DWC

(Desarrollo Web en entorno cliente)

Índice

1.- Variables

La mayoría de las veces, una aplicación JavaScript necesita trabajar con información. Aquí hay dos ejemplos:

1. Una tienda online: la información puede incluir productos que se venden y un carrito de compras.
2. Una aplicación de chat: la información puede incluir usuarios, mensajes y mucho más.

Las variables se utilizan para almacenar esta información.

Una variable es un **"almacenamiento con nombre"** para los datos. Podemos usar variables para almacenar datos de nuestros scripts.

Para crear una variable en JavaScript, use la let palabra clave.

La siguiente declaración crea (en otras palabras: *declara*) una variable con el nombre "mensaje":

```
let message;
```

Ahora, podemos poner algunos datos en él utilizando el operador de asignación =:

```
let message;

message = 'Hello'; // store the string
```

La cadena ahora se guarda en el área de memoria asociada con la variable. Podemos acceder usando el nombre de la variable:

```
let message;
message = 'Hello!';

alert(message); // shows the variable content
```

Para ser conciso, podemos combinar la declaración de variables y la asignación en una sola línea:

```
let message= 'Hello!'; // define the variable and assign the value
```

```
alert(message); // Hello!
```

También podemos declarar múltiples variables en una línea:

```
let user = 'John', age = 25, message = 'Hello';
```

Eso puede parecer más corto, pero no se recomienda. En aras de una mejor legibilidad, utilizad una sola línea por variable.

La variante multilínea es un poco más larga, pero más fácil de leer:

```
let user = 'John';
let age = 25;
let message = 'Hello';
```

Algunas personas también definen múltiples variables en este estilo multilínea:

```
let user = 'John',
  age = 25,
  message = 'Hello';
```

O incluso en el estilo de "coma primero":

```
let user = 'John'
  , age = 25
  , message = 'Hello';
```

Técnicamente, todas estas variantes hacen lo mismo. Entonces, es una cuestión de gusto personal y estética.

var en vez de let

En scripts anteriores, también puede encontrar otra palabra clave: `var` en lugar de `let`:

```
var message = 'Hello';
```

La palabra clave `var` es *casi* la misma que `let`. También declara una variable, pero de una manera ligeramente diferente, "vieja escuela".

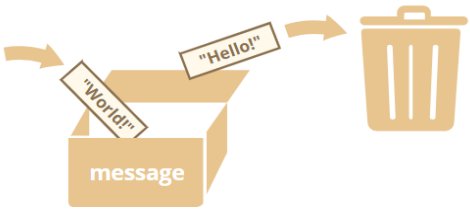
Hay diferencias sutiles entre `let` y `var`, pero todavía no nos importan.

Podemos comprender fácilmente el concepto de "variable" si lo imaginamos como una "caja" para datos, con una etiqueta con un nombre único.

Por ejemplo, la variable `message` se puede imaginar como una caja etiquetada `"message"` con el valor `"Hello!"` en ella:

Podemos poner cualquier valor en el cuadro.

También podemos cambiarlo tantas veces como queramos:




```
let message;
message = 'Hello!';
message = 'World!'; // value changed
alert(message);
```

Cuando se cambia el valor, los datos antiguos se eliminan de la variable:

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

```
let hello = 'Hello world!';
let message;

// copy 'Hello world' from hello into message
message = hello;

// now two variables hold the same data
alert(hello); // Hello world!
alert(message); // Hello world!
```

1.1.- Nombre de las variables

Hay dos limitaciones en los nombres de variables en JavaScript:

1. El nombre debe contener solo letras, dígitos o los símbolos $y _.
2. El primer carácter no debe ser un dígito.

Ejemplos de nombres válidos:

```
let userName;
let test123;
```

Cuando el nombre contiene varias palabras, se usa comúnmente **camelCase** . Es decir: las palabras van una tras otra, cada palabra excepto la primera empezara con una letra mayúscula: myVeryLongName.

Lo que es interesante: el signo de dólar '$'y el guión bajo '_'también se pueden usar en los nombres. Son símbolos regulares, al igual que las letras, sin ningún significado especial.

Estos nombres son válidos:

```
let $ = 1; // declared a variable with the name "$"
let _ = 2; // and now a variable with the name "_"

alert($ + _); // 3
```

Ejemplos de nombres de variables incorrectos:

```
let 1a; // cannot start with a digit

let my-name; // hyphens '-' aren't allowed in the name
```

El uso de mayúsculas y minúsculas importa

Las variables nombradas apple y AppLE son dos variables diferentes.

Se permiten letras no latinas, pero no se recomiendan.

Es posible utilizar cualquier idioma, incluidas letras cirílicas o incluso jeroglíficos, como este:

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

Técnicamente, no hay ningún error aquí, tales nombres están permitidos, pero existe una tradición internacional de usar el inglés en nombres variables. Incluso si estamos escribiendo un script pequeño, puede tener una larga vida por delante. Es posible que las personas de otros países necesiten leerlo alguna vez.

Nombres reservados

Hay una lista de palabras reservadas, que no se pueden usar como nombres de variables porque las usa el propio lenguaje.

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

El siguiente código da un error de sintaxis:

```
let let = 5; // can't name a variable "let", error!
let return = 5; // also can't name it "return", error!
```

Una tarea sin `use strict`

Normalmente, necesitamos definir una variable antes de usarla. Pero en los viejos tiempos, era técnicamente posible crear una variable por una simple asignación del valor sin usar `let`. Esto todavía funciona ahora si no incluimos `use strict` nuestros scripts para mantener la compatibilidad con los scripts antiguos.

```
// note: no "use strict" in this example
num = 5; // the variable "num" is created if it didn't exist
alert(num); // 5
```

Esta es una mala práctica y causaría un error en modo estricto:

```
"use strict";
num = 5; // error: num is not defined
```

2.- Constantes

Para declarar una variable constante (inmutable), use en const lugar de let:

```
const myBirthday = '18.04.1982';
```

Las variables declaradas usando constse llaman "constantes". No pueden ser reasignados. Un intento de hacerlo provocaría un error:

```
const myBirthday = '18.04.1982';
myBirthday = '01.01.2001'; // error, can't reassign the constant!
```

Cuando un programador está seguro de que una variable nunca cambiará, puede declararla const para garantizar y comunicar claramente ese hecho a todos.

2.1.- Constantes en mayúsculas

Existe una práctica generalizada de usar constantes como alias para valores difíciles de recordar que se conocen antes de la ejecución.

Dichas constantes se nombran con letras mayúsculas y guiones bajos.

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

```
const COLOR_RED = "#F00";
const COLOR_GREEN = "#0F0";
const COLOR_BLUE = "#00F";
const COLOR_ORANGE = "#FF7F00";
```

```
// ...when we need to pick a color
let color = COLOR_ORANGE;
alert(color); // #FF7F00
```

Beneficios:

- COLOR_ORANGE es mucho más fácil de recordar que "#FF7F00".
- Es mucho más fácil escribir mal "#FF7F00"que COLOR_ORANGE.
- Al leer el código, COLOR_ORANGE es mucho más significativo que #FF7F00.

¿Cuándo deberíamos usar mayúsculas para una constante y cuándo deberíamos nombrarla normalmente? Dejémoslo claro.

Ser una "constante" solo significa que el valor de una variable nunca cambia. Pero hay constantes que se conocen antes de la ejecución (como un valor hexadecimal para rojo) y hay constantes que se *calculan* en tiempo de ejecución, durante la ejecución, pero que no cambian después de su asignación inicial.

Por ejemplo:

```
const pageLoadTime = /* time taken by a webpage to load */;
```

El valor de `pageLoadTime`no se conoce antes de la carga de la página, por lo que se denomina normalmente. Pero sigue siendo una constante porque no cambia después de la asignación.

En otras palabras, las constantes con mayúscula solo se usan como alias para valores "codificados".

3.-Nombrar las cosas bien

Un nombre de variable debe tener un significado limpio y obvio, que describa los datos que almacena.

El nombre de una variable es una de las habilidades más importantes y complejas en la programación. Un vistazo rápido a los nombres de variables puede revelar qué código fue escrito por un principiante versus un desarrollador experimentado.

En un proyecto real, la mayor parte del tiempo se dedica a modificar y ampliar una base de código existente en lugar de escribir algo completamente separado desde cero. Cuando volvemos a algún código después de hacer algo más por un tiempo, es mucho más fácil encontrar información que esté bien etiquetada. O, en otras palabras, cuando las variables tienen buenos nombres.

Dedicad tiempo a pensar en el nombre correcto para una variable antes de declararla. Hacerlo os recompensará generosamente.

Algunas reglas que se deben seguir son:

- Utilice nombres legibles por humanos como `userName`o `shoppingCart`.

- Manténgase alejado de las abreviaturas o nombres cortos como `a`, `b`, `c`, a menos que realmente sepa lo que está haciendo.
- Haga que los nombres sean descriptivos y concisos al máximo. Ejemplos de malos nombres son `data`y `value`. Tales nombres no dicen nada. Está bien usarlos si el contexto del código hace que sea excepcionalmente obvio a qué datos o valor hace referencia la variable.
- Acordad los términos dentro de su equipo y en su propia mente. Si un visitante del sitio se llama "usuario", deberíamos nombrar variables relacionadas `currentUser`o en `newUser`lugar de `currentVisitor` o `newManInTown`.

¿Suena simple? De hecho lo es, pero crear nombres de variables descriptivos y concisos en la práctica no lo es.

Reutilizar o crear?

Y la última nota. Hay algunos programadores perezosos que, en lugar de declarar nuevas variables, tienden a reutilizar las existentes.

Como resultado, sus variables son como cajas en las que las personas arrojan cosas diferentes sin cambiar sus pegatinas. ¿Qué hay dentro de la caja ahora? ¿Quién sabe? Necesitamos acercarnos y verificar.

Tales programadores ahorran un poco en la declaración de variables pero pierden diez veces más en la depuración.

Una variable extra es buena, no mala.

Los minificadores y navegadores JavaScript modernos optimizan el código lo suficientemente bien, por lo que no creará problemas de rendimiento. El uso de diferentes variables para diferentes valores puede incluso ayudar al motor a optimizar su código.

4.- Resumen de variables y constantes

Podemos declarar variables para almacenar datos mediante el uso de los var, leto const palabras clave.

- let - es una declaración variable moderna.
- var- es una declaración variable de la vieja escuela. Normalmente no lo usamos
- const- es como let, pero el valor de la variable no se puede cambiar.

Las variables deben nombrarse de una manera que nos permita comprender fácilmente lo que hay dentro de ellas.

5.- Tipos de datos

Un valor en JavaScript siempre es de cierto tipo. Por ejemplo, una cadena o un número.

Hay ocho tipos de datos básicos en JavaScript.

Podemos poner cualquier tipo en una variable. Por ejemplo, una variable puede en un momento ser una cadena y luego almacenar un número:

```
// no error
let message = "hello";
message = 123456;
```

El tipo de la variable se adapta al contenido que tiene y puede modifcarse el tipo de la variable a lo largo del script en función del contenido que vaya teniendo

Los lenguajes de programación que permiten tales cosas, como JavaScript, se denominan "***tipados dinámicamente***", lo que significa que existen tipos de datos, pero las variables no están vinculadas a ninguno de ellos.

Este “problema” se soluciona con lenguajes por encima de JavaScript que dotan de tipado a las variables, como por ejemplo ***TypeScript***

5.1.- Number

```
let n = 123;
n = 12.345;
```

El tipo de *número* representa números enteros y de coma flotante.

Hay muchas operaciones para los números, por ejemplo, multiplicación *, división /, suma +, resta -, etc.

Además de los números regulares, no son los llamados “valores numéricos especiales”, que también pertenecen a este tipo de datos: Infinity, -Infinity y NaN.

- Infinity representa el infinito matemático ∞. Es un valor especial que es mayor que cualquier número.

  Podemos obtenerlo como resultado de la división por cero:
- alert(1 / 0); // Infinity
- O simplemente referencialo directamente:

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

- NaN representa un error computacional. Es el resultado de una operación matemática incorrecta o indefinida, por ejemplo:
- alert("not a number" / 2); // NaN, such division is erroneous
- NaN es “pegajoso” Cualquier otra operación en NaN devuelve NaN: alert("not a number" / 2 + 5); // NaN
- Entonces, si hay un NaN en una expresión matemática, se propaga al resultado completo.

5.2.- BigInt

En JavaScript, el tipo "número" no puede representar valores enteros mayores que 2^{53}-1, que es 9007199254740991 o menores que 2^{53}-1 para los 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 El tipo se agregó recientemente al lenguaje para representar enteros de longitud arbitraria.

Un BigIntvalor se crea agregando nal final de un entero:

```
// the "n" at the end means it's a BigInt
const bigInt = 1234567890123456789012345678901234567890n;
```

Los números BigInt rara vez se necesitan.

5.3.- String

Una cadena en JavaScript debe estar entre comillas.

```
let str = "Hello";
let str2 = 'Single quotes are ok too';
let phrase = `can embed another ${str}`;
```

En JavaScript, hay 3 tipos de entrecomillados.

1. Las comillas dobles: "Hello".
2. Las comillas simples: 'Hello'.

1. Acentos abiertos: `Hello`.

Las comillas dobles y simples son comillas "simples". Prácticamente no hay diferencia entre ellos en JavaScript.

Los backticks (acentos abiertos) son entrecomillados de "funcionalidad extendida". Nos permiten incrustar variables y expresiones en una cadena envolviéndolas ${…}, por ejemplo:

```
let name = "John";
// embed a variable
alert( `Hello, ${name}!` ); // Hello, John!
// embed an expression
alert( `the result is ${1 + 2}` ); // the result is 3
```

La expresión dentro ${…}se evalúa y el resultado se convierte en parte de la cadena. Podemos poner cualquier cosa allí: una variable como name o una expresión aritmética como 1 + 2 o algo más complejo.

Tenga en cuenta que esto solo se puede hacer en backticks. ¡Otros entrecomillados no tienen esta funcionalidad de incrustación!

```
alert( "the result is ${1 + 2}" ); // the result is ${1 + 2} (double quotes do nothing)
```

No hay tipo char .

En algunos lenguajes, hay un tipo especial de "string" para un solo carácter. Por ejemplo, en el lenguaje C y en Java se llama "char".

En JavaScript, no existe ese tipo. Sólo hay un tipo de: string. Una cadena puede constar de un solo carácter o muchos de ellos.

5.4.- Booleano (tipo lógico)

El tipo booleano tiene solo dos valores: truey false.

Este tipo se usa comúnmente para almacenar valores sí / no: true significa "sí, correcto" y false significa "no, incorrecto".

Por ejemplo:

```
let nameFieldChecked = true; // yes, name field is checked
let ageFieldChecked = false; // no, age field is not checked
```

Los valores booleanos también provienen de comparaciones:

```
let isGreater = 4 > 1;
alert( isGreater ); // true (the comparison result is "yes")
```

5.5.- El valor "nulo" null

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

Forma un tipo separado propio que contiene solo el valor null:

```
let age = null;
```

En JavaScript, nullno es una "referencia a un objeto no existente" o un "puntero nulo" como en otros idiomas.

Es solo un valor especial que representa "nada", "vacío" o "valor desconocido".

El código anterior indica que agese desconoce.

5.6.- El valor "indefinido" undefined

El valor especial undefined también se distingue. Es un tipo propio, al igual que null.

El significado de undefined es "el valor no está asignado".

Si se declara una variable, pero no se asigna, entonces su valor es undefined:

```
let age;
alert(age); // shows "undefined"
```

Técnicamente, es posible asignar explícitamente undefined a una variable:

```
let age = 100;

// change the value to undefined
age = undefined;

alert(age); // "undefined"
```

.

Pero no se recomienda hacer eso. Normalmente, uno usa null para asignar un valor "vacío" o "desconocido" a una variable, mientras que undefined está reservado como un valor inicial predeterminado para cosas no asignadas.

5.7.- Object y Symbol

El tipo object es especial.

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

Siendo tan importante, los objetos merecen un tratamiento especial. Los veremos más adelante, después de aprender más sobre las primitivas.

El tipo symbol aparece como nuevo tipo primitivo en ES6 disponemos de un nuevo tipo primitivo, los símbolos (symbol). Estos tipos de datos son utilizados como claves en objetos y mapas de datos sin que puedan ser convertidos a otros tipos, por lo que para poder acceder a estos elementos tendremos que tener una referencia al símbolo con el que se creó.

Cada vez que creamos un símbolo obtenemos un valor único, por lo que son de mucha utilidad a la hora de crear constantes con valores únicos y asegurarnos que no se va a producir confusiones por valores duplicados.

6.- El operador typeof

El operador typeof devuelve el tipo de argumento. Es útil cuando queremos procesar valores de diferentes tipos de manera diferente o simplemente queremos hacer una verificación rápida.

Es compatible con dos formas de sintaxis:

1. Como operador: typeof x.
2. En 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:

```
typeof undefined // "undefined"

typeof 0 // "number"

typeof 10n // "bigint"

typeof true // "boolean"

typeof "foo" // "string"

typeof Symbol("id") // "symbol"

typeof Math // "object"  (1)

typeof null // "object"  (2)

typeof alert // "function"  (3)
```

Las últimas tres líneas pueden necesitar una explicación adicional:

1. Math es un objeto incorporado que proporciona operaciones matemáticas. Lo veremos más adelante. Aquí, sirve solo como un ejemplo de un objeto.
2. El resultado de typeof null es "object". Ese es un error de comportamiento reconocido oficialmente , que proviene de los primeros días de JavaScript y se mantiene por compatibilidad. Definitivamente, null no es un objeto. Es un valor especial con un tipo separado propio.
3. El resultado de typeof alertes "function", porque alert es una función. Más adelante veremos funciones y veremos que no hay ningún tipo especial de "función" en JavaScript. Las funciones pertenecen al tipo de objeto. Pero los typeof trata de manera diferente, devolviendo "function". Eso también proviene de los primeros días de JavaScript.

7.- Resumen de tipos de datos

Hay 8 tipos de datos básicos en JavaScript.

- Number para números de cualquier tipo: entero o de coma flotante, los enteros están limitados por $\pm 2^{53}$.
- bigint es para números enteros de longitud arbitraria.
- String para cadenas de texto. Una cadena puede tener uno o más caracteres, no hay un tipo de un solo carácter como en otros lenguajes el tipo char.
- Boolean para true/ false.
- Null para valores desconocidos: un tipo independiente que tiene un solo valor null.
- Undefined para valores no asignados: un tipo independiente que tiene un solo valor undefined.
- Object para estructuras de datos más complejas.
- Symbol para identificadores únicos.

El operador typeof nos permite ver qué tipo está almacenado en una variable.

- Dos formas: typeof x o typeof(x).
- Devuelve una cadena con el nombre del tipo, como "string".
- Para null devuelve "object": este es un error en el lenguaje, en realidad no es un objeto.

8.-Interacción con el usuario

Hasta que veamos el DOM para poder probar nuestro código hasta ahora disponemos de la consola y la función console.log(mensaje), disponemos de otras alternativas para mostrar datos y para poder recibir datos por parte del usuario

Vamos a ver unas funciones para interactuar con el usuario: alert, prompty confirm.

8.1.- alert

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

Por ejemplo:

```
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".

8.2.- prompt

La función prompt acepta dos argumentos:

```
result = prompt(title, [default]);
```

Muestra una ventana modal con un mensaje de texto, un campo de entrada para el visitante y los botones Aceptar / Cancelar.

title

El texto para mostrar al visitante.

default

Un segundo parámetro opcional, el valor inicial para el campo de entrada.

El visitante puede escribir algo en el campo de entrada de solicitud y presionar OK. Así obtenemos ese texto en el result. O pueden cancelar la entrada

presionando Cancelar o presionando la tecla Esc, así obtenemos null como result.

La llamada a prompt devuelve el texto del campo de entrada o null si la entrada se canceló.

Por ejemplo:

```
let age = prompt('How old are you?', 100);

alert(`You are ${age} years old!`); // You are 100 years old!
```

8.3.- confirm

La sintaxis:

```
result = confirm(question);
```

La función confirm muestra una ventana modal con una question y dos botones: Aceptar y Cancelar.

El resultado es true si se presiona OK y de lo contrario false.

Por ejemplo:

```
let isBoss = confirm("Are you the boss?");

alert( isBoss ); // true if OK is pressed
```

9.- Resumen funciones interacción con usuario

Hemos visto 3 funciones específicas del navegador para interactuar con los visitantes:

alert

muestra un mensaje

prompt

muestra un mensaje pidiéndole al usuario que ingrese texto. Devuelve el texto o, si se hace clic en el botón Cancelar o Esc, null.

confirm

muestra un mensaje y espera a que el usuario presione "Aceptar" o "Cancelar". Devuelve true si presionan Aceptar y false si presionan Cancelar / Esc.

Todos estos métodos son modales: detienen la ejecución del script y no permiten que el visitante interactúe con el resto de la página hasta que se cierre la ventana.
Hay dos limitaciones compartidas por todos los métodos anteriores:

1. El navegador determina la ubicación exacta de la ventana modal. Por lo general, está en el centro.
2. El aspecto exacto de la ventana también depende del navegador. No podemos modificarlo.

Ese es el precio por la simplicidad. Hay otras formas de mostrar ventanas más agradables y una interacción más rica con el visitante.
P
ara interfaces de comunicación más avanzados y personalizados se utilizan los formularios de HTML en combinación del CSS para determinar las características visuales de la ventana

10.- Conversiones de tipo

La mayoría de las veces, los operadores y las funciones convierten automáticamente los valores que se les dan al tipo correcto.

Por ejemplo, alert convierte automáticamente cualquier valor en una cadena para mostrarlo. Las operaciones matemáticas convierten valores en números.

También hay casos en los que necesitamos convertir explícitamente un valor al tipo esperado.

10.1.- Conversión de cadenas

La conversión de cadenas ocurre cuando necesitamos la forma de cadena de un valor.

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

También podemos llamar a la función String(value)para convertir un valor en una cadena:

```
let value = true;
alert(typeof value); // boolean

value = String(value); // now value is a string "true"
alert(typeof value); // string
```

La conversión de cadenas es sobre todo obvia. El valor false se convierte "false", null se convierte "null", etc.

10.2.- Conversión numérica

La conversión numérica ocurre en funciones y expresiones matemáticas automáticamente.

Por ejemplo, cuando la división /se aplica a no números:

```
alert( "6" / "2" ); // 3, strings are converted to numbers
```

Podemos usar la función Number(value) para convertir explícitamente value a un número:

```
let str = "123";
alert(typeof str); // string
```

```
let num = Number(str); // becomes a number 123

alert(typeof num); // number
```

La conversión explícita generalmente se requiere cuando leemos un valor de una fuente basada en cadenas como un formulario de texto, pero esperamos que se introduzca un número.

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

```
let age = Number("an arbitrary string instead of a number");

alert(age); // NaN, conversion failed
```

Reglas numéricas de conversión:

Valor	Se convierte ...
`undefined`	`NaN`
`null`	`0`
`true` and `false`	`1` y `0`
`string`	Los espacios en blanco desde el principio y el final se eliminan. Si la cadena restante está vacía, el resultado es `0`. De lo contrario, el número se "lee" de la cadena. Un error da `NaN`.

Ejemplos:

```
alert( Number("   123   ") ); // 123
alert( Number("123z") );      // NaN (error reading a number at "z")
alert( Number(true) );        // 1
alert( Number(false) );       // 0
```

También disponemos de las funciones `parseInt(texto)` y `parseFloat(texto)` que convierten un número leído como texto en el número correspondiente en formato entero o decimal. Si el texto no se puede convertir a número devuelven `NaN`

10.3.- Conversión booleana

La conversión booleana es la más simple.

Ocurre en operaciones lógicas (luego veremos pruebas de condición y otras cosas similares) pero también se puede realizar explícitamente con una llamada a Boolean(value).

La regla de conversión:

- Los valores que son intuitivamente “vacío”, como 0, una cadena vacía, null, undefined, y NaN, se convierten a false.
- Los otros valores se convierten a true.

Por ejemplo:

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

alert( Boolean("hello") ); // true
alert( Boolean("") ); // false
```

La cadena con cero "0"es true

Algunos lenguajes como PHP se tratan "0" como false. Pero en JavaScript, una cadena no vacía siempre es true.

```
alert( Boolean("0") ); // true
alert( Boolean(" ") ); // spaces, also true (any non-empty string is true)
```

11.- Resumen conversión de tipos

Las tres conversiones de tipo más utilizadas son cadena, número y booleana.

- **String Conversion**- Se utiliza para obtener cadenas de texto.Se puede realizar con String(value). La conversión a cadena suele ser obvia para los valores primitivos.

- **Numeric Conversion**- Ocurre en operaciones matemáticas. Se puede realizar con Number(value).

  La conversión sigue las reglas:

Valor	Se convierte ...
undefined	NaN
null	0
true / false	1 / 0
string	La cadena se lee "tal cual", los espacios en blanco de ambos lados se ignoran. Se convierte una cadena vacía 0. Un error da NaN.

- **Boolean Conversion**- Ocurre en operaciones lógicas. Se puede realizar con Boolean(value).

  Sigue las reglas:

Valor	Se convierte ...
0` null` undefined` NaN`""	false
cualquier otro valor	true

La mayoría de estas reglas son fáciles de entender y memorizar. Las excepciones notables donde las personas suelen cometer errores son:

1. Undefined es NaN como un número, no 0.
2. "0"y cadenas de solo espacio como " " son True como un booleano.

- **Funciones de conversión de tipo númerico** disponemos de las funciones parseInt(texto) y parseFloat(texto) que convierten un número leído como texto en el número correspondiente en formato entero o decimal. Si el texto no se puede convertir a número devuelven NaN

12.- Los Operadores

Los operadores básicos son los elementos que nos permiten en una expresión poder realizar operaciones con los elementos que forman la expresión.

Términos: "unario", "binario", "operando"

Antes de continuar, comprendamos una 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 llama a estos "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:

```
let x = 1;


x = -x;
alert( x ); // -1, unary negation was applied
```

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

```
let x = 1, y = 3;
alert( y - x ); // 2, binary minus subtracts values
```

- Formalmente, en los ejemplos anteriores tenemos dos operadores diferentes que comparten el mismo símbolo: el operador de negación, un operador unario que invierte el signo, y el operador de resta, un operador binario que resta un número de otro.

12.1.- Operadores matemáticos

Se admiten las siguientes operaciones matemáticas:

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

12.1.1- Resto %

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

El resultado de a % b es el resto de la división entera de a y b.

Por ejemplo:

```
alert( 5 % 2 ); // 1, a remainder of 5 divided by 2
alert( 8 % 3 ); // 2, a remainder of 8 divided by 3
```

12.1.2- Exponenciación **

El operador de exponenciación a ** b multiplica a por sí mismo b veces.

Por ejemplo:

```
alert( 2 ** 2 ); // 4  (2 multiplied by itself 2 times)
alert( 2 ** 3 ); // 8  (2 * 2 * 2, 3 times)
alert( 2 ** 4 ); // 16 (2 * 2 * 2 * 2, 4 times)
```

Matemáticamente, la exponenciación se define también para números no enteros. Por ejemplo, una raíz cuadrada es una exponenciación por 1/2:

```
alert( 4 ** (1/2) ); // 2 (power of 1/2 is the same as a square root)
alert( 8 ** (1/3) ); // 2 (power of 1/3 is the same as a cubic root)
```

12.1.3- Concatenación de cadenas con operador +

Conozcamos las características de los operadores de JavaScript que van más allá de la aritmética básica.

Por lo general, el operador más +suma números.

Pero, si el binario +se aplica a cadenas, las fusiona (concatena):

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

Si alguno de los operandos es una cadena, entonces el otro también se convierte en una cadena.

Por ejemplo:

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

No importa si el primer operando es una cadena o el segundo.

Aquí hay un ejemplo más complejo:

```
alert(2 + 2 + '1' ); // "41" and not "221"
```

Aquí, los operadores trabajan uno tras otro. El primero `+` suma dos números, por lo que devuelve `4`, luego el siguiente le `+` agrega la cadena `1`, por lo que es como `4 + '1' = 41`.

El binario `+` es el único operador que admite cadenas de tal manera. Otros operadores aritméticos trabajan solo con números y siempre convierten sus operandos a números.

Aquí está la demostración de resta y división:

```
alert( 6 - '2' ); // 4, converts '2' to a number
alert( '6' / '2' ); // 3, converts both operands to numbers
```

12.1.4- Conversión numérica con operador unario +

El plus `+` existe en dos formas: la forma binaria que usamos anteriormente y la forma unaria.

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

Por ejemplo:

```
// No effect on numbers
let x = 1;
alert( +x ); // 1
```

```
let y = -2;
alert( +y ); // -2
```

```
// Converts non-numbers
alert( +true ); // 1
alert( +"" );   // 0
```

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

La necesidad de convertir cadenas en números surge con mucha frecuencia. Por ejemplo, si estamos obteniendo valores de campos de formulario HTML, generalmente son cadenas. ¿Qué pasa si queremos sumarlos?

El + binario los agregaría como cadenas:

```
let apples = "2";
let oranges = "3";

alert( apples + oranges ); // "23", the binary plus concatenates strings
```

Si queremos tratarlos como números, necesitamos convertirlos y luego sumarlos:

```
let apples = "2";
let oranges = "3";

// both values converted to numbers before the binary plus
alert( +apples + +oranges ); // 5

// the longer variant
// alert( Number(apples) + Number(oranges) ); // 5
```

Desde el punto de vista de un matemático, la abundancia de ventajas puede parecer extraña. Pero desde el punto de vista de un programador, no hay nada especial: primero se aplican las plus unarias, convierten las cadenas en números, y luego el plus binario las resume.

¿Por qué se aplican los plus unarios a los valores antes que los binarios? Como veremos, eso se debe a su *mayor precedencia* .

12.1.5- 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 lo de precedencia. Se dice que la multiplicación tiene *mayor prioridad* 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, escribe `(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.

12.1.6- Asignación

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

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

```
let x = 2 * 2 + 1;

alert( x ); // 5
```

La mayoría de los operadores en JavaScript devuelven un valor. Eso es obvio para `+`y `-`, pero también es cierto para `=`.

La llamada `x = value` escribe `value` en `x` *y luego la devuelve* .

Aquí hay una demostración que usa una asignación como parte de una expresión más compleja:

```
let a = 1;
let b = 2;

let c = 3 - (a = b + 1);
```

```
alert( a ); // 3
alert( c ); // 0
```

En el ejemplo anterior, el resultado de la expresión (a = b + 1) es el valor asignado a a(es decir 3). Luego se usa para evaluaciones adicionales.
Código gracioso, ¿no? Debemos entender cómo funciona, porque a veces lo vemos en las bibliotecas de JavaScript.
No esta recomendado escribir el código así. Tales trucos definitivamente no hacen que el código sea más claro o legible.

12.1.7- Encadenamiento de asignaciones

Otra característica interesante es la capacidad de encadenar tareas:

```
let a, b, c;

a = b = c = 2 + 2;

alert( a ); // 4
alert( b ); // 4
alert( c ); // 4
```

Las asignaciones encadenadas evalúan de derecha a izquierda. En primer lugar, la expresión más a la derecha 2 + 2se evalúa y se le asigna a las variables de la izquierda: c, by a. Al final, todas las variables comparten un solo valor.

Una vez más, para fines de legibilidad, es mejor dividir dicho código en pocas líneas:

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

Eso es más fácil de leer.

12.1.8- Modificar la misma variables

A menudo necesitamos aplicar un operador a una variable y almacenar el nuevo resultado en esa misma variable.

Por ejemplo:

```
let n = 2;
n = n + 5;
n = n * 2;
```

Esta notación se puede acortar utilizando los operadores `+=`y `*=`:

```
let n = 2;
n += 5; // now n = 7 (same as n = n + 5)
n *= 2; // now n = 14 (same as n = n * 2)

alert( n ); // 14
```

12.1.9- Incremento / decremento

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

Hay operadores especiales para ello:

- **Incremento** `++` aumenta una variable en 1:

  ```
  let counter = 2;
  counter++;       // works the same as counter = counter + 1, but is shorter
  alert( counter ); // 3
  ```

- **Decremento** `--` disminuye una variable en 1:

  ```
  let counter = 2;
  counter--;       // works the same as counter = counter - 1, but is shorter
  alert( counter ); // 1
  ```

Importante:

> Incremento / decremento solo se puede aplicar a variables. Intentar usarlo en un valor como `5++`dará un error.

Los operadores `++`y `--`se pueden colocar antes o después de una variable.

- Cuando el operador va después de la variable, es en “forma de sufijo”: `counter++`.
- La “forma de prefijo” es cuando el operador va delante de la variable: `++counter`.

Ambas declaraciones hacen lo mismo: aumentar `counter`en `1`.
¿Hay alguna diferencia? Sí, si el ++ se pone antes de la variables incrementa el valor de la variables y devuelve ese número incrementado. Si el ++ se pone después de la variable, incrementa el valor pero devuelve el valor sin incrementarlo.

```
let counter = 1;
let a = ++counter;

alert(a); // 2
```

```
let counter = 1;
let a = counter++; // (*) changed ++counter to counter++

alert(a); // 1
```

12.2- Operadores bit a bit

Los operadores bit a 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:

- Y (`&`)
- O (`|`)
- XOR (`^`)
- NO (`~`)
- SHIFT IZQUIERDA (`<<`)
- SHIFT DERECHA (`>>`)
- CAMBIO A LA DERECHA CERO-LLENADO (`>>>`)

Estos operadores se usan muy raramente, cuando necesitamos jugar con números en el nivel más bajo (bit a bit). No necesitaremos estos operadores pronto, ya que el desarrollo web los utiliza poco, pero en algunas áreas especiales, como la criptografía, son útiles.

12.3.- El operador , (coma)

El operador de coma ,es uno de los operadores más raros e inusuales. A veces, se usa para escribir código más corto, por lo que necesitamos saberlo para comprender lo que está sucediendo.

El operador de coma nos permite evaluar varias expresiones, dividiéndolas con una coma ,. Cada uno de ellos se evalúa pero solo se devuelve el resultado del último.

Por ejemplo:

```
let a = (1 + 2, 3 + 4);


alert( a ); // 7 (the result of 3 + 4)
```

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

¿Por qué necesitamos un operador que deseche todo excepto la última expresión?

A veces, las personas lo usan en construcciones más complejas para poner varias acciones en una línea.

Por ejemplo:

```
// three operations in one line
for (a = 1, b = 3, c = a * b; a < 10; a++) {
 ...
}
```

Tales trucos se usan en muchos marcos JavaScript. Por eso los estamos mencionando. Pero generalmente no mejoran la legibilidad del código, por lo que debemos pensar bien antes de usarlos.

12.4.- Operadores Relacionales

Conocemos muchos operadores de comparación de las matemáticas.

En JavaScript se escriben así:

- Mayor / menor que: a > b, a < b.
- Mayor / menor o igual a igual: a >= b, a <= b.
- Igual: a == b tenga en cuenta que el signo de doble igualdad =significa la prueba de igualdad, mientras que uno solo a = b significa una asignación.
- No es igual En matemáticas, la notación es ≠, pero en JavaScript está escrita como a != b.

Todos los operadores de comparación devuelven un valor booleano:

- true - significa "sí", "correcto" o "la verdad".
- false - significa "no", "incorrecto" o "no es la verdad".

Por ejemplo:

```
alert( 2 > 1 );  // true (correct)
alert( 2 == 1 ); // false (wrong)
alert( 2 != 1 ); // true (correct)
```

Se puede asignar un resultado de comparación a una variable, como cualquier valor:

```
let result = 5 > 4; // assign the result of the comparison
alert( result ); // true
```

12.4.1- Comparación de cadenas

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

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

Por ejemplo:

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

El algoritmo para comparar dos cadenas es simple:

1. Compara 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. Ya hemos terminado
3. De lo contrario, si los primeros caracteres de ambas cadenas son iguales, compare los segundos caracteres de la misma manera.
4. Repita hasta el final de cualquiera de las cadenas.
5. Si ambas cadenas terminan en la misma longitud, entonces son iguales. De lo contrario, la cadena más larga es mayor.

En los ejemplos anteriores, la comparación 'Z' > 'A' llega a un resultado en el primer paso mientras las cadenas "Glow"y "Glee" se comparan carácter por carácter:

1. G es el mismo que G.
2. l es el mismo que l.
3. o es mayor que e. Deténgase aquí. La primera cadena es mayor.

No es un diccionario real, sino un orden Unicode

El algoritmo de comparación dado anteriormente es más o menos equivalente al utilizado en diccionarios o guías telefónicas, pero no es exactamente el mismo.

Por ejemplo, el caso importa. Una letra mayúscula "A"no es igual a la minúscula "a". ¿Cuál es mayor? La minúscula "a". ¿Por qué? Debido a que el carácter en minúscula tiene un índice mayor en la tabla de codificación interna que usa JavaScript (Unicode). Volveremos a los detalles específicos y las consecuencias de esto en el capítulo Cadenas.

12.4.2.- Comparación de diferentes tipos.

Al comparar valores de diferentes tipos, JavaScript convierte los valores en números.

Por ejemplo:

```
alert( '2' > 1 ); // true, string '2' becomes a number 2
alert( '01' == 1 ); // true, string '01' becomes a number 1
```

Para valores booleanos, true se convierte en 1 y false se convierte en 0.

Por ejemplo:

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

12.4.3.- Igualdad estricta

Un control de igualdad regular ==tiene un problema. No puede diferenciarse 0 de false:

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

Lo mismo sucede con una cadena vacía:

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

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

¿Qué hacer si nos gustaría diferenciar 0entre false?

Un operador de igualdad estricta ===verifica la igualdad sin conversión de tipo.

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

Vamos a intentarlo:

```
alert( 0 === false ); // false, because the types are different
```

También hay un operador de "estricta no igualdad" !==análogo a !=.

El operador de igualdad estricta es un poco más largo de escribir, pero hace que sea obvio lo que está sucediendo y deja menos espacio para los errores.

12.4.4.- Resumen operadores relacionales

- Los operadores de comparación devuelven un valor booleano.
- Las cadenas se comparan letra por letra en el orden del "diccionario".
- Cuando se comparan valores de diferentes tipos, se convierten en números (con la exclusión de un estricto control de igualdad).

12.5.- Operadores logicos

Hay tres operadores lógicos en JavaScript: || (OR), && (AND), ! (NOT).

Aunque se llaman "lógicos", se pueden aplicar a valores de cualquier tipo, no solo booleanos. Su resultado también puede ser de cualquier tipo.

Veamos los detalles.

12.5.1.- || (OR)

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

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

En la programación clásica, el OR lógico está destinado a manipular solo valores booleanos. Si alguno de sus argumentos es true, devuelve true, de lo contrario, devuelve false.

En JavaScript, el operador es un poco más complicado y poderoso. Pero primero, veamos qué sucede con los valores booleanos.

Hay cuatro combinaciones lógicas posibles:

```
alert( true || true );   // true
alert( false || true );  // true
alert( true || false );  // true
alert( false || false ); // false
```

Como podemos ver, el resultado es siempre true excepto para el caso en que ambos operandos son false.

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

Por ejemplo, el número 1 se trata como true, el número 0 como false:

```
if (1 || 0) { // works just like if( true || false )
  alert( 'truthy!' );
}
```

La mayoría de las veces, OR ||se usa en una declaración if para probar si *alguna* de las condiciones dadas es true.

Por ejemplo:

```
let hour = 9;

if (hour < 10 || hour > 18) {
  alert( 'The office is closed.' );
}
```

Podemos pasar más condiciones:

```
let hour = 12;
let isWeekend = true;

if (hour < 10 || hour > 18 || isWeekend) {
  alert( 'The office is closed.' ); // it is the weekend
}
```

OR "||" encuentra el primer valor verdadero

La lógica descrita anteriormente es algo clásica. Ahora, traigamos las características "adicionales" de JavaScript.

El algoritmo extendido funciona de la siguiente manera.

Dados múltiples valores OR:

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

El operador OR hace lo siguiente:

- Evalúa operandos de izquierda a derecha.
- Para cada operando, lo convierte a booleano. Si el resultado es `true`, se detiene y devuelve el valor original de ese operando.
- Si se han evaluado todos los operandos (es decir, todos fueron `false`), devuelve el último operando.

Un valor se devuelve en su forma original, sin la conversión.

En otras palabras, una cadena de OR `"||"`devuelve el primer valor de verdad o el último si no se encuentra ningún valor de verdad.

Por ejemplo:

```
alert( 1 || 0 ); // 1 (1 is truthy)
```

```
alert( null || 1 ); // 1 (1 is the first truthy value)
alert( null || 0 || 1 ); // 1 (the first truthy value)

alert( undefined || null || 0 ); // 0 (all falsy, returns the last value)
```

Esto conduce a un uso interesante en comparación con un "OR puro, clásico, booleano solamente".

1. **Obteniendo el primer valor verdadero de una lista de variables o expresiones.**

   Por ejemplo, tenemos `firstName`, `lastNamey` `nickName` variables, todas opcionales.

   Usemos OR `||` para elegir el que tiene los datos y mostrarlo (o `anonymous` si no hay nada con valor true):

```
let firstName = "";
let lastName = "";
let nickName = "SuperCoder";

alert( firstName || lastName || nickName || "Anonymous"); // SuperCoder
```

   Si todas las variables fueran falsas, aparecería `Anonymous`.

2. **Evaluación de cortocircuito.**

   Otra característica del `||` operador OR es la llamada evaluación de "cortocircuito".

   Significa que `||` procesa sus argumentos hasta que se alcanza el primer valor verdadero, y luego el valor se devuelve inmediatamente, sin siquiera tocar el otro argumento. Esa importancia de esta característica se hace evidente si un operando no es solo un valor, sino una expresión con un efecto secundario, como una asignación de variable o una llamada a una función. En el siguiente ejemplo, se imprime el primer mensaje, mientras que el segundo no:

```
true || alert("printed");
false || alert("not printed");
```

En la primera línea, el || operador OR detiene la evaluación inmediatamente al ver true, por lo alert que no se ejecuta.
A veces, las personas usan esta función para ejecutar comandos solo si la condición en la parte izquierda es verdadera.

12.5.2.- Operador && (AND)

El operador AND se representa con dos símbolos de unión &&:

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

En la programación clásica, AND regresa true si ambos operandos son verdaderos y de lo contrario false:

```
alert( true && true );   // true
alert( false && true );  // false
alert( true && false );  // false
alert( false && false ); // false
```

Un ejemplo con if:

```
let hour = 12;
let minute = 30;

if (hour == 12 && minute == 30) {
  alert( 'The time is 12:30' );
}
```

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

```
if (1 && 0) { // evaluated as true && false
  alert( "won't work, because the result is falsy" );
}
```

Y "&&" encuentra el primer valor falso

Dados múltiples valores AND:

```
result = value1 && value2 && value3;
```

El `&&` operador AND hace lo siguiente:

- Evalúa operandos de izquierda a derecha.
- Para cada operando, lo convierte en booleano. Si el resultado es `false`, se detiene y devuelve el valor original de ese operando.
- Si se han evaluado todos los operandos (es decir, todos fueron verdaderos), devuelve el último operando.

En otras palabras, AND devuelve el primer valor falso o el último valor si no se encontró ninguno.

Las reglas anteriores son similares a OR. La diferencia es que AND devuelve el primer valor *falso* mientras OR devuelve el primer valor *verdadero* .

Ejemplos:

```
// if the first operand is truthy,
// AND returns the second operand:
alert( 1 && 0 ); // 0
alert( 1 && 5 ); // 5
```

```
// if the first operand is falsy,
// AND returns it. The second operand is ignored
alert( null && 5 ); // null
alert( 0 && "no matter what" ); // 0
```

También podemos pasar varios valores en una fila. Vea cómo se devuelve el primer falso:

```
alert( 1 && 2 && null && 3 ); // null
```

Cuando todos los valores son verdaderos, se devuelve el último valor:

```
alert( 1 && 2 && 3 ); // 3, the last one
```

La precedencia de AND `&&`es mayor que OR`||`

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

Así que el código `a && b || c && d` es esencialmente el mismo que si las expresiones están entre paréntesis: `(a && b) || (c && d)`.

No reemplace `if` con `||` o `&&`

A veces, las personas usan el `&&`operador AND como "más corto para escribir `if`".

Por ejemplo:

```
let x = 1;


(x > 0) && alert( 'Greater than zero!' );
```

La acción en la parte derecha de `&&`se ejecutará solo si la evaluación la alcanza. Es decir, solo si `(x > 0)` es cierto.

Entonces, básicamente tenemos un análogo para:

```
let x = 1;


if (x > 0) alert( 'Greater than zero!' );
```

Aunque, la variante con `&&` parece más corta, `if` es más obvia y tiende a ser un poco más legible. Por lo tanto, se recomienda usar cada construcción para su propósito: usar `if` si queremos si y usar `&&` si queremos AND.

12.5.3.-! (NO)

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

La sintaxis es bastante simple:

```
result = !value;
```

El operador acepta un único argumento y hace lo siguiente:

1. Convierte el operando a tipo booleano: `true/false`.
2. Devuelve el valor inverso.

Por ejemplo:

```
alert( !true ); // false
```

```
alert( !0 ); // true
```

Un doble NOT a `!!` veces se usa para convertir un valor a tipo booleano:

```
alert( !!"non-empty string" ); // true
alert( !!null ); // false
```

Es decir, el primer NO convierte el valor a booleano y devuelve el inverso, y el segundo NO lo invierte nuevamente. Al final, tenemos una conversión de valor a booleano.

Hay una forma un poco más detallada de hacer lo mismo: una `Boolean` función incorporada:

```
alert( Boolean("non-empty string") ); // true
alert( Boolean(null) ); // false
```

La precedencia de NOT `!`es el más alto de todos los operadores lógicos, por lo que siempre se ejecuta primero, antes que `&&` o `||`.