Programación funcional.

1.- Aprende sobre programación funcional:

La programación funcional es un estilo de programación donde las soluciones son simples, funciones aisladas, sin ningún efecto secundario fuera del ámbito de la función:

[INPUT => PROCESS => OUTPUT]

La programación funcional se refiere a:

1.- Funciones aisladas: sin dependencia alguna del estado del programa, el cual incluye variables globales sujetas a cambios.
2.- Funciones puras: una misma entrada siempre da una misma salida.
3.- Funciones con efectos secundarios limitados: cualquier cambio o mutación en el estado del programa fuera de la función son cuidadosamente controlados.

// Función que retorna una cadena de texto representando una taza de té verde
const prepareTea = () => 'greenTea';
 	
/*
Dada una función (representando el tipo de té) y el número de tazas necesarias, la siguiente función retorna un arreglo de cadenas de texto (cada una representando un tipo específico de té).
*/
const getTea = (numOfCups) => {
  const teaCups = [];
 	
  for(let cups = 1; cups <= numOfCups; cups += 1) {
  const teaCup = prepareTea();
  teaCups.push(teaCup);
  }
  return teaCups;
};
 	
const tea4TeamFCC = getTea(40);
	En el editor de código, las funciones prepareTea y getTea ya están definidas. Llama a la función getTea para obtener 40 tazas de té para el equipo y guárdalas en la variable tea4TeamFCC.

2.- Comprende la terminología de la programación funcional:

Los callbacks son funciones que se pasan a otra función para decidir la invocación de esa función. Es posible que las hayas visto pasar a otros métodos, por ejemplo en filter, la función callback le dice a JavaScript los criterios para filtrar un arreglo.

Las funciones que pueden ser asignadas a una variable, pasadas a otra función o devueltas desde otra función como cualquier otro valor normal, se llaman funciones de primera clase. En JavaScript todas las funciones son funciones de primera clase.
Las funciones que toman una función como argumento, o devuelven una función como valor de retorno, se denominan funciones high order.
Cuando las funciones se pasan o se devuelven desde otra función, las funciones que se pasaron o devolvieron se pueden llamar lambda

// Función que retorna una cadena de texto representando una taza de té verde
const prepareGreenTea = () => 'greenTea';
 	
// Función que retorna una cadena de texto representando una taza de té negro
const prepareBlackTea = () => 'blackTea';
 	
/*
Dada una función (representando el tipo de té) y el número de tazas necesarias, la siguiente función retorna un arreglo de cadenas de texto (cada una representando un tipo específico de té).
*/
const getTea = (prepareTea, numOfCups) => {
  const teaCups = [];
 	
  for(let cups = 1; cups <= numOfCups; cups += 1) {
  const teaCup = prepareTea();
  teaCups.push(teaCup);
  }
  return teaCups;
};
 	
const tea4GreenTeamFCC = getTea(prepareGreenTea, 27);
const tea4BlackTeamFCC = getTea(prepareBlackTea, 13);
 	
console.log(
  tea4GreenTeamFCC,
  tea4BlackTeamFCC
);
	Podemos modificar la función getTea para que acepte una función como parámetro y así poder cambiar el tipo de té que se prepara. Esto hace que getTea sea más flexible y da al programador más control cuando las solicitudes del cliente cambian.
	
	Prepara 27 tazas de té verde y 13 de té negro y almacenalas en las variables . Ten en cuenta que la función getTea ha sido modificada por lo que ahora recibe una función como primer argumento.

3.- Comprende los peligros de usar código imperativo:

La programación funcional es un buen hábito. Mantiene tu código fácil de manejar y evita que tengas errores. Pero antes de llegar allí, veamos el enfoque imperativo de la programación para destacar dónde pueden tener problemas.
El imperativo se utiliza para dar órdenes. De igual forma en programación el estilo imperativo le da a la computadora un conjunto de órdenes para llevar a cabo una tarea. A menudo las sentencias cambian el estado del programa, como actualizar variables locales. Un ejemplo clásico es un bucle for que da instrucciones exactas para iterar sobre los índices de un arreglo.
Por el contrario, la programación funcional es una forma de programación declarativa. Le dice al ordenador lo que quieres hacer llamando a un método o función.

JavaScript ofrece muchos métodos predefinidos que manejan tareas comunes para que no necesites escribir cómo debe ejecutarlas el equipo. Por ejemplo, en lugar de usar un bucle for, se podría llamar al método map, que maneja los detalles de iteración sobre un arreglo.

Esto ayuda a evitar errores semánticos, como los “Off by One Errors” (iterar un bucle más o menos veces de las necesarias).

// tabs es un arreglo de títulos de cada sitio abierto dentro de la ventana
const Window = function(tabs) {
  this.tabs = tabs; // Mantenemos un registro del arreglo dentro del objeto
};
// Cuando unes dos ventanas en una
Window.prototype.join = function(otherWindow) {
  this.tabs = this.tabs.concat(otherWindow.tabs);
  return this;
};
// Cuando abres una nueva pestaña al final
Window.prototype.tabOpen = function(tab) {
  this.tabs.push('new tab'); // Abramos una nueva pestaña por ahora
  return this;
};
// Cuando cierras una pestaña
Window.prototype.tabClose = function(index) {
 	
  // Cambia solo el código debajo de esta línea
 	
  const tabsBeforeIndex = this.tabs.splice(0, index)
 // Obtiene las pestañas antes de la pestaña
 	
  const tabsAfterIndex = this.tabs.splice(1); // Obtiene las pestañas después de la pestaña
 	
  this.tabs = tabsBeforeIndex.concat(tabsAfterIndex);
 // Únelos juntos
 	
  return this;
 };
	Está navegando por la web y quiere rastrear las pestañas que ha abierto. Intentemos modelar este código con programación orientada a objetos. Un objeto ventana está formado por pestañas. Los títulos de cada sitio abierto en cada objeto ventana se mantienen en un arreglo. Después de trabajar en el navegador, se quieren imprimir las pestañas que todavía estén abiertas. Las cerradas se eliminan de la matriz y las nuevas se añaden al final de la misma. Examine el código en el editor. Está utilizando un método que tiene efectos secundarios en el programa, causando un comportamiento incorrecto. La lista final de pestañas abiertas, almacenadas en finalTabs.tabs deberia ser ['FB', 'Gitter', 'Reddit', 'Twitter', 'Medium', 'new tab', 'Netflix', 'YouTube', 'Vine', 'GMail', 'Work mail', 'Docs', 'freeCodeCamp', 'new tab'] pero la lista producida por el código es un poco diferente.
	
	Cambia Window.prototype.tabClose para que elimine la pestaña correcta.

4.- Evita mutaciones y efectos secundarios utilizando programación funcional:

En la programación funcional, cambiar o alterar cosas se denomina mutación, y el resultado es conocido como efecto secundario. Una función, idealmente, debe ser una función pura, lo que significa que no provoca ningún efecto secundario.

let fixedValue = 4;
 	
function incrementer() {
  return fixedValue + 1
}
	Completa el código de la función incrementer para que devuelva el valor de la variable global fixedValue incrementada en uno.

5.- Pasa argumentos para evitar la dependencia externa en una función:

Otro principio de la programación funcional es declarar siempre sus dependencias de forma explícita. Esto significa que si una función depende de que una variable u objeto esté presente, después pasa esa variable u objeto directamente a la función como argumento. Esto hace que la función sea más fácil de probar, se sabe exactamente lo que necesita, y no dependerá de nada más en tu programa.

let fixedValue = 4;
 	
function incrementer(fixedValue) {
  return fixedValue + 1
}
	Actualiza la función incrementer para declarar sus dependencias. Escribelá de forma que reciba un argumento y devuelva su valor aumentado en uno.

6.- Refactoriza variables globales por fuera de funciones:

var bookList = ["The Hound of the Baskervilles", "On The Electrodynamics of Moving Bodies", "Philosophiæ Naturalis Principia Mathematica", "Disquisitiones Arithmeticae"];
 	
function add(arr, bookName) {
  let newArr = [...arr];
  newArr.push(bookName);
  return newArr;
}
 	
function remove(arr, bookName) {
  let newArr = [...arr];
  if (newArr.indexOf(bookName) >= 0) {
 	
  newArr.splice(newArr.indexOf(bookName), 1);
  return newArr;
  }
}
	Reescribe el código para que el arreglo global bookList no sea cambiado dentro de ninguna de las funciones. La función add debe agregar el bookName dado al final del arreglo pasado a esta y devolver un nuevo arreglo. La función remove debe eliminar el bookName dado del arreglo pasado a esta.

7.- Usa el método map para extraer datos de un arreglo:

El método map iterará sobre cada elemento de un arreglo y devuelve un nuevo arreglo que contiene los resultados de llamar a la función callback en cada elemento. Esto lo hace sin mutar el arreglo original. Cuando se utiliza una función callback, se pasan tres argumentos:

el elemento actual que se está procesando, el índice de ese elemento y el arreglo al que llamó el método map.

const ratings = watchList.map(item => ({
  title: item["Title"],
  rating: item["imdbRating"]
}))
//title: "valor de la propiedad 'Title'",
//rating: "valor de la propiedad 'imdbRating'"
console.log(JSON.stringify(ratings));
	Usa map para crear un nuevo array que guarde los valores de Title y imdbRating sin modificar el original.

8.- Implementa map en un prototipo:

const filteredList = watchList .filter((movie) => movie.imdbRating >= 8.0) .map((movie) => ({title: movie.Title, rating: movie.imdbRating})) console.log(filteredList);	Utiliza una combinación de filter y map en watchlist. Para filtrar un arreglo incluyendo los elementos para los cuales imdbRating sea mayor a 8. Incluye el rating y el título de cada película en un arreglo.

9.- Usa el método filter para extraer datos de un arreglo:

Array.prototype.myFilter = function(callback) {
 	
  const newArray = [];
 	
 for(let i = 0; i < this.length; i++){
 	
  if(callback(this[i], i, this) == true){
  newArray.push(this[i])
  }
 	
 }
  return newArray;
};
	Escribe tu propio Array.prototype.myFilter(), debe comportarse exactamente como filter(). Se puede acceder a la instancia de Array en el método filter usando this.

11.- Devolver parte de un arreglo mediante el método slice:

function sliceArray(anim, beginSlice, endSlice) {
  return anim.slice(beginSlice, endSlice)
}
 	
const inputAnim = ["Cat", "Dog", "Tiger", "Zebra", "Ant"];
sliceArray(inputAnim, 1, 3);
	Utiliza el método slice en la función sliceArray para retornar parte del arreglo anim dado los índice beginSlice y endSlice. La función debe devolver un arreglo.

12.- Remueve elementos de un arreglo utilizando slice en lugar de splice:

function nonMutatingSplice(cities) {
 	
  return cities.slice(0, 3);
 	
}
 	
const inputCities = ["Chicago", "Delhi", "Islamabad", "London", "Berlin"];
 	
nonMutatingSplice(inputCities);
	Reescribe la función nonMutatingSplice usando slice en lugar de splice. Debe limitar el arreglo proporcionado cities a una longitud de 3 y devolver un nuevo arreglo con los tres primeros elementos. No modifiques el arreglo original.

13.- Combina dos arreglos utilizando el método concat:

function nonMutatingConcat(original, attach) {
  return original.concat(attach)
}
 	
const first = [1, 2, 3];
const second = [4, 5];
nonMutatingConcat(first, second);
	Usa el método concat en la función nonMutationalConcat para concatenar attach al final de original. La función debe devolver el arreglo concatenado.

14.- Agrega elementos al final de un arreglo utilizando concat en lugar de push:

function nonMutatingPush(original, newItem) {
 	
  return original.concat(newItem);
 	
}
 	
const first = [1, 2, 3];
const second = [4, 5];
nonMutatingPush(first, second);
	Cambia la función nonMutationalPush de manera que utilice concat para unir newItem al final de original sin alterar los arreglos. La función debe devolver un arreglo.

15.- Utiliza el método reduce para analizar datos:

function getRating(watchList) {
 const filterList = watchList
 .filter(movie => movie.Director === "Christopher Nolan")
 .map(movie => Number(movie.imdbRating))
 .reduce((mid, rating) => mid + rating)
 / watchList.filter(movie => movie.Director === "Christopher Nolan").length
 	
 return filterList
}
 	
console.log(getRating(watchList));
	Utilice reduce para encontrar la calificación media en IMDB de las películas dirigidas por Nolan. Utiliza filter y map para obtener los datos que necesitas. Puede que necesites crear otras variables para devolver la calificación media con getRating. Ten en cuenta que los valores de calificación se guardan como cadenas de texto.

16.- Utiliza las funciones de orden superior map, filter o reduce para resolver un problema complejo:

const squareList = arr => {
 return arr
 .filter(num => num > 0 && num % parseInt(num) === 0)
 .map(num => Math.pow(num, 2));
};
 	
const squaredIntegers = squareList([-3, 4.8, 5, 3, -3.2]);
console.log(squaredIntegers);
	Completa el código para la función squareList usando cualquier combinación de map, filter y reduce. La función debe devolver un nuevo arreglo que contenga los cuadrados de solamente los enteros positivos.

17.- Ordena un arreglo alfabéticamente con el método sort:

function alphabeticalOrder(arr) {
 const sorted = arr.sort((a,b) => a > b ? 1 : -1)
 return sorted
}
 	
alphabeticalOrder(["a", "d", "c", "a", "z", "g"]);
	Utiliza el método sort en la función alphabeticalOrder para ordenar los elementos de arr en orden alfabético. La función debe devolver un arreglo ordenado.

18.- Devuelve un arreglo ordenado sin cambiar el arreglo original:

const globalArray = [5, 6, 3, 2, 9];
 	
function nonMutatingSort(arr) {
 return [].concat(arr).sort((a,b) => a - b)
}
 	
nonMutatingSort(globalArray);
	Usa el método sort para ordenar el arreglo. La función debe devolver un nuevo arreglo y no mutal la variable globalArray.

19.- Divide una cadena en un arreglo utilizando el método split:

function splitify(str) {\
// /\W/ indica todos los caracteres que no sean una letra
  return str.split(/\W/)
}
 	
splitify("Hello World,I-am code");
	Utiliza el método split dentro de la función splitify para dividir str en un arreglo de palabras. La función debe devolver un arreglo. Ten en cuenta que las palabras no siempre estan separadas por espacios y que el arreglo no debe contener signos de puntuación.

20.- Combina un arreglo en una cadena utilizando el método join:

function sentensify(str) {
 	
 let a = str.split(/\W/)
 return a.join(" ")
 	
}
 	
sentensify("May-the-force-be-with-you");
	Utiliza el método join dentro de la función instensify para hacer una oración a partir de las palabras de la cadena str. No utilices el método replace.

21.- Aplica programación funcional para convertir cadenas a slugs de URL:

function urlSlug(title) {
 	
  return title
  .toLowerCase()
  .trim()
  .split(/\s+/)
  .join('-')
 	
}
 	
urlSlug(" Winter Is Coming");
	Rellena la función urlSlug para convertir title en una cadena de texto en minúscula separada por guiones

22.- Usa el método every para comprobar que cada elemento de un arreglo atienda un criterio:

function checkPositive(arr) {
 	
  return arr.every(function (value){
  return value > 0
  })
 	
}
checkPositive([1, 2, 3, -4, 5]);
	Utiliza el método every para comprobar si cada elemento de arr es positivo. La función debe devolver un valor boolean.

23.- Usa el método some para comprobar si algún elemento de un arreglo cumple un criterio:

function checkPositive(arr) {
  return arr.some((value) => value > 0)
}
 	
checkPositive([1, 2, 3, -4, 5]);
	Utiliza some para comprobar si algún elemento de arr es positivo. La función debe devolver un valor boolean.

24.- Introducción a la currificación y a la aplicación de funciones parciales:

function add(x) {
  return function (y){
  return function (z){
  return x + y + z
  }
  }
}
add(10)(20)(30);
	Completa el cuerpo de la función add para que use currificación para agregar los parámetros x, y y z.