

EXERCISES QUE TRABAJAREMOS EN EL CUE

- EXERCISE 1: CREACIÓN DE UNA MATRIZ Y COMPRENSIÓN DE SU ITERADOR.
- EXERCISE 2: MÉTODOS DEL OBJETO ARRAY.
- EXERCISE 3: ÁLGEBRA CON ARRAYS.

EXERCISE 1: CREACIÓN DE UNA MATRIZ Y COMPRENSIÓN DE SU ITERADOR

Cómo fue establecido en Text Class Review, las matrices nos permiten almacenar más de un valor en la misma variable. Ahora, ¿cómo se escriben?

Para crear una matriz, primero se debe declarar la variable que almacenará este *Array* (nombre en inglés). Por lo tanto, empezamos con la palabra clave **var**, seguida por el nombre de la variable, y esto se iguala a los valores que pondremos separados por comas dentro de 2 brackets “[]”. A continuación, podrás ver un ejemplo:

```
1 //Declaración de una Array:
2 var nombreArray = ['dato1', 087243, false, 'dato4', datoObjeto];
```

Si se nota, los datos que contiene una matriz pueden ser de cualquier *tipo*, pero todos deben estar separados por comas.

En las matrices, nosotros identificamos cada elemento que contiene mediante su posición o *índice*. Este índice siempre parte con el valor de cero, es decir, el primer elemento de nuestro Array siempre tendrá el índice de cero.

```
1 //Declaración de una Array:
2 var nombreArray = [
3   'dato1', // índice 0.
4   087243, // índice 1.
5   false, // índice 2.
6   'dato4', // índice 3.
7   datoObjeto // índice 4.
8 ];
```

Si lo observas, lo único que hicimos fue desplegar los datos de nuestra matriz de manera vertical, para poder comentar al lado el índice que le corresponde a cada posición. La posición uno, siempre tiene un

índice de cero, y las posiciones sucesivas van incrementando su valor. Ahora, ¿cómo podemos acceder a un elemento específico de nuestra matriz? Para realizar esto, debemos utilizar el índice del elemento.

Supongamos que tenemos el siguiente Array de nombre refrigerador, el cual contiene distintos alimentos:

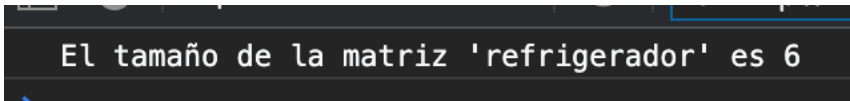
```
1 var refrigerador = ['Huevos', 'Mantequilla', 'Leche', 'Verduras', 'Carne',  
2 'Jugo'];  
3 // 6 elementos
```

A simple vista, nosotros podemos detectar que hay 6 elementos en esta matriz, pero ¿qué tendríamos que hacer si la matriz es de 300 elementos, o incluso más? En ese caso, se usará una propiedad para determinar el tamaño del Array, esta es `length`.

Mediante el siguiente código, obtendremos el valor del Array:

```
1 // Tamaño del array:  
2 console.log(`El tamaño de la matriz 'refrigerador' es  
3 ${refrigerador.length}`);
```

La consola nos retorna el siguiente valor:



```
El tamaño de la matriz 'refrigerador' es 6
```

Para acceder a un elemento específico de la matriz, debemos escribir su nombre, seguida por dos brackets, y dentro de éstos, pasamos el índice que indica la posición del elemento. A continuación, mostraremos cómo acceder a todos los elementos de la matriz refrigerador:

```
1 // Accediendo a elementos índice:  
2 console.log(`Índice 0 es ${refrigerador[0]}`);  
3 console.log(`Índice 1 es ${refrigerador[1]}`);  
4 console.log(`Índice 2 es ${refrigerador[2]}`);  
5 console.log(`Índice 3 es ${refrigerador[3]}`);  
6 console.log(`Índice 4 es ${refrigerador[4]}`);  
7 console.log(`Índice 5 es ${refrigerador[5]}`);
```

Y por consola recibimos los siguientes valores, los cuales indican la posición y el valor del elemento dentro de la matriz:


```
Índice 0 es Huevos  
Índice 1 es Mantequilla  
Índice 2 es Leche  
Índice 3 es Verduras  
Índice 4 es Carne  
Índice 5 es Jugo
```

Pero ¿qué sucedería si tratáramos de acceder a un valor que no existe en nuestro Array? Los valores no existentes resultan ser indefinidos, por defecto. Si deseamos comprobar esto, podemos tratar de acceder al siguiente índice de nuestra matriz refrigerador:

```
1 console.log(`Índice 6 es ${refrigerador[6]}`);
```

En la consola esto resulta así:

```
Índice 3 es Verduras  
Índice 4 es Carne  
Índice 5 es Jugo  
Índice 6 es undefined
```



Como puedes ver, los índices fuera del rango de los que existen dentro de un Array son indefinidos.

Una manera más práctica para acceder a todos los índices de una matriz es utilizando un ciclo **for** para iterar sobre el largo de ésta. Observa en el siguiente ejemplo:

```
1 // Iteramos sobre el largo del array  
2 for (let i = 0; i < refrigerador.length; i++) {
```

```
3 console.log(`Valor del índice ${i} es '${refrigerador[i]}'\`);
4 }
```

Entonces, aquí iteramos sobre el largo del Array **refrigerador**, y mediante la consola mostramos el valor del índice y el elemento en esa posición:

```
Valor del índice 0 es 'Huevos'
Valor del índice 1 es 'Mantequilla'
Valor del índice 2 es 'Leche'
Valor del índice 3 es 'Verduras'
Valor del índice 4 es 'Carne'
Valor del índice 5 es 'Jugo'
```

Cómo puedes ver, esto es una manera mucho más sencilla de lograr el mismo objetivo, y de forma dinámica, dado que anteriormente teníamos que escribir línea por línea todos los valores, sin embargo, con este bucle podemos iterar sobre el largo completo de una matriz, sea de 3 elementos o de 3000.

Una manera alternativa de acceder a un índice en específico, involucra emplear la propiedad que habíamos analizado anteriormente, **length**. Por ejemplo:

```
1 // Accediendo a elementos índice:
2 console.log(`Vamos a acceder al penúltimo elemento:
3 ${refrigerador[refrigerador.length - 2]}`);
```

Por consola:

```
Vamos a acceder al penúltimo elemento: Carne
```

¿Y si sabemos el elemento que queremos buscar, pero no recordamos el índice? ¿Qué se puede hacer?

En este caso, debemos usar el método `indexOf()`, que por parámetro recibe un String del valor del elemento que buscamos dentro del Array, y nos retorna el índice donde se encuentra. Note la siguiente implementación de este método, donde nosotros queremos saber el índice dónde se ubica el elemento "verduras".

```
1 var refrigerador = ['Huevos', 'Mantequilla', 'Leche', 'Verduras', 'Carne',  
2 'Jugo'];  
3 //Verduras se encuentra en el índice 3.  
4  
5 //Buscamos el índice:  
6 console.log(`Verduras se encuentra en el índice  
7 '${refrigerador.indexOf('Verduras')}'`);  
8
```

Por consola, recibimos la respuesta del método `indexOf()`:

```
Verduras se encuentra en el índice '3'
```

Hasta el momento hemos analizado unas propiedades sobre el objeto Array, y en el siguiente ejemplo estudiaremos con más detalle los métodos asociados a este objeto.

EXERCISE 2: MÉTODOS DEL OBJETO ARRAY

Ya hemos analizado unas propiedades sobre el objeto Array en JS. Éste también tiene métodos definidos, los cuales nos ayudan a interactuar con las matrices.

Dentro de la gran gama de métodos que tiene este objeto, vamos a definir dos que sirven para interactuar con los últimos elementos de nuestro Array, y posteriormente, dos más para interactuar con el primer elemento de nuestro objeto. Empezaremos viendo cómo interactuar con los últimos elementos de nuestro Array.

Para eliminar un elemento del final de una matriz, utilizamos el método `pop()`. Notemos cómo lo aplicamos en el siguiente caso:

```
1 var closet = ['abrigos', 'camisas', 'poleras', 'zapatos', 'zapatillas',
2 'pantalones', 'traje'];
3 console.log(closet);
4
5 // Eliminar un elemento del final de una matriz ( pop() )
6 console.log(`Eliminamos último elemento: '${closet.pop()}'`);
7 console.log(closet);
```

Por consola podemos ver todos los elementos de la matriz antes y después de aplicarle el método `pop`. Si nos fijamos, los números entre paréntesis al inicio de los objetos, indican el tamaño o cantidad de elementos dentro de la matriz. La flecha roja indica cómo disminuyó la cantidad de 7 a 6.



```
script.js:2 (7) ["abrigos", "camisas", "poleras", "zapatos", "zapatillas", "pantalones", "traje"]
Eliminamos último elemento: 'traje' script.js:5
script.js:6 (6) ["abrigos", "camisas", "poleras", "zapatos", "zapatillas", "pantalones"]
>
```

Y, para agregar un elemento al final de una matriz, utilizamos el método `push()` que aplicaremos a ésta misma:

```
1 // Agregar un elemento al final de una matriz ( push() )
2 console.log(`Agregamos un elemento, ahora el tamaño de la matriz:
3 '${closet.push('Cortavientos')}'`);
4 console.log(closet);
```

Por consola podemos ver cómo aumentó el tamaño de nuestro Array y, a la vez, mediante la flecha amarilla, como el nuevo elemento está en la misma posición que `"traje"`, lo cual destaca que esta acción agrega un elemento a la parte final de la matriz.

```

script.js:3
▶ (7) ["abrigos", "camisas", "poleras", "zapatos", "zapatillas", "pantalones",
"traje"]
Eliminamos último elemento: 'traje' script.js:6
script.js:7
▶ (6) ["abrigos", "camisas", "poleras", "zapatos", "zapatillas", "pantalones"]
Agregamos un elemento, ahora el tamaño de la matriz: '7' script.js:12
script.js:13
▶ (7) ["abrigos", "camisas", "poleras", "zapatos", "zapatillas", "pantalones",
"Cortavientos"]
>
  
```

Ya hemos visto dos métodos que nos permiten interactuar con el elemento al final de una matriz. A continuación, revisaremos aquellos que sirven para interactuar con el primer elemento de una matriz. En primer lugar, cómo eliminar el primer elemento de una matriz.

El método **shift()** es el que nos permite eliminar el primer elemento de una matriz:

```

1 // Eliminar un elemento del principio de una matriz ( shift() )
2 console.log(`Eliminamos el primer elemento: '${closet.shift()}'`);
3 console.log(closet);
  
```

A continuación, podemos ver el resultado, el cual indica mediante la flecha amarilla, que el elemento inicial que era "abrigos" ahora ya no está:

```

script.js:3
▶ (7) ["abrigos", "camisas", "poleras", "zapatos", "zapatillas", "pantalones", "traje"]
Eliminamos último elemento: 'traje' script.js:6
script.js:7
▶ (6) ["abrigos", "camisas", "poleras", "zapatos", "zapatillas", "pantalones"]
Agregamos un elemento, ahora el tamaño de la matriz: '7' script.js:12
script.js:13
▶ (7) ["abrigos", "camisas", "poleras", "zapatos", "zapatillas", "pantalones", "Cortavientos"]
Eliminamos el primer elemento: 'abrigos' script.js:19
script.js:20
▶ (6) ["camisas", "poleras", "zapatos", "zapatillas", "pantalones", "Cortavientos"]
>
  
```

Ahora, para agregar un elemento del comienzo de una matriz, utilizamos el método **unshift()**, el cual retorna el valor del tamaño de la matriz, tal como lo hace **pop()**:

```

1 // Agregar un elemento al comienzo de una matriz ( unshift() )
2 console.log(`Agregamos un elemento al inicio, ahora el tamaño de la matriz
3 es: '${closet.unshift('parka')}'`);
4 console.log(closet);
  
```

Mediante la consola, podemos ver que el método efectivamente logró incorporar un elemento nuevo al inicio de la matriz:


```

▶ (7) ["abrigos", "camisas", "poleras", "zapatos", "zapatillas", "pantalones", "traje"] script.js:3
Eliminamos último elemento: 'traje' script.js:6
▶ (6) ["abrigos", "camisas", "poleras", "zapatos", "zapatillas", "pantalones"] script.js:7
Agregamos un elemento, ahora el tamaño de la matriz: '7' script.js:12
▶ (7) ["abrigos", "camisas", "poleras", "zapatos", "zapatillas", "pantalones", "Cortavientos"] script.js:13
Eliminamos el primer elemento: 'abrigos' script.js:19
▶ (6) ["camisas", "poleras", "zapatos", "zapatillas", "pantalones", "Cortavientos"] script.js:20
Agregamos un elemento al inicio, ahora el tamaño de la matriz es: '7' script.js:24
▶ (7) ["parka", "camisas", "poleras", "zapatos", "zapatillas", "pantalones", "Cortavientos"] script.js:25

```

De esta forma, ya quedan analizadas tanto las propiedades, como los métodos definidos de los objetos Array en JavaScript.

EXERCISE 3: ÁLGEBRA CON ARRAYS

Hasta ahora hemos visto cómo manipular los datos dentro de los arreglos. Ahora, aprenderemos cómo interactuar con la información de más de un arreglo.

Para manipular múltiples matrices, JavaScript nos permite usar funciones algebraicas de conjuntos de datos, como: uniones, intersecciones, concatenaciones y restas. Para aprender a implementarlas, como requisito previo, debemos saber utilizar los métodos **filter** e **include**.

El método **filter()** crea una nueva matriz, con todos los elementos que pasan la prueba implementada por la función proporcionada. En otras palabras, éste filtra datos basándose en una función. Para ver cómo funciona, realizaremos un ejemplo.

```
1 // Usando el método filter
2 var edades = [32, 33, 16, 40];
3 var resultado = edades.filter(verificarMayorDeEdad);
4
5 function verificarMayorDeEdad(edad) {
6     return edad >= 18;
7 }
8 console.log(resultado)
```

En este ejemplo tenemos un Array “edades”. Si quisiéramos filtrar solo las edades de personas que son mayores de edad, debemos llamar sobre el Array, el método `filter()`, pasándole por parámetro la función `verificarMayorDeEdad`. Esta recibe una edad por parámetro -obtenida del Array “edades”- y solamente retorna las que son mayor o igual a 18 años, así se logra filtrar a los mayores de edad.

Cómo en nuestro arreglo “edades” solamente hay un valor que es menor de edad, al mostrar el resultado en nuestra consola, veremos que se logran filtrar todas las edades que son mayores.

► (3) [32, 33, 40]

javascript.js:8



Dado que hemos comprendido cómo utilizar el método `filter()`, pasaremos a ver cómo usar `includes()`. En castellano, este método se llamaría “incluye”, dado que devuelve verdadero si una matriz contiene, o *incluye*, un valor específico, y devuelve falso si no se encuentra, o *no incluye*, el valor.

Para comprenderlo, vamos a establecer un Array que contiene una serie de frutas. Si quisiéramos comprobar si es que este arreglo contiene o *incluye* cierta fruta, podríamos desarrollar el siguiente ejemplo: comprobaremos si el arreglo “frutas” contiene el valor “Mango”.

```
1 var frutas = ["Plátano", "Naranja", "Manzana", "Mango"];
2 var busqueda = frutas.includes("Mango");
3
4 console.log(busqueda);
```

Si nos fijamos en el valor de nuestra búsqueda en la consola, veremos que devuelve **true**, indicando que si se encuentra el valor "Mango" en el arreglo "frutas". Algo importante en destacar, es que este método distingue entre mayúsculas y minúsculas, por lo que "mango" retornaría **false**.

Opcionalmente, podemos colocar como segundo parámetro del método **includes**, el índice desde el cual queremos empezar a verificar si contiene un valor. Por ejemplo, podríamos buscar si el Array "frutas" contiene el valor "Mango", solo a partir del índice 2 del arreglo (que, en este caso, sería desde el tercer elemento en adelante, porque la primera posición es el índice 0).

```
1 var frutas = ["Plátano", "Naranja", "Manzana", "Mango"];
2 var busqueda = frutas.includes("Mango", 2);
3
4 console.log(busqueda);
```

Si ahora nos dirigimos a nuestra consola, veremos que el resultado fue el mismo: **true**.

Ahora que hemos aprendido a usar estos métodos, podemos comenzar a profundizar en el uso del álgebra con matrices, mediante los conceptos de: concatenación, unión, intersección y resta.

Comencemos con el primero, la concatenación, que se puede lograr llamando al método **concat()** desde un arreglo, y pasándole por parámetro el valor de otro, lo cual nos permitirá crear un nuevo arreglo que fusiona a más de uno. Por ejemplo, si quisiéramos crear un arreglo "alfanumérico" desde un arreglo de "números", y otro de "letras", podríamos realizar esto de la siguiente manera:

```
1 var numeros = [1, 2, 3];
2 var letras = ["a", "b", "c"];
3 var alfanumerico = numeros.concat(letras);
4
5 console.log(alfanumerico);
```

Si ahora nos fijamos en el resultado de la consola, veremos que logramos unir, o concatenar, ambos arreglos.

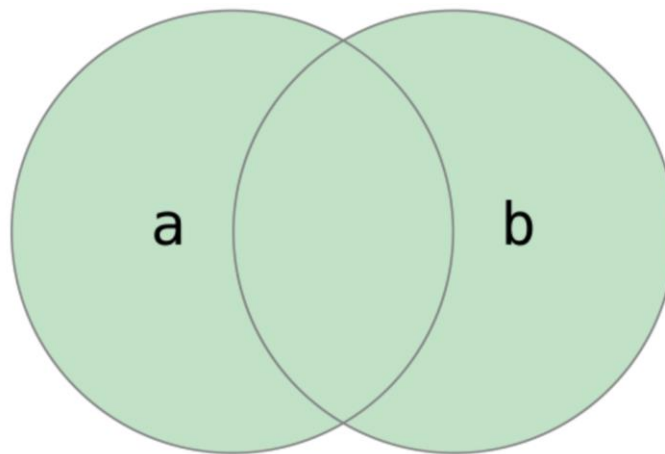
```
► (6) [1, 2, 3, 'a', 'b', 'c']
```

```
javascript.js:43
```

```
>
```

Para el resto de los conceptos que veremos, usaremos solo los siguientes 2 arreglos:

```
1 var a = [1, 2, 3]
2 var b = [2, 4, 5]
```



El siguiente concepto, será el de unión. Éste nos permite unir todos los elementos de más de una matriz, sin repetición. Para unir los arreglos **a** y **b**, debemos realizar lo siguiente:

```
1 var union = a.concat(b.filter(function (x) {
2     return !a.includes(x)
3 }));
4 console.log(union);
```

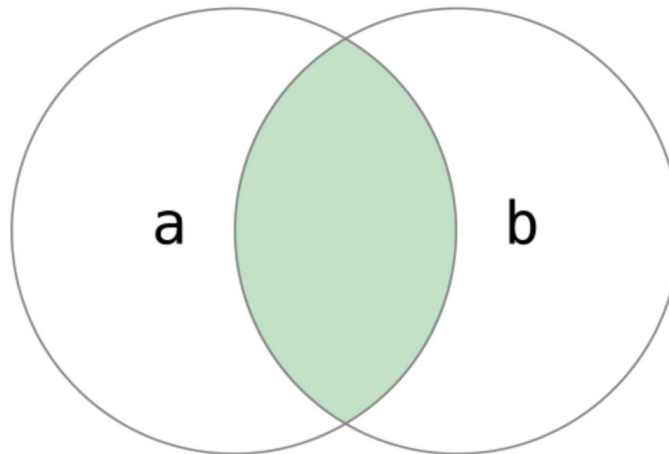
Primero llamamos al método **concat()** sobre el arreglo “a”, y por parámetro vamos a pasar el arreglo “b”, donde usando el método **filter()** haremos que solo se unan los valores de “b” que no están en “a”. Si comprobamos nuestro desarrollo mediante la consola, veremos que efectivamente se logró la unión de los datos sin repeticiones.

► (5) [1, 2, 3, 4, 5]

javascript.js:28

>

El siguiente concepto que veremos es la intersección, esto es cuando logramos unir solo los datos que se repiten, o *interceptan*, en ambos arreglos. En nuestro caso: "a" y "b".



Para realizar una intersección sobre nuestros dos conjuntos, vamos a hacer lo siguiente:

```
1 var interseccion = a.filter((function (x) {  
2     return b.includes(x)  
3 }));  
4 console.log(interseccion);
```

Mediante estas líneas creamos un arreglo llamado intersección, que consiste en todos los elementos del arreglo "a", que también están presentes en el arreglo "b". Logramos esto haciendo que la función de filtro haga uso de su parámetro x en el método `includes` llamado sobre el arreglo b. En efecto, estamos diciendo: "si un elemento del arreglo 'a' (lo cual corresponde a x), está incluido en el arreglo 'b', entonces vamos a fijar ese valor en el arreglo 'intersección'".

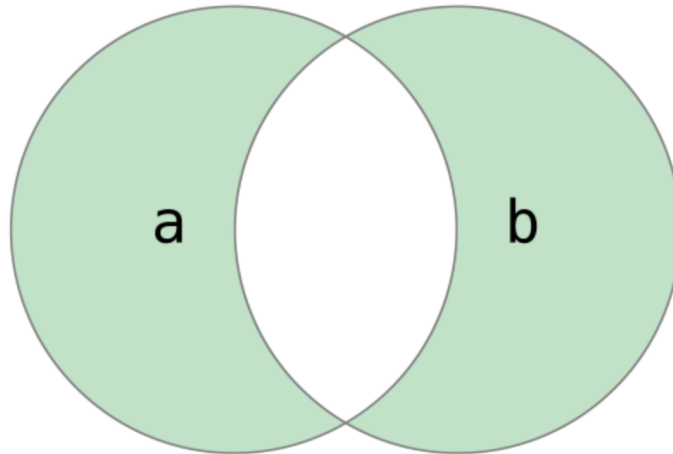
El único elemento que tienen en común los arreglos "a" y "b", es el 2. Si nos dirigimos a nuestro navegador, veremos que efectivamente se logra crear un arreglo de intersección mediante nuestro código:

▶ [2]

javascript.js:35



El último método que veremos es la diferencia o resta de arreglos.



Conceptualmente, nuestro objetivo es restar en “a”, los elementos de igual valor existentes en “b”, logrando solo mostrar los elementos que no tienen en común. Entonces, en nuestro caso, vamos a quitar los elementos de “a” que están presentes en “b”. Para alcanzar este objetivo, plantearemos el siguiente código:

```
1 var diferencia = a.concat(b).filter((function (x) {  
2   return !a.includes(x) || !b.includes(x)  
3 }));  
4 console.log(diferencia);
```

En estas líneas vamos a concatenar al arreglo “a”, todos los elementos que no tienen en común con el arreglo “b”. Es por eso que partimos pasando el segundo, como parámetro del método `concat()`. Luego, llamamos sobre esta concatenación al método `filter()`, donde mantendremos todos los elementos menos los que “a” y “b” tienen en común. Al anteponer un signo de exclamación, sobre las invocaciones al método `includes()`, logramos obtener todos los elementos que no están en “a” (4 y 5 desde b), o todos los que no están en b (en este caso 1 y 3).

La razón por la que no usamos el operador AND `&&` en este caso, es porque significaría que nuestro arreglo estaría poblado por todos los valores que no están en “a”, y que no están en “b”; es decir, no tendría ningún valor. Al final, el resultado de nuestro código en la consola es el siguiente:

```
► (4) [1, 3, 4, 5]
```

```
javascript.js:52
```

```
>
```



Como pudimos estudiar, JavaScript nos permite desarrollar con arreglos en todo tipo de formas, ya sea que necesitemos cambiar valores individuales, o mezclar los contenidos de diferentes arreglos. El uso de estos conceptos algebraicos, al trabajar con este tipo de datos, nos da la posibilidad de resolver problemas complejos de manera sencilla.