Escopo em JavaScript

Escopo (Scope) significa **âmbito ou visibilidade de uma variável** dentro de um programa JavaScript. Ele define **onde uma variável pode ser acessada** e **por quanto tempo ela existe**.

Compreender escopo é essencial para: - Evitar bugs difíceis de identificar - Evitar conflitos de nomes - Escrever código previsível e profissional - Entender o funcionamento interno do JavaScript

Em JavaScript, existem **três tipos principais de escopo**: - Escopo Global - Escopo de Função - Escopo de Bloco

O comportamento do escopo também depende: - Da palavra-chave utilizada (`var`, `let`, `const`) - Do modo de execução (`strict mode`)

---

1. Escopo Global

Uma variável possui **escopo global** quando é declarada **fora de qualquer função ou bloco**.

Variáveis globais podem ser acessadas **em qualquer parte do código**, incluindo funções, blocos e outros scripts carregados na mesma página.

```
var x = 1;    // Escopo global
let y = 2;    // Escopo global
const z = 3;  // Escopo global
```

Exemplo prático

```
let carName = "Volvo";

console.log(carName); // Acesso direto

function myFunction() {
  console.log(carName); // Acesso dentro da função
}
```

Características do escopo global

- Acessível em todo o programa
- Compartilhado entre scripts da mesma página
- Pode causar conflitos de nomes
- Dificulta manutenção e testes

Boa prática: evite variáveis globais sempre que possível. Prefira escopos locais.

---

2. Escopo de Função

Cada função em JavaScript cria seu **próprio escopo**.

Variáveis declaradas dentro de uma função **só podem ser acessadas dentro dela**.

```
function myFunction() {
  let carName = "Volvo";
  console.log(carName); // OK
}

console.log(carName); // Erro
```

Palavras-chave dentro da função

```
function example1() {
  var x = 10;   // Escopo de função
}

function example2() {
  let y = 20;   // Escopo de função
}

function example3() {
  const z = 30; // Escopo de função
}
```

Características do escopo de função

- Variáveis são locais
- Criadas quando a função é executada
- Destruídas ao final da execução
- Funções diferentes podem reutilizar os mesmos nomes
- Parâmetros funcionam como variáveis locais

```
function soma(a, b) {
  return a + b;
}
```

---

3. Escopo de Bloco

Introduzido no **ES6**, o escopo de bloco se aplica a variáveis declaradas com: - `let` - `const`

Um **bloco** é qualquer trecho de código delimitado por `{ }`, como: - `if` - `for` - `while` - `switch`

```
{
  let x = 2;
  const y = 3;
}

console.log(x); // Erro
console.log(y); // Erro
```

Em estruturas de controle

```
if (true) {
  let msg = "Olá";
}

console.log(msg); // Erro
```

`var` não possui escopo de bloco

```
{
  var x = 10;
}

console.log(x); // Funciona (não recomendado)
```

Conclusão: - `let` e `const` respeitam escopo de bloco - `var` ignora o bloco

---

4. Hoisting (Elevação)

Hoisting é o comportamento em que **declarações são processadas antes da execução do código**.

O JavaScript não move o código fisicamente, mas **interpreta como se declarações estivessem no topo do escopo**.

Com `var`

```
console.log(a); // undefined
var a = 10;
```

Interpretação real:

```
var a;
console.log(a);
a = 10;
```

---

5. Declaração vs Inicialização

• Declaração:

```
var x;
```

• Inicialização:

```
x = 5;
```

Apenas a **declaração** é hoisted.

---

6. Hoisting com `let` e `const`

`let` e `const` também sofrem hoisting, porém **não são inicializadas automaticamente**.

Isso cria a **Temporal Dead Zone (TDZ)**.

```
console.log(b); // Erro
let b = 20;
```

---

7. Temporal Dead Zone (TDZ)

A TDZ é o período entre: - A criação do escopo - A inicialização da variável

Durante esse período, a variável **existe**, mas **não pode ser acessada**.

```
{
  console.log(x); // Erro
  let x = 5;
}
```

Isso torna `let` e `const` mais seguros que `var`.

---

8. Variáveis Automaticamente Globais

Atribuir valor a uma variável **sem declarar** cria uma variável global (fora do modo estrito).

```
function myFunction() {
  carName = "Volvo";
}

myFunction();
console.log(carName);
```

Esse comportamento é perigoso e deve ser evitado.

---

9. Modo Estrito (`"use strict"`)

No modo estrito, o JavaScript **não permite variáveis não declaradas**.

```
"use strict";

function myFunction() {
  carName = "Volvo"; // Erro
}
```

Benefícios

- Evita globais acidentais
- Transforma erros silenciosos em erros reais
- Código mais seguro e previsível

---

10. Escopo Global no Navegador (`window`)

No navegador, o escopo global é o objeto `window`.

```
var carName = "Volvo";
console.log(window.carName); // Funciona
```

```
let carName = "Volvo";
console.log(window.carName); // undefined
```

Evite `var` no escopo global.

---

11. Closures

Uma **closure** ocorre quando uma função mantém acesso ao escopo onde foi criada.

```
function contador() {
  let count = 0;

  return function () {
    count++;
    return count;
  };
}

const incrementa = contador();
incrementa(); // 1
incrementa(); // 2
```

Mesmo após a execução de `contador`, a variável `count` continua existindo.

---

12. Tempo de Vida das Variáveis

- Variáveis globais: enquanto a página estiver aberta
- Variáveis de função: enquanto a função executa
- Variáveis de bloco: enquanto o bloco existir

Blocos de Código em JavaScript

Um **bloco de código** é um conjunto de instruções delimitado por `{ }`.

Eles são fundamentais para: - Controle de fluxo - Definição de escopo - Organização do código - Encapsulamento

```
{
  // bloco de código
}
```

Blocos Independentes

Blocos podem existir sem `if`, `for` ou função.

```
{
  let largura = 10;
  let altura = 20;
  let area = largura * altura;
}
```

Criam escopo local e evitam poluir o escopo global.

Encapsulamento

```
{
  const taxa = 0.15;
  let preco = 100;
  let total = preco + preco * taxa;
}
```

As variáveis não existem fora do bloco.

Organização e Boas Práticas

- Use blocos para agrupar lógica relacionada
- Prefira escopos pequenos
- Use `let` e `const`
- Evite `var`
- Evite depender de hoisting
- Escreva código previsível