

Bootcamp 2024 - Loops

Bucles

Una de las principales ventajas de la programación es la posibilidad de crear **bucles y repeticiones** para tareas específicas, y que no tengamos que realizar el mismo código varias veces de forma manual.

Tipos de Bucles

Tipo de bucle	Descripción
While	Bucles simples
For	Bucles clasicos
Do.. While	Bucles simples que se realizan como minimo una vez
For.. in	Bucles sobre posiciones de un array
For.. of	Bucles sobre elementos de un array

Conceptos de bucles



- **Condición:** Al igual que en los condicionales if, en los bucles se evalúa una condición para determinar si se debe continuar repitiendo el bucle o finalizarlo. Generalmente, se establece que si la condición es verdadera, se repite el bucle, y si es falsa, se termina. Sin embargo, esta condición puede variar según la implementación que el programador indique.
- **Iteración:** Otro concepto fundamental en los bucles es la iteración, que se refiere a cada repetición del bucle. Es importante tener en cuenta que en programación se suele comenzar a contar desde 0, por lo que la primera iteración es la número 0. Si el bucle va de 0 a 3, habrá 4 iteraciones en total.
- **Contador:** A menudo, los bucles incorporan un contador, que es una variable que lleva un registro del número de repeticiones realizadas y se utiliza para determinar cuándo finalizar el bucle. Este contador debe inicializarse (crearse y asignársele un valor) antes de iniciar el bucle.
- **Incremento:** Además del contador, es necesario incluir una parte en el bucle donde se incremente (o decremente) dicho contador. Sin esta parte, el contador no cambiaría y la condición siempre sería verdadera, lo que resultaría en un bucle infinito.
- **Bucle infinito:** Es común cometer errores al crear bucles, lo que puede llevar a quedar atrapado en un bucle infinito, es decir, una situación en la que el programa queda atrapado en el bucle y nunca termina. Como programadores, es crucial evitar esta situación. Para ello, es necesario asegurarse siempre de que exista un incremento (o decremento) y que en algún momento la condición se vuelva falsa para poder salir del bucle.

Bucle While

El bucle **while** es uno de los bucles más simples que podemos crear. Vamos a repasar el siguiente ejemplo y analizar todas sus partes, para luego analizar lo que ocurre en cada iteración del bucle. Empecemos por un fragmento sencillo del bucle:

```

1 let i = 0; // Inicialización de la variable contador
2
3 // Condición: Mientras la variable contador sea menor de 5
4 while (i < 5) {
5     console.log("Valor de i:", i);
6
7     i = i + 1; // Incrementamos el valor de i
8 }

```

Veamos que es lo que ocurre a la hora de ejecutar este código:

1. Antes de entrar en el bucle `while`, se inicializa la variable `i` al valor `0`.
2. Antes de realizar la primera **iteración** del bucle, comprobamos la **condición**.
3. Si la condición es **verdadera**, hacemos las tareas que están indentadas dentro del bucle.
4. Mostramos por pantalla el valor de `i`.
5. Luego, incrementamos el valor de `i` sumándole `1` a lo que ya teníamos en `i`.
6. Terminamos la iteración del bucle, por lo que volvemos al inicio del `while` a hacer una **nueva iteración**.
7. Volvemos al punto 2) donde comprobamos de nuevo la **condición** del bucle.
8. Repetimos hasta que la condición sea **falsa**. Entonces, salimos del bucle y continuamos el programa.

Paso a Paso dentro del bucle

Iteración del bucle	Valor de i	Descripción	Incremento
Antes de empezar	i = undefined	Antes de comenzar el programa.	
Iteración #1	i = 0	¿(0 < 5)? Verdadero. Mostramos 0 por pantalla.	i = 0 + 1
Iteración #2	i = 1	¿(1 < 5)? Verdadero. Mostramos 1 por pantalla.	i = 1 + 1
Iteración #3	i = 2	¿(2 < 5)? Verdadero. Mostramos 2 por pantalla.	i = 2 + 1
Iteración #4	i = 3	¿(3 < 5)? Verdadero. Mostramos 3 por pantalla.	i = 3 + 1
Iteración #5	i = 4	¿(4 < 5)? Verdadero. Mostramos 4 por pantalla.	i = 4 + 1
Iteración #6	i = 5	¿(5 < 5)? Falso. Salimos del bucle.	

El bucle `while` es muy simple, pero requiere no olvidarse accidentalmente de la inicialización y el incremento (*además de la condición*).

Más adelante veremos otro tipo de bucle denominado bucle `for`, que tiene una sintaxis diferente y, habitualmente, se suele utilizar más en el día a día.

La operación `i = i + 1` es lo que se suele llamar un incremento de una variable. Es muy común simplificarla como `i++`, que hace exactamente lo mismo: aumenta en 1 su valor.

Bucle Do While

Existe una variación del bucle `while` denominado **bucle do while**. La diferencia fundamental, a parte de variar un poco la sintaxis, es que este tipo de bucle siempre se ejecuta una vez, al contrario que el bucle `while` que en algún caso podría no ejecutarse nunca.

Para entenderlo bien, antes de ver el bucle `do while` vamos a analizar el siguiente ejemplo:

```
1 let i = 5;
2
3 while (i < 5) {
4   console.log("Ingresamos al Bucle");
5   i = i + 1;
6 }
7
8 console.log("Bucle finalizado");
```

Observa, que aún teniendo un bucle, este ejemplo nunca mostrará el texto `Ingresamos al Bucle`, puesto que la condición nunca será verdadera, porque ya ha empezado como falsa (`i` ya vale `5` desde el inicio). Por lo tanto, nunca se llega a realizar el interior del bucle.

Con el bucle `do while` podemos obligar a que **siempre** se realice el interior del bucle **al menos una vez**:

```
1 let i = 5;
2
3 do {
4   console.log("Ingresamos al bucle");
5   i = i + 1;
6 } while (i < 5);
7
8 console.log("Bucle finalizado");
```

1. En lugar de utilizar un `while` desde el principio junto a la condición, escribimos `do`.
2. El `while` con la condición se traslada al final del bucle.
3. Lo que ocurre en este caso es que el interior del bucle se realiza **siempre**, y sólo se analiza la condición al terminar el bucle, por lo que aunque no se cumpla, se va a realizar **al menos una vez**.

Este tipo de bucle se usa cuando queremos establecer un bucle que siempre se inicie independientemente de la variable inicial, al menos una vez.

Bucle For

Este bucle se caracteriza en que se va a repetir, revisando la condición en cada iteración, hasta que no se cumpla la condición propuesta.

El bucle **for** es quizás uno de los más utilizados en el mundo de la programación. En Javascript se utiliza exactamente igual que en otros lenguajes como Java o C/C++. Veamos un ejemplo muy similar al que hemos realizado con el bucle `while` en el tema anterior:

```
1 // for (inicialización; condición; incremento)
2 for (let i = 0; i < 5; i++) {
3   console.log("Valor de i:", i);
4 }
```

Como vemos, la sintaxis de un **bucle for** es mucho más compacta y rápida de escribir que la de un **bucle while**, sin embargo puede parecernos más críptica cuando la vemos por primera vez.

La sintaxis del bucle `for` es mucho más práctica porque te obliga a escribir la **inicialización**, la **condición** y el **incremento** antes del propio bucle, y eso hace que no te olvides de estos tres puntos fundamentales, cosa que suele ocurrir en los bucles `while`, lo que suele desembocar en un bucle infinito. Aunque también puede ocurrir en el bucle `for`, suele ser menos habitual.

Analicemos la sintaxis del bucle:

1. Separemos por `;` lo que establecemos dentro de los paréntesis del `for`.
2. Lo primero es la inicialización `let i = 0`. Esto ocurre sólo una vez antes de empezar el bucle.
3. Lo segundo es la condición `i < 5`. Esto se comprueba al principio de cada iteración.
4. Lo tercero es el incremento `i++`, es decir, `i = i + 1`. Esto ocurre al final de cada iteración.

Si lo pensamos bien, es lo mismo que hacemos en el bucle `while`, pero escribiéndolo de otra forma

i Recordá que en programación es muy habitual empezar a contar desde **cero**. Mientras que en la vida real se contaría **desde 1 hasta 10**, en programación se contaría **desde 0 hasta 9**.

Decremento [↗](#)

No nos acostumbremos a hacer los bucles de memoria, ya que las condiciones pueden variar y ser bien diferentes. Por ejemplo, vamos a hacer un bucle que en lugar de incrementar su contador, se decremente, ya que nos interesa hacer una cuenta atrás:

```
1 for (let i = 5; i > 0; i--) {
2   console.log("Valor de i:", i);
3 }
```

En este caso, vamos a arrancar el bucle con un valor de `i` de `5`. Repetiremos la iteración varias veces, y observa que el incremento que tenemos es en su lugar un **decremento**, por lo que en lugar de sumarle `1`, lo restamos. El valor de `i` iría desde `5`, a `4`, `3`, `2`, `1` y cuando se reduzca a `0`, ya no cumpliría la condición `i > 0`, por lo que terminaría el bucle, saldría de él y continuaría el resto del programa.

Interrumpir Bucles [↗](#)

Por norma general, un buen hábito de programación cuando creamos bucles es pensar la forma de hacer que esos bucles siempre vayan desde un número inicial a un número final y terminen las repeticiones. De esta forma son predecibles y fáciles de leer. Sin embargo, en algunas ocasiones nos puede interesar hacer interrupciones o saltos de iteraciones para conseguir algo más específico que con un bucle íntegro es más complejo de conseguir.

- **Continue:**

Imagina un caso donde queremos saltarnos una iteración concreta. Por ejemplo, queremos hacer un bucle que muestre los números del `0` al `10`, pero queremos que el número `5` se lo salte y no lo muestre, continuando con el resto.

Para eso podemos hacer uso de `continue`, que es una sentencia que al llegar a ella dentro de un bucle, el programa **salta** y abandona esa iteración, volviendo al principio del bucle:

```
1 for (let i = 0; i < 11; i++) {
2   if (i == 5) {
3     continue;
4   }
5
6   console.log("Valor de i:", i);
7 }
```

Así pues, en este caso se van a realizar las iteraciones desde `0` hasta `4`, en la iteración del `5` se entrará en el `if` y se evaluará como verdadera, ejecutando el `continue` y saltando a la siguiente iteración. Por lo tanto continuará desde la iteración `6` hasta la `10`.

En resumen, la iteración `5` nunca llegará al `console.log`, por lo que no se mostrará por pantalla.

- **Break :**

De la misma forma que tenemos un `continue` que interrumpe la iteración y vuelve al inicio a evaluar la condición, tenemos un `break` que nos permite interrumpir el bucle y abandonarlo. Esto puede ser bastante útil cuando queremos que se abandone el bucle por una condición especial.

Por ejemplo, cambiemos el `continue` del ejemplo anterior por el `break` :

```
1 for (let i = 0; i < 11; i++) {
2   if (i == 5) {
3     break;
4   }
5
6   console.log("Valor de i:", i);
7 }
```

En este caso, se van a realizar las iteraciones desde el `0` al `4`, y cuando lleguemos a la iteración `5`, se entrará en el `if` y como cumple su condición, se hará un `break` y se abandonará el `for`, continuando el resto del programa.

```
1 let i = 0;
2
3 while (i < 11) {
4   if (i == 5) {
5     break;
6   }
7
8   console.log("Iteración número ", i);
9   i = i + 1;
10 }
11
12 console.log("Bucle finalizado.")
```

Observa que en este caso, no hay conflicto con los bucle `while`, ya que el `break` no vuelve a evaluar la condición del bucle, sino que directamente abandona el bucle `while`, por lo que es seguro utilizarlo tanto en `for` como en `while`.

For Of [↗](#)

Ya vimos sobre bucles While y for, pero para los arrays también hay otra forma de bucle, `for...of` :

```
1 let fruits = ["Apple", "Orange", "Plum"];
2
3 // itera sobre los elementos del array
4 for (let fruit of fruits) {
5   alert( fruit );
6 }
```

`for...of` no da acceso al número del elemento en curso, solamente a su valor, pero en la mayoría de los casos eso es suficiente. Y es más corto.

For In [↗](#)

Usado en objetos [↗](#)

Para recorrer todas las claves de un objeto existe una forma especial de bucle: `for...in`. Esto es algo completamente diferente a la construcción `for(;;)`

La sintaxis:

```

1 for (key in object) {
2   // se ejecuta el cuerpo para cada clave entre las propiedades del objeto
3 }

```

Por ejemplo, mostremos todas las propiedades de `user` :

```

1 let user = {
2   name: "John",
3   age: 30,
4   isAdmin: true
5 };
6
7 for (let key in user) {
8   // claves
9   alert( key ); // name, age, isAdmin
10  // valores de las claves
11  alert( user[key] ); // John, 30, true
12 }

```

Ordenado como un objeto [↗](#)

¿Los objetos están ordenados? Es decir, si creamos un bucle sobre un objeto, ¿obtenemos todas las propiedades en el mismo orden en el que se agregaron? ¿Podemos confiar en ello?

La respuesta corta es: "ordenados de una forma especial": las propiedades de números enteros se ordenan, los demás aparecen en el orden de la creación. Entremos en detalle.

Como ejemplo, consideremos un objeto con códigos telefónicos:

```

1 let codes = {
2   "49": "Germany",
3   "41": "Switzerland",
4   "44": "Great Britain",
5   // ..,
6   "1": "USA"
7 };
8
9 for (let code in codes) {
10  alert(code); // 1, 41, 44, 49
11 }

```

El objeto puede usarse para sugerir al usuario una lista de opciones. Si estamos haciendo un sitio principalmente para el público alemán, probablemente queremos que `49` sea el primero.

Pero si ejecutamos el código, veremos una imagen totalmente diferente:

- USA (1) va primero
- Luego Switzerland (41) y así sucesivamente.

Los códigos telefónicos van en orden ascendente porque son números enteros. Entonces vemos `1, 41, 44, 49` .

...Por otro lado, si las claves no son enteras, se enumeran en el orden de creación, por ejemplo:

```

1 let user = {
2   name: "John",
3   surname: "Smith"

```

```

4  };
5  user.age = 25; // Se agrega una propiedad más
6
7  // Las propiedades que no son enteras se enumeran en el orden de creación
8  for (let prop in user) {
9    alert( prop ); // name, surname, age
10 }

```

Usado en arrays. [↗](#)

Técnicamente, y porque los arrays son objetos, es también posible usar `for...in`:

El ciclo **For in** tiene una sintaxis muy sencilla y fácil de recordar. Este tipo de *for*, usado de forma "simple" solo va a imprimir los índices del array que se le asigne.

```

1  let elementos = ["Agua", "Tierra", "Fuego", "Aire"];
2
3  for (elemento in elementos){
4    console.log(elemento);
5  }
6
7  //Imprime los índices 0,1,2,3

```

Aunque se puede jugar con los datos dentro de este ciclo para imprimir los valores que contenga el array y no sus índices.

```

1  let elementos = ["Agua", "Tierra", "Fuego", "Aire"];
2
3  for (elemento in elementos){
4    console.log(elementos[elemento]);
5  }
6
7  //Imprime los valores Agua, Tierra, Fuego y Aire
8
9  //Es muy parecido a hacer algo como elementos[i] en un for loop normal

```

Pero es una mala idea. Existen problemas potenciales con esto:

1. El bucle `for...in` itera sobre *todas las propiedades*, no solo las numéricas.

Existen objetos "simil-array" en el navegador y otros ambientes que *parecen arrays*. Esto es, tienen `length` y propiedades indexadas, pero pueden también tener propiedades no numéricas y métodos que usualmente no necesitamos. Y el bucle `for...in` los listará. Entonces si necesitamos trabajar con objetos simil-array, estas propiedades "extras" pueden volverse un problema.

2. El bucle `for...in` está optimizado para objetos genéricos, no para arrays, y es de 10 a 100 veces más lento. Por supuesto es aún muy rápido. Una optimización puede que solo sea importante en cuellos de botella, pero necesitamos ser conscientes de la diferencia.

En general, no deberíamos usar `for...in` en arrays.

For Each [↗](#)

El método `forEach` de JavaScript es una de las varias formas de recorrer un arreglo. Cada método tiene diferentes características, y depende de vos, dependiendo de lo que estés haciendo, decidir cuál usar.

Teniendo en cuenta que tenemos el siguiente arreglo a continuación:

```

1  const numeros = [1, 2, 3, 4, 5];

```

Utilizando el tradicional "bucle for" para recorrer el arreglo sería así:

```
1 for (i = 0; i < numeros.length; i++) {  
2   console.log(numeros[i]);  
3 }
```

El método `forEach` pasa una función callback para cada elemento del arreglo junto con los siguientes parámetros:

- Valor actual (requerido) - El valor del elemento actual del arreglo
- Index (opcional) - El número de índice del elemento actual
- Arreglo (opcional) - El objeto del arreglo al que pertenece el elemento actual

Permítame explicar estos parámetros paso a paso.

En primer lugar, para recorrer un arreglo utilizando el método `forEach`, se necesita una función callback (o función anónima):

```
1 numeros.forEach(function() {  
2   // código  
3 });
```

La función se ejecutará para cada elemento del arreglo. Debe tomar al menos un parámetro que represente los elementos del arreglo:

```
1 numeros.forEach(function(numero) {  
2   console.log(numero);  
3 });
```

Eso es todo lo que tenemos que hacer para recorrer el arreglo:

```
1  
2  
3  
4  
5
```

Como alternativa, puede utilizar la representación de la función de flecha de ES6 para simplificar el código:

```
1 numeros.forEach(numero => console.log(numero));
```

Parámetros opcionales [↗](#)

Index [↗](#)

Bien, ahora continuemos con los parámetros opcionales. El primero es el parámetro "index", que representa el número de índice de cada elemento.

Básicamente, podemos ver el número de índice de un elemento si lo incluimos como segundo parámetro:

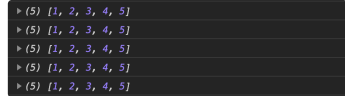
```
1 numeros.forEach((numero, index) => {  
2   console.log('Indice: ' + index + ' Valor: ' + numero);  
3 });
```

```
Indice: 0 Valor: 1  
Indice: 1 Valor: 2  
Indice: 2 Valor: 3  
Indice: 3 Valor: 4  
Indice: 4 Valor: 5
```


Arreglo

El parámetro del arreglo es el propio arreglo. También es opcional y se puede utilizar si es necesario en varias operaciones. En caso contrario, si lo llamamos, simplemente se imprimirá tantas veces como el número de elementos del arreglo:

```
1 numeros.forEach((numero, index, arreglo) => {  
2   console.log(arreglo);  
3 });
```



Map

A veces es necesario tomar un arreglo y aplicar algún procedimiento a sus elementos para obtener un nuevo arreglo con elementos modificados.

En lugar de iterar manualmente sobre el arreglo usando un bucle, puedes simplemente usar el método incorporado `Array.map()`.

El método `Array.map()` te permite iterar sobre un arreglo y modificar sus elementos utilizando una función callback. La función callback se ejecutará entonces en cada uno de los elementos del arreglo.

Por ejemplo, supón que tienes el siguiente elemento de arreglo:

```
1 let arr = [3, 4, 5, 6];
```

Un simple arreglo de JavaScript

Ahora imagina que tienes que multiplicar cada uno de los elementos del arreglo por 3. Podrías considerar usar un bucle `for` como el siguiente:

```
1 let arr = [3, 4, 5, 6];  
2  
3 for (let i = 0; i < arr.length; i++){  
4   arr[i] = arr[i] * 3;  
5 }  
6  
7 console.log(arr); // [9, 12, 15, 18]
```

Iterar sobre un arreglo utilizando el bucle for

Pero en realidad puedes utilizar el método `Array.map()` para conseguir el mismo resultado. He aquí un ejemplo:

```
1 let arr = [3, 4, 5, 6];  
2  
3 let modifiedArr = arr.map(function(element){  
4   return element * 3;  
5 });  
6  
7 console.log(modifiedArr); // [9, 12, 15, 18]
```

Iterar sobre un arreglo utilizando el método map()

El método `Array.map()` se utiliza comúnmente para aplicar algunos cambios a los elementos, ya sea multiplicando por un número específico como en el código anterior, o haciendo cualquier otra operación que pueda necesitar para su aplicación.

Cómo utilizar map() sobre un arreglo de objetos [↗](#)

Por ejemplo, puedes tener un arreglo de objetos que almacene los valores de `firstName` y `lastName` de tus amigos de la siguiente manera:

```
1 let users = [  
2   {firstName : "Susan", lastName: "Steward"},  
3   {firstName : "Daniel", lastName: "Longbottom"},  
4   {firstName : "Jacob", lastName: "Black"}  
5 ];  
6  
7
```

Un arreglo de objetos

Puedes utilizar el método `map()` para iterar sobre el arreglo y unir los valores de `firstName` y `lastName` de la siguiente manera:

```
1 let users = [  
2   {firstName : "Susan", lastName: "Steward"},  
3   {firstName : "Daniel", lastName: "Longbottom"},  
4   {firstName : "Jacob", lastName: "Black"}  
5 ];  
6  
7 let userFullnames = users.map(function(element){  
8   return `${element.firstName} ${element.lastName}`;  
9 })  
10  
11 console.log(userFullnames);  
12 // ["Susan Steward", "Daniel Longbottom", "Jacob Black"]
```

Utiliza el método `map()` para iterar sobre un arreglo de objetos

El método `map()` pasa algo más que un elemento. Veamos todos los argumentos que pasa `map()` a la función callback.

La sintaxis completa del método map() [↗](#)

La sintaxis del método `map()` es la siguiente:

```
1 arr.map(function(element, index, array){ }, this);
```

La `function()` callback es llamada en cada elemento del arreglo, y el método `map()` siempre le pasa el `element` actual, el `index` del elemento actual y el objeto `array` completo.

El argumento `this` se utilizará dentro de la función callback. Por defecto, su valor es `undefined`. Por ejemplo, aquí está cómo cambiar el valor de `this` al número `80`:

```
1 let arr = [2, 3, 5, 7]  
2  
3 arr.map(function(element, index, array){  
4   console.log(this) // 80  
5 }, 80);
```

Asignación de valor numérico al método `map()` en este argumento

También puedes probar los otros argumentos usando `console.log()` si te interesa:

```
1 let arr = [2, 3, 5, 7]  
2  
3 arr.map(function(element, index, array){  
4   console.log(element);
```

```
5 console.log(index);
6 console.log(array);
7 return element;
8 }, 80);
```

PPT Introducción a Loops



Artículos relacionados:

- [🔗 Bucles: while y for](#)
- [📺 Learn JavaScript FOR LOOPS in 5 minutes! 🔄](#)
- [📺 Learn JavaScript WHILE LOOPS in 8 minutes! 🔄](#)
- [📺 🧑 ESTRUCTURA REPETITIVA FOR EN JAVASCRIPT | ⭐ Curso JAVASCRIPT DESDE CERO 🚀 #11](#)
- [📺 🧑 ESTRUCTURA REPETITIVA DO WHILE en JAVASCRIPT | ⭐ Curso JAVASCRIPT DESDE CERO 🚀 #12](#)
- [📺 ¿FOR .. IN 🤔 FOR .. OF? 🤔Cuál DEBES utilizar! 🤔 Curso de JAVASCRIPT desde CERO #11](#)
- [📖 do...while - JavaScript | MDN](#)
- [📖 break - JavaScript | MDN](#)
- [📖 for - JavaScript | MDN](#)
- [📖 for...in - JavaScript | MDN](#)
- [📖 while - JavaScript | MDN](#)
- [🔗 ¿Por qué forEach es más rápido que for en JavaScript?](#)
- [📺 Cómo ITERAR un OBJETO en JAVASCRIPT . Respuesta de API en forma de objeto](#)
- [📖 Array.prototype.map\(\) - JavaScript | MDN](#)
- [📖 JavaScript map: Cómo utilizar la función JS .map\(\) \(método de Arreglo\)](#)

Ejercicios interactivos: 🔗

- [🌐 W3Schools JS Exercise](#)