

## 2.- JavaScript : FUNCIONES

Creando una función de forma tradicional

```
function f1(a, b) {  
    return a + b;  
}  
console.log("f1 =", f1(1, 2));
```

Creando una función asignándola a una variable, en js una variable y una función es lo mismo y se pueden pasar como parámetros FUNCIONES y devolver funciones

```
const f2 = function (a, b) {  
    return a + b;  
};  
console.log("f2 =", f2(1, 2));
```

Parámetros por defecto en una función

```
function multiply(a, b = 1) {  
    return a * b;  
}  
  
console.log(multiply(5, 2));  
// Expected output: 10  
  
console.log(multiply(5));  
// Expected output: 5
```

Función flecha

```
const f3 = (a, b) => {  
    return a + b;  
};  
  
console.log("f3 =", f3(1, 2));
```

## 2.- JavaScript : FUNCIONES

Función flecha simplificada, return implícito

```
const f4 = (a, b) => a + b;  
  
console.log("f4 =", f4(1, 2));
```

Función a la que le paso otra función como parámetro

```
const f5 = (f) => {  
    return f(1, 2);  
};  
console.log("f5 =", f5(f4));
```

Lo mismo pero de forma algo más clara

```
function f6(mifuncion) {  
    return mifuncion(1, 2);  
}  
console.log("f6 =", f6(f4));
```

Operador ternario ?

Condición ? valor si verdad : valor si falso

```
function max(a, b) {  
    return a > b ? a : b;  
}  
console.log("max =", max(3, 4));
```

## EJERCICIOS

1. Crea una función que calcule el máximo de dos números, debe llamarse max.
2. Haz lo mismo pero con la función flecha
3. Crea una función que calcule el máximo de hasta 4 números, pero debe funcionar si le faltan parámetros, como ayuda los parámetros van a ser número > 0 y si no le pasas ningún parámetro debe devolver 0. (Usa parámetros por defecto)
4. Haz una función que le pases los valores de A,B y C de una ecuación de segundo grado

## 2.- JavaScript : FUNCIONES

$ax^2 + bx + c = 0$  y devuelva el resultado y en el caso de que no tenga solución devuelva false.

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

Recuerda que no se puede dividir entre 0 ni se puede hacer la raíz cuadrada de un número negativo.

5. Haz una copia de la función anterior pero ahora no puedes usar el IF, solo el operador ternario.
- 6.

```
Dada la función function malEcu(a, b) {  
  
    return a / b;  
}  
console.log(malEcu(6, 3));  
console.log(malEcu(6, 0)); //falla
```

Haz una función llamada bienEcu a la cual le pasamos como parámetro una función, en este caso malEcu y debe devolver el resultado de la ecuación si esta funciona correctamente, y en el caso que la función devuelva NaN o Infinity debe devolver **false**.

## AMPLIACIÓN : PROGRAMACIÓN FUNCIONAL

Aunque un poco más adelante lo vamos a volver a ver, aquí dejo un poquito de teoría y algunos ejercicios:

La programación funcional es un paradigma de programación que ha ganado popularidad en los últimos años, especialmente con la adopción de JavaScript en la industria. Este enfoque de programación se centra en la evaluación declarativa, el uso de funciones puras y la inmutabilidad, entre otros principios. JavaScript, aunque es un lenguaje **multiparadigma**, ha incorporado muchas características que permiten programar de manera funcional.

La programación funcional se distingue por las siguientes características:

- **Evaluación declarativa vs. imperativa:** En lugar de describir cómo hacer algo paso a paso (imperativa), se describe qué hacer (declarativa).

## 2.- JavaScript : FUNCIONES

- **Funciones puras:** Funciones que no tienen efectos secundarios y siempre producen el mismo resultado dado el mismo conjunto de argumentos.
- **Inmutabilidad:** Evitar que una parte del programa modifique datos de forma inconsistente. Por ejemplo no mutar arrays (lo veremos más en la siguiente práctica) o modificar valores de variables globales.
- **Sin efectos secundarios** (side effects): Las funciones no deben alterar el estado del programa.

Es un paradigma, una manera de pensar y escribir código que busca reducir la programación imperativa y la programación orientada a objetos (POO). Es uno de los más usados hoy en día y por ejemplo el recomendado por REACT “lo dice Facebook 😊”

### MUTABILIDAD

```
((()=>{  
let foo = [1, 2, 3];  
let bar = foo;  
bar.push(10000);  
  
console.log(foo);  
// Salida: [1, 2, 3, 10000]  
}))();
```

Como vemos hemos creado la función usando la función flecha, y todo está envuelto también entre paréntesis para poder ejecutar directamente la función con los últimos dos ( )

Vamos a la mutabilidad, cuando igualamos foo y bar lo que hacemos es que **bar** apunta al espacio de memoria de **foo** pero no se hace una copia de **foo** con lo cual cuando modificamos **bar** estamos modificando sin saberlo o sabiéndolo también **foo**.

Para evitar la mutabilidad de las variables se puede:

- Usar const (sólo funciona con variables primitivas, ya que los arrays son mutables internamente)
- Reducir o eliminar las asignaciones
- Usar estructuras de datos Inmutables como tuplas o records
- Hacer copias de los objetos antes de modificarlos (... o structuredClone)
- No asignar objetos por referencia.
- Función freeze o seal
- No usar funciones que provocan mutaciones en arrays, como splice, sort, push...

Esto lo veremos más adelante.

## 2.- JavaScript : FUNCIONES

### Purificación de Funciones

En la programación funcional, todas las funciones deben cumplir ciertos requisitos:

- **Pocos argumentos:** Idealmente, una función debe tener pocos argumentos.
- **Una sola tarea:** Deben hacer solo una cosa. (Un fallo que se suele cometer es hacer funciones que calculan algo y a su vez lo visualizan)
- **Puras y deterministas:** Deben ser funciones puras, es decir, no deben tener efectos secundarios y deben producir el mismo resultado para los mismos argumentos.
- **Trabajan sobre datos inmutables**
- **Son testables**
- **Transparencia referencial:** Una función siempre producirá el mismo resultado dado los mismos argumentos, sin causar efectos secundarios, lo que permite que cualquier expresión con la función pueda ser reemplazada por su valor de resultado sin alterar el comportamiento del programa. Sustituir la llamada a una función por su resultado almacenado en “caché” se llama meonizar.

Además de estas condiciones, también es aconsejable seguir estas consideraciones:

- Si la función acepta un número indeterminado de argumentos es mejor aceptar un objeto o un array.
- También se puede aceptar un número indeterminado y poner ...args para convertirlos en un array dentro de la función.
- Más avanzado puede ser hacer getify/Functional-Light-JS
- Evitar que una función retorne más de un valor. Esto rompe el flujo y las hace más difíciles de leer.
- Usar closures también se considera efecto colateral.

### Ejemplos de Funciones Impuras

```
// Impura: modifica el entorno
console.log("Hola");

// Impura: no es determinista
Math.random();

// Impura: modifica el array como efecto colateral, ya veremos que
// splice extrae un trozo de un array mutandolo
array.splice(2, 3);
```

## 2.- JavaScript : FUNCIONES

| Condiciones positivas para la programación funcional               | Declaración de función con nombre | Declaración de función anónima | Función flecha | Expresión de función flecha        | Expresión de función con <code>function()</code> |
|--------------------------------------------------------------------|-----------------------------------|--------------------------------|----------------|------------------------------------|--------------------------------------------------|
| En caso de error, podemos encontrar la función que ha fallado      | ✓                                 | X                              | X              | ✓ No siempre tiene el mismo nombre | ✓ No siempre tiene el mismo nombre               |
| Puede autoreferenciarse                                            | ✓                                 | X                              | X              | ✓                                  | ✓                                                |
| La sintaxis es más consistente para argumentos y returns avanzados | ✓                                 | ✓                              | X              | X                                  | ✓                                                |
| Tiene hoisting                                                     | ✓                                 | X                              | X              | X                                  | X                                                |
| El código es breve                                                 | X                                 |                                | ✓              | ✓                                  | X                                                |

### Funciones de Orden Superior

**Las funciones de orden superior son aquellas que aceptan o retornan otras funciones.** En JavaScript, métodos como `map`, `filter` y `reduce` son funciones de orden superior. Estas son las más importantes y forman parte de los fundamentos de la programación funcional. Estas funciones forman parte del objeto `Array`.

Podemos hacer nuestras funciones de orden superior.

Otras funciones de los objetos `Array`, algunas no totalmente funcionales, también son de orden superior y muy útiles: `forEach`, `some`, `every`, `find`, `findIndex`, `flatMap`, `reduceRight`...

### Records y Tuplas

A partir de 2024, JavaScript ha incorporado `Records` y `Tuples`, que son estructuras de datos inmutables similares a los objetos y arrays respectivamente. Las tuplas y records se pueden comparar directamente.

Tampoco funciona en todos los navegadores.

```
const debitCard = #{
  type: 'Visa',
  number: 4242424242424242,
}
const wallet = #[drivingLicence, creditCard, debitCard]
```