Programación Funcional en JavaScript

Ejercicios para el taller

Links a los ejercicios en codesandbox.

Introduccion

La programación funcional es un paradigma de programación orientado en la composición de funciones. Es parte de la familia de los lenguajes declarativos (justo como dice la definición de React).

Existen dos grandes grupos, los lenguajes imperativos y declarativos.
El primer grupo define las aplicaciones como una serie de instrucciones donde le indicamos a la computadora por pasos cómo actualizar el estado de la aplicación.
Mientras que los declarativos, describen el comportamiento de la aplicación a través de expresiones.

JavaScript es un lenguaje multiparadigma, por lo que nos permite aplicar técnicas de lenguajes imperativos como declarativos. En la naturaleza hay lenguajes que se inclinan más de un lado que del otro, pero la mayoría de los lenguajes modernos soportan el multiparadigma.

```
function cuadradosImp(numeros) {
  let cuadrados = [];
  for (let i = 0; i < numeros.length; i++) {
    let numero = numeros[i];
    cuadrados.push(numero ** 2);
  }

  return cuadrados;
}
```

```
const sqrNum = (n) => n ** 2;
const cuadradosFunc = (numeros) => numeros.map((n) => sqrNum(n));
```

Podemos observar como, en el primer ejemplo, nosotros instruimos a la aplicación a cómo queremos obtener el cuadrado de los números, esta forma de leerlo puede verse más natural a la forma en la que funciona nuestro cerebro.

A diferencia de la segunda forma, nosotros declaramos que queremos obtener para poder lograrlo, en la segunda función al usar map, nosotros estamos diciendo “Quiero convertir una entrada en una salida diferente” y para poder obtener el cuadrado de los números, creamos otra declaración a la que le decimos “Quiero el cuadrado de un número”.

Disclaimer

La programación funcional en la naturaleza tiene muchas aplicaciones mas que hacer aplicaciones web y estas pueden volverse tan académicas y complejas, como simples en los casos de estos ejemplos, en esta clase nos enfocaremos en hacer que tengan sentido estos conceptos para nosotros y ver el provecho que le podemos sacar al escribir aplicaciones JavaScript.

Funciones Puras

```
async function app() {
  let promise = fetch("https://api.github.com/users/codigofacilito");
  console.log(promise);

  let result = await promise
    .then((result) => {
      console.log(result);
      return result;
    })
    .catch((err) => console.error(err))
    .finally(() => console.info("cleanup"));
  console.log(result);
}
```

Si recuerdan el ejemplo para Promises, si nosotros ejecutamos esta función un millón de veces, esta función nos podría dar resultados variados a lo largo de estas ejecuciones.

No podemos confiar en esta función ya que introduce “Efectos secundarios”. En la programación funcional se introducen restricciones para escribir aplicaciones que reduzcan el área de errores.

Por lo tanto, uno de los beneficios que obtenemos de la programación funcional es la confianza sobre nuestra aplicación, ya que nos incita a remover al máximo los efectos secundarios. Nótese que dice “Al máximo”, no es posible tener una aplicación 100% pura.

Existe una subcategoría de la programación funcional llamada “pure functional programing” que se basa en la composición de funciones estrictamente puras, sin embargo el caso de uso de estos lenguajes está muy enfocado en casos especiales.

Por lo que, los lenguajes de programación introducen una forma o técnicas para poder capturar estos efectos secundarios en las aplicaciones, cosas como imprimir en consola, hacer llamadas HTTP, actualizar una variable global, etc.

Un sneak peek es que en React, si bien no hay algo que lo prohiba, la librería está pensada en que construyas componentes Puros y tiene un mecanismo que verán en el futuro que se encarga del manejo de los efectos secundarios (coff coff useEffect).

```
function pureApp(repos) {
  return repos.map((rep) => rep.full_name);
}

/// Puente que divide mi app pura de la impura.

async function impureApp() {
  const req = await fetch("https://api.github.com/users/codigofacilito/repos");
  const repos = await req.json();
  const names = pureApp(repos);
  console.log(names);
}

/**
 * Playground.
 */
impureApp();
```

En este ejemplo podemos mirar que, aunque el cambio se haya visto simple, la división importa, naturalmente en aplicaciones más grandes el impacto es mayor, pero lo fundamental es no permitir que las cosas que están en el mundo impuro, puedan pasar al mundo puro.

Si ejecutamos un millón de veces pureApp podemos estar seguros de que esta siempre nos va a dar los resultados esperados.

Este es uno de los grandes beneficios y la popularidad de implementar este paradigma, en React, piensen en esto cuando creen componentes encargados de pintar una lista en la pantalla, para poder obtener la lista, puede que sea necesario hacer una llamada a una api y entonces ustedes podrán pintar la lista, pero si dividen la lógica que quien hace la llamada al API (efectos secundarios) y quien construye el componente basado en la lista, podrán tener un mayor control de su aplicación.

En resumen

Implementar funciones puras nos da confianza y seguridad, por lo que son muy preferidas al momento de construir aplicaciones, no es posible tener una aplicación 100%, por lo que son buenas para guardar la lógica dura de nuestras apps y alejar de ellas los efectos secundarios al máximo.

También son un buen camino para generar desacoplamiento en las aplicaciones, ya que cada una adquiere una única responsabilidad.

Funciones de alto orden

¿Qué pasa si necesitamos comunicar información entre una función pura y una función impura en dos direcciones?, del ejemplo anterior pudimos ver que al crear la separación entre las dos funciones, estas podían comunicar valores a través de dos vehículos: los argumentos y los valores de retorno.

En aplicaciones más complejas, además de solo obtener el valor de una función e imprimirlo, es necesario encadenar más acciones adicionales que cumplan con el flujo de una aplicación. Pero como vimos, no tenemos que permitir que nuestra aplicación se llene de impurezas para tratar de conseguir este objetivo.

```
function obtenerNombresDeRepositorios(repos) {
  return repos.map((rep) => rep.full_name);
}

function obtenerNombreDelUsuario(usuario) {
  return `Usuario: ${usuario.login}`;
}

function obtenerResumenDeUsuario({ nombre, repos, agregarRepo }) {
  return {
    nombre,
    repos,
    agregarRepo
  };
}

function agregarNuevoRepositorio(repos, nuevoRepo) {
  return [...repos, nuevoRepo];
}
```

```
async function myApp() {
  const githubRepos = await obtenerJson(
    "https://api.github.com/users/codigofacilito/repos"
  );
  const githubUser = await obtenerJson(
    "https://api.github.com/users/codigofacilito"
  );

  const repos = obtenerNombresDeRepositorios(githubRepos);
  const nombre = obtenerNombreDelUsuario(githubUser);
  const resumen = obtenerResumenDeUsuario({
    nombre,
    repos,
    agregarRepo: (nuevoRepo) => agregarNuevoRepositorio(repos, nuevoRepo)
  });

  console.log({ resumen });
  const nuevosRepos = resumen.agregarRepo("Nuevo");
  console.log({ nuevosRepos, resumen: resumen.repos });
}
```

Como vemos en este ejemplo, algo que hace posible a JavaScript implementar programación funcional es que las clases son “First class citizens”, esto quiere decir que se comportan como cualquier otro valor, se pueden asignar a variables, pasar como argumentos y regresar como resultado de una función.

En nuestro obtenerResumeDeUsuario le estamos diciendo que tiene la capacidad de “agregar un nuevo repo”, sin embargo no sabe cómo hasta que le proveemos una función que le enseñara a como hacerlo.

Al pasar una función como argumento o regresar una función como resultado de otra función, estamos creando funciones de alto orden.

```
function obtenerNombresDeRepositorios(repos) {
  return repos.map((rep) => rep.full_name);
}

function obtenerResumenDeUsuario(nombre, repos) {
  return (agregarRepo) => {
    return {
      nombre,
      repos,
      agregarRepo
    };
  };
}
```

Probablemente esto es algo que ya hayan estado haciendo en JavaScript y solo hacía falta el concepto, pero hacer esto es muy común y conveniente para reutilizar componentes y lógica a lo largo de sus aplicaciones.

Por ejemplo, map contiene la lógica de iterar y crear un nuevo objeto a partir de uno existente. Sin embargo no sabe como convertir tu objeto, tu le tienes que enseñar a hacerlo a través de una función. Así como obtenerResumenDeUnUsuario tiene la lógica de presentar la información de forma estructurada, pero no sabe los valores que va a presentar o las acciones que tiene que ejecutar, hasta que nosotros le enseñamos a hacerlo a través de sus argumentos.

En resumen

Las funciones de alto orden ayudan a preservar la pureza de las funciones, ya que permiten comunicar valores entre dos funciones, sin contaminarse entre ellas. JavaScript nos permite

pasar valores entre los argumentos y gracias a esto construir nuestras funciones de alto orden. También ayudan a reutilizar código entre la aplicación.

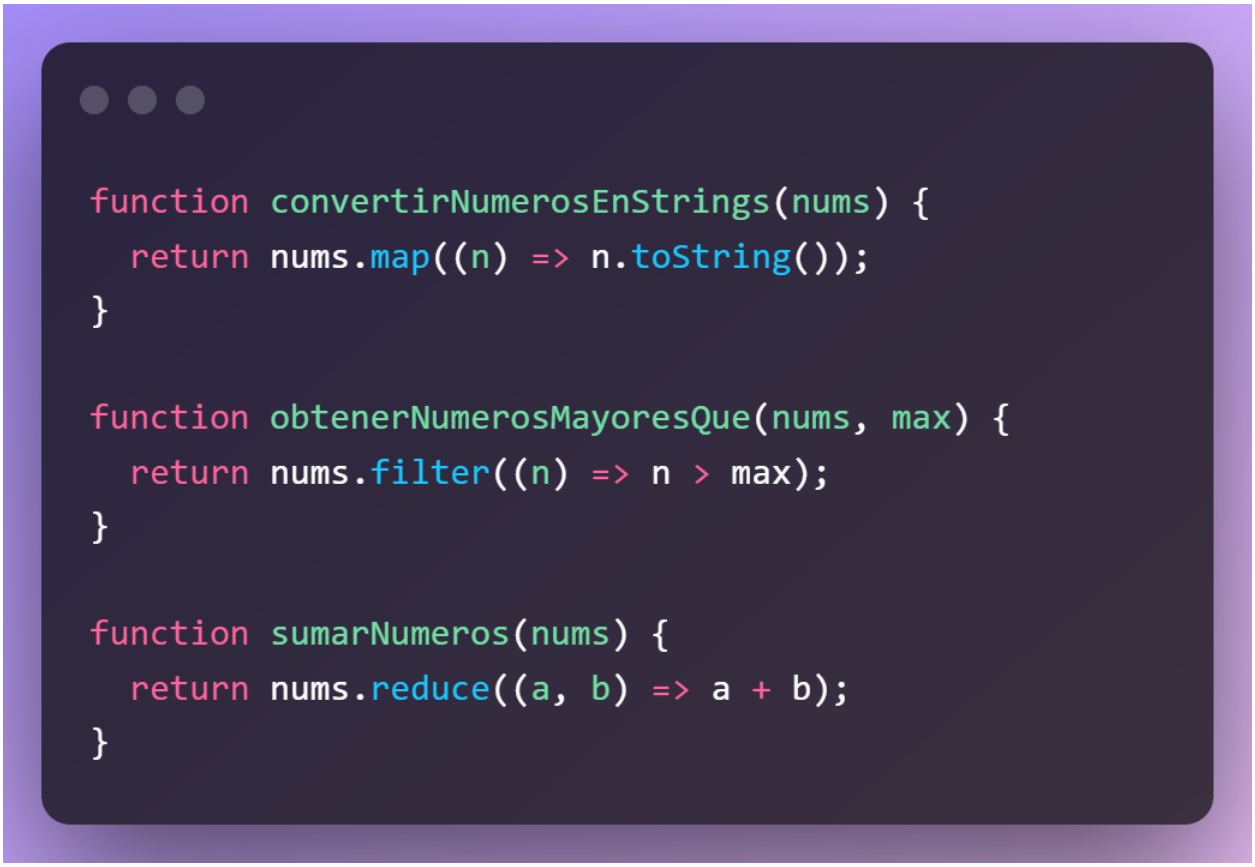
Filter, Map, Reduce

Vamos a hacer un repaso obligado en filter, map y reduce. Como ya hemos visto estos son métodos de Array y nos permite hacer operaciones de forma muy práctica. No hay que entrar en discusión de si son mejores o no que los ciclos imperativos, ya que hay que dejar algo muy claro, estos cumplen un caso de uso muy distinto.

Recuerden que la programación funcional se basa en acciones y declaraciones, no en flujos de control. Por lo que Filter, Map y Reduce cumplen un papel específico, más allá de iterar arreglos.

- Map: Convertir un tipo en otro tipo.
- Filter: Filtrar elementos.
- Reduce: Combinar dos valores en uno.

Mientras que los ciclos for, while, etc. son crear estructuras de control.



```
function convertirNumerosEnStrings(nums) {
  return nums.map((n) => n.toString());
}

function obtenerNumerosMayoresQue(nums, max) {
  return nums.filter((n) => n > max);
}

function sumarNumeros(nums) {
  return nums.reduce((a, b) => a + b);
}
```

El método Reduce es uno de mis favoritos y si quieres aprender a dominarlo al máximo te recomiendo leer sobre Transducers.

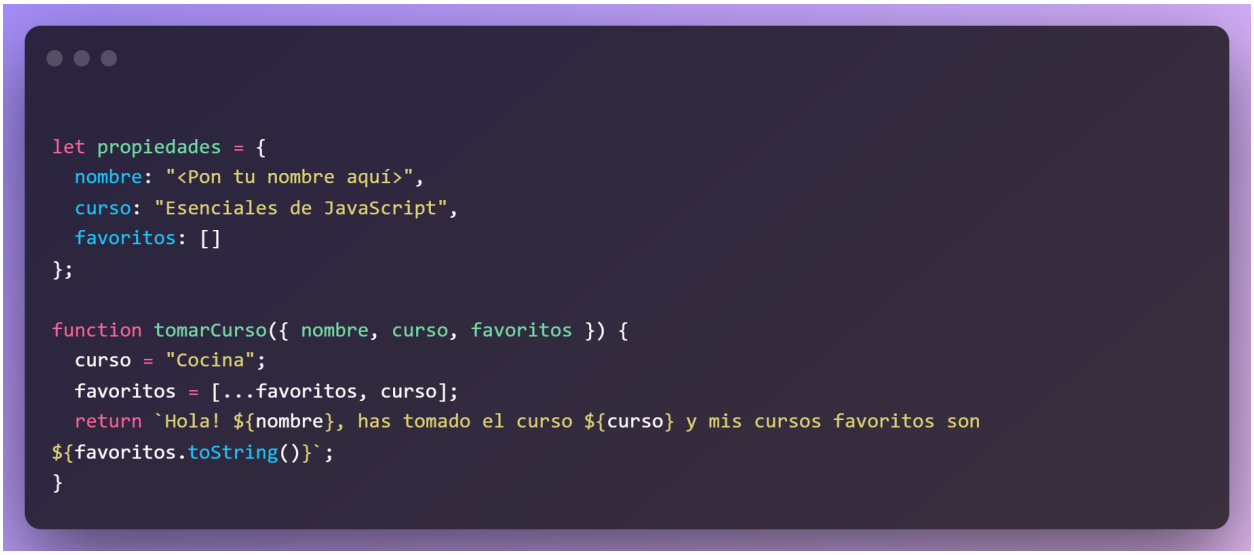
En resumen

Filter, Map y Reduce son funciones con un propósito, más que simples iteradores. Podemos ver una de las notables diferencias entre la programación funcional y la imperativa, es que aquí por defecto tenemos funciones con propósitos específicos más que herramientas para construir nosotros dichos propósitos.

Inmutabilidad

Tristemente en JavaScript no tenemos objetos inmutables, ya que como vimos en la clase pasada, los objetos aquí se pasan por referencia y podemos crear efectos secundarios de forma accidental.

Un objeto inmutable es aquel que no puede cambiar su estado una vez fue creado. En las dos grandes categorías de tipos que tenemos en JavaScript (primitivos y objetos), todos los primitivos son inmutables, pero los objetos son totalmente mutables.



```
let propiedades = {
  nombre: "<Pon tu nombre aquí>",
  curso: "Esenciales de JavaScript",
  favoritos: []
};

function tomarCurso({ nombre, curso, favoritos }) {
  curso = "Cocina";
  favoritos = [...favoritos, curso];
  return `Hola! ${nombre}, has tomado el curso ${curso} y mis cursos favoritos son
${favoritos.toString()}`;
}
```

Para esto tenemos que aprovechar herramientas que el lenguaje nos provee como el spread operator y destructuring.

A diferencia de lo que pudiéramos pensar, la reasignación dentro de una función si es correcta, mientras esta no genere efectos secundarios fuera de sí misma. Usando el destructuring podemos tomar los valores primitivos y trabajar con ellos sin mucho problema. Hay que tener cuidado aun con los Objetos, ya que estos no generan una copia profunda y pudieras mutarlos.

En React te encontrarás haciendo esto muchas veces, cada que estes pasando props entre tus componentes.

Hay una iniciativa para estandarizar las estructuras de datos inmutables, pero hasta que eso no pase, tenemos que usar esto https://tc39.es/proposal-record-tuple/tutorial/

También hay herramientas que te pueden ayudar a crear estructuras de datos inmutables, pero son dependencias externas al lenguaje.

En resumen

Por defecto en JavaScript no tenemos objetos inmutables, por lo que tenemos que usar las features del lenguaje para preservar la pureza de nuestras funciones. Const a diferencia de lo que pudiéramos pensar no crea objetos inmutables, sólo valores no reasignables.

En Resumen

La programación funcional nos ofrece técnicas y vehículos para construir buenas aplicaciones, sin embargo esto no quiere decir que sea una mejor que otra. La programación imperativa nos da control sobre los procesos de nuestra aplicación y esto puede resultar en una mejor eficiencia, ya que si tratamos todo como objetos inmutables vamos a tener un gran coste en memoria y aplicaciones más lentas.

Por lo tanto y como JavaScript es un lenguaje multiparadigma podemos tomar las mejores herramientas de ambos mundos para mejores aplicaciones. Toma en consideración el poner las cosas valiosas de tus aplicaciones en funciones puras que sigan los principios de la programación funcional y crea puentes que sirvan como pegamento para unir de forma segura todos los elementos impuros.

Referencias

- Railway oriented programming - https://vimeo.com/113707214
- Functional Light - https://github.com/getify/Functional-Light-JS
- Composing Software: The Book - https://medium.com/javascript-scene/composing-software-the-book-f31c77fc3ddc