JAVASCRIPT

Apresentação: Daniel Lessa
Adilson Silva

Introdução Sobre JAVASCRIPT

UMA BREVE INTRODUÇÃO SOBRE JAVASCRIPT

Javascript é uma linguagem de programação que os desenvolvedores usam para criar interações mais dinâmicas ao desenvolver páginas da Web, aplicativos, servidores, jogos, etc. Ele cria elementos para melhorar a interação dos visitantes do site com as páginas da web, como menus suspensos, gráficos animados e cores de fundo dinâmicas.

Principais vantagens

- **Simplicidade** – tem uma estrutura simples que o torna fácil de aprender e implementar, além de rodar mais rápido do que algumas outras linguagens. Os erros também são fáceis de detectar e corrigir.
- **Velocidade** – executa scripts diretamente no navegador da web sem se conectar a um servidor primeiro ou precisar de um compilador. Além disso, a maioria dos principais navegadores permite que o JavaScript compile códigos durante a execução do programa.
- **Versatilidade** – é compatível com outras linguagens como PHP, Perl e Java.
- **Popularidade** – muitos recursos e fóruns estão disponíveis para ajudar iniciantes com habilidades técnicas limitadas.
- **Carga do servidor** – reduz as solicitações enviadas ao servidor. A validação de dados pode ser feita através do navegador da web e as atualizações se aplicam apenas a determinadas seções da página da web.

Gerenciamento de Memória em JAVASCRIPT

- A alocação de memória em JavaScript é um processo fundamental.
- Linguagens de baixo nível, como C, tem gerenciamento de memória de baixo nível o desenvolvedor usa funções como malloc() e free() para alocar e liberar espaço na memória.
- Em contrapartida, os valores do JavaScript são alocados quando coisas (objetos, strings, variáveis etc.) são criadas e "automaticamente" liberadas quando não são mais usadas.
- Este último processo se chama garbage collection. Com sistema de gerenciamento automático de memória, através da coleta de lixo (garbage collection).

Ciclo de vida da memória

Independentemente da linguagem de programação, o ciclo de vida da memória é praticamente sempre o mesmo:

- Alocar a memória que você precisa.
- Utilizar a memória alocada (ler, escrever).
- Liberar a memória alocada quando não é mais necessária.
- A primeira e a segunda parte são explícitas em todas as linguagens. A última parte é explicita em linguagens de baixo nível, porém implícito em linguagens de alto nível como JavaScript.

Alocação Estática

Um exemplo de alocação estática na memória são as variáveis globais permanecem na memória durante toda a execução do programa o JavaScript faz isso com os valores conforme são declarados.

```
// Definindo uma constante global
const PI = 3.14159;

// Definindo um array estático
const numeros = [1, 2, 3, 4, 5];

// Definindo um objeto com propriedades fixas
const pessoa = {
    nome: "João",
    idade: 30
};

// A variável PI tem o valor fixo durante toda a execução do programa
console.log(PI); // 3.14159
```

ALOCAÇÃO DINÂMICA DE MEMÓRIA

- A alocação dinâmica de memória é um processo que permite alocar memória em tempo de execução, quando não se sabe exatamente quanto de memória será necessário. Isso evita o desperdício de memória.
- A alocação de memória também está relacionada ao escopo das variáveis. Variáveis locais (dentro de uma função) são destruídas quando a função termina, liberando a memória que ocupavam.
- Tipos Complexos: Objetos e arrays são armazenados no heap (uma área de memória usada para alocação dinâmica). A variável, nesse caso, mantém uma referência ao local na memória onde o objeto ou array está armazenado.

INVERTENDO UMA STRING

javascript

```javascript
function inverterString(str) {
    // Caso base: quando a string tem 0 ou 1 caractere, já está invertida
    if (str.length <= 1) {
        return str;
    }

    // Recursivamente inverter a substring (sem o primeiro caractere)
    return inverterString(str.slice(1)) + str[0];
}

const resultado = inverterString("caneta");
console.log(resultado); // Saída: "atenac"
```

Chamada Inicial (inverterString("caneta")):

1. A condição a função é chamada com a string "caneta".
   if (str.length <= 1) é false, pois a string tem 6 caracteres.
   Faz a chamada recursiva para "aneta" (usando str.slice(1)) e irá concatenar "c" (o primeiro caractere) ao resultado.
2. Segunda Chamada (inverterString("aneta")):
   Chama recursivamente com "neta", e concatenará "a" ao resultado.
3. Terceira Chamada (inverterString("neta")):
   Chama recursivamente com "eta", e concatenará "n" ao resultado.
4. Quarta Chamada (inverterString("eta")):
   Chama recursivamente com "ta", e concatenará "e" ao resultado.
5. Quinta Chamada (inverterString("ta")):
   Chama recursivamente com "a", e concatenará "t" ao resultado.
6. Sexta Chamada (inverterString("a")):
   Retorna "a" (caso base).
   A string final será construída à medida que as chamadas retornam, formando "atenac".

Reconstrução da String:

Agora que o caso base foi alcançado, as funções começam a retornar, e a string é reconstruída na ordem invertida:

- Sexta chamada retorna "a".
- Quinta chamada retorna "a" + "t" = "at".
- Quarta chamada retorna "at" + "e" = "ate".
- Terceira chamada retorna "ate" + "n" = "aten".
- Segunda chamada retorna "aten" + "a" = "atena".
- Primeira chamada retorna "atena" + "c" = "atenac".
- A string invertida final é "atenac", que é a saída.

Alocação Dinâmica e Recursão:

- A cada chamada recursiva, o método str.slice(1) cria novas instâncias de string que são alocadas dinamicamente. Isso porque strings em JavaScript são imutáveis, o que significa que uma nova string é criada a cada operação de slice(), e essa nova string é passada para a próxima chamada recursiva.

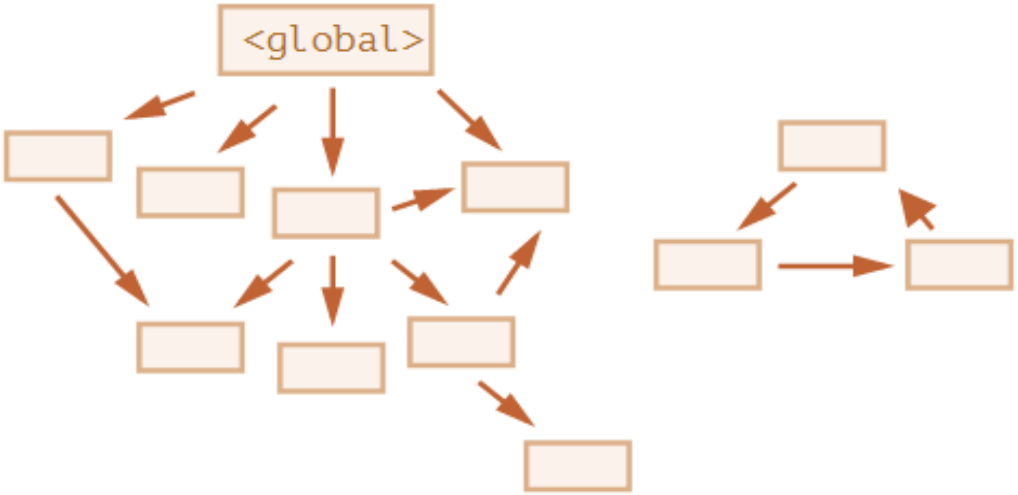
- Aqui, a alocação dinâmica ocorre na stack de chamadas recursivas, mas, tecnicamente, a alocação das novas strings (que são substrings) acontece na heap, já que JavaScript armazena strings como objetos. No entanto, como as strings são imutáveis, cada nova substring é uma nova instância que ocupa memória até que o processo recursivo termine.

GARBAGE COLLECTION

Coleta de Lixo: O motor JavaScript periodicamente executa a coleta de lixo para liberar memória que não é mais utilizada. O algoritmo mais comum é o "mark-and-sweep", que funciona em duas fases:

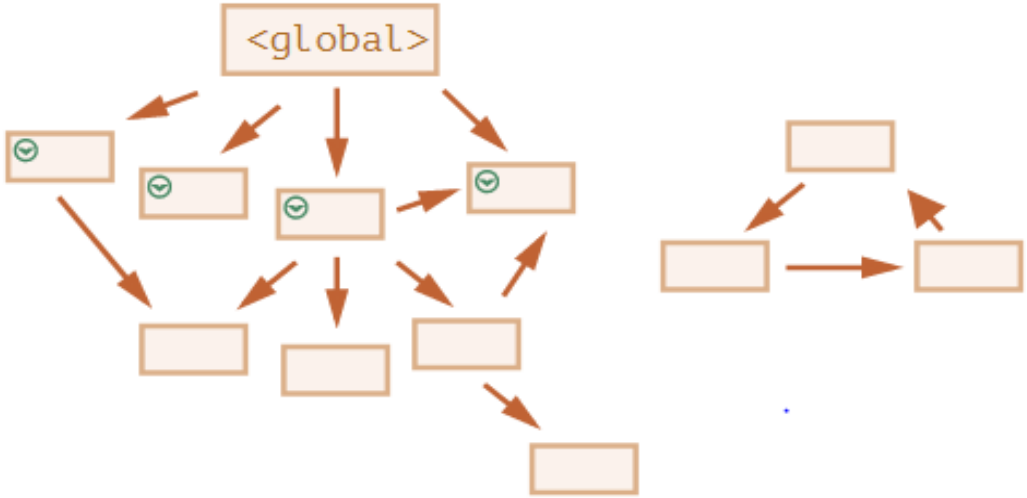
- **Marcação**: O coletor identifica quais objetos estão acessíveis (ou "marcados") a partir de variáveis em escopo. Todos os objetos visitados são lembrados, para não visitar o mesmo objeto duas vezes no futuro.
- **Varredura**: Os objetos que não estão marcados são considerados não acessíveis e, portanto, podem ser removidos da memória.

Por exemplo, deixe nossa estrutura de objeto ficar assim:

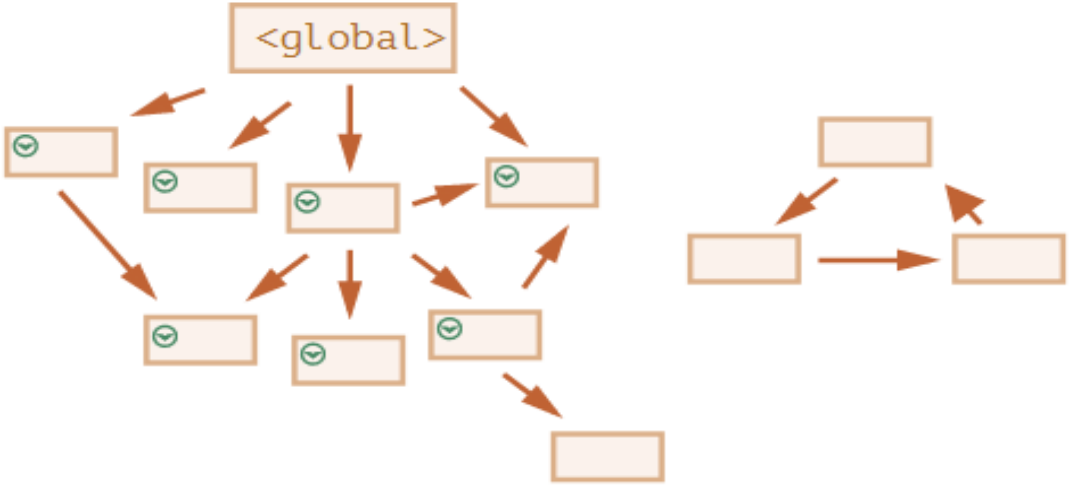


Podemos ver claramente uma “ilha inalcançável” do lado direito. Agora vamos ver como o coletor de lixo “mark-and-sweep” lida com isso.

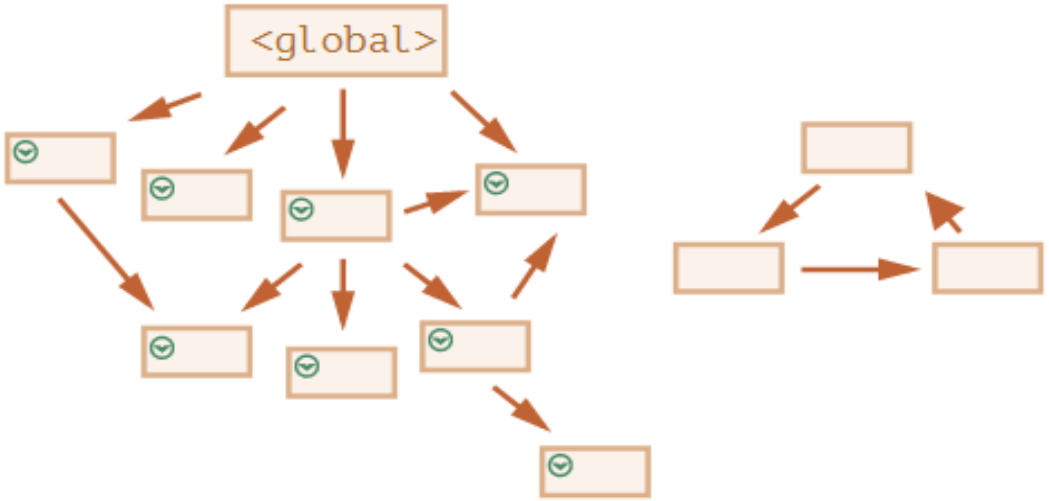
O primeiro passo marca as raízes:



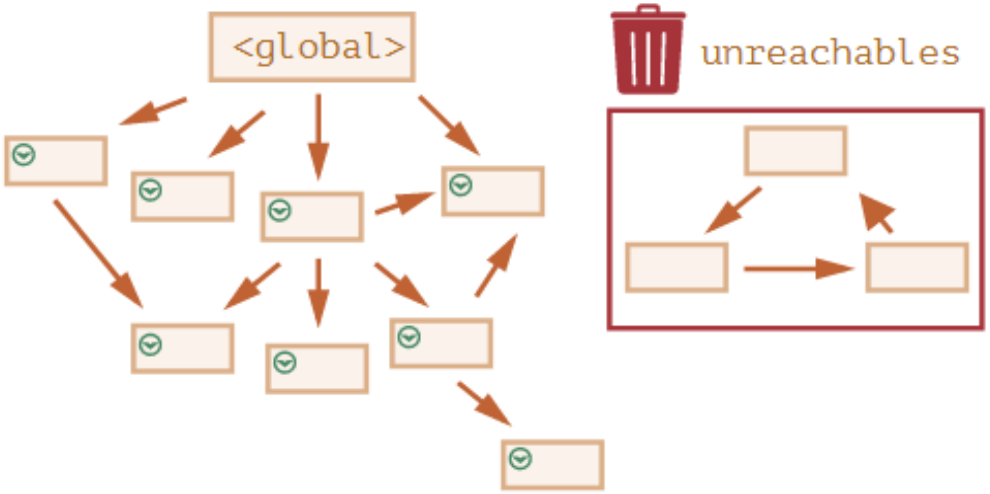
Em seguida, seguimos suas referências e marcamos os objetos referenciados:



…E continue acompanhando outras referências, enquanto possível:



Agora os objetos que não puderam ser visitados no processo serão considerados inacessíveis e serão removidos:



EM EXEMPLO PRÁTICO :

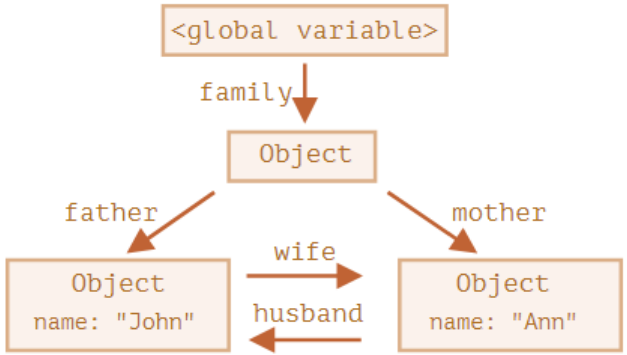
A família.

```
1  function marry(man, woman) {
2    woman.husband = man;
3    man.wife = woman;
4
5    return {
6      father: man,
7      mother: woman
8    }
9  }
10
11 let family = marry({
12   name: "John"
13 }, {
14   name: "Ann"
15 });
```

Objetos interligados

A função marry “casa” dois objetos dando a eles referências um ao outro e retorna um novo objeto que contém ambos.

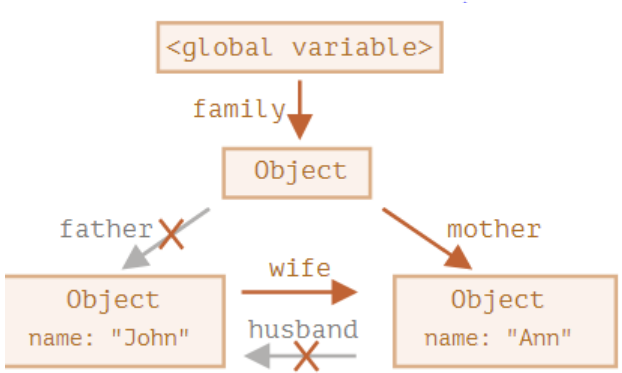
A estrutura de memória resultante:



A partir de agora, todos os objetos estão acessíveis.

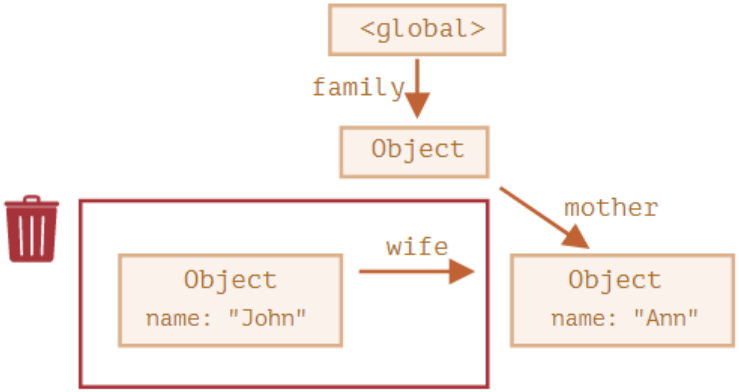
Agora vamos remover duas referências:

```
1  delete family.father;
2  delete family.mother.husband;
```



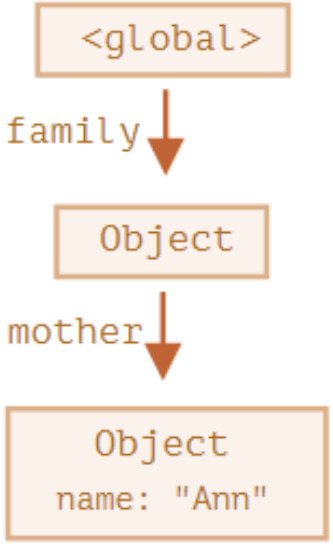
Não basta excluir apenas uma dessas duas referências, porque todos os objetos ainda estariam acessíveis.

Mas se excluirmos ambos, veremos que John não tem mais nenhuma referência de entrada:



Referências de saída não importam. Somente as de entrada podem tornar um objeto acessível. Então, John agora está inacessível e será removido da memória com todos os seus dados que também se tornaram inacessíveis.

Após a coleta de lixo:

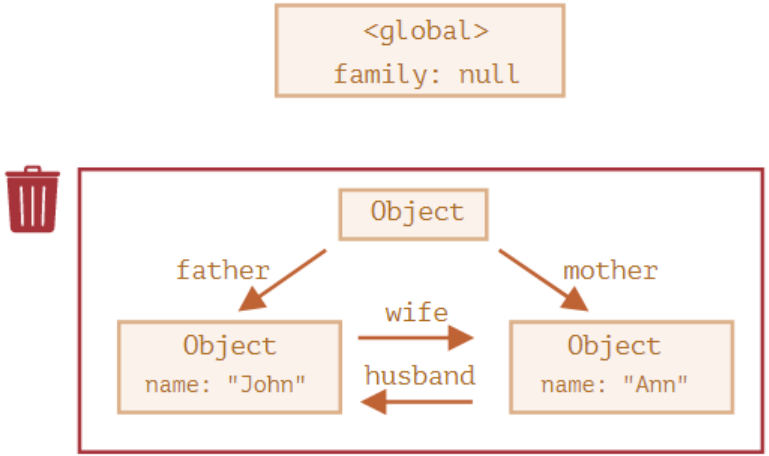


Ilha inacessível

É possível que toda a ilha de objetos interligados se torne inacessível e seja removida da memória.

O objeto de origem é o mesmo que o acima. Então:

```
1  family = null;
```



O "family" objeto antigo foi desvinculado da raiz, não há mais nenhuma referência a ele, então a ilha inteira se torna inacessível e será removida.

TIPOS DE DADOS EM JAVASCRIPT

O tipo **Boolean**: Valores lógicos: true (verdadeiro) ou false (falso).

```
let estaChovendo = true;  // A variável é um valor booleano
let solBrilhando = false; // Outro valor booleano
```

Null: Indicar que a variável não tem um valor definido ou que está intencionalmente vazia.

```
let valor = null;
console.log(valor);
```

Undefined: Variável é declarada, mas não inicializada com um valor,

```
let nome;
console.log(nome); // Saída: undefined
```

Number: Em JavaScript, não existem tipos de dados separados como float, double ou int usa um único tipo de dado para representar números: o tipo Number.

```
let inteiro = 42;        // Um número inteiro
let flutuante = 3.14;    // Um número flutuante
```

TIPOS DE DADOS EM JAVASCRIPT

BigInt: O tipo BigInt em JavaScript é usado para representar números inteiros muito grandes, cálculos de criptografia, grandes valores financeiros, ou ao lidar com identificadores únicos (como IDs de banco de dados).

```
let numeroGrande = 12345678901234567890123456789012345678901234567890n; // O "n" no final define um
console.log(numeroGrande); // Saída: 12345678901234567890123456789012345678901234567890n
```

String: Representar sequências de caracteres (texto). Permite interpolação (incluir variáveis dentro da string, como em `Bem-vindo, ${nome}!``).

```
let nome = "João";      // Usando aspas duplas
```

```
let mensagem = `Bem-vindo, ${nome}!`; // Usando template literal (com crase)
```

TIPOS DE DADOS EM JAVASCRIPT

Symbol: Symbol cria valores únicos e imutáveis, úteis para garantir que as chaves de propriedades em objetos não colidam.

```
let simbolo1 = Symbol('descrição');
let simbolo2 = Symbol('descrição');


console.log(simbolo1 == simbolo2);  // Saída: false. São símbolos únicos
```

Objetos: Uma coleção de pares chave-valor. Ele permite armazenar múltiplos valores sob um mesmo nome, como propriedades e métodos (funções associadas ao objeto). Objetos são úteis para organizar dados e representar entidades no código.

```
const pessoa = {
  nome: 'João',
  idade: 30,
  saudacao: function() { return 'Olá'; }
};
```

Tipagem dinâmica e fraca

JavaScript é uma linguagem dinâmica com tipos dinâmicos. As variáveis em JavaScript não estão diretamente associadas a nenhum tipo de valor específico, e qualquer variável pode receber (e reatribuir) valores de todos os tipos:

```
let foo = 42; // foo agora é um número
foo = "bar"; // foo agora é uma string
foo = true; // foo agora é um booleano
```

JavaScript também é uma linguagem de tipagem fraca, o que significa que permite a conversão implícita de tipo quando uma operação envolve tipos incompatíveis, em vez de gerar erros de tipo.

```
const foo = 42; //foo é um número
const result = foo + "1"; // JavaScript coage foo para uma string, então ela pode ser
concatenada com o outro operando
console.log(resultado); // 421
```

Sintaxe

A sintaxe do JavaScript é baseada em linguagens como C e Java, com chaves `{}` para delimitar blocos de código, ponto e vírgula `;` opcional (mas recomendado para evitar ambiguidades), e palavras-chaves comuns, como `if`, `for`, `function`, `let`, `const`, entre outras. JavaScript é uma linguagem case-sensitive e permite comentários de linha (`//`) e de bloco (`/* ... */`).

Principais Elementos da Sintaxe

- **Declaração de Variáveis:**
  - let: Declara uma variável local de escopo do bloco, opcionalmente, inicializando-a com um valor.
  - const: Declara uma constante de escopo de bloco, o valor de uma constante não pode ser alterado por uma atribuição, e ela não pode ser redeclarada. Toda constante deve ser inicializada.
  - var: Declara uma variável com o escopo global, opcionalmente, inicializando-a com um valor.

```
let nome = "Maria";

const PI = 3.14159;

var cidade = "Rio de Janeiro";
```

Sintaxe

Exemplos:

```
let numero = 0;

// Exemplo de if
if (numero > 0) {
  console.log("O número é positivo.");
}

// Exemplo de if-else
if (numero < 0) {
  console.log("O número é negativo.");
} else {
  console.log("O número não é negativo.");
}

// Exemplo de else if
if (numero > 0) {
  console.log("O número é positivo.");
} else if (numero < 0) {
  console.log("O número é negativo.");
} else {
  console.log("O número é zero.");
}
```

Estruturas Condicionais

```
// Exemplo de For
for (let i = 0; i < 5; i++) {
  console.log(i); // Saida: 0, 1, 2, 3, 4
}

// Exemplo de ForEach
const numeros = [0, 1, 2, 3, 4];
numeros.forEach(function(num) {
  console.log(num);  // Saida: 0, 1, 2, 3, 4
});

// Exemplo de While
let i = 0;
while (i < 5) {
  console.log(i);  // Saida: 0, 1, 2, 3, 4
  i++;
}

// Exemplo de Do While
let i = 0;
do {
  console.log(i);  // Saida: 0, 1, 2, 3, 4
  i++;
} while (i < 5);
```

Laços de Repetição

Sintaxe

- **Declaração de Arrays:**
  - Declaração de Arrays (com `[]`): A forma mais comum de declarar um array em JavaScript é usando colchetes (`[]`). Dentro dos colchetes, você coloca os elementos separados por vírgulas.

```
let numeros = [1, 2, 3, 4, 5];
console.log(numeros);  // Saída: [1, 2, 3, 4, 5]
```

```
let vazio = [];
console.log(vazio);  // Saída: []
```

  - Usando o Construtor `new Array()`: Também é possível usar o construtor `Array` para criar um array. Porém, essa forma não é tão comum, exceto em casos onde você precisa criar um array com um tamanho específico.

```
let frutas = new Array('maçã', 'banana', 'laranja');
console.log(frutas);  // Saída: ["maçã", "banana", "laranja"]
```

  - Array com Diferentes Tipos de Dados: Arrays podem conter diferentes tipos de dados ao mesmo tempo, como números, strings, objetos, etc.

```
let mistura = [1, 'texto', true, { nome: 'Objeto' }, [1, 2, 3]];
console.log(mistura);  // Saída: [1, "texto", true, { nome: "Objeto" }, [1, 2, 3]]
```

Sintaxe

- **Funções**:
  - Tradicional: É definida usando a palavra-chave `function`, seguida pelo nome da função e seus parâmetros.

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

  - Arrow Function: É uma forma mais concisa de declarar funções em JavaScript. Ela utiliza a sintaxe `() => {}`.

```
const soma = (a, b) => a + b;
```

  - Anônima: É uma função que não tem um nome explícito. Ela pode ser usada onde uma função é esperada, como em variáveis ou como argumento de outra função.

```
// A função anônima é atribuída à variável 'somar'
const somar = function(a, b) {
  return a + b;
};

// Usando a função anônima
console.log(somar(5, 3)); // Saída: 8
```

```
const numeros = [1, 2, 3, 4, 5];

// Função anônima passada como argumento para o método 'forEach'
numeros.forEach(function(num) {
  console.log(num * 2);
});
// Saída:
// 2
// 4
// 6
// 8
// 10
```

Programação Funcional

Definição: Paradigma de programação focado em **funções** e **imutabilidade**. Baseia-se em funções puras que não modificam o estado global e sempre retornam o mesmo resultado para os mesmos inputs.

Princípios Básicos:

- **Funções Puras**: Função que, para as mesmas entradas, sempre retorna a mesma saída, sem causar efeitos colaterais no programa.
- **Imutabilidade**: Dados não são modificados, mas criados novos.
- **Ausência de Estado Compartilhado**: O estado compartilhado ocorre quando múltiplas funções ou partes do código podem acessar e modificar a mesma variável ou objeto. Em programação funcional, isso é evitado.

Como Funciona no JavaScript?

- **Funções de Ordem Superior**: Funções que operam sobre outras funções ou as recebendo como parâmetro ou as retornando.
- **Funções Anônimas e Arrow Functions**: Sintaxe mais compacta para criar funções.
- **Métodos Funcionais do Array**:
  - map: Transforma os elementos de um array.
  - filter: Filtra elementos com base em uma condição.
  - reduce: Agrega valores de um array em um único resultado.

Programação Funcional

Exemplos:

1. map:

```
const numeros = [1, 2, 3, 4];
const dobrados = numeros.map(num => num * 2);
console.log(dobrados); // Saída: [2, 4, 6, 8]
```

Transforma cada elemento do array original sem modificá-lo.

2. filter:

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

Filtra elementos do array com base em uma condição.

3. reduce:

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

O acumulador é iniciado com 0 (segundo argumento de reduce) e, a cada iteração, soma o valor do elemento atual (num).

Programação Funcional

Razões e Benefícios da Programação Funcional

- **Simplificação do Código**:

  Código funcional é mais declarativo e legível.

- **Previsibilidade**:

  Funções puras facilitam testes e reduzem erros.

- **Performance e Confiabilidade**:

  Evita estados compartilhados e problemas de concorrência.

- **Popularidade Crescente**:

  Amplamente usada em frameworks modernos como React.