Moviles 3

**Variables**

* var: Puede ser declarada pero no tiene que ser inicializada, son variables globales que no respetan el scope, hace que se mezclen los valores y por eso no se usa mucho.
* const: se usa para seleccionar elementos de la interfaz
* let: Funciona igual que var manteniéndose dentro del scope

**Template Strings**

Es una nueva forma de concatenar información utilizando ` `

const nombre= 'Victor';

const trabajo= 'Curro';

//Captulo elemento HTML

const contenedorApp = document.querySelector('#app')

let html = `

            <ul>

            <li> Nombre: ${nombre}</li>

            <li> Trabajo: ${trabajo}</li>

            </ul>

            `;

//Asigno valor al elemento capturado

contenedorApp.innerHTML=html;

**Funciones**

Crear código reutilizable y mas inteligente usando parámetros

* function declaration: En este el código primero esta leyendo las funciones que hay en el documento y no importa si se llama la función antes de ser escrita va a funcionar

function saludar (nombre){

console.log (`Bienvenido ${nombre}`);

}

saludar('pepe')

* function expresión: Primero la función debe estar definida entes de ser ejecutada

const cliente =  function(nombre){

    console.log(`Hola ${nombre}`)

}

cliente('pedro')

* arrowFunctions: simplifica el código y en vez de escribir function se pasa al lado del paréntesis el símbolo => si tiene solo un parámetro puede quitarle los paréntesis.

const cliente = nombre => {

    console.log(`Hola ${nombre}`)

}

cliente('pedro')

**Objetos**

* Objeto literal: nos permite crear un objeto un poco mas estático donde nosotros damos los valores

const cliente = {

    nombre: 'Juan',

    edad: 30,

    profesion: 'dev'

}

* Object constructor: Es la forma anterior a que JavaScript permitiera crear clases para crear múltiples objetos

  function Cliente(nombre, edad, profesion){

    this.nombre = nombre;

    this.edad = edad;

    this.profesion = profesion;

    }

    const cliente1 = new Cliente ('pepe', 30, 'dev')

**Prototypes**

Se usa para que los métodos que creas solo puedan ser usados en un tipo de objeto determinado, cumpliendo con su fin y evitando que se mezcle o cambie a futuro

    function Cliente(nombre, edad, profesion){

    this.nombre = nombre;

    this.edad = edad;

    this.profesion = profesion;

    }

    Cliente.prototype.mostrarInfo = function (){

        return `el nombre es ${this.nombre}

        la edad que tiene ${this.edad}

        `

    }

**Destructuring de objetos** es como extraes valores de un objeto

**Objetc literal enhancement** Es la forma de unir diferentes variables en un objeto

const nombreObjt = {variable1,variable2, ...}

**Arreglos map** es la forma de recorrer un arreglo para mostrar lo que contiene

* Forma1

  //Acceso al document HTML y captura el ID

    const appContenedor = document.querySelector('#app');

    //Recorre el arreglo con forEach

    let html='';

    arreglo.forEach(element => {

        html +=`<li>${element}</li>`

    });

    //Envia al contenedor

    appContenedor.innerHTML=html;

* Forma 2 convierte el arreglo en otro arreglo con .map

    arreglo.map(element => {

        return 'el elemento es '+ element

    })

**Object.keys** se utiliza para saber cuales son las variables keys de un objeto

  console.log(Object.keys(objeto))

**spreadoperator** Dentro de sus funciones esta la posibilidad de combinar 2 arreglos diferentes

* Crea copia de los arreglos para no modificar los arreglos originales

    let arreglo1 = ['elemento1','elemento2']

    let arreglo2 = ['elemento3','elemento4']

    let combinacion =[...arreglo1,...arreglo2]

**Métodos en arreglos**

* En un arreglo de objetos usamos
* Arreglo.filter para hacer condiciones de filtrado
* Arreglo.find para encontrar valores específicos
* Arreglo.reduce para hacer operaciones entre los diferentes objetos

**Promises**

**Estudiar**

**Promises con Ajax**

**Consumo de API´s**

**Programación orientada a objetos**

**Clases** Primero se crea con la palabra reservada class + el nombre

* Para pasarle los argumentos al objeto se utiliza un constructor, esto es lo que primero se ejecuta una vez se cree una instancia de una tarea

class Tarea {

  constructor(nombre, prioridad){

    this.nombre = nombre;

    this. prioridad = prioridad

  }

}

let tarea1 = new Tarea('Aprender JavaScript', 'Alta')

console.log(tarea1);

* **Método** es una función que al pasarla a una clase se define como método

class Tarea {

  constructor(nombre, prioridad){

    this.nombre = nombre;

    this. prioridad = prioridad

  }

  metodomostrar(){

    return(`${this.nombre} tiene una prioridad ${this.prioridad}`);

  }

}

let tarea1 = new Tarea('Aprender JavaScript', 'Alta')

console.log(tarea1.metodomostrar);

* **Herencia** se heredan las clases y sus métodos, la ventaja es que puedes heredar de una clase padre y a partir de esto ajustar a la necesidad que tenemos

//Padre

class Tarea {

  constructor(nombre, prioridad){

    this.nombre = nombre;

    this. prioridad = prioridad

  }

  metodomostrar(){

    console.log (`${this.nombre} tiene una prioridad ${this.prioridad}`);

  }

}

//Hijo agregando otros atributos y reescribiendo metodo

class TareaHija extends Tarea{

    sonstructor(nombre, prioridad, cantidad){

      super(nombre, prioridad);

      this.cantidad = cantidad;

    }

    metodomostrar(){

      super.metodomostrar();

      console.log(`y la cantidad es ${this.cantidad}`)

    }

}

**Módulos ES6**

**Exportar e importar funciones**

* Exportar funciones o bloques individuales

export const nombreTarea = 'Funcion'

import {nombreTarea} from './App.jsx'

* Exportar por default en el cual solo puede exportar 1 por archivo

const nombreTarea = 'Funcion'

export default nombreTarea

import nombreTarea from './App.jsx'

* en caso de requerir que se exporten varios con este método, puedes crear un objeto

const nombreTarea = 'Funcion'

export default {

  tareauno : tareauno

  tareados : tareados

}

**REACT NATIVE**

Su sintaxis y componentes estan basados en react.

JSX lo que hace es producir elementos de de JS y los renderiza en la pagina WEB haciendo que el JS y el HTML se comuniquen de manera mas fluida

* **Virtual DOM** Comprueba constantemente si algún componente de React tiene cambio para así renderizar este componente y todos sus hijos, nuca los padres
* Cuanto mas bajo es el nivel de manejo de datos es mas rápido refrescar por que tiene que recargar menos componentes
* **Estructura de ficheros en React**
* Node modules: es donde se van a instalar todas nuestras dependencias de manera temporal
* Git.ignore: indica los archivos que ingnoraria al hacer un commit en algún repositorio git
* Public: Aquí se ubican los archivos estáticos de nuestra aplicación
* SRC: Es el corazón de la aplicación, contiene un index que ejecuta como primer componente APP que es la ruta primaria de la aplicación
* Package.json: Contiene la versión de la app, si es privada o no y las dependencias con las que esta trabajando la app.
  + Scripts – es un comando que ejecuta de manera sencilla otro comando mas complejo

**Componentes:** Todo componente debe ir en SRC

* Por buena practica se debe hacer una nueva carpeta que se llame components
* Todos los componentes inician con nombre en mayúsculas
* En primera parte tiene las importaciones, componentes, css, imágenes
* Tiene un return obligatorio
* Un componente se tiene que exportar para poderlo reutilizar
* Props o las propiedades de React es la forma que tenemos de pasar información de un componente a otro
* Una propiedad puede ser heredada a componentes hijos e hijos de los hijos con nuevas Props

**React Hooks**

Permiten que los componentes funcionales puedan tener ciclo de vida sin tener que escribir clases, tiene 100% de retrocompatibilidad sin que se borre nada de la api

* Importante usar la versión 16.8.6 React

**UseState**



Nos entrega un estado, una función para actualizar dicho estado y un campo para indicar el valor inicial que el estado va a tener.

* Tipos de componentes

Estructurar aplicaciones y buenas practicas

* State
* Props
* Hooks
* Redux
* PropTypes
* Leer y validar formularios
* Animaciones
* Consumir API’s
* LocalStorage
* React Router
* Server Side Rendering

Webpack y Babel

**12/08/2023**

**Introducción a React JS**

**V – C- M**

**Vista –** React

**Controlador**

**Modelo** – Conexión con la base de datos

1. Instalación

* Node js
* NPM
* CLI (Para colocar los comandos específicos de cada aplicación)

1. Inicio

* Carpeta en la que vamos a trabajar y en la ruta escribimos CMD para abrir consola
* Para validar la versión de node node -v
* Para validar la versión de NPM npm -v
* Crear el proyecto v anterior npx create-react-app NombreProyecto
* Crear el proyecto con vite npm create vite@latest
* Seguir instrucciones del consola

Para que todo se maneje por componentes se usa en React JSX que es la tecnología que permite ejecutar en JS código de HTML

* Sin JSX

function App() {

  const [count, setCount] = useState(0)

  //Para insertar un elemento sin JSX se usa la funcion React.createElement

  return ( React.createElement("h1",{},"Hola sin JSX")

  )

}

* Con JSX que nos ayuda a simplificar la sintaxis

function App() {

  const [count, setCount] = useState(0)

  //Para insertar con JSX lo mejor es insertar el fractment <>  </>

  return (

    <>

     <h1>Contenido HTML y JS</h1>

    </>

  )

}

**Componentes**

* Siempre inicia con Mayúscula
* Debe contener un Return

1. **Crear un componente**

* para crear el componente auto con visual studio
* Creamos un archivo nuevo .js
* import React from “react”;
* Creo la función
* Exporto la función

import React from "react";

const Contenedor = () => {

    return <h1>Otro componente</h1>

}

export default Contenedor

* **Para importarlo entonces**