



*JavaScript: Arrays*

**Arrays: definición**

Los **arrays** (también llamados **colecciones**) son un tipo de objeto especial que nos permite agrupar elementos. Se utilizan cuando necesitamos contar con un conjunto de valores asociados a un listado. En estas estructuras, podemos guardar cualquier tipo de dato u objeto. Es decir, es posible crear arrays de números, cadenas, booleanos, objetos e incluso un array compuesto por otros arrays ([matrices](https://developer.mozilla.org/es/docs/Learn/JavaScript/First_steps/Arrays)).

Si bien el array es un objeto, para crear una estructura de este tipo no es necesario emplear el constructor (por ejemplo: new Array()), sino que nos basta con usar los corchetes y especificar los elementos que componen la lista separados por coma (,), como podemos observar en el ejemplo:

| // Declaraciòn de array vacío  const arrayA = [];  // Declaración de array con nùmeros  const arrayB = [1,2];  // Declaracion de array con strings  const arrayC = ["C1","C2","C3"];  // Declaración de array con booleanos  const arrayD = [true,false,true,false];  // Declaración de array mixto  const arrayE = [1,false,"C4"]; |
| --- |

Dado que el array es un objeto, es preferible declarar la variable a la que se asigna esta estructura con la palabra reservada ***const***, para evitar una posible sobre-escritura de la referencia.

Por otro lado, es importante determinar que los arrays pueden contener elementos **heterogéneos**, siendo necesario un control por parte del/la programador/a sobre el tipo de elemento a incluir en la colección, con la intención de evitar problemas de procesamiento al momento de operar con elementos de distinto tipo.

Si el array es **homogéneo**, es decir, todos sus elementos tienen el mismo tipo de dato, no se presenta inconveniente alguno al momento de hacer operaciones entre los valores de la lista. Podemos observar esto en el siguiente array numérico:

| const numeros = [1,2,3,4,5];  let resultado1 = numeros[0] + numeros[2]; // 1 + 3 = 4;  let resultado2 = numeros[1] + numeros[4]; // 2 + 5 = 7;  let resultado3 = numeros[1] + numeros[1]; // 2 + 2 = 4 |
| --- |

Para acceder a un elemento de array, tenemos que emplear su posición o **índice**. Dado que los elementos de estas estructuras tienen un orden, para el ejemplo podríamos identificarlo a simple vista de la siguiente manera;

* 1 es el primer elemento del array.
* 2 es el segundo elemento del array.
* 3 es el tercer elemento del array.
* 4 es el cuarto elemento del array.
* 5 el quinto elemento del array.

Sin embargo, lo anterior sería una interpretación incompleta para determinar las posiciones reales de los elementos en la lista, porque en los arrays **las posiciones inician desde el número 0**. Entonces, para identificar el ordenamiento correcto decimos:

* 1 está en la posición 0 del array.
* 2 está en la posición 1 del array.
* 3 está en la posición 2 del array.
* 4 está en la posición 3 del array.
* 5 está en la posición 4 del array.

Ahora que tenemos correctamente identificadas las posiciones de los elementos, para acceder a cada uno empleamos el identificador del array seguido de corchetes, y las posiciones entre estos; por ejemplo: numeros[0] ,numeros[1] ,numeros[2], numeros[3] y numeros[4].

También es posible emplear la estructura **for** para acceder a un elemento del array en cada ciclo. A esta forma de acceso a los elementos se la conoce con el nombre de **“recorrido del array”**:

| const numeros = [1, 2, 3, 4, 5];  for (let index = 0; index < 5; index++) {  alert(numeros[index]);  } |
| --- |

Podemos acceder a todos los elementos en este caso porque hacemos coincidir el valor de la variable de conteo (index) con una posición válida del array. Como el **hasta** (index < 5) implica iterar hasta que index tenga un valor igual a 4, y de 0 a 4 existen cinco posiciones en la lista, se puede obtener elemento a elemento sin inconvenientes. En el caso de que quiera acceder con un índice de valor 5 al array, y no existe un elemento para dicha posición, se obtiene [undefined](https://developer.mozilla.org/es/docs/Web/JavaScript/Referencia/Objetos_globales/undefined).

Recorrer un array de principio a fin es una de las acciones más utilizadas en programación, ya que es la principal ventaja que nos ofrece el tener ordenados en un listado (array) nuestros datos. Hay diversas acciones que podemos aplicar sobre los elementos de un array recorriéndolo, incluidas acciones de búsqueda o filtración sobre el mismo.

Con el caso anterior, definimos explícitamente el **hasta** del iterador porque sabemos que consta de 5 elementos en total, pero por lo general no sabemos de forma determinada cuál es el límite de nuestro array. Por ello, recurrimos a la propiedad **length** del array que nos retorna su **longitud**, es decir la cantidad de elementos que contiene. Este valor se utiliza como límite de nuestro iterador y nos garantiza poder recorrer un array de principio a fin, sin saber de forma explícita cuántos elementos contiene:

| const numeros = [1, 2, 3, 4, 5];  for (let index = 0; index < numeros.length; index++) {  alert(numeros[index]);  } |
| --- |

En este caso, **numeros.length** retorna el valor de 5, que es la longitud del array y funciona exactamente como el límite para el iterador.

**Métodos comunes**

Como todo objeto en JavaScript, array tiene propiedades y métodos que podemos utilizar para solucionar ciertas situaciones, hacer transformaciones de los valores o realizar búsquedas y filtros.

En esta sección presentaremos los métodos más utilizados. Para conocer otras herramientas pueden verificar la especificación del objeto array en la [documentación](https://developer.mozilla.org/es/docs/Web/JavaScript/Referencia/Objetos_globales/Array).

**toString**

El método **toString** convierte un array en un string, compuesto por cada uno de los elementos del array separados por comas. Es muy útil cuando queremos incluir el detalle de los elementos del array en una salida (**console.log** o **alert**). No obstante, hay que tener en cuenta que esta transformación no se realiza adecuadamente cuando hacemos toString a un array de objetos personalizados. En el siguiente ejemplo vemos cómo podemos emplear el métodos toString para obtener una cadena desde un valor mixto:

| const miArray = ["marca", 3 ,"palabra"];  console.log( miArray.toString() );//imprime "marca,3,palabra" |
| --- |

**Push**

Este método se utiliza para sumar un elemento a un array ya existente, pasando como parámetro el valor (o variable) a agregar. Dicho elemento se agregaría al final del listado, pudiendo enviar cualquier tipo de dato como parámetro al método.

Al igual que en un string, la propiedad length nos sirve para obtener el largo de un array, es decir, cuántos elementos tiene. Cada vez que realizamos **push** de un elemento, el length del array aumenta en uno:

| const miArray = ["marca", 3, "palabra"]  miArray.push('otro elemento')  console.log(miArray.length) // --> 4  //El array ahora tiene 4 posiciones  console.log(miArray)  //["marca", 3, "palabra", “otro elemento”] |
| --- |

**Unshift**

Similar al push, el método **unshift()** nos permite agregar un elemento al inicio de un array, sobre su primera posición. Cabe destacar que cuando agregamos elementos al inicio de un array terminamos *corriendo* (o empujando) el índice de todos los elementos que había, ya que todos se corren un lugar en el array; no así cuando utilizamos push(), donde los elementos previos al agregado se mantienen en su posición:

| const miArray = ["marca", 3, "palabra"]  console.log(miArray[1]) // --> 3  miArray.unshift('otro elemento')  console.log(miArray[1]) // --> “marca”  console.log(miArray)  //[“otro elemento”, "marca", 3, "palabra"] |
| --- |

Los métodos de **push** y **unshift** pueden recibir varios parámetros, siendo éstos la cantidad de nuevos elementos que quiero agregar al array en el inicio o final:

| const nombres = ["Ana", "Julia"]  nombres.push("Juan", "Carlos")  console.log(miArray) // ["Ana", "Julia", "Juan", "Carlos"]  nombres.unshift("Luz", "Clara", “Ricardo”)  console.log(nombres)  // ["Luz", "Clara", “Ricardo”, "Ana", "Julia", "Juan", "Carlos"] |
| --- |

**Pop** y **Shift**

De manera inversa a los anteriores, tenemos métodos para eliminar un elemento en el inicio o el final del array. El método **pop()** elimina el último elemento de un array, no importa cuál fuese, mientras que el método **unshift()** elimina siempre el primer elemento en el array:

| const nombres = ["Luis", "Ana", "Julia", “Juan”]  nombres.pop()  console.log(nombres) // ["Luis", "Ana", "Julia"]  nombres.shift()  console.log(nombres) // ["Ana", "Julia"] |
| --- |

**Join**

Mediante este método podemos juntar todos los elementos de un array en una cadena string, indicando como parámetro el separador para esos elementos. Al igual que toString(), la nueva cadena creada comprende todos los elementos del array, con la diferencia de que en vez de estar separados por coma, se utiliza la subcadena por parámetro (o delimitador) para separar los elementos en la cadena resultante:

| const otroArray = ["hola", 22, "mundo"];  console.log(otroArray.join("\*")); //Imprime "hola\*22\*mundo" |
| --- |

Hay que tener en cuenta que esta transformación, al igual que en toString, no se realiza adecuadamente cuando hacemos **join** a un array de objetos personalizados.

**Concat**

A veces tenemos la necesidad de unir dos arrays, es decir, crear un nuevo array desde los elementos de dos arrays previamente declarados. Esto lo podemos hacer mediante el método **concat**, que nos retorna un único array compuesto por los elementos de ambos.

| const perros = ["Pupy", “Ronnie”]  const gatos = ["Mishi", “Garfield”, "Zuri"]  const mascotas = perros.concat(gatos)  console.log(mascotas)  // ["Pupy", “Ronnie”, "Mishi", “Garfield”, "Zuri"] |
| --- |

Es importante identificar que llamamos al método concat desde un array pasando por parámetro la segunda lista. Esta operación no es conmutativa, lo cual implica que los elementos del array desde el cual se llama al método estarán primero que los del array enviado por parámetro.

**Slice**

Este método retorna un nuevo array, que representa una copia de una parte de otro, un corte del array. Como parámetros, se envían dos índices: primero un **índice inclusivo**, que significa la posición desde donde iniciamos el corte, y segundo un **índice excluyente**que significa el final del corte:

| const nombres = ['Rita', 'Pedro', 'Miguel', 'Ana', 'Vanesa'];  const masculinos = nombres.slice(1, 3);  // array desde la posición 1 (inclusivo) a 3 (excluyente)  // es decir, elementos en posición 1 y 2  // masculinos contiene ['Pedro','Miguel'] |
| --- |

En este caso, el nuevo array se genera desde la posición 1 a la 3, pero lo que hay que tener en cuenta es que la segunda posición enviada no se incluye en el array retornado. Es decir que el valor en la posición 3 no forma parte de la copia.

Hay que observar que en ningún momento se está modificando el array **nombres**, sino que se crea un nuevo array que asignamos a la variable **masculinos**. Pero el array original sigue conteniendo los mismos valores. Para eliminar un elemento de un array podemos usar el método [splice](https://developer.mozilla.org/es/docs/Web/JavaScript/Referencia/Objetos_globales/Array/splice).

**Splice**

El método **splice()** nos permite eliminar uno o varios elementos de un array en cualquier posición (a diferencia del pop o shift que sólo eliminan al inicio o final). Este método modifica el array original sobre el cuál lo llamo. Splice recibe en principio 2 parámetros: el primero es el **índice** donde se ubica el método para trabajar, y el segundo es la **cantidad de elementos a eliminar** desde esa posición:

| const nombres = ['Rita', 'Pedro', 'Miguel', 'Ana', 'Vanesa'];  nombres.splice(1, 2)  console.log(nombres)  // ['Rita', 'Ana', 'Vanesa'] |
| --- |

En este ejemplo le digo al método **splice(1, 2)**,que se posicione en el índice 1 del array y desde allí elimine 2 elementos. Esto termina eliminando los elementos en la posición 1 y 2 del array. Si llamo al método con los valores **splice(3, 1)** terminaría eliminando un solo elemento en el índice 3:

| const nombres = ['Rita', 'Pedro', 'Miguel', 'Ana', 'Vanesa'];  nombres.splice(3, 1)  console.log(nombres)  // ['Rita', 'Pedro', 'Miguel', 'Vanesa'] |
| --- |

El método splice me sirve para eliminar cualquier elemento del array siempre y cuando conozca su índice.

**indexOf**

El método **indexOf()** nos permite obtener el índice de un elemento en un array siempre y cuando éste exista en el mismo. Recibe por parámetro el elemento que queremos buscar en el array y de existir nos retorna su índice. En el caso de que el elemento no exista, nos retornará como valor un -1 que significa que no se encontró el elemento buscado:

| const nombres = ['Rita', 'Pedro', 'Miguel', 'Ana', 'Vanesa'];  console.log( nombres.indexOf('Rita') ) // ⇒ 0  console.log( nombres.indexOf('Miguel') ) // ⇒ 2  console.log( nombres.indexOf('Ana') ) // ⇒ 3  console.log( nombres.indexOf('Julieta') ) // ⇒ -1 |
| --- |

Sabiendo que si el valor que le paso no se encuentra en el array obtengo un -1, puedo usar esto para condicionar una función que me permita eliminar cualquier elemento del array pasándolo por parámetro, en una combinación del método **indexOf()** con el método **splice():**

| const nombres = ['Rita', 'Pedro', 'Miguel', 'Ana', 'Vanesa']  const eliminar = (nombre) => {  let index = nombres.indexOf(nombre)  if (index != -1 ) {  nombres.splice(index, 1)  }  } |
| --- |

En este ejemplo escribo una función que reciba un *nombre* por parámetro. Hago un **indexOf** sobre el array de ese nombre para obtener el índice del mismo en el array. Si el index obtenido es diferente de -1 (lo que significa que existe el elemento en el array) hago un **splice** con ese índice y borro un elemento desde esa posición; o sea borro exactamente el elemento que pasé por parámetro.

**includes**

Similar al anterior, el método **includes** me permite saber si un elemento que paso por parámetro existe o no dentro de un array, retornando un valor booleano en caso afirmativo o negativo:

| const nombres = ['Rita', 'Pedro', 'Miguel', 'Ana', 'Vanesa']  console.log( nombres.includes('Rita') ) // ⇒ true  console.log( nombres.includes('Miguel') ) // ⇒ true  console.log( nombres.includes('Julieta') ) // ⇒ false |
| --- |

**reverse**

Como su nombre lo indica, el método **reverse()** invierte el orden de los elementos dentro de un array. De todas maneras, es necesario ser cuidadosos con este método porque es destructivo, es decir, modifica el array original:

| const nombres = ['Rita', 'Pedro', 'Miguel', 'Ana', 'Vanesa']  nombres.reverse()  console.log( nombres )  // ⇒ ['Vanesa', 'Ana','Miguel', 'Pedro','Rita'] |
| --- |

**Usando métodos de array**

La mayoría de los métodos de array abordados en la sección anterior se enfocan en generar nuevos resultados basados en la información del array: strings que detallan los elementos de la lista con un separador, la unión de dos array, o un nuevo array más pequeño que el original.

Es necesario realizar estas operaciones con la intención de generar entradas y salidas adecuadas. Veamos un ejemplo de cómo los métodos de array nos permiten enriquecer la información que brindamos al usuario:

| //Declaraciòn de array vacío y variable para determinar cantidad  const listaNombres = [];  let cantidad = 5;  //Empleo de do...while para cargar nombres en el array por prompt()  do{  let entrada = prompt("Ingresar nombre");  listaNombres.push(entrada.toUpperCase());  console.log(listaNombres.length);  }while(listaNombres.length != cantidad)  //Concatenamos un nuevo array de dos elementos  const nuevaLista = listaNombres.concat(["ANA","EMA"]);  //Salida con salto de línea usando join  alert(nuevaLista.join("\n")); |
| --- |

En el ejemplo se emplean las siguientes herramientas del objeto array para añadir funcionalidad:

* Usamos la propiedad **length** del array para determinar cuántos elementos tenemos cargados.
* El método **push** nos permite añadir al array la entrada ingresada por el usuario, previa transformación del string a mayúscula con su método toUpperCase.
* Con **concat** creamos una nueva colección con dos elementos adicionales. El parámetro puede ser un array creado de forma explícita: [“ANA”, “EMA”];
* El método **join** nos permite agregar el mensaje al alert, empleando "\n" como delimitador, para generar un salto de línea entre cada elemento del array, otorgando así al usuario una salida más apropiada para un listado de nombre.

*Arrays de objetos*

Mencionamos en la sección anterior que existen arrays de objetos personalizados, que son un listado de objetos creados por el usuario, ya sea de forma explícita usando objetos literales, o instanciándolos con un constructor. Veamos un caso de declaración con objetos literales:

| const objeto1 = { id: 1, producto: "Arroz" };  const array = [objeto1, { id: 2, producto: "Fideo" }];  array.push({ id: 3, producto: "Pan" }); |
| --- |

Los arrays de objetos a veces implican un tratamiento especial. Como vimos, toString y join son métodos que no funcionan correctamente para visualizar su información, y esto se debe a que cuando buscamos transformar la información de un objeto a string, por defecto se obtiene la cadena [[object Object]](https://developer.mozilla.org/es/docs/Web/JavaScript/Referencia/Objetos_globales/Object/toString). Si bien, más adelante evaluaremos técnicas de transformación para obtener correctamente la información de un objeto en formato string, la primera herramienta que se puede utilizar para acceder a la información de cada objeto en un array es el bucle **for...of**, que permite realizar el recorrido de los objetos.

**Recorriendo un array**

La sentencia **for...of** permite recorrer un [objeto iterable](https://es.javascript.info/iterable) (array), ejecutando un bloque de código por cada elemento del objeto. Para el caso de un array de objeto, la declaración de dicha estructura podría tener la siguiente forma:

| const productos = [{ id: 1, producto: "Arroz" },  { id: 2, producto: "Fideo" },  { id: 3, producto: "Pan" }];  for (const producto of productos) {  console.log(producto.id);  console.log(producto.producto);  } |
| --- |

Por cada ciclo del **for...of,** obtenemos la referencia a un elemento del array en orden. Es decir, accedemos de uno en uno a todos los elementos del array. El bucle termina una vez iterado el último de la lista. La referencia al elemento actual la tenemos en la variable creada antes de la palabra reservada **of**.

Es decir, el iterador for… of tiene la siguiente estructura:

| for (const *referencia* of *array*) { … } |
| --- |

Es un iterador que recorre el array de principio a fin, y en cada iteración accedemos al elemento en cuestión a través de la *referencia* que declaramos. Por cada iteración se ejecuta el bloque de código que definimos entre llaves.

Como es un array de objetos, podemos acceder a su estructura con las dos formas de acceso: **identificador.propiedad** (Ejemplo producto.id) o *identificador[“propiedad”]* (por ejemplo producto[“id”]).

Si el objeto tiene comportamiento, podemos emplear **for...of** para llamar a uno o más métodos pertenecientes a los objetos que componen el array, como podemos ver a continuación:

| class Producto {  constructor(*nombre*, *precio*) {  *this*.nombre = *nombre*.toUpperCase();  *this*.precio = parseFloat(*precio*);  *this*.vendido = false;  }  sumaIva() {  *this*.precio = *this*.precio \* 1.21;  }  }  //Declaramos un array de productos para almacenar objetos  const productos = [];  productos.push(**new** Producto("arroz", "125"));  productos.push(**new** Producto("fideo", "70"));  productos.push(**new** Producto("pan", "50"));  //Iteramos el array con for...of para modificarlos a todos  for (const producto of productos)  producto.sumaIva(); |
| --- |

También podemos anidar un [for...in](https://developer.mozilla.org/es/docs/Web/JavaScript/Referencia/Sentencias/for...in) en el **for...of** para recorrer todas las propiedades del objeto actual. Además, es posible emplear las ya conocidas sentencias break o continue con **for...of**.