ECMAScript ES6. Versión estandarizada de JavaScript.

1.- Compara el alcance de las palabras clave [var] y [let]:

function checkScope() {
  let i = 'function scope';
  if (true) {
  let i = 'block scope';
  console.log('Block scope i is: ', i);
  }
  console.log('Function scope i is: ', i);
  return i;
}
	El ejercicio busca ilustrar la diferencia entre el ámbito (scope) de las variables declaradas con var y let. Tenemos dos variables let con el mismo nombre que actúan en diferentes ámbitos del código.

2.- Muta un arreglo declarado con [const]:

const s = [5, 7, 2];
function editInPlace() {
  s[0] = 2;
  s[1] = 5;
  s[2] = 7;
}
editInPlace();
	El ejercicio nos pide mutar un arreglo declarado con const. Usar la declaración const previene la reasignación del identificador de una variable. Para mutar un arreglo declarado con const debemos acceder a él a través del índice [].

3.- Prevenir la mutación de un objeto:

function freezeObj() {
  const MATH_CONSTANTS = {
  PI: 3.14
  };
  Object.freeze(MATH_CONSTANTS);
  try {
  MATH_CONSTANTS.PI = 99;
  } catch(ex) {
  console.log(ex);
  }
  return MATH_CONSTANTS.PI;
}
const PI = freezeObj();
	El ejercicio pide prevenir el cambio de constantes matemáticas mediante Object.freeze(), una función que previene el cambio de información en un objeto. Arroja un error si intenta modificarse.

4.- Usa funciones flecha para escribir funciones anónimas de manera breve:

//var magic = function() {
  //return new Date();
//};
 	
const magic = () => new Date();
	El ejercicio pide reescribir una función utilizando la sintaxis de flecha. Las funciones flecha se utilizan cuando tienes funciones pequeñas que no vas a reutilizar, por lo que no necesitas nombrarlas (función anónima). Cuando la función solo tiene un valor de retorno, se puede omitir el return y los corchetes que rodean al código.

5.- Escribe una función flecha con parámetros:

//var myConcat = function(arr1, arr2) {
  //return arr1.concat(arr2);
//};
 	
const myConcat = (arr1, arr2) =>
arr1.concat(arr2);
 	
console.log(myConcat([1, 2], [3, 4, 5]));
	El ejercicio pide reescribir una función con parámetros en sintaxis flecha. Cuando solo se le da un parámetro a la función los paréntesis pueden ser omitidos. Es posible pasar más de un argumento a la función.

6.- Establece parámetros por defecto para tus funciones:

const increment = (number, value = 1) => number + value;	El ejercicio pide modificar una función agregando parámetros por defecto para que sume 1 a number si value no se especifica.

7.- Utiliza el parámetro rest con parámetros de función:

const sum = (...args) => {
  let total = 0;
  for (let i = 0; i < args.length; i++) {
  total += args[i];
  }
  return total;
}
	El ejercicio pide que modifiques una función para que sea capaz de recibir cualquier número de argumentos y devolver su suma. Con rest (...) podemos flexibilizar las funciones para que acepten un número variable de parámetros.

8.- Utiliza el operador de propagación para evaluar los arreglos en el lugar:

El operador de propagación nos permite expandir arreglos y otras expresiones en lugares donde se esperan múltiples parámetros o elementos.

const arr = [6, 89, 3, 45]; const maximus = Math.max(...arr);	//Encuentra el número máximo en un arreglo. [...arr] devuelve un arreglo desempaquetado.

const arr1 = ['JAN', 'FEB', 'MAR', 'APR', 'MAY'];
 	
let arr2;
arr2 = [...arr1];
 	
console.log(arr2);
	El ejercicio pide copiar el contenido de un arreglo a otro usando el operador de propagación.

9.- Sintaxis de desestructuración para extraer valores de objetos:

Con la desestructuración podemos asignar valores directamente desde objetos. Los dos fragmentos de código siguientes son equivalentes:

const user = { name: 'John Doe', age: 34 };
 	
const name = user.name;
const age = user.age;
	const user = { name: 'John Doe', age: 34 };
	
	const { name, age } = user;

const HIGH_TEMPERATURES = {
  yesterday: 75,
  today: 77,
  tomorrow: 80
};
 	
const {today, tomorrow} =
HIGH_TEMPERATURES;
	El ejercicio pide sustituir asignaciones con la sintaxis de desestructuración equivalente.

10.- Sintaxis de desestructuración para asignar variable desde objetos:

La desestructuración permite asignar un nuevo nombre de variable al extraer valores. Puedes hacer esto al poner el nuevo nombre después de dos puntos al asignar el valor.

const HIGH_TEMPERATURES = {
  yesterday: 75,
  today: 77,
  tomorrow: 80
};
 	
const {today : highToday, tomorrow :
highTomorrow} = HIGH_TEMPERATURES;
	El ejercicio pide reemplazar dos asignaciones con sintaxis de desestructuración y cambiar los valores del objeto.

11.- Sintaxis de desestructuración para asignar variables desde objetos anidados:

Puedes usar los mismos principios del punto 9 y 10 para desestructurar los valores desde objetos anidados.

const LOCAL_FORECAST = {
  yesterday: { low: 61, high: 75 },
  today: { low: 64, high: 77 },
  tomorrow: { low: 68, high: 80 }
};
 	
const {today: {low : lowToday, high : highToday}} = LOCAL_FORECAST;
	El ejercicio pide reemplazar dos asignaciones con sintaxis de desestructuración y cambiar los valores del objeto anidado.

12.- Sintaxis de desestructuración para asignar variables desde arreglos:

Una diferencia clave entre el operador de propagación y la desestructuración es que el primero desempaca el contenido del arreglo en una lista separada por comas [,]. En consecuencia no puedes decidir qué elementos deseas asignar como variables. Mientras que la desestructuración si nos lo permite.

const [a, b] = [1, 2, 3, 4, 5, 6]; //Asigna el primer valor a [a] y el segundo a [b]
console.log(a, b); //Devuelve 1 y 2

const [a, b,,, c] = [1, 2, 3, 4, 5, 6];//Podemos acceder al valor de cualquier índice con comas.
console.log(a, b, c); //Devuelve 1, 2 y 5

let a = 8, b = 6; [a,b] = [b,a];	El ejercicio pide usar la sintaxis de desestructuración para intercambiar los valores de a y b.

13.- Desestructuración vía elementos rest:

En algunas situaciones que implican desestructuración de arreglos, puede que queramos recolectar el resto de elementos en un arreglo por separado.

[const [a, b, ...arr] = [1, 2, 3, 4, 5, 7];]

function removeFirstTwo(list) {
  const [a, b, ...shorterList] = list;
  return shorterList;
}
 	
const source = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10];
const sourceWithoutFirstTwo = removeFirstTwo(source);
	El ejercicio pide utilizar la asignación de desestructuración con la sintaxis rest para emular el comportamiento de Array.prototype.slice().removeFirstTwo() debe devolver un sub arreglo del arreglo original list con los dos primeros elementos borrados.

14.- Utilizar sintaxis de desestructuración para pasar un objeto como parámetro de una función:

const stats = {
  max: 56.78,
  standard_deviation: 4.34,
  median: 34.54,
  mode: 23.87,
  min: -0.75,
  average: 35.85
};
 	
//const half = (stats) => (stats.max + stats.min) / 2.0;
 	
const half = ({max, min}) => (max + min)/2;
	El ejercicio pide usar la sintaxis de desestructuración dentro del argumento de la función half, para enviar solo mín y max a la función.
	
	*No entiendo cómo sabe que tiene que buscar los argumentos en el objeto stats.

15.- Crea cadenas usando plantillas literales:

Las plantillas literales son un tipo especial de cadena que facilita la creación de cadenas complejas. Permiten crear cadenas multilínea y usar características de interpolación, para crearlas.

const person = { name: "Zodiac Hasbro", age: 56 }; const greeting = `Hello, my name is ${person.name}! I am ${person.age} years old.`; console.log(greeting);	1-Se utilizan backticks [`] en lugar de comillas. 2-Es multilínea, guarda la inserción [\n] dentro de las cadenas. 3.- La sintaxis ${variable} es un marcador de posición. No requiere el uso de concatenadores [+]. 4.-Permite la inclusión de expresiones en la cadena literal. ${a + b}.

const result = {
  success: ["max-length", "no-amd", "prefer-arrow-functions"],
  failure: ["no-var", "var-on-top", "linebreak"],
  skipped: ["no-extra-semi", "no-dup-keys"]
};
function makeList(arr) {
 	
  const failureItems = [];
 	
  for(let i = 0; i < arr.length; i++){
  failureItems[i] = `<li class="text-warning">${arr[i]}</li>`;
	El ejercicio pide usar la sintaxis de plantilla literal con backticks para crear un arreglo de cadenas de elementos lista [li]. El texto de cada elemento debe ser uno de los elementos del arreglo de la propiedad failure en el objeto result y tener un atributo class con el valor [text-warning]. La función makeList debe devolver el arreglo de cadenas.
	
	Utiliza cualquier tipo de bucle.

}
 	
return failureItems;
}
 	
const failuresList = makeList(result.failure);	

16.- Escribe declaraciones concisas de objetos literales usando la abreviatura de la propiedad del objeto.

ES6 permite eliminar la redundancia de tener que escribir x: x.

const getMousePosition = (x, y) => ({ x: x, y: y });	const getMousePosition = (x, y) => ({ x, y });

const createPerson = (name, age, gender) => ({name, age, gender});	El ejercicio nos pide utilizar la abreviatura de propiedad del objeto con objetos literales para crear y devolver un objeto con las propiedades name, age y gender.

17.- Escribe funciones breves y declarativas con ES6:

ES6 nos permite eliminar la palabra clave function y los dos puntos al definir funciones en objetos.

const bicycle = {
gear: 2,
setGear(newGear) {
this.gear = newGear;
}
};
 	
bicycle.setGear(3);
console.log(bicycle.gear);
	El ejercicio pide refactorizar la función setGear dentro del objeto bicycle para que utilice la sintaxis abreviada.

18.- Usa la sintaxis de clases para definir una función constructora:

class Vegetable {
constructor(name){
this.name = name;
}
}
const carrot = new Vegetable('carrot');
	El ejercicio pide crear una clase con un método constructor.

19.- Utiliza getters (accesores) y setters (mutadores) para controlar el acceso a un objeto:

Las funciones getter están destinadas simplemente a devolver (get) el valor de una variable. Las funciones setter por otra parte sirven para modificar (set) el valor de una variable. Este cambio puede implicar cálculos, o incluso sobreescribir completamente el valor de la variable.

Ten en cuenta la sintaxis usada para invocar el getter y el setter, ni siquiera se ven como funciones. Estos son importantes porque ocultan los detalles internos de la implementación. Es convención preceder el nombre de las variables privadas de [_]. Sin embargo, la practica en sí misma no hace que la variable sea privada.

class Thermostat {
  constructor(fahrenheit){
  this.fahrenheit = fahrenheit;
  }
  get temperature(){
  return (5/9) * (this.fahrenheit - 32);
  }
  set temperature(celsius){
  this.fahrenheit = (celsius * 9.0) / 5 + 32;
  }
}
const thermos = new Thermostat(76); // Ajuste en escala Fahrenheit
let temp = thermos.temperature; // 24.44 en Celsius
thermos.temperature = 26;
temp = thermos.temperature; // 26 en Celsius
	El ejercicio pide crear una clase Thermostat con un constructor que acepta una temperatura en fahrenheit.
	Crea un getter para obtener la temperatura en celsius.
	Crea un setter para ajustar la temperatura en celsius.

20.- Crea un módulo para scripts:

Hoy JavaScript es gigante y algunos sitios web están construidos casi por completo con JS. Con la finalidad de hacerlo más modular, limpio y mantenible, ES6 introdujo una manera de compartir código fácilmente entre archivos JS. Esto implica exportar partes de un archivo para usar en uno o más archivos. Para aprovechar esta funcionalidad necesitas crear un script en tu documento HTML con un [type =”module”]. Un script que utilice [module] ahora podrá utilizar las características [import] y [export].

<html> <body> <script type="module" src="index.js"></script> </body> </html>	El ejercicio pide agregar un script tipo module al documento HTML y asignarle el archivo fuente index.js

21.- Utiliza la exportación para compartir un bloque de código:

const uppercaseString = (string) => {
  return string.toUpperCase();
}
 	
const lowercaseString = (string) => {
  return string.toLowerCase()
}
export {uppercaseString, lowercaseString};
	El ejercicio pide exportar dos funciones utilizando un método a elegir.

22.- Reutiliza código JavaScript utilizando import:

import {uppercaseString, lowercaseString} from "./string_functions.js"
 	
uppercaseString("hello");
lowercaseString("WORLD!");
	El ejercicio pide agregar la declaración import, que permita usar las funciones que se encuentran en string_functions.js.
	
	"./" busca en la misma carpeta que el archivo actual.

23.- Usar * para importar todo el contenido de un archivo:

Sí quieres importar todo el contenido de un archivo, puedes utilizar la sintaxis [import * as]. Esto creará un objeto que contiene todas las exportaciones del archivo. Por lo cual puedes acceder a todas sus funciones como si fueran las propiedades de un objeto.

import * as stringFunctions from "./string_functions.js"; stringFunctions.uppercaseString("hello"); stringFunctions.lowercaseString("WORLD!");	El ejercicio pide importar string_functions.js y almacenarlo en un objeto llamado stringsFunctions.

24.- Crear un fallback de exportación con export default:

[export default] es usado para declarar un valor fallback para un módulo o archivo por lo que solo podremos tener uno (exportación por defecto).

export default function subtract(x, y) {
  return x - y;
}
	El ejercicio da una función y dice que debe ser el valor fallback para el módulo.

25.- Importa una exportación por defecto:

Para importar una exportación por defecto necesitas utilizar la sintaxis import de manera diferente. La sintaxis difiere en un punto clave, el nombre de la función no está rodeado por llaves {}. Puedes usar cualquier nombre al importar un valor por defecto.

import subtract from "./math_functions.js"; subtract(7,4);	El ejercicio pide importar como exportación por defecto, desde math_functions.js. Y nombrar la importación como subtract.

26.- Crea una promesa en JavaScript:

Las promesas son exactamente lo que suenan, se utilizan para hacer promesas de que harás algo, habitualmente de forma asíncrona. Cuando se completa la tarea, o cumples la promesa o no la cumples. [Promise] es una función constructora por lo que necesitas [new] para crear una. Recibe una función como argumento con dos parámetros: resolve y reject, que se utilizarán para determinar el resultado de la promesa.

const makeServerRequest = new Promise((resolve, reject) => { });	El ejercicio pide crear una promesa llamada makeServerRequest y pasarle una función con parámetros resolve y reject al constructor.

27.- Completa una promesa con resolve y reject:

Una promesa tiene tres estados: pending, fulfilled y rejected. Los parámetros resolve y reject enviados a promise son utilizados, en el caso del primero, cuando quieres que tú promesa tenga éxito. Y reject cuando quieres que falle.

const makeServerRequest = new Promise((resolve, reject) => {
 	
let responseFromServer;
 	
if(responseFromServer) {
resolve ("We got the data");
} else {
reject ("Data not received");
}
});
	El ejercicio pide hacer una función promise que maneje el exito y el el fallo.
	
	Sí responseFromServer es true, llama al método resolve para completar satisfactoriamente la promesa, si es false, utiliza el método reject.

28.- Manejar una promesa cumplida usando then:

Las promesas son muy útiles cuando tienes un proceso que toma una cantidad de tiempo indefinido (algo asíncrono, por ejemplo), como una petición a un servidor.

Cuando haces una petición, a menudo, cuando termina, normalmente quieres hacer algo con la respuesta del servidor. Esto se puede lograr utilizando el método [then]. Este método se ejecuta inmediatamente después de que tu promesa se cumpla con resolve (se le pasará un argumento result).

const makeServerRequest = new Promise((resolve, reject) => {
  let responseFromServer = true;
 	
  if(responseFromServer) {
  resolve("We got the data");
 	
  makeServerRequest.then(result => {});
  console.log(result);
 	
  } else {
  reject("Data not received");
  }
});
	El ejercicio pide añadir el método then a tu promesa. Usa result como parámetro de tu función callback, asimismo imprime result en la consola.

29.- Maneja una promesa rechazada usando catch:

[catch] es el método utilizado cuando tu promesa ha sido rechazada. Se ejecuta directamente después de que sea llamado el método reject (se le pasará un argumento error).

const makeServerRequest = new Promise((resolve, reject) => {
 	
  let responseFromServer = false;
 	
  if(responseFromServer) {
  resolve("We got the data");
  } else {
  reject("Data not received");
 	
  makeServerRequest.catch(error =>{});
  console.log(error);
  }
});
 	
makeServerRequest.then(result => {
  console.log(result);
});
	El ejercicio pide añadir el método catch a tu promesa. Usa error como parámetro de tu función callback e imprime error en consola.