

# Desarrollo Web en Entorno Cliente

## Tema 2 – JavaScript avanzado y ECMA-Script

Marina Hurtado Rosales  
[marina.hurtado@escuelaartegranada.com](mailto:marina.hurtado@escuelaartegranada.com)



# Índice de contenidos

- Operador **Nullish Coalescing ??**
- Funciones **flecha**
- Programación **funcional**
- Parámetros **REST**
- Operador **spread ...**
- **Desestructuración** de objetos
- **Métodos** de arrays
- POO en Javascript. **Clases y prototipos**
- **Importación/exportación** de módulos

**Operator Nullish  
Coalescing ??**

# Operador Nullish coalescing

El operador “**nullish coalescing**” (fusión de null) se escribe con un doble signo de cierre de interrogación ??.

El resultado de **a ?? b**:

- **Será a**, si **a** no es null o undefined.
- **Será b**, si **a** es null o undefined.

```
let user;  
console.log(user ?? "Anonymous"); // Anonymous (user no definido)  
  
let name = "John";  
console.log(name ?? "Anonymous"); // John (name definido)
```

# Comparación ?? con ||

```
let height = 0; // altura cero
console.log(height || 100); // 100
console.log(height ?? 100); // 0
```

- **height || 100** verifica si **height** es “falso”.
- Como 0 es un valor **falsy**, el resultado del operador || será el segundo argumento, 100.
- **height ?? 100** verifica si **height** es **null o undefined**.
- Como no lo es, el valor de height se queda como está (0).
- En la práctica, una altura cero es un valor válido que no debería ser reemplazado por un valor por defecto. **En este caso ?? hace lo correcto, mientras que || no lo hace.**
- **NOTA:** en JavaScript existen ciertos valores que se consideran siempre falsos en comparaciones lógicas (false, 0, "", null, undefined y NaN). Se les denomina valores **falsy**.

# Operador Nullish coalescing

También existe la versión abreviada del operador

```
height??=100  
//Si height es undefined o null valdrá 100  
//En otro caso mantendrá su valor
```

La versión abreviada puede ser útil, por ejemplo, para contar palabras:

```
let frase=`Ir para casa,  
para poder cenar a tiempo,  
para poder dormir pronto`;  
  
let palabras=frase  
    .replaceAll(",","")  
    .split(" ");  
  
const contadores={};  
//Cada palabra es la clave y su valor es las veces que se repite
```

# Comparación sin y con ??

```
for (let palabra of palabras){  
  if(palabra in contadores){  
    contadores[palabra]++;  
  }else{  
    contadores[palabra]=1;  
  }  
}
```

```
for (let palabra of palabras){  
  //Inicializar variables en bucles  
  contadores[palabra]??=0;  
  contadores[palabra]++;  
}
```

# Funciones flecha



# Funciones arrow o flecha

```
//Son funciones con sintaxis reducida  
(x,y)=>{  
    return x+y;  
}
```

```
//Para darles nombre se meten en variables  
//como no se van a modificar se pone const  
const flecha=(x,y)=>{  
    return x+y;  
}  
console.log(flecha(5,6));
```

# Distintas sintaxis

```
//Si solo hay una instruccion no hace falta llaves  
//ni return  
const flecha=(x,y)=>x+y;  
}
```

```
//Si solo hay un parametro no hace falta parentesis  
const doble=x=>x*2  
//aunque me parece bastante confuso
```

```
//Si no hay parámetros, los paréntesis estarán vacíos  
//pero deben estar presentes:
```

```
const diHola= ()=>console.log("¡Hola!");
```

# Ejemplos con funciones arrow

```
const SumarIVA=(cantidad,porcentaje)=>{  
  let total;  
  
  total=cantidad+cantidad*porcentaje/100;  
  return total;  
}
```

```
const ElMayor=(num1,num2)=>{  
  let mayor;  
  
  if(num1>num2)  
  {  
    mayor=num1;  
  }else{  
    mayor=num2;  
  }  
  return mayor;  
}
```

```
let suma=SumarIVA(600,21);  
document.write(`El mayor es de 7 y 3 es ${ElMayor(3,7)}`);
```

En un principio siempre que podamos vamos a utilizar funciones arrow o flecha.

# Funciones arrow y programación funcional

- Como ya dijimos en el tema 1, en JavaScript las funciones son un tipo de datos, es decir, se pueden guardar en variables y pasarse como parámetros de otras funciones.
- La forma más compacta de las funciones flecha ayuda mediante funciones anónimas a su uso. En otras palabras, facilita el uso de **callbacks**.
- Este tipo de programación en la que una función (**función de orden superior**) se resuelve llamando a otras funciones que se le pasan como parámetros (**funciones Callback**) se llama **programación funcional**.

# Ejemplos simples de Callback

```
const exclamar(mensaje)=>{  
  console.log("¡¡"+mensaje+"!!");  
}
```

```
const neutro(mensaje)=>{  
  console.log(mensaje);  
}
```

```
const hablar=(entonacion)=>{  
  let frase=...;  
  entonacion(frase);  
}
```

```
hablar(exclamar);  
hablar(neutro);
```

```
const hablar=(entonacion)=>{  
  let frase=...;  
  entonacion(frase);  
}
```

```
hablar((mensaje)=>{  
  console.log(mensaje);  
}));
```

```
hablar((mensaje)=>{  
  console.log("¡¡"+mensaje+"!!");  
}));
```

# Funciones arrow y programación funcional

En muchos casos esto genera soluciones donde no hay **bucles (explícitamente)** y todo se resuelve con llamadas a funciones. Por ejemplo:

```
const productos=[
  {"nombre":"Bicicleta", "precio":100,"categoria":"deportes"},
  {"nombre":"TV", "precio":200,"categoria":"electronica"},
  {"nombre":"Album", "precio":10,"categoria":"papeleria"},
  {"nombre":"Libro", "precio":5,"categoria":"libreria"},
  {"nombre":"Telefono", "precio":500,"categoria":"electronica"},
  {"nombre":"Ordenador", "precio":1000,"categoria":"informatica"},
  {"nombre":"Teclado", "precio":25,"categoria":"informatica"}
];
```

```
const mostrarNombrePrecio=(articulo)=>
  console.log(`${articulo["nombre"]} precio
               ${articulo["precio"]}€<br>`);
}
productos.forEach(mostrarNombrePrecio);
```

# **Programación funcional**

# JSON para sustituir switch-case

```
const DISFRAZ_DEFECTO="PECERA";
let adversario=...
let misterio;

switch(adversario.toLowerCase()){
  case "loki":
    misterio="Lady Loki";
    break;
  case "hulk":
    misterio="Thanos";
    break;
  case "thor":
    misterio="Odin";
  case "superman":
    misterio="Darkseid";
    break;
  default:
    misterio=DISFRAZ_DEFECTO;
}
console.log( `Misterio se disfraza:${misterio}` );
```

```
const DISFRAZ_DEFECTO="PECERA";
const disfraces_misterio={
  "superman":"Darkseid",
  "thor":"Odin",
  "loki":"Lady Loki",
  "hulk":"Thanos"
}
let adversario=...
let misterio=disfraces_misterio[adversario.toLowerCase()] ?? DISFRAZ_DEFECTO;

console.log(`Misterio se disfraza:${misterio}`);
```



# Eliminar el switch con programación funcional

```
const persona = {  
  nombre: "Jose Fernández García",  
  edad: 27,  
  peso: 82,  
  altura: 180  
}  
  
for (let propiedad in persona){  
  switch(propiedad){  
    case "nombre":  
      console.log(persona[propiedad].toUpperCase());  
      break;  
    case "edad":  
      console.log(persona[propiedad] + " años");  
      break;  
    case "peso":  
      console.log(persona[propiedad] + " kilos");  
      break;  
    case "altura":  
      console.log(persona[propiedad] + " centímetros");  
      break;  
  }  
}
```

# Eliminar el switch con programación funcional

```
formatos = {}  
formatos.nombre = (nombre) => console.log(nombre.toUpperCase());  
formatos.edad = (edad) => console.log(edad + " años");  
formatos.peso = (peso) => console.log(peso + " kilos");  
formatos.altura = (altura) => console.log(altura + " centímetros");  
  
for(let propiedad in persona){  
  formatos[propiedad](persona[propiedad])  
}
```

# Parámetros REST

# Parámetros REST

Una función puede ser llamada con cualquier número de argumentos sin importar cómo sea definida.

Por ejemplo:

```
const suma=(a,b)=> {  
  return a + b;  
}  
  
console.log(suma(1,2,3,4,5));
```

Aquí no habrá ningún error por “exceso” de argumentos. Pero, por supuesto, en el resultado solo los dos primeros serán tomados en cuenta.

# Parámetros REST

El resto de parámetros pueden ser referenciados en la definición de una función con 3 puntos ... seguidos por el nombre del array que los contendrá.

```
const sumaTodo=(...numeros)=> {  
  // numeros es el nombre del array  
  let sum = 0;  
  
  for (let num of numeros){  
    sum+=num;  
  }  
  
  return sum;  
}  
  
console.log(sumaTodo(1));//1  
console.log(sumaTodo(1,2));//3  
console.log(sumaTodo(1,2,3));//6
```

# Parámetros REST

Literalmente significan “Reunir los parámetros restantes en un array”. Veremos que también se llama spread (como operador aparte) o parámetros REST en las funciones.

También funciona con la palabra reservada **function**.

```
function limpiarEspacios(...cadenas) {  
  for (let i=0; i<cadenas.length; i++) {  
    cadenas[i] = cadenas[i].trim();  
  }  
  return cadenas;  
}  
  
let cadenasLimpias = limpiarEspacios('hola ', ' algo ', ' más');  
console.log(cadenasLimpias);
```

# Parámetros REST

Podemos elegir obtener los primeros parámetros como variables y juntar solo el resto.

Aquí los primeros dos argumentos van a variables y el resto va al array títulos:

```
const mostrarNombre = (nombre, apellido, ...titulos) => {  
  console.log(nombre + ' ' + apellido); //Julio Cesar  
  
  // El resto va en el array titulos  
  // Por ejemplo titulos = ["Cónsul", "Emperador"]  
  console.log(titulos[0]); // Cónsul  
  console.log(titulos[1]); // Emperador  
  console.log(titulos.length); // 2  
}  
  
mostrarNombre("Julio", "Cesar", "Cónsul", "Emperador");
```

```
const f = (arg1, ...rest, arg2) => {  
  // error  
}  
// ...rest debe ir siempre último
```

**Operador spread ...**



# Operador spread en funciones

A veces necesitamos hacer exactamente lo opuesto. Por ejemplo, existe una función nativa **Math.max** que devuelve el número más grande de una lista:

```
console.log(Math.max(3,5,1)); // 5
```

Ahora supongamos que tenemos un array en lugar de una lista:

```
let arr = [3, 5, 1];  
console.log(Math.max(arr)); // NaN
```

¡Operador **spread** al rescate! Cuando **...arr** es usado en el objeto de una función, “expande” el objeto iterable en una lista de argumentos.

```
let arr = [3, 5, 1];  
console.log(Math.max(...arr)); // 5  
//(spread convierte el array en una lista de argumentos)
```

# Operador spread en funciones

También podemos pasar múltiples iterables de esta manera:

```
function limpiarEspacios(...cadenas) {  
  ...  
  const cadenasOriginales = ['hola ', ' algo ', ' más'];  
  let cadenasLimpias = limpiarEspacios(...cadenasOriginales);  
}
```

[ ] Podemos combinar el operador spread incluso con valores normales

```
let arr1=[1,-2,3,4];  
let arr2=[8,3,-8,1];  
  
console.log(Math.max(...arr1,...arr2)); // 8  
  
console.log(Math.max(1,...arr1,2,...arr2,25)); // 25
```

# Operador spread

Fuera de las funciones, el operador spread se puede usar también con arrays y objetos.

Creando copias de los mismos para compactar el código.

```
let arr = [1,2,3];
let arrCopy = [...arr];
//Contienen los mismo pero son arrays distintos

const letras=['a','b','c'];
const palabras=['cat','dog','horse'];
const todo=[...letras,...palabras];
//[ 'a', 'b', 'c', 'cat', 'dog', 'horse' ]

let arr = [3,5,1];
let arr2 = [8,9,15];
let merged = [0,...arr,2,...arr2];
// 0,3,5,1,2,8,9,15 |

const numeros=[1,2,3];
const masnumeros=[...numeros,4,5];
//[1,2,3,4,5]
```

# Operador spread

```
//Copiando objetos (clonación)
let obj ={a:1,b:2,c:3};
let objCopy ={...obj};

const coche={
  marca:'seat',
  modelo: 'leon',
  puertas:5
}

const motor={
  cilindros:4,
  caballos: 120,
}

const cochecompleto={...coche,...motor};

//copiar modificando y/o añadiendo datos
const deportivo={...coche,
  puertas:3,
  precio:100000}
```

# **Desestructuración de objetos**

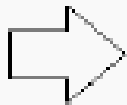
# Desestructuración de objetos

Las dos estructuras de datos más usadas en JavaScript son **Object** y **Array**.

Normalmente no necesitamos un objeto o array como un conjunto sino en piezas individuales.

Además si accedemos repetidamente a una serie de propiedades de un objeto o posiciones de un array supone un gasto de computación.

```
let valor=arra[2];  
console.log(valor);  
console.log(valor);  
console.log(valor);
```



Menos eficiente

```
console.log(arr[2]);  
console.log(arr[2]);  
console.log(arr[2]);
```

# Desestructuración de objetos

La asignación desestructurante es una sintaxis especial que nos permite “desempaquetar” arrays u objetos en varias variables, porque a veces es más conveniente.

```
let arr = ["John", "Smith"]  
  
// asignación desestructurante  
let [nombre, apellido] = arr;  
console.log(nombre); // John  
console.log(apellido); // Smith
```

Ahora podemos trabajar con variables en lugar de miembros de array. Es sólo una manera más simple de escribir:

```
let nombre = arr[0];  
let apellido = arr[1];
```

# Desestructuración de objetos

Se ve genial cuando se combina con Split u otro método que devuelve un array:

```
let [nombre, apellido] = "John Smith".split(' ');  
console.log(nombre); // John  
console.log(apellido); // Smith
```

En este código el segundo elemento del array es omitido, el tercer es asignado a title, y el resto de los elementos también se omiten.

```
let [nombre, , titulo] = ["Julius", "Caesar", "Consul", "of the Roman Republic"];  
console.log(titulo); // Consul
```



# Desestructuración de objetos

También funciona con string

```
let [a, b, c] = "abc"; // ["a", "b", "c"]
```

El lado izquierdo no tiene que estar compuesto de variables, puede ser un objeto:

```
let user = {};  
[user.name, user.apellido] = ["John", "Smith"];  
  
console.log(user.name); // John  
console.log(user.apellido); // Smith
```

TRUCO: intercambiamos valores

```
let invitado = "Jane";  
let admin = "Pete";  
[invitado, admin] = [admin, invitado];
```

# Desestructuración de objetos

Si queremos también obtener todo lo que sigue podemos agregarle un parámetro que obtiene el resto usando puntos suspensivos ...

```
let [nombre1, nombre2, ...titulos] = ["Julius",  
                                     "Caesar",  
                                     "Consul", "of the Roman Republic"];  
  
// `titulos` es un array de ítems,  
comenzando en este caso por el tercero.  
console.log(titulos[0]); // Consul  
console.log(titulos[1]); // of the Roman Republic
```

# Desestructuración de objetos

La asignación desestructurante también funciona con objetos

```
let options = {  
  titulo: "Menu",  
  width: 100,  
  height: 200  
};  
  
let {titulo, width, height} = options;  
  
console.log(titulo); // Menu  
console.log(width);  // 100  
console.log(height); // 200
```

# Desestructuración de objetos

No importa el orden sino los nombres. Esto también funciona:

```
let {height, width, titulo} = { titulo: "Menu", height: 200, width: 100 }
```

Si queremos asignar una propiedad a una variable con otro nombre:

```
let {width: w, height: h, titulo} = options;

// width -> w
// height -> h
// titulo -> titulo

console.log(titulo); // Menu
console.log(w);      // 100
console.log(h);      // 200
```

# Desestructuración de objetos

Si tenemos un objeto complejo con muchas propiedades, podemos extraer solamente las que necesitamos:

```
let options = {  
  titulo: "Menu"  
};  
// sólo extrae título como variable  
let { titulo } = options;  
console.log(titulo); // Menu
```

Si no declaramos las variables en el mismo momento de la desestructuración o retornamos un objeto en una función, hay que rodear la expresión entre paréntesis (...) para que funcione:

```
let titulo, width, height;  
({titulo, width, height} = {titulo: "Menu", width: 200, height: 100});
```

# Desestructuración de objetos

Podemos usar el patrón resto de la misma forma que lo usamos con arrays.

```
// titulo = propiedad llamada titulo  
// resto = objeto con el resto de las propiedades  
let {titulo, ...resto} = options;  
// ahora titulo="Menu", resto={height: 200, width: 100}
```

Y si cogemos width

```
let {width, ...resto} = options;  
// ahora width=100, resto={height: 200, titulo="Menu"}
```

# Desestructuración de objetos


Si un objeto o array contiene objetos y arrays anidados, podemos utilizar patrones del lado izquierdo más complejos para extraer porciones más profundas

```
let options = {
  size: {
    width: 100,
    height: 200
  },
  items: ["Cake", "Donut"],
  extra: true
};

let {
  size: { // colocar tamaño aquí
    width, //width:w
    height //height:h si queremos las variables con otros nombres
  },
  items: [item1, item2], // asignar items aquí
  extra
} = options;
```

# Desestructuración de objetos

```
let {  
  size: {  
    width,  
    height  
  },  
  items: [item1, item2],  
  title = "Menu"  
}  
  
let options = {  
  size: {  
    width: 100,  
    height: 200  
  },  
  items: ["Cake", "Donut"],  
  extra: true  
}
```



```
console.log(width); // 100  
console.log(height); // 200  
console.log(item1); // Cake  
console.log(item2); // Donut  
console.log(extra); // true
```



# **Métodos de Arrays**

# forEach

**forEach** nos permite recorrer uno a uno los elementos de un iterable.  
Podemos observar el uso de la desestructuración.

```
const productos=[
  {"nombre":"Bicicleta", "precio":100,"categoria":"deportes"},
  {"nombre":"TV", "precio":200,"categoria":"electronica"},
  {"nombre":"Album", "precio":10,"categoria":"papeleria"},
  {"nombre":"Libro", "precio":5,"categoria":"libreria"},
  {"nombre":"Telefono", "precio":500,"categoria":"electronica"},
  {"nombre":"Ordenador", "precio":1000,"categoria":"informatica"},
  {"nombre":"Teclado", "precio":25,"categoria":"informatica"}
];
```

```
productos.forEach(
  function(p){
    //Deseestructuracion de objetos
    const {nombre,precio}=p;
    console.log(`${nombre} precio
                  ${precio}€<br>`);
  }
);
```

```
productos.forEach(
  (p)=>{
    //Deseestructuracion de objetos
    const {nombre,precio}=p;
    console.log(`${nombre} precio
                  ${precio}€<br>`);
  }
);
```

# forEach

Estos métodos suelen tener parámetros opcionales para ser idénticos a los bucles

```
productos.forEach(  
  (producto, posicion)=>{  
    const {nombre, precio}=producto;  
    console.log(`El producto en la posicion ${posicion+1}  
                ${nombre} precio  
                ${precio}€<br>`);  
  }  
));
```

# find y findIndex

```
let buscado=productos.find(p=>p.nombre==="Libro");
console.log(`precio es ${buscado.precio}€<br>`);

let nombre_buscado=...cualquier nombre
let buscado=productos.find(p=>p.nombre===nombre_buscado);

if(buscado!==undefined){
    console.log(`precio es ${buscado.precio}€<br>`);
}else{
    alert("No encontrado");
}

let nombre_buscado=...cualquier nombre
let buscado=productos.findIndex(p=>p.nombre===nombre_buscado);

if(posicion!==-1){
    console.log(`precio es ${productos[posicion].precio}€<br>`);
}else{
    alert("No encontrado");
}
```

# filter, some, every y sort

{ }

```
//Un array nuevo con los objetos con 100 o menos de precio  
let filtrado_productos=productos.filter(p=>p.precio<=100);
```

```
//Existe al menos un producto con 100 o menos  
let hay_baratos=productos.some(p=>p.precio<=100);
```

```
//Todos al menos un producto con 100 o menos  
let todos_baratos=productos.every(p=>p.precio<=100);
```

```
//Para ordenar según el criterio que queramos  
//CUIDADO MUTA EL ARRAY  
//Por precio  
productos.sort((a,b)=>a.precio-b.precio);
```

```
//Por orden alfabético, locale compare tiene en cuenta acentos etc  
productos.sort((a,b)=>a.nombre.localeCompare(b.nombre));
```

[ ]

[ ]

## Cómo eliminar elementos repetidos con **filter**

```
const datos=[1,2,4,4,5,4,7,9,1,1,7,2];  
let sinrepes=datos.filter((elemento,posicion)=>datos.indexOf(elemento)===posicion);
```

## Cómo desordenar un array

```
const numeros=[3,9,1,8,7,6,5,4,2];  
numeros.sort(()=>Math.random() - 0.5);
```

## Cómo hacer una criba aleatoria

```
let numeros=[3,9,1,8,7,6,5,4,2];  
numero=numeros.filter(()=>Math.random() - 0.5);  
//Caso en el queremos un array con let
```

# map

El método **map** genera un nuevo array a partir de otro aplicando una función a cada elemento del array

```
const numeros=[1,2,3,4,5,6];  
let triples=numeros.map(n=>n*3);  
//triples deberia ser [3,6,9,12,15,18];
```

También permite “recortar datos” proyectando solo los atributos que queramos

```
let solo_nombres=productos.map(p=>p.nombre);  
//solo nombres deberia contener ["Bicicleta","TV","Album"]
```

# map

Incluso permite devolver arrays de objetos recortados

```
//Recorte más concreto sin mutar el array original
let quitar_categorias=productos.map(
  (p)=>{
    const {nombre,precio}=p;
    return {
      nombre,
      precio
    }
  }
)
```

O ampliar usando **...spread**

```
//Ampliación sin mutar el array original
let añadir_existencias=productos.map(
  (p)=>({
    ...p,      //En lugar de aleatorio cualquier cosa
    "existencias":Math.floor(Math.random()*101)
  })
);
//Necesarios los parentesis para que no crea que es un bloque de código
```



# El propio array como parámetro

Todos los métodos de los array incluyen un parámetro opcional por defecto que es el propio array. Siempre lo reciben y no hace falta ponerlo.

Esto es útil cuando se encadenan varias llamadas a métodos.

```
const datos=[1,2,4,4,6,4,6,9,1,1,6,2];  
let sinrepes_transformado=  
  datos.map(d=>d**Math.random()*3)  
    .filter((elemento,posicion,array)=>array.indexOf(elemento)===posicion);  
//El parametro array es el resultado del map  
//si pusieramos el array datos el código no funcionaria bien
```

# Combinación de métodos de arrays

Encadenando funciones de arrays podemos conseguir códigos compactos.

POR FAVOR no hay que llevarlo al extremo.

```
const orden_precio=productos.sort((a,b)=>b.precio-a.precio);  
const solo_nombres=orden_precio.map(p=>p.nombre);  
console.log("El más caro:"+solo_nombres[0]);
```

```
console.log("El más caro:"+  
  productos.sort((a,b)=>b.precio-a.precio).map(p=>p.nombre)[0]);  
//map es lo ultimo que se ejecuta y devuelve un array
```

Los métodos de array son realmente interesantes, pero no se pueden aplicar a objetos directamente por ejemplo.

```
let persona = {  
  "nombre": "Fernando",  
  "apellidos": "Garcio Escudero",  
  "edad": 48,  
  "telefono": "555666777"  
}
```

Solo podríamos usar el for...in para iterar sobre este objeto.

Hay una solución alternativa mediante la clase **Object**.

# Métodos de Object: keys, values, entries

Estos métodos general arrays con los distintos componentes de un objeto clave-valor.

```
Object.keys(persona)
['apellidos', 'telefono', 'edad', 'nombre']
Object.values(persona)
['Garcio Escudero', '555666777', '48', 'Fernando']
Object.entries(persona)
[['apellidos', 'Garcio Escudero'],
 ['telefono', '555666777'],
 ['edad', '48'],
 ['nombre', 'Fernando']]
```

Estos métodos son similares a los Map que existen en el lenguaje Java

# Métodos de Object: keys, values, entries

```
Object.keys(persona).forEach(  
  (clave)=>{  
    | console.log(clave+" "+persona.getItem(clave))  
  });
```

```
Object.values(persona).forEach(  
  (valores)=>{  
    | console.log(valores)  
  });
```

```
Object.entries(persona).forEach(  
  (entrada)=>{  
    | console.log(entrada[0]+" "+entrada[1])  
  });
```

# Métodos de Object: keys, values, entries

Esto nos permite hacer cosas sólo con métodos.

```
let sueldos={  
  "Juan":2300,  
  "Ana":1500,  
  "Fernando":1000  
};  
  
suma=0;  
Object.values(sueldos).forEach(  
  sueldo=>{suma+=sueldo};  
)  
console.log(suma) //4800
```

# Métodos de Object: fromEntries

Si usamos **map** obtenemos una copia transformada.

```
aumentados=Object.entries(sueldos)
                    .map(entrada=>[entrada[0],entrada[1]*1.05]);
//[['Juan', 2415],['Ana', 1575],['Fernando', 1050]]
```

Seguimos teniendo un array que no es un objeto, pero volver a transformarlo en objeto es fácil con el método **fromEntries**

```
sueldos=Object.fromEntries(aumentados)
//{Juan:2415,Ana:1575,Fernando:1050}
```

# Métodos de Object: fromEntries

El método **fromEntries** es muy interesante porque podemos generar objetos de cualquier conjunto de arrays.

```
nombres=["Bicicleta","Album","Libro"];  
precios=[200,100,13];
```

```
[ ] Object.fromEntries(nombres.map((valor,pos)=>[valor,precios[pos]]));  
{  
  "Bicicleta":200,  
  "Album":100,  
  "Libro":13  
}
```