hola") ); // true
5 alert( Boolean("") ); // false
```

# Operadores básicos, matemáticas

Conocemos varios operadores matemáticos porque nos los enseñaron en la escuela. Son cosas como la suma `+`, multiplicación `*`, resta `-`, etcétera.

En este capítulo, nos vamos a concentrar en los aspectos de los operadores que no están cubiertos en la aritmética escolar.

## Términos: “unario”, “binario”, “operando”

Antes de continuar, comprendamos la terminología común.

- *Un operando* – es a lo que se aplican los operadores. Por ejemplo, en la multiplicación de `5 * 2` hay dos operandos: el operando izquierdo es `5` y el operando derecho es `2`. A veces, la gente los llama “argumentos” en lugar de “operandos”.
- Un operador es *unario* si tiene un solo operando. Por ejemplo, la negación unaria `-` invierte el signo de un número:

```
1 let x = 1;
2
3 x = -x;
4 alert( x ); // -1, se aplicó negación unaria
```

- Un operador es *binario* si tiene dos operandos. El mismo negativo también existe en forma binaria:

```
1 let x = 1, y = 3;
2 alert( y - x ); // 2, binario negativo resta valores
```

Formalmente, estamos hablando de dos operadores distintos: la negación unaria (un operando: revierte el símbolo) y la resta binaria (dos operandos: resta).

## Matemáticas

Están soportadas las siguientes operaciones:

- Suma `+`,
- Resta `-`,
- Multiplicación `*`,
- División `/`,
- Resto `%`,
- Exponenciación `**`.

## Resto %

El operador resto `%`, a pesar de su apariencia, no está relacionado con porcentajes.

El resultado de `a % b` es el **resto** de la división entera de `a` por `b`.

Por ejemplo:

```
1 alert( 5 % 2 ); // 1 es un resto de 5 dividido por 2
2 alert( 8 % 3 ); // 2 es un resto de 8 dividido por 3
```

## Exponenciación \*\*

El operador exponenciación `a ** b` eleva `a` a la potencia de `b`.

En matemáticas de la escuela, lo escribimos como  $a^b$ .

Por ejemplo:

```
1 alert( 2 ** 2 ); // 22 = 4
2 alert( 2 ** 3 ); // 23 = 8
3 alert( 2 ** 4 ); // 24 = 16
```

## Concatenación de cadenas con el binario +

Ahora veamos características especiales de JavaScript que van más allá de las aritméticas escolares.

Normalmente el operador `+` suma números.

Pero si se aplica el `+` binario a una cadena, los une (concatena):

```
1 let s = "my" + "string";
2 alert(s); // mystring
```

Tenga presente que si uno de los operandos es una cadena, el otro es convertido a una cadena también.

Por ejemplo:

```
1 alert( '1' + 2 ); // "12"
2 alert( 2 + '1' ); // "21"
```

## Conversión numérica, unario +

La suma `+` existe en dos formas: la forma binaria que utilizamos arriba y la forma unaria.

El unario suma o, en otras palabras, el operador suma `+` aplicado a un solo valor, no hace nada a los números. Pero si el operando no es un número, el unario suma lo convierte en un número.

Por ejemplo:

```
1 // Sin efecto en números
2 let x = 1;
3 alert( +x ); // 1
4
5 let y = -2;
6 alert( +y ); // -2
7
8 // Convierte los no números
9 alert( +true ); // 1
10 alert( +"" ); // 0
```

Realmente hace lo mismo que `Number(...)`, pero es más corto.

## Precedencia del operador

Si una expresión tiene más de un operador, el orden de ejecución se define por su *precedencia* o, en otras palabras, el orden de prioridad predeterminado de los operadores.

Desde la escuela, todos sabemos que la multiplicación en la expresión `1 + 2 * 2` debe calcularse antes de la suma. Eso es exactamente la precedencia. Se dice que la multiplicación tiene *una mayor precedencia* que la suma.

Los paréntesis anulan cualquier precedencia, por lo que si no estamos satisfechos con el orden predeterminado, podemos usarlos para cambiarlo. Por ejemplo, escriba `(1 + 2) * 2`.

Hay muchos operadores en JavaScript. Cada operador tiene un número de precedencia correspondiente. El que tiene el número más grande se ejecuta primero. Si la precedencia es la misma, el orden de ejecución es de izquierda a derecha.

Aquí hay un extracto de la [tabla de precedencia](#) (no necesita recordar esto, pero tenga en cuenta que los operadores unarios son más altos que el operador binario correspondiente):



Precedencia	Nombre	Signo
...	...	...
16	suma unaria	+
16	negación unaria	-
16	exponenciación	**
14	multiplicación	*
14	división	/
13	suma	+
13	resta	-
...	...	...
3	asignación	=
...	...	...

Como podemos ver, la “suma unaria” tiene una prioridad de 16, que es mayor que el 13 de “suma” (suma binaria). Es por eso que, en la expresión `+apples + +oranges`, las sumas unarias funcionan antes de la adición.

## Asignación

Tengamos en cuenta que una asignación `=` también es un operador. Está listado en la tabla de precedencia con la prioridad muy baja de 3.

Es por eso que, cuando asignamos una variable, como `x = 2 * 2 + 1`, los cálculos se realizan primero y luego se evalúa el `=`, almacenando el resultado en `x`.

```
1 let x = 2 * 2 + 1;
2
3 alert( x ); // 5
```

### Asignación = devuelve un valor

El hecho de que `=` sea un operador, no una construcción “mágica” del lenguaje, tiene un implicación interesante.

Todos los operadores en JavaScript devuelven un valor. Esto es obvio para `+` y `-`, pero también es cierto para `=`.

La llamada `x = value` escribe el `value` en `x` y luego lo devuelve.

## Asignaciones encadenadas

Otra característica interesante es la habilidad para encadenar asignaciones:

```
1 let a, b, c;  
2  
3 a = b = c = 2 + 2;  
4  
5 alert( a ); // 4  
6 alert( b ); // 4  
7 alert( c ); // 4
```

Las asignaciones encadenadas evalúan de derecha a izquierda. Primero, se evalúa la expresión más a la derecha `2 + 2` y luego se asigna a las variables de la izquierda: `c`, `b` y `a`. Al final, todas las variables comparten un solo valor.

Una vez más, con el propósito de la legibilidad es mejor separa tal código en unas pocas líneas:

```
1 c = 2 + 2;  
2 b = c;  
3 a = c;
```

## Incremento/decremento

Aumentar o disminuir un número en uno es una de las operaciones numéricas más comunes.

Entonces, hay operadores especiales para ello:

- **Incremento** `++` incrementa una variable por 1:

```
1 let counter = 2;  
2 counter++; // funciona igual que counter = counter + 1, pe  
3 alert( counter ); // 3
```

- **Decremento** `--` decrementa una variable por 1:

```
1 let counter = 2;  
2 counter--; // funciona igual que counter = counter - 1, pe  
3 alert( counter ); // 1
```

Aclaremos. Tal como conocemos, todos los operadores devuelven un valor. Incremento/decremento no es una excepción. La forma prefijo devuelve el nuevo valor mientras que la forma sufijo devuelve el valor anterior (antes del incremento/decremento).

Para ver la diferencia, aquí hay un ejemplo:

```
1 let counter = 1;
2 let a = ++counter; // (*)
3
4 alert(a); // 2
```

En la línea `(*)`, la forma *prefijo* `++counter` incrementa `counter` y devuelve el nuevo valor, `2`. Por lo tanto, el `alert` muestra `2`.

Ahora usemos la forma sufijo:

```
1 let counter = 1;
2 let a = counter++; // (*) cambiado ++counter a counter++
3
4 alert(a); // 1
```

### **i** Incremento/decremento entre otros operadores

Los operadores `++/--` también pueden ser usados dentro de expresiones. Su precedencia es más alta que la mayoría de los otros operadores aritméticos.

Por ejemplo:

```
1 let counter = 1;
2 alert( 2 * ++counter ); // 4
```

Compara con:

```
1 let counter = 1;
2 alert( 2 * counter++ ); // 2, porque counter++ devuelve el
```

Aunque técnicamente está bien, tal notación generalmente hace que el código sea menos legible. Una línea hace varias cosas, no es bueno.

Mientras lees el código, un rápido escaneo ocular "vertical" puede pasar por alto fácilmente algo como `'counter++'` y no será obvio que la variable aumentó.

# Operadores a nivel de bit

Los operadores a nivel bit tratan los argumentos como números enteros de 32 bits y trabajan en el nivel de su representación binaria.

Estos operadores no son específicos de JavaScript. Son compatibles con la mayoría de los lenguajes de programación.

La lista de operadores:

- AND ( `&` )
- OR ( `|` )
- XOR ( `^` )
- NOT ( `~` )

## Coma

El operador coma `,` es uno de los operadores más raros e inusuales. A veces, es utilizado para escribir código más corto, entonces tenemos que saberlo para poder entender qué está pasando.

El operador coma nos permite evaluar varias expresiones, dividiéndolas con una coma `,`. Cada una de ellas es evaluada pero sólo el resultado de la última es devuelto.

Por ejemplo:

```
1 let a = (1 + 2, 3 + 4);
2
3 alert( a ); // 7 (el resultado de 3 + 4)
```

Aquí, se evalúa la primera expresión `1 + 2` y se desecha su resultado. Luego, se evalúa `3 + 4` y se devuelve como resultado.

# Comparaciones

Conocemos muchos operadores de comparación de las matemáticas:

En Javascript se escriben así:

- Mayor/menor que: `a > b`, `a < b`.
- Mayor/menor o igual que: `a >= b`, `a <= b`.
- Igual: `a == b` (ten en cuenta el doble signo `==`. Un solo símbolo `a = b` significaría una asignación).
- Distinto. En matemáticas la notación es  $\neq$ , pero en JavaScript se escribe como una asignación con un signo de exclamación delante: `a != b`.

En este artículo, aprenderemos más sobre los diferentes tipos de comparaciones y de cómo las realiza JavaScript, incluidas las peculiaridades importantes.

Al final, encontrará una buena receta para evitar problemas relacionadas con las "peculiaridades" de JavaScript.

## Booleano es el resultado

Como todos los demás operadores, una comparación retorna un valor. En este caso, el valor es un booleano.

- `true` – significa "sí", "correcto" o "verdad".
- `false` – significa "no", "equivocado" o "no verdad".

Por ejemplo:

```
1 alert( 2 > 1 ); // true (correcto)
2 alert( 2 == 1 ); // false (incorrecto)
3 alert( 2 != 1 ); // true (correcto)
```

El resultado de una comparación puede asignarse a una variable, igual que cualquier valor:

```
1 let result = 5 > 4; // asignar el resultado de la comparación
2 alert( result ); // true
```

## Comparación de cadenas

Para ver si una cadena es "mayor" que otra, JavaScript utiliza el llamado orden "de diccionario" o "lexicográfico".

En otras palabras, las cadenas se comparan letra por letra.

Por ejemplo:

```
1 alert( 'Z' > 'A' ); // true
2 alert( 'Glow' > 'Glee' ); // true
3 alert( 'Bee' > 'Be' ); // true
```

El algoritmo para comparar dos cadenas es simple:

1. Compare el primer carácter de ambas cadenas.
2. Si el primer carácter de la primera cadena es mayor (o menor) que el de la otra cadena, entonces la primera cadena es mayor (o menor) que la segunda. Hemos terminado.
3. De lo contrario, si los primeros caracteres de ambas cadenas son los mismos, compare los segundos caracteres de la misma manera.
4. Repita hasta el final de cada cadena.
5. Si ambas cadenas tienen la misma longitud, entonces son iguales. De lo contrario, la cadena más larga es mayor.

## Igualdad estricta

Una comparación regular de igualdad `==` tiene un problema. No puede diferenciar `0` de `false`:

```
1 alert( 0 == false ); // true
```

Lo mismo sucede con una cadena vacía:

```
1 alert( '' == false ); // true
```

Esto sucede porque los operandos de diferentes tipos son convertidos a números por el operador de igualdad `==`. Una cadena vacía, al igual que `false`, se convierte en un cero.

¿Qué hacer si queremos diferenciar `0` de `false`?

**Un operador de igualdad estricto `===` comprueba la igualdad sin conversión de tipo.**

En otras palabras, si `a` y `b` son de diferentes tipos, entonces `a === b` retorna inmediatamente `false` sin intentar convertirlos.



## Comparación con nulos e indefinidos

Veamos más casos extremos.

Hay un comportamiento no intuitivo cuando se compara `null` o `undefined` con otros valores.

### Para un control de igualdad estricto `===`

Estos valores son diferentes, porque cada uno de ellos es de un tipo diferente.

```
1 alert( null === undefined ); // false
```

### Para una comparación no estricta `==`

Hay una regla especial. Estos dos son una "pareja dulce": son iguales entre sí (en el sentido de `==`), pero no a ningún otro valor.

```
1 alert( null == undefined ); // true
```

## Ejecución condicional: `if`, `'?'`

Algunas veces, necesitamos ejecutar diferentes acciones basadas en diferentes condiciones.

Para esto podemos usar la sentencia `if` y el operador condicional `?`, también llamado operador de "signo de interrogación".

### La sentencia `"if"`

La sentencia `if(...)` evalúa la condición en los paréntesis, y si el resultado es `true` ejecuta un bloque de código.

Por ejemplo:

```
1 let year = prompt('¿En que año fué publicada la especificación EC  
2  
3 if (year == 2015) alert( '¡Estás en lo cierto!' );
```

Aquí la condición es una simple igualdad ( `year == 2015` ), pero podría ser mucho mas complejo.

Si quisiéramos ejecutar más de una sentencia, debemos encerrar nuestro bloque de código entre llaves:

```
1  if (year == 2015) {  
2    alert( "¡Es Correcto!" );  
3    alert( "¡Eres muy inteligente!" );  
4  }
```

Recomendamos siempre encerrar nuestro bloque de código entre llaves `{}` siempre que se utilice la sentencia `if`, inclusive si solo se va a ejecutar una sola sentencia en este caso. Hacer eso mejora la legibilidad.

## Conversión Booleana

La sentencia `if (...)` evalúa la expresión dentro de sus paréntesis y convierte el resultado en booleano.

Recordemos las reglas de conversión del capítulo [Conversiones de Tipos](#):

- El número `0`, un string vacío `""`, `null`, `undefined`, y `NaN` se convierte en `false`. Por esto son llamados valores "falso".
- El resto de los valores se convierten en `true`, entonces los llamaremos valores "verdadero".

Entonces, el código bajo esta condición nunca se ejecutaría:

```
1  if (0) { // 0 es falso  
2    ...  
3  }
```

...y dentro de esta condición siempre se ejecutará:

```
1  if (1) { // 1 es verdadero  
2    ...  
3  }
```

## La cláusula “else”

La sentencia `if` quizás contenga un bloque “else” opcional. Este se ejecutará cuando la condición sea falsa.

Por ejemplo:

```
1 let year = prompt('¿En qué año fue publicada la especificación EC');
2
3 if (year == 2015) {
4   alert( '¡Lo adivinaste, correcto!' );
5 } else {
6   alert( '¿Cómo puedes estar tan equivocado?' ); // cualquier valor
7 }
```

## [Muchas condiciones: “else if”](#)

Algunas veces, queremos probar variantes de una condición. La cláusula `else if` nos permite hacer esto.

Por ejemplo:

```
1 let year = prompt('¿En qué año fue publicada la especificación EC');
2
3 if (year < 2015) {
4   alert( 'Muy poco...' );
5 } else if (year > 2015) {
6   alert( 'Muy Tarde' );
7 } else {
8   alert( '¡Exactamente!' );
9 }
```

En el código de arriba, JavaScript primero revisa si `year < 2015`. Si esto es falso, continúa a la siguiente condición `year > 2015`. Si esta también es falsa, mostrará la última `alert`.

## Operador ternario '?'

A veces necesitamos asignar una variable dependiendo de alguna condición.

Por ejemplo:

```
1 let accessAllowed;  
2 let age = prompt('¿Qué edad tienes?', '');  
3  
4 if (age > 18) {  
5     accessAllowed = true;  
6 } else {  
7     accessAllowed = false;  
8 }  
9  
10 alert(accessAllowed);
```

Entonces el operador "ternario" también llamado "signo de interrogación" nos permite ejecutar esto en una forma más corta y simple.

Entonces el operador "ternario" también llamado "signo de interrogación" nos permite ejecutar esto en una forma más corta y simple.

El operador está representado por un signo de interrogación de cierre `?`. A veces es llamado "ternario" porque el operador tiene tres operandos. Es el único operador de JavaScript que tiene esta cantidad de ellos.

La Sintaxis es:

```
1 let result = condition ? value1 : value2;
```

Se evalúa `condition`: si es verdadera entonces devuelve `value1`, de lo contrario `value2`.

Por ejemplo:

```
1 let accessAllowed = (age > 18) ? true : false;
```

Técnicamente, podemos omitir el paréntesis alrededor de `age > 18`. El operador de signo de interrogación tiene una precedencia baja, por lo que se ejecuta después de la comparación `>`.

## Múltiples '?'

Una secuencia de operadores de signos de interrogación `?` puede devolver un valor que depende de más de una condición.

Por ejemplo:

```
1 let age = prompt('¿edad?', 18);
2
3 let message = (age < 3) ? '¡Hola, bebé!' :
4   (age < 18) ? '¡Hola!' :
5   (age < 100) ? '¡Felicidades!' :
6   '¡Qué edad tan inusual!';
7
8 alert( message );
```

## Uso no tradicional de '?'

A veces el signo de interrogación cerrado `?` se utiliza para reemplazar `if`:

```
1 let company = prompt('¿Qué compañía creó JavaScript?', '');
2
3 (company == 'Netscape') ?
4   alert('¡Correcto!') : alert('Equivocado.');
```

Dependiendo de la condición `company == 'Netscape'`, se ejecutará la primera o la segunda expresión del operador `?` y se mostrará una alerta.

Aquí no asignamos el resultado de una variable. En vez de esto, ejecutamos diferentes códigos dependiendo de la condición.

**No se recomienda el uso del operador de signo de interrogación de esta forma.**

La notación es más corta que la sentencia equivalente con `if`, lo cual seduce a algunos programadores. Pero es menos legible.

Aquí está el mismo código utilizando la sentencia `if` para comparar:

# Operadores Lógicos

Hay cuatro operadores lógicos en JavaScript: `||` (O), `&&` (Y), `!` (NO), `??` (Fusión de nulos). Aquí cubrimos los primeros tres, el operador `??` se verá en el siguiente artículo.

Aunque sean llamados lógicos, pueden ser aplicados a valores de cualquier tipo, no solo booleanos. El resultado también puede ser de cualquier tipo.

Veamos los detalles.

## || (OR)

El operador `OR` se representa con dos símbolos de línea vertical:

```
1 result = a || b;
```

En la programación clásica, el OR lógico está pensado para manipular solo valores booleanos. Si cualquiera de sus argumentos es `true`, retorna `true`, de lo contrario retorna `false`.

Hay cuatro combinaciones lógicas posibles:

```
1 alert(true || true); // true (verdadero)
2 alert(false || true); // true
3 alert(true || false); // true
4 alert(false || false); // false (falso)
```



Como podemos ver, el resultado es siempre `true` excepto cuando ambos operandos son `false`.

Si un operando no es un booleano, se lo convierte a booleano para la evaluación.

Por ejemplo, el número `1` es tratado como `true`, el número `0` como `false`:

```
1 if (1 || 0) { // Funciona como if( true || false )
2   alert("valor verdadero!");
3 }
```



Por ejemplo:

```
1 let hour = 9;
2
3 if (hour < 10 || hour > 18) {
4   alert( 'La oficina esta cerrada.' );
5 }
```

Podemos pasar mas condiciones:

```
1 let hour = 12;
2 let isWeekend = true;
3
4 if (hour < 10 || hour > 18 || isWeekend) {
5   alert("La oficina esta cerrada."); // Es fin de semana
6 }
```

## OR "||" encuentra el primer valor verdadero

La lógica descrita arriba es algo clásica. Ahora, mostremos las características "extra" de JavaScript.

El algoritmo extendido trabaja de la siguiente forma.

Dado múltiples valores aplicados al operador OR:

```
1 result = value1 || value2 || value3;
```

El operador OR `||` realiza lo siguiente:

- Evalúa los operandos de izquierda a derecha.
- Para cada operando, convierte el valor a booleano. Si el resultado es `true`, se detiene y retorna el valor original de ese operando.
- Si todos los operandos han sido evaluados (todos eran `false`), retorna el ultimo operando.

Un valor es retornado en su forma original, sin la conversión.



# && (AND)

El operador AND es representado con dos ampersands `&&`:

```
1 result = a && b;
```

En la programación clásica, AND retorna `true` si ambos operandos son valores verdaderos y `false` en cualquier otro caso.

```
1 alert(true && true); // true
2 alert(false && true); // false
3 alert(true && false); // false
4 alert(false && false); // false
```

## **i** La precedencia de AND `&&` es mayor que la de OR `||`

La precedencia del operador AND `&&` es mayor que la de OR `||`.

Así que el código `a && b || c && d` es básicamente el mismo que si la expresiones `&&` estuvieran entre paréntesis: `(a && b) || (c && d)`

Un ejemplo con `if`:

```
1 let hour = 12;
2 let minute = 30;
3
4 if (hour == 12 && minute == 30) {
5   alert("La hora es 12:30");
6 }
```

Al igual que con OR, cualquier valor es permitido como operando de AND:

```
1 if (1 && 0) { // evaluado como true && false
2   alert("no funcionará porque el resultado es un valor falso");
3 }
```

# ! (NOT)

El operador booleano NOT se representa con un signo de exclamación `!`.

La sintaxis es bastante simple:

```
1 result = !value;
```

El operador acepta un solo argumento y realiza lo siguiente:

1. Convierte el operando al tipo booleano: `true/false`.
2. Retorna el valor contrario.

Por ejemplo:

```
1 alert(!true); // false
2 alert(!0); // true
```

## Bucles: while y for

Usualmente necesitamos repetir acciones.

Por ejemplo, mostrar los elementos de una lista uno tras otro o simplemente ejecutar el mismo código para cada número del 1 al 10.

Los *Bucles* son una forma de repetir el mismo código varias veces.

### El bucle “while”

El bucle `while` (mientras) tiene la siguiente sintaxis:

```
1 while (condition) {
2   // código
3   // llamado "cuerpo del bucle"
4 }
```

Mientras la condición `condition` sea verdadera, el `código` del cuerpo del bucle será ejecutado.

Mientras la condición `condition` sea verdadera, el `código` del cuerpo del bucle será ejecutado.

Por ejemplo, el bucle debajo imprime `i` mientras se cumpla `i < 3`:

```
1 let i = 0;
2 while (i < 3) { // muestra 0, luego 1, luego 2
3   alert( i );
4   i++;
5 }
```

Cada ejecución del cuerpo del bucle se llama *iteración*. El bucle en el ejemplo de arriba realiza 3 iteraciones.

Si faltara `i++` en el ejemplo de arriba, el bucle sería repetido (en teoría) eternamente. En la práctica, el navegador tiene maneras de detener tales bucles desmedidos; y en el JavaScript del lado del servidor, podemos eliminar el proceso.

Cualquier expresión o variable puede usarse como condición del bucle, no solo las comparaciones: El `while` evaluará y transformará la condición a un booleano.

Por ejemplo, una manera más corta de escribir `while (i != 0)` es `while (i)`:

```
1 let i = 3;
2 while (i) { // cuando i sea 0, la condición se volverá falsa y el
3   alert( i );
4   i--;
5 }
```

### **i** Las llaves no son requeridas para un cuerpo de una sola línea

Si el cuerpo del bucle tiene una sola sentencia, podemos omitir las llaves `{...}`:

```
1 let i = 3;
2 while (i) alert(i--);
```

## El bucle “do...while”

La comprobación de la condición puede ser movida *debajo* del cuerpo del bucle usando la sintaxis `do..while`:

```
1 do {  
2   // cuerpo del bucle  
3 } while (condition);
```

El bucle primero ejecuta el cuerpo, luego comprueba la condición, y, mientras sea un valor verdadero, la ejecuta una y otra vez.

Por ejemplo:

```
1 let i = 0;  
2 do {  
3   alert( i );  
4   i++;  
5 } while (i < 3);
```



## El bucle “for”

El bucle `for` es más complejo, pero también el más usado.

Se ve así:

```
1 for (begin; condition; step) { // (comienzo, condición, paso)  
2   // ... cuerpo del bucle ...  
3 }
```

Aprendamos el significado de cada parte con un ejemplo. El bucle debajo corre `alert(i)` para `i` desde `0` hasta (pero no incluyendo) `3`:

```
1 for (let i = 0; i < 3; i++) { // muestra 0, luego 1, luego 2  
2   alert(i);  
3 }
```



## Rompiendo el bucle

Normalmente, se sale de un bucle cuando la condición se vuelve falsa.

Pero podemos forzar una salida en cualquier momento usando la directiva especial `break`.

Por ejemplo, el bucle debajo le pide al usuario por una serie de números, "rompiéndolo" cuando un número no es ingresado:

```
1 let sum = 0;
2
3 while (true) {
4
5   let value = +prompt("Ingresa un número", '');
6
7   if (!value) break; // (*)
8
9   sum += value;
10
11 }
12 alert( 'Suma: ' + sum );
```

## Continuar a la siguiente iteración

La directiva `continue` es una "versión más ligera" de `break`. No detiene el bucle completo. En su lugar, detiene la iteración actual y fuerza al bucle a comenzar una nueva (si la condición lo permite).

Podemos usarlo si hemos terminado con la iteración actual y nos gustaría movernos a la siguiente.

El bucle debajo usa `continue` para mostrar solo valores impares:

```
1 for (let i = 0; i < 10; i++) {
2
3   // si es verdadero, saltar el resto del cuerpo
4   if (i % 2 == 0) continue;
5
6   alert(i); // 1, luego 3, 5, 7, 9
7 }
```

Para los valores pares de `i`, la directiva `continue` deja de ejecutar el cuerpo y pasa el control a la siguiente iteración de `for` (con el siguiente número). Así que el `alert` solo es llamado para valores impares.

# Etiquetas para break/continue

```
1  <!DOCTYPE html>
2  <script>
3    "use strict";
4
5    outer: for (let i = 0; i < 3; i++) {
6
7      for (let j = 0; j < 3; j++) {
8
9        let input = prompt(`Value at coords (${i},${j})`, '');
10
11        // Si es una cadena de texto vacía o se canceló, entonces
12        // salir de ambos bucles
13        if (!input) break outer; // (*)
14
15        // hacer algo con el valor...
16      }
17    }
18    alert('Listo!');
19  </script>
```

## La sentencia "switch"

Una sentencia `switch` puede reemplazar múltiples condiciones `if`.

Provee una mejor manera de comparar un valor con múltiples variantes.

### La sintaxis

`switch` tiene uno o mas bloques `case` y un opcional `default`.

Se ve de esta forma:

```
1  switch(x) {
2    case 'valor1': // if (x === 'valor1')
3      ...
4      [break]
5
6    case 'valor2': // if (x === 'valor2')
7      ...
8      [break]
9
10   default:
11     ...
12     [break]
13 }
```

## Ejemplo

Un ejemplo de `switch` (se resalta el código ejecutado):

```
1  let a = 2 + 2;  
2  
3  switch (a) {  
4    case 3:  
5      alert( 'Muy pequeño' );  
6      break;  
7    case 4:  
8      alert( '¡Exacto!' );  
9      break;  
10   case 5:  
11     alert( 'Muy grande' );  
12     break;  
13   default:  
14     alert( "Desconozco estos valores" );  
15 }
```



