

Recuerden poner a
grabar la clase



Clase 3

Arrays

Diplomatura UNTREF




Temario

- Estructura basica de un array de elementos
- La propiedad length
- Ciclo de iteraciones
 - Ciclo FOR
 - Ciclo While
 - Ciclo Do while
- Ciclos y condicionales, break,continue

Estructura basica de un array

Estructura basica de un array


un array es una estructura de datos que se utiliza para almacenar y organizar una colección de elementos. Un array puede contener cualquier tipo de dato, como números, cadenas de texto, booleanos, objetos u otros arrays. Los arrays en JavaScript son objetos especiales con propiedades y métodos que permiten acceder y manipular los elementos que contienen.

A dark-themed code editor window with three small circles in the top-left corner. It contains a single line of JavaScript code: `let miArray = [];` where `let` is red and the rest is white.

```
let miArray = [];
```

Estructura basica de un array

un array es una estructura de datos que se utiliza para almacenar y organizar una colección de elementos. Un array puede contener cualquier tipo de dato, como números, cadenas de texto, booleanos, objetos u otros arrays. Los arrays en JavaScript son objetos especiales con propiedades y métodos que permiten acceder y manipular los elementos que contienen.

A dark-themed code editor window with three small circles in the top-left corner. It contains a single line of JavaScript code: `let miArray = [];` where `let` is red and the rest is white.

```
let miArray = [];
```

Acceso a elementos de un array

Los elementos en un array están ordenados y se acceden mediante su índice. Los índices comienzan desde 0 para el primer elemento y van aumentando de uno en uno.

```
let miArray = [10, 20, 30];  
console.log(miArray[0]); // Imprime 10  
console.log(miArray[2]); // Imprime 30
```

Propiedad length en arrays

Los arrays en JavaScript tienen una propiedad llamada length que indica la cantidad de elementos que contiene. La propiedad length siempre devuelve un valor mayor en 1 que el índice del último elemento.

```
Let miArray = [10, 20, 30];  
console.log(miArray.length); // Imprime 3
```


Modificación de elementos en arrays

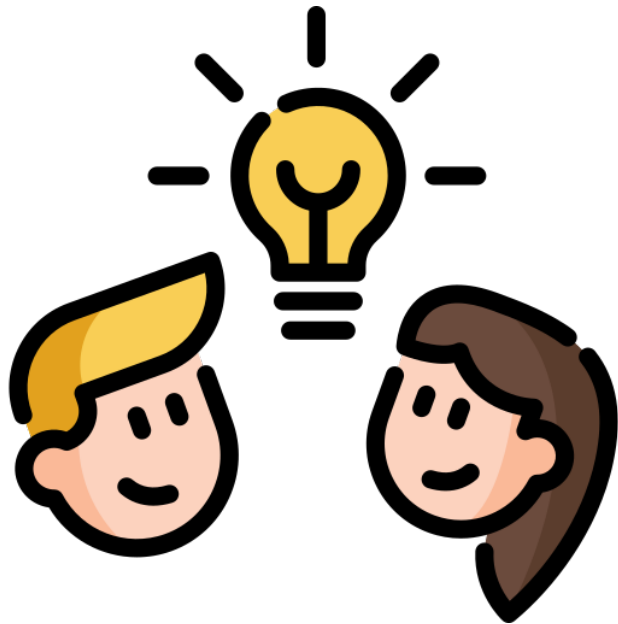
Los elementos de un array pueden ser modificados asignando un nuevo valor a un índice específico

```
let miArray = [10, 20, 30];  
miArray[1] = 50; // Modifica el segundo elemento a 50  
console.log(miArray); // Imprime [10, 50, 30]
```

Metodos de array

Metodos de arrays

Los arrays en JavaScript tienen una serie de métodos incorporados que facilitan su manipulación. Algunos de los métodos más comunes son push, pop, shift, unshift, slice, splice, concat, join, indexOf, forEach y muchos más. Estos métodos permiten agregar, eliminar, modificar y realizar operaciones en los elementos del array.



Método push

Agrega uno o más elementos al final del array y devuelve la nueva longitud del array.

```
let miArray = [1, 2, 3];  
miArray.push(4, 5);  
console.log(miArray); // Imprime [1, 2, 3, 4, 5]
```

Método pop

Elimina el último elemento del array y lo devuelve.

```
const numeros = [1, 2, 3, 4, 5];  
const elementoEliminado = numeros.pop();  
let elementoEliminado = miArray.pop();  
console.log(elementoEliminado); // Resultado: 5  
console.log(numeros); // Resultado: [1, 2, 3, 4]
```

Método shift

Elimina el último elemento del array y lo devuelve.

```
let miArray = [1, 2, 3];  
let elementoEliminado = miArray.shift();  
console.log(miArray); // Imprime [2, 3]  
console.log(elementoEliminado); // Imprime 1
```

Método Unshift

Agrega uno o más elementos al inicio del array y devuelve la nueva longitud del array.

```
let miArray = [2, 3];  
miArray.unshift(0, 1);  
console.log(miArray); // Imprime [0, 1, 2, 3]
```

Método Slice

Devuelve una copia superficial de una porción del array, especificada mediante el índice de inicio y fin.

```
let miArray = [1, 2, 3, 4, 5];  
let subArray = miArray.slice(1, 4);  
console.log(subArray); // Imprime [2, 3, 4]
```


Método Splice

Cambia el contenido de un array eliminando, reemplazando o agregando elementos en el lugar. También puede devolver los elementos eliminados en un nuevo array.

```
let miArray = [1, 2, 3, 4, 5];  
let elementosEliminados = miArray.splice(2, 2, 'a', 'b');  
console.log(miArray); // Imprime [1, 2, 'a', 'b', 5]  
console.log(elementosEliminados); // Imprime [3, 4]
```

Método concat

Este método se utiliza para combinar dos o más arrays, creando un nuevo array resultante. No modifica los arrays originales.

```
let array1 = [1, 2, 3];  
let array2 = [4, 5, 6];  
let newArray = array1.concat(array2);  
console.log(newArray); // Imprime [1, 2, 3, 4, 5, 6]
```

Método join

Convierte todos los elementos de un array en una cadena de texto, separados por un delimitador especificado. Retorna la cadena resultante.

```
let miArray = [1, 2, 3, 4, 5];  
let cadena = miArray.join('-');  
console.log(cadena); // Imprime "1-2-3-4-5"
```

Método indexOf

Devuelve el índice de la primera aparición de un elemento en el array. Si el elemento no se encuentra, retorna -1.

```
let miArray = [10, 20, 30, 40, 50];  
console.log(miArray.indexOf(30)); // Imprime 2  
console.log(miArray.indexOf(60)); // Imprime -1
```

Método findIndex

se utiliza para encontrar el índice del primer elemento en un array que cumpla con una condición especificada. Devuelve el índice del elemento encontrado, o -1 si no se encuentra ningún elemento que cumpla la condición.



```
const frutas = ["manzana", "banana", "naranja", "pera"];  
const indice = frutas.findIndex(fruta => fruta === "naranja");  
  
console.log(indice); // Resultado: 2
```

Método sort

Se utiliza para ordenar los elementos de un array en su lugar, es decir, modifica el array original y lo ordena.

```
const numeros = [4, 2, 1, 3, 5];  
numeros.sort();  
  
console.log(numeros); // Resultado: [1, 2, 3, 4, 5]
```

Método includes

se utiliza para verificar si un array contiene un elemento específico. Devuelve true si el elemento está presente en el array y false en caso contrario.

```
const frutas = ["manzana", "banana", "naranja"];  
console.log(frutas.includes("banana")); // Resultado: true  
console.log(frutas.includes("pera")); // Resultado: false
```

Método reverse

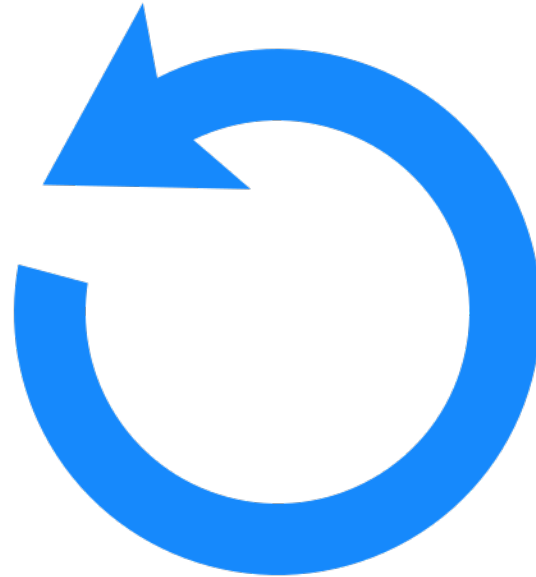
se utiliza para invertir el orden de los elementos en un array. Modifica el array original y devuelve una referencia al array invertido.

```
const numeros = [1, 2, 3, 4, 5];  
numeros.reverse();  
  
console.log(numeros); // Resultado: [5, 4, 3, 2, 1]
```


Ciclos de iteraciones

Ciclos de iteraciones

También conocidos como bucles, son estructuras de control que permiten ejecutar repetidamente un bloque de código mientras se cumpla una condición determinada. Estas estructuras son fundamentales para realizar tareas repetitivas y procesar conjuntos de datos de manera eficiente.



Ciclos for

Ciclo for

El ciclo for es una estructura de control en JavaScript que se utiliza para repetir un bloque de código un número específico de veces. El bucle for consta de tres partes: la inicialización, la condición y la expresión de actualización.

El ciclo for es especialmente útil cuando sabes de antemano cuántas veces quieres repetir un bloque de código. Puedes personalizar la inicialización, la condición y la expresión de actualización según tus necesidades.

```
for (inicialización; condición; expresión de actualización) {  
  // Bloque de código a ejecutar en cada iteración  
}
```

Análisis de cada parte del ciclo for

```
for (inicialización; condición; expresión de actualización) {  
    // Bloque de código a ejecutar en cada iteración  
}
```

- **La inicialización** se utiliza para declarar e inicializar la variable de control del bucle. Normalmente, se utiliza una variable de contador y se le asigna un valor inicial.
- **La condición** es una expresión booleana que se evalúa antes de cada iteración. Si la condición es verdadera, el bloque de código se ejecuta. Si la condición es falsa, el bucle se detiene y la ejecución continúa con la siguiente instrucción después del bucle for.
- **La expresión** de actualización se utiliza para modificar la variable de control del bucle después de cada iteración. Esto generalmente se hace para incrementar o decrementar el valor de la variable.

Ejemplo del ciclo for

En este ejemplo, la inicialización `let i = 1` establece la variable de control `i` en 1. La condición `i <= 5` se evalúa antes de cada iteración para verificar si `i` es menor o igual a 5. La expresión de actualización `i++` se ejecuta después de cada iteración para incrementar el valor de `i` en 1.

```
for (let i = 1; i <= 5; i++) {  
  console.log(i);  
}
```

Ciclo while

Ciclo while

El ciclo **while** es otra estructura de control en JavaScript que permite repetir un bloque de código mientras se cumpla una condición específica. A **diferencia** del ciclo **for**, el ciclo **while** no tiene una parte de inicialización o expresión de actualización. Se basa únicamente en la evaluación de una condición para determinar si debe seguir ejecutándose o no.

```
while (condición) {  
    // Bloque de código a ejecutar en cada iteración  
}
```


Ciclo while, su condición

La condición es una expresión booleana que se evalúa antes de cada iteración. Si la condición es verdadera, el bloque de código se ejecuta. Si la condición es falsa, el ciclo while se detiene y la ejecución continúa con la siguiente instrucción después del bloque while.

```
while (condición) {  
    // Bloque de código a ejecutar en cada iteración  
}
```

Ciclo while, su condición

La condición es una expresión booleana que se evalúa antes de cada iteración. Si la condición es verdadera, el bloque de código se ejecuta. Si la condición es falsa, el ciclo while se detiene y la ejecución continúa con la siguiente instrucción después del bloque while.

```
let i = 1;
//i <=5 seria la condicion
while (i <= 5) {
  console.log(i);
  i++;
}
```

Ciclo while, IMPORTANTE

Es importante asegurarse de que la condición en el ciclo while se actualice adecuadamente dentro del bloque de código para evitar bucles infinitos. Si la condición nunca se vuelve falsa, el ciclo continuará ejecutándose indefinidamente.

El ciclo while es útil cuando no se conoce de antemano el número exacto de iteraciones y se quiere repetir un bloque de código hasta que se cumpla una condición específica.



Ciclo Do while

Ciclo Do while

El ciclo do-while es una estructura de control en JavaScript que permite ejecutar un bloque de código al menos una vez y luego repetirlo mientras se cumpla una condición específica. A diferencia del ciclo while, la condición se evalúa después de cada iteración, lo que garantiza que el bloque de código se ejecute al menos una vez, independientemente de si la condición es verdadera o falsa inicialmente.

```
do {  
    // Bloque de código a ejecutar en cada iteración  
} while (condición);
```

Ciclo Do while, estructura

- El bloque de código se ejecuta primero antes de evaluar la condición.
- Después de cada iteración, se verifica la condición.
- Si la condición es verdadera, el bloque de código se ejecuta nuevamente.
- Si la condición es falsa, el ciclo do-while se detiene y la ejecución continúa con la siguiente instrucción después del bloque do-while.

```
do {  
    // Bloque de código a ejecutar en cada iteración  
} while (condición);
```

Ciclos y condicionales

Ciclos y condicionales

Los ciclos y los condicionales están estrechamente relacionados y se utilizan conjuntamente para controlar el flujo de ejecución de un programa. La relación entre ambos radica en que los condicionales se utilizan para tomar decisiones basadas en una condición, y los ciclos se utilizan para repetir un bloque de código hasta que se cumpla una condición.



Relaciones Ciclos y condicionales

Los **ciclos** y los **condicionales** permiten **controlar el flujo de ejecución de un programa** de manera dinámica. Los condicionales se utilizan para decidir si se ejecuta un bloque de código, y los ciclos se utilizan para repetir un bloque de código hasta que se cumpla una condición.



Relaciones Ciclos y condicionales

Los condicionales **se utilizan para evaluar una expresión o condición booleana y ejecutar diferentes bloques de código según el resultado**. Esto es útil para tomar decisiones en función de diferentes situaciones. Por ejemplo, se puede usar un condicional para comprobar si un número es mayor que 10 y ejecutar un código específico en consecuencia.



Relaciones Ciclos y condicionales

Los ciclos se utilizan para **repetir un bloque de código varias veces hasta que se cumpla una condición específica**. Los ciclos permiten automatizar tareas que deben repetirse, como iterar sobre una lista de elementos o realizar cálculos repetitivos. Por ejemplo, se puede utilizar un ciclo para recorrer un array e imprimir cada elemento.



Relaciones Ciclos y condicionales

Los ciclos y los condicionales se **pueden combinar para lograr un control de flujo más sofisticado**. Por ejemplo, se puede utilizar un ciclo while junto con un condicional if para repetir un bloque de código hasta que se cumpla una condición y luego tomar decisiones dentro del ciclo en función de condiciones adicionales.



Relaciones Ciclos y condicionales

Los condicionales también **se pueden utilizar dentro de los ciclos** para tomar decisiones en cada iteración. Esto permite realizar acciones diferentes en función de condiciones cambiantes en cada ciclo.



Relaciones Ciclos y condicionales

Supongamos que queremos imprimir los números del 1 al 10, pero solo queremos imprimir los números impares.

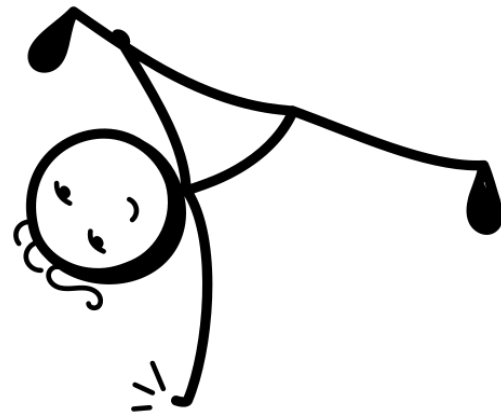
En este ejemplo, utilizamos un ciclo for para iterar del 1 al 10. En cada iteración, evaluamos si el número actual i es impar utilizando el operador de módulo $\%$. Si el resultado de $i \% 2$ no es igual a cero, significa que i es un número impar. En ese caso, imprimimos el número utilizando `console.log()`.

```
for (let i = 1; i <= 10; i++) {  
  if (i % 2 !== 0) {  
    console.log(i);  
  }  
}
```

Break y continue

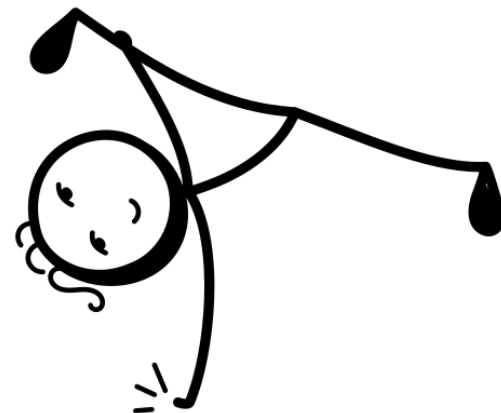
Break

En los condicionales, la declaración break se utiliza para interrumpir la ejecución de un bucle o una estructura de control en el momento en que se cumple una condición específica. Cuando se encuentra la declaración break, el programa sale inmediatamente del bucle o estructura de control más cercana y continúa con la siguiente instrucción después del bloque de código.



Break

La declaración `break` se utiliza comúnmente en conjunción con condicionales, como en un condicional `switch` o un ciclo `for` o `while`, para terminar prematuramente la ejecución cuando se alcanza una condición deseada.



Break en acción

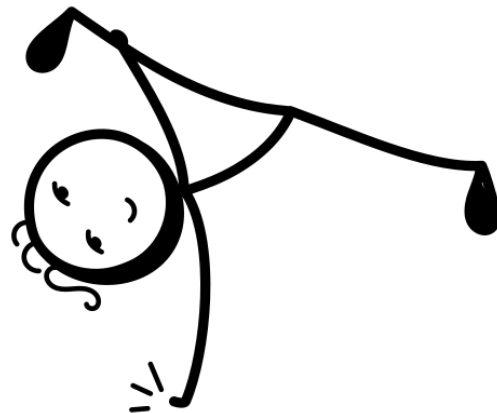
En este ejemplo, el ciclo for itera del 1 al 10. Sin embargo, cuando i es igual a 5, se encuentra la declaración break, lo que provoca que el bucle se detenga inmediatamente. Como resultado, solo se imprimen los números del 1 al 4 en la consola.

```
for (let i = 1; i <= 10; i++) {  
  if (i === 5) {  
    break;  
  }  
  console.log(i);  
}
```

Break

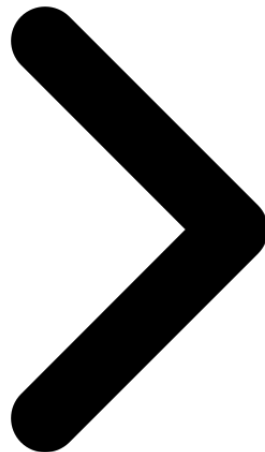
Es importante tener en cuenta que break solo afecta al bucle o estructura de control más cercana. Si hay bucles o estructuras de control anidadas, break solo romperá el bucle en el que se encuentra.

La declaración break es útil para controlar el flujo de ejecución en situaciones donde deseas detener la ejecución de un bucle o una estructura de control antes de que alcance su finalización normal.



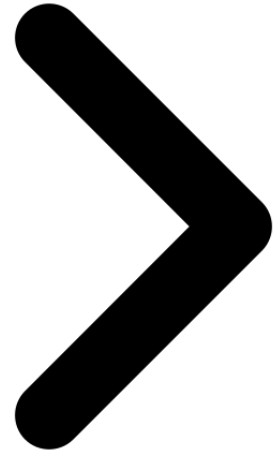
Continue

En los condicionales, la declaración `continue` se utiliza para omitir la ejecución restante de una iteración actual de un bucle y pasar a la siguiente iteración sin ejecutar el resto del código dentro del bloque de esa iteración. Cuando se encuentra la declaración `continue`, se salta cualquier código que siga en la iteración actual y se pasa directamente a la siguiente iteración.



Continue

La declaración `continue` se utiliza comúnmente en ciclos como `for` o `while` para omitir ciertas iteraciones en función de una condición específica, y continuar con la siguiente iteración.



Continue en acción

En este ejemplo, el ciclo for itera del 1 al 5. Sin embargo, cuando i es igual a 3, se encuentra la declaración continue, lo que provoca que se omita la ejecución restante de esa iteración y se pase directamente a la siguiente iteración. Como resultado, se imprimirán los números 1, 2, 4 y 5 en la consola, pero el número 3 se omite.

```
for (let i = 1; i <= 5; i++) {
  if (i === 3) {
    continue;
  }
  console.log(i);
}
```

Continue

Es importante destacar que `continue` solo afecta a la iteración actual del bucle en el que se encuentra. Después de que se ejecuta `continue`, se pasa a la siguiente iteración y se continúa con el ciclo.

La declaración `continue` es útil cuando deseas omitir ciertas iteraciones en función de una condición y continuar con el bucle. Se puede utilizar para filtrar elementos en un conjunto de datos o para evitar ciertas operaciones en ciertas condiciones.

¿Dudas o consultas?





¡Muchas Gracias!