Funciones

JS permite permite tratar las funciones como valores, pasarlas como parámetros o incluso retornar una función dentro de otra función. Esto le da muchas posibilidades y por eso se dice que las funciones son **first class citizens** en JS.

Funciones como variables

En el siguiente ejemplo guardarmos la función . Si en la llamada un parámetro intermedio no queremos enviarlo, habrá que hacerlo como *undefined* para que coja el valor por defecto.
Ojo, hay que definir la variable createBooking antes de usarla.

```
const bookings = [];

const createBooking = function (
  flightNum,
  numPassengers = 1,
  price = 199 * numPassengers
) {
  // ES5
  // numPassengers = numPassengers || 1;
  // price = price || 199;

  const booking = {
    flightNum,
    numPassengers,
    price,
```

```
  };
  console.log(booking);
  bookings.push(booking);
};

createBooking('LH123');
createBooking('LH123', 2, 800);
createBooking('LH123', 2);
createBooking('LH123', 5);

createBooking('LH123', undefined, 1000);
```

High order functions

Son funciones que reciben como argumento otra función o que retornan una función.
En el siguiente ejemplo addEventLister es la high order function y high5, que se pasa como parámetro, una función de callback:

```
const notificar = function () {
  console.log('Me han pulsado');
};
document.body.addEventListener('click', notificar);
```

En muchas ocasiones tiene sentido definir la función de callback dentro del propio parámetro ya que no se va a usar en otro sitio.

```
document.body.addEventListener(
        'click',
        function () {
          console.log('Me han pulsado');
```

```
        }
);
```

En ámbos casos han sido funciones anónimas, en el primer caso porque la guardarmos en una variable y en el segundo porque se pasa como parámetro el propio código de la función.

Funciones flecha

Presentan una sintáxis más corta (se eliminan normalmente los keywords: *return* y *function*). Tienen ciertas diferencias pero las pasaremos por alto.

```
// Función anónima
(function (a) {
  return a + 100;
});

// 1. Quitamos el keyword "function" y a cambio
colocamos // la flecha
(a) => {
  return a + 100;
};

// 2. Si quitamos las llaves, el return va implícito:
(a) => a + 100;

// 3. Como hay un sólo parámetro podemos quitar
también
```

```
// los paréntesis
a => a + 100;
```

Functions Returning Functions

Se usan mucho en programación funcional.

```
const tablaMultiplicar = function (num1) {
  return function (num2) {
    console.log(`${num1} x ${num2} = ${num1*num2} `);
  };
};

const tablaDel2 = tablaMultiplicar(2);
tablaDel2(1);
tablaDel2(2);
tablaDel2(3);
```

Con arrow functions:

```
const tablaMultiplicar = (num1) => (num2) =>
    console.log(`${num1} x ${num2} = ${num1*num2} `);
```

Arrays

Utilizamos la documentación de Mozilla

Usaremos programación funcional, por ser más legible, y tener JS muchos métodos que nos ayudan, especialmente en el tema de Arrays.

Vamos a sumar el siguiente Array:

const movimientos = [-500, +200, -300, 800]

```
// con for
// sería una programación imperativa,

// foreach
let total = 0
movimientos.forEach(
        (mov, i) => {
                total += mov
                console.log(total, `contador: ${i}`)
        }
);

// con for of:


total = 0
// for of
for (const mov of movimientos) {
        total += mov
        console.log(total)
}
```

Otra opción de hacerlo sería con **reduce**:

```
const suma = movimientos.reduce(
        (acc, cur) => acc + cur,
        0,
```

```
)
console.log(suma)
```

Vamos a imaginar que queremos solo sumar los resultados positivos. Habrá que hacer primero un filtro y luego sobre el mismo ejecutar el reduce anterior.

filter recibe como parámetro una función que por cada elemento del array devuelve verdadero o falso en función de si debe permanecer o no en el array.

```
const ingresos = movimientos.filter(
        function (mov) {
                if (mov>0) return true
                return false
        }
)
```

Con función flecha:

```
const ingresos = movimientos.filter((mov) => mov > 0)
console.log(ingresos)
```

Al ser programación funcional podemos concatenar métodos:

```
const ingresosTotales = movimientos
        .filter((mov) => mov > 0)
        .reduce((acc, cur) => acc + cur, 0)
console.log(ingresosTotales)
```

Otro ejemplo distinto, de un array de datos en € queremos obtener el mismo conjunto de datos pero en $.
Para ello utilizamos la función ***map***

```
const movimientosEnEuros = [-500, +200, -300, 800]
const eurToDolar = 1.02
const movimientosEnDolar = movimientosEnEuros.map(
        (mov) => mov * eurToDolar
)
console.log(movimientosEnDolar)
```

Podríamos continuar sacando solo ingresos o la suma de todos.