**JavaScript Introducción**

**¿Qué es JavaScript?** JavaScript es un lenguaje de programación ampliamente utilizado en el desarrollo web. Originalmente, fue creado para agregar interactividad a las páginas web, permitiendo a los desarrolladores crear elementos dinámicos como menús desplegables, animaciones, formularios interactivos, y mucho más.

**Características del lenguaje JavaScript**

* **Lenguaje de Scripting:** JavaScript es un lenguaje interpretado, lo que significa que se ejecuta directamente del navegador sin necesidad de ser compilado.
* **Ejecutado en el navegador:** Se ejecuta del lado del cliente, es decir, en el navegador web del usuario, aunque también puede ejecutarse en el lado del servidor con entornos como, por ejemplo: Node.js.
* **Interactivo y dinámico:** Permite modificar el contenido y la estructura del documento HTML en tiempo real, respondiendo acciones del usuario, como clicks, movimientos del mouse o la entrada de datos de formulario.
* **Compatible con HTML y CSS:** Se integra fácilmente con HTML y CSS, los otros dos pilares del desarrollo web, para controlar la estructura y el estilo de una página web.
* **Amplio ecosistema:** Tiene una gran cantidad de bibliotecas y frameworks como React, Angular y Vue.js, que facilitan el desarrollo de aplicaciones web complejas.
* **Característica de alto nivel:** Incluye características de alto nivel: POO (Programación orientada a objetos, funcional (se basa en funciones), basado en eventos, manejo de promesas y asincronía).
* **Funciones en Java:** Las funciones en JavaScript también son objetos.

**¿Qué es TypeScript?** Es un lenguaje de programación que se basa en JavaScript, agregando características adicionales, principalmente el tipado estático. Fue desarrollado por Microsoft para abordar algunas de las limitaciones de JavaScript, especialmente en proyectos grandes y complejos.

**Características del lenguaje TypeScript**

* **SuperSet de JavaScript:** Es un superset de JavaScript lo que significa que todo el código JavaScript valido también es código TypeScript valido. Esto permite a desarrolladores adaptar TypeScript de manera gradual a sus proyectos JavaScript existentes.
* **Tipado Estático:** Una de las principales diferencias entre JavaScript es que TypeScript introduce un tipado estático. Esto significa que los tipos de variables, parámetros de funciones y valores de retorno se pueden definir explícitamente, lo que ayuda a detectar errores durante el desarrollo.
* **Mejor soporte para IDE’s:** Gracias al tipado estático, TypeScript ofrece una mejor integración con entornos de desarrollo (IDEs), proporcionando autocompletado, refactorización más segura y una mejor detección de errores en tiempo de desarrollo.
* **Escalabilidad:** TypeScript es particularmente útil en proyectos grandes, donde el tipado y las características avanzadas pueden ayudar a gestionar la complejidad del código y evitar errores difíciles de detectar.

**¿Qué es Hoisting?** En JS es un comportamiento en el cual las declaraciones de variables y funciones se “elevan” (hoisted) al comienzo de su contexto de ejecución, antes que se ejecute el código. Esto significa que puedes utilizar variables y funciones en tu código antes de que sean declaradas.

**Hoisting con variable Var** Cuando se declara una variable con “**var**”, su declaración se eleva al comienzo de la función o el contexto global, pero su inicialización no lo es. Esto significa que la variable se declara, pero no se le asigna un valor hasta que el código llega a la línea de asignación.

Una captura de pantalla de un celular con texto e imagen

Descripción generada automáticamente con confianza media

**Diferencias entre las variables var, let y const**

* **Var**: Alcance de función, puede ser redeclarada y reasignada, se eleva, pero se inicializa como undefined.
* **Let**: Alcance de bloque, puede ser reasignada pero no redeclarada en el mismo bloque, se eleva, pero no se inicializa.
* **Const**: Alcance de bloque, no puede ser reasignada ni redeclarada en el mismo bloque, se eleva, pero no se inicializa; las propiedades internas de objetos/arrays pueden modificarse.

**Datos Primitivos:** Son aquellos que no son objetos y no tienen métodos. Son inmutables, lo que significa que su valor no puede cambiar

* **String:** Representa una secuencia de caracteres.
* **Number:** Representa números enteros y flotantes.
* **Boolean:** Representa un valor lógico que puede ser **True** o **False.**
* **Null:** Representa la ausencia intencional de cualquier valor de objeto y debe ser asignado explícitamente.
* **Undefined:** Representa una variable que ha sido declarada pero no inicializada.
* **Symbol:** Representa un valor único y anónimo. Los símbolos son útiles para crear claves únicas en objetos.
* **BigInt:** Representa números enteros de tamaño arbitrario que pueden ser mayores que el máximo de **number.**

**Datos Compuestos:** Son los tipos de datos no primitivos, son aquellos que pueden contener múltiples valores y estructuras complejas, pueden almacenar y manipular colecciones de datos, objetos con propiedades y métodos.

* **Object:** Un objeto es una colección de pares clave-valor, donde las claves son cadenas de texto o symbols y el valor cualquier tipo de dato.
* **Array:** Una lista ordenada de valores, donde cada valor este asociado a un índice numérico. Los valores de un array puede ser cualquier tipo de dato.
* **Función:** Son objetos de primera clase, lo que significa que pueden ser asignadas a variables, pasadas como parámetro a otras funciones y devueltas por funciones y están diseñadas para una tarea especifica.
* **Date:** Se utiliza para trabajar con fechas y horas, permitiendo crear, manipular y formatear fechas.
* **RegExp:** El objeto regExp (Expresiones regulares) se utiliza para definir patrones de búsqueda dentro de cadena de texto, son útiles para tareas de validación, búsqueda y reemplazo de texto.
* **Map:** Colección de pares clave-valor donde las claves pueden ser cualquier tipo de dato y mantiene un orden de inserción de los elementos.
* **Set:** Colección de valores únicos, es decir, no permite duplicados. Los valores pueden ser cualquier tipo de dato.
* **WeakMap:** Similar a **map** pero las claves deben de ser objetos y el valor cualquier tipo de dato
* **WeakSet**: Similar a un **set** pero el valor tiene que ser un objeto.
* **Strings en JavaScript:** Inmutables: Una vez que se crea un string, no se puede cambiar su contenido, es decir, si queremos modificar un índice el string no se va a poder, pero si queremos modificar todo el string si, por que lo que hace es el puntero apuntar a la nueva dirección de memoria de ese nuevo string creado.
* **Objetos y Arrays en JavaScript:** Los objetos y arrays son mutables, es decir, la dirección de memoria es inmutable, pero el contenido si es mutable.
* **¿Qué es el DOM?** Document Object Model, es una interface de programación para los documentos HTML y XML. Representa una estructura de un documento en forma de árbol, donde cada nodo del árbol es un objeto que corresponde a una parte del documento. El DOM permite a los lenguajes de programación como JavaScript acceder y manipular el contenido, estructura y el estilo de un documento web de manera dinámica.
* **Diferencia entre funciones y métodos**
* **Función:** Una función es un bloque de código que realiza una tarea específica y puede aceptar entradas (argumentos) y devolver una salida (valor de retorno). Las funciones son componentes fundamentales de la programación y se utilizan para modularizar el código, hacerlo más legible y reutilizable. En la programación funcional, las funciones son tratadas como ciudadanos de primera clase y pueden ser pasadas como argumentos a otras funciones y devueltas como valores.
* **Método:** Un método es una función que está asociada con un objeto o una clase en el contexto de la programación orientada a objetos (POO). Los métodos son acciones o comportamientos específicos que un objeto puede realizar. Están definidos en la definición de una clase y se invocan en una instancia (objeto) de esa clase. Los métodos pueden acceder a los datos internos (propiedades) del objeto en el que se invocan.

Funciones en JavaScript

**Función Declarativa (function declaration):** Las funciones declarativas son definidas con la palabra clave “**function”** y tienen un nombre

Texto

Descripción generada automáticamente

**Características**

* **Hoisting:** Las funciones declarativas son "levantadas" (hoisting) al comienzo del contexto de ejecución, lo que significa que puedes llamarlas antes de declararlas en el código.
* **Named Function:** Tienen un nombre asociado, lo que facilita la depuración y la reutilización.

**Ventajas**

* Se pueden llamar antes de ser definidas debido al hoisting.
* Claridad en la estructura del código, especialmente en scripts más largos.

**Desventajas**

* Puede resultar en código menos modular si se abusa de ellas.

**Expresión de Función (Function Expression):** Las expresiones de función son funciones definidas como parte de una expresión. Estas pueden ser anónimas o nombradas.

**Características**

* **No hoisting:** Las expresiones de función no se levantan. Deben definirse para ser utilizadas.
* **Anónimas o nombradas:** Pueden tener un nombre, aunque es opcional, si son anónimas, son más difíciles de depurar.

**Texto

Descripción generada automáticamente con confianza media**

**Ventajas**

* Puedes definir funciones dentro de otras funciones, lo que permite un código más modular.
* Puede ser útil en casos donde necesitas una función que se utilice en un contexto limitado, como un callback.

**Desventajas**

* No se pueden llamar antes de ser definidas.
* Las funciones anónimas pueden dificultar la depuración, ya que no tienen un nombre en los mensajes de error.

**Funciones flecha (Arrow Function):** Las arrow functions son una sintaxis más concisa para definir funciones.

**Características**

* **Sintaxis concisa:** Menos código para escribir, especialmente útil para funciones pequeñas.
* **Herencia de this:** No tienen su propio this, sino que heredan el valor de this del contexto donde fueron definidas.
* **No hay “Arguments”:** No tienen acceso al objeto arguments.

**Ventajas**

* Ideal para funciones pequeñas y como callbacks en métodos como map, filter, reduce.
* Evitan problemas con el contexto de this en funciones internas.

**Desventajas**

* No se pueden usar como constructores.
* No son adecuadas para métodos de objetos si se necesita un this específico.
* Sin acceso a arguments.

**Funciones Constructoras:** Las funciones constructoras son utilizadas para crear instancias de objetos.

**Características**

* **New keyword:** Deben ser llamadas con new, lo que crea una nueva instancia de un objeto.
* **Prototipo:** Las funciones constructoras permiten agregar métodos al prototipo del objeto.

**Texto

Descripción generada automáticamente**

**Ventajas**

* Permiten la creación de múltiples instancias con propiedades y métodos compartidos.
* Útiles para patrones de diseño orientados a objetos.

**Desventajas**

* Pueden ser menos intuitivas para principiantes debido al uso de new.
* Arrow functions no pueden ser usadas como constructoras.

Texto

Descripción generada automáticamente**Ejemplos de las diferentes funciones**

**` Prototype `en Funciones Constructoras:** Cuando creas una función constructora en JavaScript, esta función tiene una propiedad llamada prototype. Este objeto prototype es compartido por todas las instancias creadas mediante esa función constructora.

Cualquier propiedad o método que añadas al prototype estará disponible para todas las instancias del objeto creado con esa función constructora.

Texto

Descripción generada automáticamente

En este ejemplo, el método saludar se define en el prototype de Persona. Esto significa que no se crea una copia del método para cada instancia, sino que todas las instancias de Persona comparten la misma función saludar a través del prototype.

**Herencia prototípica:** JavaScript no utiliza la herencia basada en clases como otros lenguajes (aunque la sintaxis de clases fue introducida en ES6 para simplificar la herencia basada en prototipos).

En su lugar, JavaScript utiliza la herencia prototípica, donde un objeto puede heredar propiedades y métodos de otro objeto.

Cada objeto tiene un enlace interno a un prototipo (accedido a través de \_\_proto\_\_), y este prototipo puede tener un prototipo propio, formando una cadena conocida como **cadena de prototipos** o **prototype chain.**

Texto

Descripción generada automáticamente

**Objeto literal en JavaScript:** Un objeto literal en JavaScript es una forma de definir objetos utilizando una sintaxis concisa y directa. Es la manera más común y sencilla de crear objetos en JavaScript, permitiendo agrupar pares de clave-valor (también conocidos como propiedades) dentro de llaves {}.

Texto

Descripción generada automáticamente

En JavaScript, arguments es un objeto similar a un array que está disponible dentro de todas las funciones (excepto en las arrow functions). Este objeto contiene todos los parámetros pasados a la función, independientemente de cuántos fueron definidos explícitamente en la declaración de la función. Es útil cuando necesitas trabajar con un número variable de argumentos o cuando no sabes cuántos argumentos serán pasados a la función.

**Características del Objeto arguments**

1. **Similar a un Array**: Aunque arguments se comporta como un array (puedes acceder a sus elementos usando índices), no es un array verdadero. No tiene métodos de array como forEach, map, o reduce, pero puedes convertirlo en un array real si es necesario.
2. **Acceso a Todos los Parámetros**: Puedes acceder a cada uno de los argumentos pasados a la función a través de índices (arguments[0], arguments[1], etc.).
3. **Longitud de arguments**: La propiedad length de arguments indica cuántos argumentos fueron pasados a la función.
4. **Disponible en Funciones Normales**: arguments está disponible en todas las funciones normales definidas con la palabra clave function. Sin embargo, **no** está disponible en **arrow functions**.

**Operador spread o operador de propagación**

Interfaz de usuario gráfica, Texto, Aplicación

Descripción generada automáticamente

CallBack’s

Los callbacks en JavaScript son funciones que se pasan como argumentos a otras funciones y se ejecutan después de que se complete una tarea o evento particular. En esencia, son una forma de manejar la asincronía en JavaScript y permiten controlar el flujo de ejecución de manera más flexible.

**Estructuras de datos:** Son importantes porque proporcionan formas eficientes y organizadas de almacenar, manipular y acceder los datos.

**Array:** Se utiliza para almacenar y organizar una colección de elementos relacionados. También podemos tener arreglo de arreglos.

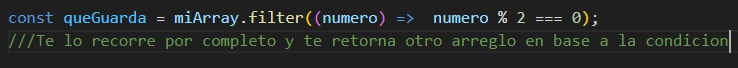
**let miArray = [1,2,3,4,5];**

**miArray = [...miArray, 1000] ;**

Spread Operator, va a agregar el dato mil a la última posición, pero la diferencia es que crea una nueva referencia con los datos.

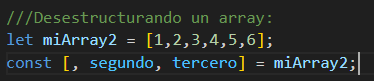
**Método filter():** Sirve para filtrar y se usan callbacks.

**Forma declarativa**



**Forma imperativa:** La forma imperativa es la que veníamos trabajando con C, con for, otro arreglo, código mas largo.

**Desestructurando un arreglo:**



Lo que hace es, en la variable segundo almacena un dos y la variable tercera almacena el numero tres.

Interfaz de usuario gráfica

Descripción generada automáticamente con confianza media

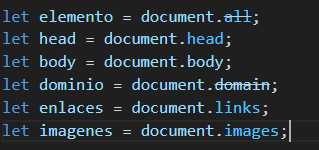
Esto lo que hace es a partir de la posición 2 en adelante va a almacenar todos los valores del array.

Conceptos de Callback

Son funciones que se pasan como argumentos a otras funciones y no bloquean el flujo normal del programa. Puede ser o no asíncrono.

JavaScript/TypeScript

**JavaScript-DOM: (Document Object Model):** Representación de la estructura de un documento HTML o XML que permite a los programadores acceder y manipular los elementos de la pagina web de manera dinámica.



**Deprécate (obsoleto):**  Significa que esa funcionalidad (Cuando nos aparecen tachados).

* No se recomienda usar.
* Probablemente será eliminada en futuras versiones.

**document.getElementByClassName()**

* Selecciona **todos los elementos** que tienen una clase específica.
* **Retorna**: Una **colección de elementos** (HTMLCollection) que coincide con el nombre de la clase. No es un array, pero se puede iterar.

**document.getElementById():**

* Selecciona **un único elemento** basado en su atributo id.
* **Retorna**: El primer elemento que coincida con el id, o null si no existe.

**document.getElementsByTagName():**

* Selecciona todos los elementos que tienen una etiqueta HTML específica (como div, p, h1, etc.).
* **Retorna**: Una **colección de elementos** (HTMLCollection) con el nombre de la etiqueta.

**document.querySelector():**

* Selecciona el primer elemento que coincida con un selector CSS (clase, id, etiqueta, combinaciones de selectores, etc.).
* **Retorna**: El primer elemento que coincida con el selector, o null si no se encuentra.

**document.querySelectorAll():**

* Selecciona **todos los elementos** que coincidan con un **selector CSS**.
* **Retorna**: Una **NodeList**, que es similar a un array y puede ser iterada.

Texto, Aplicación

Descripción generada automáticamente

**Traversing del DOM:** Se refiere a la manipulación y navegación a través de la estructura del DOM.

Promise-JS

Las promesas son asincrónicas y las promesas son no bloqueantes.

Una **promesa** es un objeto que representa la eventual finalización (o falla) de una operación asíncrona y su valor resultante. Permite manejar el resultado de operaciones que toman tiempo (como peticiones a servidores o temporizadores), sin bloquear el hilo principal de ejecución.

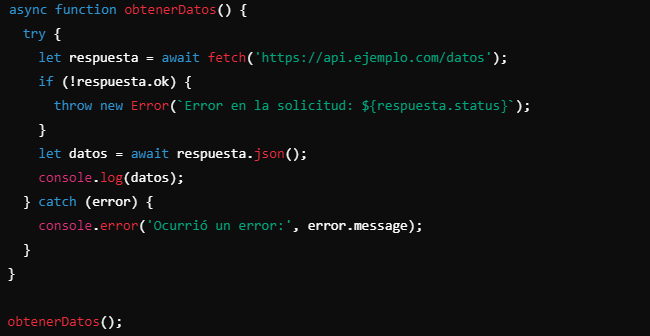
**Estados de una promise:**

* **Pendiente:** La operación aun no se ha completado y la promesa esta esperando que se resuelva o se rechace.
* **Cumplida:** La operación se completó con éxito y la promesa se resolvió con un valor.
* **Rechazada:** La operación fallo y la promesa se rechazo con un motivo o razón del fallo.

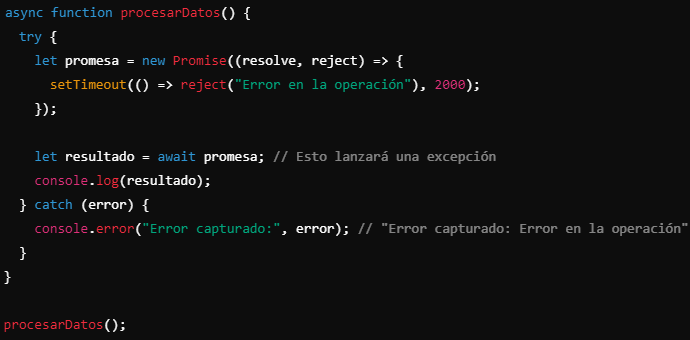
Async: La palabra clave **async** se utiliza para declarar una función asíncrona. Una función marcada con async siempre devuelve una promesa. Esto facilita trabajar con promesas de manera más sencilla y legible, sin necesidad de utilizar explícitamente los métodos .then() o .catch().

Await: La palabra clave **await** solo se puede utilizar dentro de una función asíncrona y se usa para esperar el resultado de una promesa. Mientras el código "espera", la ejecución del programa no se bloquea, lo que permite manejar el código de forma secuencial sin perder el carácter asíncrono.

**Ejemplo de código que no tiene fallas**



**Ejemplo de código que va a entrar en el error**

****

¿Qué es una petición HTTP?

**Una request es:** Un mensaje que un cliente, como un navegador web, envía a un servidor para solicitar recursos o realizar acciones sobre esos recursos

**HTTP es un protocolo de comunicación** entre clientes y servidores en la web.

**Estructura de una petición HTTP**

* **Método:** Operación que el cliente quiere realizar (GET, POST, ETC).
* **URL:** El recurso que se solicita en el servidor.
* **Cabeceras (header):** Información adicional sobre la petición (tipo de contenido, de autorización)
* **Cuerpo (body):** Cuerpo de la petición. Datos que se envían al servidor (POST o PUT).

**Postman:** Le pasamos la URL y nos testea a ver si funciona la petición.

**Para que sirve JSON-Server:** **JSON Server** es una herramienta poderosa para **probar y simular una API** REST de forma rápida y sin esfuerzo. Es ideal para proyectos en fase de desarrollo o prototipado, y permite a los desarrolladores frontend trabajar de manera independiente del backend durante el ciclo de desarrollo.

**Métodos de petición HTTP:**

**GET:** Solicita un recurso del servidor. No modifica nada en el servidor, solo obtiene datos.

* **Ejemplo:** Obtener una lista de usuarios de un servidor.

**POST:** Envia datos al servidor para crear un nuevo recurso.

* **Ejemplo:** Enviar un formulario para crear un nuevo usuario.

**PUT:** Envía datos al servidor para reemplazar completamente un recurso existente.

* **Ejemplo:** Actualizar todos los datos de un usuario existente.

**PATCH:** Envía datos para modificar parcialmente un recurso existente.

* **Ejemplo:** Actualizar solo el nombre de un usuario.

**DELETE:** Solicita la eliminación de un recurso en el servidor.

* **Ejemplo:** Eliminar un usuario.

**HEAD:** Similar a GET, pero solo recupera los encabezados de la respuesta, sin el cuerpo.

* **Ejemplo:** Verificar si un recurso existe sin descargar su contenido.

**OPTIONS:** Solicita las opciones de comunicación permitidas para un recurso en el servidor.

* **Ejemplo:** Verificar qué métodos HTTP están permitidos en una URL específica.

**CONNECT:** Se utiliza para establecer un túnel hacia un servidor mediante el protocolo HTTP, comúnmente utilizado para proxies.

**TRACE:** Realiza una solicitud de bucle de retorno para verificar la ruta que sigue una solicitud a través de servidores intermedios.

**Fetch():** Permite consumir un recurso. **fetch()** es una función nativa de JavaScript que devuelve una **promesa**, por lo que puedes manejar su resultado utilizando tanto la sintaxis de **promesas (.then()/.catch())** como la sintaxis de **async/await**.