

Esta página ha sido traducida del inglés por la comunidad. Aprende más y únete a la comunidad de MDN Web Docs.

Código de bucle

Los lenguajes de programación son muy útiles para completar rápidamente tareas repetitivas, desde múltiples cálculos básicos hasta casi cualquier otra situación en la que tenga que completar muchos elementos de trabajo similares. Aquí veremos las estructuras de bucle disponibles en JavaScript que manejan tales necesidades.

Prerrequisitos:	Conocimientos básicos de informática, una comprensión básica de HTML y CSS, Primeros pasos de JavaScript .
Objetivo:	Comprender cómo usar bucles en JavaScript.

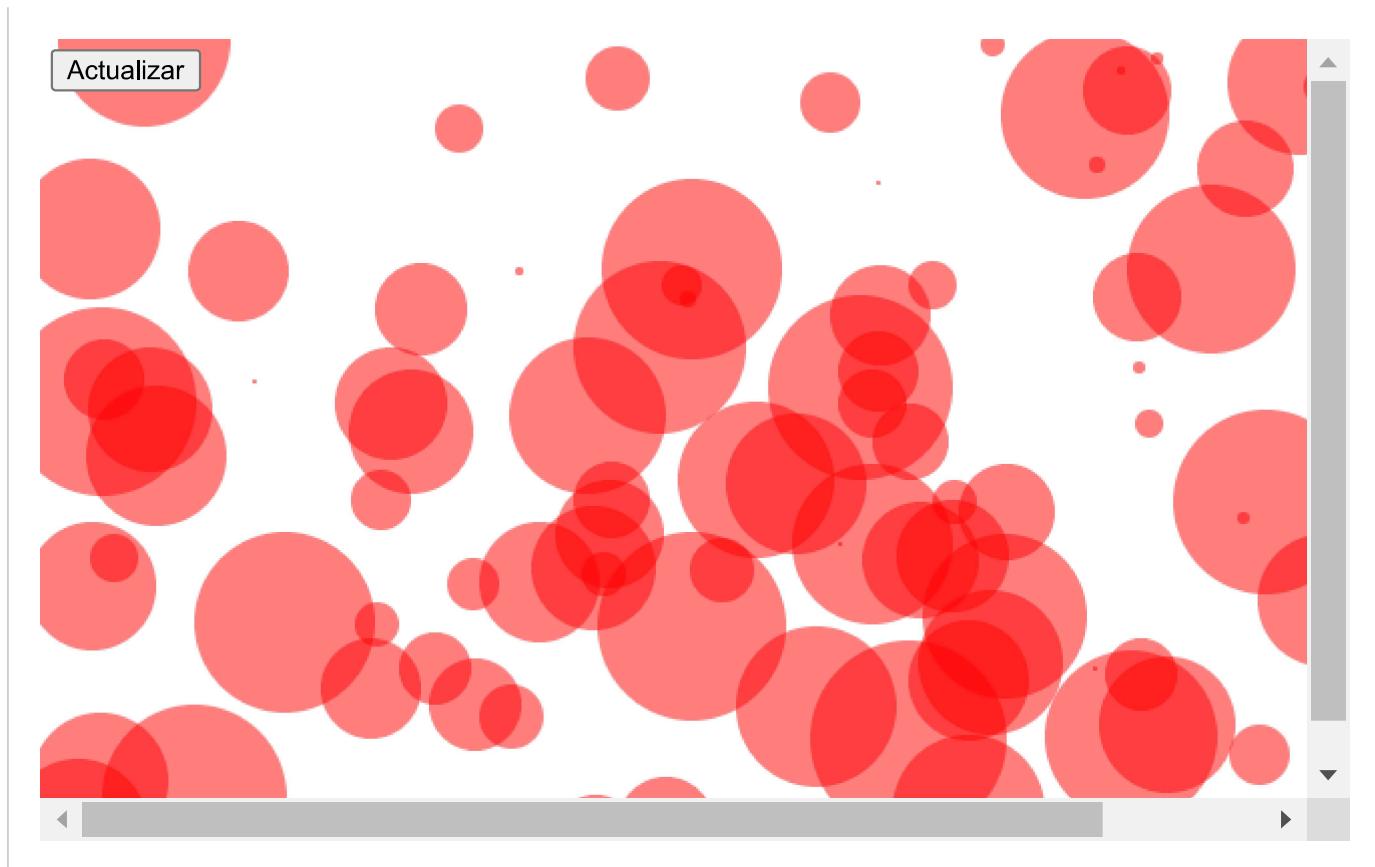
¿Por qué son útiles los bucles?

Los bucles se tratan de hacer lo mismo una y otra vez. A menudo, el código será ligeramente diferente cada vez que dure el bucle, o se ejecutará el mismo código pero con diferentes variables.

Ejemplo de código de bucles

Supongamos que queremos dibujar 100 círculos aleatorios en un elemento [`<canvas>`](#) (pulse el botón *Actualizar* para ejecutar el ejemplo una y otra vez para ver diferentes conjuntos aleatorios):

Play



Este es el código JavaScript que implementa este ejemplo:

JS

Play

```
const btn = document.querySelector("button");
const canvas = document.querySelector("canvas");
const ctx = canvas.getContext("2d");

document.addEventListener("DOMContentLoaded", () => {
  canvas.width = document.documentElement.clientWidth;
  canvas.height = document.documentElement.clientHeight;
});

function random(number) {
  return Math.floor(Math.random() * number);
}

function draw() {
  ctx.clearRect(0, 0, canvas.width, canvas.height);
  for (let i = 0; i < 100; i++) {
    ctx.beginPath();
    ctx.fillStyle = "rgba(255,0,0,0.5)";
    ctx.arc(
      random(canvas.width),
```

```

    random(canvas.height),
    random(50),
    0,
    2 * Math.PI,
);
ctx.fill();
}

}

btn.addEventListener("click", draw);

```

Con y sin bucles

No tienes que entender todo el código por ahora, pero veamos la parte del código que realmente dibuja los 100 círculos:

JS

```

for (let i = 0; i < 100; i++) {
  ctx.beginPath();
  ctx.fillStyle = "rgba(255,0,0,0.5)";
  ctx.arc(
    random(canvas.width),
    random(canvas.height),
    random(50),
    0,
    2 * Math.PI,
  );
  ctx.fill();
}

```

- `random(x)`, definido anteriormente en el código, devuelve un número entero entre `0` y `x-1`.

Deberías tener la idea básica: estamos usando un bucle para ejecutar 100 iteraciones de este código, cada una de las cuales dibuja un círculo en una posición aleatoria en la página. La cantidad de código necesaria sería la misma si estuviéramos dibujando 100 círculos, 1000 o 10.000. Solo un número tiene que cambiar.

Si no estuviéramos usando un bucle aquí, tendríamos que repetir el siguiente código para cada círculo que quisiéramos dibujar:

JS

```
ctx.beginPath();
ctx.fillStyle = "rgba(255,0,0,0.5)";
ctx.arc(
  random(canvas.width),
  random(canvas.height),
  random(50),
  0,
  2 * Math.PI,
);
ctx.fill();
```

Esto sería muy aburrido y difícil de mantener.

Recorriendo una colección

La mayoría de las veces, cuando usas un bucle, tendrás una colección de artículos y querrás hacer algo con cada artículo.

Un tipo de colección es el [Arreglo](#), que conocimos en el capítulo de [Arreglos](#) de este curso. Pero también hay otras colecciones en JavaScript, como [Set](#) y [Map](#) (inglés).

 mdn web docs

La herramienta básica para recorrer una colección es el bucle [for...of](#):

JS

```
const cats = ["Leopardo", "Serval", "Jaguar", "Tigre", "Caracal", "León"];

for (const cat of cats) {
  console.log(cat);
}
```

En este ejemplo, `for (const cat of cats)` dice:

1. Dada la colección `cats`, consigue el primer artículo de la colección.
2. Asignarlo a la variable `cat` y luego ejecutar el código entre las llaves `{}`.

3. Obtén el siguiente elemento y repite (2) hasta que hayas llegado al final de la colección.

map() y filter()

JavaScript también tiene bucles más especializados para colecciones, y mencionaremos dos de ellos aquí.

Puede usar `map()` para hacer algo con cada elemento de una colección y crear una nueva colección que contenga los elementos modificados:

JS

```
function toUpper(string) {  
    return string.toUpperCase();  
}  
  
const cats = ["Leopardo", "Serval", "Jaguar", "Tigre", "Caracal", "León"];  
  
const upperCats = cats.map(toUpper);  
  
console.log(upperCats);  
// [ "LEOPARDO", "SERVAL", "JAGUAR", "TIGRE", "CARACAL", "LEÓN" ]
```

Aquí pasamos una función a `cats.map()`, y `map()` llama a la función una vez por cada elemento de la matriz, pasando el elemento. A continuación, añade el valor devuelto de cada llamada de función a un nuevo arreglo y, finalmente, devuelve el nuevo arreglo. En este caso, la función que proporcionamos convierte el elemento en mayúsculas, por lo que la matriz resultante contiene todos nuestros gatos en mayúsculas:

JS

```
[ "LEOPARDO", "SERVAL", "JAGUAR", "TIGRE", "CARACAL", "LEÓN" ]
```

Puede usar `filter()` para probar cada elemento de una colección y crear una nueva colección que contenga solo elementos que coincidan:

JS

```
function lCat(cat) {  
    return cat.startsWith("L");  
}
```

```

}

const cats = ["Leopardo", "Serval", "Jaguar", "Tigre", "Caracal", "León"];

const filtered = cats.filter(lCat);

console.log(filtrado);
// [ "Leopardo", "León" ]

```

Esto se parece mucho a `map()`, excepto que la función que pasamos devuelve un [booleano](#): si devuelve `true`, entonces el elemento se incluye en el nuevo arreglo. Nuestra función prueba que el elemento comienza con la letra "L", por lo que el resultado es una matriz que contiene solo gatos cuyos nombres comienzan con "L":

JS

```
[ "Leopardo", "León" ]
```

Tenga en cuenta que `map()` y `filter()` se usan a menudo con [expresiones de funciones](#), que aprenderemos en el módulo [Functions](#)^(inglés). Usando expresiones de función podríamos reescribir el ejemplo anterior para que sea mucho más compacto:

JS

```

const cats = ["Leopardo", "Serval", "Jaguar", "Tigre", "Caracal", "León"];

const filter = cats.filter((cat) => cat.startsWith("L"));
console.log(filtrado);
// [ "Leopardo", "León" ]

```

El bucle estándar `for`

En el ejemplo anterior de "círculos de dibujo", no tiene una colección de elementos para recorrer: realmente solo desea ejecutar el mismo código 100 veces. En un caso como ese, debes usar el bucle [`for`](#). Tiene la siguiente sintaxis:

JS

```

for (inicializador; condición; expresión-final) {
  // código a ejecutar
}

```

Aquí tenemos:

1. La palabra clave `for`, seguida de algunos paréntesis.
2. Dentro de los paréntesis tenemos tres ítems, separados por punto y coma:
 - i. Un **inicializador**: generalmente es una variable establecida en un número, que se incrementa para contar el número de veces que se ha ejecutado el bucle. También se denomina a veces **variable de contador**.
 - ii. Una **condición**: define cuándo el bucle debe dejar de funcionar. Esta es generalmente una expresión que presenta un operador de comparación, una prueba para ver si se ha cumplido la condición de salida.
 - iii. Una **expresión-final**: siempre se evalúa (o ejecuta) cada vez que el bucle ha pasado por una iteración completa. Por lo general, sirve para incrementar (o en algunos casos disminuir) la variable contadora, para acercarla al punto en que la condición ya no es `true`.
3. Algunas llaves que contienen un bloque de código: este código se ejecutará cada vez que el bucle se repita.

Cálculo de cuadrados

Veamos un ejemplo real para que podamos visualizar lo que estos hacen con mayor claridad.

HTML

Play

```
<button id="calculate">Calcular</button>
<button id="clear">Borrar</button>
<pre id="results"></pre>
```

JS

Play

```
const results = document.querySelector("#results");

function calculate() {
  for (let i = 1; i < 10; i++) {
    const newResult = `${i} x ${i} = ${i * i}`;
    results.textContent += `${newResult}\n`;
  }
  results.textContent += "\n¡Finalizado!";
}
```

```
const calculateBtn = document.querySelector("#calculate");
const clearBtn = document.querySelector("#clear");

calculateBtn.addEventListener("click", calculate);
clearBtn.addEventListener("click", () => (results.textContent = ""));
```

Esto nos da el siguiente resultado:

The screenshot shows a simple user interface for a script. At the top right is a 'Play' button. Below it are two rectangular buttons with rounded corners, labeled 'Calcular' and 'Borrar' respectively. The rest of the page is blank white space.

Este código calcula los cuadrados de los números del 1 al 9 y escribe el resultado. El núcleo del código es el bucle `for` que realiza el cálculo.

Desglosemos la línea `for (let i = 1; i < 10; i++)` en sus tres partes:

1. `let i = 1`: la variable del contador, `i`, comienza en `1`. Tenga en cuenta que tenemos que usar `let` para el contador, porque lo estamos reasignando cada vez que damos la vuelta al bucle.
2. `i < 10`: sigue dando la vuelta al bucle mientras `i` sea menor que `10`.
3. `i++`: añade uno a `i` cada vez que recorras el bucle.

Dentro del bucle, calculamos el cuadrado del valor actual de `i`, es decir: `i * i`. Creamos una cadena que expresa el cálculo que realizamos y el resultado, y añadimos esta cadena al texto de salida. También añadimos `\n`, por lo que la siguiente cadena que añadamos comenzará en una nueva línea. De manera que:

1. Durante la primera ejecución, `i = 1`, por lo que añadiremos $1 \times 1 = 1$.
2. Durante la segunda ejecución, `i = 2`, por lo que añadiremos $2 \times 2 = 4$.
3. Y así sucesivamente...
4. Cuando `i` sea igual a `10`, dejaremos de ejecutar el bucle y pasaremos directamente al siguiente código debajo del bucle, imprimiendo el mensaje `¡Finalizado!` en una nueva línea.

Recorriendo colecciones con un bucle for

Puede usar un bucle `for` para iterar a través de una colección, en lugar de un bucle `for...of`.

Echemos un vistazo de nuevo a nuestro ejemplo anterior "for...of":

JS

```
const cats = ["Leopardo", "Serval", "Jaguar", "Tigre", "Caracal", "León"];  
  
for (const cat of cats) {  
  console.log(cat);  
}
```

Podríamos reescribir ese código así:

JS

```
const cats = ["Leopardo", "Serval", "Jaguar", "Tigre", "Caracal", "León"];  
  
for (let i = 0; i < cats.length; i++) {  
  console.log(cats[i]);  
}
```

En este bucle, comenzamos `i` en `0` y nos detenemos cuando `i` alcanza la longitud del arreglo. Luego, dentro del bucle, estamos usando `i` para acceder a cada elemento del arreglo a su vez.

Esto funciona muy bien, y en las primeras versiones de JavaScript, `for...of` no existía, por lo que esta era la forma estándar de iterar a través de un arreglo. Sin embargo, ofrece más posibilidades de introducir errores en tu código. Por ejemplo:

- puede comenzar `i` en `1`, olvidando que el primer índice del arreglo es cero, no `1`.
- puede detenerse en `i <= cats.length`, olvidando que el último índice de matriz está en `length - 1`.

Por razones como esta, generalmente es mejor usar `for...of` si puedes.

A veces todavía necesitas usar un bucle `for` para iterar a través de un arreglo. Por ejemplo, en el siguiente código queremos registrar un mensaje que enumere a nuestros gatos:

JS

```
const cats = ["Pete", "Biggles", "Jasmine"];

let myFavoriteCats = "Mis gatos se llaman ";

for (const cat of cats) {
  myFavoriteCats += `${cat}, `;
}

console.log(myFavoriteCats); // "Mis gatos se llaman Pete, Biggles, Jasmine,"
```

La oración de salida final no está muy bien formada:

Mis gatos se llaman Pete, Biggles, Jasmine,

Preferiríamos que manejara al último gato de manera diferente, así:

Mis gatos se llaman Pete, Biggles y Jasmine.

Pero para hacer esto necesitamos saber cuándo estamos en la iteración final del bucle, y para hacerlo podemos usar un bucle `for` y examinar el valor de `i`:

JS

```
const cats = ["Pete", "Biggles", "Jasmine"];

let myFavoriteCats = "Mis gatos se llaman ";
```

```

for (let i = 0; i < cats.length; i++) {
  if (i === cats.length - 1) {
    // Estamos al final del arreglo
    myFavoriteCats += `y ${cats[i]}.`;
  } else {
    myFavoriteCats += `${cats[i]}, `;
  }
}

console.log(myFavoriteCats); // "Mis gatos se llaman Pete, Biggles y Jasmine."

```

Saliendo de bucles con break

Si desea salir de un bucle antes de que se hayan completado todas las iteraciones, puede usar la instrucción `break`. Ya vimos esto en el artículo anterior cuando analizamos las [sentencias switch](#): cuando se cumple un caso en una sentencia switch que coincide con la expresión de entrada, la sentencia `break` sale inmediatamente de la sentencia switch y pasa al código después de ella.

Es lo mismo con los bucles: una instrucción `break` saldrá inmediatamente del bucle y hará que el navegador pase a cualquier código que lo siga.

Digamos que queríamos buscar a través de una serie de contactos y números de teléfono y devolver solo el número que queríamos encontrar. Primero, un HTML simple: un [`<input>`](#) de texto que nos permite ingresar un nombre para buscar, un elemento [`<button>`](#) para enviar una búsqueda y un elemento [`<p>`](#) para mostrar los resultados en:

HTML

Play

```

<label for="search">Buscar por nombre de contacto: </label>
<input id="search" type="text" />
<button>Buscar</button>

<p></p>

```

Ahora pasemos a JavaScript:

JS

Play

```

const contacts = [
  "Chris:2232322",

```

```

"Sarah:3453456",
"Bill:7654322",
"Mary:9998769",
"Dianne:9384975",
];

const para = document.querySelector("p");
const input = document.querySelector("input");
const btn = document.querySelector("button");

btn.addEventListener("click", () => {
  const searchName = input.value.toLowerCase();
  input.value = "";
  input.focus();
  para.textContent = "";
  for (const contact of contacts) {
    const splitContact = contact.split(":");
    if (splitContact[0].toLowerCase() === searchName) {
      para.textContent = `El número de ${splitContact[0]} es ${splitContact[1]}.`;
      break;
    }
  }
  if (para.textContent === "") {
    para.textContent = "Contacto no encontrado.";
  }
});

```

Play

1. En primer lugar, tenemos algunas definiciones de variables: tenemos una variedad de información de contacto, y cada elemento es una cadena que contiene un nombre y un número de teléfono separados por dos puntos.
2. A continuación, adjuntamos un detector de eventos al botón (`btn`) para que cuando se pulse se ejecute algún código para realizar la búsqueda y devolver los resultados.
3. Almacenamos el valor introducido en la entrada de texto en una variable llamada `searchName`, antes de vaciar la entrada de texto y volver a enfocarla, listos para la

siguiente búsqueda. Tenga en cuenta que también ejecutamos el método `toLowerCase()` en la cadena, de modo que las búsquedas no distingan entre mayúsculas y minúsculas.

4. Ahora pasemos a la parte interesante, el bucle `for...of`:

- i. Dentro del bucle, primero dividimos el contacto actual en el carácter de dos puntos y almacenamos los dos valores resultantes en un arreglo llamado `splitContact`.
- ii. Luego usamos una instrucción condicional para probar si `splitContact[0]` (el nombre del contacto, nuevamente en minúsculas con `toLowerCase()`) es igual al `searchName` ingresado. Si es así, introducimos una cadena en el párrafo para informar cuál es el número del contacto y usamos `break` para finalizar el bucle.

5. Después del bucle, verificamos si configuramos un contacto y, de lo contrario, configuramos el texto del párrafo como "Contacto no encontrado".

Nota: También puedes ver el [código fuente completo en GitHub](#) (también [verlo en vivo](#)).

Omitir iteraciones con `continue`

La instrucción `continue` funciona de manera similar a `break`, pero en lugar de salir del bucle por completo, salta a la siguiente iteración del bucle. Veamos otro ejemplo que toma un número como entrada y devuelve solo los números que son cuadrados de enteros (números enteros).

El HTML es básicamente el mismo que el último ejemplo: una entrada numérica simple y un párrafo para la salida.

HTML

Play

```
<label for="number">Introducir número: </label>
<input id="number" type="number" />
<button>Generar cuadrados enteros</button>

<p>Resultado:</p>
```

El JavaScript también es casi el mismo, aunque el bucle en sí es un poco diferente:

JS

Play

```
const para = document.querySelector("p");
const input = document.querySelector("input");
const btn = document.querySelector("button");

btn.addEventListener("click", () => {
  para.textContent = "Resultado: ";
  const num = input.value;
  input.value = "";
  input.focus();
  for (let i = 1; i <= num; i++) {
    let sqRoot = Math.sqrt(i);
    if (Math.floor(sqRoot) !== sqRoot) {
      continue;
    }
    para.textContent += `${i} `;
  }
});
```

Este es el resultado:

Play

1. En este caso, la entrada debe ser un número (`num`). Al bucle `for` se le da un contador que comienza en 1 (ya que no estamos interesados en 0 en este caso), una condición de salida que dice que el bucle se detendrá cuando el contador sea más grande que la entrada `num`, y un iterador que suma 1 al contador cada vez.
2. Dentro del bucle, encontramos la raíz cuadrada de cada número usando [`Math.sqrt\(i\)`](#), luego verificamos si la raíz cuadrada es un entero probando si es igual a sí misma cuando se ha redondeado al entero más cercano (esto es lo que [`Math.floor\(\)`](#) hace al número que se pasa).
3. Si la raíz cuadrada y la raíz cuadrada redondeada hacia abajo no son iguales entre sí (`!==`), significa que la raíz cuadrada no es un número entero, por lo que no nos

interesa. En tal caso, usamos la instrucción `continue` para saltar a la siguiente iteración de bucle sin registrar el número en ninguna parte.

4. Si la raíz cuadrada es un número entero, omitimos por completo el bloque `if`, por lo que no se ejecuta la instrucción `continue`; en su lugar, concatenamos el valor `i` actual más un espacio al final del contenido del párrafo.

Nota: También puedes ver el [código fuente completo en GitHub](#) (también [verlo en vivo](#)).

while y do...while

`for` no es el único tipo de bucle disponible en JavaScript. En realidad, hay muchos otros y, aunque no es necesario que entiendas todos estos ahora, vale la pena echar un vistazo a la estructura de un par de otros para que puedas reconocer las mismas características en el trabajo de una manera ligeramente diferente.

Primero, echemos un vistazo al bucle [`while`](#). La sintaxis de este bucle se ve así:

JS

```
inicializador
while(condición) {
  // código a ejecutar

  expresión-final
}
```

Esto funciona de una manera muy similar al bucle `for`, excepto que la variable inicializadora se establece antes del bucle, y la expresión final se incluye dentro del bucle después del código a ejecutar, en lugar de que estos dos elementos se incluyan dentro de los paréntesis. La condición se incluye dentro de los paréntesis, que están precedidos por la palabra clave `while` en lugar de `for`.

Los mismos tres elementos todavía están presentes, y todavía están definidos en el mismo orden en que están en el bucle `for`. Esto se debe a que debe tener un inicializador definido antes de poder verificar si la condición es verdadera o no. La expresión final se ejecuta

después de que se haya ejecutado el código dentro del bucle (se ha completado una iteración), lo que solo ocurrirá si la condición sigue siendo cierta.

Echemos un vistazo de nuevo a nuestro ejemplo de lista de gatos, pero reescrito para usar un bucle while:

JS

```
const cats = ["Pete", "Biggles", "Jasmine"];  
  
let myFavoriteCats = "Mis gatos se llaman ";  
  
let i = 0;  
  
while (i < cats.length) {  
  if (i === cats.length - 1) {  
    myFavoriteCats += `y ${cats[i]}.`;  
  } else {  
    myFavoriteCats += `${cats[i]}, `;  
  }  
  
  i++;  
}  
  
console.log(myFavoriteCats); // "Mis gatos se llaman Pete, Biggles y Jasmine."
```

Nota: Esto sigue funcionando exactamente como se esperaba: échale un vistazo al [código fuente completo](#) (también puedes ver el [código fuente completo](#)).

El bucle [do...while](#) es muy similar, pero proporciona una variación en la estructura while:

JS

```
inicializador  
do {  
  // código a ejecutar  
  
  expresión-final  
} while (condición)
```

En este caso, el inicializador vuelve a aparecer primero, antes de que comience el bucle. La palabra clave precede directamente a las llaves que contienen el código a ejecutar y la expresión final.

La principal diferencia entre un bucle `do...while` y un bucle `while` es que *el código dentro de un bucle do...while siempre se ejecuta al menos una vez*. Esto se debe a que la condición viene después del código dentro del bucle. Así que siempre ejecutamos ese código, luego verificamos si necesitamos ejecutarlo de nuevo. En los bucles `while` y `for`, la comprobación es lo primero, por lo que es posible que el código nunca se ejecute.

Volvamos a escribir nuestro ejemplo de listado de gatos para usar un bucle de "do...while":

JS

```
const cats = ["Pete", "Biggles", "Jasmine"];  
  
let myFavoriteCats = "Mis gatos se llaman ";  
  
let i = 0;  
  
do {  
  if (i === cats.length - 1) {  
    myFavoriteCats += `y ${cats[i]}.`;  
  } else {  
    myFavoriteCats += `${cats[i]}, `;  
  }  
  i++;  
} while (i < cats.length);  
  
console.log(myFavoriteCats); // "Mis gatos se llaman Pete, Biggles y Jasmine."
```

Nota: De nuevo, esto funciona igual que lo esperado: échale un vistazo al [código fuente completo](#) (también puedes ver el [código fuente completo](#)).

Advertencia: Con `while` y `do...while`, como con todos los bucles, debe asegurarse de que el inicializador se incremente o, según el caso, se disminuya, para que la

condición finalmente se vuelva falsa. Si no, el bucle continuará para siempre y el navegador lo obligará a detenerse o se bloqueará. Esto se llama un **bucle infinito**.

Aprendizaje activo: iniciar cuenta regresiva

En este ejercicio, queremos que imprima una cuenta regresiva de lanzamiento simple para la caja de salida, desde 10 hasta Blastoff. En concreto, queremos:

- Bucle de 10 a 0. Te hemos proporcionado un inicializador: `let i = 10;` .
- Para cada iteración, cree un nuevo párrafo y añádalo a la salida `<div>`, que hemos seleccionado usando `const output = document.querySelector('.output');` . En los comentarios, le proporcionamos tres líneas de código que deben usarse en algún lugar dentro del bucle:
 - `const para = document.createElement('p');` : crea un nuevo párrafo.
 - `output.appendChild(para);` : añade el párrafo a la salida `<div>` .
 - `para.textContent =` : hace que el texto dentro del párrafo sea igual a lo que pongas en el lado derecho, después del signo igual.
- Los diferentes números de iteración requieren que se coloque un texto diferente en el párrafo para esa iteración (necesitarás una instrucción condicional y varias líneas de `para.textContent =`):
 - Si el número es 10, imprima "Cuenta regresiva 10" en el párrafo.
 - Si el número es 0, imprima "Blast off!" en el párrafo.
 - Para cualquier otro número, imprime solo el número en el párrafo.
- ¡Recuerda incluir un iterador! Sin embargo, en este ejemplo estamos contando hacia atrás después de cada iteración, no hacia arriba, por lo que **no** quieres `i++`: ¿cómo iteras hacia abajo?

Nota: Si comienza a escribir el bucle (por ejemplo, `(while(i>=0))`), es posible que el navegador se atasque porque aún no ha ingresado la condición final. Así que ten cuidado con esto. Puedes empezar a escribir tu código en un comentario para hacer frente a este problema y eliminar el comentario después de terminar.

Si comete un error, siempre puede restablecer el ejemplo con el botón "Restablecer". Si te quedas realmente atascado, pulsa "Mostrar solución" para ver una solución.

HTML

Play

```
<h2>Salida en vivo</h2>
<div class="output" style="height: 410px;overflow: auto;"></div>

<h2>Código editable</h2>
<p class="a11y-label">
  Pulse Esc para alejar el foco del área de código (Tab inserta un carácter de
  tabulación).
</p>
<textarea id="code" class="playable-code" style="height: 300px; width: 95%">
let output = document.querySelector('.output');
output.innerHTML = '';
// let i = 10;
// const para = document.createElement('p');
// para.textContent = ;
// output.appendChild(para);
</textarea>

<div class="playable-buttons">
  <input id="reset" type="button" value="Restablecer" />
  <input id="solution" type="button" value="Mostrar solución" />
</div>
```

CSS

Play

```
html {
  font-family: sans-serif;
}

h2 {
  font-size: 16px;
}

.a11y-label {
  margin: 0;
  text-align: right;
  font-size: 0.7rem;
  width: 98%;
```

```
}
```

```
body {  
  margin: 10px;  
  background: #f5f9fa;  
}
```

Play

Aprendizaje activo: llenar una lista de invitados

En este ejercicio, queremos que tomes una lista de nombres almacenados en un arreglo y los pongas en una lista de invitados. Pero no es tan fácil: ¡no queremos dejar entrar a Phil y Lola porque son codiciosos y groseros, y siempre comen toda la comida! Tenemos dos listas, una para que los huéspedes la admitan y otra para que los huéspedes la rechacen.

En concreto, queremos:

- Escribe un bucle que itere a través del arreglo `people`.
- Durante cada iteración de bucle, compruebe si el elemento del arreglo actual es igual a "Phil" o "Lola" utilizando una instrucción condicional:
 - Si es así, concatene el elemento del arreglo al final de `textContent` del párrafo `refused`, seguido de una coma y un espacio.
 - Si no es así, concatene el elemento del arreglo hasta el final del `textContent` del párrafo `admitted`, seguido de una coma y un espacio.

Ya te hemos proporcionado:

- `refused.textContent +=` : los inicios de una línea que concatenará algo al final de `refused.textContent`.
- `admitted.textContent +=` ; los inicios de una línea que concatenará algo al final de `admitted.textContent`.

Pregunta de bonificación adicional: después de completar las tareas anteriores con éxito, te quedarán dos listas de nombres, separadas por comas, pero estarán desordenadas: habrá una coma al final de cada una. ¿Puedes averiguar cómo escribir líneas que corten la última coma en cada caso y añadir un punto al final? Consulta el artículo [Métodos de cadenas útiles](#) para obtener ayuda.

Si comete un error, siempre puede restablecer el ejemplo con el botón "Restablecer". Si te quedas realmente atascado, pulsa "Mostrar solución" para ver una solución.

HTML

Play

```
<h2>Salida en vivo</h2>
<div class="output" style="height: 100px; overflow: auto;">
  <p class="admitted">Admitir:</p>
```

```
<p class="refused">Rechazar:</p>
</div>

<h2>Código editable</h2>
<p class="a11y-label">
  Pulse Esc para alejar el foco del área de código (Tab inserta un carácter de tabulación).
</p>
<textarea id="code" class="playable-code" style="height: 400px; width: 95%">
const people = ['Chris', 'Anne', 'Colin', 'Terri', 'Phil', 'Lola', 'Sam', 'Kay', 'Bruce'];

const admitted = document.querySelector('.admitted');
const refused = document.querySelector('.refused');
admitted.textContent = 'Admitir: ';
refused.textContent = 'Rechazar: ';

// El bucle comienza aquí

// refused.textContent += ;
// admitted.textContent += ;

</textarea>

<div class="playable-buttons">
  <input id="reset" type="button" value="Restablecer" />
  <input id="solution" type="button" value="Mostrar solución" />
</div>
```

Play

¿Qué tipo de bucle debes usar?

Si está iterando a través de un arreglo o algún otro objeto que lo admite, y no necesita acceder a la posición de índice de cada elemento, entonces `for...of` es la mejor opción. Es más fácil de leer y hay menos para equivocarse.

Para otros usos, los bucles `for`, `while` y `do...while` son en gran medida intercambiables. Todos se pueden usar para resolver los mismos problemas, y cuál uses dependerá en gran medida de tus preferencias personales: cuál te resulta más fácil de recordar o más intuitivo. Recomendaríamos `for`, al menos para empezar, ya que es probablemente el más

fácil para recordar todo: el inicializador, la condición y la expresión final tienen que ir perfectamente entre paréntesis, por lo que es fácil ver dónde están y comprobar que no te los estás perdiendo.

Echémosles un vistazo a todos de nuevo.

Primero `for...of`:

JS

```
for (const elemento of arreglo) {  
    // código a ejecutar  
}
```

`for`:

JS

```
for (inicializador; condición; expresión-final) {  
    // código a ejecutar  
}
```

`while`:

JS

```
inicializador  
while(condición) {  
    // código a ejecutar  
  
    expresión-final  
}
```

y finalmente `do...while`:

JS

```
inicializador  
do {  
    // código a ejecutar
```

```
expresión-final  
} while (condición)
```

Nota: También hay otros tipos/características de bucle, que son útiles en situaciones avanzadas/especializadas y más allá del alcance de este artículo. Si quieras ir más allá con tu aprendizaje en bucle, lee nuestra [Guía de bucles e iteraciones avanzada](#).

Pon a prueba tus habilidades

Has llegado al final de este artículo, pero ¿puedes recordar la información más importante? Puedes encontrar algunas pruebas adicionales para verificar que has conservado esta información antes de continuar. Consulta [Pon a prueba tus habilidades: bucles](#)^(inglés).

Conclusión

Este artículo te ha revelado los conceptos básicos que hay detrás y las diferentes opciones disponibles al hacer bucles de código en JavaScript. ¡Ahora deberías tener claro por qué los bucles son un buen mecanismo para lidiar con el código repetitivo y tener ganas de usarlos en tus propios ejemplos!

Si hay algo que no entendiste, vuelve a leer el artículo o [ponte en contacto con nosotros](#) para pedir ayuda.

Vease también

- [Bucles e iteración en detalle](#)
- [for...of referencia](#)
- [Referencia de la declaración for](#)
- Referencias de [while](#) y [do...while](#)
- Referencias de [break](#) y [continue](#)

Help improve MDN





Was this page helpful to you?

Yes

No

[Learn how to contribute.](#)

This page was last modified on 27 oct 2023 by [MDN contributors](#).