



*Operadores avanzados*

**Sugar Syntax**

A medida que el lenguaje va evolucionando con el tiempo, aparecen nuevos operadores que nos permiten resolver tareas recurrentes de forma simplificada. Es normal que vayan surgiendo con las nuevas versiones de JS nuevas herramientas que nos permitan optimizar el código que escribimos.

***Sugar Syntax*** es el nombre que se le da a estos operadores que funcionan como *simplificaciones* de tareas más complejas. El operador **++** es un ejemplo de esto:

| let num = 10  // aumentar en 1 el valor  num = num + 1  // primera simplificacion  num += 1  // o bien  num++ |
| --- |

La operación es aumentar el valor de la variable en 1 y tenemos distintas opciones para lograr lo mismo. Salvo la primera, las otras dos son ejemplos de *sugar syntax,* donde se aplican operadores que se crean para simplificar la tarea anterior. Nótese que el resultado es escribir una operación compleja con mucho menos código, lo cual nos permite pensar soluciones en un nivel más alto, ya que estamos trabajando con un mayor nivel de abstracción resumiendo una tarea compleja en un operador sencillo.

**Operador ternario**

El operador ternario es una simplificación de la estructura condicional *if…else*. Cuando nos encontramos frente a una estructura del tipo *condición - caso 1 - caso 2*, podemos utilizar el operador ternario para simplificarlo. Es decir, el *ternario* significa que es un condicional que consta *sí o sí* de tres partes: la condición, el caso de ejecución en caso que se cumpla, y el caso *else* si no se cumple. Por ejemplo:

| let temperatura = 31  if (temperatura > 30) {  alert("Día caluroso!")  } else {  alert("Día agradable")  } |
| --- |

El operador ternario tiene la siguiente sintaxis:

| condicion ? caso1 : caso2 |
| --- |

En este caso, lo que está a la *izquierda* del signo **?** es la *condición*, la cual resuelve true o false; a la derecha se escribe la instrucción a ejecutar si la condición es verdadera, y luego del **:** se escribe la instrucción si la condición es falsa. Nótese que es una sintaxis para estructuras simples, donde la condición es generalmente pequeña y los casos a resolver también. Con un operador ternario el ejemplo anterior quedaría así:

| temperatura > 30 ? alert("Día caluroso!") : alert("Día agradable") |
| --- |

Algo interesante que ofrece el operador ternario es un *return implícito* para cada caso. Es decir, no sólo ejecuta la instrucción resuelta, sino que hace ***return*** de la misma. Esto es muy útil cuando queremos retornar valores de forma condicional, lo cual con una estructura tradicional sería más extenso. Por ejemplo:

| let permiso  if (usuario.edad >= 18) {  permiso = true  } else {  permiso = false  }  if (permiso) {  alert("Puede comprar cerveza")  } else {  alert("No puede comprar")  } |
| --- |

Al tener una variable cuyo valor inicial depende del resultado de un condicional, vemos que primero debemos declarar la variable y luego evaluar cual va a ser el valor a asignarle. Con el operador ternario podemos reducir esto a una sola línea ya que hacemos *return* de uno de los casos, y por lo tanto lo asignamos en la declaración:

| const usuario = {  nombre: "John Doe",  edad: 22  }  // declaramos y asignamos condicionalmente  const permiso = (usuario.edad >= 18) ? true : false  // mostramos el mensaje  permiso ? alert("Puede comprar cerveza") : alert("No puede comprar") |
| --- |

Nótese lo simple y clara que se vuelve la escritura de esta forma cuando trabajamos con condicionales sencillos que se pueden traducir en un operador ternario. Para estructuras más complejas de control de flujos recaemos en la estructura if tradicional y está bien que así sea. Lo importante es remarcar la ventaja de este tipo de operadores y saber aplicarlos con precisión.

**Operador lógico AND**

El operador lógico AND ( **&&** ) también es una reducción de un condicional, pero trata de ejecutar (o retornar) algo *sólo si la condición es verdadera*. Es decir, es una reducción de un*if* sencillo con un solo bloque de ejecución:

| const carrito = []  if (carrito.length === 0) {  console.log("El carrito está vacío!")  }  // con operador AND  carrito.length === 0 && console.log("El carrito está vacío!") |
| --- |

Todos estos operadores también tienen *return implícito*, por lo que se puede utilizar para enviar o asignar valores:

| const usuario = {  nombre: "John Doe",  edad: 14  }  const registroIngreso = usuario.edad >= 18 && new Date()  console.log(registroIngreso) |
| --- |

En este ejemplo, si el usuario tiene edad mayor o igual que 18 se almacena una nueva fecha en la variable. En el caso de que la condición resulte falsa, el operador AND retornará **false** en cambio.

| const usuario = {  nombre: "John Doe",  edad: 14  }  const registroIngreso = usuario.edad >= 18 && new Date()  console.log(registroIngreso) // FALSE |
| --- |

**Operador lógico OR**

El operador lógico OR ( || ) es sintácticamente similar al anterior, con la diferencia que consta de dos operandos y no de una condición explícita:

***operando1 || operando2***

El operador OR retorna el operando de la izquierda si *no es* ***falsy***, y si no retorna el de la derecha. Es decir, si el operador de la izquierda es *distinto de 0, null, undefined, NaN, false,* o *string vacío;* entonces retorna ese operando, y sino retorna el de la derecha. Esto es muy versátil para condicionar asignaciones de variables o de parámetros de forma sencilla.

| const usuario1 = {  nombre: "John Doe",  edad: 14  }  const usuario2 = null  console.log( usuario1 || "El usuario no existe" )  // { nombre: 'John Doe', edad: 14 }  console.log( usuario2 || "El usuario no existe" )  // El usuario no existe |
| --- |

Es útil cuando quiero inicializar variables de forma condicionada, evaluando algún valor previo. Por ejemplo, si quiero recuperar el último estado de un carrito de compras del usuario almacenado en localStorage al iniciar mi app, debería hacer algo como lo siguiente:

| let carrito  let carritoLocalStorage = JSON.parse( localStorage.getItem('carrito') )  if (carritoLocalStorage) {  carrito = carritoLocalStorage  } else {  carrito = []  } |
| --- |

Según si hay algún carrito guardado o no en localSotrage, mi variable *carrito* se inicia con ese valor recuperado o como un array vacío. Con el operador || puedo simplificar esto en una instrucción sencilla y clara:

| const carrito = JSON.parse(localStorage.getItem('carrito')) || [] |
| --- |

Si *no hay* nada guardado en localStorage (es decir, obtengo *null* en la izquierda) se retorna el array vacío de la derecha, y sino se retorna lo recuperado del LS.

**Tabla de valores falsy**

Para que quede claro como son los returns del operador ||, se adjunta la siguiente tabla de evaluación de valores falsy:

| console.log( 0 || "Falsy") // Falsy  console.log( 40 || "Falsy") // 40  console.log( null || "Falsy") // Falsy  console.log( undefined || "Falsy") // Falsy  console.log( "Hola Mundo" || "Falsy") // Hola Mundo  console.log( "" || "Falsy") // Falsy  console.log( NaN || "Falsy") // Falsy  console.log( true || "Falsy") // true  console.log( false || "Falsy") // Falsy |
| --- |

**Nullish coalescing operator**

El operador ?? , conocido como *nullish coalescing operator*, funciona exactamente igual que el operador ||, con la diferencia de que retorna el operador de la izquierda sólo si el de la derecha es ***null*** o ***undefined***. Es decir, es similar al anterior pero admite más valores como ‘verdaderos’. Útil cuando en esta evaluación necesitamos incluir valores como el 0 o strings vacíos. En comparación con la tabla anterior, sólo obtenemos Nullish en dos casos:

| console.log( 0 ?? "Nullish") // 0  console.log( 40 ?? "Nullish") // 40  console.log( null ?? "Nullish") // Nullish  console.log( undefined ?? "Nullish") // Nullish  console.log( "Hola Mundo" ?? "Nullish") // Hola Mundo  console.log( "" ?? "Nullish") // ""  console.log( NaN ?? "Nullish") // NaN  console.log( true ?? "Nullish") // true  console.log( false ?? "Nullish") // false |
| --- |

**Acceso condicional a un objeto**

Si intentamos acceder a un objeto que no existe naturalmente obtendremos un error. Sin embargo, podemos utilizar el operador **?** sobre la referencia de un objeto para condicionar su acceso y tener un mejor control de errores.

| const usuario = null  console.log( usuario.nombre || "El usuario no existe" )  // Error: "No se pueden leer propiedadesde NULL"  console.log( usuario?.nombre || "El usuario no existe")  // "El usuario no existe" |
| --- |

En este caso, *usuario.nombre* retorna un error de acceso, ya que no se puede leer propiedades de *null*; pero *usuario****?****.nombre* retorna **undefined** en cambio, y permite trabajar el acceso de forma condicional evitando errores en la ejecución.

El operador **?** también puede aplicarse sobre propiedades que sean objetos, para evaluar su existencia/validez y controlar los flujos del programa.

| const usuario = {  nombre: "John Doe",  edad: 22,  cursos: {  javascript: "aprobado"  }  }  console.log( usuario?.cursos?.javascript || "La propiedad no existe")  // "aprobado"  console.log( usuario?.trabajos?.coderhouse || "La propiedad no existe")  // "La propiedad no existe" |
| --- |

*Desestructuración*

Muchas veces queremos acceder a propiedades de objetos y almacenarlas en variables diferentes para un posterior uso. Típicamente haríamos algo como lo siguiente para esto:

| const usuario = {  nombre: "John Doe",  edad: 32  }  let nombre = usuario.nombre  let edad = usuario.edad |
| --- |

Declaramos variables y en ellas almacenamos los valores de las propiedades. Hacemos esto para trabajar con inmutabilidad, es decir utilizar esos valores sin riesgo de alterar las propiedades del objeto.

Sin embargo, podemos utilizar la **desestructuración** para simplificar y agilizar este proceso. *¿En qué consiste?* Es una técnica que nos permite declarar variables donde guardar propiedades de un objeto de forma rápida y directa. La sintaxis es la siguiente:

let { prop1, prop2 } = objeto

Nótese las llaves a la izquierda del operador =, esto significa que estamos *desestructurando un objeto*. Significa que estamos creando dos variables, *prop1* y *prop2*, donde se almacenan las propiedades con el mismo nombre del objeto que referenciamos a la derecha.

Es decir, los nombres de las variables deben *coincidir exactamente* con los nombres de las propiedades que queremos obtener del objeto. En el ejemplo anterior, podemos desestructurar el objeto de la siguiente forma:

| const usuario = {  nombre: "John Doe",  edad: 32  }  const { nombre, edad} = usuario  console.log(nombre) // "John Doe"  console.log(edad) // 32 |
| --- |

Si intentamos desestructurar una propiedad inexistente en el objeto, obtendremos *undefined*. Cada propiedad que queramos desestructurar del objeto las declaramos separadas por comas:

| const usuario = {  nombre: "John Doe",  edad: 32  }  const { telefono } = usuario // undefined |
| --- |

Recordemos que que en este caso y en los anteriores, estamos ***declarando variables*** con los nombres *nombre, edad, y teléfono*; por lo que luego las referenciamos con este nombre.

Si queremos acceder a propiedades más internas dentro de un objeto, es decir desestructurar alguna propiedad que sea a la vez un objeto, es posible hacerlo siguiendo el mismo patrón:

| const usuario = {  nombre: "John Doe",  edad: 32,  telefono: {  cel: 113334444,  casa: null,  trabajo: 113325555  }  }  const { nombre, telefono: {trabajo} } = usuario  console.log(nombre) // "John Doe"  console.log(trabajo) // 113325555 |
| --- |

En este caso, como telefono es un objeto, desestructuramos la propiedad *trabajo* de éste, dentro de la desestructuración de *usuario*. Nótese que finalmente se terminan declarando dos variables, nombre y trabajo.

**Alias**

Para que la desestructuración funcione debe haber coincidencia con los nombres de las propiedades del objeto. Sin embargo a veces puede que los nombres de las propiedades no sean muy descriptivos para el uso que queremos darle, y por ello podemos desestructurarlas con un *alias,* es decir declarar la variable con un nombre alternativo tras haber desestructurado el objeto. Esto lo hacemos simplemente con el operador **:** luego del nombre de la propiedad:

| const item = {  item\_id: 432,  product\_name: "Some product",  price\_per\_unit: 5600  }  const {  item\_id: id,  product\_name: nombre,  price\_per\_unit: precio  } = item  console.log(id) // 432  console.log(nombre) // "Some product"  console.log(precio) // 5600 |
| --- |

En este caso desestructuramos todas las propiedades de *item*, pero lo almacenamos en variables denominadas ***id, nombre, precio***, a través del alias que indicamos para cada una.

**Desestructuración en parámetros**

Si en una función recibimos objetos por parámetros, también es posible desestructurarlos directamente en el llamado, definiendo esto al declarar la función. Por ejemplo, supongamos una función que recibe un objeto *producto* por parámetro y debe trabajar con sus propiedades *id* y *nombre*.

| const producto = {  id: 10,  nombre: "Curso Javascript",  precio: 12500  }  const desestructurar = (item) => {  // desestructurando dentro del bloque  const {id, nombre} = item  console.log(id, nombre)  }  desestructurar(producto) // 10 Curso Javascript |
| --- |

Sabiendo qué es lo que vamos a recibir y qué necesitamos desestructurar, podemos traducir esto con la siguiente lógica:

| // desestructurando lo que reciba por parámetro  const desestructurar = ( {id, nombre} ) => {  console.log(id, nombre)  }  desestructurar(producto) // 10 Curso Javascript |
| --- |

Otro ejemplo, en este caso capturando las posiciones *x* e *y* del objeto evento del click sobre la pantalla, mostrando esas posiciones por consola. Esto es muy útil cuando trabajamos con objetos grandes (como el de evento) y sólo necesitamos pocas propiedades de éste:

| window.addEventListener('click', ( {x, y} ) => {  console.log(x, y)  }) |
| --- |

**Desestructuración de arrays**

Es posible desestructurar arrays de forma similar, usando corchetes [] en vez de llaves. La diferencia con la desestructuración de objetos es que la de arrays es **posicional**. Es decir, declaramos las variables en orden y estas almacenan los valores de las mismas posiciones del array de referencia:

| const nombres = ["Juan", "Julieta", "Carlos", "Mariela"]  const [a, b] = nombres  console.log(a) // "Juan"  console.log(b) // "Julieta" |
| --- |

No funciona aquí la coincidencia por nombres, sino que se toman los valores según la posición. Las dos primeras variables que declaramos tomarán los valores de los dos primeros elementos del array. Si queremos acceder a otras posiciones, o mejor dicho omitir las primeras, podemos hacerlo dejando espacios vacíos con comas:

| const nombres = ["Juan", "Julieta", "Carlos", "Mariela"]  // omito las dos primeras posiciones  const [,, a, b] = nombres  console.log(a) // "Carlos"  console.log(b) // "Mariela" |
| --- |

*Spread*

El operador spread ( **…** ) es una herramienta que nos permite, como su nombre indica, *desparramar* un array u objeto. Esto significa cambiar la forma en la que *presentamos* este array u objeto. Por ejemplo, si paso un array por parámetro a alguna función, ésta recibe el array entero como tal:

| const nombres = ["Juan", "Julieta", "Carlos", "Mariela"]  console.log(nombres) // ["Juan", "Julieta", "Carlos", "Mariela"] |
| --- |

Vemos que se hace console.log del array entero porque eso enviamos por parámetro. Pero si en cambio enviamos un *spread* del array, veremos lo siguiente:

| const nombres = ["Juan", "Julieta", "Carlos", "Mariela"]  // spread ... del array  console.log(...nombres) // Juan Julieta Carlos Mariela  // equivalente a:  console.log("Juan", "Julieta", "Carlo", "Mariela") |
| --- |

Vemos que el resultado del *spread* es equivalente a enviar los cuatro elementos del array por separado como parámetros. Pues ésto es exactamente lo que hace el *spread*: al aplicarse sobre un array, en vez de enviarlo como *array,* envía todos sus elementos como parámetros individuales.

Así, si el array *nombres* tiene 4 elementos, al hacer *console.log( nombres )* estamos enviando 1 parámetro, pero al ejecutar *console.log(* ***…nombres*** *)*estamos enviando **cuatro** parámetros a la función.

Esto es útil cuando tenemos datos ordenados dentro de una colección pero trabajamos con funciones que *no* funcionan recibiendo arrays, sino una serie de parámetros individuales, como puede ser la función Math.max() o Math.min().

Por ejemplo, si tengo un array de números y necesito saber cuál es el mayor o menor, podría intentar lo siguiente:

| const numeros = [4, 77, 92, 10, 3, -32, 54, 11]  console.log( Math.max(numeros) ) // NaN |
| --- |

Recibimos NaN porque la función no admite recibir un *array* por parámetro, sino que necesita recibir una serie de parámetros numéricos. Con el operador spread podemos solucionar esto, y Math.max() recibirá *cada elemento del array como un parámetro individual:*

| const numeros = [4, 77, 92, 10, 3, -32, 54, 11]  console.log( Math.max(...numeros) ) // 92 |
| --- |

También podemos hacer *spread* de un array dentro de otras estructuras que lo admitan, como puede ser otro array o un objeto. Esto nos permite por ejemplo replicar el contenido de un array dentro de otra estructura al *desparramar* su contenido dentro. Si lo hacemos dentro de un objeto veremos algo interesante, que cada propiedad toma como nombre el índice de los elementos:

| const nombres1 = ["Juan", "Julieta"]  const nombres2 = ["Carlos", "Mariela"]  // spread de los dos arrays dentro de otro  const nombres = [...nombres1, ...nombres2]  console.log(nombres) // ["Juan", "Julieta", "Carlos", "Mariela"]  // spread del array en un objeto  const nombresObj = {  ...nombres  }  console.log(nombresObj)  // { '0': 'Juan', '1': 'Julieta', '2': 'Carlos', '3': 'Mariela' } |
| --- |

**Spread de objetos**

Se puede hacer spread de objetos también, pero *debe hacerse dentro de una estructura que lo permita*, como suele ser otro objeto. Un spread aplicado sobre un objeto presentaría cada par de *clave-valor* separado por comas, y ésto en una función no sería admisible, pero sí puedo serlo dentro de otro objeto.

Esto suele ser útil cuando queremos replicar o modificar estructuras de objetos, ya que nos permite primero listar todas sus propiedades y valores y luego modificar/agregar las que queramos:

| const usuario1 = {  nombre: "Juan",  edad: 24,  curso: "Javascript"  }  // lista todas las propiedades y valores de usuario1 dentro de otro objeto  const usuario2 = {  ...usuario1  }  console.log(usuario2) // { nombre: 'Juan', edad: 24, curso: 'Javascript' }  const usuario3 = {  ...usuario1,  curso: "ReactJS",  email: "juan@doe.com"  }  console.log(usuario3)  // { nombre: 'Juan', edad: 24, curso: 'ReactJS', email: 'juan@doe.com' } |
| --- |

En el último ejemplo vemos que agregamos una propiedad y que *modificamos* la propiedad curso. Recordemos que no podemos tener dos propiedades con el mismo nombre, y en tal caso prevalece la última declarada, que es lo que ocurre aquí. El spread de usuario1 lista todas las propiedades, incluida *curso*, dentro de *usuario3*, y luego la sobreescribimos con un nuevo valor.

**Rest Parameters**

El operador *spread* también puede utilizarse dentro de la declaración de una función para indicar que queremos recibir una **cantidad indeterminada de parámetros**.

Supongamos que quiero tener una función para sumar *cualquier cantidad de números* que reciba por parámetro.

Puedo hacer esto con el operador spread definiendo *rest parameters*, lo que significa que mi función va a recibir una cantidad indeterminada de parámetros, pero los va a agrupar dentro de un *array* con el nombre que defina, y con eso trabajará dentro:

| function sumar(...numeros) {  console.log(numeros)  }  sumar(4, 2) // [ 4, 2 ]  sumar(10, 15, 30, 5) // [ 10, 15, 30, 5 ] |
| --- |

Vemos que con esta sintaxis el parámetro *…numeros* se define como un **array** donde se guardan todos los argumentos enviados que coincidan con esa posición.

De esta forma podemos escribir funciones que reciban múltiples parámetros, sin saber con precisión cuántos serán, pudiendo trabajarlos luego como un array dentro de la función.

Siguiendo el ejemplo anterior, podemos tomar este array *numeros* y retornar la suma de todos los elementos que reciba con un *reduce:*

| *function sumar(...numeros) {*  *return numeros.reduce((acc, n) => acc + n, 0)*  *}*  *console.log( sumar(4, 2) ) // 6*  *console.log( sumar(10, 15, 30, 5) ) // 60*  *console.log( sumar(100, 300, 50) ) // 450* |
| --- |