**Introducción al paradigma de programación**

**¿Qué es un “paradigma de programación?**

Un paradigma es como una forma de pensar y organizar las ideas cuando programamos.

Imaginen cuando quiera volver de la universidad a sus casas. De regreso uno podría tomar diferentes caminos, utilizar diferentes estrategias (elijo la ruta más corta, o con menor tránsito, etc. )y los medios (si voy caminando, me tomo un colectivo, uber/Didi, etc.), pero todos van a permitirme llegar al destino.

Bueno, los paradigmas de programación son diferentes formas de pensar un problema pero que todos llegan a la solución. ¿Cuál es mejor o peor?, eso depende del problema a solucionar dado que muchas veces se utilizan **múltiples paradigmas** para poder llegar a una solución. Entonces no debemos pensar en los algoritmos de programación como una competencia entre sí, sino más bien como estrategias que pueden ser combinadas para llegar a la resolución de un problema mediante código fuente.

Son varios los paradigmas de programación que podemos encontrar, entre ellos podemos nombrar a los:

**Imperativo vs declarativo**

* **Imperativo**: Aquí se piensa el ¿Cómo hacer tal cosa?
* **Declarativo**: En el declarativo se piensa ¿Qué es lo quiero?, es decir se piensa en el resultado.

**Otras formas de pensar…**

* **Funcional**: Acá pensamos en funciones puras y composición de forma de evitar efectos colaterales, es decir evitar afectar a otras secciones del código.
* **Orientado a objetos**: se piensa en modelos con estado y comportamiento (objetos que colaboran) y que mutan.
* **Lógico/Reglas**: se piensa en hechos y reglas y luego un motor deduce. Es una de las bases clásicas de la inteligencia artificial. Un ejemplo re una regla seria “SI fiebre Y tos ENTONCES gripe”

Si llegaste hasta aquí quizás te preguntes del ¿Por qué separe Imperativo y Declarativo del resto, funcional, orientado a objeto… ?, Bueno porque lo Imperativo y Declarativo es independiente del paradigma. Con esto quiero decir que en cualquier paradigma que elijas funcional, orientado a objeto podes escribir partes del código de forma imperativa y otras en declarativas.

En esta asignatura nosotros nos vamos a centrar en el paradigma funcional y orientado a objetos. Pero les recomiendo que investiguen por su cuenta las nuevas metodologías de pensar las soluciones a los problemas, por ejemplo el orientado a aspecto y otros que aún están en laboratorio como los relacionados con la computación cuántica o la programación neuromórfica, que intenta inspirarse en la estructura del cerebro para crear nuevos paradigma. Si bien este es un ámbito que se sigue explorando, lo interesante de todo esto es que podría revolucionar del como pensamos en algoritmos y paradigmas de programación.

**Entonces estas primeras clases veremos el Paradigma funcional**

La programación funcional ha ganado bastante popularidad en los últimos años, sobre todo porque facilita escribir código más conciso y fácil de probar, especialmente en entornos como el desarrollo web moderno. Así que hoy en día muchos equipos combinan lo mejor de ambos mundos: objetos y un poco de funcional. Javascript es un lenguaje que permite combinar múltiples paradigmas y en firmwares modernos como Angular, Ionic se puede apreciar el paradigma funcional como parte de la estrategia de mejora en los procesos de construcción de softwares.

En definitiva, no hay una sola reina del baile, pero podríamos decir que la programación orientada a objetos sigue siendo la más usada, la funcional le pisa los talones.

Entonces para que nos quede súper claro!, los paradigmas son como tener un equipo de superhéroes donde cada uno tiene su poder especial y se van turnando según la misión.

Entonces, manos a la obra. Debemos tener presente que:

Analicemos la siguiente definición de una función. Esta se encuentra de forma **Imperativa** porque se describe paso a paso lo que se tiene que hacer.

**let suma = 0;**

**for (let i = 1; i <= 5; i++) {**

**suma = suma + i;**

**}**

**console.log(suma); // 15**

En cambio, la siguiente función se encuentra desarrollada de forma **Declarativa** porque se ocultan los paso a paso, es decir que decimos qué queremos: “reducí la lista sumando”, **no** detallamos los pasos internos del recorrido sino que en su lugar (el método reduce lo hace por nosotros).

**const numeros = [1,2,3,4,5];**

**const suma = numeros.reduce((acc, n) => acc + n, 0);**

**console.log(suma); // 15**

Ambas definiciones son válidas y llegan al mismo resultado.

Para el transcurso del desarrollo de las prácticas, vamos a usar en lo posible, la forma “declarativa” porque es una de las formas que más se utilizan en los lenguajes nuevos de programación.

**Funciones de primera clase y de orden superior**

**Funciones**

Una **función** es como una **receta** o un **mecanismo** dentro de un programa:

* Recibe ciertos **ingredientes** (llamados parámetros o argumentos).
* Realiza un **proceso** (una serie de pasos definidos).
* Devuelve un **resultado** (un valor) o ejecuta una **acción**.

En otras palabras, una función es un **bloque de código reutilizable** que nos evita repetir las mismas instrucciones cada vez que queremos hacer lo mismo.

**Funciones de primera clase**

En Javascript, estas funciones se tratan como valores, podés guardarlas en variables, pasarlas como argumentos, devolverlas como resultado y guardarlas en estructuras (arrays/objetos).

Unas de las ventajas que nos ofrece el lenguaje es que nos permite escribir la funciones de formas compacta al estilo de:

**() => { … }**

Acá vamos a tener dos partes fundamentales de una función superior o de primer orden (ciudadanas). La primera parte son los “**( )**” que son la parte parametrizable de la función, es decir que es la que puede o no recibir argumentos, por ejemplo () o (a,b), (objeto), (unaFuncion), etc. La segunda es **=> {}** que corresponde al cuerpo de la función, es decir donde se especificará lo que se debe hacer.

* Parámetros: (...) → pueden estar vacíos (), uno x, o varios (a, b).
* Cuerpo: => { ... } → bloque con llaves. Si usas llaves, para devolver algo poné return.
* Forma compacta (return implícito): a => a + 1
* Devolver objeto literal: x => ({ valor: x })

Entonces en vez de tener una definición convencional como escribir:

**// Declaración tradicional**

**function sumar(a, b) {**

**return a + b;**

**}**

Si utilizando el anonimato podemos hacer lo equivalente a:

**// compacta (return explícito)**

**const sumar = (a, b) => { return a + b; };**

Si bien, a primera vista no parece ser un cambio importante, cuando veamos ejemplos más complejos daremos cuenta del potencial de poder expresar estas funciones de esta manera.

**Funciones Ciudadanas**

**Aquí es donde aparece el concepto de funciones Ciudadanas donde el término “Ciudadanas” se refiere a todas las funciones en lenguajes donde son tratadas como valores comunes. Esto quiere decir que para el lenguaje Javascript, todas las funciones son ciudadanas porque se pueden tratar como valores comunes.**

Si observas el resultado de la función “ **(a, b) => { return a + b; };”** será tratado como un valor común que terminará siendo asignado a la variable “sumar” .

Veamos otra forma de escribir la función y que sea más compacta es sin utilizar las llaves “{}”. Cuando se usan las llaves debemos obligatoriamente utilizar la palabra reservada “return”, de lo contrario el programa no funcionara. Sin embargo, si escribimos la función anónima sin las llaves, el return ya vienen explicito con lo cual quedaría de la siguiente forma:

**// compacta (return implícito)**

**const sumar = (a,b) => a + b;**

Tómate un tiempito para analizar las tres definiciones y te darás cuenta que son tres formas de contar la misma película, pero escribir funciones anónimas de la forma (a,b) => a + b; aumenta el potencial de la estructura del lenguaje.

**Funciones de orden superior**

Una función de orden superior es una que recibe funciones o devuelve funciones.

Son la caja de herramientas del estilo declarativo. En javascript esto se lleva a puede realizar definiendo a la función como se muestra en el ejemplo:

Esta es una típica función de orden superior:

const aplicar = (fn, x) => fn(x); // orden superior

aplicar(x => x \* 2, 5);

*// 1° parámetro x => x \* 2*

*// 2° parámetro el 5*

// da como resultado 10

Como se puede notar, ocurren dor características fundamentales que describen la naturaleza de una función de orden superior (**regla de oro**):

* **La función puede o no recibe como parámetro a otra función.**
* **La función retorna una función.**

Con que cumpla una de esas dos, ya es de orden superior. ¿Y si cumple ambas?, también lo es.

Veamos unos casos más!

// NO es de orden superior porque (no recibe ni devuelve funciones)

sumar(a, b) { return a + b; }

// Si lo es porque RECIBE una función

const aplicar = (fn, x) => fn(x);

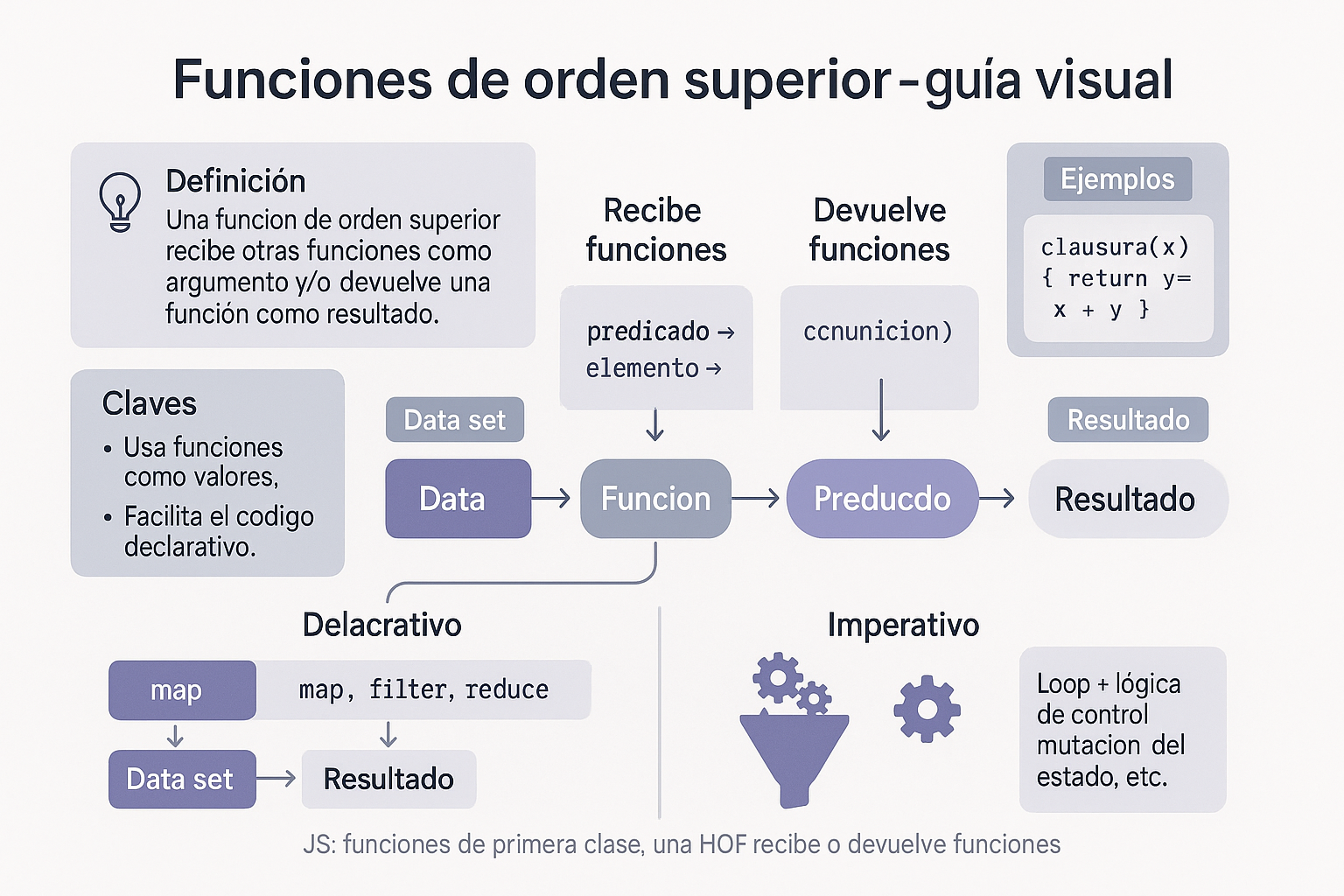
// Si lo es porque DEVUELVE una función

const mayorQue = (n) => (x) => x > n;

// Si lo es por ambas razones

const comp = (f, g) => (x) => f(g(x));

En la siguiente imagen pueden tener un abstracción de los conceptos ya explicado en este documento donde se pueden apreciar las condiciones necesarias para que un función sea de orden superior (HOF) o como se la conoce en ingles Higher-Order Function.



https://www.freecodecamp.org/espanol/news/funciones-de-orden-superior-en-javascript-guia-para-principiantes/