**CONCEPTOS DE PROGRAMACIÓN:**

* **Elemento** es la referencia a un componente individual de una estructura de datos o de una interfaz de usuario. Por ejemplo, en un array, cada **valor** dentro del **array** se considera un elemento. En HTML, un elemento puede ser una **etiqueta** o componente individual, como un **<div>** o un **<input>.**
* **Valor** es un dato específico que puede ser asignado a una variable o utilizado en una expresión. Los valores pueden ser números, cadenas de texto, booleanos, objetos, arrays, entre otros tipos de datos.
* **Parámetro** es una variable local dentro de una función que recibe un valor específico cuando se llama a la función. Los parámetros permiten a una función aceptar datos de entrada y trabajar con ellos dentro del cuerpo de la función. Una vez que **invocamos** a la **function**, esos **parámetros** pasaran a ser un **argumento**.
* **Objeto** en la programación orientada a objetos, es una instancia de una clase que encapsula datos y comportamiento. Los objetos tienen **atributos** (también llamados **propiedades**) que representan los datos que contienen, y **métodos** que representan las **acciones** que pueden realizar.
* **Método** es una función que está **asociada** a un **objeto** en la programación orientada a objetos. Los métodos se utilizan para **representar el comportamiento o las acciones que un objeto puede realizar**. Por ejemplo, un objeto de tipo **String** en JavaScript tiene **métodos** **como toUpperCase() o substring()** que pueden ser llamados para realizar operaciones en la **cadena de texto.**
* **Condición** es una expresión **booleana** que evalúa si una determinada declaración es verdadera o falsa. Las **condiciones** son **fundamentales** en la programación para controlar el **flujo del programa**, como en las **estructuras de control condicionales** **como if, else if, else**, y **bucles** como **while** y **for**, donde se ejecutan ciertas acciones dependiendo del resultado de la evaluación de la condición.
* **DATOS = PROPIEDADES && COMPORTAMIENTOS = MÉTODOS.**
* **Iteración** es la repetición de un conjunto de instrucciones o bloques de código varias veces, basadas en una **condición**. Es una manera de **ejecutar una tarea repetitiva** hasta que se cumpla la **condición** específicada.

**MODULO 2:**

**Repositorio:** es un espacio de una nube utilizado para almacenar y proyectar los diversos cambios aplicados y desarrollados en los proyectos.

1. **Guardar, mantener y manipular archivos a lo largo del tiempo**
2. **Posibilidad de trabajo colaborativo**
3. **Proyectos guardados en un almacén “repositorio”**

Un commit no es más que una captura instantánea de los cambios preparados en ese momento del proyecto. Las instantáneas confirmadas pueden considerarse como versiones "seguras" de un proyecto.

A continuación de compartimos los comandos más usados con git:

1. **git init:** comando para inicializar un repositorio local.
2. **git** **add:** comando para que nuestro repositorio sepa de la existencia de un archivo o de sus últimos cambios, no almacena las actualizaciones de forma definitiva, únicamente las guarda en algo que conocemos como “Staging Area” (área de montaje o ensayo). Se puede armar como git add ArchivoEjemplo.js (solo agrega ese archivo) o git add (agrega todos los archivos modificados de la carpeta donde estas con el punto.
3. **git commit -m "mensaje":** comando para almacenar definitivamente todos los cambios que por ahora viven en el staging área. En el mensaje ponemos el mensaje que explica los cambios commiteados.
4. **git push:** comando para mandar nuestros commits a un servidor remoto.
5. **git status:** ofrece una descripción del estado de los archivos.
6. **git pull:** sirve para recibir cambios de repositorio remoto a local.

Recuerda que Git nunca cambiará las versiones seguras de tu proyecto, a no ser que se lo pidas expresamente. De esta forma, estamos guardando el estado actual de nuestro proyecto, para volver a visitarlo de ser necesario y por supuesto para mantenerlo a lo largo del tiempo.

**Posibles errores**

A continuación veremos algunos de los errores más típicos que pueden encontrarte cuando trabajas con Git y GitHub.

**Agregar un repositorio dentro de otro:**

✅ Antes de clonar un repositorio, fíjate que no estés parado dentro de un repositorio. Utiliza el comando ***"cd .."*** para salir de esa carpeta contenedora.

**git init | Carpeta del sistema**

✅ Si alguna vez inicializas un repositorio de git dentro de una carpeta del sistema de tu computadora, ¡Descuida! Solo tienes que entrar a la terminal, pararte dentro de esta carpeta y utilizar el comando: "*rm -rf .git"*.

No funciona el comando: "code ."

**Hay dos formas en las que puedes abrir el repositorio en tu Visual Studio Code:**

1. Dirígete a la carpeta del repositorio. Has click derecho encima de la carpeta, y preciona: "*Abrir con Code*".

2. Abre el Visual Studio Code. Busca la opción "archivo"/"file" en la barra de herramientas. Has click sobre el botón que dice "abrir carpeta". Localiza tu repositorio y simplemente ábrelo.

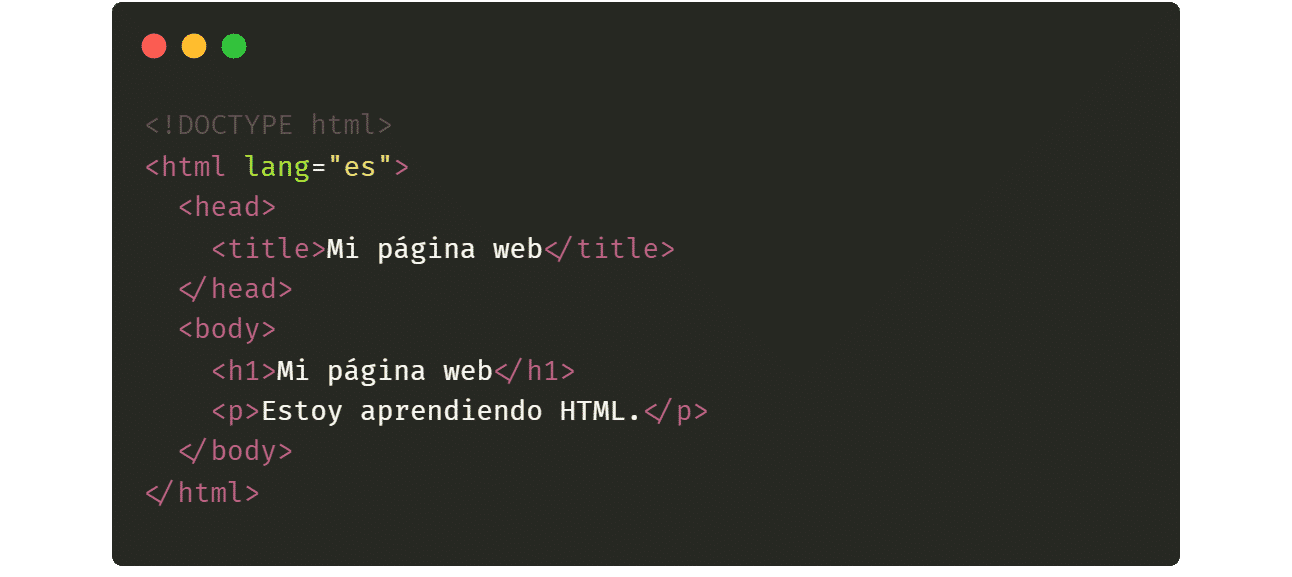
**MODULO 3:**

**HTML.**

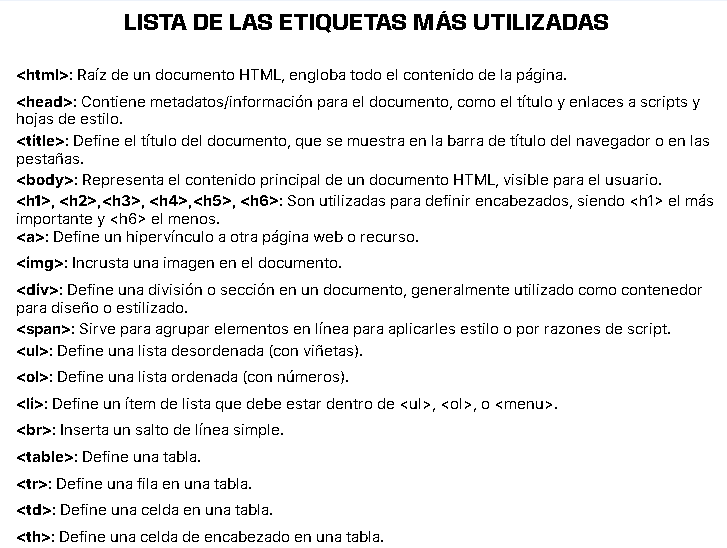
**Definición: HyperText Markup Language** (HTML) es un lenguaje de marcado (no de programación) utilizado para crear páginas web. Este tipo de código te permitirá crear una página web sencilla.

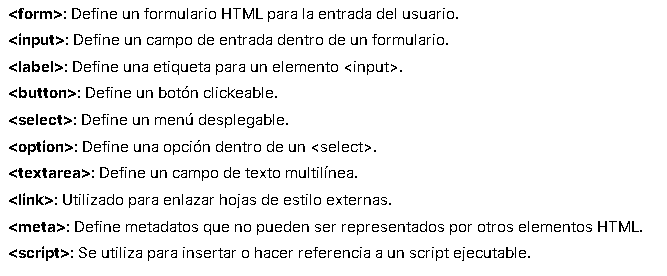
**Estructura de un archivo HTML**

**\*IDENTACIÓN: espacio dejado al comienzo de cada línea.**



**Elementos = etiquetas** con las que identificamos y estructuramos la página.





**Etiquetas de Contenido HTML y Funciones.**

Son etiquetas que nos sirven para diseccionar e identificar cada sector de la página para agilizar su comprensión y su desarrollo.

**Loren:** es una función que genera texto aleatorio para ocupar el espacio.

**<span>:** etiqueta para textos cortos los cuales deseamos en un mismo renglón.

**<hr />:** línea que separa parte del contenido de la página.

**<header>:** área del encabezado.

**<footer>:** barra debajo del contenido principal; utilizado mayormente para ubicar referencias, reseñas, enlaces, etc.

**<section>:** como dice su nombre, para aclarar y formar secciones.

**<article>:** otra etiqueta para diseccionar pero algo más pequeño.

**<div>:** “caja universal” para agrupar elementos.

<section>

        <div>

            <h2>Compártenos tu opinión.</h2>

            <form action="/submit-opinion" method="post">

                <h3><label for="opinion">Tú opinión:</label></h3>

                <textarea class="opinion" name="opinion" rows="4" cols="50" placeholder="Escribe tu opinión aquí..."></textarea>

                <input type="submit" value="Submit Opinion" class="submit">

            </form>

        </div>

    </section>

En este ejemplo:

• El **<textarea>** se utiliza para crear un cuadro de entrada de texto de varias líneas. Los atributos **rows** y **cols** determinan el número visible de filas y columnas.

• El **<label>** está asociado con el **for** asociado al **<textarea>**. Esto ayuda a mejorar la accesibilidad al permitir que los lectores de pantalla sepan a qué entrada corresponde la etiqueta.

• El **placeholder** proporciona una pista al usuario sobre el tipo de entrada que se espera.

• El **<form>** envuelve los elementos de entrada, y los atributos **action** y **method** especifican dónde se deben enviar los datos del formulario y el método HTTP que se utilizará al enviar el formulario. En este ejemplo, los datos del formulario se envían a **"/submit-opinion"** mediante el método **POST**.

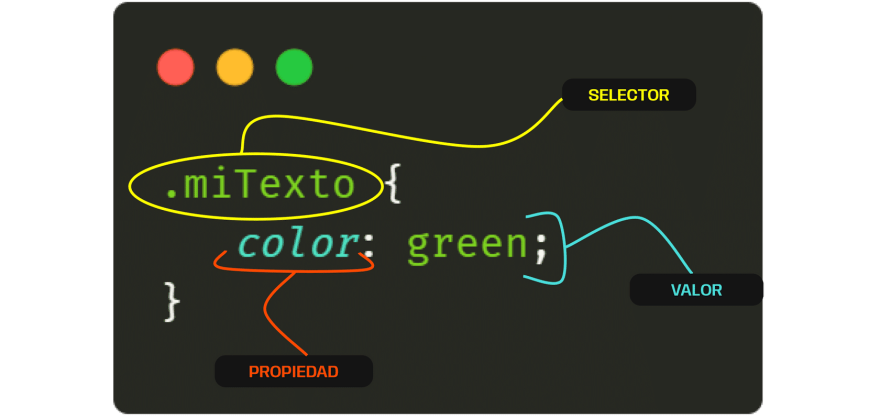
• El **<input>** **con type="submit"** crea un botón en el que los usuarios pueden hacer clic para enviar el formulario.

**Modulo 4:**  
**CSS:**

**Cascading Style Sheets**(hojas de estilo en cascada). Este permite a los desarrolladores web controlar el diseño y la apariencia de sus sitios web separando el contenido (como el texto y las imágenes) de la presentación (como los colores, las fuentes y el diseño).

Los atributos son propiedades o "configuraciones" que nos permitirán modificar ese elemento de alguna manera (por ejemplo, los estilos).

El atributo **style** seguido de una propiedad y su valor, nos permite cambiar el color del fondo y de un texto.

Es más recomendado tener un archivo específicamente de estilos de CSS, ya que de esta forma el código estará mejor organizado.  
Utilizaremos un **selector**llamado **clase**, que nos permitirá elegir a qué elemento HTML queremos darle estilos.

El **selector**será una referencia que nos permitirá seleccionar un elemento HTML. La **propiedad**dirá qué característica queremos cambiar de ese elemento. Por último, el **valor**dirá cuál será el estilo que tendrá esa propiedad.

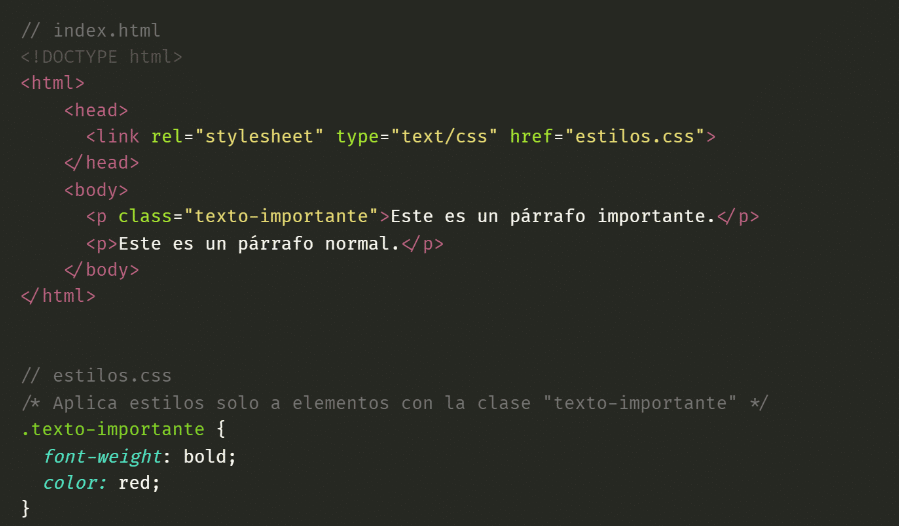
Para **conectar** un archivo HTML con un CSS lo que se utiliza es dentro del código **<head>** se inserta lo siguiente: **<link rel="stylesheet" type="text/css" href="./estilos.css" />**

**Una vez hecho esto, colocaremos el atributo “class” dentro de los selectores/etiquetas que utilizaremos y en las que así aplicaremos los estilos de nuestro archivo CSS.**

**Selectores de CSS:**

**Más populares:**

1. **Clase** se utiliza para seleccionar elementos HTML que tienen una clase específica. Se define mediante un punto seguido del nombre de la clase que deseas seleccionar.



1. **Etiqueta** se utiliza para seleccionar todos los elementos HTML que coinciden con un nombre de etiqueta específico. Es uno de los selectores más simples y selecciona todos los elementos que tengan esa etiqueta en su estructura HTML.



1. **ID**se utiliza para seleccionar un elemento HTML específico en función de su atributo de ID. Se define precediendo el nombre del ID con un signo de numeral (#) y es la clase que ante cualquier conflicto con los elementos que hayamos elegido para aplicarlo será primeramente seleccionada y procesada por el navegador

****

1. **Universal Selector (\*)**: Selecciona todos los elementos de la página.
2. **Tipo Selector (element)**: Selecciona todos los elementos del tipo especificado.
3. **Clase Selector (.class)**: Selecciona todos los elementos que tienen la clase especificada.
4. **ID Selector (#id)**: Selecciona un único elemento con el ID especificado.
5. **Descendant Selector (ancestor descendant)**: Selecciona todos los descendientes de un elemento padre dado.
6. **Child Selector (parent > child)**: Selecciona todos los hijos directos de un elemento padre dado.
7. **Adjacent Sibling Selector (prev + next)**: Selecciona el elemento siguiente inmediato después de otro elemento.
8. **Attribute Selector ([attribute=value])**: Selecciona elementos con un atributo específico y un valor específico.
9. **Pseudo-class Selector (:pseudo-class)**: Selecciona elementos basados en su estado o posición.
10. **Pseudo-element Selector (::pseudo-element)**: Selecciona partes específicas de un elemento, como el primer línea o primer letter.
11. **Grouping Selector (selector1, selector2)**: Agrupa varios selectores para aplicar las mismas reglas a múltiples elementos.



**Propiedades de CSS:**

***Viewport width (vw):*** ancho del punto de vista de la pantalla en la que estemos.

**BORDER + WIDTH = VW + Margin (distanciamiento del margen) = AUTO.**

Hay diferentes formas de medir el tamaño de la fuente. Tales como **pixeles**, **em** (unidad relativa del tamaño de fuente base), **decimales**, **vw**,entre otros.

**Padding:** es la propiedad para crear un margen interno sobre el contenido; para “despegar” el contenido dese lo interno del **border**.

**Fit-content:** encuadrar el contenido según su **width**.

**Border-radius:** borde redondeado.

**Transition:** tiempo definido en el que la **clase** desarrollara X acción.

**Flexbox:** valor dentro de CSS destinado al **display**, permitiendo posicionar, alinear y direccionar los elementos HTML en la pantalla sencillamente.

Una de las maneras más comunes de eliminar las configuraciones predeterminadas de los navegadores ante una página web es crear un archivo CSS, relacionado a HTML, quitando las configuraciones con nuestros comandos.

Cuando queremos trabajar con **Flexbox** es importante aclarar, definir y operar únicamente sobre el **Flexbox** que vayamos a atender.

**Display: flex;** establecer que trabajaremos en flexbox con esa clase.

**Flex-direction (horizontal): row (**misma línea del texto**) / row-reverse / column / column-reverse;**

**Justify-content (vertical) : flex-top / right / etc** o **space-around / between / evenly;** alrededor/entre/bordeynextelement.

**Gap:** espacio entre un elemento y otro.

**Align-items:** alinear elementos según lo establecido, como **flex-start**, **flex-end**, **center**, **baseline**, **stretch.**

Cuando es una **columna**, **flex-direction** mueve en **horizontal** y **justify-content** en **vertical.**

A veces al no ser suficiente poner en fila o columna los elementos se utiliza el comando **order:** 1 u otros para mover/alinear el elemento seleccionado.

**Align-self:** usa las mismas funciones y realiza lo mismo que **align-items** pero para el elemento especificado.

**Flex-direction:** Esta propiedad establece la dirección en la que los elementos flexibles dentro del contenedor flexible son colocados en el eje principal. Los valores comunes para esta propiedad son:

**row**: Los elementos se colocan en la misma dirección del texto.

**row-reverse:** Los elementos se colocan en dirección opuesta al texto.

**column:** Los elementos se colocan de arriba hacia abajo.

**column-reverse:** Los elementos se colocan de abajo hacia arriba.

**Flex-wrap:** Esta propiedad establece si los elementos flexibles dentro del contenedor flexible pueden envolverse (o no) en múltiples líneas. Los valores comunes para esta propiedad son:

**nowrap:** Los elementos se ajustan en una sola línea (no se envuelven).

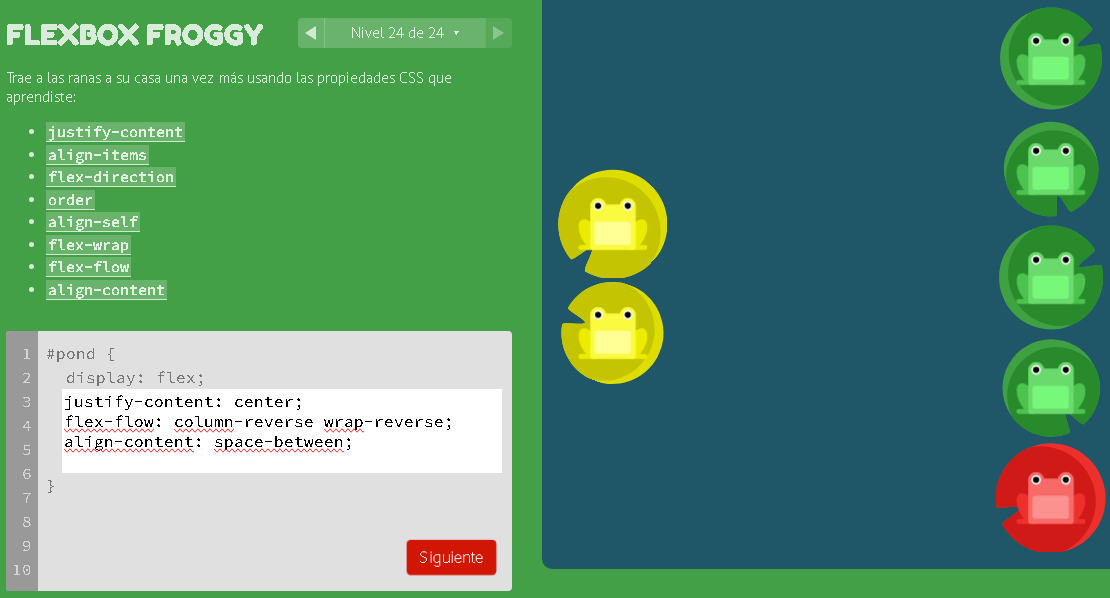
**wrap:** Los elementos se envuelven en líneas adicionales si es necesario.

**wrap-reverse:** Los elementos se envuelven en líneas adicionales, pero en sentido inverso.

**Flex-flow:** combinación de **Flex-Direction** y **Flex-wrap.**

**Align-content: flex-start / flex-end / center / space-between / space-around / stretch;** este comando define el espacio entre las líneas, no en los **ítems**. Si hay una sola línea, no tiene efecto.

Lo que veremos que hacemos en la siguiente imagen es: **a)** centrar la imagen; **b)** revertir la dirección de la columna y de las líneas adicionales; **c)** dar espacio entre los elementos.



Las **pseudoclases**de CSS se utilizan para hacer que un elemento HTML tenga un **comportamiento** **concreto**. Estas se escriben luego de un selector y antecedidos por doble puntos **(:)**. Tales como **hover**, el cual genera una animación al pasar el mouse sobre el elemento. La animación puede ser cualquiera, como cambiar de color o el tamaño.

**Lenguaje de Programación:**

Podemos denominarlo como un lenguaje formal (o artificial, es decir, un lenguaje con reglas gramaticales bien definidas) que le proporciona a una persona, en este caso el programador, la capacidad de escribir (o programar) una serie de instrucciones o secuencias de órdenes en forma de algoritmos con el fin de controlar el comportamiento físico o lógico de un sistema informático, de manera que se puedan obtener diversas clases de datos o ejecutar determinadas tareas.

Los más populares últimamente son:

1. JS / TS.
2. Python.
3. Java.
4. C#.
5. PHP.
6. C / C++.
7. Ruby.
8. Go.

Como tales, **C** y **Java** son lenguajes compilados, pasando por el **compilador**, filtrando que todo esté correcto para **instalar o aplicar** lo programado.

**Módulo 5: Java Script.**

Hoy en día los lenguajes más utilizados por las empresas son Python, Java y Java Script.

**JavaScript** es un **lenguaje de programación interpretado** -como **Phyton**- dinámico que tiene la capacidad de ser utilizado en variados dispositivos, desde ordenadores hasta teléfonos inteligentes.

Basado en objetos, dinámico y amigable para principiantes, JavaScript es el pilar de múltiples herramientas que sus fanáticos fueron ampliando con nuevas funcionalidades a lo largo del tiempo.

Si bien era conocido como un lenguaje de programación solamente de **front-end**, en los últimos años tomó una posición muy fuerte también en el lado del servidor, a través de Node.js. Esto significa que alguien puede saber programar todo el stack, esto es: tanto el **front** como el **back-end**, ¡aprendiendo un único lenguaje!

Es un lenguaje en el que se usa **client-side**. Lo cual nos permite crear la **interfaz** de los usuarios en la web, y realizar las tareas del server.

Además, es un lenguaje de **tipado dinámico**, lo que significa que no necesitamos declarar y establecer un tipo –como número o cadena de caracteres– específico e inflexible para las variables, lo cual da mucha libertad y sencillez, en especial al aprender el lenguaje. Técnicamente, JavaScript es un **lenguaje de programación interpretado**, por lo que no es necesario compilar los programas para ejecutarlos.

**Velocidad 🏃**

En esencia, JavaScript tiende a ser muy rápido. A menudo se ejecuta inmediatamente en el navegador; en tanto no requiera recursos externos, JavaScript no tiene permitido retrasarse por llamados del servidor backend.

**Popularidad 🏆**

JavaScript está en todos lados y se espera que su popularidad siga en alza. Si navegas por recursos y plataformas tales como StackOverflow y GitHub verás la cantidad de proyectos desarrollados en base a este lenguaje. Es más, con la llegada de Node.js, se ha incrementado su uso en el back-end y por lo tanto incorporado cada vez con más frecuencia en los tech stack de todo tipo de startups y compañías.

**Server Load 💻**

JavaScript trabaja del lado del cliente lo que reduce la demanda de servidores en general.

**Compatibilidad 🔁**

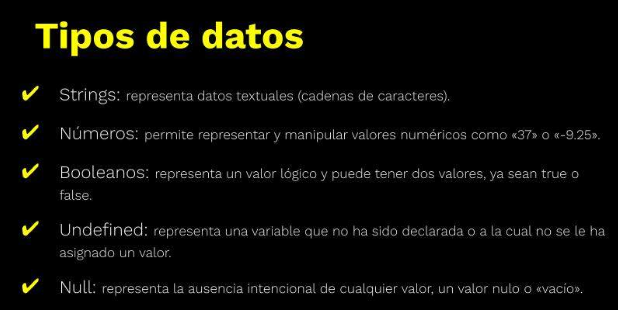
A diferencia de otros lenguajes scripting, JavaScript puede ser usado en cualquier página web y en diferentes tipos de aplicaciones gracias al soporte que ofrecen otros lenguajes como Pearl y PHP.

**Simplicidad 💆‍♀️**

La sintaxis de JavaScript está inspirada en Java, y en relación a otros lenguajes de programación igualmente populares, es relativamente sencillo de aprender.

**Versatilidad 🚀**

Hay muchos métodos para usar JavaScript mediante servidores Node.js.  Si tú estás en bootstrap Node.js con Express, puedes usar un documento database como MongoDB, y usar JavaScript en el frontend para clientes.  Una gran ventaja de JavaScript es que es el único lenguaje que te permite trabajar en modo Full Stack.



**SÍMBOLOS y FUNCIONES:**

Una **function ()** es un bloque de código reutilizable que realiza una tarea específica cada vez que es **“llamada”**. Lo que tenemos en sus **{}** **llaves** son su **sentencia** y **scope** (alcance).

**==** Significa **equivalente / igualdad.**

**===** Significa **equivalente estricto** o **igualdad estricta.** Comparamos tanto el **value** como el **typeOf**.

**!** Significa **negación.**

**DIFERENCIA OBJETO / MÉTODO:**

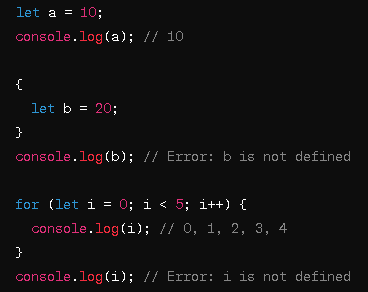
**-Objeto:** en JS es es un entidad independiente con propiedades y tipos. Compáralo con una taza, por ejemplo. Una taza es un objeto con propiedades. Una taza tiene un color, un diseño, un peso, un material del que está hecha, etc.

**-Método:** en JS es un bloque de código que contiene una serie de instrucciones. Tales como **.pow()**.

**Console.log ()** función principal: demostrar en la terminal o consola lo escrito entre los paréntesis.

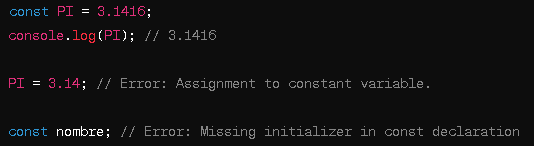
**Statements:** es una comparación que me da como resultado un booleano. **Algoritmo:** secuencia de órdenes definidas y sistemáticas que cumplen “cierta” tarea.

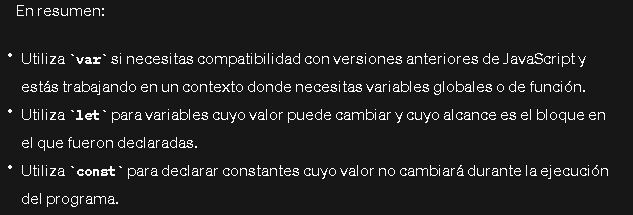
Las funciones pueden ser lineales, constantes, lineales o exponenciales. Se habla de 2 ejes: recursos y tiempos. Dependiendo qué necesites hacer, buscarás una complejidad u otra.

**Variables:** contenedores que guardan un valor que puede llegar a modificarse en el desarrollo.

**LET-VAR-CONST.**  
**LAS 3** son diferentes maneras de definir o crear una variable, sólo que cada una tiene su alcance y su comportamiento:  
**-Var:** tienen un alcance global o de función, es decir que podemos acceder tanto desde dentro de la función como desde fuera de la función o bloque donde fueron declaradas. Pueden ser re-declaradas o configuradas dentro desde su mismo bloque, pero puede ocurrir el problema de “hoisting” que, al “llamarlas” desde fuera luego de su declaración ocurra un error.

**-Let:** solo existe y se “imprime” dentro de su bloque, no pueden ser re-declaradas dentro de tal, no se les puede acceder antes de su declaración y no tienen problemas de “hoisting”.

**-Const:** es utilizada para crear **constantes**, es decir valores que no serán cambiados durante su ejecución. Tampoco sale de su bloque, no tiene problemas de “hoisting” pero deben ser inicializadas al momento de su declaración y no pueden ser reasignadas.

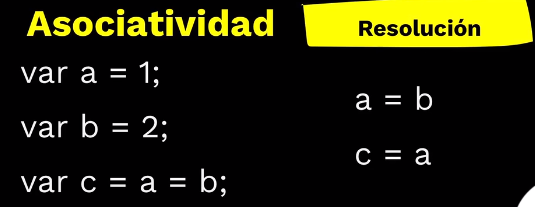


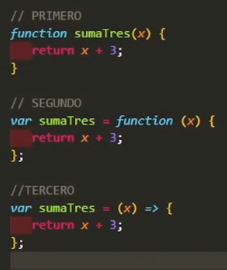
**MÉTODO LENGTH**

Este es un método que solo se puede aplicar al tipo de dato string (más adelante veremos que también lo podemos utilizar en arreglos). En este ejemplo podemos ver que si lo intentamos aplicar en un número o en un booleano, nos devuelve error.

**Operadores.**

Los operadores tienen precedencia, lo cual es que unos son operados antes que otros. Los básicos irían en el orden de **multiplicar (\*)**, **porcentaje o resto (%)**, **dividir (/)**, **Sumar (+)**, **restar (-)**. Así como los operadores de comparación son -sin importar- el orden ya: **mayor que (>), menor que (<), mayor o igual (>=), menor o igual (<=), igual == (se representa con doble igual), distinto de (!=), incremento (+=) y decremento (-=).**

La asociatividad dependerá de qué tipo de operación se realice. Tal como vemos en la imagen que va de **derecha a izquierda**, mientras que por ejemplo en la división, seguiría lo común de **izquierda a derecha**.

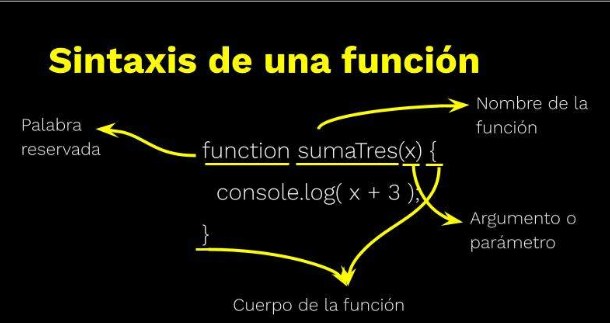


Un operador importante es el **operador del módulo**, usualmente representado por el % en la mayoría de los lenguajes de programación, nos entrega el residuo de una división

Existen diferentes maneras de escribir una función, con diferentes sintaxis.

Algo a tener en cuenta dentro de JS es que los **operadores** tienen **procedencia**, es decir, **tienen una jerarquía** en la que específicas **funciones** son **operadas antes que otras**

En la siguiente imagen, veremos 3 ejemplos diferentes, en el cual el más viable es el 2.

**Console.log():** es una función para imprimir y probar la función de lo programado en la consola. Es utilizado como “método de prueba”. No puede quedar en ningún programa.

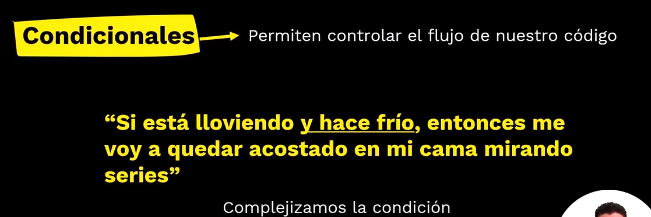
**Return:** es una función que retornará en la terminal el valor que queramos comprobar de nuestra edición. Y algo a tener en claro: todo lo escrito **por debajo** del **return** no se usará o tomará, ya que una vez **retorna** la función, todo lo demás no se toma.

Los **parámetros** son palabras claves que nos permitirán introducir info externa dentro de una **función**.

Existen 3 tipos de **nomenclaturas** en JS:

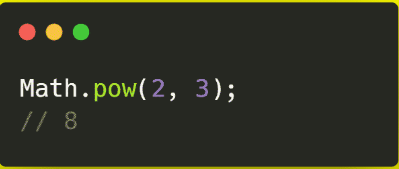
1. **CamelCase:** Este modo de nombrar implica la unión de dos o más palabras sin espacios entre ellas, pero diferenciadas por una letra mayúscula inicial a partir de la segunda palabra, por ejemplo: holaMundo / funcionEdadAños.
2. **PascalCase:** Es similar a camelCase pero la primera letra de la frase también se inicia con mayúscula, al igual que cada palabra. Se mantiene la práctica de no tener espacios entre palabras y cada palabra siempre empieza con su primera letra mayúscula. Por ejemplo: HolaMundo / FuncionEdadAños
3. **Snake-case:** La nomenclatura snake\_case  es definida de esta forma porque siempre la usamos sobre el piso. Las letras siempre  serán minúsculas y las  palabras están separadas por un guion bajo de esta forma: hola\_mundo / funcion\_edad\_años

Existen también los **condicionales (If/else), que** son estructuras fundamentales para la resolución de problemas.



**Console.log** tiene sus métodos propios para operar. Otro que tiene lo mismo es **Math**:

1. **Math.pow :** Este método nos permite potenciar un número. Por ejemplo, 2 elevado al cubo, sería igual a multiplicar 2 x 2 x 2. Es decir, 8. El primer número es la base, y el segundo será la potencia.

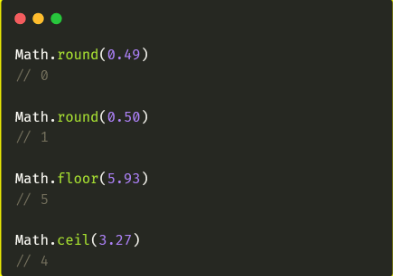


1. A veces necesitamos redondear un número decimal, ya que muchas veces las operaciones matemáticas pueden resultar en números con coma. Para esto, JavaScript nos ofrece tres métodos

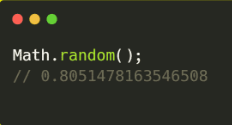
**Math.round**redondeará el número decimal al entero más próximo. Es decir que, si tenemos 0.49, el resultado será 0. En cambio, si tenemos 0.50, el resultado será 1.

**Math.floor**redondeará el número decimal al entero de menor valor. Por ejemplo, si tenemos el número 5.93, el resultado será 5.

**Math.ceil**este método, al contrario que el anterior, redondeará un número al siguiente entero. Por ejemplo, si tenemos 3.27 el resultado será 4.



1. **Math.random** Este último método nos permitirá generar un número aleatorio. Es importante tener en cuenta que el número que se generará es decimal, y puede ser cualquiera entre el 0 y el 1.



Existen también **Math.max()** y **Math.min()** los cuales brindarán el número máximo y mínimo de lo disponible.

**Programación Declarativa:** son métodos que declaran definitivamente lo declarado en el comando con la función necesaria.

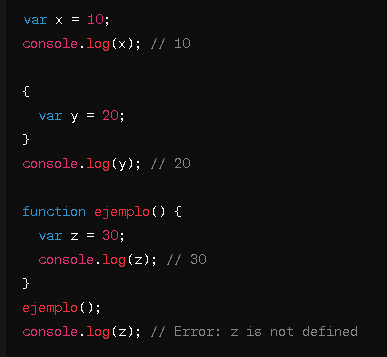
**Programación Imperativa:** es la programación en la que nosotros definimos y aclaramos los parámetros y comandos de una función.



**Cláusula de guardia:** es una forma de trabajar las condiciones a través de la negación.

Una manera de saber si un número es **par** es **% 2 === 0.** La misma fórmula se puede utilizar para saber si X número es % (divisible) por **3 (ejemplo) === 0.**

**If:** es para crear los **booleanos**, para dar las posibilidades de **true**, **false** o **null/undefined** según los **parámetros** y **valores** aplicados en el proyecto; aclarando con un **else**.

**Booleanos:** son valores que son **true** y **false** para comparar y comprobar valores aplicados en un código.

En la imagen podemos comprobar cómo y cuántas veces es necesario utilizar **if, else y else if** para aclarar el código según los valores a aplicarle. **“Si el número es mayor que 0 “es positivo”; si es menor “es negativo”; pero si es 0 *false***.

Para medir áreas se utilizan diferentes formas. Como por ej. :  
Cuadrado o perímetro = **lado \* 4;**Rectángulo = **alto \* ancho;**Triángulo = **(base \* altura) / 2;**

Una función importante y rara a tener en cuenta es si **el número es primo**:

function esNumeroPrimo(numero) {

  // La función recibe un argumento "numero".

  // Determina si este corresponde a un número primo.

  // De ser así, retorna true.

  // De lo contrario, retorna false.

  // IMPORTANTE: Recuerda que un número primo es aquel que

  // solo es divisible por sí mismo y por 1.

  // Tu código:

  if (numero <= 1) { //si el número es menor o igual a 1...

    return false;

  }

    for (let i = 2; i <= numero / 2; i++) { //si el número vale 0...

    if (numero % i === 0) {

      return false;

    }

  }

    return true;

}

module.exports = esNumeroPrimo;

**Operadores Lógicos.**



El **&& / AND** nos permite conjugar 2 o más condiciones al mismo tiempo:

function mayorMenorYPar(num) {

    if(num > 5 && num < 10 && num % 2===0){

        console.log(true);

    } else {

        console.log(false);

    }

}

mayorMenorYPar(7)

Para que el resultado sea **true**, en este operador todas las condiciones deben ser cumplidas.

**|| / Or** nos permite conjugar condiciones para distinguir los resultados según la condición.

function operadorOr(str){

    if(str === 'Henry' || str.length < 2 ){

        console.log(true);

    } else {

        console.log(false);

    }

}

operadorOr('H');

Este operador para que sea **true** necesita que mínimo una condición se cumpla.

**! / NOT** nos permite verificar condiciones opuestas a las normales. Lo que hace el símbolo es preguntar por el valor opuesto al parámetro establecido.

function negacion(permiso){

    if(permiso !== true) console.log('Tiene permiso en la terminal')

}

negacion(false);

Ahora realizamos una función con todos los operadores lógicos juntos:

function condicionCompleja(num){

    if (num > 9 && num % 2 === 0 || num === 3){

    console.log(true);

} else {

    console.log(false);

    }

}

condicionCompleja(10); //true por ser mayor a 9

condicionCompleja(6); //false por no cumplir las condiciones AND

condicionCompleja(3); //true por ser igual a 3

condicionCompleja(5); //false por no cumplir ninguna condición

1. Las expresiones se evalúan en orden, y la computadora omitirá cualquier expresión redundante. En una declaración **&&**, si la primera expresión es false, la segunda expresión no se evaluará porque AMBAS expresiones deben ser true.
2. Lo mismo para la declaración**||**. Si la primera expresión es verdadero, la segunda no se evaluará porque solo debe haber una declaración verdadera para cumplir con los requisitos del operador.
3. Utiliza **paréntesis**. Como vimos en el segundo ejemplo de la negación **!**, utilizamos paréntesis para evaluar PRIMERO lo que estaba dentro de ellos, y luego aplicamos la negación. Podemos ajustar cualquier expresión entre paréntesis y se evaluará antes que toda la expresión en conjunto.

**Veracidad.**

En JS existe la **veracidad**, es decir, referirse a la verdad sobre si algo es **true** o **false**. La forma de procesar sigue siendo binaria, 1 y 0, por lo que en todo lo del lenguaje habrá una data para resumir en estos dos valores.

Aquí siguen unos ejemplos para probar desde la terminal:  
**1 // true**

**0 // false**

**-1 // true**

**true // true**

**false // false**

**'string' // true**

**null // false**

**undefined // false**

**[] // true**

**Recursos Adicionales.**

La declaración **switch** evalúa una expresión, comparando el valor de esa expresión con una instancia case, y ejecuta declaraciones asociadas a ese case. Por ejemplo:

let expr = "Naranjas";

switch (expr) {

    case "Naranjas":

    console.log("El kilogramo de naranjas cuesta $0.59.");

    break;

    case "Manzanas":

    console.log("El kilogramo de manzanas cuesta $0.32.");

    break;

    case "Platanos":

    console.log("El kilogramo de platanos cuesta $0.48.");

    break;

    case "Cerezas":

    console.log("El kilogramo de cerezas cuesta $3.00.");

    break;

    case "Mangos":

    case "Papayas":

    console.log("El kilogramo de mangos y papayas cuesta $2.79.");

    break;

    default:

    console.log("Lo lamentamos, por el momento no disponemos de " + expr + ".");

}

console.log("¿Hay algo más que quisiera consultar?");

Aquí el resultado dependerá de la **expr** escrita, tal como pueden ser todos los **strings** desarrollados en cada case.  
Ahora, al **no haber un break**, se ejecutarán de todas formas los **case** siguientes, hasta que el programa **encuentre un break o default;** sin olvidar que cumpla el valor de la **var**.

Hay muchas formas de trabajar con **switch** tales como la siguiente:

var Animal = "Jirafa";

switch (Animal) {

case "Vaca":

case "Jirafa":

case "Perro":

case "Cerdo":

    console.log("Este animal subirá al Arca de Noé.");

    break;

case "Dinosaurio":

default:

    console.log("Este animal no lo hará.");

}

El resultado aquí saldrá sólo una vez gracias a que el **valor** de la **var** se produce **una sola vez** y luego es encontrado un **break**. Si eliminamos ese **break**, saldría la impresión del **case** con el **default**.

**Operaciones encadenadas con múltiples casos**

Este es un ejemplo de una sentencia **switch** secuencial con múltiples operaciones, donde, dependiendo del valor entero dado, se pueden recibir diferentes resultados. Esto demuestra que el algoritmo correrá en el orden en que se coloquen las declaraciones case, y que no tiene que ser numéricamente secuencial. En JavaScript, también es posible combinar definiciones con valores **string** dentro de estas declaraciones **case**.

var foo = 1;

var output = "Salida: ";

switch (foo) {

case 1:

    output += "¿Y ";

case 2:

    output += "Cuál ";

    output += "Es ";

case 3:

    output += "Tu ";

case 4:

    output += "Nombre";

case 5:

    output += "?";

    console.log(output);

    break;

case 6:

    output += "!";

    console.log(output);

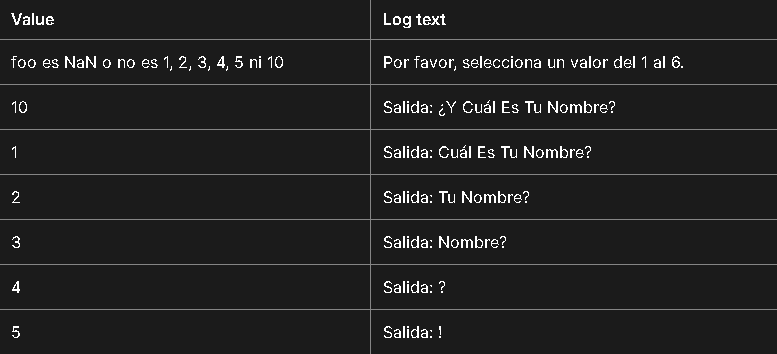
    break;

default:

    console.log("Por favor, selecciona un valor del 1 al 6.");

}

Y según lo expresado los resultados podrían ser los siguientes:



**do...while**

La sentencia (hacer mientras) crea un bucle que ejecuta una sentencia especificada, hasta que la condición de comprobación se evalúa como falsa. La condición se evalúa después de ejecutar la sentencia, dando como resultado que la sentencia especificada se ejecute al menos una vez.

let result = '';

let i = 0;

do {

i = i + 1;

result = result + i;

} while (i < 5);

console.log(result);

// Expected output: "12345"

En sintaxis, sería:   
**let**

**Do {**

**Sentencia**

**} While (condición) .**

Otro ejemplo sería el siguiente:

function doWhile(num) {

  // Aumenta el valor de "num" recibido en 5 hasta un límite de 8 veces.

  // Retorna el valor final.

  // PISTA: Utiliza el bucle do-while.

  // Tu código:

  var contador = 0; // Variable para llevar la cuenta de las veces que se incrementa num

  do {

    num += 5; // Suma 5 al valor de num

    contador++; // Incrementa el contador en 1

  } while (contador < 8); // Continúa el bucle mientras el contador sea menor que 8

  return num; // Retorna el valor final de num

}

module.exports = doWhile;

**Continue.**

**Continue** termina la ejecución de las sentencias de la iteración actual del bucle actual o la etiqueta y continua la ejecución del bucle con la próxima iteración.

**Descripción**

En contraste con la sentencia break, continue no termina la ejecución del bucle por completo; en cambio,

* En un bucle while, salta de regreso a la condición.
* En un bucle for, salta a la expresión actualizada.

La sentencia **continue** puede incluir una etiqueta opcional que permite al programa saltar a la siguiente iteración del bucle etiquetado en vez del bucle actual. En este caso, la sentencia continue necesita estar anidada dentro de esta sentecia etiquetada.

**Ejemplos**

let i = 0;

let n = 0;

while (i < 5) {

i++;

if (i == 3) {

    continue;

}

n += i;

}

console.log("El valor final de n es:", n);

En este ejemplo muestra un bucle **while** que tiene una sentencia **continue** que se ejecuta cuando el valor de **i** es **3**. Así, **n** toma los valores 1, 3, 7 y 12.

**Usando continue con una etiqueta**

En el siguiente ejemplo, una sentencia etiquetada **checkiandj** contiene una sentencia etiquetada **checkj**. Si se encuentra **continue**, el programa continua hasta encima de la sentencia **checkj**. Cada vez que se encuentra **continue**, **checkj** se reitera hasta que su condición devuelve **false**. Cuando se devuelve **false**, el recordatorio de la sentencia **checkiandj** se completa.

Si **continue** tuviese una etiqueta **checkiandj**, el programa continuaría hasta encima de la sentencia **checkiandj**.

let i = 0;

let j = 10;

checkiandj: while (i < 4) {

console.log("Valor de i dentro de checkiandj:", i);

console.log(i);

i += 1;

checkj: while (j > 4) {

    console.log("Valor de j dentro de checkj:", j);

    console.log(j);

    j -= 1;

    if (j % 2 == 0) continue checkj;

    console.log(j + " is odd.");

}

console.log("i = " + i);

console.log("j = " + j);

}  
  
Resultados =

Algo a tener en cuenta es que el objeto **document** es parte del entorno del navegador y se utiliza para manipular el DOM (Modelo de Objetos del Documento). Si estás ejecutando este código en Node.js, no podrás usar **document.write().**

**Break.**

La sentencia **break** incluye una **etiqueta** opcional que permite al programa salir de una sentencia etiquetada. La sentencia **break** necesita estar anidada dentro de la sentencia etiquetada. La sentencia etiquetada puede ser cualquier tipo de sentencia; no tiene que ser una sentencia de bucle.

function comprobarBreak(x) {

    var i = 0;

    while (i < 6) {

    if (i == 3) break;

    i++;

    }

    return i \* x ;

}

console.log(comprobarBreak(5));

Para dar un resumen concreto de lo que sucedería:

1. **function comprobarBreak(x**) **{** : Aquí se define la función **comprobarBreak** que toma un parámetro x.
2. **var i = 0;** : Se inicializa una variable i con el valor 0. Esta variable se utilizará como contador en el bucle while.
3. **while (i < 6) {** : Se inicia un bucle while que se ejecutará siempre que el valor de i sea menor que 6.
4. **if (i == 3) break;** : Dentro del bucle while, hay una declaración if que comprueba si el valor de i es igual a 3. Si es así, la instrucción break detiene la ejecución del bucle, lo que significa que saldrá del bucle inmediatamente.
5. **i++;** : Después de cada iteración del bucle while, el valor de i se incrementa en uno.
6. **return i \* x;** : Finalmente, fuera del bucle while, se devuelve el producto de i y x. Dado que el bucle se detiene cuando i alcanza el valor de 3, esto significa que el resultado devuelto será 3 veces x.

Entonces, en resumen, esta función devuelve el valor de i (que es el último valor antes de que el bucle se detenga debido al break) multiplicado por el valor de x

**toString().**

La función **toString()** en JavaScript se utiliza para convertir un objeto en una cadena de caracteres. Esta función es heredada por todos los objetos en JavaScript a través del prototipo **Object**.

Cuando se llama a **toString()** en un objeto, JavaScript busca una implementación específica de la función **toString()** para ese objeto en particular. Si no se ha definido una implementación específica para el objeto, entonces se utiliza la implementación predeterminada heredada de **Object.prototype**, que normalmente devuelve una cadena que representa el objeto en forma de **"[object tipo]".**

var array = [1, 2, 3];

console.log(array.toString()); // Devuelve "1,2,3"

var fecha = new Date();

console.log(fecha.toString()); // Devuelve una representación de cadena de la fecha y hora

var objeto = {

    nombre: "Juan",

    edad: 30

};

console.log(objeto.toString()); // Devuelve "[object Object]"

Si quisiéramos que la **var objeto** nos diese resultado, debería **sobreescribirse**:

var objeto = {

    nombre: "Juan",

    edad: 30

};

objeto.toString = function() {

    return `Nombre: ${this.nombre}, Edad: ${this.edad}`;

};

  console.log(objeto.toString()); // Devuelve "Nombre: Juan, Edad: 30"

O personalizando una función:

var objeto = {

    nombre: "Juan",

    edad: 30,

    toString: function() {

    return `Nombre: ${this.nombre}, Edad: ${this.edad}`;

    }

};

  console.log(objeto.toString()); // Devuelve "Nombre: Juan, Edad: 30"

Una forma de expresar si un **número es positivo es >=0.**

Una **función** a tener en cuenta es la siguiente para los **enteros**:

function esEntero(num) {

  // Retorna true si "num" es un entero, ya sea positivo, negativo o cero.

  // Ejemplo: 0.8   ---> false

  // Ejemplo: 1     ---> true

  // Ejemplo: (-10) ---> true

  // De lo contrario, retorna false.

  // Tu código:

  return num === Math.floor(num) || num === Math.ceil(num);

}

Un dato importante a tener en cuenta en JS es lo siguiente:  
  
El símbolo **"?"** que se encuentra entre dos valores es parte de una **expresión condicional ternaria** en JavaScript. La expresión condicional ternaria es una forma concisa de escribir una declaración **if...else** en una sola línea.

La estructura general de una expresión condicional ternaria es la siguiente:

**condición ? expresiónSiTrue : expresiónSiFalse**

Aquí, condición es una expresión que se evalúa como verdadera o falsa. Si la condición es **verdadera**, se devuelve **expresiónSiTrue;** de lo **contrario**, se devuelve **expresiónSiFalse**.

**producto === -0 ? 0 : producto**

La expresión **producto === 0 es la condición**. Si **producto** es **igual** a **0**, se devuelve 0; de lo **contrario**, se devuelve **producto**. Es una forma de asegurarse de que si el producto es **-0**, se convierta a **0** antes de devolverlo. Esto garantiza que el valor devuelto sea 0 en lugar de -0.

**SUMAR HASTA.**

function sumarHastaN(n) {

  // La función recibe un número "n" por argumento.

  // Devuelve la suma de todos los números desde 1 hasta n.

  // Tu código:

  var suma = 0; //1ro

  for (var i = 1; i <= n; i++) { //2do

    suma += i; //2do

  }

  return suma;

}

module.exports = sumarHastaN;

Esta función primero inicializa una variable suma en 0. Luego, utiliza un **bucle** **for** para iterar desde 1 hasta n, y en cada iteración, agrega el valor actual de i a la variable suma. Finalmente, devuelve el valor total de la suma.

**INCLUDES.**

La función **includes** en JavaScript se utiliza para determinar si un **array** incluye un determinado elemento, devolviendo **true** o **false** según sea el caso. Su sintaxis es la siguiente:

**array.includes(elementoBuscado)**

**Dónde:**

**array** es el array en el que deseas buscar el elemento.

**elementoBuscado** es el elemento que estás buscando en el array.

Si el **elementoBuscado** está presente en el **array**, **includes** devuelve true; de lo contrario, devuelve false.

Por ejemplo:

**var array = [1, 2, 3, 4, 5];**

**console.log(array.includes(3));** // Devuelve true, ya que 3 está presente en el array

**console.log(array.includes(6));** // Devuelve false, ya que 6 no está presente en el array

**Invertir texto:**

function invertirTexto(texto) {

  // La función recibe un argumento "texto" que es un string.

  // Tu objetivo es invertir el string y devolver el string invertido.

  // Tu código:

  // Manejo de casos especiales

  if (texto === '') {

    return texto; // Si el texto es vacío, retorna el mismo texto

  }

  // Convierte el texto en un array de caracteres, invierte el orden y luego lo une nuevamente en un string

  return texto.split('').reverse().join('');

}

module.exports = invertirTexto;

En este código, **split('')** convierte el texto en un array de caracteres, **reverse()** invierte el orden de los elementos en el **array**, y **join('')** los une nuevamente en un string.

**Palíndromo**

function esPalindromo(string) {

  // La función recibe un argumento "string".

  // Verifica si este string es palíndromo o no.

  // Retorna true si lo es, caso contrario, retorna false.

  // IMPORTANTE: Un palíndromo es una palabra o frase

  // que se lee igual hacia adelante que hacia atrás.

  // Tu código:

// Convierte el string a minúsculas y elimina los caracteres no alfabéticos

  string = string.toLowerCase().replace(/[^a-z]/g, '');

  // Compara el string original con su reverso

  return string === string.split('').reverse().join('');

}

Siendo más explicitos, primero convertimos el **string** a **minúsculas** usando **toLowerCase()** para que la comparación no distinga entre mayúsculas y minúsculas. Luego, usamos una expresión regular **/[^a-z]/g** junto con **replace()** para eliminar todos los caracteres que no son letras del alfabeto. Después, comparamos el **string** **original** con su **reverso**. Si son iguales, la función retorna **true**, lo que indica que el string es un **palíndromo**; de lo contrario, retorna **false**.

**Anagrama**

function esAnagrama(str1, str2) {

  // La función recibe dos argumentos "str1" y "str2" que son strings.

  // Determina si los dos strings son anagramas.

  // Devuelve true de ser así, sino retorna false.

  // IMPORTANTE: Un anagrama es una palabra que se forma

  // con las mismas letras que otra, pero en orden diferente.

  // Tu código:

    // Elimina los espacios y convierte a minúsculas

    str1 = str1.replace(/\s/g, '').toLowerCase();

    str2 = str2.replace(/\s/g, '').toLowerCase();

    // Si las longitudes de los strings son diferentes, no pueden ser anagramas

    if (str1.length !== str2.length) {

      return false;

    }

    // Convierte los strings en arrays, los ordena y los convierte nuevamente en strings

    const sortedStr1 = str1.split('').sort().join('');

    const sortedStr2 = str2.split('').sort().join('');

    // Compara los strings ordenados

    return sortedStr1 === sortedStr2;

  }

module.exports = esAnagrama;

**Array, Arreglos o Matriz.**

Es uno de los elementos más utilizados y valorados en JavaScript, siendo un **objeto global** con el fin de gestionar y almacenar información; tales como **username**, **password**, **receipt**, etc. Existen 2 conceptos dentro de un **array** que se parecen pero no son iguales:

1. Uno de ellos es el **elemento**, el dato que hayamos utilizado y guardado.
2. El otro es el concepto **índice**. No representan el dato en sí, sino la posición o ubicación donde se encuentra el elemento. Al igual que la mayoría de las cuentas de **JS**, su número inicial es el **0**, no el **1**.

Esto nos lleva a que la cantidad de **elementos** que tendrá un **array** siempre será un número mayor que la **cantidad de índices**. Y a tener en cuenta: **todos los elementos de un arreglo llevan una condición**.

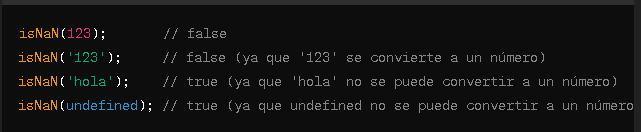
Uno de los códigos del **índice** es **indexOf()**, la cual busca el elemento especificado dentro del **array** y devuelve el índice de su primera ocurrencia. Si el elemento no se encuentra, devuelve -1.

El método más utilizado en los **array** es el **Length**, el cual nos permite saber cuántos elementos tiene nuestro arreglo, o sea, cuál es su **extensión**.

El **array** al ser un **objeto global** está asociado a muchos **métodos (**funciones **nativas** o preestablecidas en los lenguajes de programación), tales como:

**Sort()** lo que hace en este código es ordenarlos alfabéticamente o de menor a mayor.

**isNaN()** se utiliza para determinar si un valor pasado como argumento es de tipo NaN (Not a Number) o no. Retorna true si el valor es NaN, de lo contrario retorna false.

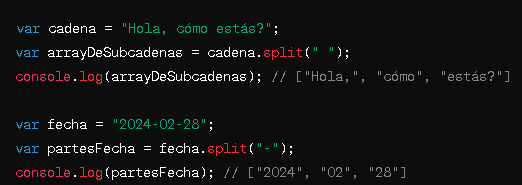


**Métodos de Inserción.**

1. **Push():** añade uno o más elementos al final del arreglo, y devuelve la nueva longitud del arreglo.
2. **Unshift():** añade uno o más elementos al inicio del arreglo, devolviendo su nueva longitud.
3. **Pop():** elimina el último elemento de un array.
4. **Shift() :** elimina el primer elemento de un array.

**Métodos varios.**

1. **Includes() :** incluye un elemento dentro de un array, devolviendo un valor **booleano** según el elemento encontrado.
2. **Every() :** determina si **todos los elementos** de un array **cumplen o no** una misma condición.
3. **Split() :** se utiliza para dividir una cadena en un **array** en **subcadenas** basándose en un **separador especificado** y luego **devuelve** ese array.

****

1. **Join () :** convierte un arreglo en un string, uniendo todos los elementos de este en una misma cadena; obviamente con los **(‘ ‘)**.

**Métodos de Recorrido.**

* **forEach() :** nos permite recorrer elemento por elemento dentro de un array.

array.forEach(**function**(**elemento**, **índice**, **arreglo**) {

// bloque de código a ejecutar

});

**let** numeros = [1, 2, 3, 4, 5];

numeros.**forEach**(function(numero) {

console.log(numero);

});

**- Elemento:** El valor del elemento actual del array.

**- Índice:** El índice del elemento actual en el array.

**- Arreglo:** El array sobre el que se está iterando.

Siendo más claro, **for** es más genérico y puede utilizarse para iterar sobre cualquier tipo de iterable, mientras que **forEach** es específico para **arrays** y proporciona una forma más **clara** y **concisa** de iterar sobre ellos. La elección **dependerá** de los **requisitos específicos del código** y de la **preferencia** **del** **desarrollador** en cuanto a claridad y simplicidad.

A diferencia del bucle **for**, **forEach()** es más conciso y claro para iterar sobre arrays, pero no se puede usar para iterar sobre otros tipos de iterables ni se puede **interrumpir** o **detener** su ejecución como en el caso de **for** con **break** o **return**.

* **map() :** a diferencia de **forEach**, crea una copia y pueden realizarse cambios.

**Bucles de los Array.**

En **JS** dos bucles nativos y usados en **array** son **for** y **while**.

En el bucle **for** como explicamos antes, recibiremos entre sus **paréntesis 3 parámetros**. Y siempre declararemos una **variante de iteración**.

**ToLowerCase:** convierte cada elemento en una minúscula.

**ToUpperCase:** convierte cada elemento en una mayúscula.

**Bucles FOR y WHILE.**

Bucles o ciclos de iteración nos permiten realizar acciones repetitivas de manera más simple. En sí son una secuencia de instrucciones a las que se le asigna una condición, y se ejecutará hasta que tal condición deje de cumplirse.

**Bucle Infinito.**

Un bucle infinito se produce cuando un bucle comienza a ejecutarse, pero jamás termina. Esto se debe a que nunca se llega a la condición de quiebre.

**Esto consume muchos recursos de la computadora, por lo que hay que intentar evitarlos.**

**Bucle For**

Por un lado, el Bucle **For** crea un bucle que consiste en tres expresiones opcionales, encerradas en paréntesis y separadas por puntos y comas, seguidas de una sentencia ejecutada en un bucle. Trabaja con un número determinado de repeticiones, repitiéndose una vez por cada elemento que el arreglo tenga.

**var suma = 0;**

**for (inicio; condición; actualización){sentencia}**

**for (let i = 0; i < 5; i++) {**

**suma = suma + i;**

**console.log('Variable de iteración: ', i);**

**}**

**console.log('Variable suma: ', suma);**

Para explicar el código:

* **Inicio:** Se ejecuta una vez al principio, típicamente se utiliza para inicializar variables.
* **Condición:** Se evalúa en cada iteración y si es verdadera, el bloque de código se ejecuta. Si es falsa, la iteración termina.
* **Actualización:** Se ejecuta al final de cada iteración, típicamente se utiliza para actualizar variables de control.

**Bucle while**

El **Bucle While**es usado cuando no tenemos la certeza de cuántos pasos vamos a necesitar hasta finalizar la ejecución, trabajando con condiciones, y mientras tales se cumplan, el bucle continuará. Por ejemplo, podríamos pensar en una función que siga agregando +1 hasta llegar a un número determinado.

**Let suma = 0;**

**while(suma < 3) {**

**suma = suma + 1;**

**console.log(suma);**

**};**

**Aquí tenemos uno de los ejercicios de JS con la potenciación y WHILE:**

function esPotenciaDeDos(numero) {

  // La función recibe un "numero" por argumento.

  // Determina si es una potencia de 2.

  // Devuelve true si lo es, sino devuelve false.

  // PISTA: Utiliza un bucle while.

  // Tu código:

  if (numero <= 0) {

    return false; // Los números negativos o cero no son potencias de 2

  }

  while (numero > 1) {

    if (numero % 2 !== 0) {

      return false; // Si en algún momento el número no es divisible entre 2, no es potencia de 2

    }

    numero = numero / 2; // Dividimos el número por 2

  }

  return true; // Si llegamos a 1, entonces es una potencia de 2

}

module.exports = esPotenciaDeDos;

**EJERCICIOS:**

function encontrarElemento(elemento, array) {

  // Busca el elemento pasado por argumento dentro del array.

  // Si no se encuentra, retorna -1.

  // Tu código:

  return array.indexOf(elemento);

}

El método **indexOf()** devuelve el índice del primer elemento coincidente dentro del array, o -1 si el elemento no se encuentra.

function obtenerElementoAleatorio(array) {

  // Devuelve un elemento aleatorio del arreglo array.

  // PISTA: Usa el método Math.random().

  // Tu código:

// Genera un índice aleatorio usando Math.random y la longitud del array.

  var indiceAleatorio = Math.floor(Math.random() \* array.length);

//Devuelve el elemento obtenido al índice.

  return array[indiceAleatorio];

}

module.exports = obtenerElementoAleatorio;

function obtenerPrimerStringLargo(array) {

  // Devuelve el primer string con más de 5 caracteres en el array.

  // Tu código:

  // Itera sobre cada elemento del array utilizando un bucle for.

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

    // Verifica si el elemento en la posición i es un string y tiene más de 5 caracteres.

    if (typeof array[i] === 'string' && array[i].length > 5) {

      // Devuelve el primer string largo encontrado.

      return array[i];

    }

  }

  // Si ningún string largo es encontrado, devuelve null o undefined.

  return null; // o return undefined;

}

function duplicarElementos(array) {

  // Duplica (multiplica x2) cada elemento del array de números.

  // Devuelve un array con los duplicados.

  // Tu código:

  // Crea un nuevo array para almacenar los elementos duplicados.

  var arrayDuplicado = [];

  // Utiliza forEach para iterar sobre cada elemento del array.

  array.forEach(num => {

    // Multiplica cada elemento por 2 y añádelo al nuevo array.

    arrayDuplicado.push(num \* 2);

  });

  // Devuelve el nuevo array con los elementos duplicados.

  return arrayDuplicado;

}

module.exports = duplicarElementos;

function convertirStringAMayusculas(array) {

  // Convierte a mayúsculas todos los strings del array.

  // Retorna el arreglo resultante.

  // Tu código:

  // Utiliza el método map() para iterar sobre cada elemento del array.

  // Aplica toUpperCase() a cada string para convertirlo a mayúsculas.

  // Retorna el arreglo resultante.

  return array.map(function(elemento) {

    // Verifica si el elemento es un string antes de convertirlo a mayúsculas.

    if (typeof elemento === 'string') {

      return elemento.toUpperCase();

    } else {

      return elemento; // Devuelve el elemento sin modificar si no es un string.

function filtrarNumerosPares(array) {

  // Devuelve un arreglo solo con los números pares presentes en el array.

  // Tu código:

  // Utiliza el método filter() para crear un nuevo array con los números pares.

  return array.filter(function(elemento) {

    // Filtra solo los números pares.

    return elemento % 2 === 0;

  });

}

module.exports = filtrarNumerosPares;

**Filter()** es un método que genera un nuevo array imprimiendo **sólo los elementos que cumplen la condición**.

function promedioResultadosTest(resultadosTest) {

  // Itera sobre los elementos del arreglo resultadosTest y devuelve el promedio de las notas.

  // Tu código:

  // Inicializa la variable suma para almacenar la suma de las notas.

  let suma = 0;

  // Itera sobre cada nota en el array y suma sus valores.

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

    suma += resultadosTest[i];

  }

  // Calcula el promedio dividiendo la suma total por la cantidad de notas.

  let promedio = suma / resultadosTest.length;

  // Devuelve el promedio calculado.

  return promedio;

}

module.exports = promedioResultadosTest;

function contarElementosMayoresA10(array) {

  // Cuenta y devuelve la cantidad de elementos mayores a 10 en el array de números.

  // Tu código:

  return array.filter(function(elemento) {

    return elemento > 10;

  }).length;

}

module.exports = contarElementosMayoresA10;

El añadir el método **.length** es lo que dará la **cantidad** de elementos en la impresión.

function encontrarIndiceMayor(array) {

  // Encuentra el índice del número más grande en el array de números.

  // Devuelve el valor de este índice.

  // Tu código:

// Si el array está vacío, devuelve 0.

if (array.length === 0) {

  return 0;

}

// Inicializa el índice del número mayor como 0 y el número mayor como el primer elemento del array.

let indiceMayor = 0;

let numeroMayor = array[0];

// Itera sobre cada elemento del array empezando desde el segundo elemento.

for (let i = 1; i < array.length; i++) {

  // Si el elemento actual es mayor que el número mayor registrado,

  // actualiza el índice del número mayor y el número mayor.

  if (array[i] > numeroMayor) {

    indiceMayor = i;

    numeroMayor = array[i];

  }

}

// Devuelve el índice del número mayor encontrado.

return indiceMayor;

}

module.exports = encontrarIndiceMayor;

function multiplicarElementosPorIndice(array) {

  // Multiplica cada elemento del array por su índice.

  // Devuelve el nuevo arreglo con los resultados.

  // Tu código:.

  if (array.length === 0) {

    return [];

  }

  let resultados = [];

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

    resultados.push(array[i] \* i);

  }

  return resultados;

}

module.exports = multiplicarElementosPorIndice;

function multiplicarArgumentos() {

  // Usa la palabra clave `arguments` para multiplicar todos los argumentos y devolver el producto.

  // Si no se pasan argumentos retorna 0. Si se pasa un argumento, simplemente retórnalo.

  // [PISTA]: "arguments" es un arreglo.

  // Tu código:

  if (arguments.length === 0) {

    return 0;

  }

  if (arguments.length === 1) {

    return arguments[0];

  }

    // Si se pasan múltiples argumentos, inicializa el producto como 1

    let producto = 1;

    // Itera sobre cada argumento y multiplica su valor al producto

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

      producto \*= arguments[i];

    }

    // Devuelve el producto total

    return producto;

}

module.exports = multiplicarArgumentos;

function todosIguales(array) {

  // Si todos los elementos del arreglo son iguales, retornar true.

  // Caso contrario, retornar false.

  // Tu código:

  // Almacena el primer elemento del array para compararlo con el resto

  let primerElemento = array[0];

  // Itera sobre cada elemento del array a partir del segundo elemento

  for (let i = 1; i < array.length; i++) {

    // Si algún elemento es diferente al primer elemento, devuelve false

    if (array[i] !== primerElemento) {

      return false;

    }

  }

  // Si todos los elementos son iguales, devuelve true

  return true;

}

module.exports = todosIguales;

Ante **una sola línea de código** dentro de un **“if”** no es necesario aplicar las llaves.

**Control de flujo**: