4. ARRAYS

Arrays

Un array es un **objeto** que contiene varios valores encerrados entre corchetes y separados por comas.

Intenta ingresar las siguientes líneas en ty consola:

```
let miArrayDeNombres = ['Chris', 'Bob', 'Jim'];
let miArrayDeNumeros = [10, 15, 40];
```

Una vez que se definen estos arrays, puedes acceder a cada valor por su ubicación dentro del array. Prueba estas lineas:

```
miArrayNombre[0]; // debe devolver 'Chris'
miArrayNumérico[2]; // debe devolver 40
```

Los **corchetes**

- permiten especificar un **valor de índice** correspondiente a la posición del valor que desea devolver.
- es posible que hayas notado que las arrays en JavaScript tienen **un índice cero**: el primer elemento está en el índice 0.

Literales array

Un **literal array**:

- es una lista de cero o más expresiones, cada una de las cuales representa un **elemento de array**, encerrado entre **corchetes** (`[]`).
- cuando creas un array utilizando un literal array, se inicializa con los valores especificados como sus elementos y su `length` (longitud) se establece en la cantidad de argumentos especificados.

El siguiente ejemplo crea el array de **`cafes`** con tres elementos y una longitud de tres:

```
const cafes = ['Tostado Francés', 'Colombiano', 'Kona'];
```

Si se crea en un script de nivel superior un array:

- JavaScript interpreta el array cada vez que evalúa la expresión que contiene el literal array.
- además, si un literal array se utiliza en una función, se creará dicho literal cada vez que se llama a la función .

Nota: Los literales array crean objetos `Array`

Comas adicionales en los literales de array

Si colocas dos comas seguidas en un literal array, el array deja un **espacio/ranura vacía** para el elemento no especificado.

- el siguiente ejemplo crea la array **`pez`**:

```
const pez = ['León', , 'Ángel'];
```

- cuando registres esta array, verás:

```
console.log(pez);
// [ 'León', vacío , 'Ángel']
```

- ten en cuenta que el segundo elemento está "vacío", que no es exactamente el mismo que el valor **`undefined`** .
- cuando se utilizan **métodos de recorrido de arrays** como `Array.prototype.map`, las ranuras vacías se saltan .
- sin embargo, al acceder usando **`pez[1]`** se devuelve **`undefined`** .

Si incluyes una **coma** final al final de la lista de elementos, la coma se ignora.

- en el siguiente ejemplo, la `length` del array es 3.
- no existe un elemento **`miLista[3]`**, pues los índices son 0, 1, 3 (0 = primer elemento, 1 = segundo elemento, ...)
- todas las demás comas en la lista indican un nuevo elemento.

```
const miLista = ['inicio', , 'escuela' , ];
```

En el siguiente ejemplo, la `length` del array es cuatro y faltan **`miLista[0]`** y **`miLista[2]`**.

```
const miLista = [ , 'hogar', , 'escuela'];
```

En el siguiente ejemplo

- la `length` del array es 4 y faltan `miLista[1]` y `miLista[3]` .
- **solo se ignora la última coma.**

```
const miLista = ['casa', , 'escuela', , ];
```

Nota: Las comas finales ayudan a mantener limpias las diferencias de git cuando tiene un array de varias líneas, porque agregar un elemento al final solo agrega una línea, pero no modifica la línea anterior.

```
const miLista = [
"hogar",
"escuela",
+ "hospital",
];
```

Comprender el comportamiento de las comas adicionales es importante para entender JavaScript como lenguaje.

- sin embargo, al escribir tu propio código, debes declarar explícitamente los elementos que faltan como `undefined` ,
- o al menos insertar un comentario para resaltar su ausencia.
- al hacer esto incrementamos la **claridad** y la facilitad de **mantenimiento** del código .

```
const miLista = ['casa', /* vacío */ , 'escuela', /* vacío */ , ];
```

Colecciones indexadas: Arrays y arrays tipados

Recordemos que:

- son objetos normales para los que existe una relación particular entre las propiedades de clave entera y la propiedad de `length`.
- además, las arrays heredan de `Array.prototype`, que proporciona un puñado de métodos convenientes para manipular arrays.
  - por ejemplo, `indexOf()` busca un valor en la array y `push()` agrega un elemento a la array.
  - esto convierte a los Arrays en un candidato perfecto para representar **listas ordenadas** .

Los Typed Arrays permiten ofrecer una vista, de un búfer de datos binarios subyacente, similar a un array y ofrecen muchos métodos que tienen una semántica similar a sus correspondientes métodos de un array "auténtico".

- "Typed Array" es un término genérico para una variedad de estructuras de datos, incluidos `Int8Array`, `Float32Array` , etc.
- Las arrays tipadas se usan a menudo junto con `ArrayBuffer` y `DataView`

AVANZADO: Ejemplo de arrays tipados

```
const f2b = (x) => new Uint8Array(new Float64Array([x]).buffer);
const b2f = (x) => new Float64Array(x.buffer)[0];

// Obtener una representación de bytes de NaN
constante n = f2b(NaN);

// Cambia el primer bit, que es el bit de signo y no importa para NaN
n[7] = 255;
constante nan2 = b2f(n);
console.log(nan2); // NaN
console.log(Objeto.es(nan2, NaN)); // verdadero
console.log(f2b(NaN)); // Uint8Array(8) [0, 0, 0, 0, 0, 0, 248, 127]
console.log(f2b(nan2)); // Uint8Array(8) [1, 0, 0, 0, 0, 0, 248, 127]
```

NaN = 7F F8 00 00 00 00 00 00
127 248

-NaN = FF F8 00 00 00 00 00 00
255 248