En JavaScript, el flujo se refiere a la forma en que las instrucciones de un programa se ejecutan en un orden específico. El control de flujo es un concepto fundamental en la programación que nos permite tomar decisiones y repetir acciones basadas en condiciones específicas. En JavaScript, existen varias estructuras que facilitan este control:

Condicionales

Los condicionales son estructuras que permiten que un programa tome decisiones ejecutando diferentes bloques de código según si una condición es verdadera o falsa. Una condición es una expresión lógica que se evalúa como `true` (verdadero) o `false` (falso). Según este resultado, el programa decidirá qué bloque de código ejecutar.

La sintaxis básica de un condicional utiliza las palabras clave `if`, `else` y `else if`. Estas estructuras nos permiten manejar escenarios simples o complejos, dependiendo de las necesidades del programa.

Estructura básica de un condicional

La forma más simple de un condicional utiliza la palabra clave `if`, seguida de una condición entre paréntesis `()` y un bloque de código entre llaves `{}` que se ejecuta si la condición es verdadera:

```javascript
if (condición) {
  // Código que se ejecuta si la condición es verdadera
}
```

Extensiones del condicional:

`if` : Ejecuta un bloque de código si la condición especificada es verdadera. Es útil para manejar decisiones simples o iniciales en un programa.

```javascript
let edad = 18;
if (edad >= 18) {
  console.log("Eres mayor de edad.");
}
```

En este caso, el programa verifica si la variable `edad` es mayor o igual a 18. Si es así, muestra un mensaje en la consola.

`else` : Proporciona una alternativa en caso de que la condición del `if` sea falsa. Esto asegura que siempre se ejecutará algún bloque de código.

```javascript
if (edad >= 18) {
  console.log("Eres mayor de edad.");
} else {
  console.log("Eres menor de edad.");
}
```

Aquí, si la condición `edad >= 18` es falsa, el programa ejecutará el bloque de código dentro de `else`.

`else if` : Permite manejar múltiples condiciones secuenciales. Se evalúan en orden, y se ejecuta el primer bloque cuyo `else if` sea verdadero. Si ninguna de las condiciones se cumple, puede incluirse un bloque `else` como último recurso.

```javascript
if (edad < 13) {
  console.log("Eres un niño.");
} else if (edad < 18) {
  console.log("Eres un adolescente.");
} else {
  console.log("Eres un adulto.");
}
```

Este ejemplo evalúa diferentes rangos de edad. Primero comprueba si `edad` es menor de 13, luego si está entre 13 y 17, y finalmente asume que es mayor o igual a 18.

Usos comunes de los condicionales

- **Validaciones**: Verificar entradas de usuario, como si una contraseña cumple ciertos criterios.
- **Toma de decisiones**: Determinar qué ruta lógica seguir en un programa, como calcular descuentos o mostrar mensajes personalizados.
- **Flujo dinámico**: Adaptar el comportamiento del programa según diferentes escenarios, como en juegos, aplicaciones interactivas o sistemas basados en reglas.

Estas estructuras son esenciales para construir programas dinámicos y adaptables. Con una combinación adecuada de `if`, `else if` y `else`, puedes manejar todo tipo de situaciones lógicas en tus aplicaciones.

Operadores Ternarios

El operador ternario es una estructura compacta y elegante para tomar decisiones en JavaScript. Permite evaluar una condición y devolver un valor u otro, dependiendo del resultado de dicha evaluación. Es una alternativa concisa a los bloques `if...else` cuando solo necesitas ejecutar una acción simple basada en una condición.

Estructura básica

La sintaxis del operador ternario es la siguiente:

```
condición ? expresión_si_verdadero : expresión_si_falso;
```

- **`condición`**: Es la expresión lógica que se evalúa como `true` o `false`.
- **`expresión_si_verdadero`**: El valor o acción que se ejecuta si la condición es verdadera.
- **`expresión_si_falso`**: El valor o acción que se ejecuta si la condición es falsa.

Ejemplo básico

```
let edad = 18;
let mensaje = edad >= 18 ? "Eres mayor de edad" : "Eres menor de edad";
console.log(mensaje);
```

En este ejemplo:

- Si `edad >= 18` es `true`, el valor de `mensaje` será `"Eres mayor de edad"`.
- Si `edad >= 18` es `false`, el valor de `mensaje` será `"Eres menor de edad"`.

Usos comunes del operador ternario

1. **Asignar valores basados en una condición**

```
let descuento = esMiembro ? 0.1 : 0.05; console.log("Descuento aplicado:", descuento);
```

2. **Simplificar `if...else` cuando es sencillo** En lugar de escribir:

```
if (usuarioActivo) {
    mensaje = "Bienvenido de nuevo";
} else {
    mensaje = "Por favor, inicia sesión";
}
```

Puedes usar:

```
let mensaje = usuarioActivo ? "Bienvenido de nuevo" : "Por favor, inicia sesión";
```

3. **Uso en plantillas o en valores devueltos**

```
let estado = puntuacion >= 60 ? "Aprobado" : "Reprobado";
```

Anidación de operadores ternarios

Es posible anidar operadores ternarios, aunque esto puede afectar la legibilidad. Solo se recomienda en casos donde las condiciones sean simples.

Ejemplo:

```
let nota = 85;
let calificacion =
    nota >= 90 ? "A"
        : nota >= 80 ? "B"
            : nota >= 70 ? "C"
                : nota >= 60 ? "D"
                    : "F";
console.log("Calificación:", calificacion);
```

Resultado:

```
Calificación: B
```

Ventajas del operador ternario

- Es conciso y compacto, ideal para expresiones cortas.
- Puede mejorar la legibilidad en situaciones simples.
- Reduce líneas de código comparado con los bloques `if...else`.

Cuándo NO usar operadores ternarios

Aunque los ternarios son útiles, hay situaciones donde no son la mejor opción:

1. **Condiciones complejas**: Si la lógica involucra varias condiciones o múltiples pasos, es mejor usar un bloque `if...else` por claridad.
2. **Anidaciones excesivas**: Más de uno o dos operadores ternarios anidados pueden dificultar la comprensión del código.

Ejemplo a evitar:

```
let resultado = a > b ? (a > c ? "A es mayor" : "C es mayor") : (b > c ? "B es mayor" : "C es ma
```

Esto es funcional, pero difícil de leer. En su lugar, usa un bloque `if...else` claro.

Switch

El bloque `switch` es otra estructura de control de flujo que permite evaluar una expresión y ejecutar diferentes bloques de código dependiendo del valor de esa expresión. Es especialmente útil cuando hay múltiples valores posibles que deben manejarse de forma distinta, evitando una larga cadena de `else if`.

Estructura básica de un `switch`

Un bloque `switch` evalúa una expresión una vez y compara su valor con una serie de casos definidos. Cuando encuentra una coincidencia, ejecuta el código asociado a ese caso. La estructura básica es la siguiente:

```
switch (expresión) {
    case valor1:
        // Código que se ejecuta si la expresión === valor1
        break;
    case valor2:
        // Código que se ejecuta si la expresión === valor2
        break;
    // Más casos...
```

```
    default:
        // Código que se ejecuta si no hay coincidencias
}
```

Partes de un bloque `switch`:

1. `switch (expresión)`: La expresión que será evaluada. Puede ser una variable, un cálculo o cualquier expresión válida en JavaScript.
2. `case valor:`: Define un posible valor de la expresión. Si coincide, se ejecuta el código asociado a este caso.
3. `break`: Termina la ejecución del bloque `switch` después de un caso. Sin `break`, el programa continuará ejecutando el código de los casos siguientes (comportamiento conocido como *fall-through*).
4. `default`: Un caso opcional que se ejecuta si ningún `case` coincide con el valor de la expresión.

Ejemplo básico de un `switch`

```
let dia = 3;

switch (dia) {
    case 1:
        console.log("Hoy es lunes.");
        break;
    case 2:
        console.log("Hoy es martes.");
        break;
    case 3:
        console.log("Hoy es miércoles.");
        break;
    case 4:
        console.log("Hoy es jueves.");
        break;
    case 5:
        console.log("Hoy es viernes.");
        break;
    default:
        console.log("Es fin de semana.");
}
```

En este ejemplo, el programa evalúa el valor de `dia`. Si `dia` es igual a `3`, se imprime "Hoy es miércoles". Si `dia` no coincide con ninguno de los valores definidos, se ejecuta el bloque `default`.

Comportamiento de `fall-through`

Si se omite la instrucción `break`, el programa continuará ejecutando los bloques de código siguientes, independientemente de si coinciden o no. Esto puede ser útil en situaciones específicas.

Ejemplo de `fall-through`:

```
let fruta = "manzana";

switch (fruta) {
    case "manzana":
    case "pera":
        console.log("Esta fruta es jugosa.");
        break;
    case "plátano":
        console.log("Esta fruta es amarilla.");
        break;
    default:
        console.log("Fruta no reconocida.");
}
```

En este caso, tanto "manzana" como "pera" ejecutan el mismo bloque de código, gracias a la ausencia de `break` entre ellos.

Usos comunes del `switch`

- **Menús de selección**: Elegir entre diferentes opciones en función de un valor.
- **Configuraciones de estado**: Manejar diferentes estados en un programa (por ejemplo, un semáforo o estados de usuario en un juego).
- **Validación de entrada de datos**: Procesar diferentes valores de entrada de una forma estructurada.
- **Reemplazo de cadenas de `else if`**: Cuando hay muchos valores posibles para una misma variable, un `switch` puede ser más legible y eficiente que múltiples condicionales.

Comparación con `if...else if`

- `switch` **es más adecuado** cuando hay múltiples valores fijos que se desean comparar con una sola expresión.
- `if...else if` **es más flexible**, ya que permite condiciones más complejas que no se limitan a la comparación de igualdad.

Por ejemplo, si necesitas verificar si un número está dentro de un rango, un `if` es mejor:

```
let numero = 15;

if (numero > 10 && numero < 20) {
    console.log("El número está en el rango de 11 a 19.");
}
```

Mientras que, para comparaciones exactas, un `switch` es más limpio y fácil de leer:

```
let numero = 15;

switch (numero) {
    case 10:
        console.log("El número es 10.");
        break;
    case 15:
        console.log("El número es 15.");
        break;
    case 20:
        console.log("El número es 20.");
        break;
    default:
        console.log("El número no está en la lista.");
}
```

Bucles

Los bucles son las estructuras de control que permiten repetir un bloque de código varias veces, ya sea un número fijo de iteraciones o mientras se cumpla una condición específica. Son esenciales para manejar tareas repetitivas de manera eficiente.

En JavaScript, los bucles más comunes son: `for`, `while`, `do...while` y los métodos iterativos (como `for...in` y `for...of`). Vamos a verlos a continuación

Bucle `for`

El bucle `for` se utiliza cuando sabemos de antemano cuántas veces queremos repetir un bloque de código. Es ideal para recorrer arrays, listas y secuencias.

Sintaxis básica:

```
for (inicialización; condición; actualización) {
    // Código que se ejecuta en cada iteración
}
```

- **Inicialización**: Se ejecuta una vez al inicio del bucle. Generalmente, se usa para declarar una variable contador.
- **Condición**: Se evalúa antes de cada iteración. Si es verdadera, se ejecuta el bloque de código; si es falsa, el bucle termina.
- **Actualización**: Se ejecuta al final de cada iteración, generalmente para modificar el contador.

Ejemplo:

```
for (let i = 0; i < 5; i++) {
    console.log("Iteración número:", i);
}
```

Este código imprime los números del 0 al 4. La variable `i` comienza en 0, el bucle continúa mientras `i < 5` y se incrementa en cada iteración.

Bucle `while`

El bucle `while` repite un bloque de código mientras una condición sea verdadera. Es útil cuando no sabemos de antemano cuántas iteraciones se necesitan, pero sí tenemos una condición clara.

Sintaxis básica:

```
while (condición) {
    // Código que se ejecuta mientras la condición sea verdadera
}
```

Ejemplo:

```
let contador = 0;

while (contador < 5) {
    console.log("Contador:", contador);
    contador++;
}
```

Aquí, el bucle continúa mientras `contador` sea menor que 5. En cada iteración, se imprime el valor de `contador` y luego se incrementa.

Bucle `do...while`

El bucle `do...while` es similar al `while`, pero garantiza que el bloque de código se ejecute al menos una vez, incluso si la condición inicial es falsa.

Sintaxis básica:

```
do {
    // Código que se ejecuta al menos una vez
} while (condición);
```

Ejemplo:

```
let numero = 5;

do {
    console.log("El número es:", numero);
    numero--;
} while (numero > 0);
```

En este caso, el bloque se ejecuta mientras `numero > 0`. Incluso si `numero` comienza con un valor que no cumple la condición, el bloque se ejecutará una vez.

Bucle `for...in`

El bucle `for...in` recorre las propiedades de un objeto. Es útil para acceder a las claves de un objeto.

Sintaxis básica:

```
for (let clave in objeto) {
    // Código que se ejecuta para cada clave
}
```

Ejemplo:

```
let persona = { nombre: "Juan", edad: 30, ciudad: "Madrid" };

for (let clave in persona) {
    console.log(clave + ":", persona[clave]);
}
```

Este código recorre el objeto `persona` y muestra cada clave con su valor asociado.

Bucle `for...of`

El bucle `for...of` recorre elementos iterables como arrays, cadenas de texto, mapas y conjuntos. Es útil para trabajar con los valores directamente.

Sintaxis básica:

```
for (let valor of iterable) {
    // Código que se ejecuta para cada valor
}
```

Ejemplo:

```
let frutas = ["manzana", "plátano", "pera"];

for (let fruta of frutas) {
    console.log("Fruta:", fruta);
}
```

Aquí, el bucle recorre el array `frutas` y muestra cada elemento.

Cuándo usar cada tipo de bucle

- `for`: Ideal para recorrer secuencias con un número fijo de iteraciones o cuando necesitas un contador explícito.
- `while`: Útil para iteraciones indefinidas que dependen de una condición dinámica.
- `do...while`: Perfecto si necesitas garantizar al menos una ejecución del bloque antes de verificar la condición.
- `for...in`: Excelente para recorrer las propiedades de un objeto.
- `for...of`: Preferido para iterar sobre valores en arrays u otros objetos iterables.

Usos comunes de los bucles

- Recorrer arrays y objetos.
- Repetir tareas hasta que se cumpla una condición (como buscar un valor específico).
- Automatizar operaciones masivas, como sumar elementos de un array o generar datos dinámicos.

- Procesar entradas del usuario o datos provenientes de APIs.