Programación Funcional

DEV.F::
DESARROLLAMOS(PERSONAS);

¿Qué es?

- **Es un paradigma de programación.**
- Es una forma de construir software a partir de solamente funciones.
- La programación funcional es declarativa.
- Tiende a ser más concisa, más predecible y más fácil de leer.
- Se compone de funciones puras (no utilizas librerías, funciones o procedimientos externos)

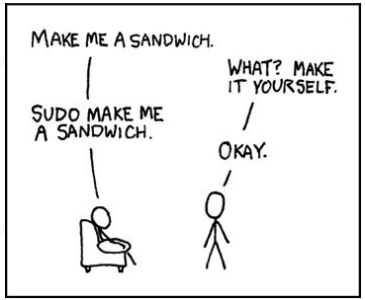
Data in =>

Data out

Programación Declarativa e Imperativa

- **Declarativa:** Declaras una base de conocimiento, restricciones o afirmaciones para resolver el problema pero no dices como llegar a la solución.

  ¿Qué estamos haciendo? No cómo lo hacemos



- **Imperativa:** Declaras un algoritmo y una serie de instrucciones o pasos para llegar a una solución.

  Implementar algoritmo.

Lenguajes de programación funcionales

Funcionales

Haskell

Elm

Ocaml

Erlang

Con características funcionales

C#

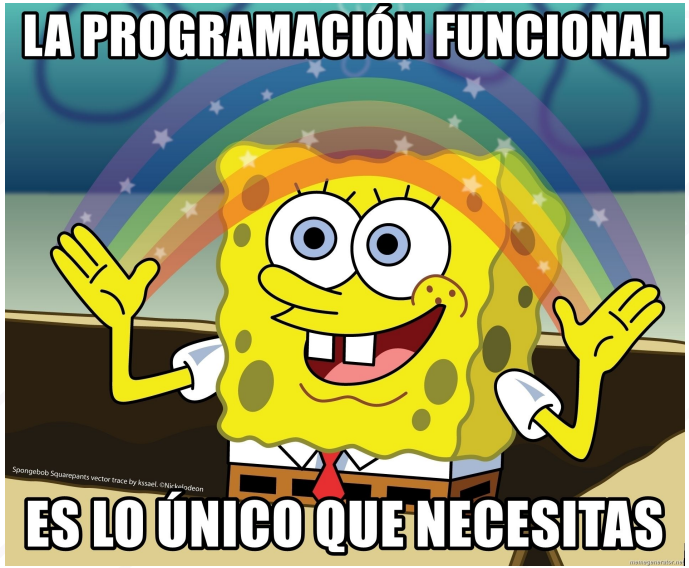
Java

Kotlin

Ruby

Conceptos comunes

- Pure functions
- Avoid side effects
- Function composition
- Avoid shared state
- Avoid mutating state



Pure functions

No tiene efectos de estado que la afecten, lo que quiere decir que siempre que le pasemos un número A este dará como resultado un número B, como ejemplo si le damos como número un 3 este nos arrojará como número un 4.

```
function sumaUnoAlNumero(numero) {
    return numero + 1;
}
```

Avoid side effects

Tiene efectos de estados porque un agente externo como es Math.random() está afectando el la fiabilidad con la que nos dará los resultados la función, en el sentido de que el número A que le demos no necesariamente nos dará siempre un mismo resultado B.

```
function sumaNumeroRandom(numero) {
    return numero + Math.random()
}
```

DEV.F

Function composition / Callbacks

Hablamos de composición de funciones cuando combinamos funciones para producir nuevas funciones de mayor complejidad.

Es una función que recibe como parámetro otra función.

La función “callback” por lo regular va a realizar algo con los resultados de la función que la está ejecutando.

```
function saludar(nombre) {
    alert('Hola ' + nombre);
}

function procesarEntradaUsuario(callback) {
    var nombre = prompt('Por favor ingresa tu nombre.');
    callback(nombre);
}

procesarEntradaUsuario(saludar);
```

DEV.F

Avoid shared state

Cuando hablamos de estado compartido nos referimos a cualquier variable, objeto o espacio de memoria que exista en un ámbito compartido.

Es preferible hacer una copia del elemento al que se desea acceder.

```
function copiaEmpleados(empleados){
    let newEmpleados = new Array();
    empleados.forEach(empleado => newEmpleados.push([...empleado]));

    return newEmpleados

}

function cambiarSalario(empleados,cantidad){
    let copEmpleados = copiaEmpleados(empleados)
    copEmpleados.forEach((empleado) => {
        empleado[1] = empleado[1] + cantidad;
    })

    return copEmpleados
}
```

Avoid mutating state

Un objeto inmutable es un objeto que no se puede modificar una vez creado.

Por el contrario, un objeto mutable es cualquier objeto que se puede modificar después de su creación.

```
const a = Object.freeze({
  foo: 'Hello',
  bar: 'world',
  baz: '!'
});

a.foo = 'Goodbye';
// Error: Cannot assign to read only property 'foo' of object Object
```

POO vs FP

No hay mejor o peor elección todo depende de la complejidad del problema y que se necesita hacer

- Dice que los datos (atributos) y las acciones (métodos) deben estar juntas en un solo “Objeto”.
- Dice que la Herencia y encapsulación hacen más fácil la forma de abstraer y asegurar el codigo
- Piensa en re-usabilidad a través de la herencia

- Dice que los datos y las acciones deben estar separadas ya que son distintivamente diferentes.
- Dice que separar métodos de atributos es mucho mejor ya que evita menores errores de lógica ya que se necesita un mayor nivel de abstracción
- Piensa en re-usabilidad a través de funciones pequeñas

Ejemplo

Digamos que dirigimos una empresa y acabamos de decidir otorgarle a todos los empleados un aumento de $ 10,000.00.

¿Cómo podríamos escribir un script para hacer este cambio?

Solucion con POO

```
class Empleado {
    constructor(nombre,salario){
        this.nombre = nombre;
        this.salario = salario;
    }
    cambiarSalario(aumento){
        this.salario = this.salario + aumento
    }
    descripcion(){
       return `El trabajador: ${this.nombre} gana ${this.salario}`
    }


}
```

Solucion con POO

```
var empleados  = [
    Empleado("Luis Torres",25000),
    Empleado("Maria Arriaga",42000)
]
```

Solucion con POO

```
empleados.forEach((empleado) =>{
    empleado.cambiarSalario(10000)
});
```

Solución con FP

```
var empleados = [
    ["Luis Torres",25000],
    ["Maria Arriaga",42000]
]
```

Solución con FP

```
function copiaEmpleados(empleados){
    let newEmpleados = new Array();
    empleados.forEach(empleado => newEmpleados.push([...empleado]));

    return newEmpleados

}

function cambiarSalario(empleados,cantidad){
    let copEmpleados = copiaEmpleados(empleados)
    copEmpleados.forEach((empleado) => {
        empleado[1] = empleado[1] + cantidad;
    })

    return copEmpleados
}
```

Solución con FP

```
var empleadoFelices = cambiarSalario(empleados,10000)

empleadoFelices.forEach((empleado) =>{

        console.log(`El empleado: ${empleado[0]} gana ${empleado[1]}`)
})
```

Funciones y Métodos de los Arrays

filter()

pop()

find()

push()

sort()

map()

forEach()

splice()

reduce()

...

forEach()

Itera un array con los resultados de la llamada a la función indicada aplicados a cada uno de sus elementos.

No genera un nuevo array.

```
//              0   1   2   3   4
const array = [10, 22, 33, 54, 12];

// Iterar arreglo => Manera imperativa
// for (let i = 0; i < array.length; i++) {
//   console.log(array[i]);
// }

// Declarativa
const eachArray = array.forEach((value) => console.log(value));
console.log(eachArray);
```

map()

Crea un nuevo array con los resultados de la llamada a la función indicada aplicados a cada uno de sus elementos.

```
var numbers = [1, 5, 10, 15];
var doubles = numbers.map(function(elem,index,arr) {
   return elem * 2;
});
// doubles is now [2, 10, 20, 30]
// numbers is still [1, 5, 10, 15]

var numbers2 = [1, 4, 9];
var roots = numbers2.map(Math.sqrt);
// roots is now [1, 2, 3]
// numbers is still [1, 4, 9]
```

filter()

Crea un nuevo array con todos los elementos que cumplan la condición implementada por la función dada.

```
var words = ['spray', 'limit', 'elite', 'exuberant',
             'destruction', 'present'];

const result = words.filter(word => word.length > 6);
const result2 = words.filter(function(elem,index,arr){
 return elem.length > 6;
})
var result3 = words.filter(word => word === 'elite');

console.log(result);
// expected output: Array ["exuberant", "destruction", "present"]
```

sort()

El método **sort()** ordena los elementos de un arreglo localmente y devuelve el arreglo ordenado. El modo de ordenación por defecto responde a la posición del valor del string de acuerdo a su valor Unicode.

```
var frutas = ['guindas', 'manzanas', 'bananas'];
frutas.sort(); // ['bananas', 'guindas', 'manzanas']

var puntos = [1, 10, 2, 21];
puntos.sort(); // [1, 10, 2, 21]
// Tenga en cuenta que 10 viene antes que 2
// porque '10' viene antes que '2' según la posición del valor Unicode.

var cosas = ['word', 'Word', '1 Word', '2 Words'];
cosas.sort(); // ['1 Word', '2 Words', 'Word', 'word']
// En Unicode, los números vienen antes que las letras mayúsculas
// y estas vienen antes que las letras minúsculas.
```

reduce()

El método reduce() ejecuta una función reductora sobre cada elemento de un array, devolviendo como resultado **un único valor.**

```
const array1 = [1, 2, 3, 4];
const reducer = (accumulator, currentValue) => accumulator + currentValue;

// 1 + 2 + 3 + 4
console.log(array1.reduce(reducer));
// expected output: 10

// 5 + 1 + 2 + 3 + 4
console.log(array1.reduce(reducer, 5));
// expected output: 15
```