
Ciclos e iterables

A medida que comenzamos a escribir un código más complicado, es necesario y deseable no repetirnos. Este concepto en programación se conoce como seco (DRY), o no se repite (**Don't Repeat Yourself**). Este concepto se vuelve realmente importante cuando trabajamos con grandes conjuntos de datos. Por ejemplo, imagina que te encuentras con ganas de crear 10 nuevas filas en una tabla. Puedes escribir el mismo código diez veces seguidos, pero eso no sería eficiente ni un buen uso de tu tiempo.

Para evitar repetirnos, usamos ciclos e iteración, y JavaScript tiene varias formas de realizar ambas tareas. En este documento, veremos cómo podemos agregarlos además de lo que hemos aprendido hasta ahora.

Ciclos

Hay tres formas de ciclos en JavaScript. El primero que veremos es la cláusula *While*. La cláusula *while* acepta una declaración lógica, y aunque esa declaración lógica es verdadera, el código dentro de las `{ }` se continuará ejecutando. En el siguiente ejemplo, el resultado es que `console.log` se ejecutará diez veces:

```
let x = 1
while(x < 10) {
  console.log(x)
  ++x
}
```

Como no queremos que un ciclo *while* se ejecute indefinidamente, a menudo ajustamos la variable o la condición cada vez que se ejecuta el *while*. En el ejemplo anterior, cada vez que se ejecuta, también usamos el operador `++` para agregar 1 a `x`. `++ x` es shorthand, y es el equivalente a escribir `x += 1`. lo opuesto de `++` es `--`, que resta 1 de una variable. Si no hicieras esto, crearías un ciclo infinito, lo que finalmente causaría que tu código se rompa.

Un resumen de cómo funciona el ciclo *while* arriba se muestra en la figura 1.

El resultado de nuestro ciclo anterior se muestra en la figura 2. Dado que `(x < 10)` sigue siendo cierto hasta que `x` ha recibido 10, obtenemos 10 nuevas líneas en el registro de la consola.

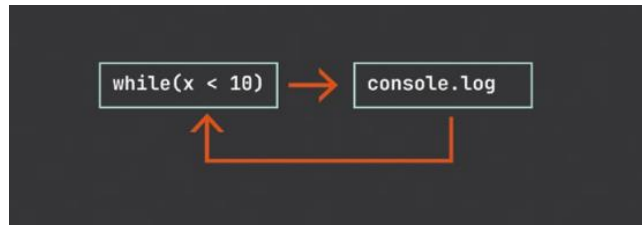


Figura 1: Un ciclo while seguirá ejecutando código hasta que la declaración lógica ya no sea cierta. En este ejemplo, si no agregas o multiplicas algo por x, la declaración lógica siempre seguirá siendo verdadera, lo que lleva a un ciclo infinito y un error.

```
> let x = 1
  while(x < 10) {
    console.log(x)
    ++x
  }
1
2
3
4
5
6
7
8
9
< 10
>
```

Figura 2: Al usar el ciclo while, podemos hacer una acción varias veces, sin tener que escribir lo que queremos hacer muchas veces. Es importante modificar a un participante en tu condición while; De lo contrario, puedes obtener un ciclo infinito.

Una forma alternativa de realizar un ciclo while usando una sintaxis similar es con do...while. Funciona de la misma manera, excepto que la condición se verifica después de que se ejecuta el código, lo que significa que el código siempre se ejecutará al menos una vez. En el siguiente ejemplo, el código ejecuta un registro de consola que dice "hello world". Con un ciclo normal, no se ejecutaría en absoluto ya que x no es inferior a 1:

```
let x = 1
do {
  console.log("hello world")
} while(x < 1)
```

La última forma en que podemos realizar un ciclo es con una declaración for. Para la declaración combina la definición de la variable, la declaración lógica y la modificación, todo

en una declaración. En el siguiente ejemplo, recreamos la declaración `while` de la figura 1, pero usando en su lugar un ciclo `for`:

```
for(let x = 1; x < 10; ++x) {
  console.log(x)
}
```

El ciclo `for` es más ordenado que un ciclo `while`, ya que todas sus condiciones se limitan a una línea. Funcionalmente, sin embargo, hace lo mismo que lo que hace `while`. La modificación definida en el ciclo `for`, que es `++ x`, solo se ejecutará después de que se haya ejecutado el cuerpo del ciclo. La tabla 1 resume cómo funciona el ciclo `for`.

Tabla 1: Un desglose para la declaración `for`.

Código	Sección	Descripción
<code>for</code>	Iniciador	Inicia el ciclo <code>for</code> .
<code>let x = 1</code>	Variable	Define una sola variable que se utilizará en el ciclo <code>for</code> .
<code>x < 10</code>	Condición	Mientras esto sigue siendo cierto, el cuerpo del ciclo <code>for</code> se ejecutará.
<code>++x</code>	Modificador	Alguna modificación que ocurre. No necesitas incluir la variable, pero generalmente lo hace. También podría ser algo como <code>x += 2</code> .
<code>console.log(x)</code>	Cuerpo	Se ejecuta mientras la condición sigue siendo cierta.

Break y continuación en ciclos

A veces, si se cumple una determinada condición en un ciclo, queremos que el ciclo deje de volver a funcionar de inmediato. Por ejemplo, supongamos que estamos agregando 2 a un valor y queremos que el ciclo se rompa si alcanza el valor 10. En ese caso, podemos usar una instrucción `if` para romper condicionalmente el ciclo:

```
for(let x = 0; x < 20; x += 2) {
  if(x === 10) break
  console.log(x)
}
```

Como rompemos antes del registro de la consola, este ciclo solo muestra 0, 2, 4, 6, 8. Otro concepto similar es continuar (`continue`), que, en lugar de romper todo el ciclo, solo rompe la iteración actual.

```
for(let x = 0; x < 20; x += 2) {
  if(x === 10) continue
  console.log(x)
}
```

El uso de `continuar` solo rompe la iteración actual, por lo que el código anterior ahora mostrará 0, 2, 4, 6, 8, 12, 14, 16, 18, solo omitiendo un registro de consola en el ciclo cuando el valor es 10.

Etiquetas de ciclo

A medida que nuestro código se vuelve más complicado, podemos terminar con ciclos dentro de los ciclos. Eso puede conducir a algunos problemas interesantes al usar `break`, ya que `break` y `continue` no saben qué ciclo deseas romper o continuar.

Para resolver esto, podemos usar etiquetas para definir qué ciclo se supone que debe romperse y continuar. Aquí hay un ejemplo de un ciclo etiquetado, donde usamos las etiquetas `xloop` y `yloop` para referirnos al ciclo exterior e interno, respectivamente:

```
xLoop: for(let x = 1; x < 4; x += 2) {  
  yLoop: for(let y = 1; y < 4; y += 2) {  
    console.log(`xLoop: ${x * y}`)  
  }  
  console.log(`yLoop: ${x}`)  
}
```

Si quisiéramos romper `xloop` desde dentro de `yloop`, podemos escribir `break xloop`. Esto tiene el efecto de romper todo `xloop` si el valor de `x * y` es 4, evitando cualquier línea de `console.log(x)`. Puedes ver la salida de esto en la figura 3:

```
xLoop: for(let x = 1; x < 4; x += 2) {  
  yLoop: for(let y = 1; y < 4; y += 2) {  
    if(x === 1 || x === 3) break xLoop;  
    console.log(`xLoop: ${x * y}`)  
  }  
  console.log(`yLoop: ${x}`)  
}
```

Nota: Mientras estamos usando las etiquetas `xloop` y `yloop` en el ejemplo anterior, puedes etiquetar tus ciclos de cualquier manera que mejor te parezca, ¡estos son solo nombres inventados!

Iteración

Al usar ciclos, generalmente estamos utilizando variables y declaraciones lógicas para iterar a través de declaraciones hasta que dejen de devolverlo. Esto nos impide repetirnos y simplifica nuestro código.

```

> xLoop: for(let x = 1; x < 4; x += 2) {
  yLoop: for(let y = 1; y < 4; y += 2) {
    if(x === 2) break xLoop;
    console.log(`xLoop: ${x * y}`)
  }
  console.log(`yLoop: ${x}`)
}
xLoop: 1
xLoop: 3
yLoop: 1
xLoop: 3
xLoop: 9
yLoop: 3

```

Figura 3: En este ejemplo, cuando x es igual a 2, xloop está roto. Eso significa que todo el ciclo solo se ejecuta dos veces. Si rompieras a yloop cuando x era 2, entonces xloop continuaría funcionando, mientras que yloop se habría detenido.

En el documento anterior, hablamos sobre cómo los objetos y los arreglos actúan como un almacén de datos en JavaScript. Eventualmente, queremos usar los datos contenidos por ellos en nuestro código.

Podemos acceder a cada uno individualmente, pero ¿qué pasa si queremos hacer algo con todos los elementos en un arreglo u objeto? Podríamos escribirlos en líneas separadas, como se muestra en el siguiente ejemplo, pero esto rápidamente se vuelve inmanejable a medida que los arreglos y los objetos se vuelven más grandes:

```

let myArray = [ "banana", "pineapples", "strawberries" ]
console.log(`I have 1 ${myArray[0]}`)
console.log(`I have 2 ${myArray[1]}`)
console.log(`I have 3 ${myArray[2]}`)

```

Para resolver este problema, algunos tipos de datos en JavaScript son iterables. Por ejemplo, los arreglos y las cadenas son iterables, pero los objetos no lo son. Otros tipos de datos como mapas y conjuntos también son iterables, pero los cubriremos con mucha más profundidad en futuros temas.

Iterables y ciclos for

Los datos iterables pueden tener sus valores completamente extraídos con el for...de o for...en ciclo, que asignará cada elemento de un iterable a una variable.

En el siguiente ejemplo, "ítem" se refiere a cada elemento individual en el arreglo. Cada elemento se registrará en una línea separada. Puedes ver cómo se ve esto en la figura 4.

```

let x = [ "lightning", "apple", "squid", "speaker" ]

```

```
for(let item of x) {
  console.log(item)
}
```

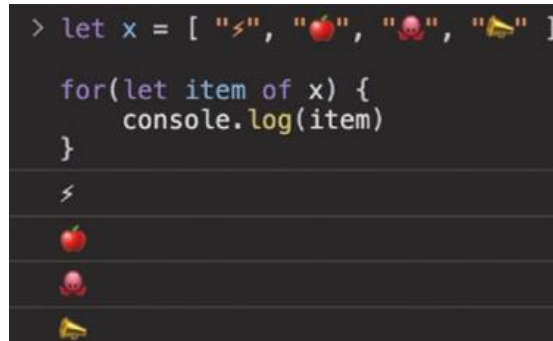


Figura 4: El ciclo for...de nos permitirá acceder a cada elemento de un arreglo o iterable, como se muestra en la imagen anterior.

for... en, por otro lado, se referirá al índice, y no al elemento en sí. En el siguiente ejemplo, la salida es 0, 1, 2, 3, mientras que en para ... de, fue rayo, manzana, calamar, altavoz como salida:

for...de y for...en difieren en la forma en que manejan valores indefinidos. Por ejemplo, considera el siguiente ejemplo en el que un arreglo consta de un solo elemento de arreglo en la ubicación 5. Cuando se usa for...de, devolverá 4 valores indefinidos, seguido de "algún valor":

```
let x = []
x[5] = "some value"
for(let item of x) {
  console.log(item)
  // Will show:
  // undefined, undefined, undefined, undefined, undefined
  // "some value"
}
```

Con for...en, solo obtenemos índices y no valores de arreglo, por lo que, por lo tanto, omite los valores indefinidos:

```
for(let item in x) {
  console.log(item)
  // Will show: 5
}
```

Métodos de arreglos forEach

Si bien no todos los tipos implementan un método para la iteración, los arreglos sí. Para ayudar con la iteración, los arreglos también tienen un método especial llamado `forEach`. Este método acepta una función con tres argumentos, como se muestra en el siguiente ejemplo:

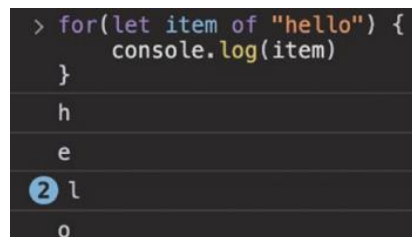
```
let x = [ "lightning", "apple", "squid", "speaker" ]
x.forEach(function(item, index, array) {
  console.log(`${item} is at index ${index}`)
})
```

El método `forEach` te proporciona el índice, el elemento (`item`) y el arreglo completo en cada iteración de tu arreglo. Si bien este método es útil, es más lento que un ciclo `for`, por lo que usar un ciclo `for` cuando puedes es tu mejor opción.

Iteración de cadena

Dado que las cadenas también son iterables, `for...de` y `for...en` también trabajan en ellas. En el siguiente ejemplo, usar `for...de` en la cadena "hello" produce una salida de todos los caracteres en esa cadena. El siguiente código en la consola log muestra individualmente cada letra, por lo que h, e, l, l, o. Puedes ver esta salida en la figura 5.

```
for(let item of "hello") {
  console.log(item)
}
```



```
> for(let item of "hello") {
  console.log(item)
}
h
e
2 l
o
```

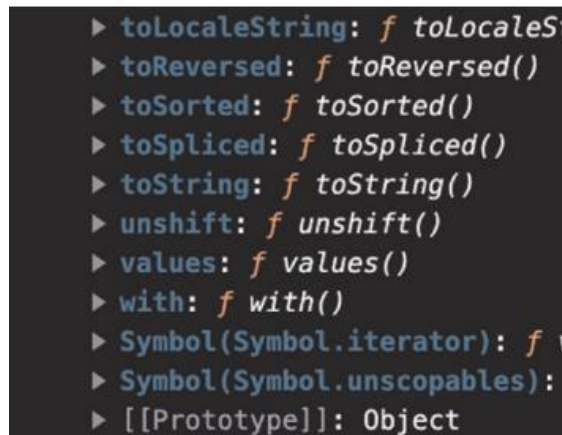
Figura 5: Usar `for...de` en una cadena también iterará a través de cada carácter en esa cadena. Si usas `for...en`, obtendrás un número en cada línea para cada letra.

Protocolo de iteración

En el documento anterior, discutimos cómo los diferentes tipos de datos, como los arreglos, heredan de un prototipo. Esto se llama herencia prototípica.

Cuando definimos un arreglo usando un `new Array()` o la notación de paréntesis cuadrado, estamos haciendo una nueva instancia de `Array`, que hereda muchos métodos y propiedades

de `Array.prototype`. Cualquier tipo de datos, como arreglos o cadenas, que tengan iterabilidad, heredarán una propiedad llamada `Symbol.iterator` de su cadena prototipo. Esto se conoce como el protocolo de iteración. Todos los tipos que tienen el protocolo de iteración son iterables. El protocolo de iteración es, por lo tanto, la razón por la que podemos usar `for...en` y `for...de` en arreglos y cadenas. Puedes ver la propiedad del protocolo de iteración en la figura 6.



```

▶ toLocaleString: f toLocaleSt
▶ toReversed: f toReversed()
▶ toSorted: f toSorted()
▶ toSpliced: f toSpliced()
▶ toString: f toString()
▶ unshift: f unshift()
▶ values: f values()
▶ with: f with()
▶ Symbol(Symbol.iterator): f v
▶ Symbol(Symbol.unscopables):
▶ [[Prototype]]: Object

```

Figura 6: Puedes ver el protocolo iterator mediante el registro de la consola de un prototipo iterable, como `console.log (Array.prototype)`.

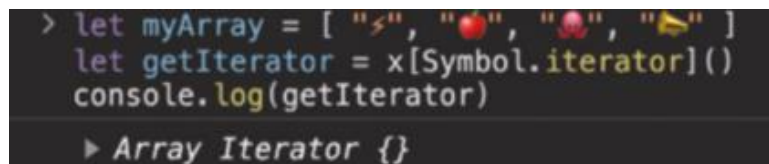
Otra característica genial, que viene con el protocolo de iteración, es que permiten que el iterable se atravesase. Puedes hacerlo accediendo a la llave del protocolo de iteración, `Symbol.iterator`, directamente. Dado que `Symbol.iterator` es solo otra llave heredada de `Array.Prototype`, podemos acceder a ella tal como lo habríamos ejecutado `myArray.at (-1)`:

```

let myArray = [ "lightning", "apple", "squid", "speaker" ]
let getIterator = myArray[Symbol.iterator]()
console.log(getIterator)

```

`Symbol.iterator` son dos palabras separadas por un punto, por lo que tenemos que acceder a él a través de paréntesis cuadrados. También lo ejecutamos en la función `sticking()` al final, ya que en realidad es una función. Los resultados de esto se pueden ver en la figura 7.



```

> let myArray = [ "🚩", "🍎", "🐙", "🔊" ]
let getIterator = x[Symbol.iterator]()
console.log(getIterator)
▶ Array Iterator {}

```

Figura 7: Al acceder al `Symbol.iterator`, creamos un nuevo objeto llamado `Array Iterator`. Este iterador nos permite atravesar un iterable.

Dentro del iterador del arreglo `[[Prototype]]`, encontrarás un método llamado `next()`. Esta función te permite iterar por un elemento a la vez y recuerda dónde en la secuencia lo dejaste por última vez. Por ejemplo, ejecutar `getIterator.next()` te dará un objeto que contiene si la iteración se realiza (que solo es verdadera si ha iterado a través de todos los valores) y el valor del siguiente elemento en la matriz:

```
let myArray = [ "lightning", "apple", "squid", "speaker" ]
let getIterator = myArray[Symbol.iterator]()
console.log(getIterator.next()) // { value: 'lightning', done: false }
```

Si sigues ejecutando `next()`, seguirá obteniendo el siguiente elemento, como se muestra en el siguiente ejemplo:

```
let myArray = [ "lightning", "apple", "squid", "speaker" ]
let getIterator = myArray[Symbol.iterator]()
console.log(getIterator.next()) // { value: 'lightning', done: false }
console.log(getIterator.next()) // { value: 'apple', done: false }
console.log(getIterator.next()) // { value: 'squid', done: false }
console.log(getIterator.next()) // { value: 'speaker', done: true }
console.log(getIterator.next()) // { value: undefined, done: true }
```

Los objetos no son iterables de forma predeterminada

Si intentas usar en la console log, `Object.prototype`, no encontrará `Symbol.Iterator`, que implica que los objetos no son iterables. La razón principal por la cual los objetos no son iterables de forma predeterminada es porque contienen *llaves* y *valores*. Decidir qué iterar o cómo formatear esa iteración es algo que JavaScript te deja.

Afortunadamente, hay una manera fácil de iterar a través de un objeto, y es convertirlo en un tipo de datos iterable, como un arreglo. Hay tres métodos para hacer esto:

- `Object.keys()`, que extrae todas las llaves como un arreglo.
- `Object.values()`, que extrae todos los valores como un arreglo.
- `Object.entries()`, que extrae todas las llaves y arreglos como un arreglo de pares de valores llave.

No hemos visto estos métodos de objetos antes, pero todos están disponibles en el objeto global `Object`. Considera el siguiente ejemplo, donde tenemos un arreglo llamado `myObject`. Podemos usar tanto `object.keys()` como `object.values()` directamente en este objeto para extraer un arreglo de cada uno:

```
let myObject = {
  firstName: "John",
  lastName: "Doe",
  age: 140
}
let myObjectKeys = Object.keys(myObject)
console.log(myObjectKeys) //[ "firstName", "lastName", "age" ]
let myObjectValues = Object.values(myObject)
console.log(myObjectValues) //[ "John", "Doe", 140 ]
```

Una vez que hemos extraído un arreglo, podemos ejecutar esto a través de un ciclo for ya que los arreglos son iterables. Esto se muestra en el siguiente ejemplo y también en la figura 8.

```
let myObject = {
  firstName: "John",
  lastName: "Doe",
  age: 140
}
let myObjectKeys = Object.keys(myObject)
for(let item of myObjectKeys) {
  console.log(item)
}
```



```
> let myObject = {
  firstName: "John",
  lastName: "Doe",
  age: 140
}

let myObjectKeys = Object.keys(myObject)
for(let item of myObjectKeys) {
  console.log(item)
}
```

firstName
lastName
age

Figura 8: Métodos como `Object.keys()` convierten los objetos en arreglos Iterables, que luego podemos iterar.

`Object.entries()` es ligeramente diferente de `Object.keys/values()`, ya que crea un arreglo de pares llave -valor. Puedes ver la salida del `Objeto.entries` cuando se aplican a `myObjectKeys` en la figura 9.

Usando la destrucción, que cubrimos en el documento anterior, podemos acceder a la llave y al valor de un objeto usando `Object.entries`. En el siguiente ejemplo, hacemos exactamente eso usando un ciclo `for... de`. La salida de este ciclo `for` también se puede ver en la figura 10.



```
> let myObject = {
  firstName: "John",
  lastName: "Doe",
  age: 140
}

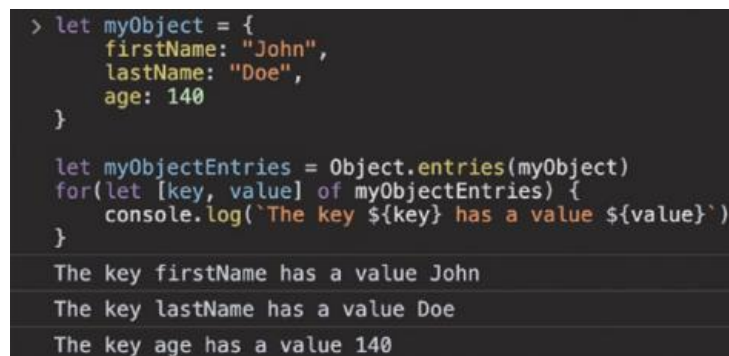
let myObjectEntries = Object.entries(myObject)
console.log(myObjectEntries)
```

The screenshot shows the output of the code in a dark-themed editor. The output is an array of three entries, each being an array of two elements: a key and a value. The entries are: `['firstName', 'John']`, `['lastName', 'Doe']`, and `['age', 140]`. The array has a length of 3 and a prototype of `Array(0)`.

Figura 9: `Object.entries` crea un arreglo de pares llave -valor. Cada par llave -valor en sí también es un arreglo.

```
let myObject = {
  firstName: "John",
  lastName: "Doe",
  age: 140
}

let myObjectEntries = Object.entries(myObject)
for(const [key, value] of myObjectEntries) {
  console.log(`The key ${key} has a value ${value}`)
}
```



```
> let myObject = {
  firstName: "John",
  lastName: "Doe",
  age: 140
}

let myObjectEntries = Object.entries(myObject)
for(let [key, value] of myObjectEntries) {
  console.log(`The key ${key} has a value ${value}`)
}
```

The screenshot shows the output of the code in a dark-themed editor. The output consists of three lines of text: `The key firstName has a value John`, `The key lastName has a value Doe`, and `The key age has a value 140`.

Figura 10: Los pares llave -valor generados por `Object.entries` se pueden acceder fácilmente utilizando la variable destructuring dentro de un ciclo `for`. Aquí recuperamos todas las llaves y valores en `myObject`.

Resumen

En este documento, hemos cubierto todos los diferentes tipos de ciclos en JavaScript. También hemos discutido cómo los `breaks`, `continues` y las etiquetas funcionan en el contexto de los ciclos. El ciclo no solo se limita a las declaraciones lógicas, por lo que también hemos entrado en muchos detalles sobre cómo iterar sobre arreglos, cadenas y cualquier otro tipo de datos que implique el protocolo de iteración.

Los objetos, el tipo de datos más importante en JavaScript, no son directamente iterables. Sin embargo, podemos convertirlos en arreglos iterables a través de métodos como `Object.entries()`.

A lo largo del desarrollo de aplicaciones en el curso, necesitaremos acceder a arreglos, objetos y otros iterables. Comprender estos conceptos hace que escribir código sea más eficiente, y usaremos el código que hemos aprendido aquí en futuros temas.