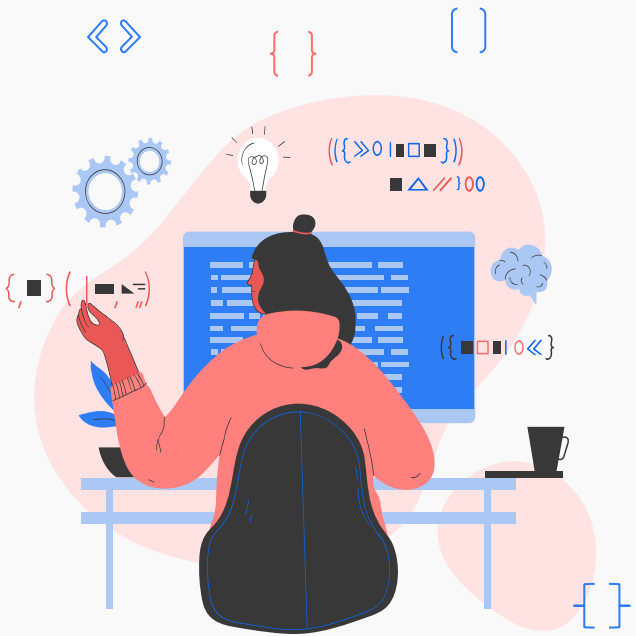
Desarrollo Web en Entorno Cliente

Tema 2 – JavaScript avanzado y ECMA-Script

Marina Hurtado Rosales
marina.hurtado@escuelaartegranada.com



Indice de contenidos

Operador 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] + " centimetros");
            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 + " centimetros");

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}
```