

# M1 - 5 - JS Bucles y Arrays

## Bucles

Hasta ahora hemos visto estructuras de control que nos permitían mostrar unas instrucciones u otras dependiendo de lo que fuera pasando a lo largo de nuestro programa, normalmente decisiones tomadas por el usuario. Pero el flujo del programa siempre era descendente.

Ahora veremos otras estructuras de control que nos permiten, en ciertos casos y mientras se cumplan ciertas condiciones, modificar este flujo descendente, pudiendo ejecutar una sentencia ninguna, una o más de una vez.

### While

La estructura while nos permite, según la condición indicada, ejecutar ninguna, una o varias veces las instrucciones que tenemos entre las llaves. El comportamiento del ‘while’ está definido por la condición, las instrucciones que tenemos entre las llaves se ejecutarán MIENTRAS la condición que se encuentra entre los paréntesis sea ‘**true**’. Esto nos lleva a varias posibilidades:

- Qué la primera vez que se evalúa la condición el resultado sea ‘**false**’ → en este caso el código que tenemos entre las llaves no se va a ejecutar, y seguiremos con el resto del programa. No se va a ejecutar ni una vez
- Si la primera vez que se evalúa la condición el resultado es ‘**true**’, el código que tenemos entre las llaves, si que se va a ejecutar, pero una vez que se ejecute en vez de seguir con el flujo descendente, el programa va a **VOLVER** a evaluar la condición del while ¿ Por qué? Por que ese código se va a ejecutar **MIENTRAS** esa condición siga siendo ‘**true**’, por lo tanto y hasta que la condición no se evalúe como ‘false’, el programa va a seguir volviendo a la condición para evaluarla.

```
while(condición) {  
    ...  
}
```

Veamos un ejemplo real:

En este caso hemos declarado una variable ‘i’ a la que le hemos asignado un valor de 0. Después, tenemos un ‘while’ cuya condición para que se ejecute el bucle es que ‘i’ sea menor que tres.

Cuando llegamos a esta condición por primera vez, el valor de ‘i’ es 0, por lo tanto, ‘i’ es menor que 3, y esta condición se evalúa como verdadera, con lo que las instrucciones que tenemos entre las llaves si que se van a ejecutar. Una de estas instrucciones que tenemos dentro de las llaves es ‘`i = i + 1;`’ es decir, le sumamos uno a ‘i’, así que después de ejecutar estas líneas por primera vez, ‘i’ ya no será 0, ahora será 1.

👉 Recordar que `i = i + 1`; también podemos indicarlo de otras maneras :

`i += 1` o `i++;`

Una vez hemos ejecutado el código por primera vez e ‘i’ es igual a 1, el programa volverá al ‘while’ a evaluar la condición (esto lo hará SIEMPRE hasta que la condición se evalúe como falsa). Al volver a comprobar la condición, ‘i’ que ahora es 1 sigue siendo más pequeño que 3, con lo que se volverán a ejecutar las instrucciones dentro de las llaves, volviendo a sumar 1 a ‘i’ y modificando su valor a 2.

De nuevo, el programa volverá al while a comprobar de nuevo la condición que sigue siendo ‘**true**’ puesto que ‘i’ es 2 y sigue siendo más pequeño que 3. Cómo la condición es verdadera, se vuelve a ejecutar el código que tenemos dentro de las llaves y se vuelve a sumar uno a ‘i’ que ahora va a tener un valor de 3. Cómo bien sabéis el programa va a volver a evaluar la condición, pero esta vez ‘i’ tiene un valor de 3 y, por lo tanto, no es menor que 3, la condición se evalúa como ‘**false**’. Ahora no se va a ejecutar el código que tenemos entre llaves, pero tampoco se va a volver a comprobar la condición, porque una vez se evalúa como falsa ya no se vuelve a comprobar. Entonces salimos del bucle, y el programa sigue.

```
let i = 0;
while(i < 3) {
    console.log(i);
    i = i + 1;
}
```

## Do... while

La estructura ‘do...while’ es muy similar a la del ‘while’ salvo por una diferencia en su comportamiento.

En el caso del ‘do...while’ la condición se evalúa después de ejecutar las instrucciones que se encuentran dentro de las llaves, con lo que **AL MENOS** una vez sí que se ejecuta, en el caso del ‘while’ al evaluar la condición al principio, no tiene por qué ejecutarse ni siquiera una vez.

En resumen, el comportamiento del ‘do...while’ vendría a ser → ‘Ejecuta estas líneas de código, y si la condición es verdadera vuelve a ejecutarlo’, y así hasta que la condición se evalúe como falsa.

```
do {  
    ...  
} while(condición);
```

En este caso, he declarado la variable ‘respuesta’ y, a continuación, entramos en el bucle, se ejecutan las instrucciones que tenemos entre las llaves dónde saludamos al usuario y le preguntamos si quiere que le volvamos a saludar. Recogemos la respuesta del usuario y la almacenamos en la variable respuesta. Una vez se ha ejecutado, entramos a evaluar la condición, si la respuesta del usuario es ‘`==`’ a ‘`'si'`’, entonces se vuelve a ejecutar el código que tenemos entre las llaves, volviendo a saludar al usuario y a preguntarle si quiere que le volvamos a saludar. Cuando se vuelve a evaluar la condición, si la respuesta del usuario **NO** es ‘`==`’ a ‘`'si'`’ ya no se va a volver a ejecutar el código que tenemos entre las llaves y el programa terminará, puesto que no tenemos ninguna instrucción más.

```
<script>  
    let respuesta;  
    do {  
        window.alert('Hola!');  
        respuesta = window.prompt('quieres que te vuelva a saludar?');  
    } while(respuesta.toLowerCase() === 'si');  
</script>
```

## For

```
for ([expresión-inicial];[condición];[expresión-final]) {  
    ...  
}
```

El bucle ‘for’ lo utilizamos para repetir una o varias instrucciones un número determinado de veces.

Su estructura, es muy parecida a la estructura del ‘while’ pero más simplificada. Entre paréntesis definimos una expresión inicial que resulta la declaración de una variable ‘i’ a la cual le asignamos un valor inicial ‘**let i = 1**’. A continuación, indicamos la condición que debe cumplirse para que el bucle siga ejecutándose, en este caso ‘**i <=3**’, es decir, el bucle se va a repetir siempre y cuando ‘i’ sea menor o igual que 3. Por último, definimos una expresión final que irá incrementando el valor de ‘i’ en uno ‘**i++**’.

```
let i = 1
while (i<=3){
    console.log(i)
    i++
}

for(let i = 1; i<=3; i++){
    console.log(i)
}
```

Veamos cómo se comporta:

Cuando nuestro programa llegue a la línea dónde está el bucle, nos encontramos con la expresión inicial se declara la variable, ‘i’ es igual a 0, se comprueba la condición ( i es menor que 3) “**true**”, entonces se ejecutan las instrucciones ( en este caso la instrucción) que se encuentran dentro de las llaves, en este ejemplo se mostrará por consola el valor de ‘i’ que ahora es igual a 0.

Una vez se ha ejecutado esta instrucción, el programa se dirige a la expresión final y se suma uno al valor de ‘i’ (ahora i = 1). A continuación, vuelve a empezar y se evalúa la condición, ( i (1) es menor que 3) → “**true**” y se vuelve a ejecutar la instrucción que tenemos entre las llaves, en este caso se muestra por consola el valor de la variable ‘i’, que ahora es igual a 1.

Este proceso se vuelve a repetir:

expresión final → sumamos uno a ‘i’ → i = 2

condición → i (2) es menor que 3 → TRUE

al ser ‘**true**’ la condición se ejecuta el código que tenemos entre las llaves

Y de nuevo se vuelve a repetir el proceso:

expresión final → sumamos uno a ‘i’ → i = 3

condición → i (3) no es menor que 3 → “**false**”

Ahora la condición es ‘false’ y por lo tanto, ya no se cumple la condición, ni se va a ejecutar lo que tenemos entre las llaves. El programa sale del bucle y sigue con el resto de instrucciones.

Podemos utilizar el bucle para recorrer cada uno de los caracteres de un string, de una cadena de texto, por ejemplo:

1. En el bucle, declaramos la variable ‘i’ a la cual asignamos un valor inicial de 0, esto es muy importante porque si recordáis, el primer carácter de una cadena de texto (string) ocupa la posición 0.
2. En la condición, indicamos que el bucle se va a repetir siempre que el valor de ‘i’ sea menor que la longitud de la palabra. Para esto vamos a usar la propiedad `.length` que nos devuelve el número de caracteres que contiene una cadena de texto. En este caso queremos saber la longitud del string que contiene la variable ‘palabra’, que es dónde hemos almacenado la palabra introducida por el usuario
3. En la expresión final, indicaremos que a cada vuelta incrementaremos el valor de i en uno.

Cuando el programa llega al for se encuentra que `i = 0` y `palabra.length = 4` por lo tanto la condición es ‘**true**’ y se ejecuta el código que tenemos entre las llaves. Este código muestra por pantalla, de la palabra que ha introducido el usuario los caracteres que van desde ‘i’ a ‘`i+1`’. Cómo ahora mismo `i` es igual a 0, estaríamos indicando que queremos coger a partir de la cadena dada ‘palabra’ los caracteres desde la posición 0 hasta el que se encuentra en la posición `i+1` (1), cómo este último no lo incluye, en realidad, solamente estamos mostrando un único carácter que se corresponde con el primer carácter de la cadena.

Esto se va a repetir mientras `i` sea menor que `palabra.length` (4 en este caso), ejecutándose el código 4 veces mientras ‘`i`’ va incrementando su valor en cada vuelta ( 0,1,2,3 recorriendo todas las posiciones del string) hasta que su valor es 4, que la condición ya no se cumple.

```
<script>
let palabra = window.prompt('Escribe una palabra'); // 'hola'

console.log(palabra.substring(0,1));
console.log(palabra.substring(1,2));
console.log(palabra.substring(2,3));
console.log(palabra.substring(3,4));
```

```
let palabra = window.prompt('Escribe una palabra'); // 'hola'

for(let i = 0; i < palabra.length; i = i+ 1) {
    console.log(palabra.substring(i,i+1));
}

</script>
```

## ¿Cómo sabemos que estructura utilizar?

- **While** : Utilizamos while cuando el código o las instrucciones que están entre las llaves del bucle no tienen por qué ejecutarse por lo menos una vez. Es decir, si la condición se cumple se ejecutará, una o varias veces, pero si no se cumple no tiene por qué ejecutarse. Pero además, no sabemos específicamente cuántas veces se va a repetir.
- **Do while** : Utilizamos do..while, cuando al menos el código que se encuentra dentro de las llaves se tiene que ejecutar al menos una vez. Pero en este caso, tampoco sabemos específicamente cuántas veces se va a repetir el bucle. Por ejemplo, cuando pedimos la contraseña a un usuario, una vez al menos se la vamos a pedir. Si el usuario introduce una contraseña incorrecta se lo volveremos a preguntar y así hasta que el usuario introduzca bien la contraseña, pero no sabemos específicamente cuántas veces va a introducirla mal, es decir, no sabemos cuantas veces se va a repetir el bucle.
- **For** : Utilizamos el ‘for’ cuando queremos repetir el código que tenemos entre llaves ‘x’ veces en específico. Ya sea exactamente, tantas veces como caracteres tiene un string, como años tenga una persona, como alumnos haya en una clase..

# Arrays

Los arrays son ‘colecciones’, ‘listas’ de variables o valores del mismo o distinto tipo de dato. Es decir, podemos tener un array ‘lista’ de números, de strings, de booleanos...

Por ejemplo, imaginemos que necesitamos manejar los meses del año. Podemos crear una variable por cada uno de los meses:

```
let primerMes = "Enero";
let segundoMes = "Febrero";
let tercerMes = "Marzo";
...
let duodecimoMes = "Diciembre";
```

Aunque resultaría muy poco práctico y bastante tedioso el tener que estar creando una variable por cada uno de los meses. A través de un array podríamos agruparlos en una ‘lista’ colección, lo que resultaría más eficiente y fácil de manejar a lo largo del programa.

## Estructura general de un array

Para definir un array utilizamos los corchetes, a través de los símbolos “[ ]” que indicarán el principio y el final del array. Entre los corchetes introducimos los valores separados por comas.

```
let mes = ["Enero", "Febrero", "Marzo", "Abril", "Mayo", "Junio",
"Julio", "Agosto", "Septiembre", "Octubre", "Noviembre", "Diciembre"];
```

De esta manera, tenemos una única variable a través de la cual podemos acceder a cada uno de los valores almacenados en ella.

Cada uno de los elementos tiene una posición dentro del array, las posiciones van desde 0 (al igual que en los strings) hasta su longitud menos 1. Es decir, “Enero” estaría en la posición 0 dentro del array “mes”, “Febrero” en la posición 1, “Marzo” en la posición 2, “Abril” en la posición 3... y así hasta “Diciembre” que estaría en la posición 11. Sin embargo, “mes” tendría 12 elementos.

## Crear un array

En el ejemplo anterior hemos visto cómo podemos crear un array de manera muy simple, asignándole valores en el momento que lo creamos.

```
» let animales = ["perro", "gato", "elefante"];
```

Pero también podemos crear un array “vacío” y después, ir añadiendo cada uno de los elementos que queremos introducir en nuestro array.

Inicializamos un array vacío:

```
>> let numeros = [];
```

Podemos introducir elementos en nuestro array en una posición determinada. En este caso, por ejemplo, vamos a introducir elementos en las posiciones 0,1,2,3 del array.

```
>> let numeros = [];
← undefined
>> numeros[0] = 2;
← 2
>> numeros[1] = 4;
← 4
>> numeros[2] = 6;
← 6
>> numeros[3] = 8;
← 8
```

```
>> numeros
← ▶ Array(4) [ 2, 4, 6, 8 ]
```

Siguiendo con este ejemplo, podríamos introducir un elemento en la posición 10 del array, aunque solo tenemos 4 elementos, ¿Qué pasará con las posiciones intermedias a las que no se les ha asignado ningún valor?. No pasa nada, no te va a dar error pero las posiciones intermedias a las que no se les ha asignado ningún valor, se le da por defecto un valor de **undefined**, están vacíos. Aunque no se recomienda hacer, está bien que lo sepáis por si algún día os pasa.

También podemos añadir elementos a nuestro array usando el método **push()** al cual le pasaremos por parámetro el elemento que queremos introducir. Este método añade los elementos **al final** del array. Es decir, si el array está vacío, el primer elemento que introducimos se introducirá en la posición 0 del array, el segundo en el 1, el tercero en el 2... y así sucesivamente. Por ejemplo en el array ‘numeros’ que hemos creado anteriormente tenemos 4 elementos, con posiciones de 0 a 3. Si utilizamos el método **push()** para incorporar un nuevo elemento, este se incluirá ocupando la posición 4, como 5º elemento del array.

```
>> numeros
<- ▶ Array(4) [ 2, 4, 6, 8 ]
>> numeros.push(10)
<- 5
>> numeros
<- ▶ Array(5) [ 2, 4, 6, 8, 10 ]
```

Podemos introducir los valores en nuestro array, a través de variables:

```
>> let animal1 = "perro";
<- undefined
>> let animal2 = "gato";
<- undefined
>> let animal3 = "elefante";
<- undefined
>> let animales = [animal1, animal2, animal3];
```

### Mostrar los valores de un array

Para mostrar los valores de un array, solamente tenemos que indicar el nombre del array y entre corchetes indicar la posición del valor que deseamos mostrar.

```
>> let animales = ["perro", "gato", "elefante"];
<- undefined
>> animales[0]
<- "perro"
>> animales[1]
<- "gato"
```

En este caso sólo tenemos 3 elementos en nuestro array, pero si tuviéramos 20 elementos y quisieramos mostrar todos sus valores, resultaría bastante tedioso. Para evitarlo podemos **utilizar un bucle**. Inicializamos la variable 'i' en 0 puesto que la primera posición del array es 0. Despues definimos la condición, la instrucción que se encuentra entre los corchetes del bucle se va a seguir ejecutando mientras sea menor que 3. Y por último por cada vuelta incrementamos el valor de 'i' en 1. De esta manera 'i' por cada vuelta irá incrementando su

valor, desde 0,1,2.. hasta 3, cuando la condición no se va a cumplir y, por lo tanto, saldremos del bucle. Esos valores que va a tener ‘i’ se corresponden con las posiciones de nuestro array animal, lo que aprovecharemos recorrer y mostrar los valores del array.

```
>> for(let i = 0; i < 3; i++) {  
    console.log(animales[i]);  
}  
  
perro  
gato  
elefante
```

### .length

A través de la propiedad length podemos obtener la cantidad de elementos que contiene un array. Podemos utilizar dicha propiedad para definir la condición del bucle, y de esta manera recorrer cada una de las posiciones del array.

```
>> for(let i = 0; i < animales.length; i++) {  
    console.log(animales[i]);  
}  
  
perro  
gato  
elefante
```

## Constantes

Como veíamos anteriormente, podemos declarar variables constantes para evitar que su valor sea modificado o reasignado. En el caso de los arrays, podemos declararlos también como constantes, aunque su comportamiento es algo distinto en los arrays. En este caso, aún declarado como constante, se pueden añadir nuevos elementos al array e incluso modificar los valores.

```
» const nums = [1, 2, 3];
← undefined
» nums[2] = 4
← 4
» nums
← ▶ Array(3) [ 1, 2, 4 ]
```

- Tenemos que tener en cuenta que si creamos un array a partir de otro, realmente estamos creando dos variables que apuntan hacia el mismo array, es decir, que si decidimos modificar uno de los valores de uno de los arrays, el otro también se verá modificado. **Cuidado con esto.**

```
» nums
← ▶ Array(3) [ 1, 2, 4 ]
» let otros = nums
← undefined
» otros
← ▶ Array(3) [ 1, 2, 4 ]
» otros[1] = 1000
← 1000
» nums
← ▶ Array(3) [ 1, 1000, 4 ]
```

## Modificando Arrays

Podemos utilizar diferentes métodos que nos permiten modificar un array fácilmente, uno de ellos es `push()`, anteriormente visto para introducir nuevos elementos en nuestro Array.

### `array.pop()`

Utilizamos este método para eliminar el último elemento del array, es decir, el que se encuentra en la última posición. Además, este nos devuelve el valor que ha sido eliminado.

```
>> numeros
← ▶ Array(5) [ 2, 4, 6, 8, 10 ]
>> numeros.pop()
← 10
```

### `array.slice()` y `array.splice()`

**Splice** : Este método recibe dos parámetros, el primero indicará una posición dentro del array, el segundo indica el número de elementos que deseamos eliminar desde la posición anteriormente indicada.

De esta manera, nos permite eliminar varios elementos de nuestro array a la vez y además, este método nos devuelve un array con los números que han sido eliminados.

```
> let array = [1,4,6,7,3,5]
← undefined
> array.splice(1,2)
← ▶ (2) [4, 6]
> array
← (4) [1, 7, 3, 5]
```

**Slice** : El método slice es muy parecido al anterior con unas pequeñas diferencias. También nos permite eliminar elementos de un array, pero no modifica el mismo, lo que hace es crear un array nuevo sin los valores que hemos eliminado, pero el array del que partimos se mantiene intacto. Como el anterior, recibe 2 parámetros, el primero indicará una posición dentro del array, y el segundo indica la posición siguiente al último valor que queremos eliminar.

A partir del anterior array, vamos a utilizar el método slice para eliminar los elementos de la posición 1 y 2.

```
> array = [1,7,3,5]
< ▶ (4) [1, 7, 3, 5]
> let eliminados = array.slice(1,3)
< undefined
> eliminados
< ▶ (2) [7, 3]
> array
< (4) [1, 7, 3, 5]
```

### array.indexOf()

A través de indexOf() podemos conocer la posición de un elemento en concreto dentro del array. Si buscamos un elemento que no se encuentra en el array, el método nos devuelve un -1.

```
← ▶ Array(3) [ 4, 6, 8 ]
» numeros.indexOf(6)
← 2
» numeros.indexOf(66)
← -1
```

### array.join()

Este método nos devuelve un string con los elementos de nuestro array separados por comas. Entre paréntesis podemos pasarle un string, que sustituirá a la coma separando los elementos del array.

```
← ▶ Array(4) [ 2, 4, 6, 8 ]
» numeros.join()
← "2,4,6,8"

```

  

```
» numeros.join('hola')
← "2hola4hola6hola8"
```

### array.concat()

Podemos utilizar el método concat, para añadir más de un elemento a la vez en nuestro array, añade los elementos de 'numeros2' al array que le pasamos por parámetro al final de 'numeros1'.

```
> let numeros1 = [1,2,3,4]
< undefined
> let numeros2 = [5,6,7,8]
< undefined
> numeros1.concat(numeros2)
< ▶ (8) [1, 2, 3, 4, 5, 6, 7, 8]
>
```