Taller de Desarrollo en la Web con JavaScript

Compañía: *Multicomputos.*

Departamento: *Aplicaciones.*

Instructor: *Ing. Heri Espinosa.*

Fecha de inicio: *07/05/2024.*

INDICE

2.1 Hacer una página de selección que muestre 2 imágenes con título, descripción y un check para seleccionar.

2.2 Hacer una página que muestre un formulario de registro.

2.3 Hacer los siguientes ejercicios de JavaScript:

a) Mostrar por consola las variables: nombre, apellido, edad, sexo, rd, estatura.
b) Mostrar por consola el resultado de sumar, restar, multiplicar, dividir y resultado del residuo de una división.
c) Manejo y gestión de los arrays.
d) Manejo y gestión de los objetos.
e) Practica con funciones.
f) Practica con clases.
g) Practica con condicionantes.
h) Practica con bucles.

3. TRABAJANDO CON EVENTOS Y EL INTERNET:

3.1 Que es el DOM?

3.2 ¿Qué es BOM?

3.3 ¿Qué son los Eventos?

3.5 ¿Qué es la Propagación?

3.6 ¿Qué es el internet?

3.7 ¿Qué es la Web?

3.8 ¿Qué es HTTP / HTTPS?

3.9 ¿Qué es la Asincronía?

3.0.1 ¿Peticiones AJAX?

3.0.2 ¿Qué son las APIs?

4. CONTINUACION DE PRACTICAS CON HTML, CSS Y JAVASSCRIPT:

4.1 Hacer una página de selección que muestre 2 imágenes con título, descripción y un check para seleccionar.

4.2 Hacer una página que muestre un formulario de registro.

5. CONOCIENDO EL LADO DEL SERVIDOR:

5.1 ¿Qué es una computadora?

5.2 ¿Qué es un Servidor?

5.3 Ecosistema de un servidor.

5.4 ¿Diferencia entre una computadora cliente y una computadora servidor?

5.5 Node.js
   a) ¿Qué es Node.js?
   b) Características de Node.js
   c) Como funciona Node.js
   d) ¿Cuáles son las limitaciones de Node.js?
   e) Ventajas y desventajas de Node.js

6. CREANDO NUESTRO PRIMER SERVIDOR:

   6.1 Crear un servidor que envié un texto (“Hola Mundo”) en una petición Get.

   6.2 Crear un servidor con las rutas básicas (Get, Post, Put y Delete).

7. PROYECTO FINAL: CREAR UNA APLICACIÓN FULL STACK CON JAVASCRIPT.

   7.1 Aplicación de registro de usuarios.

1. Primeros pasos:

1.1 ¿Qué es JavaScript?

JavaScript es un lenguaje de programación de alto nivel que se utiliza principalmente para crear interactividad en páginas web. Es un componente fundamental de la web moderna, ya que se utiliza para agregar funcionalidades dinámicas a los sitios web, como animaciones, validación de formularios, manipulación del DOM (Document Object Model) y comunicación con servidores para cargar o enviar datos sin necesidad de recargar la página completa. Además de su uso en el desarrollo web, JavaScript también se puede utilizar en otros entornos, como aplicaciones móviles e incluso desarrollo de servidores a través de plataformas como Node.js.

JavaScript es un lenguaje funcional y desde la norma de ECMAScript5 (2015) puede considerarse un lenguaje de programación orientado a objetos basado en prototipos, con algunas peculiaridades. JavaScript trabaja con un objeto global que gobierna a todos los demás objetos (el señor de los objetos).

En resumen, JavaScript funciona como un lenguaje de programación que permite agregar interactividad y dinamismo a las páginas web, manipular el contenido y la estructura de la página, manejar eventos del usuario y comunicarse con el servidor. Su versatilidad y ubicuidad lo convierten en una herramienta fundamental en el desarrollo web moderno.

1.2 Características principales:

- Interpretado: JavaScript es un lenguaje interpretado, lo que significa que el código se ejecuta directamente por el navegador web sin necesidad de compilación.
- Basado en prototipos: En lugar de usar clases para la herencia, JavaScript utiliza prototipos para la herencia de objetos.
- Tipado dinámico: Las variables en JavaScript no tienen tipos de datos fijos, lo que permite una flexibilidad considerable.
- Funciones de primera clase: Las funciones en JavaScript son objetos de primera clase, lo que significa que se pueden asignar a variables, pasar como argumentos y devolver como valores de otras funciones.

1.3 ¿Para qué se utiliza JavaScript?

Anteriormente, las páginas web eran estáticas, similares a las páginas de un libro. Una página estática mostraba principalmente información en un diseño fijo y no todo aquello que esperamos de un sitio web moderno. JavaScript surgió como una tecnología del lado del navegador para hacer que las aplicaciones web fueran más dinámicas. Por medio de JavaScript, los navegadores eran capaces de responder a la interacción de los usuarios y cambiar la distribución del contenido en la página web.

A medida que el lenguaje evolucionó, los desarrolladores de JavaScript establecieron bibliotecas, marcos y prácticas de programación y comenzaron a utilizarlo fuera de los navegadores web. En la actualidad, puede utilizar JavaScript para el desarrollo tanto del lado del cliente como del lado del servidor.

<u>Uso en desarrollo web</u>:

- Manipulación del DOM: JavaScript se utiliza para interactuar con el DOM de una página web, lo que permite modificar el contenido, la estructura y el estilo de la página.
- Eventos: Permite la detección y manejo de eventos del usuario, como hacer clic en un botón o mover el mouse sobre un elemento.
- AJAX: JavaScript se utiliza para realizar solicitudes asíncronas al servidor web, lo que permite actualizar partes de una página sin necesidad de recargarla por completo.
- Frameworks y bibliotecas: Existen numerosos frameworks y bibliotecas de JavaScript, como React.js, AngularJS, Vue.js, y jQuery, que simplifican el desarrollo web y ofrecen funcionalidades adicionales.

Uso fuera del navegador: Aunque JavaScript es más conocido por su uso en el desarrollo web, también se puede utilizar en otros entornos, como:

- Node.js: Permite ejecutar JavaScript en el lado del servidor, lo que facilita la creación de aplicaciones web escalables y de alto rendimiento.
- Desarrollo de aplicaciones móviles: Frameworks como React Native y Ionic permiten el desarrollo de aplicaciones móviles utilizando JavaScript.
- Desarrollo de aplicaciones de escritorio: Con herramientas como Electron, es posible desarrollar aplicaciones de escritorio utilizando tecnologías web, incluido JavaScript.

1.4 ¿Cómo funciona JavaScript?

Todos los lenguajes de programación funcionan mediante la traducción de sintaxis similar a la del inglés a código de máquina, que posteriormente el sistema operativo se encarga de ejecutar. JavaScript se clasifica principalmente como un lenguaje de scripting o interpretado. El código JavaScript es interpretado, es decir, directamente traducido a código de lenguaje de máquina subyacente mediante un motor de JavaScript. En el caso de otros lenguajes de programación, un compilador se encarga de compilar todo el código en código de máquina en un paso diferente. En consecuencia, todos los lenguajes de scripts son lenguajes de programación, pero no todos los lenguajes de programación son lenguajes de scripts.

Interpretación en el navegador: Cuando un navegador carga una página web que contiene código JavaScript, el navegador interpreta el código JavaScript y lo ejecuta directamente en el entorno del navegador. Esto significa que el navegador lee línea por línea el código JavaScript y realiza las acciones especificadas, como modificar el contenido de la página, manejar eventos del usuario o realizar solicitudes al servidor.

Manipulación del DOM: Una de las funciones principales de JavaScript en el navegador es interactuar con el Document Object Model (DOM) de la página web. El DOM representa la estructura de la página como un árbol de nodos, donde cada elemento HTML es un nodo en el árbol. JavaScript puede acceder y manipular estos nodos para cambiar el contenido, el estilo y la estructura de la página dinámicamente.

Comunicación con el servidor: JavaScript también se utiliza para realizar solicitudes asíncronas al servidor web utilizando la técnica conocida como AJAX (Asynchronous JavaScript and XML). Esto permite actualizar partes de una página sin necesidad de recargarla por completo, lo que proporciona una experiencia de usuario más fluida.

Manejo de eventos: JavaScript permite manejar eventos del usuario, como hacer clic en un botón, mover el mouse sobre un elemento o enviar un formulario. Los eventos son desencadenados por acciones del usuario y JavaScript puede estar atento a estos eventos y ejecutar código en respuesta a ellos.

1.5 Motor de JavaScript:

Un motor JavaScript es un programa de computación que ejecuta código JavaScript. Los primeros motores de JavaScript eran verdaderos intérpretes, pero todos los motores modernos utilizan el método justo a tiempo o la compilación en tiempo de ejecución para mejorar el rendimiento.

1.6 JavaScript del lado del cliente:

El código que se ejecuta en el explorador se conoce como código de lado-cliente, y su principal preocupación es la mejora de la apariencia y el comportamiento de una página web entregada. Esto incluye la selección y estilo de los componentes UI, la creación de layouts, navegación, validación de formularios, etc.

JavaScript del cliente se refiere a la forma en que JavaScript funciona en el navegador. En este caso, el motor de JavaScript está dentro del código del navegador. Todos los principales navegadores web incluyen sus propios motores de JavaScript incorporados.
Los desarrolladores de aplicaciones web escriben código JavaScript con diferentes funciones asociadas a varios eventos, como hacer clic con el ratón o situar el ratón sobre un elemento. Estas funciones realizan cambios en HTML y CSS.

A continuación, se muestra una perspectiva general del funcionamiento de JavaScript del lado del cliente:

1. El navegador carga una página web cuando recibe una visita.
2. Durante la carga, el navegador convierte la página y todos sus elementos, como los botones, las etiquetas y los cuadros desplegables, en una estructura de datos denominada modelo de objetos del documento (DOM).
3. El motor JavaScript del navegador convierte el código JavaScript en código intermedio. Se trata de un código intermediario entre la sintaxis de JavaScript y la máquina.
4. Diferentes eventos, como hacer clic con el ratón en un botón, desencadenan la ejecución del bloque de código JavaScript asociado. Posteriormente, el motor interpreta el código intermedio y realiza cambios en el DOM.
5. El navegador muestra el nuevo DOM.

1.7 JavaScript del lado del servidor:

El código de lado-servidor gestiona tareas como la validación de los datos enviados y las peticiones, usando bases de datos para almacenar y recuperar datos, y enviando los datos correctos al cliente según se requiera.

JavaScript del lado del servidor se refiere al uso de JavaScript para desarrollar aplicaciones y servicios en el lado del servidor. Esto es posible gracias a plataformas como Node.js, que ejecutan JavaScript fuera del navegador, permitiendo así construir aplicaciones web completas utilizando JavaScript tanto en el frontend como en el backend. En este caso, una función de JavaScript del lado del servidor puede acceder a la base de datos, realizar diferentes operaciones lógicas y responder a varios eventos desencadenados por el sistema operativo del servidor. Node.js utiliza el motor V8 de Google Chrome para ejecutar código JavaScript en el servidor de manera eficiente.

1.8 ¿Diferencia entre utilizar JS en el cliente y en el servidor?

El uso de JavaScript en el cliente y en el servidor difiere en su propósito y contexto de ejecución:

Cliente (Navegador Web):

- Interactividad del usuario: En el navegador, JavaScript se utiliza principalmente para mejorar la interactividad del usuario en las páginas web. Esto incluye manipulación del DOM, manejo de eventos del usuario (como clics de mouse o teclado), validación de formularios y animaciones.
- Funcionalidades del frontend: JavaScript es esencial para agregar dinamismo y funcionalidades atractivas a las interfaces de usuario de las aplicaciones web. Esto puede incluir efectos visuales, actualizaciones de contenido dinámico y solicitudes asíncronas al servidor (AJAX) para cargar o enviar datos sin recargar la página completa.
- Ejecución en el navegador: El código JavaScript en el cliente se ejecuta en el navegador web del usuario, lo que significa que el rendimiento y la seguridad están determinados en gran medida por las capacidades y restricciones del navegador.

Servidor (Node.js u otros entornos):

- Lógica del servidor: En el lado del servidor, JavaScript se utiliza para escribir la lógica del backend de las aplicaciones web. Esto incluye manejar solicitudes de clientes, acceder a bases de datos, procesar datos, autenticar usuarios y generar respuestas para enviar al cliente.
- Escalabilidad y rendimiento: JavaScript en el servidor, especialmente con Node.js, se valora por su capacidad para manejar grandes volúmenes de solicitudes simultáneas de manera eficiente. Esto se debe a su modelo de operaciones de entrada/salida no bloqueantes y basado en eventos.
- Ejecución fuera del navegador: El código JavaScript en el servidor se ejecuta fuera del navegador, generalmente en un entorno de servidor web como Node.js. Esto permite el acceso a funcionalidades específicas del sistema operativo y una mayor flexibilidad en la implementación de la lógica del servidor.

En resumen, aunque JavaScript se utiliza tanto en el cliente como en el servidor, sus roles y contextos de ejecución son distintos. En el cliente, se enfoca en mejorar la experiencia del usuario y la interactividad en el navegador web, mientras que, en el servidor, se encarga de la lógica del backend y la gestión de solicitudes del cliente en el servidor web.

1.9 ¿Cuáles son las limitaciones de JavaScript?

JavaScript es un lenguaje poderoso y versátil, pero también tiene algunas limitaciones:

- Seguridad: Dado que JavaScript se ejecuta en el navegador del cliente, puede ser vulnerable a ataques maliciosos como inyección de código, cross-site scripting (XSS) y otras vulnerabilidades de seguridad si no se implementan adecuadas medidas de seguridad.
- Rendimiento: Aunque los motores de JavaScript han mejorado significativamente en términos de rendimiento, el código JavaScript puede ser menos eficiente que otros lenguajes de programación, especialmente en tareas computacionalmente intensivas. Esto puede afectar el rendimiento de las aplicaciones web, especialmente en dispositivos con recursos limitados.
- Dependencia del navegador: JavaScript puede comportarse de manera diferente en diferentes navegadores debido a la implementación del motor JavaScript de cada navegador. Esto puede requerir pruebas y ajustes adicionales para garantizar la compatibilidad entre navegadores.

- Sin acceso directo al sistema operativo: Por razones de seguridad, JavaScript en el navegador no tiene acceso directo al sistema operativo subyacente del dispositivo del usuario, lo que limita su capacidad para realizar tareas como leer/escribir archivos en el sistema de archivos local.
- Limitaciones de memoria y recursos: JavaScript en el navegador está limitado por las capacidades del navegador y el dispositivo del usuario en términos de memoria y recursos disponibles. Esto puede afectar el rendimiento y la capacidad de ejecutar aplicaciones web complejas en dispositivos con recursos limitados.
- Dificultad para el desarrollo de aplicaciones de gran escala: Aunque es posible desarrollar aplicaciones web de gran escala con JavaScript, puede ser más difícil mantener la estructura y el orden del código a medida que la aplicación crece en tamaño y complejidad. Esto puede requerir el uso de patrones de diseño y prácticas de desarrollo específicas para mantener el código organizado y mantenible.

Algo a resaltar es que JavaScript es un lenguaje de tipado débil, es decir, no permite al programador definir el tipo de las variables. Una variable puede almacenar cualquier tipo de datos durante el tiempo de ejecución, y las operaciones asumen el tipo de variable. El resultado también se puede proyectar a otro tipo de datos; por ejemplo, una operación podría devolver el resultado como la cadena “5” en lugar del número 5. Esto puede ocasionar problemas de codificación accidentales y errores en el código a causa de los fallos de tipo.

A pesar de estas limitaciones, JavaScript sigue siendo uno de los lenguajes de programación más populares y ampliamente utilizados en el desarrollo web debido a su versatilidad, interoperabilidad y amplio ecosistema de herramientas y bibliotecas.

1.10 ¿Ventajas y desventajas de JavaScript?

- Ventajas:
  1. Ubicuidad: JavaScript se ejecuta en todos los navegadores web modernos, lo que lo convierte en una opción muy versátil para el desarrollo web.
  2. Facilidad de aprendizaje: Comparado con otros lenguajes de programación, JavaScript es relativamente fácil de aprender, especialmente para aquellos que ya están familiarizados con HTML y CSS.
  3. Flexibilidad: JavaScript es un lenguaje flexible que se puede utilizar tanto para el desarrollo frontend como backend (Node.js), así como para la creación de aplicaciones móviles e incluso aplicaciones de escritorio.
  4. Interactividad: Permite la creación de páginas web interactivas y dinámicas, lo que mejora la experiencia del usuario.
  5. Gran cantidad de bibliotecas y frameworks: Existen numerosas bibliotecas y frameworks como React, Angular y Vue.js que facilitan el desarrollo de aplicaciones web complejas.

- Desventajas:
  1. Seguridad: Al ejecutarse en el lado del cliente, JavaScript puede ser vulnerable a ataques de seguridad como la inyección de código malicioso (XSS) si no se implementan medidas adecuadas de seguridad.
  2. Rendimiento: Aunque ha mejorado con el tiempo, JavaScript a menudo es criticado por su rendimiento en comparación con otros lenguajes de programación, especialmente en aplicaciones que requieren un alto nivel de procesamiento del lado del cliente.

3. Dependencia del cliente: Al depender del navegador del cliente, el rendimiento y la ejecución del código JavaScript pueden variar según el dispositivo y la velocidad de la conexión a Internet del usuario.
4. SEO: Aunque ha habido avances en el SEO para aplicaciones web basadas en JavaScript, todavía puede ser más difícil de indexar para los motores de búsqueda en comparación con las páginas web estáticas o generadas en el servidor.

1.11 Conceptos básicos de JavaScript:

- Variables:
  Las variables son contenedores que se utilizan para almacenar datos. Estos datos pueden ser de diversos tipos, como números, cadenas de texto, booleanos, objetos, funciones, entre otros. Las variables permiten a los programadores referirse a estos datos mediante un nombre simbólico y manipularlos dentro de un programa.

  En JavaScript, puedes declarar una variable utilizando la palabra clave var, let o const. Por ejemplo:

  ```
  var edad;
  let nombre;
  const PI = 3.14159;
  ```

  Después de declarar una variable, puedes asignarle un valor utilizando el operador de asignación (=). Siguiendo el ejemplo anterior a las variables edad y nombre se le puede asignar un valor de la siguiente manera.

  ```
  edad = 25;
  nombre = "Juan";
  ```

  JavaScript es un lenguaje de tipado dinámico, lo que significa que no necesitas especificar explícitamente el tipo de datos de una variable al declararla. El tipo de datos de una variable se determina automáticamente en tiempo de ejecución según el valor que se le asigna. Por ejemplo:

  ```
  var edad = 25; // edad es de tipo número
  let nombre = "Juan"; // nombre es de tipo cadena de texto
  ```

  Ámbito de las variables: Las variables en JavaScript tienen ámbitos que pueden ser globales o locales. Una variable declarada fuera de una función tiene un ámbito global, lo que significa que puede ser accedida desde cualquier parte del código. Una variable declarada dentro de una función tiene un ámbito local y solo puede ser accedida dentro de esa función.

  Caracteristicas de var, let y const:
  - var: Ámbito de función: Las variables declaradas con var tienen un ámbito de función. Esto significa que son visibles dentro de la función en la que están declaradas, incluso antes de la línea de declaración.

    Hoisting: Las declaraciones de variables var son "levantadas" (hoisted) al comienzo del ámbito de la función o, si se declaran fuera de una función, al comienzo del ámbito global. Esto significa que puedes acceder a la variable antes de la declaración, aunque su valor será undefined.

Reasignación y redeclaración: Puedes reasignar y redeclarar variables var en el mismo ámbito sin lanzar un error.

- let: Ámbito de bloque: Las variables declaradas con let tienen un ámbito de bloque. Esto significa que están limitadas al bloque en el que están declaradas, como un bucle for o un bloque if.

  No hay hoisting: A diferencia de var, las variables let no son hoisted, lo que significa que no puedes acceder a ellas antes de la línea de declaración.

  No se puede redeclarar en el mismo ámbito: Intentar redeclarar una variable let en el mismo ámbito lanzará un error.

- const: Constantes inmutables: Las variables declaradas con const son constantes y no pueden ser reasignadas después de la inicialización. Sin embargo, el valor de un objeto o un array declarado con const aún puede cambiar, ya que solo la referencia a ese objeto o array es constante, no sus propiedades o elementos.

  Ámbito de bloque: Al igual que let, las variables const tienen un ámbito de bloque y no están hoisted.

  No se puede redeclarar en el mismo ámbito: Intentar redeclarar una variable const en el mismo ámbito lanzará un error.

Dato curioso: ¿Que es Hoisting?

El hoisting es un comportamiento en JavaScript donde las declaraciones de variables y funciones son "levantadas" (hoisted) al inicio de su ámbito de ejecución antes de que se ejecute el código. Esto significa que, aunque puedas utilizar una variable o función antes de su declaración en el código, en realidad la declaración se mueve al inicio del ámbito de ejecución durante la fase de compilación.

El hoisting ocurre en dos casos principales:

1. Hoisting de variables: Cuando declaras una variable utilizando var, la declaración de la variable se levanta al inicio del ámbito de la función o, si está fuera de una función, al inicio del ámbito global. Sin embargo, la asignación de valor permanece en su lugar original. Por ejemplo:

   ```
   console.log(miVariable); // undefined
   var miVariable = 5;
   ```

   En este caso, aunque miVariable se utiliza antes de su declaración, el código es interpretado como:

   ```
   var miVariable;
   console.log(miVariable); // undefined
   miVariable = 5;
   ```

2. Hoisting de funciones: Cuando declaras una función utilizando la declaración de función (function myFunction() { ... }), la declaración completa de la función se levanta al inicio del ámbito de ejecución. Por ejemplo:

   ```
   miFuncion(); // "Hola"
   function miFuncion() {
   ```

```
    console.log("Hola");
}
```

En este caso, la llamada a miFuncion() se realiza antes de la declaración de la función, pero la declaración de la función se levanta al inicio, lo que permite que la llamada se realice sin errores.

Es importante tener en cuenta que el hoisting solo mueve la declaración de variables y funciones, no sus asignaciones o definiciones. Por lo tanto, mientras que puedes acceder a una variable declarada con var antes de su declaración, su valor será undefined. Del mismo modo, puedes llamar a una función antes de su declaración, pero su cuerpo no se ejecutará hasta que se alcance esa línea de código.

- Array/s (Matrices).
  Un array en JavaScript, también conocido como matriz o arreglo, es una estructura de datos que se utiliza para almacenar una colección de elementos, donde cada elemento puede ser de cualquier tipo de datos, como números, cadenas, booleanos, objetos, u otros arrays. Los arrays en JavaScript son objetos que tienen propiedades y métodos incorporados que facilitan la manipulación de datos.

  Los arrays son una parte fundamental de JavaScript y se utilizan ampliamente en el desarrollo de aplicaciones web para almacenar y manipular conjuntos de datos de manera eficiente. Puedes recorrer todos los elementos de un array utilizando bucles como for, forEach(), for...of, map(), filter(), reduce(), etc. Estos bucles te permiten realizar operaciones en cada elemento del array. Su versatilidad y variedad de métodos incorporados hacen que sean una herramienta poderosa para trabajar con datos estructurados.

  Puedes declarar un array utilizando corchetes [] y separando los elementos por comas. Por ejemplo:

```
let miArray = [1, 2, 3, “Heri”, true, 20];
```

  Todos los arrays en JavaScript tienen una propiedad length que indica el número de elementos que contiene el array. Por ejemplo:

```
console.log(miArray.length); // 6
```

  Para poder acceder a los elementos de un arreglo (array) utilizamos su índice, que comienza desde cero (0), debemos escribir el nombre de nuestro array, que en este caso es miArray, y luego insertar entre corchetes [] la posición (índice) del elemento al que queremos acceder. Por ejemplo:

```
console.log(miArray[0]); // 1
console.log(miArray[2]); // 3
console.log(miArray[4]); // true
```

  Arrays multidimensionales o Matrices: Los arrays en JavaScript pueden contener otros arrays como elementos, lo que permite crear estructuras de datos multidimensionales. Por ejemplo:

```
let matriz = [
    [1, 2, 3],
    [4, 5, 6],
```

```
        [7, 8, 9]
];
```

Métodos de arrays: JavaScript proporciona una variedad de métodos incorporados para manipular arrays, como push(), pop(), shift(), unshift(), concat(), slice(), splice(), indexOf(), includes(), join(), entre otros. Estos métodos permiten agregar, eliminar, modificar y acceder a elementos de un array de manera fácil y eficiente.

- ✓ push(): Agrega uno o más elementos al final de un array y devuelve la nueva longitud del array.

```
let frutas = ["manzana", "banana"];
frutas.push("naranja");
// frutas ahora es ["manzana", "banana", "naranja"]
```

- ✓ pop(): Elimina el último elemento de un array y devuelve ese elemento.

```
let frutas = ["manzana", "banana", "naranja"];
let ultimo = frutas.pop();
// frutas ahora es ["manzana", "banana"]
// último es "naranja"
```

- ✓ shift(): Elimina el primer elemento de un array y devuelve ese elemento.

```
let frutas = ["manzana", "banana", "naranja"];
let primero = frutas.shift();
// frutas ahora es ["banana", "naranja"]
// primero es "manzana"
```

- ✓ unshift(): Agrega uno o más elementos al inicio de un array y devuelve la nueva longitud del array.

```
let frutas = ["banana", "naranja"];
frutas.unshift("manzana");
// frutas ahora es ["manzana", "banana", "naranja"]
```

- ✓ concat(): Combina dos o más arrays y devuelve un nuevo array.

```
let frutas = ["manzana", "banana"];
let verduras = ["espinaca", "zanahoria"];
let comida = frutas.concat(verduras);
// comida es ["manzana", "banana", "espinaca", "zanahoria"]
```

- ✓ slice(): Devuelve una copia superficial de una porción de un array como un nuevo array.

```
let frutas = ["manzana", "banana", "naranja", "uva", "sandía"];
let citricas = frutas.slice(2, 4);
// cítricas es ["naranja", "uva"]
```

- ✓ splice(): Cambia el contenido de un array eliminando elementos existentes y/o agregando nuevos elementos.

```
let frutas = ["manzana", "banana", "naranja", "uva"];
frutas.splice(2, 1, "pera", "mango");
// frutas ahora es ["manzana", "banana", "pera", "mango", "uva"]
```

- ✓ indexOf(): Este método busca un valor específico dentro de un array y devuelve el índice de la primera ocurrencia del valor, o -1 si el valor no está presente.

```
let frutas = ["manzana", "banana", "naranja", "uva"];
console.log(frutas.indexOf("uva")); // Salida: 3
```

- ✓ Includes(): Este método determina si un array contiene un determinado elemento y devuelve true o false según el caso.

```
let frutas = ["manzana", "banana", "naranja", "uva"];
console.log(frutas.indexOf("uva")); // Salida: true
console.log(frutas.indexOf("melon")); // Salida: false
```

- ✓ join(): Este método se utiliza para unir todos los elementos de un array en una sola cadena, con un separador especificado entre cada elemento.

```
// Unir elementos de un array en una cadena con un espacio como separador
let palabrasArray = ["Hola", "mundo,", "cómo", "estás?"];
let cadena = palabrasArray.join(" ");
console.log(cadena); // "Hola mundo, cómo estás?"
```

- Objetos.

  En JavaScript, un objeto es una estructura de datos que permite almacenar colecciones de datos y funcionalidades asociadas. Los objetos se componen de propiedades, que son pares clave-valor, y de métodos, que son funciones que operan sobre los datos del objeto. El valor de una propiedad puede ser un dato primitivo (como un número, cadena o booleano), una función (en cuyo caso se llama método), u otro objeto.

  Un objeto se puede crear de varias maneras:
  - ✓ Directamente, con una expresión (notación de objeto literal).
  - ✓ Con Object.create o new Object().
  - ✓ A partir de una function.
  - ✓ A partir de una class.

  En cualquier caso, recuerda que los objetos que creas viven dentro de un objeto global de Javascript, y heredan propiedades y métodos de este objeto global. Propiedades y métodos que puedes sobreescribir.

  Notación de objeto literal: La forma más común y sencilla de crear un objeto es utilizando la notación de objeto literal.

```
const persona = {
    nombre: "Juan",
    edad: 30,
    saludo: function() {
        console.log("Hola, me llamo " + this.nombre);
```

```
    }
};

console.log(persona.nombre); // "Juan"
console.log(persona.edad); // 30
persona.saludo(); // "Hola, me llamo Juan"
```

new Object(): Otra manera de crear un objeto es utilizando el constructor Object, aunque esta forma es menos común.

```
let persona = new Object();
persona.nombre = "Juan";
persona.edad = 30;
persona.saludo = function() {
  console.log("Hola, me llamo " + this.nombre);
};

console.log(persona.nombre); // "Juan"
console.log(persona.edad); // 30
persona.saludo(); // "Hola, me llamo Juan"
```

Las propiedades de un objeto pueden ser accedidas usando la notación de punto o la notación de corchetes.

```
let persona = {
  nombre: "Juan",
  edad: 30
};

// Notación de punto
console.log(persona.nombre); // "Juan"
// Notación de corchetes
console.log(persona["edad"]); // 30
```

- Operadores lógicos y operadores de comparación.

Los operadores lógicos comparan valores booleanos y devuelven respuestas booleanas. Estos se utilizan para combinar o invertir valores booleanos (verdadero o falso). Estos operadores son esenciales para controlar el flujo de un programa y tomar decisiones basadas en múltiples condiciones. Los principales operadores lógicos en JavaScript son **&&** (AND), **||** (OR), y **!** (NOT).

Operador lógico AND (**&&**)
Este operador lógico compara dos expresiones. Si la primera se evalúa como "verdadera" (truthy), la sentencia devolverá el valor de la segunda expresión. Si la primera expresión se evalúa como "falsa"(falsy), la sentencia devolverá el valor de la primera expresión.

Cuando solo se incluyen valores booleanos (true o false), se devuelve verdadero si las dos expresiones son verdaderas. Si una o ambas expresiones son falsas, la sentencia completa se devolverá como falsa.

```
let a = true;
```

```
let b = false;

console.log(a && b); // false
console.log(a && true); // true
console.log(b && false); // false
```

Operador lógico OR (**||**)
Este operador lógico compara dos expresiones. Si la primera se evalúa como "falsa", la sentencia devolverá el valor de la segunda expresión. Si la primera se evalúa como "verdadera", la sentencia devolverá el valor de la primera expresión.

Cuando solo se incluyen valores booleanos (true o false), se devuelve como true si cualquiera de las dos expresiones es verdadera. Ambas expresiones pueden ser verdaderas, pero solo se necesita una para que el resultado sea verdadero.

```
    let a = true;
let b = false;

console.log(a || b); // true
console.log(a || false); // true
console.log(b || false); // false
```

NOTA: Como se pudo observar previamente cuando los operadores lógicos comparan dos o más expresiones en JavaScript, se utiliza la técnica llamada “evaluación de cortocircuito”. Esto significa que la evaluación se detiene tan pronto como el resultado de la expresión esté determinado.

```
let a = false;
let b = true;

console.log(a && (b = false)); // false, b no se evalúa, sigue siendo true
console.log(b); // true
console.log(a || (b = false)); // true, b se evalúa a false
console.log(b); // false
```

Operador lógico NOT (**!**)
El operador lógico NOT no realiza ninguna comparación como lo hacen los operadores AND y OR. Además, se opera en solo un operando.

Se utiliza un símbolo "**!**" (signo de exclamación) para representar un operador NOT. El operador lógico NOT invierte el valor de su operando. Si el operando es true, devuelve false, y viceversa.

```
let a = true;
let b = false;
let c = “Laura y Eric”

console.log(!a); // false
console.log(!b); // true
console.log(!c); // false
```

Usos del operador NOT:

- ✓ Convertir la expresión en un booleano.
- ✓ Devolver el valor contrario del booleano obtenido en el último paso.

Los operadores de comparación se utilizan para comparar dos valores y devolver un valor booleano (true o false) basado en la relación entre ellos. Los principales operadores de comparación en JavaScript son: **==** (Igualdad), **===** (Estrictamente igual), **!=** (Desigualdad), **!==** (Estrictamente desigual), **>** (Mayor que), **>=** (Mayor o igual que), **<** (Menor que), **<=** (Menor o igual que), **isNaN()** (Compara si el valor es un Not-a-Number).

Igualdad (**==**)
El operador **==** compara dos valores para determinar si son iguales. Si los valores tienen diferentes tipos, el operador == realiza una conversión de tipo implícita para intentar igualar los valores antes de compararlos.

Conversión de tipo: Se produce una conversión de tipo cuando se comparan valores de diferentes tipos. Por ejemplo, el número 5 se convierte en la cadena "5" y viceversa antes de la comparación.

Casos especiales: null y undefined se consideran iguales cuando se comparan con **==**.

```
console.log(5 == '5'); // true, se convierte '5' a número antes de comparar
console.log('0' == false); // true, se convierte '0' a 0 y false a 0 antes de comparar
console.log(null == undefined); // true, estos dos son considerados iguales
console.log(NaN == NaN); // false, NaN no es igual a nada, ni siquiera a sí mismo
```

Estrictamente Igual (**===**)
El operador **===** compara dos valores para determinar si son estrictamente iguales, es decir, si tienen el mismo valor y el mismo tipo sin realizar ninguna conversión de tipo.

Sin conversión de tipo: No se realiza ninguna conversión de tipo antes de la comparación, por lo que los tipos deben coincidir para que el resultado sea true.

```
console.log(5 === '5'); // false, tipos diferentes
console.log(5 === 5); // true, mismos tipos y valores
console.log(null === undefined); // false, tipos diferentes
console.log(NaN === NaN); // false, NaN no es igual a nada
```

Desigualdad (**!=**)
El operador **!=** compara dos valores para determinar si no son iguales. Similar a ==, realiza una conversión de tipo si es necesario.

```
console.log(5 != '5'); // false, después de la conversión, los valores son iguales
console.log('hello' != 'world'); // true, los valores son diferentes
console.log(true != 1); // false, true se convierte a 1 antes de comparar
console.log(null != undefined); // false, se consideran iguales
```

Estrictamente Desigual (**!==**)
El operador **!==** compara dos valores para determinar si no son estrictamente iguales, es decir, si tienen diferentes valores o tipos sin realizar ninguna conversión de tipo.

```
console.log(5 !== '5'); // true, tipos diferentes
console.log(5 !== 5); // false, mismos tipos y valores
console.log(true !== 1); // true, tipos diferentes
console.log(null !== undefined); // true, tipos diferentes
```

Mayor Que (**>**)

El operador > compara dos valores para determinar si el primer valor es mayor que el segundo.

Strings: Los strings se comparan lexicográficamente basados en el valor Unicode de sus caracteres.

```
console.log(10 > 5); // true
console.log('b' > 'a'); // true, 'b' tiene un valor Unicode mayor que 'a'
console.log('apple' > 'banana'); // false, 'a' tiene un valor Unicode menor que 'b'
console.log(false > -1); // true, false se convierte a 0 antes de comparar
```

Mayor o Igual Que (**>=**)

El operador > compara dos valores para determinar si el primer valor es mayor que el segundo.

```
console.log(10 >= 5); // true
console.log(5 >= 5); // true
console.log('b' >= 'a'); // true
console.log('apple' >= 'banana'); // false
```

Menor Que (**<**)

El operador < compara dos valores para determinar si el primer valor es menor que el segundo.

```
console.log(5 < 10); // true
console.log('a' < 'b'); // true
console.log('apple' < 'banana'); // true
console.log(true < 2); // true, true se convierte a 1 antes de comparar
```

Menor o Igual Que (**<=**)

El operador <= compara dos valores para determinar si el primer valor es menor o igual que el segundo.

```
console.log(5 <= 10); // true
console.log(5 <= 5); // true
console.log('a' <= 'b'); // true
console.log('apple' <= 'banana'); // true
```

NOTA: Comparación de Objetos: Cuando se comparan objetos con cualquiera de estos operadores, se compara su referencia en la memoria, no sus valores internos. Dos objetos distintos con los mismos valores internos no son considerados iguales.

```
let obj1 = { key: 'value' };
let obj2 = { key: 'value' };
let obj3 = obj1;

console.log(obj1 == obj2); // false, diferentes referencias en memoria
console.log(obj1 === obj2); // false, diferentes referencias en memoria
console.log(obj1 == obj3); // true, misma referencia en memoria
console.log(obj1 === obj3); // true, misma referencia en memoria
```

Comparar la igualdad de objetos en JavaScript:
https://www.freecodecamp.org/espanol/news/operadores-de-comparacion-en-javascript-como-comparar-la-igualdad-de-objetos-en-js/

- Funciones.

Las funciones son bloques de códigos diseñado para realizar una o varias tareas en particular, son fundamentales en JavaScript, por brindan la ventaja de ser reutilizables y permitir modularizar programas. Se utiliza la palabra clave “función” para definir una función, seguido del nombre de la misma.

Componentes de una función:

- ✓ function: Palabra clave para definir una función.
- ✓ nombreFuncion: Nombre de la función (opcional para funciones anónimas).
- ✓ parámetros: Lista de parámetros separados por comas (opcionales).
- ✓ cuerpo de la función: Código que se ejecuta cuando la función es llamada.
- ✓ return: Palabra clave para devolver un valor desde la función (opcional).

```
function sumar(a, b) {
  return a + b;
}

let resultado = sumar(3, 4); // Llamada a la función
console.log(resultado); // 7
```

Tipos de funciones:

Funciones Declaradas: Las funciones declaradas se definen usando la sintaxis básica function y son elevadas (hoisted) al principio de su contexto de ejecución.

```
function saludar() {
    console.log("Hola!");
}

saludar(); // "Hola!"
```

Funciones Expresadas: Las funciones expresadas se definen mediante expresiones y no son elevadas. Pueden ser anónimas o nombradas.

```
// Función anónima
const sumar = function(a, b) {
   return a + b;
};

// Función nombrada
const restar = function restar(a, b) {
  return a - b;
};

console.log(sumar(5, 3)); // 8
console.log(restar(5, 3)); // 2
```

Funciones Flecha (Arrow Functions): Las funciones flecha son una sintaxis más corta para escribir funciones. Si la función tiene solamente una sentencia que devuelve un valor, el uso de funciones flecha nos permite eliminar las llaves y la palabra return. Incluso utilizando parámetros también podemos ver mucho más reducido el código.

```
// Sintaxis básica de una función flecha
const multiplicar = (a, b) => a * b;

// Si la función tiene un solo parámetro, se pueden omitir los paréntesis
const cuadrado = x => x * x;

// Para funciones con más de una línea en el cuerpo, se necesitan llaves y return
const sumarYMultiplicar = (a, b) => {
  let suma = a + b;
  return suma * 2;
};

console.log(sumarYMultiplicar(2, 3)); // 10
```

Las funciones pueden tener parámetros:

- ✓ Predeterminados, estos se asignan utilizando el operador de asignación "=".

```
function saludar(nombre = "Desconocido") {
    console.log("Hola, " + nombre);
}

saludar(); // "Hola, Desconocido"
saludar("Juan"); // "Hola, Juan"
```

- ✓ Rest, el operador rest (**...**) permite representar un número indefinido de argumentos como un array.

```
function sumarTodos(...numeros) {
    return numeros.reduce((acumulador, actual) => acumulador + actual, 0);
}

console.log(sumarTodos(1, 2, 3, 4)); // 10
```

NOTA: Dentro de las características avanzadas de las funciones, te invito a investigar sobre las funciones anidadas, las clausuras (closures), funciones de orden superior (HOF), y funciones como objetos y sus métodos.

- Clases.

Tradicionalmente, JavaScript no soportaba clases de forma nativa, pero desde la norma de ECMAScript5 (2015) puede considerarse un lenguaje de programación orientado a objetos (POO), pero basado en prototipos, con algunas peculiaridades.

Las clases en JavaScript, al igual que los objetos y las funciones, son una parte fundamental de la sintaxis de este lenguaje de programación. Con ellas, podemos crear distintos objetos con características específicas, permitiéndonos crear un código más eficiente. No posee todas las características de otros lenguajes orientados a objetos como Java o C++, pero si es capaz de manejar y crear objetos. Además, permite la herencia, pero no la sobrecarga de funciones.

En resumen, JavaScript no tiene clases solo definiciones de objetos, constructores, que se usan como modelos para construir otros objetos. Estos objetos padre o modelos serían lo más parecido a las clases de la programación orientada a objetos. La sintaxis de la clase en JavaScript es como azúcar sintáctico, es decir, un sistema que «endulza» la forma de trabajar para que sea más agradable y familiar.

Herencia: Al hablar de heredar se habla de que los objetos descendientes pueden acceder a propiedades y métodos del objeto desde donde se crearon (objeto padre). No se trata de que reciba copias de su constructor sino referencias a sus propiedades y métodos. Esto lo implementa mediante prototype.

Ejemplo de herencia en clases:

```
class Animal {
    constructor(nombre) {
        this.nombre = nombre;
    }

    hacerSonido() {
        console.log(`${this.nombre} está haciendo un sonido.`);
    }
}
```

Clase Derivada: Perro
Luego, definimos la clase Perro que extiende de Animal y añade sus propios métodos:

```
class Perro extends Animal {
    constructor(nombre, edad, raza) {
        super(nombre, edad); // Llama al constructor de la clase base
        this.raza = raza;
    }

    ladrar() {
        console.log(`${this.nombre} está ladrando.`);
    }
}
```

Crear Instancias y Usar Métodos
Finalmente, creamos instancias de Animal y Perro, y usamos sus métodos para ver cómo funcionan la herencia y la sobreescritura de métodos.

```
const miAnimal = new Animal('Simba');
miAnimal.hacerSonido(); // Simba está haciendo un sonido.

const miPerro = new Perro('Max', 'Labrador');
miPerro.hacerSonido(); // Max está haciendo un sonido. (Método heredado de Animal)
miPerro.ladrar(); // Max está ladrando.
```

This: Es una referencia que se utiliza en una función para referirse al objeto que la invocó. En el contexto de clases en JavaScript, this se refiere a la instancia de la clase sobre la que se está trabajando. Es decir, dentro de los métodos de una clase, this hace referencia al objeto que ha

sido creado a partir de esa clase. A continuación, se detalla cómo se utiliza this en diferentes partes de una clase y su comportamiento:

Constructor: En el constructor de una clase, this se refiere a la instancia que está siendo creada.

```
class Coche {
    constructor(marca, modelo) {
        this.marca = marca;
        this.modelo = modelo;
    }
}
```

Aquí, this.marca y this.modelo se refieren a las propiedades de la instancia que está siendo creada.

Métodos: Dentro de los métodos de la clase, this se refiere a la instancia que llama al método.

```
class Coche {
    constructor(marca, modelo) {
        this.marca = marca;
        this.modelo = modelo;
    }
    arrancar() {
        console.log(`${this.marca} ${this.modelo} está arrancando.`);
    }
}

const miCoche = new Coche('Toyota', 'Corolla');
miCoche.arrancar(); // Toyota Corolla está arrancando.
```

En el método arrancar, this.marca y this.modelo se refieren a las propiedades de la instancia miCoche.

Prototype: Los prototipos son un conjunto de normas para integrar Programación Orientada a Objetos en JavaScript. El prototipo es una propiedad de nombre prototype, que es en sí un objeto, denominado objeto prototipo, y que reside en la función constructor del objeto. A través de esta propiedad prototype es que podemos agregarle al objeto nuevas propiedades y métodos.

Un objeto prototipo puede tener a su vez otro objeto prototipo del cual hereda, lo que se conoce como cadena de prototipos. Esto permite que los objetos puedan tener propiedades y métodos que no han sido declarados por ellos mismos.

La herencia de prototipos funciona de la siguiente manera:

- ✓ Los objetos Date heredan de **Date.prototype**
- ✓ Los objetos Number heredan de **Number.prototype**
- ✓ Los objetos Array heredan de **Array.prototype**
- ✓ Entre otros.

A su vez, todos los objetos heredan de Object.prototype, que se encuentra en lo más alto de la cadena de prototipos.

Si exploramos por ejemplo el prototipo Date.prototype podemos ver algunos de los métodos que serán accesibles a cada instancia de Date:

```
> Date.prototype
< ▼{constructor: ƒ, toString: ƒ, toDateString: ƒ, toTimeString: ƒ, toISOString: ƒ, …}
    ▶ constructor: ƒ Date()
    ▶ toString: ƒ toString()
    ▶ toDateString: ƒ toDateString()
    ▶ toTimeString: ƒ toTimeString()
    ▶ toISOString: ƒ toISOString()
    ▶ toUTCString: ƒ toUTCString()
    ▶ toGMTString: ƒ toUTCString()
    ▶ getDate: ƒ getDate()
    ▶ setDate: ƒ setDate()
    ▶ getDay: ƒ getDay()
    ▶ getFullYear: ƒ getFullYear()
    ▶ setFullYear: ƒ setFullYear()
    ▶ getHours: ƒ getHours()
    ▶ setHours: ƒ setHours()
    ▶ getMilliseconds: ƒ getMilliseconds()
    ▶ setMilliseconds: ƒ setMilliseconds()
    ▶ getMinutes: ƒ getMinutes()
    ▶ setMinutes: ƒ setMinutes()
    ▶ getMonth: ƒ getMonth()
    ▶ setMonth: ƒ setMonth()
    ▶ getSeconds: ƒ getSeconds()
    ▶ setSeconds: ƒ setSeconds()
    ▶ getTime: ƒ getTime()
    ▶ setTime: ƒ setTime()
    ▶ getTimezoneOffset: ƒ getTimezoneOffset()
    ▶ getUTCDate: ƒ getUTCDate()
    ▶ setUTCDate: ƒ setUTCDate()
    ▶ getUTCDay: ƒ getUTCDay()
    ▶ getUTCFullYear: ƒ getUTCFullYear()
    ▶ setUTCFullYear: ƒ setUTCFullYear()
    ▶ getUTCHours: ƒ getUTCHours()
    ▶ setUTCHours: ƒ setUTCHours()
```

```
const ahora = new Date();

/**************************************
'ahora' hereda el prototipo de Date
y por lo tanto podemos usar sus métodos
**************************************/
console.log(ahora.getMonth()); // devuelve el número de mes
```

Ahora veámoslo en una función constructor propia:

```
// Declaro un constructor
function Automovil (marca, modelo, color) {
  this.marca = marca;
  this.modelo = modelo;
  this.color = color;
}

// Lo instancio
var miAuto = new Automovil('Tesla', 'Cybertruck', 'Negro');

// Agrego la propiedad ruedas a la propiedad prototype del constructor
Automovil.prototype.ruedas = 4;

// La instancia de Automovil hereda la propiedad 'ruedas' del prototype
console.log(miAuto.color); // "Negro"
console.log(miAuto.ruedas); // 4
```

En este ejemplo, la instancia miAuto del objeto Automovil hereda propiedades y métodos de Automovil.prototype.

Las instancias de objetos, como miAuto, heredan además una propiedad __proto__ que referencia al prototype de la función constructor, en este caso Automovil.prototype.

Automovil.prototype === miAuto.__proto__; // true

NOTA: Automovil.prototype.ruedas = 4 hace que la propiedad ruedas esté disponible en todas las instancias de Automovil, no es lo mismo que la siguiente sintaxis: Automovil.ruedas = 4, esto hace que la propiedad ruedas esté disponible solo en el constructor Automovil como una propiedad estática y no en todas las instancias.

Veamos otro ejemplo:

```
class Rectangle {
  constructor(height, width) {
    this.height = height
    this.width = width
  }

  get area() {
    return this.calcArea()
  }

  calcArea() {
    return this.height * this.width
  }
}

const square = new Rectangle(10, 10)

console.log(square.area) // 100
```

```
function Rectangle(height, width) {
  this.height = height
  this.width = width
}

Rectangle.prototype.calcArea = function calcArea() {
  return this.height * this.width
}
```

Ambos ejemplos son equivalentes.

Los métodos getter y setter en las clases vinculan una propiedad Object a una función que será llamada cuando se busque esa propiedad. Esto es solo azúcar sintáctico para ayudar a que sea más fácil buscar o establecer propiedades.

NOTA: Investigar los métodos get y set de las clases.

- Condicionantes.
  En JavaScript, los condicionantes (también conocidos como estructuras de control condicional) se utilizan para ejecutar diferentes bloques de código en función de ciertas condiciones. Estas se representan con las siguientes declaraciones: if, else if y else, switch y operador ternario.

  La declaración **if** se utiliza para ejecutar un bloque de código si una condición especificada es verdadera. La declaración **else if** se puede usar para probar múltiples condiciones y la declaración **else** se ejecuta si ninguna de las condiciones anteriores es verdadera.

```
if (condición) {
  // Bloque de código a ejecutar si la condición es verdadera
} else if (otraCondición) {
  // Bloque de código a ejecutar si la otra condición es verdadera
} else {
  // Bloque de código a ejecutar si ninguna de las condiciones anteriores es verdadera
}
```

  Switch se utiliza para ejecutar diferentes bloques de código en función del valor de una expresión. Es una alternativa más estructurada y legible que múltiples declaraciones if else if.

```
switch (expresión) {
  case valor1:
    // Bloque de código a ejecutar si expresión === valor1
  break;
  case valor2:
    // Bloque de código a ejecutar si expresión === valor2
  break;
   // … Añadir más casos según sea necesario
  default:
    // Bloque de código a ejecutar si ninguno de los casos anteriores coincide
}
```

  Operador Ternario: El operador ternario es una forma abreviada de la declaración if else que puede ser útil para asignaciones simples o retornos de funciones.
  condición ? expresiónSiVerdadera : expresiónSiFalsa;

```
const esMayorDeEdad = edad >= 18 ? 'Sí, es mayor de edad.' : 'No, es menor de edad.';
console.log(esMayorDeEdad); // Sí, es mayor de edad.
```

- Ciclos (Bucles).
  Los bucles son estructuras que permiten ejecutar repetidamente un bloque de código mientras se cumple una condición específica. Los bucles son esenciales para manejar tareas repetitivas y operar sobre colecciones de datos.

En JavaScript, existen varios tipos de bucles, cada uno con su sintaxis y uso específico. Vamos a dividirlos por dos secciones que son los condicionantes y los de arreglos.

Ciclos condicionantes:

for: El bucle for es uno de los bucles más comunes y se utiliza cuando sabes de antemano cuántas veces necesitas iterar. Está compuesto por tres partes: inicialización, condición y actualización.

```
for (inicialización; condición; actualización) {
        // código a ejecutar
}
```

- ✓ Inicialización: Se ejecuta una vez al comienzo del bucle y se utiliza para declarar e inicializar variables.
- ✓ Condición: Se evalúa antes de cada iteración. Si es verdadera, el bucle se ejecuta; si es falsa, el bucle termina.
- ✓ Actualización: Se ejecuta al final de cada iteración y se utiliza para actualizar la variable de control.

```
for (let i = 0; i < 5; i++) {
        console.log(i);
}
// Salida: 0, 1, 2, 3, 4
```

En este ejemplo, let i = 0 inicializa la variable i a 0, i < 5 es la condición que debe ser verdadera para que el bucle continúe, y i++ incrementa i en 1 después de cada iteración.

for in: El bucle for...in se utiliza para iterar sobre todas las propiedades enumerables de un objeto. Es ideal para trabajar con objetos donde necesitas acceder a las claves.

```
const respuestas = { a: 1, b: 2, c: 3 };
for (let propiedad in respuestas) {
        console.log(`${propiedad}: ${respuestas[propiedad]}`);
}
 // Salida: // a: 1
            // b: 2
            // c: 3
```

Aquí, propiedad toma el nombre de cada propiedad del objeto “respuestas” en cada iteración, y se puede acceder a su valor usando respuestas[propiedad].

for of: El bucle for...of se usa para iterar sobre valores de objetos iterables como arrays, strings, maps, sets, etc. Es más conveniente y legible que un for tradicional cuando trabajas con iterables.

```
const array = [10, 20, 30];
for (let valor of array) {
        console.log(valor);
}
```

```
// Salida: 10, 20, 30
```

En este ejemplo, valor toma el valor de cada elemento en array en cada iteración.

while: El bucle while ejecuta su bloque de código siempre que la condición especificada sea verdadera. Es útil cuando no sabes cuántas veces necesitarás iterar de antemano y solo quieres que el bucle se ejecute mientras una condición sea verdadera.

```
let i = 0;
 while (i < 5) {
        console.log(i);
        i++;
 }
// Salida: 0, 1, 2, 3, 4
```

Aquí, i < 5 es la condición que se evalúa antes de cada iteración. Si es verdadera, se ejecuta el bloque de código; si es falsa, el bucle se detiene.

do while: El bucle do...while es similar al while, pero con una diferencia clave: garantiza que el bloque de código se ejecutará al menos una vez, ya que la condición se evalúa después de ejecutar el bloque de código.

```
let i = 0;
do {
      console.log(i);
      i++;
} while (i < 5);
// Salida: 0, 1, 2, 3, 4
```

En este ejemplo, console.log(i) se ejecuta primero, y luego se evalúa i < 5. El bucle se repite mientras la condición sea verdadera.

En resumen: Los bucles son herramientas poderosas en JavaScript que permiten manejar colecciones de datos y realizar operaciones repetitivas de manera eficiente. El for se usa para iteraciones con un número conocido de veces, el while y do...while para condiciones booleanas, y for...in y for...of para iterar sobre propiedades de objetos y elementos de iterables, respectivamente. Cada tipo de bucle tiene su lugar dependiendo del contexto y la estructura de datos con la que estés trabajando.

Ciclos de Arreglos:

forEach(): El método forEach es específico para arrays y se utiliza para ejecutar una función una vez por cada elemento de un array. Es una alternativa más legible y concisa a un bucle for.

```
const array = [10, 20, 30];
array.forEach((valor, índice) => {
   console.log(`Índice ${índice}: ${valor}`);
});
```

```
// Salida:
// Índice 0: 10
// Índice 1: 20
// Índice 2: 30
```

map(): El método map crea un nuevo array con los resultados de aplicar una función a cada uno de los elementos del array original. No modifica el array original.

Syntaxis: array.map(callback(elemento, índice, array), thisArg)

- callback: Función que se ejecuta para cada elemento del array.
- elemento: El elemento actual que está siendo procesado.
- índice (opcional): El índice del elemento actual.
- array (opcional): El array sobre el cual se está llamando map().
- thisArg (opcional): Valor para usar como this al ejecutar el callback.

```
const numeros = [1, 2, 3, 4];
const cuadrados = numeros.map((elemento, indice) => `{${indice}: ${ elemento * elemento }}`);
console.log(cuadrados); // Salida: [{0: 1}', '{1: 4}', '{2: 9}', '{3: 16}]
```

Ejemplo usando el thisArg:

```
const multiplicador = {
        factor: 3,
};

const numeros = [1, 2, 3];
const multiplicados = numeros.map(function(numero) {
        return numero * this.factor;
}, multiplicador);
console.log(multiplicados); // [3, 6, 9]
```

En este caso, thisArg se utiliza para pasar el objeto multiplicador como el contexto (this) para la función de callback. Cada número en el array numeros se multiplica por el factor definido en multiplicador.

filter(): El método filter crea un nuevo array con todos los elementos que pasen la prueba implementada por la función proporcionada. No modifica el array original.

Syntaxis: array.map(callback(elemento, índice, array), thisArg)

- callback: Función que se ejecuta para cada elemento del array.
- elemento: El elemento actual que está siendo procesado.
- índice (opcional): El índice del elemento actual.
- array (opcional): El array sobre el cual se está llamando map().
- thisArg (opcional): Valor para usar como this al ejecutar el callback.

```
const numeros = [1, 2, 3, 4, 5, 6];
const pares = numeros.filter(num => num % 2 === 0);
console.log(pares); // Salida: [2, 4, 6]
```

find(): El método find devuelve el primer elemento del array que cumpla con la función de prueba proporcionada. Si ningún elemento pasa la prueba, devuelve undefined.

Syntaxis: array.map(callback(elemento, índice, array), thisArg)

- callback: Función que se ejecuta para cada elemento del array.
- elemento: El elemento actual que está siendo procesado.
- índice (opcional): El índice del elemento actual.
- array (opcional): El array sobre el cual se está llamando map().
- thisArg (opcional): Valor para usar como this al ejecutar el callback.

```
const personas = [
    { nombre: 'Juan', edad: 28 },
    { nombre: 'Ana', edad: 22 },
    { nombre: 'Luis', edad: 32 }
];
const joven = personas.find(persona => persona.edad < 30);
console.log(joven); // Salida: { nombre: 'Juan', edad: 28 }
```

reducer(): El método reduce aplica una función a un acumulador y a cada valor de la array (de izquierda a derecha) para reducirlo a un único valor. Este método no modifica el array original.

Syntaxis: array.reduce(callback(acumulador, valorActual, índice, array), valorInicial)

- callback: Función que se ejecuta para cada elemento del array.
- acumulador: Acumulador que recoge el valor devuelto en la última invocación del callback.
- valorActual: El elemento actual que está siendo procesado.
- índice (opcional): El índice del elemento actual.
- array (opcional): El array sobre el cual se está llamando reduce().
- valorInicial (opcional): Valor para usar como primer argumento al primer llamamiento del callback.

```
const numeros = [1, 2, 3, 4];
const suma = numeros.reduce((acumulador, valorActual) => acumulador + valorActual, 0);
console.log(suma); // Salida: 10
```

En resumen: Estos métodos son muy potentes y permiten escribir código más claro y funcional, aprovechando las capacidades de programación funcional de JavaScript. Entre los beneficios de estos métodos están: Mejoran la eficiencia, la legibilidad y la mantenibilidad del código, el poder encadenar un método con otros métodos, entre otros. Adoptar estos métodos puede llevar a un desarrollo más eficiente y a un código que es más fácil de entender y menos propenso a errores.

- Funciones de fechas y números.

Estas funciones y métodos te proporcionan una variedad de herramientas para manipular y formatear números y fechas en JavaScript de manera efectiva y eficiente. En este apartado ilustraremos la mayor parte solo con ejemplos:

Funciones de fechas: El objeto Date proporciona métodos para trabajar con fechas y horas.

Fecha Actual:

```
let now = new Date();
console.log(now);
```

Fecha Especifica:

```
let specificDate = new Date(2023, 5, 11); // Año, Mes (0-11), Día
console.log(specificDate);
```

getFullYear(): Devuelve el año (4 dígitos) de la fecha:

```
let now = new Date();
console.log(now.getFullYear()); // 2024
```

getMonth(): Devuelve el mes (0-11) de la fecha:

```
console.log(now.getMonth()); // 5 (junio)
```

getDate(): Devuelve el día del mes (1-31):

```
console.log(now.getDate()); // 11
```

getDay(): Devuelve el día de la semana (0-6):

```
console.log(now.getDay()); // 2 (martes)
```

getHours(): Devuelve la hora (0-23):

```
console.log(now.getHours()); // 14 (por ejemplo)
```

getMinutes(): Devuelve los minutos (0-59):

```
console.log(now.getMinutes()); // 30 (por ejemplo)
```

getSeconds(): Devuelve los segundos (0-59):

```
console.log(now.getSeconds()); // 45 (por ejemplo)
```

getMilliseconds(): Devuelve los milisegundos (0-999):

```
console.log(now.getMilliseconds()); // 500 (por ejemplo)
```

getTime(): Devuelve el número de milisegundos desde el 1 de enero de 1970:

```
console.log(now.getTime()); // 1591933445000 (por ejemplo)
```

Métodos para Establecer Fecha y Hora:

setFullYear(year, [month], [day]): Establece el año, y opcionalmente el mes y el día.

```
let date = new Date();
date.setFullYear(2025);
console.log(date.getFullYear()); // 2025
```

setHours(hour, [min], [sec], [ms]): Establece la hora, y opcionalmente los minutos, segundos y milisegundos.

```
date.setHours(10, 30, 0, 0);
console.log(date.getHours()); // 10
```

```
console.log(date.getMinutes()); // 30
```

Otras Funciones Útiles de Date:

toLocaleDateString([locales], [options]): Devuelve una cadena con la fecha formateada de acuerdo a las convenciones locales.

```
let date = new Date();
console.log(date.toLocaleDateString('es-ES')); // "11/6/2024"
```

toLocaleTimeString([locales], [options]): Devuelve una cadena con la hora formateada de acuerdo a las convenciones locales.

```
console.log(date.toLocaleTimeString('es-ES')); // "14:30:45"
```

toISOString(): Devuelve una cadena en formato ISO 8601.

```
console.log(date.toISOString()); // "2024-06-11T14:30:45.500Z"
```

Funciones de números: JavaScript ofrece una variedad de funciones y métodos para manipular y trabajar con números. Estas funciones son parte del objeto global Math y del objeto Number. Acontinuacion veremos algunos ejemplos:

Math:

Math.ceil(): Redondea un número hacia arriba al entero más cercano.

```
console.log(Math.ceil(4.2)); // 5
```

Math.floor(): Redondea un número hacia abajo al entero más cercano.

```
console.log(Math.floor(4.7)); // 4
```

Math.round(): Redondea un número al entero más cercano.

```
console.log(Math.round(4.5)); // 5
console.log(Math.round(4.4)); // 4
```

Math.max(): Devuelve el mayor de cero o más números.

```
console.log(Math.max(1, 2, 3)); // 3
```

Math.min(): Devuelve el menor de cero o más números.

```
console.log(Math.min(1, 2, 3)); // 1
```

Math.random(): Devuelve un número pseudoaleatorio entre 0 (incluido) y 1 (excluido).

```
console.log(Math.random()); // 0.123456789 (varía cada vez)
```

Math.sqrt(): Devuelve la raíz cuadrada de un número.

```
console.log(Math.sqrt(16)); // 4
```

Math.pow(): Devuelve la base elevada al exponente.

```
console.log(Math.pow(2, 3)); // 8
```

Number:

Number.isNaN(): Determina si el valor es NaN (Not-A-Number).

```
console.log(Number.isNaN(NaN)); // true
console.log(Number.isNaN(123)); // false
```

Number.isFinite(): Determina si el valor es un número finito.

```
console.log(Number.isFinite(123)); // true
console.log(Number.isFinite(Infinity)); // false
```

Number.isInteger(): Determina si el valor es un número entero.

```
console.log(Number.isInteger(4)); // true
console.log(Number.isInteger(4.5)); // false
```

Number.parseInt(): Convierte una cadena a un número entero.

```
console.log(Number.parseInt('123')); // 123
console.log(Number.parseInt('123.45')); // 123
```

Number.parseFloat(): Convierte una cadena a un número de punto flotante.

```
console.log(Number.parseFloat('123.45')); // 123.45
```

Ejemplo Combinado: Generar un número aleatorio entre un rango específico

```
function getRandomInt(min, max) {
    return Math.floor(Math.random() * (max - min + 1)) + min;
}
console.log(getRandomInt(1, 10)); // Número aleatorio entre 1 y 10
```

- Herramientas especiales dentro del código de JavaScript.
  Hay varias herramientas y métodos que son muy útiles para depuración, medición de rendimiento y otras tareas durante el desarrollo. Aquí hay una lista de algunas de las más importantes y cómo se utilizan:

  console.log(): Muestra un mensaje o un objeto en la consola.

  NOTA: dentro del objeto console, aparte de .log(), también podemos contar con:

  - ✓ .error()
  - ✓ .warn()
  - ✓ .info()
  - ✓ .table()
  - ✓ .dir()
  - ✓ .time() y timeEnd()
  - ✓ .group() y groupEnd()
  - ✓ .assert()
  - ✓ .clear()

  debugger: Pausa la ejecución del script y permite inspeccionar el código en las herramientas de desarrollo.

  alert(): Muestra un cuadro de alerta con un mensaje y un botón de aceptar.

  ```
  alert("Hola, Mundo!");
  ```

  confirm(): Muestra un cuadro de diálogo con un mensaje y botones de aceptar y cancelar.

  ```
  const respuesta = confirm("¿Estás seguro?");
  console.log(respuesta); // true si el usuario hace clic en "Aceptar", false en "Cancelar"
  ```

prompt(): Muestra un cuadro de diálogo que solicita al usuario que ingrese un texto.

```
const nombre = prompt("¿Cuál es tu nombre?");
console.log(nombre);
```

NOTA: Herramientas de Desarrollo del Navegador: La mayoría de los navegadores modernos (Chrome, Firefox, Edge) tienen herramientas de desarrollo integradas accesibles mediante F12 o Ctrl+Shift+I. Estas herramientas incluyen:

- Debugger: Para pausar y examinar el estado del código.
- Inspector: Para examinar y modificar el DOM y CSS.
- Console: Para interactuar con el JavaScript de la página.
- Network: Para analizar las solicitudes de red.
- Performance: Para medir el rendimiento del código.

Estas herramientas y métodos son esenciales para un desarrollo más eficiente y para la resolución de problemas en JavaScript. Usarlas adecuadamente puede mejorar significativamente tu flujo de trabajo y la calidad de tu código.

1.12 Ejercitando la mente:

NOTA: Para realizar los siguientes ejercicios debes descargar Node.js y Visual studio code (recomendado).
Node.js: https://nodejs.org/en
Visual studio code: https://code.visualstudio.com/

* Ejercicios con arreglos: El resultado de todos los ejercicios debe ser mostrador por consola.

1. Recorrer el arreglo: const numeros = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10], y guardar en una variable los números pares.
2. Recorrer el arreglo: const nombres = [“Heri”, “Julia”, “Jose”, “Pedro”, “Fleirin”, “Michael”, “Laura”, “Homero”], y crear dos espacios en memoria, una que guarde los nombres que solo tengan 4 letras, y otra que guarde los nombres en reverso.
3. Recorrer el arreglo: const user = [{nombre: “Daniel”, edad: 15}, {nombre: “Jorge”, edad: 18}, {nombre: “Diana”, edad: 30}, {nombre: “Alba”, edad: 35}], y agregar una propiedad nueva a cada objeto, llamada “esAdulto” y asignarle un valor booleano, dependiendo de la edad, si es igual o mayor a 18 asignar true, si no false.

* Agregando funciones: El resultado de todos los ejercicios debe ser mostrador por consola.

1. Crear una función que no haga hoister cuando se ejecute el documento, esta función recibirá como parámetro un número y devolverá el numero multiplicado por el mismo.
2. Crear una función anonima que haga hoister, no devuelva nada y muestre por consola “Hola, estoy aprendiendo”.
3. Crear una función anónima, de una sola línea, que reciba como parámetro una palabra cualquiera y retorne la misma palabra en UPPERCASE.
4. Const options = [{option: 1, nombre: “UPPERCASE”}, {option: 2, nombre: “lowercase”}, {option: 3, nombre: “Capitalize”}]. Crear una función que reciba como primer parámetro un arreglo (validar: que sea un arreglo de string), y segundo parámetro una opción (validar: que sea de tipo number y que el valor este dentro de las opciones que tiene el arreglo de objeto

"options"), y retornar un nuevo arreglo con el cambio aplicado a todos los valores del arreglo, dependiendo la opción que le envien.

* Agregando clases: El resultado de todos los ejercicios debe ser mostrador por consola.

1. Crear una clase de nombre "Animal", que tenga un constructor que reciba las propiedades nombre, edad y raza (validar: edad sea de tipo número). Esta clase debe tener dos métodos, uno llamado "esAdulto" que reciba dos parámetros "edadAnimal" y "nombreAnimal" (validar: si el animal tiene una edad de 2 o más años es adulto, sino es bebe), otro llamado "corre", que retorne un mensaje (string) que indique que el animal puede y está corriendo, este método se alimentara de las variables dentro del entorno de la clase (validar: para que el perro pueda correr debe ser adulto). (instanciar y ejecutar los dos métodos de la class Animal).

2. Crear una clase "Perro" que extienda de la clase Animal anteriormente creada, y que tenga una nueva propiedad llamada "color", la cual asignes un valor en la instancia de la misma. (instanciar y ejecutar los dos métodos heredados de la class Animal).

3. Crear una clase "HijoPerro" que extienda de la clase Perro anteriormente creada, crear un método para mostrar todas las informaciones insertadas llamado "mostrar" y que le asignes todos los valores requeridos en las dos clases anteriores. La edad debe ser (instanciar y ejecutar los dos métodos heredados).

¿Qué es HTML?

- Etiquetas.
- Estructura.

¿Qué es CSS?

- Selectores.
- Especificidad.
- Propiedades.

2. Practicas básicas de HTML, CSS y ejercicios con JS:

- Proyecto con HTML.
- Proyecto con HTML y CSS.
- Ejercicios con JS.

3. Trabajando con eventos y el internet:

¿Qué es BOM?

¿Qué es DOM?

¿Que son los Eventos?

¿Qué es la Propagación?

¿Qué es el internet?

¿Qué es la Web?

¿Qué es HTTP / HTTPS?

¿Qué es la Asincronía?

¿Peticiones AJAX?

¿Qué son las APIs?

¿Qué son las APIs?

4. *Continuación de prácticas con HTML, CSS, y JS:*

- Proyecto con HTML, CSS y JS
- Proyecto con HTML, CSS, JS y peticiones AJAX

5. *Conociendo el lado del servidor:*

¿Qué es una computadora?

¿Qué es un servidor?

Los servidores web esperan los mensajes de petición de los clientes, los procesan cuando llegan y responden al explorador web con un mensaje de respuesta HTTP. La repuesta contiene una línea de estado indicando si la petición ha tenido éxito o no (ej, "HTTP/1.1 200 OK" en caso de éxito). El cuerpo de una respuesta exitosa a una petición podría contener el resurso solicitado (ej, una nueva página HTML, o una imagen, etc...), que el explorador web podría presentar en pantalla.

¿Ecosistema de un servidor?

¿Diferencia entre una computadora cliente y una computadora servidor?

¿Qué es Node.js?

Node.js Es una plataforma de código abierto construida sobre el motor V8 de JavaScript de Google Chrome. Permite ejecutar JavaScript del lado del servidor, lo que significa que puedes usar JavaScript para escribir código del backend de tus aplicaciones web.

Características de Node.js:

- Event-driven y non-blocking I/O: Node.js utiliza un modelo de operaciones de entrada/salida no bloqueantes y basado en eventos, lo que lo hace eficiente para manejar una gran cantidad de conexiones simultáneas. Esto es especialmente útil para aplicaciones en tiempo real como chats, juegos en línea y aplicaciones colaborativas.
- Ecosistema de módulos: Node.js cuenta con un amplio ecosistema de módulos y paquetes disponibles a través de npm (Node Package Manager), que es el sistema de gestión de paquetes de Node.js. Esto facilita la integración de bibliotecas y herramientas de terceros en tus aplicaciones.
- Escalabilidad: Node.js es altamente escalable y puede manejar grandes volúmenes de solicitudes concurrentes de manera eficiente, lo que lo hace adecuado para aplicaciones de alta demanda.

- Desarrollo full-stack con JavaScript: Al utilizar JavaScript tanto en el frontend como en el backend, puedes tener un equipo de desarrollo que domina un solo lenguaje de programación para toda la aplicación, lo que puede simplificar el desarrollo y la colaboración entre equipos.
- Rendimiento: Node.js ofrece un rendimiento rápido gracias a su modelo de ejecución no bloqueante y al uso del motor V8 de Google Chrome.

En resumen, JavaScript del lado del servidor, impulsado por plataformas como Node.js, ofrece una forma eficiente y escalable de construir aplicaciones web completas utilizando un único lenguaje de programación en todo el stack tecnológico. Esto ha llevado a un aumento significativo en la popularidad y adopción de JavaScript en el desarrollo web.