

# JavaScript Basic

### Guía #2: Advanced JavaScript

Esta guía hace parte del curso mencionado en el título y les permitirá estudiar con más detenimiento los conceptos de clase, además de servir como instrumento de la misma. La guía contará con código base, el desarrollo conceptual de los temas y enlaces externos para el estudio independiente.

## 1. Asignaciones por destructuración

Gracias a la destructuración es posible realizar la extracción de datos de estructuras como arreglos y objetos con una sintaxis equivalente a la construcción de las mismas.

### 1.1 Destructuración de arreglos

var starks = ["Arya", "Bran", "Sansa"];  
   
 //forma tradicional  
 var azhorAhai = starks[0];  
 var cuervoTresOjos = starks[1];  
 var esa = starks[2];  
   
 //con destructuración  
 var [azhorAhai, cuervoTresOjos, esa] = starks;

La destructuración no sólo funciona para ahorrar líneas de código…

### Ejercicio:

Escriba una función que permita intercambiar los valores de dos variables de tal forma que:

var a = 2;  
 var b = 5;  
   
 //f(a,b) Output: b = 2 & a = 5



### Una posible solución:

function swap(a,b){  
 let temp = a;  
 a = b;  
 b = temp;  
 let retorno = [a,b];  
 return retorno;  
 }

Ahora si aprovechamos al máximo la destructuración…



### Una solución Full pros JavaScript

function swap(a,b){  
 return [a,b] = [b,a]; //sin variable temporal y con múltiple retorno.  
 }  
   
 [a,b] = swap(a,b); //Claramente siendo tan pros ya no haría falta la función swap.

**Nota:** podemos ignorar elementos de un múltiple retorno [a,] = swap(a,b); sólo asignará a la variable a el primer retorno de la función.

### 1.2 Destructuración de objetos

Básicamente como la de arreglos, pero con posibilidades más interesantes que permitirán adicionar semántica al código.

A continuación usaremos un fragmento del dataset de las casas de [westeros](https://awoiaf.westeros.org/index.php/Houses_of_Westeros) para usar la destructuración de objetos:

const westeros = {  
 houses: {  
 crownlands: {  
 royal: [  
 "Baratheon"  
 ],  
 noble: [  
 "Blount","Buckwell","Byrch","Bywater","Chelsted","Chyttering","Cressey","Darke","Edgerton","Farring","Follard","Gaunt","Harte","Hayford","Langward","Mallery","Manning","Massey","Pyle","Rambton","Rollingford","Rosby","Rykker","Staunton","Stokeworth","Thorne","Wendwater"  
 ],  
 narrowSea: [  
 "Baratheon of Dragonstone","Bar Emmon", "Celtigar", "Sunglass", "Velaryon"  
 ],  
 crackclawPoint:[  
 "Boggs", "Brune of Brownhollow", "Brune of the Dyre Den", "Cave", "Crabb", "Hardy", "Pyne"  
 ],  
 other:[  
 "Dargood", "Darkwood", "Hollard", "Longwaters"  
 ],  
 exhiled:[  
 "Targaryen"  
 ],  
 extinct:[  
 "Blackfyre", "Cargyll", "Darklyn"  
 ]  
 },  
 north:{  
 great:[  
 "startk"  
 ],  
 noble:[  
 "Ashwood","Bole","Bolton","Branch","Cassel","Cerwyn","Condon","Dustin","Flint of Flint's Finger","Flint of Widow's Watch","Forrester","Glenmore","Holt","Hornwood","Ironsmith","Karstark","Lake","Lightfoot","Locke","Long","Manderly","Marsh","Mollen","Mormont","Moss","Overton","Poole","Reed","Ryswell","Slate","Stout","Umber","Waterman","Wells","Whitehill","Woods","Woolfield"  
 ]  
 }  
 }  
 }

Supongamos que requerimos trabajar con propiedades separadas del objeto, en este caso queremos extraer las casas nobles del norte y de las tierras de la corona respectivamente:

var {north, crownlands} = westeros.houses;  
   
 console.log("The North houses: ",north);  
 console.log("The Crownland houses: ",crownlands);

### Renombramiento mediante destructuración

Ahora supongamos que queremos usar nombres menos apropiados y más comúnes para los demás desarrolladores que trabajen en el proyecto:

var {north: northerners, crownlands: southerners} = westeros.houses;  
   
 console.log("The North houses: ", northerners);  
 console.log("The Crownland houses: ",southerners);

Así mismo podremos usarlo dentro de ciclos for, entre otras funcionalidades, para más información click [acá](https://developer.mozilla.org/es/docs/Web/JavaScript/Referencia/Operadores/Destructuring_assignment#For_para_iteraciones_con_destructuring)

## 2. Operadores de propagación (Spread Operators)

Nos permiten expandir expresiones que podrían esperar múltiples argumentos o elementos.

### 2.1 Arreglos literales

Supongamos que tenemos el siguiente arreglo:

let avengers = ["Iron man", "Thor", "Hulk", "El Capi"]

, y ahora queremos adicionar de la manera menos compleja el arreglo

let guardianes = ["Peter Quill", "Gamora", "Drax el destructor", "Rocket", "Groot"]

justo después de Thor. ¿Cómo lo harías?



Usemos operadores de propagación:

let avengers = ["Iron man", "Thor", "Hulk", "El Capi"];  
 let guardianes = ["Peter Quill", "Gamora", "Drax el destructor", "Rocket", "Groot"]  
   
 avengers = [avengers[0], avengers[1], ...guardianes, avengers[2]];  
 console.log(avengers);

Y sí, también podemos esto para pasar parámetros a una función o a la misma creación de un objeto.

## 3. Operadores de filtro, mapeo, reducción e iteración

A continuación hablaremos de métodos importantes para hacernos la vida más fácil con arreglos y objetos.

### 3.1 Método [filter()](https://developer.mozilla.org/es/docs/Web/JavaScript/Referencia/Objetos_globales/Array/filter)

Con este método podremos crear un nuevo arreglo que contenga todos los elementos que cumplen con la condición implementada dentro del bloque.

let guardianes = ["Peter Quill", "Gamora", "Drax el destructor", "Rocket", "Groot"];  
   
 const resultado = guardianes.filter(guardian => guardian != "Gamora");  
   
 console.log(resultado);

Ahora hagamos algo de gente grande con el método filter, desarrollemos una función que nos permita hacer búsqueda, en este caso para que se retorne un vengador que coincida con la query:

let avengers = ["Iron man", "Thor", "Hulk", "El Capi"];  
  
/\*\*  
 \*   
 \* @param query string   
 \* @param elements arreglo en donde vamos a buscar  
 \*/  
function search(query,elements) {  
 //retornamos el arreglo resultante del filtro  
 return elements.filter(function(el) {  
 //'el' es en el caso de un arreglo ["a","b","c"] un string  
 //String.indexOF => https://developer.mozilla.org/es/docs/Web/JavaScript/Referencia/Objetos\_globales/String/indexOf  
 return el.toLowerCase().indexOf(query.toLowerCase()) > -1;  
 })  
 }  
   
 console.log(search("o",avengers));  
 //Output: [ 'Iron man', 'Thor' ]

### 3.2 Método [map()](https://developer.mozilla.org/es/docs/Web/JavaScript/Referencia/Objetos_globales/Array/map)

El método map retorna un arreglo con los resultados de la aplicación de la función provista a cada elemento.

Ahora nuestro propósito es realizar operaciones con cada elemento de un conjunto y obtener el resultado sin alterar el conjunto original, y es acá en donde entra Array.map().

let avengers = ["Iron man", "Thor", "Hulk", "El Capi"];  
 //vamos a crear a los avengers recargados  
 let supaAvengers = avengers.map((avenger) => {  
 return `${avenger} recargado`;  
 })  
   
 console.log(supaAvengers);  
 /\* Output:  
 [ 'Iron man recargado',  
 'Thor recargado',  
 'Hulk recargado',  
 'El Capi recargado' ]  
 \*/

Ahora otro ejemplo del mundo real, replanteemos un conjunto de datos recibidos de la forma:

let users = [  
 { id: 1234, username: "cperdio" },  
 { id: 2345, username: "nontendio" },  
 { id: 3456, username: "lagarro" },  
 { id: 4567, username: "admin" }  
 ];  
   
 let orderedUsers = users.map((user)=>{  
 let oUser = {};  
 oUser[user.id] = user.username;  
 return oUser;  
 });

### 3.3 Método [reduce()](https://developer.mozilla.org/es/docs/Web/JavaScript/Referencia/Objetos_globales/Array/reduce)

El método reduce() nos permite aplicar una función a un acumulador y los elementos del arreglo hasta reducirlo a un único valor.

let r = [0,1,2,3,4].reduce(function(valorAnterior, valorActual, indice, vector){  
 console.log(valorAnterior);  
 console.log(valorActual);  
 console.log(indice);  
 console.log(vector);  
 return valorAnterior + valorActual;  
 });  
   
 console.log(r);  
 //Output: 10