# Aula 10 – Introdução ao JavaScript e Variáveis

Imagine um site como um carro. O HTML seria a estrutura, a carroceria, o chassi – tudo o que define o que o carro é. O CSS, por sua vez, seria a pintura, os estofamentos, as rodas esportivas – tudo o que o torna bonito e estilizado. Mas o que faz o carro andar, ligar o rádio, acender os faróis ou abrir os vidros? É a mecânica, a eletrônica, o motor. No mundo da web, essa "mecânica" que dá vida e interatividade é o JavaScript. Ele transforma páginas estáticas em experiências dinâmicas e responsivas.

Nesta aula, vamos dar os primeiros passos nesse universo fascinante. Você descobrirá por que o JavaScript se tornou a espinha dorsal de quase toda aplicação web moderna, desde simples formulários interativos até complexas plataformas de streaming. Entenderemos como ele se encaixa no ecossistema de desenvolvimento frontend e como as tendências atuais, como a performance e a acessibilidade, são intrínsecas ao seu uso.

Nosso objetivo é que, ao final desta jornada, você não apenas compreenda o papel fundamental do JavaScript, mas também seja capaz de escrever seu primeiro código funcional. Abordaremos a sintaxe básica, aprenderá a incluir scripts em suas páginas e, crucialmente, dominará o conceito de variáveis e constantes – os "contêineres" onde guardamos informações. Veremos também os tipos de dados mais comuns e como identificá-los, preparando o terreno para manipular informações de forma eficaz.

# O Coração Dinâmico da Web: O Papel do JavaScript

## A Web Estática do Passado

No início da internet, as páginas eram como panfletos digitais: estáticas, informativas, mas sem muita interação. Você clicava em um link e era levado a outra página, mas a página em si não "respondia" aos seus comandos de forma inteligente. Era um mundo de informações unidirecionais, onde a experiência do usuário era limitada à leitura e navegação linear. Essa limitação logo se tornou um gargalo para a evolução da web.

## A Revolução JavaScript

Foi nesse cenário que o JavaScript surgiu, inicialmente para pequenas validações de formulários e efeitos visuais simples. No entanto, sua capacidade de manipular o conteúdo, o estilo e o comportamento das páginas em tempo real, diretamente no navegador do usuário, revolucionou tudo. Ele permitiu que os desenvolvedores criassem interfaces ricas, com menus interativos, galerias de imagens dinâmicas, jogos, e até mesmo aplicações completas que funcionam sem recarregar a página a cada ação.

**JavaScript Hoje:** O JavaScript não é apenas uma linguagem para "enfeitar" a web; ele é a força motriz por trás de frameworks e bibliotecas poderosas como React, Angular e Vue.js, que constroem as aplicações web mais sofisticadas que usamos diariamente. Ele é essencial para garantir que as páginas sejam rápidas (contribuindo para as Core Web Vitals), acessíveis (permitindo a criação de componentes interativos que funcionam bem com tecnologias assistivas) e responsivas, adaptando-se a diferentes dispositivos e interações.

# Desvendando a Linguagem: Sintaxe Básica e Comentários

### Sintaxe

Toda linguagem, seja ela humana ou de programação, possui suas regras e sua forma de expressar ideias. No JavaScript, essa estrutura é chamada de sintaxe. É o conjunto de regras que define como você deve escrever seu código para que o navegador (ou outro ambiente de execução) possa entendê-lo e executá-lo corretamente.

### Ponto e Vírgula

A maioria dos comandos, ou "declarações", termina com um ponto e vírgula (;), embora em muitos casos ele seja opcional devido a um recurso chamado Automatic Semicolon Insertion (ASI). No entanto, é uma boa prática utilizá-lo para evitar ambiguidades e garantir a clareza do código, especialmente para iniciantes.

## Exemplo de Sintaxe Básica

```
// Este é um exemplo simples de uma declaração em JavaScript
console.log("Olá, mundo!"); // Exibe uma mensagem no console do navegador
```

## Comentários: Anotações no Código

Além de escrever o código que será executado, é fundamental poder adicionar anotações e explicações dentro do próprio código. É aqui que entram os **comentários**. Pense neles como notas adesivas que você cola em um livro para lembrar de algo importante ou explicar um trecho complexo. O navegador ignora completamente os comentários; eles existem apenas para os desenvolvedores, facilitando a compreensão do código, tanto para quem o escreveu quanto para quem o lerá no futuro.

### Comentários de Linha Única

Começam com duas barras (//) e se estendem até o final da linha. São ideais para explicações rápidas ou para desativar temporariamente uma linha de código.

### Comentários de Múltiplas Linhas

Começam com /* e terminam com */. São perfeitos para blocos maiores de texto, como descrições de funções ou seções inteiras do código.

```
// Este é um comentário de linha única.
// Ele explica o que a linha de código abaixo faz.
let nome = "João"; // Declara uma variável chamada 'nome' e atribui o valor "João"

/* Este é um comentário de múltiplas linhas.
 Ele pode ser usado para descrever um bloco de código maior,
 ou para fornecer informações mais detalhadas sobre uma função.
*/
function saudacao(nomeUsuario) {
 console.log("Olá, " + nomeUsuario + "!");
}
```

# Dando Vida à Página: Inclusão de Scripts

Depois de entender a sintaxe básica e como documentar seu código com comentários, o próximo passo lógico é saber como fazer com que o navegador execute esse JavaScript. Afinal, de que adianta escrever um código brilhante se ele não puder interagir com sua página HTML? A inclusão de scripts é o elo que conecta o seu código JavaScript ao documento HTML, permitindo que ele manipule elementos, responda a eventos e crie a experiência dinâmica que esperamos da web moderna.

A forma mais comum e recomendada de incluir JavaScript em uma página HTML é utilizando a tag <script>. Essa tag atua como uma ponte, indicando ao navegador que o conteúdo entre suas tags de abertura e fechamento, ou o arquivo referenciado por ela, deve ser interpretado como código JavaScript.

## Duas Abordagens Principais

### Scripts Internos

O código JavaScript é escrito diretamente dentro da tag <script> no seu arquivo HTML. Isso é útil para pequenos trechos de código específicos para aquela página, mas pode deixar o HTML poluído e dificultar a manutenção em projetos maiores.

### Scripts Externos

O código JavaScript é escrito em um arquivo separado (com extensão .js) e depois "linkado" ao HTML usando o atributo src na tag <script>. Esta é a prática recomendada para a maioria dos projetos, pois organiza o código, facilita a reutilização e melhora a legibilidade e manutenção.

## Exemplo: Script Interno

```
<!DOCTYPE html>
<html lang="pt-br">
<head>
  <meta charset="UTF-8">
  <title>Minha Primeira Página com JS</title>
</head>
<body>
  <h1>Bem-vindo!</h1>
  <button onclick="alert('Você clicou no botão!')">Clique-me</button>
  <script>
    // Script interno: executado diretamente na página
    console.log("Este script está dentro do HTML.");
  </script>
</body>
</html>
```

## Exemplo: Script Externo

```
<!DOCTYPE html>
<html lang="pt-br">
<head>
  <meta charset="UTF-8">
  <title>Minha Página com Script Externo</title>
</head>
<body>
  <h1>Olá do Script Externo!</h1>
  <script src="meu-script.js"></script> <!-- Link para o arquivo JS externo -->
</body>
</html>
```

E no arquivo meu-script.js:

```
// Este é o conteúdo do arquivo meu-script.js
console.log("Este script veio de um arquivo externo!");
```

## Onde Colocar os Scripts e Atributos Especiais

Tradicionalmente, os scripts eram colocados no <head> da página. No entanto, isso podia atrasar o carregamento do conteúdo visual, pois o navegador parava de renderizar o HTML para executar o JavaScript. A melhor prática moderna é colocar a tag <script> (especialmente as externas) **antes do fechamento da tag </body>**. Isso garante que o HTML seja carregado e exibido primeiro, proporcionando uma experiência mais rápida ao usuário.

### async

O script é baixado em paralelo com a análise do HTML e executado assim que estiver disponível, sem bloquear a renderização. A ordem de execução dos scripts com async não é garantida.

### defer

O script é baixado em paralelo com a análise do HTML, mas sua execução é adiada até que todo o HTML tenha sido analisado. A ordem de execução é mantida para scripts com defer.

**Recomendação:** Para a maioria dos casos, especialmente quando o JavaScript precisa interagir com o DOM (Document Object Model) já carregado, o atributo `defer` é a escolha mais segura e performática, pois garante que o HTML esteja pronto antes da execução do script.

```
<!DOCTYPE html>
<html lang="pt-br">
<head>
 <meta charset="UTF-8">
 <title>Otimizando o Carregamento</title>
</head>
<body>
 <h1>Página Otimizada!</h1>
 <script src="script-essencial.js" defer></script> <!-- Carrega em paralelo, executa após HTML -->
 <script src="script-analitico.js" async></script> <!-- Carrega em paralelo, executa assim que pronto -->
</body>
</html>
```

# Guardando Informações: Variáveis e Constantes

Imagine que você está organizando uma festa e precisa guardar o nome dos convidados, a data do evento e a quantidade de salgadinhos. Você não vai simplesmente gritar essas informações para o ar; você as anota em algum lugar: uma lista, um calendário, um bilhete. No mundo da programação, quando precisamos armazenar dados para usá-los mais tarde, fazemos algo semelhante: usamos **variáveis** e **constantes**. Elas são como "caixas" ou "contêineres" nomeados na memória do computador, onde podemos guardar valores.

A capacidade de armazenar e manipular dados é o que torna o JavaScript tão poderoso. Sem variáveis, cada pedaço de informação teria que ser reescrito toda vez que fosse usado, tornando o código repetitivo, difícil de ler e impossível de manter. Variáveis e constantes nos permitem dar um nome significativo a um valor, facilitando a referência e a modificação desses dados ao longo da execução do nosso programa.

Historicamente, o JavaScript utilizava apenas a palavra-chave var para declarar variáveis. No entanto, com a evolução da linguagem e a necessidade de um controle de escopo mais robusto (onde a variável é acessível), foram introduzidas let e const no ES6 (ECMAScript 2015). Essas novas formas de declaração se tornaram o padrão de mercado, especialmente em projetos modernos que utilizam ferramentas como Vite, pois promovem um código mais previsível e com menos erros.

## var: A Declaração Tradicional (e suas Armadilhas)

A palavra-chave var foi a única forma de declarar variáveis em JavaScript por muitos anos. Ela tem um escopo de função, o que significa que uma variável declarada com var dentro de uma função é acessível em qualquer lugar dentro dessa função, mas não fora dela. Fora de funções, ela tem escopo global.

Um dos principais problemas de var é o "hoisting" (içamento) e a possibilidade de redeclaração. Variáveis declaradas com var são "içadas" para o topo de seu escopo, o que pode levar a comportamentos inesperados. Além disso, você pode redeclarar a mesma variável com var sem erro, o que pode sobrescrever valores acidentalmente.

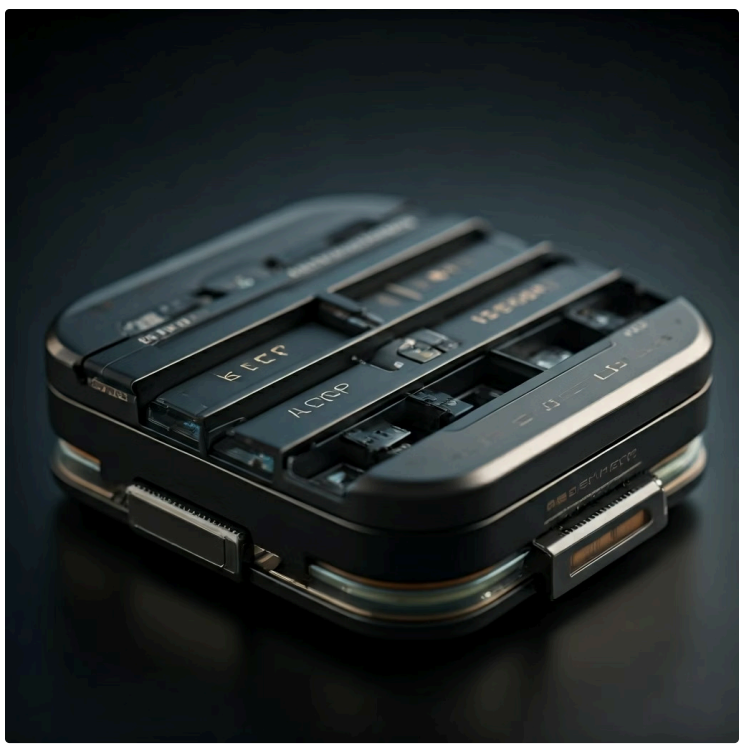
```
var nomeAntigo = "Maria";
console.log(nomeAntigo); // Saída: Maria

var nomeAntigo = "Ana"; // Redeclaração permitida, mas pode causar confusão
console.log(nomeAntigo); // Saída: Ana

if (true) {
    var idadeAntiga = 30;
}
console.log(idadeAntiga); // Saída: 30 (var não tem escopo de bloco, o que é uma armadilha)
```

⚠️ **Atenção:** Devido a essas características que podem levar a bugs difíceis de depurar, a prática recomendada em JavaScript moderno é **evitar o uso de var** em favor de let e const.

# let: A Variável Moderna e Flexível



Com a introdução do ES6, a palavra-chave let surgiu para resolver muitos dos problemas associados a var, tornando a declaração de variáveis mais segura e previsível. A principal diferença é que let possui **escopo de bloco**. Isso significa que uma variável declarada com let só é acessível dentro do bloco de código (delimitado por chaves {}) onde foi definida.

Pense no escopo de bloco como um conjunto de gavetas em uma cômoda. Se você guarda algo na gaveta do quarto, ele não está visível na gaveta da sala. Da mesma forma, uma variável let declarada dentro de um if, for ou qualquer outro bloco de código, só existe ali dentro. Isso ajuda a evitar conflitos de nomes e efeitos colaterais indesejados em partes diferentes do seu programa.

Além do escopo de bloco, let também não permite a redeclaração da mesma variável no mesmo escopo. Se você tentar declarar uma variável com o mesmo nome duas vezes usando let no mesmo bloco, o JavaScript irá gerar um erro, o que é ótimo para identificar e corrigir problemas de lógica logo cedo.

```
let nomeModerno = "Carlos";
console.log(nomeModerno); // Saída: Carlos

// let nomeModerno = "Pedro"; // ERRO: 'nomeModerno' já foi declarado.
// console.log(nomeModerno);

if (true) {
    let idadeModerna = 25;
    console.log(idadeModerna); // Saída: 25 (acessível dentro do bloco)
}
// console.log(idadeModerna); // ERRO: 'idadeModerna' não está definida (fora do escopo do bloco)
```

## const: Para Valores que Não Mudam

Enquanto let é para variáveis que podem ter seu valor alterado ao longo do tempo, const (de "constante") é usada para declarar valores que **não devem ser reatribuídos** após sua inicialização. Assim como let, const também possui escopo de bloco.

Imagine que você está definindo o valor de PI (3.14159) ou o nome de um site que nunca muda. Seria um erro se, acidentalmente, você tentasse mudar esses valores em algum ponto do seu código. const garante que, uma vez atribuído um valor, ele permanecerá o mesmo. Se você tentar reatribuir um novo valor a uma constante, o JavaScript lançará um erro.

> **Importante:** Ao declarar uma const, você **deve inicializá-la imediatamente** com um valor. Não é possível declarar uma constante sem atribuir-lhe um valor no mesmo momento.

```
const PI = 3.14159;
console.log(PI); // Saída: 3.14159

// PI = 3.14; // ERRO: Atribuição a uma variável constante.
// console.log(PI);

const URL_API = "https://api.meusite.com";
console.log(URL_API); // Saída: https://api.meusite.com

if (true) {
    const MAX_TENTATIVAS = 3;
    console.log(MAX_TENTATIVAS); // Saída: 3
}
// console.log(MAX_TENTATIVAS); // ERRO: 'MAX_TENTATIVAS' não está definida
```

## Quando usar let e const?

### Regra de Ouro

Use **const por padrão**. Se você sabe que o valor de uma variável não mudará, const é a escolha mais segura e clara.

### Quando Usar let

Use **let apenas quando souber que o valor precisará ser reatribuído** em algum momento.

Essa abordagem ajuda a escrever um código mais robusto, fácil de entender e com menos chances de erros acidentais de reatribuição.

# Comparativo: var, let e const

Para solidificar a compreensão sobre as diferenças entre var, let e const, vamos analisar um quadro comparativo. Entender essas distinções é crucial para escrever código JavaScript moderno, eficiente e livre de armadilhas. A escolha correta da palavra-chave para declarar suas variáveis e constantes impacta diretamente a legibilidade, a manutenção e a robustez do seu projeto.

A transição de var para let e const não foi apenas uma questão de preferência, mas uma evolução necessária da linguagem para lidar com a complexidade crescente das aplicações web. Ao adotar let e const, você está alinhando seu código com as melhores práticas da comunidade e com as expectativas de ferramentas modernas como o Vite, que incentivam um desenvolvimento mais estruturado e menos propenso a erros.

| Característica | var | let | const |
|---|---|---|---|
| **Escopo** | Função ou Global | Bloco | Bloco |
| **Reatribuição** | Permitida | Permitida | Não permitida (após inicialização) |
| **Redeclaração** | Permitida no mesmo escopo | Não permitida no mesmo escopo | Não permitida no mesmo escopo |
| **Hoisting** | Sim (inicializado com undefined) | Sim (mas em "Temporal Dead Zone") | Sim (mas em "Temporal Dead Zone") |
| **Inicialização** | Opcional (inicializa com undefined) | Opcional (inicializa com undefined) | Obrigatória no momento da declaração |
| **Melhor Uso** | Evitar em código moderno | Variáveis que mudarão de valor | Valores fixos e referências imutáveis |

**Observação sobre Hoisting e "Temporal Dead Zone" (TDZ):** Embora let e const também sofram "hoisting", eles não são inicializados com undefined como var. Em vez disso, eles entram em uma "Temporal Dead Zone" (TDZ) do início do escopo até o ponto onde são declarados. Tentar acessar uma variável let ou const antes de sua declaração explícita resultará em um ReferenceError, o que é uma segurança a mais contra o uso indevido.

# Os Tipos de Dados Primitivos: A Essência da Informação

Agora que sabemos como guardar informações em variáveis e constantes, a próxima pergunta natural é: que tipo de informações podemos guardar? Assim como em um supermercado você tem seções para frutas, laticínios, carnes, etc., o JavaScript categoriza os dados que ele pode manipular em **tipos de dados**. Entender esses tipos é fundamental, pois a forma como você interage com um dado (somar, comparar, exibir) depende do seu tipo.

O JavaScript é uma linguagem de tipagem dinâmica, o que significa que você não precisa declarar explicitamente o tipo de uma variável quando a cria (como em outras linguagens). O interpretador do JavaScript infere o tipo com base no valor que você atribui a ela. No entanto, isso não significa que os tipos não existam; eles são a base de como o JavaScript processa e armazena os dados.

Os tipos de dados primitivos são os blocos de construção mais básicos de informação em JavaScript. Eles representam valores simples e imutáveis (ou seja, seu valor não pode ser alterado diretamente, apenas substituído por um novo valor). Conhecer esses tipos é o primeiro passo para manipular dados de forma eficaz e evitar erros comuns.

### String: O Texto da Web

Uma **string** é uma sequência de caracteres, como palavras, frases ou qualquer texto. Em JavaScript, strings são delimitadas por aspas simples (' '), aspas duplas (" ") ou template literais (crases `).

### Number: Os Números da Computação

O tipo **number** representa tanto números inteiros quanto números de ponto flutuante (decimais). O JavaScript não faz distinção entre eles; ambos são tratados como number.

### Boolean: O Verdadeiro ou Falso da Lógica

Um **boolean** representa um valor lógico: true (verdadeiro) ou false (falso). É fundamental para tomadas de decisão e controle de fluxo em seu código.

## Exemplos de String

```
let saudacao = "Olá, mundo!"; // Aspas duplas
let mensagem = 'Bem-vindo ao curso.'; // Aspas simples
let nomeUsuario = "Alice";
let cumprimento = `Olá, ${nomeUsuario}! Como você está?`; // Template literal

console.log(saudacao); // Saída: Olá, mundo!
console.log(mensagem); // Saída: Bem-vindo ao curso.
console.log(cumprimento); // Saída: Olá, Alice! Como você está?
```

## Exemplos de Number

```
let idade = 30; // Número inteiro
let preco = 99.99; // Número de ponto flutuante
let resultado = idade + preco; // Operação matemática

console.log(idade); // Saída: 30
console.log(preco); // Saída: 99.99
console.log(resultado); // Saída: 129.99
```

## Exemplos de Boolean

```
let estaLogado = true;
let temPermissao = false;

if (estaLogado) {
    console.log("Usuário está online.");
} else {
    console.log("Usuário está offline.");
}
```

## null e undefined: A Ausência de Valor

### undefined

Significa que uma variável foi declarada, mas ainda não recebeu um valor. É o valor padrão de variáveis não inicializadas.

### null

Representa a ausência intencional de qualquer valor de objeto. É um valor que você atribui explicitamente para indicar "nada" ou "vazio".

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

let objetoVazio = null;
console.log(objetoVazio); // Saída: null
```

## Symbol e BigInt: Tipos Mais Recentes

- **Symbol:** Introduzido no ES6, Symbol cria um identificador único e imutável. É usado principalmente para chaves de propriedades de objetos que você quer garantir que não colidam com outras chaves.
- **BigInt:** Introduzido mais recentemente, BigInt permite representar números inteiros de tamanho arbitrário, algo que o tipo Number padrão não consegue lidar com precisão para valores muito grandes.

```
const idUnico = Symbol('id');
const outroIdUnico = Symbol('id');
console.log(idUnico === outroIdUnico); // Saída: false (são únicos)

const numeroGigante = 9007199254740991n; // O 'n' no final indica um BigInt
console.log(numeroGigante + 1n); // Saída: 9007199254740992n
```

# Desvendando o Tipo: O Operador typeof

Em JavaScript, como vimos, as variáveis não têm um tipo fixo declarado explicitamente. Uma variável pode começar armazenando um número e, mais tarde, receber uma string. Essa flexibilidade é uma característica da linguagem, mas às vezes precisamos saber qual é o tipo de dado que uma variável está armazenando em um determinado momento. É aí que entra o operador **typeof**.

O typeof é uma ferramenta simples, mas extremamente útil, que nos permite inspecionar o tipo de um valor ou de uma variável. Ele retorna uma string indicando o tipo de dado do operando. Pense nele como um "detetive de tipos" que te diz a natureza da informação que você está manipulando. Isso é particularmente importante em cenários onde você recebe dados de fontes externas (como uma API ou um formulário do usuário) e precisa garantir que eles estão no formato esperado antes de processá-los.

Saber o tipo de dado é crucial para evitar erros. Por exemplo, tentar somar uma string com um número pode levar a um resultado inesperado (concatenação em vez de soma aritmética). O typeof ajuda a implementar verificações e a garantir que suas operações sejam aplicadas aos tipos de dados corretos.

```
let nome = "Maria";
let idade = 28;
let estaAtivo = true;
let semValor;
let nulo = null;
let id = Symbol('usuario');
let numeroGrande = 123456789012345678901n;

console.log(typeof nome);       // Saída: "string"
console.log(typeof idade);      // Saída: "number"
console.log(typeof estaAtivo);  // Saída: "boolean"
console.log(typeof semValor);   // Saída: "undefined"
console.log(typeof id);         // Saída: "symbol"
console.log(typeof numeroGrande); // Saída: "bigint"

// Um detalhe importante: typeof null
console.log(typeof nulo);       // Saída: "object" (Este é um bug histórico do JavaScript, mas é importante saber)
```

> ⚠️ **Atenção:** Apesar da peculiaridade de `typeof null` retornar "object", para todos os outros tipos primitivos, typeof funciona como esperado, retornando uma string com o nome do tipo em minúsculas. Ele é uma ferramenta indispensável para depuração e para a criação de código mais robusto e defensivo, garantindo que você esteja sempre ciente da natureza dos dados com os quais está trabalhando.

# Próximos Passos e Recursos

**Gabarito**

1. c)
2. c)
3. a)
4. c)

## Próxima Aula

**Aula 11:** Na próxima aula, daremos um passo adiante e exploraremos os **Operadores e Estruturas Condicionais**. Você aprenderá a realizar cálculos, comparações e a tomar decisões em seu código, permitindo que seus programas reajam de forma inteligente a diferentes situações.

## Recursos Adicionais

### MDN Web Docs (Mozilla Developer Network)

Para documentação oficial e aprofundada sobre JavaScript.

### JavaScript.info

Para tutoriais detalhados e exemplos práticos.

### FreeCodeCamp

Para exercícios interativos e projetos de codificação.

**NOTA IMPORTANTE:** As informações técnicas desta aula estão atualizadas até 2025. Consulte sempre fontes oficiais como o MDN Web Docs para verificar alterações e as últimas especificações da linguagem.