ACTIVIDAD 3.1

JavaScript es uno de los lenguajes mas ampliamente usado en el ámbito web, eso se debe a que es un lenguaje interpretado, que reacciona a eventos y que además esta preparado para ser utilizado por el paradigma de POO, este ultimo le da la capacidad de crear código que posteriormente puede ser reutilizable en otros proyectos. JavaScript es en esencia lo que transforma las páginas estáticas en páginas completamente dinámicas e interactivas, permitiéndole actualizarse en tiempo real sin tener que recargar toda la web.

ACTIVIDAD 3.2

Arrow Functions.

¿Que son?

Las funciones flecha, introducidas en ES6 (ECMAScript 2015), ofrecen una sintaxis más corta y concisa para escribir expresiones de funciones en JavaScript. Una de sus características más importantes es que no tienen su propio this, arguments, super, o new.target.

(param1, param2) => { / sentencias */ } ó () => { /* sentencias */ }*

Ventajas

- **Sintaxis más limpia y corta**: Hacen el código más legible y fácil de escribir.
- **this léxico**: No crean su propio contexto this. En su lugar, heredan el this del ámbito en el que fueron creadas. Esto resuelve muchos de los problemas y la confusión asociados con la palabra clave this en JavaScript tradicional, especialmente en callbacks y manejo de eventos.

¿Cuándo usarlas?

Son ideales para funciones que no son métodos de un objeto, y especialmente útiles en callbacks para métodos de array como .map(), .filter() y .reduce(), donde la brevedad y el this léxico son muy beneficiosos.

Arrays y Métodos (.map(), .filter(), .reduce()).

¿Qué son?

Los arrays son estructuras de datos tipo lista que permiten almacenar múltiples valores en una sola variable. JavaScript proporciona métodos de orden superior muy potentes para iterar y transformar arrays de manera declarativa.

.map()

- **¿Qué es?:** Crea un **nuevo** array con los resultados de llamar a una función para cada elemento del array original.
- **Estructura:** array.map(callback(elemento, indice, array))
- **Ventajas:** Permite transformar cada elemento de un array de manera inmutable (sin modificar el original).
- **¿Cuándo usarlo?:** Cuando necesitas transformar cada elemento de un array en algo nuevo.

```
const numeros = [1, 2, 3, 4];

const dobles = numeros.map(num => num * 2);


console.log(dobles); // Salida: [2, 4, 6, 8]

console.log(numeros); // Salida: [1, 2, 3, 4] (el original
```

.filter()

- **¿Qué es?:** Crea un **nuevo** array con todos los elementos que pasen una prueba (una función que devuelve true).
- **Estructura:** array.filter(callback(elemento, indice, array))
- **Ventajas:** Permite seleccionar elementos de un array basados en una condición de forma inmutable.
- **¿Cuándo usarlo?:** Cuando necesitas obtener un subconjunto de un array que cumpla con ciertos criterios.

```
const edades = [15, 22, 18, 30, 17];

const mayoresDeEdad = edades.filter(edad => edad >= 18);


console.log(mayoresDeEdad); // Salida: [22, 18, 30]
```

.reduce()

- ¿Qué es?: Aplica una función a un acumulador y a cada elemento de un array (de izquierda a derecha) para reducirlo a un único valor.
- Estructura: array.reduce(callback(acumulador, elemento, indice, array), valorInicial)

- Ventajas: Es extremadamente versátil; puede usarse para sumar, aplanar arrays, agrupar objetos y mucho más.
- ¿Cuándo usarlo?: Cuando necesitas "resumir" un array en un solo valor (un número, un objeto, otro array, etc.).

```
const numeros = [1, 2, 3, 4, 5];

const sumaTotal = numeros.reduce((acumulador, numero) => acumulador + numero, 0);


console.log(sumaTotal); // Salida: 15
```

Operadores Rest y Spread.

Operador Rest (...)

- **¿Qué es?**: Agrupa el "resto" de los elementos en una sola entidad. En el contexto de los parámetros de una función, agrupa los argumentos restantes en un array.
- **Estructura (en parámetros de función)**: function miFuncion(param1, param2, ...restoDeArgumentos) { ... }
- **Ventajas**: Permite crear funciones que aceptan un número variable de argumentos de una manera más limpia y explícita que el antiguo objeto arguments.
- **¿Cuándo usarlo?**: Cuando quieres que una función acepte un número indefinido de argumentos y manipularlos fácilmente como un array.

```
function sumar(...numeros) {
  return numeros.reduce((total, num) => total + num, 0);
}


console.log(sumar(1, 2, 3));       // Salida: 6
console.log(sumar(10, 20, 30, 40)); // Salida: 100
```

Operador Spread (...)

- **¿Qué es?:** "Expande" los elementos de un iterable (como un array o un string) en lugares donde se esperan cero o más argumentos (para llamadas a funciones) o elementos (para literales de array).
- **Estructura:**
    - En arrays: const nuevoArray = [...arrayOriginal, 'nuevo_elemento'];
    - En llamadas a funciones: miFuncion(...miArray);
- **Ventajas:** Proporciona una forma muy concisa y legible para copiar arrays, concatenar arrays y pasar elementos de un array como argumentos a una función.
- **¿Cuándo usarlo?:** Para crear copias superficiales de arrays u objetos, para fusionar arrays o para pasar los elementos de un array como argumentos individuales a una función.

```
// Copiar y concatenar arrays
const arr1 = [1, 2, 3];
const arr2 = [4, 5, 6];
const arrCopia = [...arr1];
const arrFusionado = [...arr1, ...arr2];


console.log(arrCopia);      // Salida: [1, 2, 3]
console.log(arrFusionado);  // Salida: [1, 2, 3, 4, 5, 6]


// En llamadas a funciones
const numeros = [5, 1, 7, 3];
console.log(Math.max(...numeros)); // Salida: 7 (equivalente a Math.max(5, 1, 7, 3))
```

Módulos.

¿Qué son?

Los módulos permiten dividir el código en archivos separados y reutilizables. Cada módulo tiene su propio ámbito; las variables, funciones y clases declaradas en un módulo no son visibles fuera de él, a menos que se exporten explícitamente.

Estructura

Se utilizan las palabras clave export para hacer que el código esté disponible para otros módulos, e import para usar el código de otro módulo.

- Exportar:
    - Exportaciones nombradas (múltiples por módulo):

```
// archivo: utils.js
export const PI = 3.1416;
export function sumar(a, b) {
  return a + b;
}
```

    - Exportación por defecto (una por módulo):

```
// archivo: Persona.js
export default class Persona {
  // ...
}
```

- Importar:
    - Importaciones nombradas:

```
// archivo: main.js
import { PI, sumar } from './utils.js';
```

- **Importación por defecto:**

```
// archivo: main.js

import Persona from './Persona.js';
```

Ventajas

- **Organización:** Mantiene el código estructurado y más fácil de navegar.
- **Reutilización:** Permite reutilizar funciones, clases y variables en diferentes partes de la aplicación.
- **Mantenibilidad:** Facilita la depuración y el mantenimiento al aislar la funcionalidad.
- **Evita la contaminación del ámbito global:** Previene conflictos de nombres entre diferentes partes del código.

¿Cuándo usarlos?

Siempre, en cualquier aplicación que no sea trivial. Son la base de la arquitectura de las aplicaciones web modernas.

Sincrónico y Asincrónico

¿Qué es?

- **Sincrónico**: El código se ejecuta línea por línea, en orden secuencial. Una tarea debe completarse antes de que comience la siguiente. Si una tarea tarda mucho, bloquea todo el hilo de ejecución.
- **Asincrónico**: Permite que las tareas de larga duración (como peticiones a una API, lectura de archivos, etc.) se ejecuten en segundo plano. El resto del código no se bloquea y puede continuar ejecutándose. Cuando la tarea asincrónica termina, el resultado se maneja.

En JavaScript, la asincronía se maneja principalmente con Promesas y la sintaxis async/await.

Promesas (Promises)

- **¿Qué son?:** Un objeto que representa la eventual finalización (o fallo) de una operación asíncrona y su valor resultante. Una promesa puede estar en uno de tres estados:
  - **Pendiente (pending):** Estado inicial, ni cumplida ni rechazada.
  - **Cumplida (fulfilled):** La operación se completó con éxito.

    - **Rechazada (rejected)**: La operación falló.

- **Estructura**: Se consumen usando los métodos .then() (para el éxito), .catch() (para el error) y .finally() (se ejecuta siempre).
- **Ventajas**: Evitan el "Callback Hell" (anidamiento excesivo de callbacks), permitiendo un código más legible y manejable para operaciones asíncronas.

Fetch

fetch es una API del navegador moderna para realizar peticiones de red (por ejemplo, a una API REST). Está basada en promesas.

Async/Await

- **¿Qué es?**: Es "azúcar sintáctico" sobre las promesas. Permite escribir código asíncrono que se ve y se comporta más como código síncrono, haciéndolo más fácil de leer y entender.
- **Estructura**:
    - **async**: Se coloca antes de una función para indicar que devolverá una promesa.
    - **await**: Se usa dentro de una función async para pausar la ejecución y esperar a que una promesa se resuelva.
- **Ventajas**: Mejora drásticamente la legibilidad y la estructura del código asíncrono, especialmente cuando hay múltiples pasos dependientes.

React Hooks

Los Hooks son funciones que te permiten "engancharte" a las características de estado y ciclo de vida de React desde componentes de función.

useEffect

- **¿Qué es?**: Permite ejecutar efectos secundarios en componentes de función. Estos efectos pueden ser peticiones de datos, suscripciones, o manipulaciones manuales del DOM.
- **Estructura**: useEffect(setup, dependencias?)
    - **setup**: La función que contiene la lógica del efecto.
    - **dependencias**: Un array opcional. El efecto se volverá a ejecutar solo si los valores en este array han cambiado entre renderizados.

- **Ventajas:** Centraliza la lógica de los efectos secundarios que antes se distribuía en métodos del ciclo de vida de las clases (componentDidMount, componentDidUpdate, componentWillUnmount).
- **¿Cuándo usarlo**?: Para buscar datos, establecer suscripciones, y cualquier otra cosa que necesite interactuar con sistemas externos.

useRef

- **¿Qué es?:** Devuelve un objeto de referencia mutable (.current) que persiste durante todo el ciclo de vida del componente.
- **Estructura:** const miRef = useRef(valorInicial);
- **Ventajas**:
    1. Permite acceder directamente a un nodo del DOM.
    2. Permite mantener un valor mutable que no provoca un nuevo renderizado cuando cambia.
- **¿Cuándo usarlo**?: Para gestionar el foco, la reproducción de medios o para integrar con bibliotecas de terceros que manipulan el DOM. También para almacenar valores que no deben desencadenar un renderizado.

useMemo

- **¿Qué es?:** Memoriza (cachea) el resultado de una función costosa**.**
- **Estructura:** const valorMemorizado = useMemo(() => calcularValor(a, b), [a, b]);
- **Ventajas**: Evita recálculos costosos en cada renderizado si las dependencias no han cambiado, mejorando el rendimiento.
- **¿Cuándo usarlo?:** Cuando tienes cálculos computacionalmente caros que no quieres volver a ejecutar en cada renderizado.

useCallback

- **¿Qué es**?: Memoriza (cachea) una función**.**
- **Estructura**: const funcionMemorizada = useCallback(() => { hacerAlgo(a, b); }, [a, b]);
- **Ventajas:** Evita que se cree una nueva instancia de una función en cada renderizado. Es útil cuando se pasa una función como prop a un componente hijo que está optimizado (por ejemplo, con React.memo), previniendo renderizados innecesarios del hijo.

- **¿Cuándo usarlo?:** Para optimizar el rendimiento al pasar callbacks a componentes hijos memorizados.

useContext

- **¿Qué es?:** Permite leer y suscribirse a un contexto de React sin introducir anidamiento ("prop drilling").
- **Estructura:** const valor = useContext(MiContexto);
- **Ventajas**: Facilita el acceso a datos globales (como un tema, información de usuario, etc.) en cualquier nivel del árbol de componentes sin tener que pasar props manualmente a través de todos los niveles.
- **¿Cuándo usarlo?:** Para gestionar el estado global de la aplicación.

useReducer

- **¿Qué es?:** Es una alternativa a useState para gestionar lógicas de estado más complejas.
- **Estructura:** const [estado, dispatch] = useReducer(reducer, estadoInicial);
    - **reducer:** Una función (estado, accion) => nuevoEstado que especifica cómo cambia el estado.
    - **dispatch**: Una función que se llama para "despachar" una acción y desencadenar una actualización del estado.
- **Ventajas**: Centraliza la lógica de actualización del estado, haciéndola más predecible y fácil de probar. Es ideal cuando el próximo estado depende del anterior o cuando hay múltiples sub-valores en el estado.
- **¿Cuándo usarlo?:** Para manejar estados complejos con transiciones bien definidas, como en un formulario complejo, un carrito de compras o un juego.