

Guía de Bucles y arreglos en JavaScript

Guía para estudiantes: Arreglos en JavaScript

Explicación Detallada de Ciclos, Sintaxis y Arreglos

1.1 ¿Qué son los ciclos (loops) en JavaScript?

Los ciclos o bucles son estructuras de control que nos permiten ejecutar un bloque de código repetidamente mientras se cumpla una condición. Los bucles en JavaScript son muy útiles cuando necesitamos realizar operaciones repetitivas o iterar sobre colecciones de datos (como arreglos). Existen varios tipos de ciclos en JavaScript:

1. **for**: Es el ciclo más común, usado cuando sabemos cuántas veces necesitamos repetir el código.

Sintaxis del **for**:

javascript

 Copiar código

```
for (inicialización; condición; incremento) {  
    // Código a ejecutar en cada iteración  
}
```

Ejemplo:

javascript

 Copiar código

```
for (let i = 0; i < 5; i++) {  
    console.log(i); // Imprime Los números del 0 al 4  
}
```

2. **while**: Repite el bloque de código mientras se cumpla la condición. Es útil cuando no sabemos cuántas veces se debe ejecutar el código.

Sintaxis del `while`:

javascript

 Copiar código

```
while (condición) {  
    // Código a ejecutar  
}
```

Ejemplo:

javascript


 Copiar código

```
let i = 0;  
while (i < 5) {  
    console.log(i); // Imprime Los números del 0 al 4  
    i++; // Aumenta el valor de i en 1 en cada iteración  
}
```

3. **do...while**: Similar al `while`, pero la condición se evalúa al final, lo que asegura que el bloque de código se ejecute al menos una vez.

Sintaxis del `do...while`:


javascript

 Copiar código

```
do {  
    // Código a ejecutar  
} while (condición);
```

Ejemplo:

javascript

 Copiar código

```
let i = 0;  
do {  
    console.log(i); // Imprime Los números del 0 al 4  
    i++;  
} while (i < 5);
```

1. Introducción a los Arreglos

En JavaScript, un **arreglo** (o **array**) es un objeto que se utiliza para almacenar múltiples valores en una sola variable. Los arreglos pueden contener datos de diferentes tipos (números, cadenas de texto, objetos, funciones, etc.), e incluso otros arreglos. La ventaja de usar un arreglo es que puedes organizar y manipular grandes cantidades de datos de manera más eficiente.

2. Sintaxis Básica

Los arreglos se definen usando corchetes `[]`. Los elementos dentro del arreglo están separados por comas.

```
javascript  
  
let colores = ['rojo', 'verde', 'azul'];
```

En este ejemplo, hemos creado un arreglo llamado `colores` que contiene tres cadenas de texto.

3. Acceso a los Elementos de un Arreglo

Los elementos de un arreglo en JavaScript se acceden utilizando su **índice**. Los índices comienzan desde 0.

```
javascript  
  
console.log(colores[0]); // Imprime 'rojo'  
console.log(colores[1]); // Imprime 'verde'  
console.log(colores[2]); // Imprime 'azul'
```

4. Modificar Elementos de un Arreglo

Puedes modificar un elemento de un arreglo accediendo al índice correspondiente y asignando un nuevo valor.

```
javascript  
  
colores[0] = 'amarillo'; // Cambia 'rojo' por 'amarillo'  
console.log(colores); // ['amarillo', 'verde', 'azul']
```



5. Métodos Comunes de los Arreglos

JavaScript ofrece varios métodos útiles para trabajar con arreglos. Aquí algunos de los más comunes:

```
javascript

colores.push('morado');
console.log(colores); // ['amarillo', 'verde', 'azul', 'morado']
```

- **pop()**: Elimina el último elemento del arreglo.

```
javascript

colores.pop();
console.log(colores); // ['amarillo', 'verde', 'azul']
```

- **shift()**: Elimina el primer elemento del arreglo.

```
javascript

colores.shift();
console.log(colores); // ['verde', 'azul']
```

- **unshift()**: Añade un nuevo elemento al principio del arreglo.

```
javascript

colores.unshift('naranja');
console.log(colores); // ['naranja', 'verde', 'azul']
```

```
javascript

console.log(colores.indexOf('azul')); // 2
console.log(colores.indexOf('rojo')); // -1
```

- **length:** Devuelve la longitud del arreglo.

```
javascript  
  
console.log(colores.length); // 3
```

- **splice()**
Cambia el contenido de un arreglo eliminando, reemplazando o agregando elementos en una posición específica.

```
javascript  
  
let frutas = ['manzana', 'banana', 'cereza'];  
frutas.splice(1, 1, 'kiwi'); // Elimina 'banana' y agrega 'kiwi' en su lugar  
console.log(frutas); // ['manzana', 'kiwi', 'cereza']
```

- **find()**
Devuelve el primer elemento que cumple con la condición especificada en la función de prueba. Si no lo encuentra, devuelve `undefined`.

```
javascript  
  
let numeros = [5, 12, 8, 130, 44];  
let resultado = numeros.find(num => num > 10);  
console.log(resultado); // 12
```

- **findIndex()**
Devuelve el índice del primer elemento que cumple con la condición especificada en la función de prueba. Si no lo encuentra, devuelve `-1`.

```
javascript  
  
let numeros = [5, 12, 8, 130, 44];  
let indice = numeros.findIndex(num => num > 10);  
console.log(indice); // 1
```

- **at()**
Permite acceder a un elemento en una posición específica, soportando índices negativos para contar desde el final del arreglo.

```
javascript

let frutas = ['manzana', 'banana', 'cereza'];
console.log(frutas.at(1));    // 'banana'
console.log(frutas.at(-1));   // 'cereza'
```

6. Métodos para transformar el arreglo

- **map()**
Crea un nuevo arreglo con los resultados de la ejecución de una función sobre cada uno de los elementos del arreglo original.

```
javascript

let numeros = [1, 2, 3];
let duplicados = numeros.map(num => num * 2);
console.log(duplicados); // [2, 4, 6]
```

- **filter()**
Crea un nuevo arreglo con todos los elementos que pasen una prueba especificada en una función.

```
javascript

let numeros = [1, 2, 3, 4, 5];
let pares = numeros.filter(num => num % 2 === 0);
console.log(pares); // [2, 4]
```

- **reduce()**
Ejecuta una función sobre un acumulador y cada elemento del arreglo (de izquierda a derecha) para reducirlo a un único valor.

```
javascript

let numeros = [1, 2, 3, 4];
let suma = numeros.reduce((acumulador, num) => acumulador + num, 0);
console.log(suma); // 10
```

- **reduceRight()**
Similar a `reduce()`, pero ejecuta la función desde el final del arreglo hacia el principio.

```
javascript

let numeros = [1, 2, 3, 4];
let suma = numeros.reduceRight((acumulador, num) => acumulador + num, 0);
console.log(suma); // 10
```

- **sort()**
Ordena los elementos del arreglo en su lugar, y devuelve el arreglo ordenado.

```
javascript

let numeros = [3, 1, 4, 2];
numeros.sort();
console.log(numeros); // [1, 2, 3, 4]
```

- **reverse()**
Invierte el orden de los elementos de un arreglo en su lugar.

```
javascript

let frutas = ['manzana', 'banana', 'cereza'];
frutas.reverse();
console.log(frutas); // ['cereza', 'banana', 'manzana']
```

- **forEach()**
Ejecuta una función en cada uno de los elementos del arreglo, pero no devuelve un nuevo arreglo. Es útil cuando quieres realizar efectos secundarios (como mostrar algo por consola).

```
javascript

let frutas = ['manzana', 'banana', 'cereza'];
frutas.forEach((fruta, index) => {
  console.log(index, fruta);
});
```


7. Arreglos Multidimensionales

Un arreglo multidimensional es un arreglo que contiene otros arreglos como elementos. Puedes acceder a los elementos internos utilizando índices adicionales.

```
javascript

let matriz = [[1, 2], [3, 4], [5, 6]];
console.log(matriz[0][1]); // 2
console.log(matriz[2][0]); // 5
```

8. Ejemplo Explicativo

Vamos a crear un ejemplo donde se combina todo lo aprendido sobre los arreglos. Imaginemos que tenemos un arreglo de estudiantes, donde cada estudiante tiene un nombre y una calificación.

```
javascript

let estudiantes = [
  { nombre: 'Ana', calificacion: 8 },
  { nombre: 'Luis', calificacion: 6 },
  { nombre: 'Carlos', calificacion: 7 }
];
```

```
// Imprimir los nombres de los estudiantes
estudiantes.forEach(function(estudiante) {
  console.log(estudiante.nombre);
});
```

```
// Añadir un nuevo estudiante
estudiantes.push({ nombre: 'Sofía', calificacion: 9 });
console.log(estudiantes);
```

```
// Modificar la calificación de un estudiante
estudiantes[1].calificacion = 7;
console.log(estudiantes);
```

```
// Eliminar el último estudiante  
estudiantes.pop();  
console.log(estudiantes);
```

En este ejemplo:

1. Hemos creado un arreglo de objetos `estudiantes`, donde cada objeto tiene un `nombre` y una `calificacion`.
2. Usamos `forEach()` para recorrer el arreglo e imprimir los nombres.
3. Añadimos un nuevo estudiante usando `push()`.
4. Modificamos la calificación de un estudiante.
5. Finalmente, eliminamos el último estudiante con `pop()`.

Actividad: Gestión de Productos en una Tienda de Electrónica

Caso para resolver: Imagina que eres el encargado de gestionar el inventario de una tienda de electrónica. El dueño de la tienda te ha solicitado un programa que le ayude a mantener un control sobre los productos que tiene disponibles y que le avise cuando un producto se quede sin stock o cuando esté próximo a agotarse.

1. Crear un Inventario

Crea un arreglo llamado `inventario` que contenga los siguientes productos (usa objetos con las propiedades: `nombre`, `codigo`, `cantidad`, `precio`):

- Producto 1: **Televisor 4K**, código: TV001, cantidad: 10, precio: 300
- Producto 2: **Auriculares Bluetooth**, código: AU002, cantidad: 2, precio: 50
- Producto 3: **Smartphone**, código: SP003, cantidad: 0, precio: 500
- Producto 4: **Laptop**, código: LP004, cantidad: 15, precio: 800

2. Verificar Stock

Escribe una función que recorra el arreglo `inventario` y verifique el stock de cada producto. Si la cantidad es 0, debe mostrar "Producto agotado". Si la cantidad es menor o

igual a 5, debe mostrar "Próximo a agotarse". Si la cantidad es mayor a 5, debe mostrar "En stock".

3. Actualizar Stock

Agrega una opción para actualizar la cantidad de un producto. Crea una función que reciba el código del producto y la nueva cantidad, y luego actualice el inventario. Si el producto no existe, muestra un mensaje de error.

4. Agregar Nuevo Producto

Usa el método `push()` para agregar un nuevo producto al inventario. El nuevo producto será:

- Producto: **Cámara Digital**, código: CA005, cantidad: 8, precio: 250

5. Eliminar Producto

Crea una función para eliminar un producto del inventario. El producto debe eliminarse basándose en su código. Si el producto no existe, muestra un mensaje de error.

6. Mostrar Inventario Completo

Escribe una función que recorra el arreglo `inventario` y muestre el nombre de cada producto, su código y su cantidad disponible. Además, si el producto está "en stock", debe mostrar el valor total de ese producto (precio por unidad * cantidad disponible).

7. Buscar Producto por Nombre

Escribe una función que permita buscar un producto por su nombre. Si el producto existe, muestra su código y cantidad disponible. Si no se encuentra, muestra un mensaje de error.

8. Precio Total del Inventario

Usa el método `reduce()` para calcular el valor total del inventario, sumando el precio de cada producto multiplicado por su cantidad disponible.

9. Ordenar Inventario por Precio

Usa el método `sort()` para ordenar el inventario de productos de menor a mayor precio.

10. Verificar si hay Productos en Stock

Escribe una función que use el método `some()` para verificar si hay algún producto en el inventario con más de 5 unidades disponibles. Devuelve `true` si hay productos en stock, de lo contrario, devuelve `false`.

11. Duplicar Precio de Productos en Stock

Usa el método `map()` para crear un nuevo arreglo con los productos que están "en stock" y aumenta su precio en un 20%. Muestra el nuevo arreglo con los precios actualizados.

12. Reemplazar Producto

Imagina que un producto debe ser reemplazado debido a que está dañado o discontinuado. Crea una función que, dado un código de producto, reemplace ese producto por uno nuevo con los siguientes datos:

- Nuevo Producto: **Reproductor Blu-ray**, código: RB006, cantidad: 10, precio: 150

Instrucciones para Resolver la Actividad:

1. Crea un archivo JavaScript donde irás resolviendo cada uno de los enunciados paso a paso.
2. Asegúrate de utilizar los métodos de arreglos como `push()`, `pop()`, `shift()`, `splice()`, `map()`, `filter()`, `reduce()`, `forEach()`, `sort()`, entre otros, para resolver cada tarea.
3. Organiza tu código de forma clara, utilizando funciones para cada uno de los enunciados.
4. Comenta tu código explicando lo que hace cada sección, para facilitar la comprensión.