JAVASCRIPT

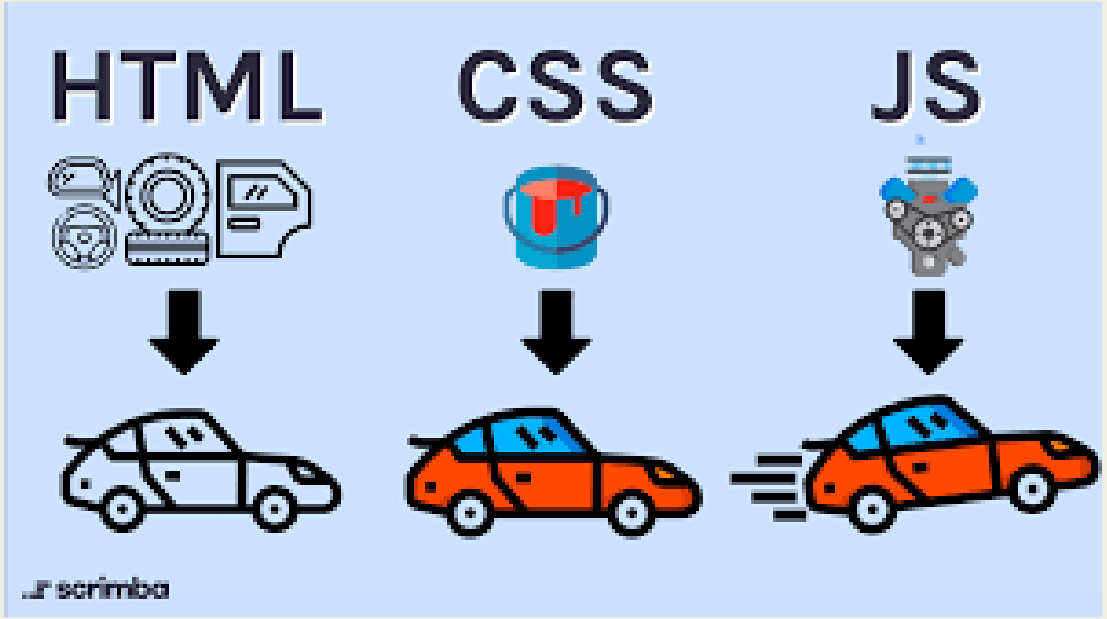
INTRODUCCIÓN

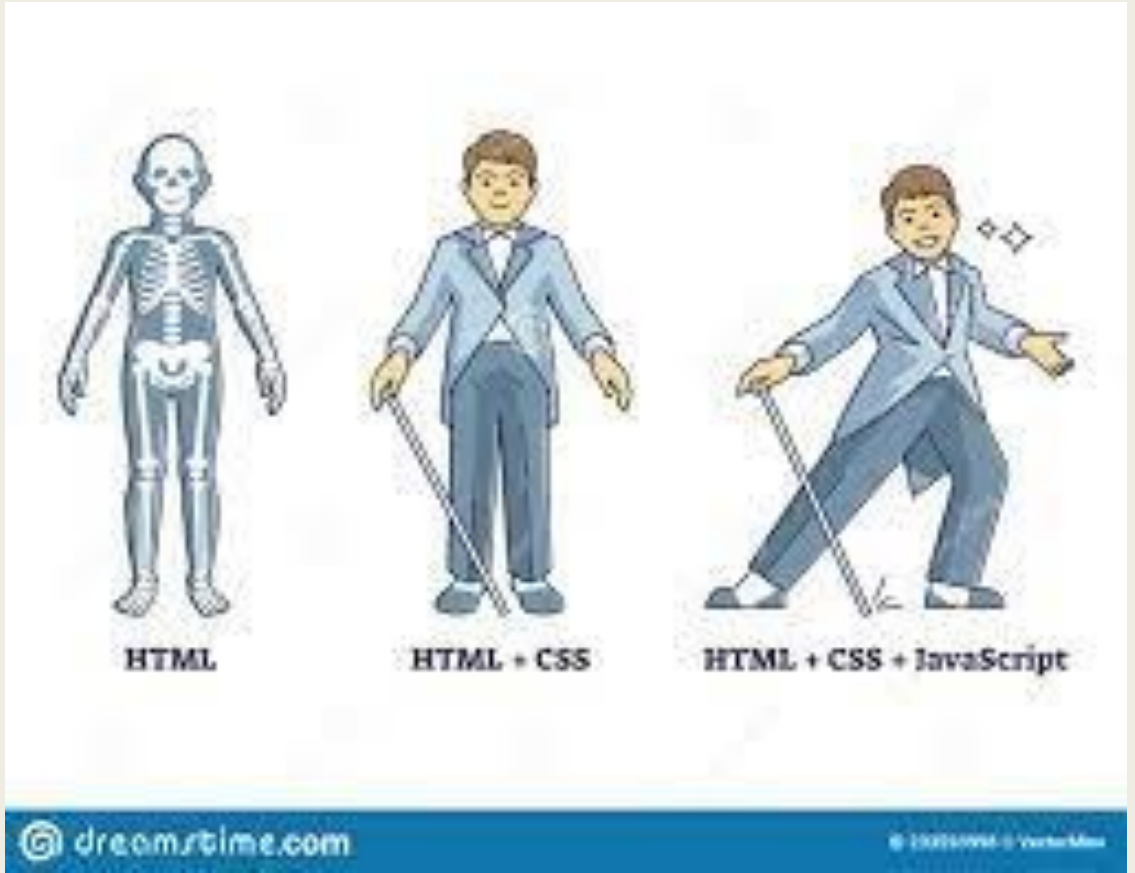
- Página web (como mínimo) = HTML y CSS

- **IMPORTANTE**: la separación bien marcada de estos dos pilares permite que si en algún momento necesitamos modificar la información (o el diseño) de la página, no tengamos también que lidiar con modificaciones en el otro.

- Con estos dos lenguajes podemos hacer muchas cosas, pero hay otras que son imposibles, o serían mucho más fáciles con un lenguaje de programación.
- En este punto aparece **Javascript**.

INTRODUCCIÓN





INTRODUCCIÓN

- Es un **lenguaje de programación**, un mecanismo con el que podemos decirle a nuestro navegador **qué tareas** debe realizar, en **qué orden** y **cuántas veces** (por ejemplo).
- Muchas de las tareas que realizamos con HTML y CSS se podrían realizar con Javascript.
- Javascript nos ofrece una mayor flexibilidad y un abanico de posibilidades más grande, y que bien usadas, pueden ahorrarnos bastante tiempo.
- Ejemplo:

```
<div class="item">
      <p>Número: <span class="numero">1</span></p>  <p>Número:
<span class="numero">2</span></p>      <p>Número: <span
class="numero">3</span></p>     <p>Número: <span
class="numero">4</span></p>     <p>Número: <span
class="numero">5</span></p>
</div>
```

INTRODUCCIÓN

- Supón que tenemos que crear una lista de números desde el 1 hasta el 500:
  - hacerlo solo con HTML sería muy tedioso: tendríamos que copiar y pegar esas filas hasta llegar a 500;
  - con Javascript, podemos decirle al navegador que escriba el primer párrafo <p>, que luego escriba el mismo pero sumándole uno al número, y que repita esto hasta 500.
- Con HTML habría que escribir 500 líneas mientras que con Javascript serían unas 10 líneas.

¿QUÉ ES JAVASCRIPT?

- Es un lenguaje de programación que se utiliza principalmente para crear páginas web dinámicas:
  - Una página web dinámica es aquella que incorpora efectos: texto que aparece y desaparece, animaciones, acciones que se activan al pulsar botones, ventanas con mensajes de aviso al usuario…
- Es un lenguaje interpretado: no es necesario compilar los programas para ejecutarlos. Los programas escritos con JavaScript se pueden probar directamente en cualquier navegador sin necesidad de procesos intermedios.
- JavaScript no guarda ninguna relación directa con el lenguaje de programación Java.

ESPECIFICACIONES OFICIALES

- ECMAScript es la especificación donde se mencionan todos los detalles de cómo debe funcionar y comportarse Javascript en un navegador.
- Así, diferentes navegadores (Chrome, Firefox, Opera, Edge, Safari...) saben cómo deben desarrollar los motores de Javascript para que cualquier código o programa funcione exactamente igual, independientemente del navegador que se utilice.
- ECMAScript suele venir acompañado de un número que indica la versión o revisión de la que hablamos. En cada nueva versión se modifican detalles sobre Javascript y/o se añaden nuevas funcionalidades.

ESPECIFICACIONES OFICIALES

- Los navegadores web intentan cumplir la especificación ECMAScript al máximo nivel, pero no todos ellos lo consiguen. Pueden existir ciertas discrepancias.
- Pueden existir navegadores que cumplan la especificación ECMAScript 6 al 80% y otros que sólo la cumplan al 60%. Puede haber características que no funcionen en un navegador y en otros sí.

Ed.	Fecha	Nombre formal/informal	Cambios significativos
1	JUN/1997	ECMAScript 1997 (ES1)	Primera edición
2	JUN/1998	ECMAScript 1998 (ES2)	Cambios leves
3	DIC/1999	ECMAScript 1999 (ES3)	RegExp, try/catch, etc...
4	AGO/2008	ECMAScript 2008 (ES4)	Versión abandonada.
5	DIC/2009	ECMAScript 2009 (ES5)	Strict mode, JSON, etc...
5.1	DIC/2011	ECMAScript 2011 (ES5.1)	Cambios leves

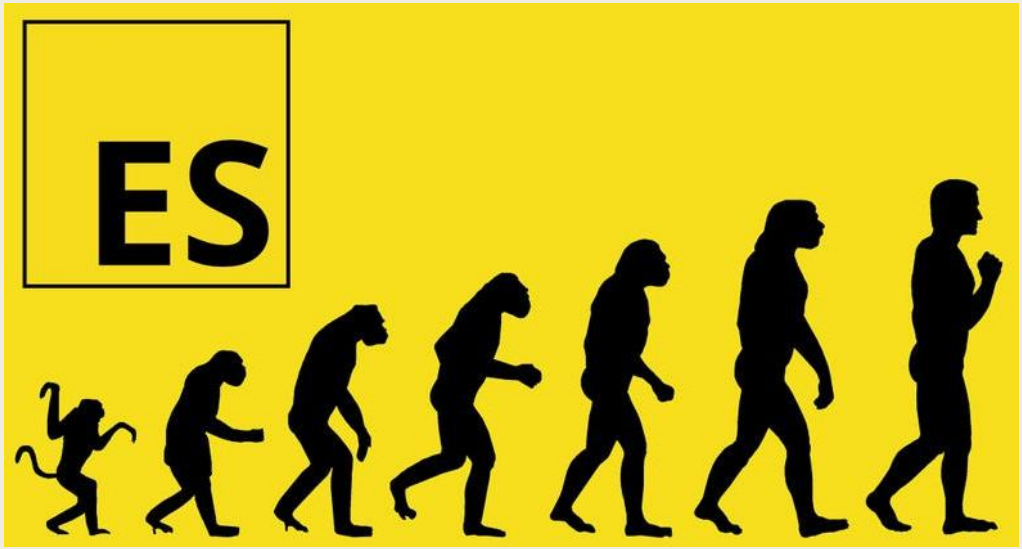
ESPECIFICACIONES OFICIALES

- A partir del año 2015, se marcó un antes y un después en el mundo de Javascript, estableciendo una serie de cambios que lo transformarían en un lenguaje moderno, partiendo desde la especificación de dicho año, hasta la actualidad:

Ed.	Fecha	ECMA Script	Cambios significativos
6	JUN/2015	ES2015	Clases, módulos, hashmaps, sets, for of, proxies...
7	JUN/2016	ES2016	Array includes(), Exponenciación **
8	JUN/2017	ES2017	Async/await
9	JUN/2018	ES2018	Rest/Spread operator, Promise.finally()...
10	JUN/2019	ES2019	flat, flatMap, trimStart(), optional error catch...
11	JUN/2020	ES2020	Dynamic imports, BigInt, Promise.allSettled

ESPECIFICACIONES OFICIALES

- Algunos navegadores implementan pequeñas funcionalidades de versiones posteriores de ECMAScript antes que otras, para ir testeando y probando características, por lo que no es raro que algunas características de futuras especificaciones puedan estar implementadas en algunos navegadores.
- Una buena forma de conocer en qué estado se encuentra un navegador concreto en su especificación de ECMAScript es consultando la tabla de compatibilidad Kangax. En la columna «Desktop browsers» se ve el porcentaje de compatibilidad con las diferentes características de determinadas especificaciones de ECMAScript.



CÓMO INCLUIR JAVASCRIPT EN DOCUMENTOS HTML

- La integración es muy flexible: existen tres formas para incluir código JavaScript en las páginas web:
  1. El **código JavaScript** se encierra **entre etiquetas <script>** y se incluye en cualquier parte del documento.
  2. Las instrucciones **JavaScript** se pueden incluir **en un archivo externo** de tipo JavaScript que se enlaza mediante la etiqueta <script>. Se pueden crear todos los archivos JavaScript que sean necesarios y cada documento HTML puede enlazar tantos archivos JavaScript como necesite.
  3. El código JavaScript se incluye en el elemento HTML sobre el que quiera utilizarse.

CÓMO INCLUIR JAVASCRIPT

Etiqueta <script>…</script> dentro de <head>.


```
<!DOCTYPE html>
<html>
  <head>
    <title>Ejemplo JavaScript en el documento</title>
    <script type="text/javascript">
      alert("Un mensaje de prueba");
    </script>
  </head>
  <body>
    <p>Un párrafo de texto.</p>
  </body>
</html>
```

CÓMO INCLUIR JAVASCRIPT EN EL DOCUMENTO HTML

- Este método se emplea cuando se define un bloque pequeño de código o cuando se quieren incluir instrucciones específicas en un determinado documento HTML que completen las instrucciones y funciones que se incluyen por defecto en todos los documentos del sitio web.
- **Inconveniente:** si se quiere hacer una modificación en el bloque de código, es necesario modificar todas las páginas que lo incluyan.

CÓMO INCLUIR JAVASCRIPT

Documento externo.

- Archivo codigo.js

```
alert("Un mensaje de prueba");
```

- Documento XHTML

```
<!DOCTYPE html>
<html>
  <head>
    <title>Ejemplo JavaScript en el documento</title>
    <script src="/js/codigo.js"></script>
  </head>
  <body>
    <p>Un párrafo de texto.</p>
  </body>
</html>
```

CÓMO INCLUIR JAVASCRIPT EN UN DOCUMENTO HTML

- Este método requiere definir el atributo src, que es el que indica la URL correspondiente al archivo JavaScript que se quiere enlazar. **Cada etiqueta <script> solamente puede enlazar un único archivo**, pero en una misma página **se pueden incluir tantas etiquetas <script> como sean necesarias.**
- Los archivos de tipo JavaScript son documentos normales de texto con la extensión .js, que se pueden crear con cualquier editor de texto como Notepad, VSC, Vi, etc.
- **Ventajas:**
  - se simplifica el código HTML de la página,
  - se puede reutilizar el mismo código JavaScript, y
  - cualquier modificación realizada en el archivo JavaScript se ve reflejada inmediatamente en todas las páginas HTML que lo enlazan.

CÓMO INCLUIR JAVASCRIPT

En línea en el documento HTML.


```
<!DOCTYPE html>
<html>
  <head>
    <title>Ejemplo JavaScript en el documento</title>
  </head>
  <body>
    <p onclick="alert('Un mensaje de prueba')">
      Un párrafo de texto.
    </p>
  </body>
</html>
```

- **Inconvenientes:**
  - ensucia el código HTML de la página,
  - complica el mantenimiento del código JavaScript.

CÓMO INCLUIR JAVASCRIPT EN UN DOCUMENTO HTML

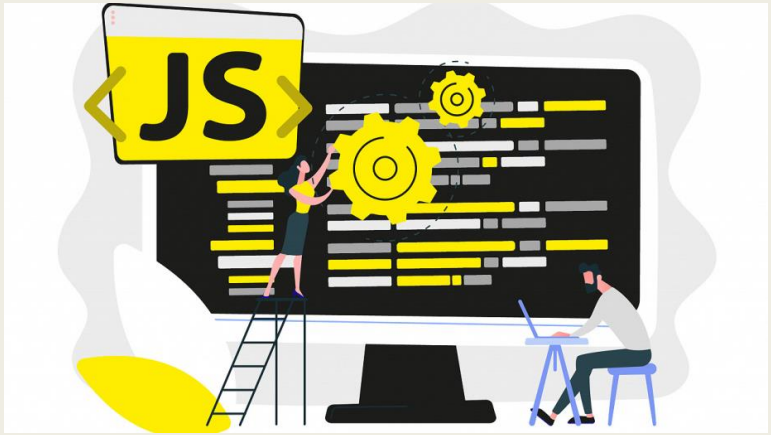
- **Esquema general de una página web**:
  - documento HTML donde están todas las etiquetas HTML de la página,
  - a lo largo del documento, pueden existir referencias a otros documentos, como archivos CSS o archivos Javascript.
- Si dentro del documento HTML se encuentra una referencia a un archivo CSS, el navegador lo descarga y lo aplica, cambiando su apariencia visual. De la misma forma, si encuentra una referencia a un archivo Javascript, el navegador lo descarga y ejecuta las órdenes o acciones que allí se indican.

- La etiqueta script puede estar en cualquier sitio del documento HTML

Ubicación	¿Cómo descarga el archivo?	Estado de la página
<head>	ANTES de empezar a dibujar la página.	Página aún no dibujada.
<body>	DURANTE el dibujado de la página.	Dibujada hasta donde está la etiqueta <script>.
Antes de </body>	DESPUÉS de dibujar la página.	Dibujada al 100%.

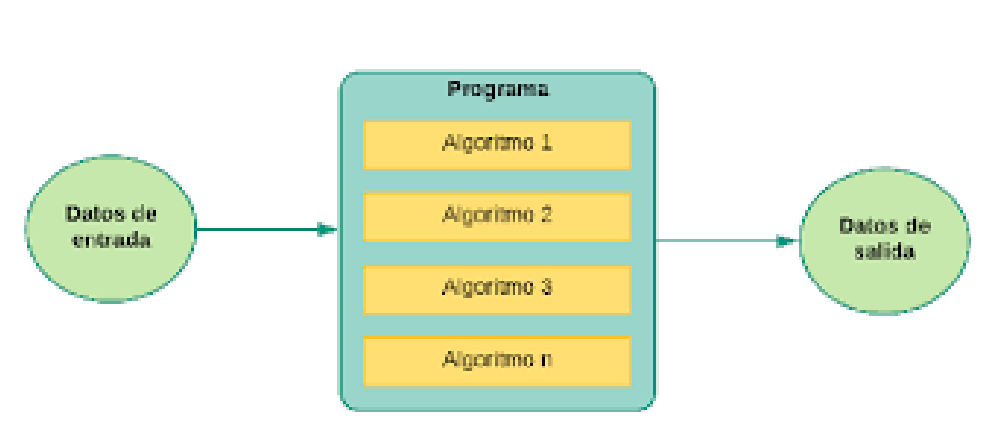
CÓMO INCLUIR JAVASCRIPT EN UN DOCUMENTO HTML

- El navegador puede descargar un documento Javascript en cualquier momento de la carga de la página. Importante saber cuál es el más oportuno para nosotros:
  - Si queremos que **Javascript actúe antes de que se muestre la página**, la opción de colocarlo en el **<head>** es la más adecuada.
  - Si queremos que **actúe una vez se haya terminado de cargar la página**, la opción de colocarlo **justo antes del </body>** es la más adecuada.



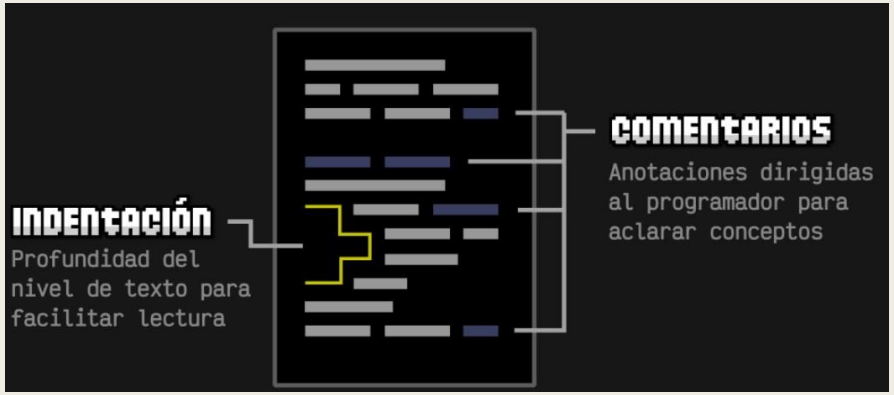
GLOSARIO GENERAL

- **Programa**: conjunto total de código que desarrollas. En Desarrollo web, quizás el término más utilizado es sitio web (más orientado a contenido web) o aplicación web (más orientado a funcionalidad web). También se suele generalizar utilizando términos como «script» o «código Javascript», haciendo referencia a fragmentos más pequeños de esa aplicación.
- **Algoritmo**: conjunto de pasos conocidos, en un determinado orden, para conseguir realizar una tarea satisfactoriamente y lograr un objetivo.



GLOSARIO GENERAL

- **Comentarios**: fragmentos de texto o anotaciones que el navegador ignora y no repercuten en el programa. Sirven para dejar por escrito detalles importantes para el programador. De esta forma cuando volvamos al código, nos será más rápido comprenderlo. Es una **buena costumbre comentar** en la medida de lo posible nuestro código.
- **Indentación**: colocar espacios o tabuladores antes del código, para indicar si nos encontramos dentro de un **if**, de un **bucle**, etc... Esta **práctica** es **muy importante y necesaria**.



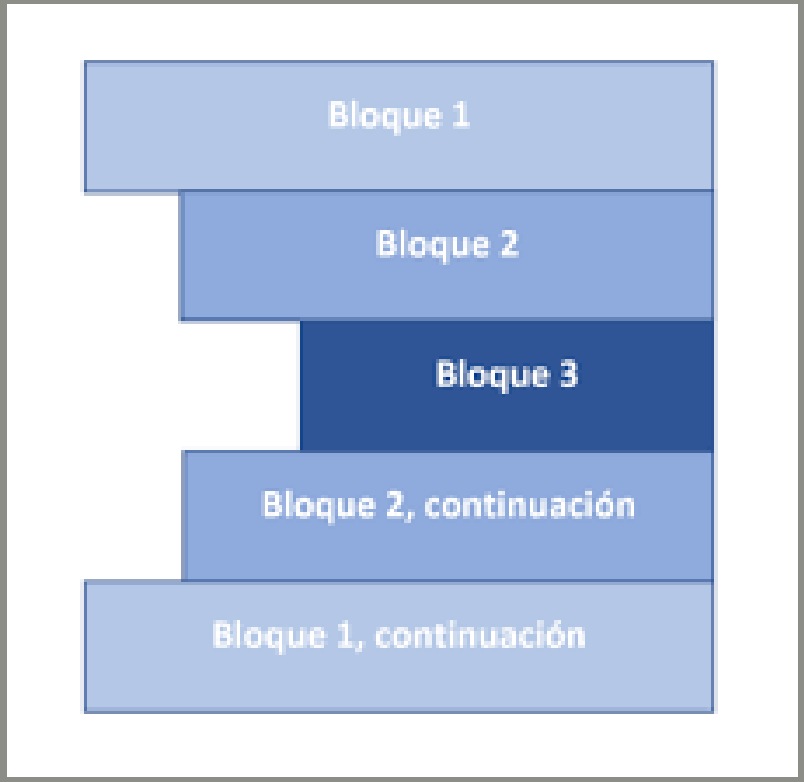
GLOSARIO

Consejos sobre comentarios

- No comentar detalles redundantes. No escribir lo que se hace, **escribir por qué se hace**.
- **Nombrar** variables/funciones/clases **de forma descriptiva** en lugar de usar comentarios para describirlas.
- Ser **conciso y concreto**. Resumir. No escribir párrafos.
- Usar siempre el **mismo** idioma y **estilo de comentarios**.
- **Si se modifica código**, **revisar** también **los comentarios**. Comentarios desactualizados, son inservibles.
- En Javascript existen dos tipos de comentarios: los **comentarios de una sola línea** y los **comentarios de múltiples líneas**.
  - **Una línea**: // y sólo comenta esa línea.
  - **Múltiples líneas**: comentarios extensos
    /* comenta todo el texto que se escribe hasta cerrar el comentario con un */

GLOSARIO

Indentación



- Ejemplo:

```
function action() {
  for (i = 0; i < 10; i++) {
        console.log("Iteración #", i);
        if (i == 9) {
                console.log("Estoy en la última iteración");
        } // if
  } // for
} // function
```

- Indentar muestra visualmente qué fragmento de código actúa dentro de otro. Así, es sencillo saber a qué nivel está actuando cada línea de código.

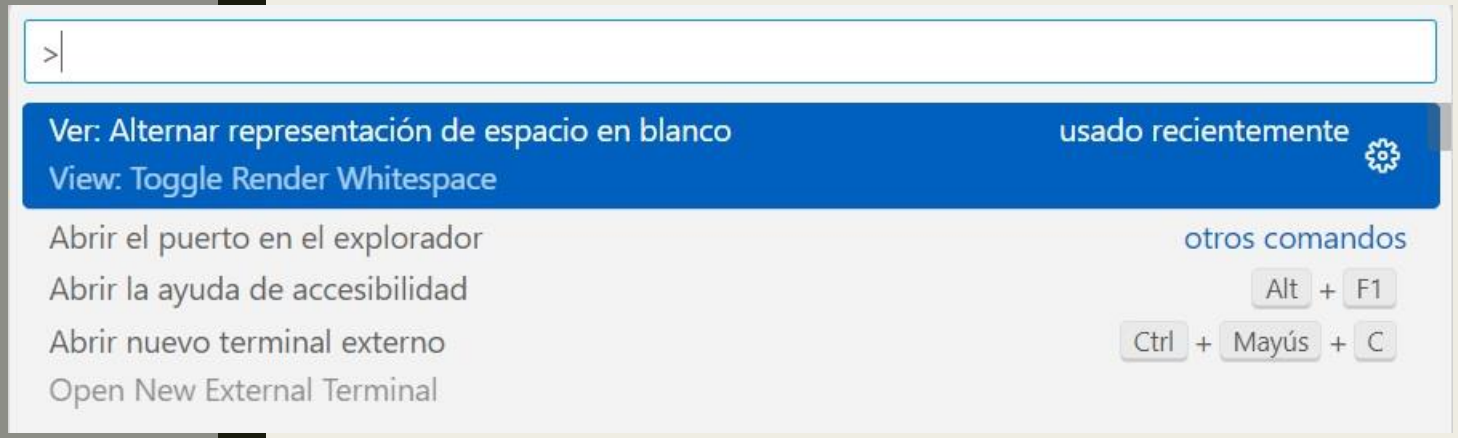
```
function action() { for (i = 0; i < 10; i++) {      console.log("Iteración    #", i);
  if (i == 9) console.log("Estoy en la última iteración"); } }
```

- Puede parecer que este código aprovecha mejor el espacio porque ocupa menos, pero es mucho menos legible debido a su falta de indentación: no queda claro qué instrucciones están dentro de otras.

GLOSARIO

Indentación

- Indentar código: ¿usar espacios o tabuladores? Importante es ser coherente y siempre utilizar la misma.
- En **VSCode** puedes **activar la visualización** de esta característica mediante puntos · (espacios) o flechas **->** (tabulador), pulsando la tecla **F1** y buscando/activando la opción «Alternar representación de espacio en blanco» (Toggle Render Whitespace).



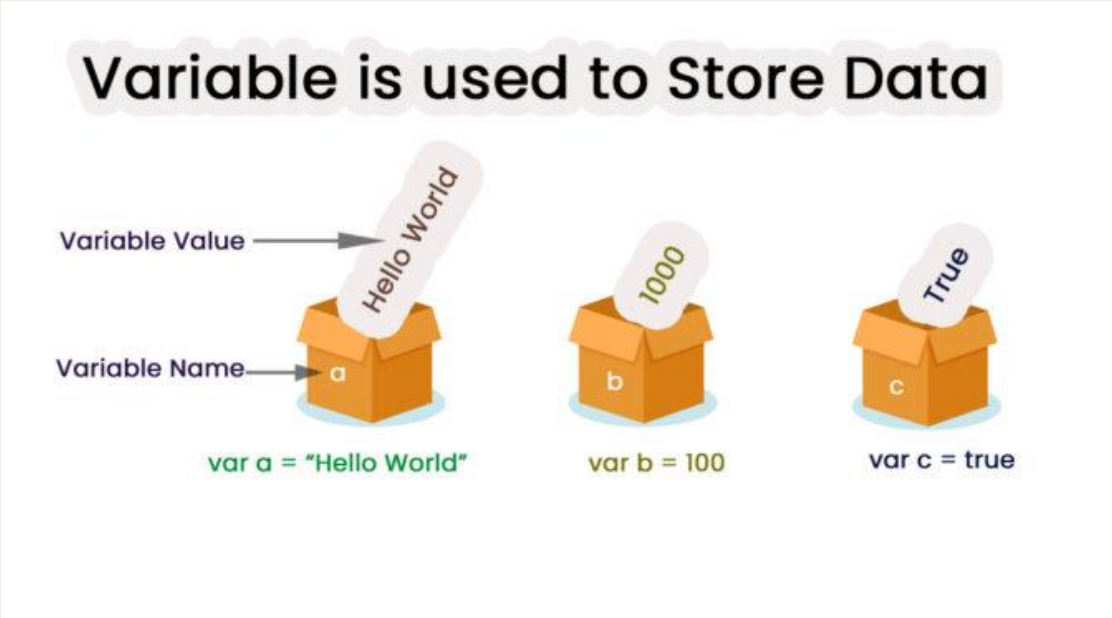
GLOSARIO GENERAL

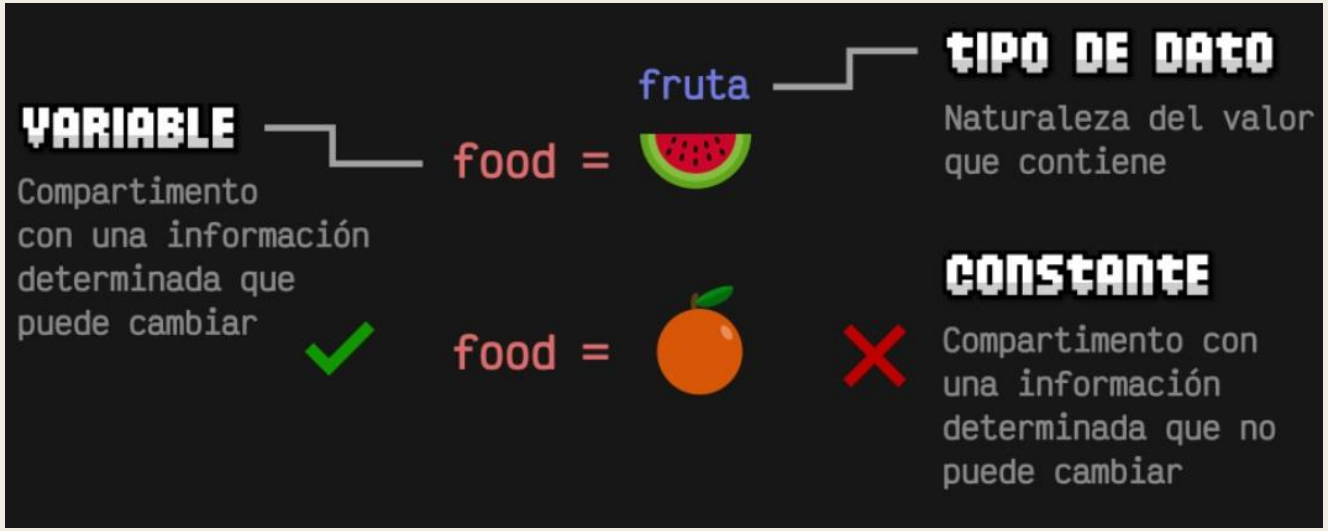
- **Variables**: nombre que se le da a pequeños «compartimentos» (espacios en memoria) donde se guarda una información determinada, de forma muy similar a las incógnitas en matemáticas. Se llaman variables porque podemos modificarlas a lo largo del programa, según necesitemos. Un programa puede tener muchas variables, y cada una de ellas tendrá:
  - Un **nombre:** para identificarlas y diferenciarlas de otras variables.
  - Un **valor:** la información que contienen.
  - Un **tipo de dato:** la naturaleza del valor que contiene.

```
x = 5; // nombre: x, valor: 5, tipo de dato: número
y = "Hola"; // nombre: y, valor: Hola, tipo de dato: texto
Manz = "me"; // nombre: Manz, valor: me, tipo de dato: texto
```

- **Constantes**: mismo concepto de una variable, salvo que en este caso, la información que contiene no puede variar (será siempre constante).

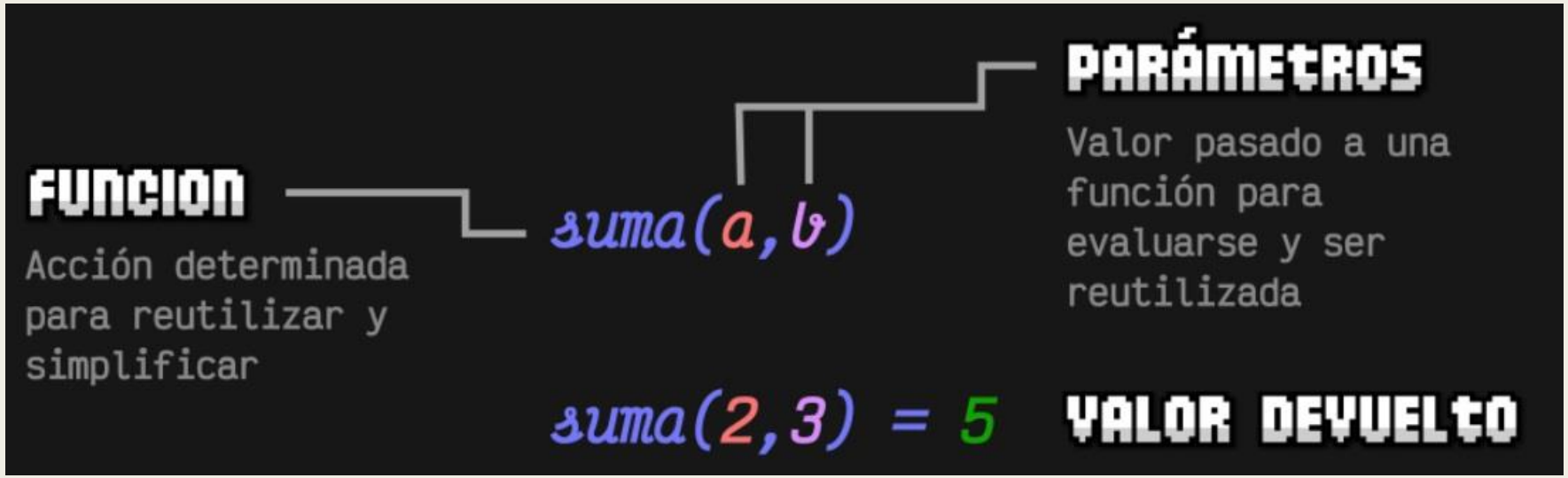
GLOSARIO GENERAL





GLOSARIO GENERAL

- **Funciones**: cuando comenzamos a programar, nuestro código se va haciendo cada vez más y más grande, por lo que hay que buscar formas de organizarlo y mantenerlo lo más simple posible. Las funciones son agrupaciones de código que, entre otras cosas, evitan que tengamos que escribir varias veces lo mismo en nuestro código. Una función contendrá una o más acciones a realizar y cada vez que ejecutemos una función, se realizarán todas ellas.
- **Parámetros**: nombre que reciben las variables que se le pasan a las funciones. Muchas veces también se les denomina argumentos.



GLOSARIO GENERAL

- **Bucles**: muchas veces necesitaremos realizar tareas repetitivas. Una de las ventajas de la programación es que permite automatizar acciones y no es necesario hacerlas varias veces. Los **bucles permiten indicar el número de veces que se repetirá una acción**. De esta forma, sólo la escribimos una vez en nuestro código, y simplemente indicamos el número de veces que queremos que se repita.
- **Iteración**: cuando el programa está en un bucle repitiendo varias veces la misma tarea, cada una de esas repeticiones se denomina **iteración**.

GLOSARIO GENERAL

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

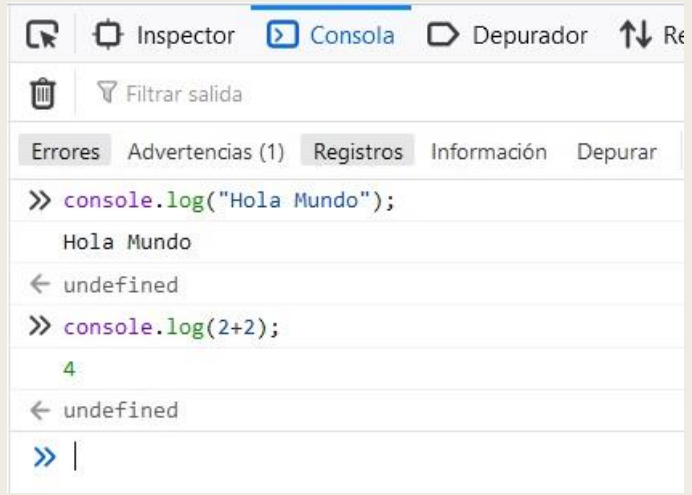
SINTAXIS

- Conjunto de reglas que deben seguirse al escribir el código fuente de los programas para considerarse correctos.
- La sintaxis de JavaScript es muy similar a la de otros lenguajes de programación. Normas básicas:
- **No se tienen en cuenta los espacios en blanco y las nuevas líneas**: el intérprete de JavaScript ignora cualquier espacio en blanco sobrante, por lo que el código se puede ordenar de forma adecuada para entenderlo mejor (tabulando las líneas, añadiendo espacios, creando nuevas líneas, etc.).
- **Se distinguen las mayúsculas y minúsculas**.
- **No se define el tipo de las variables**: al crear una variable, no es necesario indicar el tipo de dato que almacenará. De esta forma, una misma variable puede almacenar diferentes tipos de datos durante la ejecución del script.
- **No es necesario terminar cada sentencia con el carácter de punto y coma (;)**: JavaScript no obliga a hacerlo, pero es conveniente.

LA CONSOLA JAVASCRIPT

- Es una **herramienta** que se encuentra en la mayoría de los **navegadores web** y permite a los desarrolladores **interactuar** con el código JavaScript **y depurar** su aplicación.
- Para **acceder** a la consola Javascript del navegador:
  - **CTRL+SHIFT+I:** lleva al Inspector de elementos del navegador. Ahora nos interesa la **pestaña Console**.
  - En algunos navegadores **CTRL+SHIFT+J** nos lleva directamente a la consola.
- En esta consola, podemos escribir funciones o sentencias de Javascript que estarán actuando en la página que se encuentra en la pestaña actual del navegador. De esta forma podremos observar los resultados que nos devuelve en la consola al realizar diferentes acciones:

```
console.log("Hola Mundo"); //al pulsar enter nos muestra el texto «Hola Mundo».
console.log(2 + 2); //procesa la operación y nos devuelve 4.
```



LA CONSOLA JAVASCRIPT

- Para mostrar estos textos en la consola Javascript hemos utilizado la **función console.log**. Existen varias más:

FUNCIÓN	DESCRIPCIÓN
console.log()	Muestra la información proporcionada en la consola Javascript.
console.info()	Equivalente al anterior. Se utiliza para mensajes de información.
console.warn()	Muestra información de advertencia. Aparece en amarillo.
console.error()	Muestra información de error. Aparece en rojo.
console.clear()	Limpia la consola. Equivalente a pulsar CTRL+L o escribir `clear()`.

- console.log() y sus funciones hermanas permiten añadir varios datos en una misma línea, separándolo por comas:

```
console.log("¡Hola a todos! Observen este número: ", 5 + 18);
```

EL PRIMER SCRIPT

```
<!DOCTYPE html>
<html lang=“es”>
  <head>
    <meta charset=“utf-8”/>
    <title>El primer script</title>
    <script type="text/javascript">
      alert("Hola Mundo!");
    </script>
  </head>
  <body>
    <p>Esta página contiene el primer script</p>
  </body>
</html>
```

EL PRIMER SCRIPT

- La instrucción alert() es una de las utilidades que incluye JavaScript y permite mostrar un mensaje en la pantalla del usuario. Si se visualiza la página web de este primer script en cualquier navegador, automáticamente se mostrará una ventana con el mensaje "Hola Mundo!".
- El funcionamiento de la utilidad alert() en los distintos navegadores es idéntico. Sin embargo, existen diferencias visuales en la forma en la que se presentan los mensajes.

- *Ejercicio 1*

PROGRAMACIÓN BÁSICA

- Antes de comenzar a desarrollar programas y utilidades con JavaScript, es necesario conocer los elementos básicos con los que se construyen las aplicaciones en JavaScript.

VARIABLES Y CONSTANTES

- Las variables en los lenguajes de programación siguen una lógica similar a las variables utilizadas en otros ámbitos como las matemáticas: es un elemento que se emplea para almacenar y hacer referencia a otro valor.
- Si no existieran variables, un programa que suma dos números podría escribirse como:

  resultado = 3 + 1

- Esto es poco útil, sólo sirve para el caso en el que el primer sumando sea 3 y el segundo 1. En cualquier otro caso, el *programa* obtiene un resultado incorrecto.

PROGRAMACIÓN BÁSICA VARIABLES Y CONSTANTES

- El programa se puede rehacer utilizando variables para almacenar y referirse a cada número:

  numero_1 = 3

  numero_2 = 1

  resultado = numero_1 + numero_2

- Los elementos numero_1 y numero_2 son **variables** que almacenan los valores que utiliza el programa. El resultado se calcula siempre en función del valor almacenado por las variables. Si se modifica el valor de las variables numero_1 y numero_2, el programa sigue funcionando correctamente.

PROGRAMACIÓN BÁSICA VARIABLES Y CONSTANTES

- **Variables y constantes**
  - En javascript es muy sencillo declarar y utilizar variables. Hay que tener unos conceptos previos claros para evitar confusiones.
  - **Inicialización**: en Javascript, se utiliza la palabra clave ***let*** para declarar variables. **Si** una variable **no se inicializa con un valor concreto, contendrá** un valor especial llamado **undefined**, que significa que su valor no está definido aún, o lo que es lo mismo, que no contiene información.

```
let a;          // Declaramos una variable con nombre "a", pero no le asociamos contenido.
let b = 0;      // Declaramos una variable con nombre "b", y le asociamos el número 0.

console.log(b); // Muestra 0 (el valor guardado en la variable "b")
console.log(a); // Muestra "undefined" (no hay valor guardado en la variable "a")
```

PROGRAMACIÓN BÁSICA VARIABLES Y CONSTANTES

- En JavaScript no es obligatorio inicializar las variables, ya que se pueden declarar por una parte y asignarles un valor posteriormente.
- En JavaSript tampoco es necesario declarar las variables. En otras palabras, se pueden utilizar variables que no se han definido anteriormente mediante la palabra reservada let.

```
let numero_1 = 3;
let numero_2 = 1;
resultado = numero_1 + numero_2;
```

```
numero_1 = 3;
numero_2 = 1;
resultado = numero_1 + numero_2;
```

- Con las variables no declaradas JavaScript crea una variable global (para cada una de ellas -más adelante se verán las diferencias entre variables locales y globales) y les asigna el valor correspondiente.

PROGRAMACIÓN BÁSICA
NOMBRAR VARIABLES Y CONSTANTES

- El nombre de una variable también se conoce como **identificador** y debe cumplir las siguientes normas:
  - Sólo puede estar formado por letras, números y los símbolos $ (dólar) y _ (guion bajo).
  - El primer carácter no puede ser un número.
- Ejemplos válidos:

```
let $numero1;
let _$letra;
let $$$otroNumero;
let $_a__$4;
```

- Identificadores incorrectos:

```
let 1numero; // Empieza por un número
let numero;1_123; // Contiene un carácter ";"
```

PROGRAMACIÓN BÁSICA
NOMBRAR VARIABLES Y CONSTANTES

- Las **constantes** deben ir siempre en **MAYÚSCULAS**.
- Las **variables siempre** deben **empezar por letra minúscula**.

Nombre	Descripción	Ejemplo
camelCase (variables)	Primera palabra todo minúsculas. El resto, todo minúsculas salvo primera letra.	precioProducto
constantes	Todo en mayúsculas.	DESCUENTO

```
> const MAX_SIZE = 4;
  MAX_SIZE = 10;
✕ ▶ Uncaught TypeError: Assignment to constant variable.
      at <anonymous>:2:10
```

TIPOS DE DATOS

- Aunque todas las variables y constantes de JavaScript se crean de la misma forma, la forma en la que se les asigna un valor depende del tipo de valor que se quiere almacenar (números, textos, etc.).
- El **tipo de dato** es la naturaleza del **contenido de una variable o constante**. Por ejemplo, una variable con contenido 44, su tipo de dato es un número. Una variable con contenido "Manz", su tipo de dato es texto.

Tipo	Descripción	Ejemplo
Number	Valor numérico (enteros, decimales, etc...)	**let iva = 16;** *// variable tipo entero* **let total = 234.65;** *// variable tipo decimal*
BigInt	Valor numérico grande	**let maxValor = 123456789012345678 9n;**
String	Valor de texto (cadenas de texto, carácteres, etc...)	**let nombre = 'MZ‘;** **let mensaje = "Bienvenido a nuestro sitio web";** **let nombreProducto = 'Producto ABC';** **let letraSeleccionada = 'c';** */* texto1 tiene comillas simples, se encierra con comillas dobles */* **let texto1 = "Una frase con 'comillas simples' dentro";** */* texto2 tiene comillas dobles, se encierra con comillas simples */* **let texto2 = 'Una frase con "comillas dobles" dentro';**

TIPOS DE DATOS

- A veces las cadenas de texto contienen caracteres que son difíciles de incluir en una variable de texto (tabulador, ENTER, etc.) Para resolver estos problemas, JavaScript define un mecanismo para incluir de forma sencilla caracteres especiales y problemáticos dentro de una cadena de texto.
- El mecanismo consiste en sustituir el carácter problemático por una combinación simple de caracteres:

Si se quiere incluir…	Se debe incluir…
Una nueva línea	\n
Un tabulador	\t
Una comilla simple	\'
Una comilla doble	\"
Una barra inclinada	\\

- *Ejercicio 2*

TIPOS DE DATOS

Tipo	Descripción	Ejemplo
Boolean	Valor booleano (valores verdadero o falso)	**let clienteRegistrado = false;** **let ivaIncluido = true;**
Undefined	Valor sin definir (variable sin inicializar)	undefined
Array	También llamados vectores, matrices e incluso *arreglos*. Es una colección de variables, que pueden ser todas del mismo tipo o cada una de un tipo diferente.	**let dias = ["Lunes", "Martes", "Miércoles", "Jueves", "Viernes", "Sábado", "Domingo"];** **let diaSeleccionado = dias[0];** // *diaSeleccionado = "Lunes"* **let otroDia = dias[5];** // *otroDia = "Sábado"*
Function	Función (función guardada en una variable)	function() {}
Symbol	Símbolo (valor único)	Symbol(1)
Object	Objeto (estructura más compleja)	{}

- *Ejercicio 3*

TIPOS DE DATOS

- **Javascript** es un **lenguajes dinámico**: al crear una variable no es necesario indicarle el tipo de dato que va a contener, deduce o infiere el tipo de dato dependiendo del valor que le hayamos asignado. Así, a lo largo del programa, dicha variable puede «cambiar» a tipos de datos diferentes.
- Aunque no tengamos que indicar el tipo de dato, **los tipos de datos existen**, se puede acceder a ellos e incluso **convertir entre un tipo de dato y otro**.
- Ejemplo:

```
const VALUE_AS_TEXT = "50";
const VALUE_AS_NUMBER = 50;
const CONVERTED_VALUE = Number("50");

console.log(VALUE_AS_TEXT); // "50"
console.log(VALUE_AS_NUMBER); // 50
console.log(CONVERTED_VALUE); // 50
```

- **VALUE_AS_TEXT** tiene un tipo de dato de texto, mientras que **VALUE_AS_NUMBER** tiene un tipo de dato numérico. Se ha utilizado **Number()** para forzar un texto y convertirlo a otro tipo de dato (numérico).

TIPOS DE DATOS

- Hay varias formas de saber qué tipo de dato tiene una variable en Javascript:
- **Utilizando typeof()**
  - Si tenemos dudas, devuelve el tipo de dato de la variable que le pasemos por parámetro:

```
console.log(typeof 42); // Expected output: "number“
console.log(typeof 'blubber');// Expected output: string“
console.log(typeof true);// Expected output: "boolean“
console.log(typeof undeclaredVariable);// Expected output: "undefined“
let variable="50";
console.log(typeof (variable)); // Expected output: string“
console.log(typeof notDefined); // undefined
```

- **CUIDADO**: la función typeof() no nos servirá para variables con tipos de datos más complejos, ya que siempre los mostrará como object. Es mejor utilizar **constructor.name**.

OPERADORES

- Permiten manipular el valor de las variables, realizar operaciones matemáticas con sus valores y comparar diferentes variables. Permiten a los programas realizar cálculos complejos y tomar decisiones lógicas en función de comparaciones y otros tipos de condiciones.

ASIGNACIÓN

- Se utiliza para guardar un valor específico en una variable. El símbolo utilizado es =

```
let numero1 = 3;
```

- A la izquierda del operador, siempre debe indicarse el nombre de una variable. A la derecha del operador, se pueden indicar variables, valores, condiciones lógicas, etc:

```
let numero1 = 3;
let numero2 = 4;
5 = numero1; /* Error, la asignación siempre se realiza a una variable, por lo que en la izquierda no se puede indicar un número */
numero1 = 5; // Ahora, la variable numero1 vale 5
numero1 = numero2; // Ahora, la variable numero1 vale 4
```

OPERADORES INCREMENTO Y DECREMENTO

- Solo válidos para variables numéricas. Se utilizan para **incrementar o decrementar en una unidad** el valor de una variable.

- **Incremento**

```
let numero = 5;
++numero;
alert(numero); // numero = 6
```

- El operador de **incremento** se indica mediante el **prefijo ++ en el nombre de la variable.**
- Ejemplo equivalente:

```
let numero = 5;
numero = numero + 1;
alert(numero); // numero = 6
```

- **Decremento**

```
let numero = 5;
--numero;
alert(numero); // numero = 4
```

- El operador de **decremento** se indica mediante el **prefijo -- en el nombre de la variable.**
- Ejemplo equivalente:

```
let numero = 5;
numero = numero - 1;
alert(numero); // numero = 4
```

OPERADORES INCREMENTO Y DECREMENTO

- Los operadores de incremento y decremento se pueden indicar como sufijo. La diferencia es que el incremento se hace **después** de la operación en la que participen, si no están en operación, su resultado es el mismo:

```
let numero = 5;
numero++;
alert(numero); // numero = 6
```

- Sin embargo en:

```
let numero1 = 5;
let numero2 = 2;
numero3 = numero1++ + numero2;
// numero3 = 7, numero1 = 6
-----------------------------------
let numero1 = 5;
let numero2 = 2;
numero3 = ++numero1 + numero2;
// numero3 = 8, numero1 = 6
```

OPERADORES LÓGICOS

- Se utilizan para tomar decisiones sobre las instrucciones que debería ejecutar el programa en función de ciertas condiciones.
- **El resultado** de cualquier operación que utilice operadores lógicos **siempre es un valor lógico o *booleano***.

NEGACIÓN

- Se utiliza para obtener el valor contrario al valor de la variable:

```
let visible = true;
alert(!visible); // Muestra "false" y no "true"
```

- Se obtiene prefijando el símbolo ! al identificador de la variable.

variable	!variable
true	false
false	true

OPERADORES LÓGICOS - NEGACIÓN

- Si la variable original es de tipo *booleano*, es muy sencillo obtener su negación.
- ¿Qué sucede cuando la variable es un número o una cadena de texto? Para obtener la negación en este tipo de variables, se realiza en primer lugar su conversión a un valor *booleano*:
  - Si la variable contiene un **número**, se transforma en **false si vale 0** y en **true para cualquier otro número.**
  - Si la variable contiene una **cadena de texto**, se transforma en **false si la cadena es vacía ("")** y en **true en cualquier otro caso.**

```
let cantidad = 0;
vacio = !cantidad; // vacio = true
cantidad = 2;
vacio = !cantidad; // vacio = false
```

```
var mensaje = "";
mensajeVacio = !mensaje; // mensajeVacio = true
mensaje = "Bienvenido";
mensajeVacio = !mensaje; // mensajeVacio = false
```

OPERADORES LÓGICOS - AND

- La operación lógica AND obtiene su resultado combinando dos valores booleanos. El operador se indica mediante el símbolo **&&** y su resultado solamente es true si los dos operandos son true:

variable1	variable2	variable1&&variable2
true	true	true
true	false	false
false	true	false
false	false	false

```
let valor1 = true;

let valor2 = false;

resultado = valor1 && valor2;
//resultado = false

valor1 = true;

valor2 = true;

resultado = valor1 && valor2;
//resultado = true
```

OPERADORES LÓGICOS - OR

- También combina dos valores booleanos. El operador se indica mediante el símbolo || y su resultado es true si alguno de los dos operandos es true:

variable1	variable2	variable1||variable2
true	true	true
true	false	true
false	true	true
false	false	False

```
let valor1 = true;

let valor2 = false;

resultado = valor1 || valor2;
//resultado = true

valor1 = false;

valor2 = false;

resultado = valor1 || valor2;
//resultado = false
```

OPERADORES MATEMÁTICOS

- JavaScript permite realizar manipulaciones matemáticas sobre el valor de las variables numéricas. Los operadores definidos son: suma (+), resta (-), multiplicación (*), división (/) y módulo (%).

```
let numero1 = 10;
let numero2 = 5;
resultado = numero1 / numero2; // resultado = 2
resultado = 3 + numero1; // resultado = 13
resultado = numero2 – 4; // resultado = 1
resultado = numero1 * numero 2; // resultado = 50
resultado = numero1 % numero2; // resultado = 0
numero1 = 9;
numero2 = 5;
resultado = numero1 % numero2; // resultado = 4
```

OPERADORES MATEMÁTICOS

- Los operadores matemáticos también se pueden combinar con el operador de asignación para abreviar su notación:

```
let numero1 = 5;
numero1 += 3; // numero1 = numero1 + 3 = 8
numero1 -= 1; // numero1 = numero1 - 1 = 4
numero1 *= 2; // numero1 = numero1 * 2 = 10
numero1 /= 5; // numero1 = numero1 / 5 = 1
numero1 %= 4; // numero1 = numero1 % 4 = 1
```

OPERADORES RELACIONALES

- mayor que (>), menor que (<), mayor o igual (>=), menor o igual (<=), igual que (==) y distinto de (!=).
- El resultado de todos estos operadores siempre es un valor booleano:

```
let numero1 = 3;
let numero2 = 5;
resultado = numero1 > numero2; // resultado = false
resultado = numero1 < numero2; // resultado = true
numero1 = 5;
numero2 = 5;
resultado = numero1 >= numero2; // resultado = true
resultado = numero1 <= numero2; // resultado = true
resultado = numero1 == numero2; // resultado = true
resultado = numero1 != numero2; // resultado = false
```

OPERADORES RELACIONALES

- Se debe tener especial cuidado con el operador de igualdad (==). Éste se utiliza para comparar el valor de dos variables, por lo que es muy diferente del operador =, que se utiliza para asignar un valor a una variable:

```
// El operador "=" asigna valores
let numero1 = 5;
resultado = numero1 = 3; // numero1 = 3 y resultado = 3
// El operador "==" compara variables
let numero1 = 5;
resultado = numero1 == 3; // numero1 = 5 y resultado = false
```

- Los operadores relacionales también se pueden utilizar con variables de tipo cadena de texto:

```
let texto1 = "hola";
let texto2 = "hola";
let texto3 = "adios";
resultado = texto1 == texto3; // resultado = false
resultado = texto1 != texto2; // resultado = false
resultado = texto3 >= texto2; // resultado = false
```

OPERADORES RELACIONALES

- Cuando se utilizan cadenas de texto, los operadores "mayor que" (>) y "menor que" (<) siguen un razonamiento no intuitivo: se compara letra a letra comenzando desde la izquierda hasta que se encuentre una diferencia entre las dos cadenas de texto. Para determinar si una letra es mayor o menor que otra, las mayúsculas se consideran menores que las minúsculas y las primeras letras del alfabeto son menores que las últimas (a es menor que b, b es menor que c, A es menor que a, etc.).

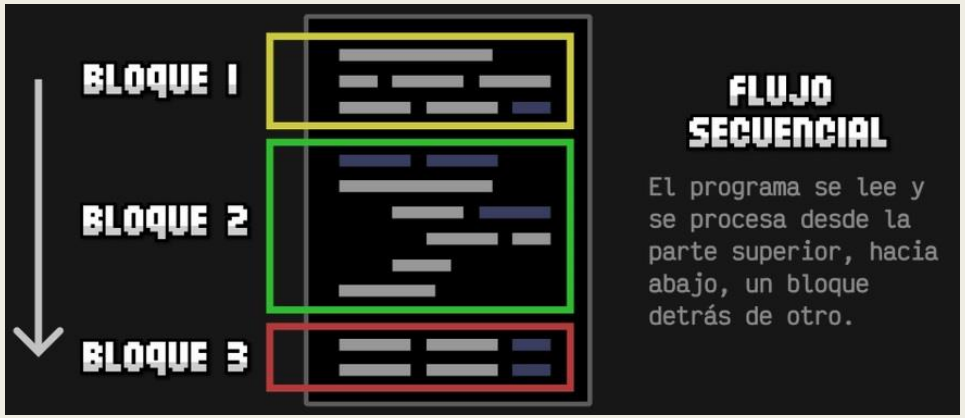
- *Ejercicio 4*

FLUJO DE EJECUCIÓN

- Los programas que se pueden realizar utilizando solamente variables y operadores son una simple sucesión lineal de instrucciones básicas.
- Sin embargo, no se pueden realizar programas que muestren un mensaje si el valor de una variable es igual a un valor determinado y no muestren el mensaje en el resto de casos. Tampoco se puede repetir de forma eficiente una misma instrucción, como por ejemplo sumar un determinado valor a todos los elementos de un array.
- Para realizar este tipo de programas son necesarias las **estructuras de control de flujo**, que son instrucciones del tipo *"si se cumple esta condición, hazlo; si no se cumple, haz esto otro"*. También existen instrucciones del tipo *"repite esto mientras se cumpla esta condición"*.
- Si se utilizan estructuras de control de flujo, los programas dejan de ser una sucesión lineal de instrucciones para convertirse en programas *inteligentes* que pueden tomar decisiones en función del valor de las variables.

FLUJO DE EJECUCIÓN

- En el código de programación, en líneas generales, el flujo del programa se lee desde **arriba hacia abajo, y de izquierda a derecha.**



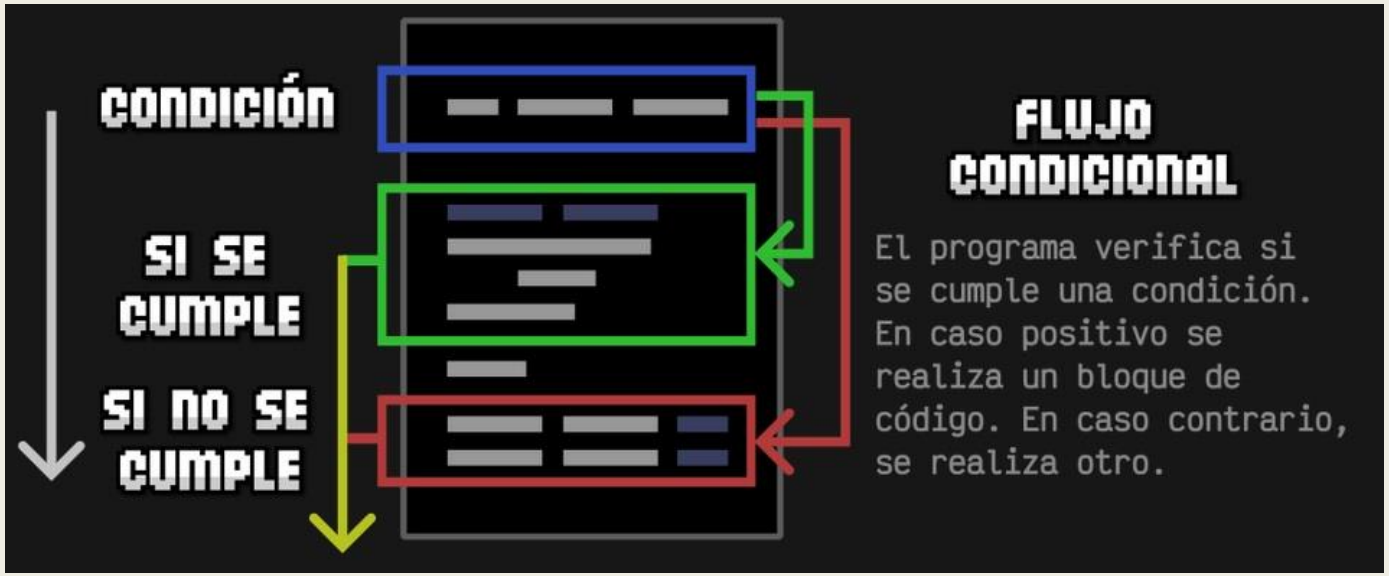
- A medida que vamos escribiendo código se puede complicar o introducir variaciones que hacen que este flujo se vuelva más complejo.

Nombre	Descripción
Condiciones	Bifurcaciones donde el flujo se divide en dos o más caminos.
Bucles	Repeticiones de un código idéntico varias veces o hasta cumplir una condición.
Funciones	Fragmento de código que realiza una tarea, abstrayendo información.
Anidación	Fragmentos o bloques de código dentro de otros.
Estructuras de datos	Lugares o «compartimentos» donde podemos guardar información.

ESTRUCTURAS DE CONTROL DE FLUJO CONDICIONALES

- Fragmentos de código donde se **establece una cierta condición para evaluar si realizar un bloque de código u otro**.

- Al hacer un programa establecemos condiciones o decisiones. Buscamos **que se realice una acción A si se cumple una condición o una acción B si no se cumple.** Este es el primer tipo de estructuras de control que encontraremos.



ESTRUCTURAS DE CONTROL DE FLUJO CONDICIONALES

- Tenemos varias estructuras de control condicionales:

Estructura de control	Descripción
if	**Condición simple**: si ocurre algo, haz lo siguiente...
if/else	**Condición con alternativa**: si ocurre algo, haz esto, sino, haz lo esto otro...
?:	**Operador ternario**: equivalente a If/else, forma abreviada.
switch	**Estructura para casos específicos**: similar a varios If/else anidados.

ESTRUCTURAS DE CONTROL DE FLUJO CONDICIONALES - IF

- Con él podemos indicar en **el programa** que se **tome un camino sólo si se cumple la condición que establezcamos**.

```
let nota = 7;
console.log("He realizado mi examen.");

// Condición (si la nota es mayor o igual a 5)
if (nota ⩾ 5) {
  console.log("¡Estoy aprobado!");
}
```

- **Cuando dentro de las llaves ({ }) sólo tenemos una línea, se pueden omitir dichas llaves.** Aún así, es recomendable ponerlas siempre si tenemos dudas o no estamos seguros.

ESTRUCTURAS DE CONTROL DE FLUJO CONDICIONALES - IF

- La estructura más utilizada en JavaScript y en la mayoría de lenguajes de programación es la **estructura if**. Se emplea para tomar decisiones en función de una condición. Su definición formal es:

```
if(condicion) {
...
}
```

- Si la condición se cumple (es decir, si su valor es true) se ejecutan todas las instrucciones que se encuentran dentro de {...}. Si la condición no se cumple (es decir, si su valor es false) no se ejecuta ninguna instrucción contenida en {...} y el programa continúa ejecutando el resto de instrucciones del script.

```
let mostrarMensaje = true;
if(mostrarMensaje) {
  alert("Hola Mundo");
}
```

ESTRUCTURAS DE CONTROL DE FLUJO CONDICIONALES

- Ejemplo equivalente:

```
var mostrarMensaje = true;
if(mostrarMensaje == true) {
  alert("Hola Mundo");
}
```

- **Cuidado:** las **comparaciones siempre** se realizan con el operador ==, ya que el operador = solamente asigna valores.

```
var mostrarMensaje = true;
// Se comparan los dos valores
if(mostrarMensaje == false) {
  ...
}
// Error - Se asigna el valor "false" a la variable
if(mostrarMensaje = false) {
  ...
}
```

ESTRUCTURAS DE CONTROL DE FLUJO CONDICIONALES - IF

- La condición que controla el *if()* puede combinar los diferentes operadores lógicos y relacionales mostrados anteriormente:

```
let mostrado = false;
if(!mostrado) {
  alert("Es la primera vez que se muestra el mensaje");
}
```

- Los operadores ***AND*** y **OR** permiten encadenar varias condiciones simples para construir condiciones complejas:

```
let mostrado = false;
let usuarioPermiteMensajes = true;
if(!mostrado && usuarioPermiteMensajes) {
  alert("Es la primera vez que se muestra el mensaje");
}
```

ESTRUCTURAS DE CONTROL DE FLUJO CONDICIONALES - IF

- La condición anterior está formada por una operación AND sobre dos variables. A su vez, a la primera variable se le aplica el operador de negación antes de realizar la operación AND. De esta forma, como el valor de mostrado es false, el valor !mostrado sería true.
- Como la variable usuarioPermiteMensajes vale true, el resultado de !mostrado && usuarioPermiteMensajes sería igual a true && true, por lo que el resultado final de la condición del if() sería true y por tanto, se ejecutan las instrucciones que se encuentran dentro del bloque del if().

- *Ejercicio 5*

ESTRUCTURAS DE CONTROL DE FLUJO CONDICIONALES – IF..ELSE

- Se puede dar el caso que queramos establecer una **alternativa a una condición**. Para eso utilizamos el **if** seguido de un **else**. Con esto podemos establecer **una acción A si se cumple la condición**, y **una acción B si no se cumple**.

```
let nota = 7;
console.log("He realizado mi examen. Mi resultado es el siguiente:");

if (nota < 5) {
  // Acción A: nota es menor que 5
  calificacion = "suspendido";
} else {
  // Acción B: Cualquier otro caso diferente a A (nota es mayor o igual que 5)
  calificacion = "aprobado";
}

console.log("Estoy", calificacion);
```

ESTRUCTURAS DE CONTROL DE FLUJO CONDICIONALES – IF..ELSE

```
let nota = 7;
console.log("He realizado mi examen. Mi resultado es el siguiente:");

if (nota < 5) {
  // Acción A (nota es menor que 5)
  calificacion = "suspendido";
}
if (nota >= 5) {
  // Acción B (nota es mayor o igual que 5)
  calificacion = "aprobado";
}

console.log("Estoy", calificacion);
```

- Este código es equivalente al anterior pero no igual:
  - en el anterior sólo existe un if, sólo se realiza la comprobación de una condición.
  - en éste se realizan dos if, se evalúan dos condiciones independientes.
- Este caso no repercute demasiado, pero es **importante** ver que el primer ejemplo realiza menos tareas para conseguir lo mismo.

ESTRUCTURAS DE CONTROL DE FLUJO CONDICIONALES – IF..ELSE

- Si la condición se cumple (es decir, si su valor es `true`) se ejecutan todas las instrucciones que se encuentran dentro del `if()`. Si la condición no se cumple (es decir, si su valor es `false`) se ejecutan todas las instrucciones contenidas en `else { }`.

```
let edad = 18;
if(edad >= 18) {
  alert("Eres mayor de edad");
}
else {
  alert("Todavía eres menor de edad");
}
```

ESTRUCTURAS DE CONTROL DE FLUJO CONDICIONALES – IF..ELSE

- El siguiente ejemplo compara variables de tipo cadena de texto:

```
let nombre = "";
if(nombre == "") {
  alert("Aún no nos has dicho tu nombre");
}
else {
  alert("Hemos guardado tu nombre");
}
```

- En el ejemplo, si la cadena de texto almacenada en la variable `nombre` es vacía (es decir, es igual a `""`) se muestra el mensaje definido en el `if()`. En otro caso, se muestra el mensaje definido en el bloque `else { }`.

ESTRUCTURAS DE CONTROL DE FLUJO CONDICIONALES – IF MÚLTIPLE

- Es posible que necesitemos crear un condicional múltiple con más de 2 condiciones, por ejemplo, para establecer la calificación específica. Para ello, podemos **anidar varios if/else uno dentro de otro**, de la siguiente forma:

```
let nota = 7;
console.log("He realizado mi examen.");

// Condición
if (nota < 5) {
  calificacion = "Insuficiente";
} else if (nota < 6) {
  calificación = "Suficiente";
} else if (nota < 8) {
  calificacion = "Bien";
} else if (nota ⩽ 9) {
  calificacion = "Notable";
} else {
  calificacion = "Sobresaliente";
}

console.log("He obtenido un", calificacion);
```

ESTRUCTURAS DE CONTROL DE FLUJO CONDICIONALES – IF MÚLTIPLE

- La estructura if...else se puede encadenar para realizar varias comprobaciones seguidas:

```
if(edad < 12) {
  alert("Todavía eres muy pequeño");
}
else if(edad < 19) {
  alert("Eres un adolescente");
}
else if(edad < 35) {
  alert("Aun sigues siendo joven");
}
else {
  alert("Piensa en cuidarte un poco más");
}
```

- No es obligatorio que la combinación de estructuras `if...else` acabe con la instrucción `else`, ya que puede terminar con una instrucción de tipo `else if()`.
- *Ejercicio 6*

ESTRUCTURAS DE CONTROL DE FLUJO CONDICIONALES – SWITCH

- Anidar como en el ejemplo anterior varios if suele ser muy poco legible y produce un código repetitivo y feo. En algunos casos se podría utilizar otra estructura de control llamada switch.

- La estructura de control **switch** permite definir **casos específicos a realizar cuando la variable** expuesta como condición **sea igual a los valores** que se especifican a continuación mediante cada **case**:

ESTRUCTURAS DE CONTROL DE FLUJO CONDICIONALES – SWITCH

```
let nota = 7;
console.log("He realizado mi examen. Mi resultado es el siguiente:");

switch (nota) {
  case 10:
    calificacion = "Sobresaliente";
    break;
  case 9:
  case 8:
    calificacion = "Notable";
    break;
  case 7:
  case 6:
    calificacion = "Bien";
    break;
  case 5:
    calificacion = "Suficiente";
    break;
  case 4:
  case 3:
  case 2:
  case 1:
  case 0:
    calificacion = "Insuficiente";
    break;
  default:
    // Cualquier otro caso
    calificacion = "Nota errónea";
    break;
}

console.log("He obtenido un", calificacion);
```

ESTRUCTURAS DE CONTROL DE FLUJO CONDICIONALES – SWITCH

- La sentencia **switch** establece que vamos a realizar **múltiples condiciones** analizando la variable nota.
  - **Cada condición** se establece mediante **un case**, seguido del valor posible de cada caso.
  - El switch comienza evaluando el primer case, y continua con el resto, hacia abajo.
  - Algunos **case** tienen un **break**. Esto hace que deje de evaluar y **se salga del switch**.
  - Los **case que no tienen break**, no se interrumpen, sino que se **salta al siguiente case**.
  - El caso especial **default** es como un else. Si no entra en ninguno de los anteriores, entra en default.

ESTRUCTURAS DE CONTROL DE FLUJO CONDICIONALES – SWITCH

- Este ejemplo no es exactamente equivalente al de if/else. Éste funcionaría si sólo permitimos notas que sean números enteros. En el caso de que nota fuera, por ejemplo, 7.5, mostraría *Nota errónea*, ya que no entraría en los casos anteriores.
- El ejemplo de los if múltiples sí controla casos de números decimales porque establece comparaciones de rangos con mayor o menor (> ó <), cosa que con el **switch no se puede**. El **switch** está indicado **para casos con valores concretos y específicos**.
- Recuerda que **al final de cada caso es necesario indicar un break para terminar y salir del switch**. En el caso que no sea haga, el programa saltará al siguiente case, incluso aunque no se cumpla la condición específica.

ESTRUCTURAS DE CONTROL DE FLUJO CONDICIONALES – OPERADOR TERNARIO

- Es una alternativa al condicional if/else de una forma mucho más compacta y breve, que en muchos casos resulta más legible.
- Hay que tener cuidado. Su sobreutilización puede ser contraproducente y producir un código más difícil de leer.
- La sintaxis de un operador ternario es la siguiente:

```
condición ? valor verdadero : valor falso;
```

```
let nota = 7;
console.log("He realizado mi examen. Mi resultado es el siguiente:");

// Operador ternario: (condición ? verdadero : falso)
let calificacion = nota < 5 ? "suspendido" : "aprobado";

console.log("Estoy", calificacion);
```

ESTRUCTURAS DE CONTROL DE FLUJO CONDICIONALES – OPERADOR TERNARIO

- Observa que **guardamos en calificacion el resultado del operador ternario.**
  - La condición es nota < 5, se escribe al principio, previo al ?.
  - Si la condición es cierta, el ternario devuelve "suspendido".
  - Si la condición es falsa, el ternario devuelve "aprobado".
- La idea del operador ternario es que podemos condensar mucho código y tener un if en una sola línea. Es muy práctico, legible e **ideal para ejemplos pequeños** donde almacenamos la información en una variable para luego utilizarla.

ESTRUCTURAS DE CONTROL DE FLUJO CONDICIONALES – OPERADOR TERNARIO

- Los **operadores ternarios sólo** se recomiendan cuando se trata de **if muy pequeños.** Si intentamos realizar una comprobación de if múltiples con el operador ternario, la sintaxis resultar compleja y difícil de leer.

```
let nota = 7;
console.log("He realizado mi examen.");

let calificacion =
  nota < 5 ? "Insuficiente" :
  nota < 6 ? "Suficiente" :
  nota < 8 ? "Bien" :
  nota ⩽ 9 ? "Notable" :
  "Sobresaliente";

console.log("He obtenido un", calificacion);
```

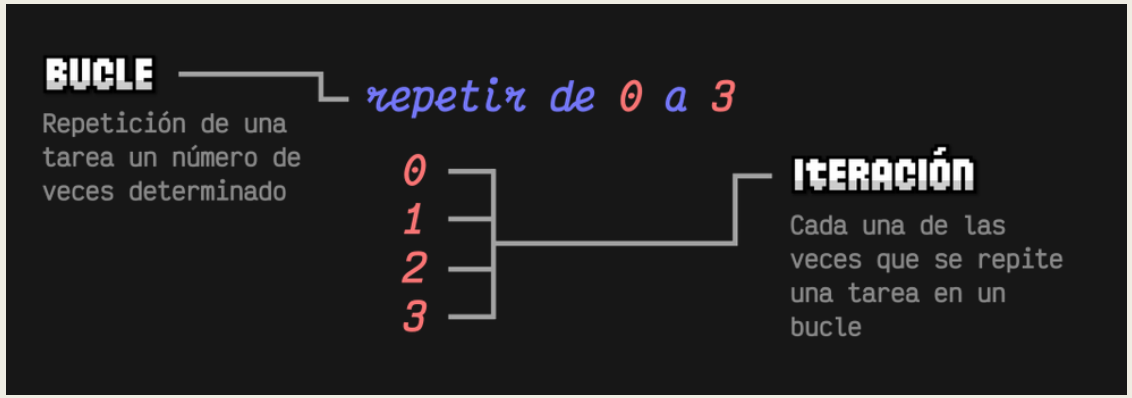
- El "valor falso" del primer operador ternario, es un nuevo operador ternario, que a su vez su valor falso es un nuevo operador ternario, y así con varios casos más.
- Aunque en principio puede resultar interesante porque es bastante compacto y se escribe poco código, se trata de una estructura y sintaxis extraña, por lo que recuerda que **los operadores ternarios anidados no suelen estar muy bien vistos**.

ESTRUCTURAS DE CONTROL DE FLUJO BUCLES

- Son bloques de código que **se pueden ejecutar varias veces dependiendo de una condición**. Esto ahorra tener que repetir código muchas veces, y hace que el programa sea más pequeño y más fácil de leer y escribir.
- El flujo de un bucle es el siguiente:
  1. Evaluar y determinar si una condición es cierta o falsa.
  2. Si es cierta, ejecuta el bloque de código, realiza un cambio relacionado con la condición y vuelve a 1.
  3. Si es falsa, sale del bucle y continua el programa.

Cada una de las **repeticione**s que realiza es un concepto que se denomina **iteración** o vuelta.

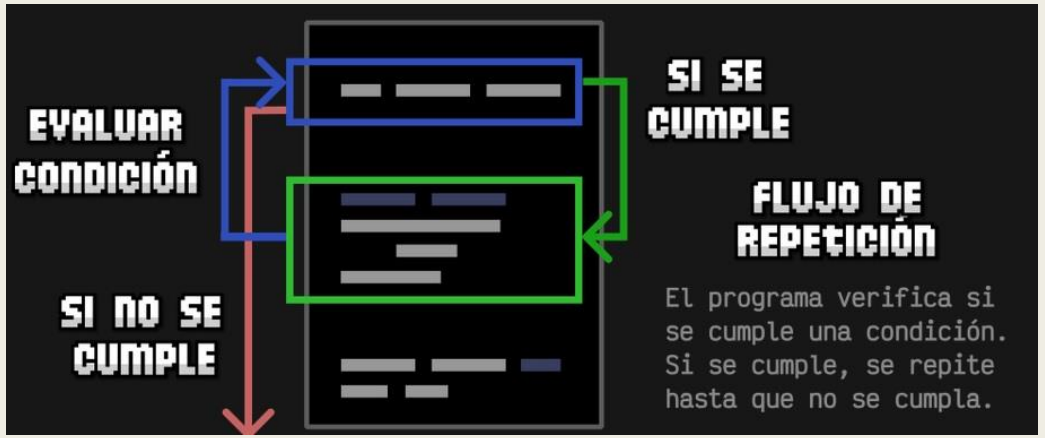
ESTRUCTURAS DE CONTROL DE FLUJO BUCLES



- Tipos de bucles:

Tipo de bucle	Descripción
for	Bucles clásicos por excelencia.
while	Bucles simples.
do..while	Bucles simples que se realizan siempre como mínimo una vez.
for..in	Bucles sobre posiciones de un array.
for..of	Bucles sobre elementos de un array.
Array functions	Bucles específicos sobre arrays.

ESTRUCTURAS DE CONTROL DE FLUJO BUCLES



- **Condición**: en los bucles se evalúa una condición para saber si se debe seguir repitiendo el bucle o se debe finalizar. Si se evalúa como verdadera, se vuelve a repetir el bucle. Si se evalúa como falsa, se finaliza.
- **Iteración**: cada una de las repeticiones de un bucle. ,Por ejemplo, si un bucle se repite de 0 a 3, se dice que hay 4 iteraciones. Cuidado con los bucles, porque en programación se suele empezar a contar en 0.
- **Contador**: muchas veces los bucles incorporan un contador, que no es más que algo que irá guardando un número para contar las repeticiones realizadas, y así finalizar cuando se llegue a otro número concreto. Dicho contador hay que inicializarlo (crearlo y darle un valor) antes de comenzar el bucle.

ESTRUCTURAS DE CONTROL DE FLUJO BUCLES

- **Incremento**: al tener un contador en un bucle, debemos tener una parte donde hagamos un incremento (o un decremento) de él. Si no lo tuviéramos, el contador no cambiaría y sería imposible salir del bucle.
- **Bucle infinito:** situación donde nuestro programa se queda eternamente en bucle y nunca termina. Para evitar que ocurra esto, debemos comprobar que existe un incremento (o decremento) del contador y que en algún momento la condición va a ser falsa y se podrá salir del bucle.

ESTRUCTURAS DE CONTROL DE FLUJO BUCLES - FOR

- Se caracteriza porque se va a repetir, revisando la condición en cada iteración, hasta que no se cumpla la condición propuesta.
- Este bucle es uno de los más utilizados en programación. En Javascript se utiliza igual que en otros lenguajes (Java o C/C++).

```
// for (inicialización; condición; incremento)
for (let i = 0; i < 5; i++) {
  console.log("Valor de i:", i);
}
```

- Analizamos la sintaxis del bucle:
  1. Se separa por ; lo que hay dentro de los paréntesis del for.
  2. Lo primero es la inicialización let i = 0. Esto ocurre sólo una vez antes de empezar el bucle.
  3. Lo segundo es la condición i < 5. Esto se comprueba al principio de cada iteración.
  4. Lo tercero es el incremento i++ (i = i + 1). Esto ocurre al final de cada iteración.

ESTRUCTURAS DE CONTROL DE FLUJO BUCLES - FOR

- Decremento: bucle que en lugar de incrementar su contador, se decrementa (interesa hacer una cuenta atrás):

```
for (let i = 5; i > 0; i--) {
  console.log("Valor de i:", i);
}
```

- Incremento múltiple: no es habitual, pero es posible añadir varias inicializaciones o incrementos en un bucle for separando por comas.

```
for (i = 0, j = 5; i < 5; i++, j--) {
  console.log("Valor de i y j:", i, j);
}
```

```
let i = 0;
let j = 5;

while (i < 5) {
  console.log("Valor de i: ", i);
  console.log("Valor de j: ", j);

  i++;
  j--;
}
```

ESTRUCTURAS DE CONTROL DE FLUJO BUCLES - FOR

- Las estructuras if y if...else no son muy eficientes cuando se desea ejecutar de forma repetitiva una instrucción. Por ejemplo, si se quiere mostrar un mensaje cinco veces, se podría pensar en utilizar el siguiente if:

```
let veces = 0;
if(veces < 4) {
  alert("Mensaje");
veces++;
}
```

- El script solo muestra una vez el mensaje por pantalla. La ejecución de la estructura `if()` no se repite y la comprobación de la condición sólo se realiza una vez, independientemente de que dentro del `if()` se modifique el valor de la variable utilizada en la condición.
- La estructura `for` permite realizar este tipo de repeticiones.

ESTRUCTURAS DE CONTROL DE FLUJO BUCLES - FOR

```
let mensaje = "Hola, estoy dentro de un bucle";
for(let i = 0; i < 5; i++) {
  alert(mensaje);
}
```

- **Parte de la inicialización** del bucle: let i = 0;
- Por tanto, en primer lugar se crea la variable i y se le asigna el valor de 0. Esta zona de inicialización solamente se tiene en consideración justo antes de comenzar a ejecutar el bucle. Las siguientes repeticiones no tienen en cuenta esta parte de inicialización.
- La **zona de condición** del bucle es: i < 5
- Los bucles se siguen ejecutando mientras se cumplan las condiciones y se dejan de ejecutar justo después de comprobar que la condición no se cumple. En este caso, mientras la variable i valga menos de 5 el bucle se ejecuta indefinidamente.

ESTRUCTURAS DE CONTROL DE FLUJO BUCLES - FOR

- Como la variable i se ha inicializado a un valor de 0 y la condición para salir del bucle es que i sea menor que 5, si no se modifica el valor de i de alguna forma, el bucle se repetiría indefinidamente.
- Por ese motivo, es imprescindible indicar la zona de actualización, en la que se modifica el valor de las variables que controlan el bucle: i++
- Normalmente, la variable que controla los bucles for se llama i, ya que recuerda a la palabra índice y su nombre tan corto ahorra mucho tiempo y espacio.
- El ejemplo que mostraba los días de la semana contenidos en un array se puede rehacer de forma más sencilla utilizando la estructura for:

```
let dias = ["Lunes", "Martes", "Miércoles", "Jueves", "Viernes", "Sábado",
"Domingo"];
for(let i=0; i<7; i++) {
  alert(dias[i]);
}
```

- *Ejercicio 7*

ESTRUCTURAS DE CONTROL DE FLUJO BUCLES – FOR..IN

- Una estructura de control derivada de for es la estructura for...in.
- Su definición exacta implica el uso de objetos, que es un elemento de programación avanzada. Se va a presentar la estructura for...in adaptada a su uso en arrays.
- Su definición formal adaptada a los arrays es:

```
for(indice in array) {
  ...
}
```

- Si se quieren recorrer todos los elementos que forman un array, la estructura for...in es la forma más eficiente de hacerlo:

```
let dias = ["Lunes", "Martes", "Miércoles", "Jueves", "Viernes", "Sábado",
"Domingo"];
for(i in dias) {
  alert(dias[i]);
}
```

La variable que se indica como índice es la que se puede utilizar dentro del bucle for...in para acceder a los elementos del array. De esta forma, en la primera repetición del bucle la variable i vale 0 y en la última vale 6.

ESTRUCTURAS DE CONTROL DE FLUJO BUCLES – FOR..OF

- Otra estructura de control derivada de for es la estructura for...of.
- Esta estructura itera sobre los valores de un objeto iterable.

```
for(indice of variableIterable) {
  ...}
```

- La estructura for...of recorre los valores del array:

```
const cars = ["BMW", "Volvo", "Mini"];

let text = "";
for (let x of cars) {
    text += x;
}
console.log("text: "+text);

let language = "JavaScript";

let text = "";
for (let x of language) {
      text += x;
}
```

La variable que se indica como índice, para cada iteración, toma el valor del siguiente elemento en el array.

De esta forma, en la primera repetición del bucle la variable x vale “BMW” y en la última vale “Mini”.

ESTRUCTURAS DE CONTROL DE FLUJO BUCLES – WHILE

```
let i = 0;  // Inicialización de la variable contador

// Condición: Mientras la variable contador sea menor de 5
while (i < 5) {
  console.log("Valor de i:", i);

  i = i + 1; // Incrementamos el valor de i
}
```

- Analizamos la ejecución del código:
  1. Se crea e inicializa la variable i, al valor 0.
  2. Antes de realizar la primera iteración del bucle, comprueba la condición.
  3. Si la condición es verdadera, hace las tareas de dentro del bucle.
     i. Muestra por consola el valor de i.
     ii. Incrementa el valor de i sumando 1 a lo que ya tenía i.
  4. Termina la iteración del bucle. Vuelve al inicio del while.
  5. Vuelve al punto 2) donde comprueba de nuevo la condición del bucle.
  6. Repite hasta que la condición sea falsa. Entonces, sale del bucle y continua el programa.

ESTRUCTURAS DE CONTROL DE FLUJO BUCLES – WHILE

- Para entender mejor el bucle ayuda hacer una traza de lo que haría un programa internamente: ir realizando cada paso, para entenderlo bien.

Iteración del bucle	Valor de i	Descripción	Incremento
Antes de empezar	i = undefined	Antes de comenzar el programa.	
Iteración #1	i = 0	¿(0 < 5)? Verdadero. Mostramos 0 por pantalla.	i = 0 + 1
Iteración #2	i = 1	¿(1 < 5)? Verdadero. Mostramos 1 por pantalla.	i = 1 + 1
Iteración #3	i = 2	¿(2 < 5)? Verdadero. Mostramos 2 por pantalla.	i = 2 + 1
Iteración #4	i = 3	¿(3 < 5)? Verdadero. Mostramos 3 por pantalla.	i = 3 + 1
Iteración #5	i = 4	¿(4 < 5)? Verdadero. Mostramos 4 por pantalla.	i = 4 + 1
Iteración #6	i = 5	¿(5 < 5)? Falso. Salimos del bucle.	

- La operación i = i + 1 es lo que se suele llamar un incremento de una variable. Es muy común simplificarla como i++, que hace exactamente lo mismo: aumenta en 1 su valor.

ESTRUCTURAS DE CONTROL DE FLUJO BUCLES – DO..WHILE

- La diferencia fundamental con el bucle while es que este tipo de bucle **siempre se ejecuta una vez.**

```
let i = 5;

while (i < 5) {
  console.log("Hola a todos");
  i = i + 1;
}

console.log("Bucle finalizado");
```

```
let i = 5;

do {
  console.log("Hola a todos");
  i = i + 1;
} while (i < 5);

console.log("Bucle finalizado");
```

- Analizamos el código del bucle do while:
  1. Comienza con do, en lugar de con while.
  2. El while con la condición se traslada al final del bucle.
  3. El interior del bucle se realiza siempre, y sólo se analiza la condición al terminar el bucle, por lo que, aunque no se cumpla, se va a realizar al menos una vez.
- Interesante cuando queremos establecer un bucle que se realice una vez, independientemente de si cumple o no la condición.

ESTRUCTURAS DE CONTROL DE FLUJO INTERRUMPIR BUCLES

- Al crear bucles, estos van **desde un número inicial a un número final** y así terminan las repeticiones.
- En algunas ocasiones puede interesar hacer **interrupciones o saltos de iteraciones** para conseguir algo más específico.
- **Saltar una iteración:**
  - Imagina un caso donde queremos saltar una iteración concreta. Queremos un bucle que muestre los números del 0 al 10, pero que salte el número 5 y no lo muestre, continuando con el resto.
  - Para eso podemos hacer uso de **continue**: sentencia que al llegar a ella dentro de un bucle, el programa salta y abandona esa iteración, volviendo al principio del bucle:

```
for (let i = 0; i < 11; i++) {
  if (i == 5) {
    continue;
  }

  console.log("Valor de i:", i);
}
```

ESTRUCTURAS DE CONTROL DE FLUJO INTERRUMPIR BUCLES

- Analizamos el código:
  1. Se realizan las iteraciones desde 0 hasta 4.
  2. En la iteración del 5 se entra en el if y se evalúa como verdadera.
     i. Ejecuta el ***continue*** y salta a la siguiente iteración.
  3. Continúa desde la iteración 6 hasta la 10.
- La iteración 5 nunca llega al *console.log*, por lo que no se muestra por pantalla.
- Alternativas:

```
for (let i = 0; i < 11; i++) {
  if (i != 5) {
    console.log("Valor de i:", i);
  }
}
```

```
for (let i = 0; i < 5; i++) {
  console.log("Valor de i:", i);
}

for (let i = 6; i < 11; i++) {
  console.log("Valor de i:", i);
}
```

  1. Si el contenido del bucle fuera más complejo, sería menos legible.
  2. Evita el uso de comprobaciones if anidadas dentro del bucle, pero usa dos bucles distintos.

ESTRUCTURAS DE CONTROL DE FLUJO INTERRUMPIR BUCLES

```
let cadena = "En un lugar de la Mancha de cuyo nombre no quiero acordarme...";
let letras = cadena.split("");
let resultado = "";
for(i in letras) {
	if(letras[i] == 'a') {
		continue;
	}
	else {
		resultado += letras[i];
	}
}
alert(resultado);
// muestra "En un lugr de l Mnch de cuyo nombre no quiero cordrme..."
```

ESTRUCTURAS DE CONTROL DE FLUJO INTERRUMPIR BUCLES

- **Interrupción:**
  - Con **break** que nos permite interrumpir el bucle y abandonarlo. Esto puede ser bastante útil cuando queremos que se abandone el bucle por una condición especial.
- Por ejemplo, cambiemos el *continue* del ejemplo anterior por el *break*:

```
for (let i = 0; i < 11; i++) {
  if (i == 5) {
    break;
  }

  console.log("Valor de i:", i);
}
```

- En este caso, se van a realizar las iteraciones desde el 0 al 4, y cuando lleguemos a la iteración 5, se entrará en el if y como cumple su condición, se hará un break y se abandonará el for, continuando el resto del programa.

ESTRUCTURAS DE CONTROL DE FLUJO INTERRUMPIR BUCLES

```
let i = 0;

while (i < 11) {
  if (i == 5) {
    break;
  }

  console.log("Iteración número ", i);
  i = i + 1;
}

console.log("Bucle finalizado.")
```

- *break* puede utilizarse tanto en *for* como en *while*.
- Estas interrupciones **sólo deben usarse en casos muy específicos**. Siempre es preferible que el bucle sea predecible y no tenga demasiadas situaciones excepcionales que interrumpan el flujo, ya que a la larga son difíciles de modificar y mantener.

ESTRUCTURAS DE CONTROL DE FLUJO INTERRUMPIR BUCLES

```
let cadena = "En un lugar de la Mancha de cuyo nombre no quiero acordarme...";
let letras = cadena.split("");
let resultado = "";
for(i in letras) {
	if(letras[i] == 'a') {
		break;
	}
	else {
		resultado += letras[i];
	}
}
alert(resultado);
// muestra "En un lug"
```

ÁMBITOS O CONTEXTOS

- Ámbito, scope o contexto: es la **zona de alcance que tiene una variable**. Existen dos ámbitos muy bien definidos:
  - **Ámbito global**: existe a lo largo de todo el programa o aplicación.
  - **Ámbito local**: existe sólo en una pequeña región del programa.
- Normalmente, esto se determina fácilmente observando las llaves { y } que abren un nuevo ámbito o contexto.
- Cuando inicializamos una variable al principio del programa y le asignamos un valor, ese valor está disponible a lo largo de todo el programa.
- Por ejemplo, si consultamos el valor de una variable antes de inicializarla, no existe:

```
console.log(e);
// Uncaught ReferenceError: e is not defined
let e = 40;
console.log(e); // 40 (existe porque ya se ha inicializado en la línea anterior)
```

ÁMBITOS O CONTEXTOS

- En el ejemplo anterior, el **ámbito** de la variable **e** comienza **a partir de su inicialización** y "**vive" hasta el final del programa**. A esto se le llama **ámbito global** y es el caso más sencillo.
- Esto se puede ir complicando a medida que vamos abriendo llaves de instrucciones como if, else, for o while. Observa el siguiente ejemplo:

```
let a = 1;
console.log(e);
// Uncaught ReferenceError: e is not defined
if (a == 1) {
    let e = 40;
    console.log(e); // 40, existe
}
console.log(e); // Uncaught ReferenceError: e is not defined
```

- La variable **a** existe en un **ámbito global**, mientras que la variable **e** sólo existe en el interior del if: es un **ámbito local**.

8. ÁMBITOS O CONTEXTOS

ÁMBITOS O CONTEXTOS

- En las versiones modernas de Javascript (a partir de ECMAScript 2015), se introduce la palabra clave **let** en sustitución de la antigua var **para declarar variables** y **const para declarar constantes**. Tanto con let como con const, estaremos utilizando los ámbitos clásicos de programación: ámbito global y ámbito local.
- Veamos otro ejemplo, esta vez utilizando un bucle for:

```
console.log("Antes: ", p);
// En este punto, p no está definida
for (let p = 0; p < 3; p++) {
    console.log("Valor de p: ", p);
// Aquí, p estará definida como 0, 1, 2
}
console.log("Después: ", p);
// En este punto, vuelve a no estar definida
```

ÁMBITOS O CONTEXTOS

- Varios detalles:
  1. Utilizando **let** en el bucle for, la **variable** p **sólo** está **definida dentro del bucle** (**ámbito local**).
  2. Observa que tanto antes como después del bucle, p no existe.
- Aunque omitamos las llaves del for (en este caso es posible porque sólo contiene una línea), los ámbitos siguen existiendo. Hacemos referencia a las llaves porque se ve mejor, pero lo que importa es si se declara la variable dentro del bucle for o no.
- **Declarar variables con var** ya **se considera legacy (obsoleto)** y no debe usarse. Si te interesa saber un poco más por si te lo encuentras, en este enfoque tradicional de Javascript, existen ámbitos diferentes: el **ámbito global** y el **ámbito a nivel de función**.

ÁMBITOS O CONTEXTOS

- Veamos este otro ejemplo (concepto de función):

```
var a = 1;
console.log(a);// Aquí accedemos a la "a" global, que vale 1
function x() {
    console.log(a);// En esta línea el valor de "a" es undefined
    var a = 5;// Aquí creamos una variable "a" a nivel de función
  console.log(a);// Aquí el valor de "a" es 5 (a nivel de función)
  console.log(window.a);//Aquí el valor de "a" es 1 (ámbito global)
}
x(); // Aquí se ejecuta el código de la función x()
console.log(a); // En esta línea el valor de "a" es 1
```

ÁMBITOS O CONTEXTOS

- En el ejemplo anterior vemos que el valor de **a** dentro de una función no es el 1 inicial, sino que estamos en otro ámbito diferente donde la variable a anterior no existe: un **ámbito a nivel de función**. **Mientras estemos dentro de una función, las variables inicializadas en ella estarán en el ámbito de la propia función**.
- Estamos usando el **objeto especial window** para acceder directamente al ámbito global independientemente de dónde nos encontremos. Esto ocurre así porque **las variables globales se almacenan dentro del objeto window** (la pestaña actual del navegador web).

ÁMBITOS O CONTEXTOS

- Si eliminamos el var de la línea var a = 5 del ejemplo anterior, observa la diferencia:

```
var a = 1;
console.log(a); // Aquí accedemos a la "a" global, que vale 1
function x() {
    console.log(a); // En esta línea el valor de "a" es 1
    a = 5; // Aquí creamos una variable "a" en el ámbito anterior
  console.log(a);//Aquí el valor de "a" es 5 (a nivel de función)
  console.log(window.a); // Aquí el valor de "a" es 5 (ámbito global)
}
x(); // Aquí se ejecuta el código de la función x()
console.log(a); // En esta línea el valor de "a" es 5
```

- En lugar de crear una variable en el ámbito de la función, se modifica el valor de la variable **a** a nivel global. Dependiendo de dónde y cómo accedamos a la variable a, obtendremos un valor u otro.

ÁMBITOS O CONTEXTOS

- ¿Qué sucede si una función/estructura de control define una variable local con el mismo nombre que una variable global que ya existe? Las variables locales prevalecen sobre las globales, pero **sólo dentro de la función/estructura de control**:

```
let mensaje = "gana la de fuera";
function muestraMensaje() {
    let mensaje = "gana la de dentro";
    alert(mensaje);
}
alert(mensaje);
muestraMensaje();
alert(mensaje);
```

```
let mensaje = "gana la de fuera";
alert(mensaje);
if (true){
    let mensaje = "gana la de dentro";
    alert(mensaje);
}
alert(mensaje);
```