



Promesas

Async/Await





>_

**Las funciones asíncronas
permiten que usemos
expresiones “await” dentro
del cuerpo de la función.**






>_

**Estas expresiones nos
permiten trabajar con
promesas como si fueran
código síncrono.**



 Veámoslo en práctica:

- Async/Await





ES Modules



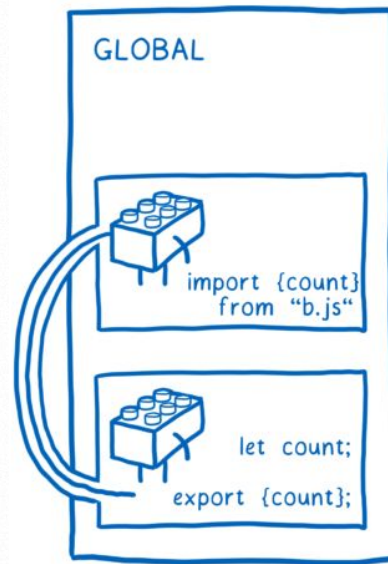
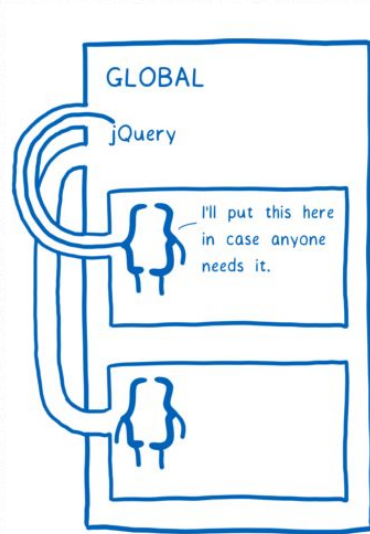


**Un módulo nos permite
dividir la funcionalidad del
código en piezas más
pequeñas que sean más
fáciles de mantener.**





Como explica Link Clark en [este artículo](#). Los módulos nos permiten abstraer lógica, y comunicarla con otros módulos (sin usar el scope global).





Los módulos introducen su propio Scope, el llamado Module Scope.





Un módulo de JavaScript se distingue de un archivo de JavaScript tradicional porque o bien, exporta código o lo importa. Ambos archivos, el que importa y el que exporta son considerados módulos.





Veámoslo en práctica:

- Módulos
- Para esta práctica usaremos [stackblitz](https://stackblitz.com)





Repaso de otras características del lenguaje





Veámoslo en práctica:

- Template literals
- Destructuring
- Spread syntax
- Operator rest
- Optional chaining





códigofacilito



Programación funcional

Uriel - CTO de Código Facilito





Programación funcional





- >_ Introducción
- >_ Lenguajes imperativos y declarativos
- >_ Programación funcional
- >_ Efectos secundarios
- >_ Funciones puras





- >_ High Order Functions
- >_ Map, Filter, Reduce, ForEach
- >_ Inmutabilidad
- >_ Ventajas y desventajas





**La programación funcional
es un paradigma de
programación basado en
el uso de funciones.**





Lenguajes imperativos vs declarativos



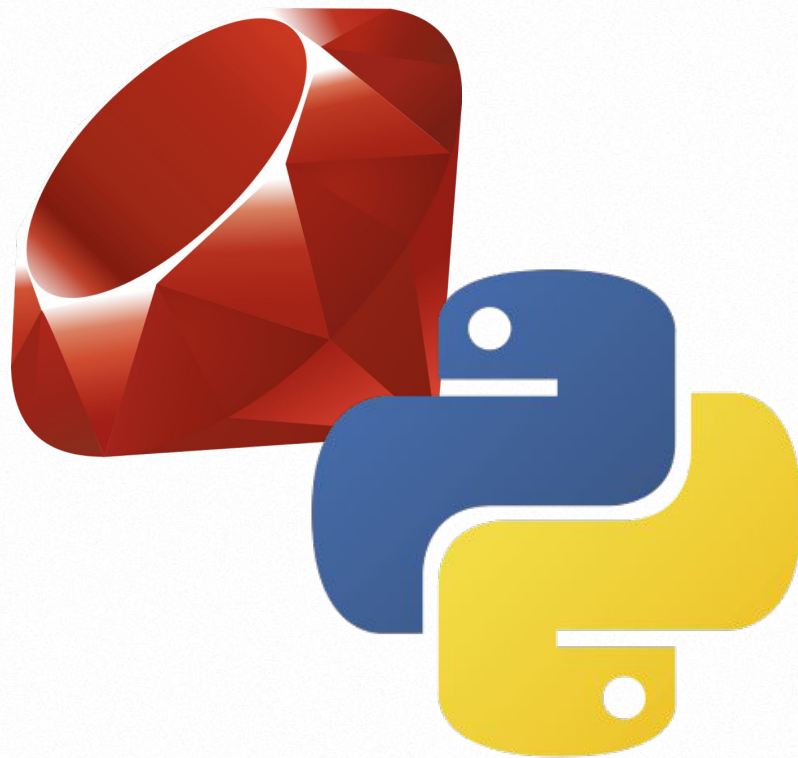


**Podemos dividir los
lenguajes de
programación en dos
grandes grupos: los
imperativos y los
declarativos.**





Los imperativos son aquellos que necesitan recibir **instrucciones específicas** sobre qué tienen que hacer, hacia qué parte del código moverse, cuántas repeticiones hacer, para poder solucionar un problema.





Los declarativos por su parte, se trata de decir **qué quiero**, sin preocuparte por los detalles, en este tipo de lenguajes debe existir un intérprete que tome estas decisiones y las ejecute.





**Un lenguaje puede
soportar distintos
paradigmas.**





>_



```
let numeros = [1,2,3,4];  
  
let cuadrados = [];  
  
for(let i = 0; i < numeros.length; i++){  
  let numero = numeros[i];  
  cuadrados.push(numero * numero);  
}  
  
console.log(cuadrados);
```





>_



```
let numeros = [1,2,3,4];  
let cuadrados = numeros.map(numero => numero * numero);  
console.log(cuadrados);
```





```
let numeros = [1,2,3,4];  
  
let cuadrados = [];  
  
for(let i = 0; i < numeros.length; i++){  
  let numero = numeros[i];  
  cuadrados.push(numero * numero);  
}  
  
console.log(cuadrados);
```



```
let numeros = [1,2,3,4];  
  
let cuadrados = numeros.map(  
  numero => numero * numero  
);  
  
console.log(cuadrados);
```





Qué es la programación funcional





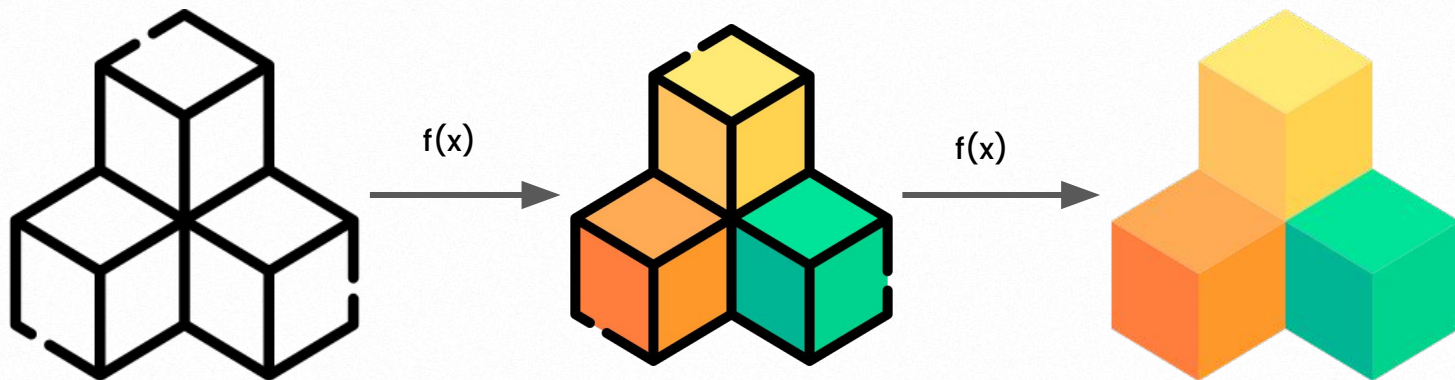
Aunque existen detalles matemáticos alrededor del trabajo en programación funcional, en este clase abordaremos el concepto con una perspectiva práctica.





Escribir un programa en programación funcional es similar a definir una rutina con distintos pasos

Donde cada paso es una función



También conocido como pipelining.





Para lograrlo aplican algunas reglas

- **Funciones de responsabilidad única (una sola tarea)**
- **Funciones puras (no producen efectos secundarios)**





Efectos secundarios





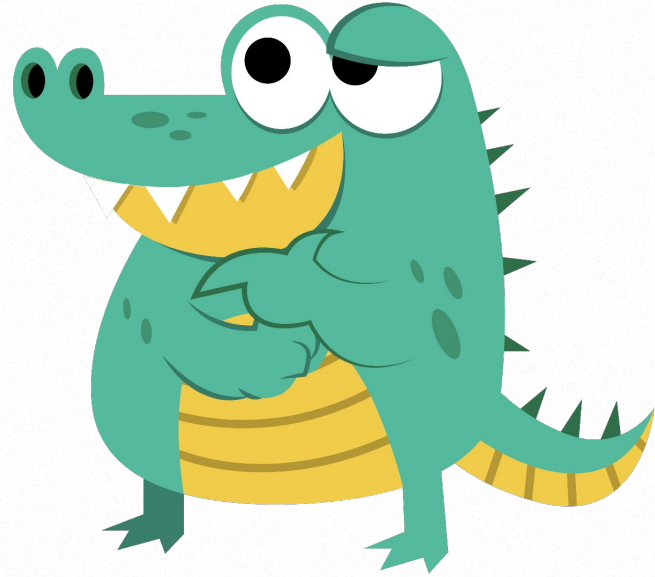
>_

Decimos que una función produce efectos secundarios cuando interactúa con elementos que están fuera del cuerpo de la función.





Una función debe recibir la información que necesita vía parámetros, y comunicar resultados vía el retorno





Algunos ejemplos de efectos secundarios son:

- **Usar una variable global**
- **Modificar una variable referenciada**
- **Imprimir logs**
- **Llamar a una base de datos**
- **Hacer una llamada a una API**
- **Modificar algo en pantalla**
- **Escribir en un archivo**
- **Etc**





Funciones puras





Decimos que una función
es pura si es **determinista**
y si no produce **efectos**
secundarios.





>_

Decimos que algo es determinista cuando podemos determinar el resultado que tendrá una operación.





¿Qué hace a esta función impura?

```
let numeroUno = 20;
let numeroDos = 10;

function suma(){
  console.log(numeroUno + numeroDos);
}
```





Para mí:

- **Usar variables globales**
- **Imprimir en consola**





**Esta función no es
determinista**



```
let numeroUno = 20;~  
let numeroDos = 10;~  
~  
function suma(){~  
  console.log(numeroUno + numeroDos);~  
}~
```



 Veamos otro ejemplo

- Funciones impuras con efectos secundarios





¿Es esta función pura?



```
async function getReposName(){  
  let repos = await (  
    await fetch("https://api.github.com/users/codigofacilito/repos")  
  ).json();  
  for(let i = 0; i < repos.length; i++){  
    console.log(repos[i].name);  
  }  
}  
  
getReposName();
```





Para mí: NO

- **Tiene una llamada a una API externa**
- **Imprime en consola**





No es determinista porque

- **Puede fallar si la API no está disponible.**
- **Entrega diferentes resultados según la respuesta de la API cambie**
- **Puede fallar si los logs no están disponibles o si algo falla al escribir en los logs**





Antes de arreglarla, abordemos algo muy importante:





Funciones puras

Siempre existen efectos secundarios





>_

Un programa que no produce efectos secundarios, es un programa que no hace nada.





¿Cómo podemos mejorar nuestra función? Sin eliminar la llamada a la API

```
async function getReposName(){
  let repos = await (
    await fetch("https://api.github.com/users/codigofacilito/repos")
  ).json();
  for(let i = 0; i < repos.length; i++){
    console.log(repos[i].name);
  }
}

getReposName();
```



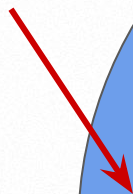


En un programa diseñado de manera funcional, buscamos tener un core de funciones puras con la lógica de negocio, y en el exterior las tareas que finalmente producen efectos secundarios

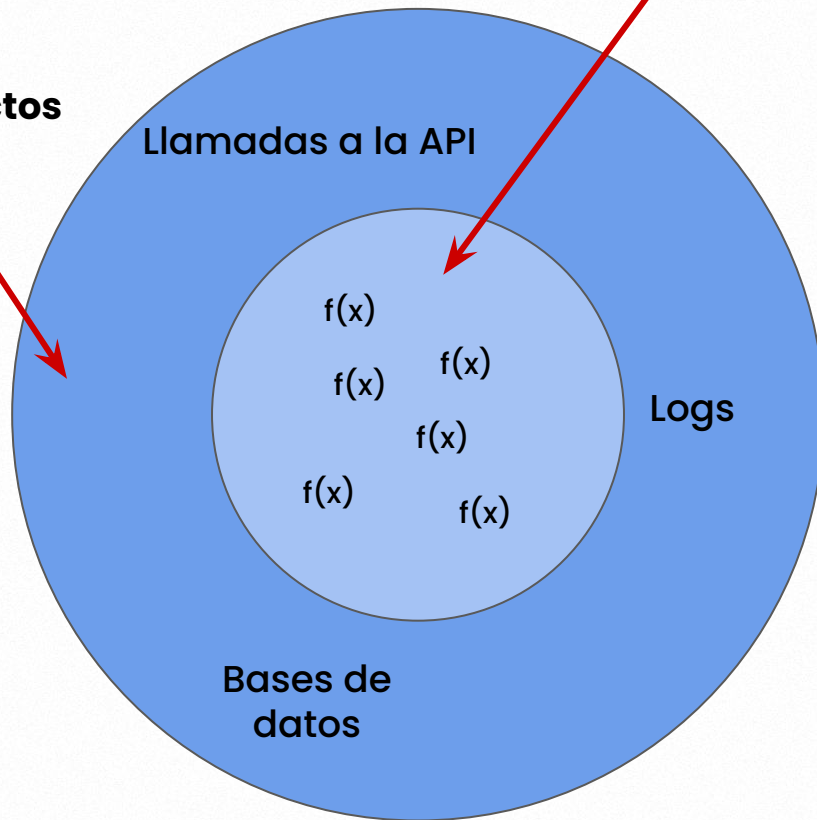
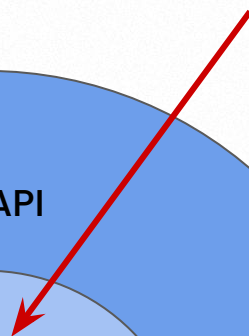




Intérprete (efectos secundarios)



Lógica de negocio





>_

Considera un problema en el que:

- **Debes extraer valores de un excel**
- **Y ejecutar un cálculo sobre estos valores.**





Mi función:

- 1. Abrir excel**
- 2. Leer los valores**
- 3. Guardar los valores en un arreglo**
- 4. Ejecutar cálculo sobre el arreglo.**

**¿Cómo nos aseguramos
que esto funciona bien?**





Sin una separación, tendríamos que preparar un entorno de pruebas que simule el trabajo con excel, para isolar el cálculo y testearlo.

¿Suena complejo?

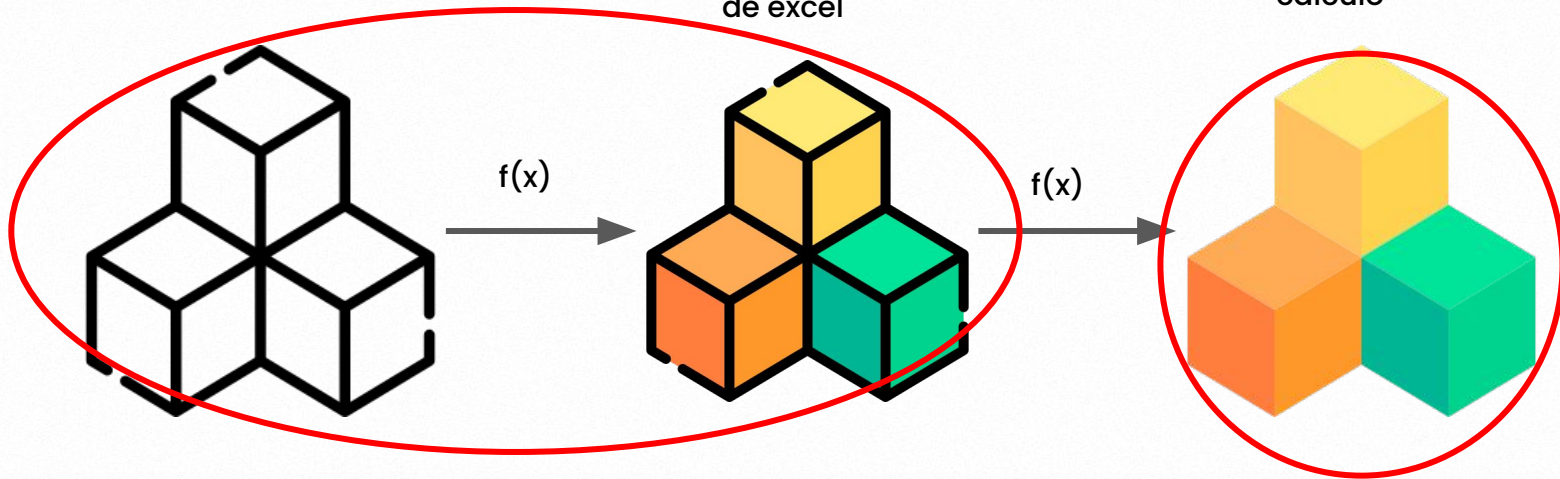




Abrir excel

Leer datos
de excel

Hacer el
cálculo



**No importa que estas no
sean funciones puras.**

**Cuando separamos es
más fácil validar
funcionalidad**





Podemos decir que:

En un programa de programación funcional tenemos una serie de funciones puras que manejan la lógica de negocio, y luego una capa que usa estas funciones y que produce efectos secundarios.



En práctica

- Solucionemos el ejercicio anterior.





```
function getReposName(repos){  
  return repos.map(repo => repo.name);  
}  
  
(async function(){  
  let endpoint = "https://api.github.com/users/codigofacilito/repos"  
  let repos = await (  
    await fetch(endpoint)  
  ).json();  
  
  let names = getReposName(repos);  
  
  for(let i = 0; i < names.length; i++){  
    console.log(names[i]);  
  }  
})();
```





>_

No nos podemos deshacer del llamado a la API o de la impresión en consola, pero los separamos de una función que, además de ahora ser pura, también es más simple y cumple una única responsabilidad.





High Order Functions





Llamamos High Order Functions a las funciones que reciben una función como argumento o retornan una función como resultado.



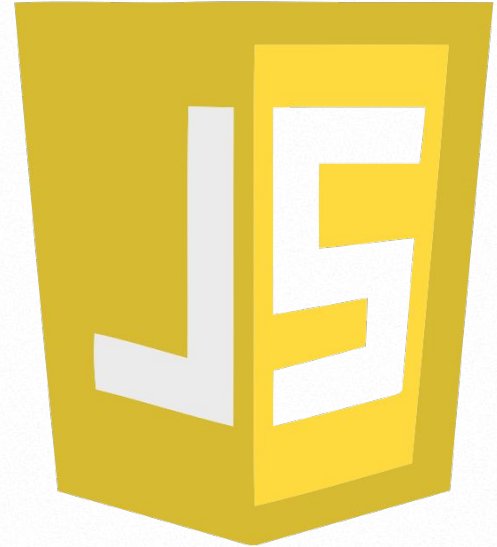


En JavaScript las funciones son objetos de primera clase.

Puede ser usadas como argumentos, retorno o almacenadas en variables.



JavaScript





```
let numeros = [1,2,3,4];  
  
let cuadrados = [];  
  
for(let i = 0; i < numeros.length; i++){  
  let numero = numeros[i];  
  cuadrados.push(numero * numero);  
}  
  
console.log(cuadrados);
```



```
let numeros = [1,2,3,4];  
  
let cuadrados = numeros.map(  
  numero => numero * numero  
);  
  
console.log(cuadrados);
```

Este es un ejemplo de
HOF



```
// fetchById is the "thunk action creator"~
export function fetchById(todoId) {~
  // fetchByIdThunk is the "thunk function"~
  return async function fetchByIdThunk(dispatch, getState) {~
    const response = await client.get(`/fakeApi/todo/${todoId}`)~
    dispatch(todosLoaded(response.todos))~
  }~
}
```

**Otro ejemplo de HOF
(retornar funciones)**





High Order Functions

**forEach, map, filter,
reduce**



Veámoslo en práctica

- `forEach`
- `map`
- `filter`
- `reduce`





High Order Functions

Por qué usamos HOF en lugar de ciclos






Las ventajas en el uso de estas funciones sobre iteraciones o ciclos, pueden ser discutidas.

Una de ellas es composición.



 Veámoslo en práctica





**No es mal código usar
ciclos, sobre funciones.
Son dos maneras de
pensar y de abordar un
problema.**



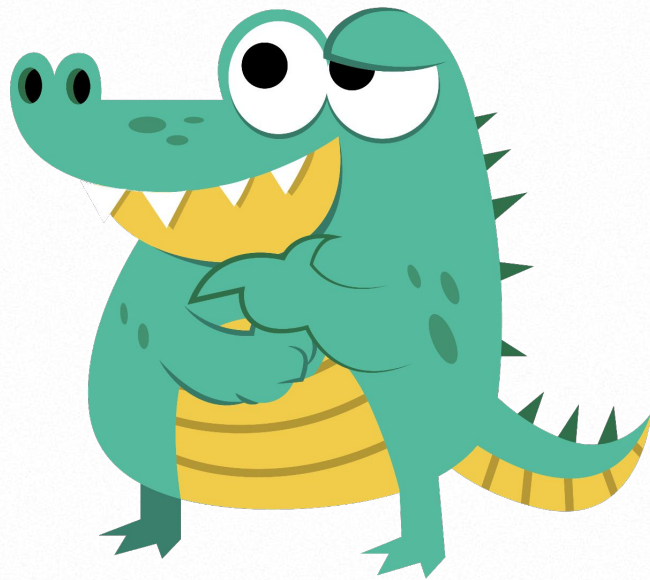


Inmutabilidad





Inmutabilidad es el concepto con que nos referimos a un objeto que no puede modificar su valor.





**¿Sabías que antes ya has
trabajado con objetos
inmutables?**





**Los tipos primitivos como
cadenas y números en
JavaScript son inmutables.**





**Antes de verlo en práctica
recuerda:**

**En un sistema inmutable,
los valores pueden ser
reemplazados, pero la
estructura no debe ser
modificada.**





Veámoslo en práctica:

- Primitivos inmutables
- Estructuras mutables
- Cómo hacer cambios respetando inmutabilidad.





Ventajas y desventajas de la programación funcional.





Ventajas:

- **Complejidad del código:**
Dividimos el problema en soluciones pequeñas que son más fáciles de mantener





Ventajas:

- **Pruebas unitarias:**

Una función pura es más fácil de testear





Ventajas:

- **División:**

Cada función está aislada, por lo que es más fácil modificarla o reemplazarla sin alterar el sistema





Ventajas:

- **Expresividad del código:**

El código que resulta de un enfoque funcional puede ser en muchas ocasiones más legible y expresivo





Ventajas:

- **Menos bugs o bugs más fáciles de encontrar:**

Aplicando conceptos como el de inmutabilidad, nos aseguramos que las operaciones que realiza una función no terminen por afectar las de otra función

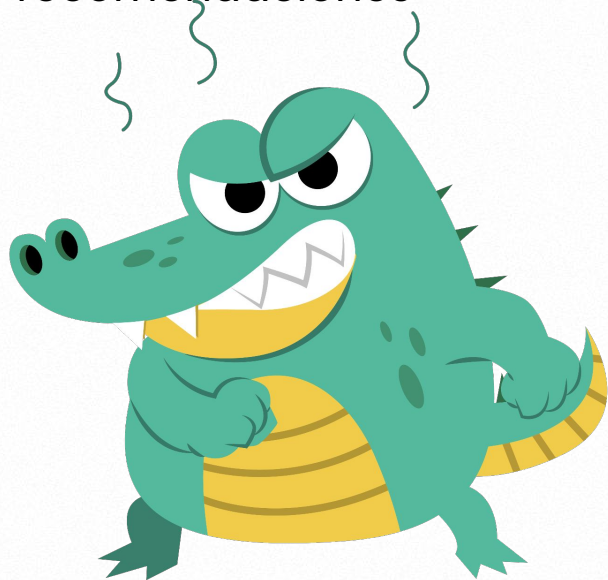




Ventajas:

- **Overhead al organizar el código**

En muchas ocasiones, tratar de respetar las recomendaciones del paradigma terminará por hacernos pensar dos veces dónde colocar nuestros elementos





Ventajas:

- **Manejo de estructuras grandes**
Copiar toda una estructura para no modificarla puede ser costoso en términos de recursos





Ventajas:

- **State management**

A diferencia de otros paradigmas como el de objetos, donde el manejo de estado es parte del paradigma y de los objetos, en programación funcional es necesario implementar alguna estrategia





Fin.

