

 Esta página foi traduzida do inglês pela comunidade. Saiba mais e junte-se à comunidade MDN Web Docs.

[View in English](#)

[Always switch to English](#)

# O que é JavaScript?

Menu: [Primeiros passos com JavaScript](#)

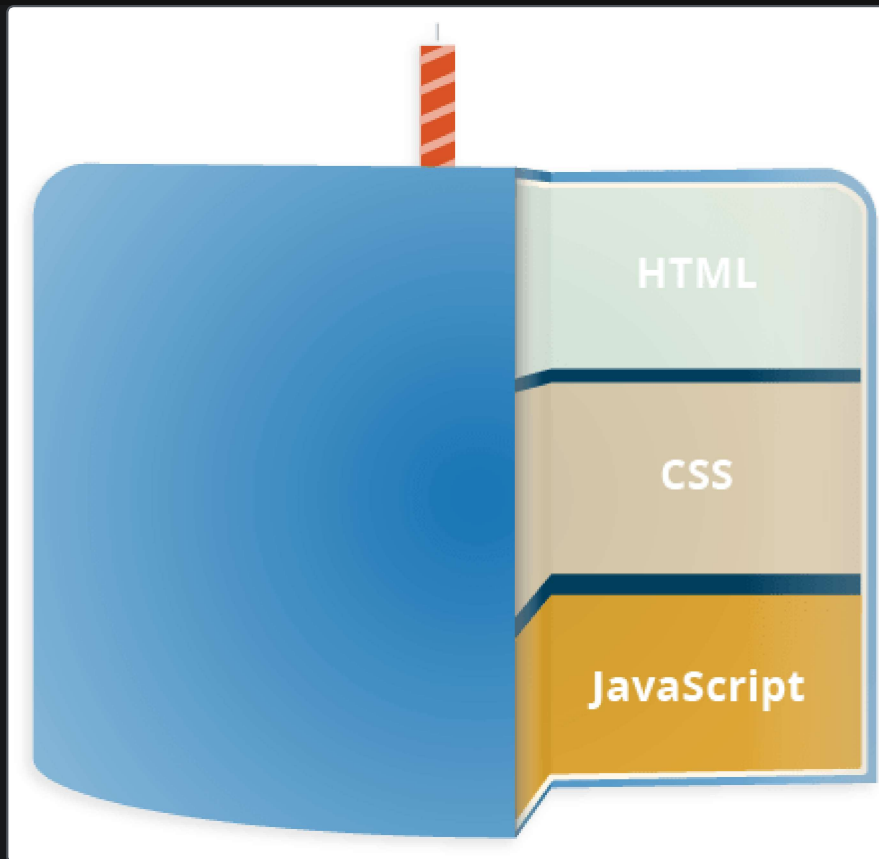
[Próxima](#) →

Sejam bem-vindos ao curso de JavaScript para iniciantes do MDN! Neste primeiro artigo vamos fazer uma análise profunda da linguagem, respondendo questões como "O que é JavaScript?", e "O que ele faz?", para você se sentir confortável com a proposta da linguagem.

Pré requisitos:	Interação básica com o computador, entendimento básico de HTML e CSS.
Objetivo:	Se familiarizar com a linguagem, com o que ela pode fazer, e como tudo isso pode ser utilizado em um website.

## Definição de alto nível

JavaScript é uma linguagem de programação que permite a você implementar itens complexos em páginas web — toda vez que uma página da web faz mais do que simplesmente mostrar a você informação estática — mostrando conteúdo que se atualiza em um intervalo de tempo, mapas interativos ou gráficos 2D/3D animados, etc. — você pode apostar que o JavaScript provavelmente está envolvido. É a terceira camada do bolo das tecnologias padrões da web, duas das quais ([HTML](#) e [CSS](#)) nós falamos com muito mais detalhes em outras partes da Área de Aprendizado.



- **HTML** é a linguagem de marcação que nós usamos para estruturar e dar significado para o nosso conteúdo web. Por exemplo, definindo parágrafos, cabeçalhos, tabelas de conteúdo, ou inserindo imagens e vídeos na página.
- **CSS** é uma linguagem de regras de estilo que nós usamos para aplicar estilo ao nosso conteúdo HTML. Por exemplo, definindo cores de fundo e fontes, e posicionando nosso conteúdo em múltiplas colunas.
- **JavaScript** é uma linguagem de programação que permite a você [criar conteúdo que se atualiza dinamicamente, controlar mídias, imagens animadas, e tudo o mais que há de interessante](#). Ok, não tudo, mas é maravilhoso o que você pode efetuar com algumas linhas de código JavaScript.

As três camadas ficam muito bem uma em cima da outra. Vamos exemplificar com um simples bloco de texto. Nós podemos marcá-lo usando HTML para dar estrutura e propósito:

HTML

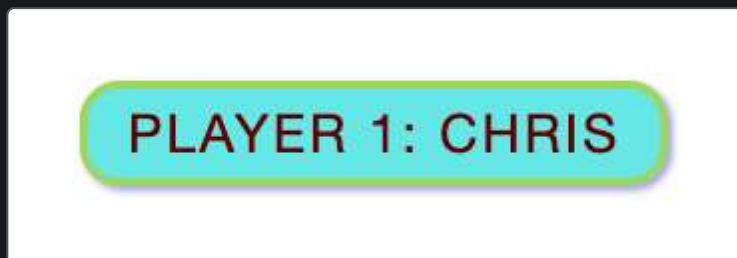
```
<p>Jogador 1: Chris</p>
```

Player 1: Chris

Nós podemos adicionar um pouco de CSS na mistura, para deixar nosso parágrafo um pouco mais atraente:

CSS

```
p {
  font-family: "helvetica neue", helvetica, sans-serif;
  letter-spacing: 1px;
  text-transform: uppercase;
  text-align: center;
  border: 2px solid rgba(0, 0, 200, 0.6);
  background: rgba(0, 0, 200, 0.3);
  color: rgba(0, 0, 200, 0.6);
  box-shadow: 1px 1px 2px rgba(0, 0, 200, 0.4);
  border-radius: 10px;
  padding: 3px 10px;
  display: inline-block;
  cursor: pointer;
}
```



E finalmente, nós podemos adicionar JavaScript para implementar um comportamento dinâmico:

JS

```
const para = document.querySelector("p");

para.addEventListener("click", atualizarNome);

function atualizarNome() {
  var nome = prompt("Insira um novo nome");
  para.textContent = "Jogador 1: " + nome;
}
```

JOGADOR 1: CHRIS

Experimente clicar no botão acima para ver o que acontece (note também que você pode encontrar essa demonstração no GitHub — veja o código fonte [↗](#) ou veja funcionar [↗](#))!

JavaScript pode fazer muito mais do que isso — vamos explorar com mais detalhes.

# Então o que ele pode realmente fazer?

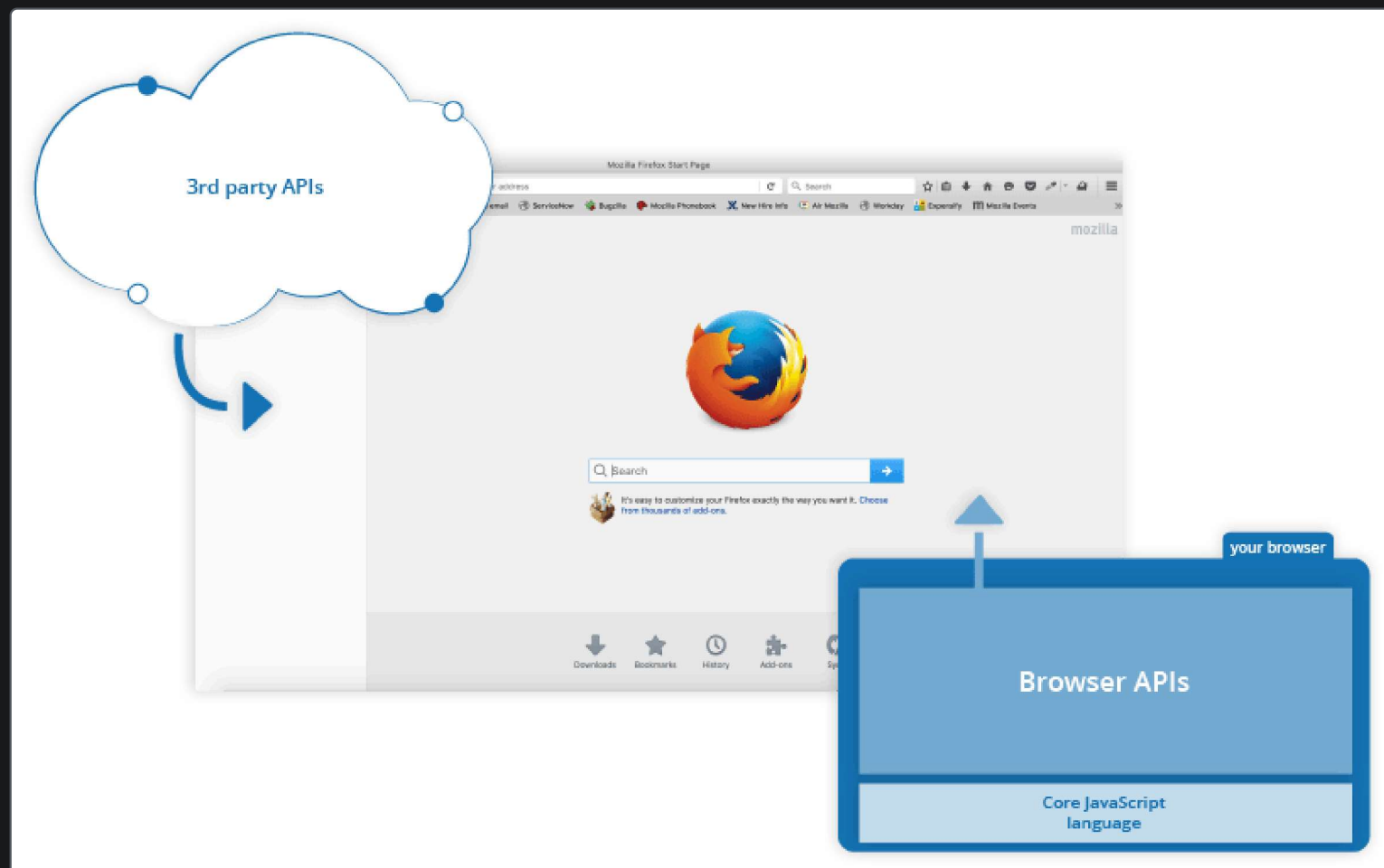
O núcleo da linguagem JavaScript consiste em alguns benefícios comuns da programação que permite a você fazer coisas como:

- Armazenar conteúdo útil em variáveis. No exemplo acima, a propósito, nós pedimos que um novo nome seja inserido e armazenamos o nome em uma variável chamada `nome`.
- Operações com pedaços de texto (conhecidos como "strings" em programação). No exemplo acima, nós pegamos a string "Jogador 1: " e concatenamos (juntamos) com a variável `nome` para criar o texto completo "Jogador 1: Chris".
- Executar o código em resposta a determinados eventos que ocorrem em uma página da Web. Nós usamos o `click` no nosso exemplo acima para que quando clicassem no botão, rodasse o código que atualiza o texto.
- E muito mais!

O que é ainda mais empolgante é a funcionalidade construída no topo do núcleo da linguagem JavaScript. As APIs (Application Programming Interfaces - Interface de Programação de Aplicativos) proveem a você superpoderes extras para usar no seu código JavaScript.

APIs são conjuntos prontos de blocos de construção de código que permitem que um desenvolvedor implemente programas que seriam difíceis ou impossíveis de implementar. Eles fazem o mesmo para a programação que os kits de móveis prontos para a construção de casas - é muito mais fácil pegar os painéis prontos e parafusá-los para formar uma estante de livros do que para elaborar o design, sair e encontrar a madeira, cortar todos os painéis no tamanho e formato certos, encontrar os parafusos de tamanho correto e *depois* montá-los para formar uma estante de livros.

Elas geralmente se dividem em duas categorias.



**APIs de navegadores** já vem implementadas no navegador, e são capazes de expor dados do ambiente do computador, ou fazer coisas complexas e úteis. Por exemplo:

- A API DOM ([Document Object Model](#)) permite a você manipular HTML e CSS, criando, removendo e mudando HTML, aplicando dinamicamente novos estilos para a sua página, etc. Toda vez que você vê uma janela pop-up aparecer em uma página, ou vê algum novo conteúdo sendo exibido (como nós vimos acima na nossa simples demonstração), isso é o DOM em ação.
- A API de Geolocalização recupera informações geográficas. É assim que o [Google Maps](#) consegue encontrar sua localização e colocar em um mapa.
- As APIs Canvas e WebGL permite a você criar gráficos 2D e 3D animados. Há pessoas fazendo algumas coisas fantásticas usando essas tecnologias web — veja [Chrome Experiments](#) e [webgl.samples](#).
- APIs de áudio e vídeo (inglês) como `HTMLMediaElement` (inglês) e WebRTC permitem a você fazer coisas realmente interessantes com multimídia, tanto tocar música e vídeo em uma página da web, como capturar vídeos com a sua câmera e exibir no computador de outra pessoa (veja [Snapshot demo](#) para ter uma ideia).

**Nota:** Muitas demonstrações acima não vão funcionar em navegadores antigos — quando você for experimentar, é uma boa ideia usar browsers modernos como Firefox, Edge ou Opera para ver o código funcionar. Você vai precisar estudar [testes cross browser](#) (inglês) com mais detalhes quando você estiver chegando perto de produzir código (código real que as pessoas vão usar).

**APIs de terceiros** não estão implementados no navegador automaticamente, e você geralmente tem que pegar seu código e informações em algum lugar da Web. Por exemplo:

- A API do [Twitter](#) permite a você fazer coisas como exibir seus últimos tweets no seu website.
- A API do [Google Maps](#) permite a você inserir mapas customizados no seu site e outras diversas funcionalidades.

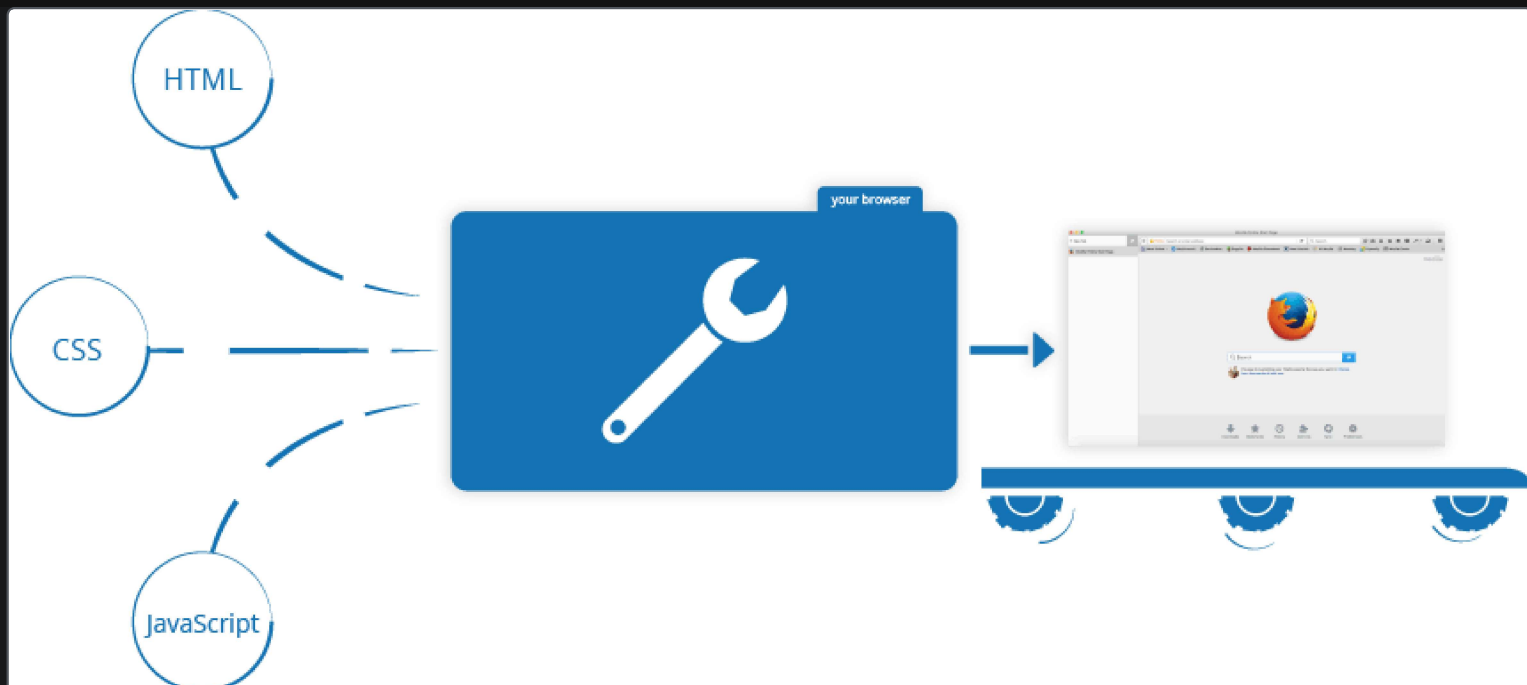
**Nota:** Essas APIs são avançadas e nós não vamos falar sobre nenhuma delas nesse módulo. Você pode achar muito mais sobre elas em nosso módulo [APIs web no lado cliente](#).

Tem muito mais coisas disponíveis! Contudo, não fique animado ainda. Você não estará pronto para desenvolver o próximo Facebook, Google Maps ou Instagram depois de estudar JavaScript por 24 horas — há um monte de coisas básicas para estudar primeiro. E é por isso que você está aqui — vamos começar!

## O que JavaScript está fazendo na sua página web?

Aqui nós vamos realmente começar a ver algum código, e enquanto fazemos isso vamos explorar o que realmente acontece quando você roda algum código JavaScript na sua página.

Vamos recapitular brevemente a história do que acontece quando você carrega uma página web em um navegador (falamos sobre isso no nosso artigo [Como o CSS funciona](#)). Quando você carrega uma página web no seu navegador, você está executando seu código (o HTML, CSS e JavaScript) dentro de um ambiente de execução (a guia do navegador). Isso é como uma fábrica que pega a matéria prima (o código) e transforma em um produto (a página web).



Um uso muito comum do JavaScript é modificar dinamicamente HTML e CSS para atualizar uma interface do usuário, por meio da API do Document Object Model (conforme mencionado acima). Observe que o código em seus documentos web geralmente é carregado e executado na ordem em que aparece na página. Se o JavaScript carregar e tentar executar antes do carregamento do HTML e CSS afetado, poderão ocorrer erros. Você aprenderá maneiras de contornar isso mais adiante neste artigo, na seção [Estratégias de carregamento de scripts](#).

## Segurança do navegador

Cada guia do navegador tem seu próprio espaço para executar código (esses espaços são chamados de "ambientes de execução", em termos técnicos) — isso significa que na maioria dos casos o código em cada guia está sendo executado separadamente, e o código em uma guia não pode afetar diretamente o código de outra guia — ou de outro website. Isso é uma boa medida de segurança — se esse não fosse o caso, então hackers poderiam começar a escrever código para roubar informações de outros websites, e fazer outras coisas más.

**Nota:** Há muitas maneiras de trocar código e conteúdo entre diferentes websites/guias de uma forma segura, mas são técnicas avançadas que não serão estudadas nesse curso.

## Ordem de execução do JavaScript

Quando o navegador encontra um bloco de código JavaScript, ele geralmente [executa na ordem, de cima para baixo](#). Isso significa que você precisa [ter cuidado com a ordem na qual você coloca as coisas](#). Por exemplo, vamos voltar ao bloco JavaScript que nós vimos no primeiro exemplo:

JS

```
const para = document.querySelector("p");

para.addEventListener("click", atualizarNome);

function atualizarNome() {
  let nome = prompt("Informe um novo nome:");
```

```
para.textContent = "Jogador 1: " + nome;
}
```

Aqui nós estamos selecionando um parágrafo (linha 1) e anexando a ele um *event listener* (linha 3). Então, quando o parágrafo recebe um clique, o bloco de código `atualizarNome()` (linhas 5 a 8) é executado. O bloco de código `atualizarNome()` (esses tipos de bloco de código reutilizáveis são chamados "funções") pede ao usuário que informe um novo nome, e então insere esse nome no parágrafo, atualizando-o.

Se você inverte a ordem das duas primeiras linhas de código, ele não funcionaria — em vez disso, você receberia um erro no console do navegador — `TypeError: para is undefined`. Isso significa que o objeto `para` não existe ainda, então nós não podemos adicionar um *event listener* a ele.

❗ **Nota:** Esse é um erro muito comum — você precisa verificar se os objetos aos quais você se refere no seu código existem antes de você tentar anexar coisas a eles.

## Código interpretado x compilado

Você pode ouvir os termos **interpretado** e **compilado** no contexto da programação. JavaScript é uma [linguagem interpretada](#) — o código é executado de cima para baixo e o resultado da execução do código é imediatamente retornado. Você não tem que transformar o código em algo diferente antes do navegador executa-lo.

[Linguagens compiladas](#), por outro lado, [são transformadas \(compiladas\) em algo diferente antes que sejam executadas pelo computador](#). Por exemplo, C/C++ são compiladas em linguagem Assembly, e depois são executadas pelo computador.

JavaScript é uma linguagem de programação leve e interpretada. O navegador recebe o código JavaScript em sua forma de texto original e executa o script a partir dele. Do ponto de vista técnico, a maioria dos intérpretes modernos de JavaScript realmente usa uma técnica chamada **compilação just-in-time** para melhorar o desempenho; o código-fonte JavaScript é compilado em um formato binário mais rápido enquanto o script está sendo usado, para que possa ser executado o mais rápido possível. No entanto, o JavaScript ainda é considerado uma linguagem interpretada, pois a compilação é manipulada em tempo de execução, e não antes.

Há vantagens em ambos os tipos de linguagem, mas nós não iremos discutir no momento.

## Lado do servidor x Lado do cliente

Você pode também ouvir os termos **lado do servidor** (*server-side*) e **lado do cliente** (*client-side*), especialmente no contexto de desenvolvimento web. [Códigos do lado do cliente são executados no computador do usuário](#) — quando uma página web é visualizada, o código do lado do cliente é baixado, executado e exibido pelo navegador. Nesse módulo JavaScript nós estamos explicitamente falando sobre **JavaScript do lado do cliente**.

Códigos do lado do servidor, por outro lado, [são executados no servidor e o resultado da execução é baixado e exibido no navegador](#). Exemplos de linguagens do lado do servidor populares incluem PHP, Python, Ruby, e ASP.NET. E JavaScript! JavaScript também pode ser usada como uma linguagem *server-side*, por exemplo, no popular ambiente Node.js — você pode encontrar mais sobre JavaScript do lado do servidor no nosso tópico [Websites dinâmicos - Programação do lado do servidor](#).

## Código dinâmico x estático

A palavra **dinâmico** é usada para descrever tanto o JavaScript *client-side* como o *server-side* — essa palavra se refere a habilidade de atualizar a exibição de uma página web/app para mostrar coisas diferentes em circunstâncias diferentes, gerando novo conteúdo como solicitado. Código do lado do servidor dinamicamente gera novo conteúdo no servidor, puxando dados de um banco de dados, enquanto que JavaScript do lado do cliente dinamicamente gera novo conteúdo dentro do navegador do cliente, como criar uma nova tabela HTML com dados recebidos do servidor e mostrar a tabela em uma página web exibida para o usuário. Os significados são



ligeiramente diferente nos dois contextos, porém relacionados, e ambos (JavaScript *server-side* e *client-side*) geralmente trabalham juntos.

Uma página web sem atualizações dinâmicas é chamada de **estática** — ela só mostra o mesmo conteúdo o tempo todo.

# Como você adiciona JavaScript na sua página?

O JavaScript é inserido na sua página de uma maneira similar ao CSS. Enquanto o CSS usa o elemento `<link>` para aplicar folhas de estilo externas e o elemento `<style>` para aplicar folhas de estilo internas, o JavaScript só precisa de um amigo no mundo do HTML — o elemento `<script>`. Vamos aprender como funciona.

## JavaScript interno

1. Antes de tudo, faça uma cópia local do nosso arquivo de exemplo [aplicando-javascript.html](#). Salve-o em alguma pasta, de uma forma sensata.
2. Abra o arquivo no seu navegador web e no seu editor de texto. Você verá que o HTML cria uma simples página web contendo um botão clicável.
3. Agora, vá até o seu editor de texto e adicione o código a seguir antes da tag de fechamento `</body>`:

HTML

```
<script>
  // O JavaScript fica aqui
</script>
```

4. Agora nós vamos adicionar um pouco de JavaScript dentro do nosso elemento `<script>` para que a página faça algo mais interessante — adicione o seguinte código abaixo da linha `// O JavaScript fica aqui`:

JS

```
function criarParagrafo() {
  let para = document.createElement("p");
  para.textContent = "Você clicou no botão!";
  document.body.appendChild(para);
}

const botoes = document.querySelectorAll("button");

for (var i = 0; i < botoes.length; i++) {
  botoes[i].addEventListener("click", criarParagrafo);
}
```

5. Salve seu arquivo e recarregue a página — agora você deveria ver que quando você clique no botão, um novo parágrafo é gerado e colocado logo abaixo.



- ⓘ **Nota:** Se seu exemplo não parece funcionar, leia cada passo novamente e confira que você fez tudo certo. Você salvou sua cópia local do código inicial como um arquivo .html? Você adicionou o elemento `<script>` imediatamente antes da tag `</body>`? Você digitou o código JavaScript exatamente como ele está sendo mostrado? **JavaScript é uma linguagem case sensitive (isso significa que a linguagem vê diferença entre letras maiúsculas e minúsculas) e muito confusa, então você precisa digitar a sintaxe exatamente como foi mostrada, senão não vai funcionar.**

- ⓘ **Nota:** Você pode ver essa versão no GitHub como [aplicando-javascript-interno.html](#) (veja funcionar também).

## JavaScript externo

Isso funciona muito bem, mas e se nós quiséssemos colocar nosso JavaScript em um arquivo externo? Vamos explorar isso agora.

1. Primeiro, crie um novo arquivo na mesma pasta que está o arquivo HTML de exemplo. Chame-o de `script.js` — tenha certeza de que o nome do arquivo tem a extensão `.js`, pois é assim que ele será reconhecido como JavaScript.
2. Agora substitua o elemento atual `<script>` pelo seguinte código:

HTML

```
<script src="script.js" defer></script>
```

3. Em `script.js`, adicione o seguinte script:

JS

```
function createParagraph() {  
  let para = document.createElement("p");  
  para.textContent = "Você clicou no botão!";  
  document.body.appendChild(para);  
}  
  
const buttons = document.querySelectorAll("button");  
  
for (let i = 0; i < buttons.length; i++) {  
  buttons[i].addEventListener("click", createParagraph);  
}
```

4. Salve e atualize seu navegador, e você deverá ver a mesma coisa! Funciona igualmente, mas agora nós temos o JavaScript em um arquivo externo. Isso é **geralmente uma coisa boa em termos de organização de código, e faz com que seja possível reutilizar o código em múltiplos arquivos HTML**. Além disso, o HTML fica mais legível sem grandes pedaços de script no meio dele.

- ⓘ **Nota:** Você pode ver essa versão no GitHub como [aplicando-javascript-externo.html](#) e [script.js](#) (veja funcionar também).

# Manipuladores de JavaScript inline

Note que às vezes você vai encontrar código JavaScript escrito dentro do HTML. Isso deve ser algo como:

JS

```
function criarParagrafo() {  
  let para = document.createElement("p");  
  para.textContent = "Você clicou o botao!";  
  document.body.appendChild(para);  
}
```



HTML

```
<button onclick="criarParagrafo()">Me clique!</button>
```



Você pode tentar essa versão na nossa demonstração abaixo.

Me clique!

Essa demonstração tem exatamente a mesma funcionalidade que vimos nas primeiras duas seções, exceto que o elemento `<button>` inclui um manipulador *inline* `onclick` para fazer a função ser executada quando o botão é clicado.

**Contudo, por favor, não faça isso.** É uma má prática poluir seu HTML com JavaScript, e isso é ineficiente — você teria que incluir o atributo `onclick="criarParagrafo()"` em todo botão que você quisesse aplicar JavaScript.

Usando uma estrutura feita de puro JavaScript permite a você selecionar todos os botões usando uma instrução. O código que nós usamos acima para servir a esse propósito se parece com isso:

JS

```
const botoes = document.querySelectorAll("button");  
  
for (var i = 0; i < botoes.length; i++) {  
  botoes[i].addEventListener("click", criarParagrafo);  
}
```

Isso talvez parece ser mais do que o atributo `onclick`, mas isso vai funcionar para todos os botões, não importa quantos tem na página, e quantos forem adicionados ou removidos. O JavaScript não precisará ser mudado.

❗ **Nota:** Tente editar a sua versão do arquivo `aplicar-javascript.html`, adicionando alguns botões a mais. Quando você recarregar, você deverá ver que todos os botões, quando clicados, irão criar parágrafos. Agradável, não?

## Estratégias para o carregamento de scripts

Há um considerável número de problemas envolvendo o carregamento de scripts na ordem correta. Infelizmente, nada é tão simples quanto parece ser! Um problema comum é que todo o HTML de uma página é carregado na ordem em que ele aparece. Se você estiver usando Javascript para manipular alguns elementos da página (sendo mais preciso, manipular o [Document Object Model](#)), seu código não irá funcionar caso o JavaScript for carregado e executado antes mesmo dos elementos HTML estarem disponíveis.

Nos exemplos acima, tanto nos scripts internos ou externos, o JavaScript é carregado e acionado dentro do cabeçalho do documento, antes do corpo da página ser completamente carregado. Isso poderá causar algum erro. Assim, temos algumas soluções para isso.

No exemplo interno, você pode ver essa estrutura em volta do código:

JS

```
document.addEventListener("DOMContentLoaded", function() {  
    ...  
});
```

Isso é um *event listener* (ouvidor de eventos)\*, que ouve e aguarda o disparo do evento "DOMContentLoaded" vindo do *browser*, evento este que significa que o corpo do HTML está completamente carregado e pronto. O código JavaScript que estiver dentro desse bloco não será executado até que o evento seja disparado, portanto, o erro será evitado (você irá [aprender sobre eventos](#) mais tarde).

No exemplo externo, nós usamos um recurso moderno do JavaScript para resolver esse problema: Trata-se do atributo `defer`, que informa ao *browser* para continuar renderizando o conteúdo HTML uma vez que a tag `<script>` foi atingida.

JS

```
<script src="script.js" defer></script>
```

Neste caso, ambos script e HTML irão carregar de forma simultânea e o código irá funcionar.

❗ **Nota:** No caso externo, nós não precisamos utilizar o evento `DOMContentLoaded` porque o atributo `defer` resolve o nosso problema. Nós não utilizamos `defer` como solução para os exemplos internos pois `defer` funciona apenas com scripts externos.

Uma solução à moda antiga para esse problema era colocar o elemento script bem no final do body da página (antes da tag `</body>`). Com isso, os scripts iriam carregar logo após todo o conteúdo HTML. O problema com esse tipo de solução é que o carregamento/renderização do script seria completamente bloqueado até que todo o conteúdo HTML fosse analisado. Em sites de maior escala, com muitos scripts, essa solução causaria um grande problema de performance e deixaria o site lento.

## ASYNC E DEFER

Atualmente, há dois recursos bem modernos que poderemos usar para evitar o problema com o bloqueio de scripts — `async` e `defer` (que vimos acima). Vamos ver as diferenças entre esses dois?

Os scripts que são carregados usando o atributo `async` (veja abaixo) irão baixar o script sem bloquear a renderização da página e irão executar imediatamente após o script terminar de ser disponibilizado. Nesse modo você não tem garantia nenhuma que os scripts carregados irão rodar em uma ordem específica, mas saberá que dessa forma eles não irão impedir o carregamento do restante da página. O melhor uso para o `async` é quando os scripts de uma página rodam de forma independente entre si e também não dependem de nenhum outro script.

Por exemplo, se você tiver os seguintes elementos script:

HTML

```
<script async src="js/vendor/jquery.js"></script>

<script async src="js/script2.js"></script>

<script async src="js/script3.js"></script>
```

Você não pode garantir que o script. `jquery.js` carregará antes ou depois do `script2.js` e `script3.js` . Nesse caso, se alguma função desses scripts dependerem de algo vindo do `jquery` , ela produzirá um erro pois o `jquery` ainda não foi definido/carregado quando os scripts executaram essa função.

`async` deve ser usado quando houver muitos scripts rodando no *background*, e você precisa que estejam disponíveis o mais rápido possível. Por exemplo, talvez você tenha muitos arquivos de dados de um jogo para carregar que serão necessários assim que o jogo iniciar, mas por enquanto, você só quer entrar e ver a tela de carregamento, a do título do jogo e o *lobby*, sem ser bloqueado pelo carregamento desses scripts.

Scripts que são carregados utilizando o atributo `defer` (veja abaixo) irão rodar exatamente na ordem em que aparecem na página e serão executados assim que o script e o conteúdo for baixado.

HTML

```
<script defer src="js/vendor/jquery.js"></script>

<script defer src="js/script2.js"></script>

<script defer src="js/script3.js"></script>
```

Todos os scripts com o atributo `defer` irão carregar na ordem que aparecem na página. No segundo exemplo, podemos ter a certeza que o script `jquery.js` irá carregar antes do `script2.js` e `script3.js` e o `script2.js` irá carregar antes do `script3.js` . Os scripts não irão rodar sem que antes todo o conteúdo da página seja carregado, que no caso, é muito útil se os seus scripts dependem de um DOM completamente disponibilizado em tela (por exemplo, scripts que modificam um elemento).

Resumindo:

- `async` e `defer` instruem o *browser* a baixar os scripts numa *thread* (processo) à parte, enquanto o resto da página (o DOM, etc.) está sendo baixado e disponibilizado de forma não bloqueante.
- Se os seus scripts precisam rodar imediatamente, sem que dependam de outros para serem executados, use `async` .
- Se seus scripts dependem de outros scripts ou do DOM completamente disponível em tela, carregue-os usando `defer` e coloque os elementos `<script>` na ordem exata que deseja que sejam carregados.

# Comentários

Assim como HTML e CSS, é possível escrever comentários dentro do seu código JavaScript que serão ignorados pelo navegador, e existirão simplesmente para prover instruções aos seus colegas desenvolvedores sobre como o código funciona (e pra você, se você tiver que voltar ao seu código depois de 6 meses e não se lembrar do que fez). Comentários são muito úteis, e você deveria usá-los frequentemente, principalmente quando seus códigos forem muito grandes. Há dois tipos:

- Um comentário de uma linha é escrito depois de duas barras. Por exemplo:

JS

```
// Eu sou um comentário
```

- Um comentário de múltiplas linhas é escrito entre os caracteres `/*` e `*/`. Por exemplo:

JS

```
/*  
    Eu também sou  
    um comentário  
*/
```

Então, por exemplo, você poderia fazer anotações na nossa última demonstração de código JavaScript, da seguinte forma:

JS

```
// Função: Cria um novo parágrafo e o insere no fim do arquivo HTML.  
  
function criarParagrafo() {  
    var para = document.createElement("p");  
    para.textContent = "Você clicou no botão!";  
    document.body.appendChild(para);  
}  
  
/*  
    1. Captura referências de todos os botões na página e armazena isso em um array.  
    2. Vai até todos os botões e adiciona um event listener click a cada um deles.  
  
    Quando cada botão é clicado, a função criarParagrafo() será executada.  
*/  
  
const botoes = document.querySelectorAll("button");  
  
for (var i = 0; i < botoes.length; i++) {  
    botoes[i].addEventListener("click", criarParagrafo);  
}
```