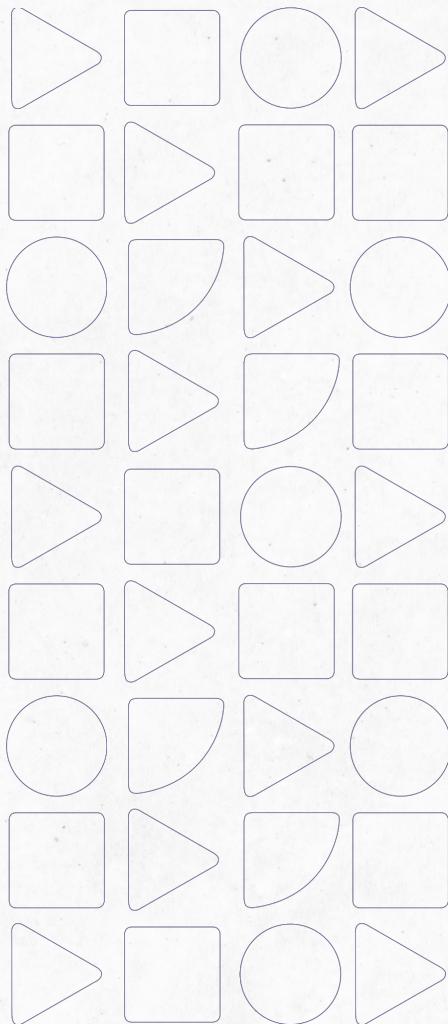


# Funções

**Disciplina:** Linguagem de Programação



## Conteúdos:

Funções.

## Habilidade(s):

- Conhecer e entender a estrutura modular da função e suas aplicações;
- Definir funções usando a palavra-chave *function* e a chamá-las com argumentos;
- Resolução de problemas para dividir problemas complexos em partes menores.

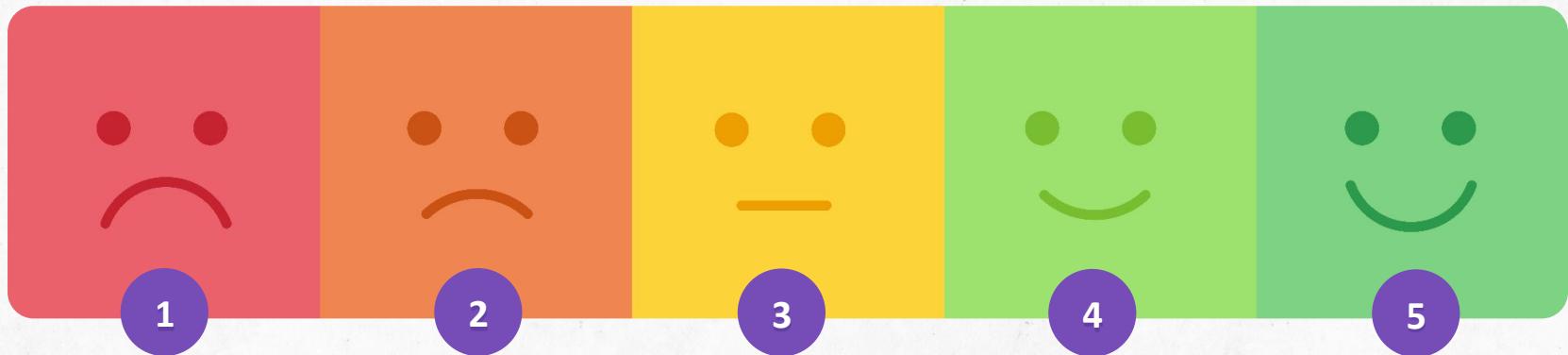
# Bloco 1

---

Entendendo o que são *arrays* (vetores).

## Quem é você?

Na hora de seguir um **passo a passo**, como você se sente? Em uma escala de 1 a 5, como lida com a situação?





**Quando se fala de instruções, cada pessoa  
lida de forma diferente. Na área da  
computação, as instruções são ótimas aliadas  
na organização correta de um código.**

# A importância da funções

As funções são um conjunto de instruções que executam uma determinada tarefa, devolvendo um resultado como retorno.

Isso **economiza tempo e esforço**, pois evita a necessidade de escrever o mesmo código diversas vezes.



# Sintaxe das funções

```
function nomeDaFuncao(parametro1, parametro2, ...) {  
    return valorDeRetorno;  
}
```

**function:** a palavra-chave *function* é usada para declarar uma função em JavaScript;

**nomeDaFuncao:** deve ser um identificador válido e seguir as regras de nomenclatura de variáveis em JavaScript;

**return valorDeRetorno:** é usada para especificar o valor que a função deve retornar quando terminar de executar.

# Vamos analisar o exemplo

Imagine que você está fazendo um banquete para a sua família e precisa ferver a água diversas vezes. Para fazer isso, você segue uma série de instruções.

## Sem as funções

```
// Passo 1: Encha a chaleira com água  
console.log("Passo 1: Encha a chaleira com água.");  
// ... código para encher a chaleira ...  
  
// Passo 2: Ligue a chaleira  
console.log("Passo 2: Ligue a chaleira.");  
// ... código para ligar a chaleira ...  
  
// Passo 3: Espere a água ferver  
console.log("Passo 3: Espere a água ferver.");  
// ... código para esperar a água ferver ...
```



Nessa abordagem, você realiza cada passo manualmente e com um código sequencial. Se você precisar ferver água novamente para outra receita, terá que repetir todos esses passos, copiando e colando o código, o que o torna propenso a erros.

# Vamos analisar o exemplo

Aqui, como as funções seriam utilizadas?

Com as funções

```
ferverAgua();
```



Nessa abordagem, você encapsulou todo o processo de ferver água em uma função chamada **ferverAgua**.

Agora, sempre que precisar ferver água, basta chamar essa função!

## Função anônima

É uma função que não possui um nome identificador associado a ela. Em vez de definir uma função com um nome específico usando a palavra-chave ***function***, você cria a função diretamente como uma expressão.

São úteis quando se precisa uma função temporária em um contexto específico e não se deseja poluir o escopo com um nome de função que não será reutilizado em outros lugares do código.



# Explorando as funções anônimas

Observe a função anônima que multiplica dois números.

```
const multiplicar = function(x, y) {  
    return x * y;  
};
```

Nesse exemplo, a função anônima é atribuída a uma variável (`multiplicar`).

# Bloco 2

---

Explorando a função de flecha (*arrow*).

# A função das profissões

Escreva em um *post-it* uma profissão de sua escolha e, ao finalizar, entregue ao professor.

O professor reunirá todos os *post-its*, sorteando um para cada aluno. Ao receber o seu, cada estudante deverá escrever funções específicas para a profissão sorteada em um código JavaScript.

Veja o exemplo:

Psicóloga

```
function ouvirPaciente(paciente) {  
    console.log(nomePsicologa + " está ouvindo o paciente " + paciente + ".");  
  
    var conselhosParaJoao = ["Pratique a autocompaixão.", "Tente manter uma  
    rotina diária saudável.", "Considere a terapia cognitivo-comportamental."];  
    aconselharPaciente("João", conselhosParaJoao)
```

# Função de flecha (*arrow function*)

Já imaginou usar muitas funções em um código?

A **função de flecha** é especialmente útil para os programas que recebem muitas funções, pois, em comparação com o modelo tradicional, é uma forma mais **concisa** de escrevê-las.



Curiosidade: elas são chamadas “funções de flecha” devido à sintaxe que usa uma seta =>.



## Observando a atuação da *arrow function*

Analise como esse código poderia ser representado em seu modo tradicional, utilizando a *arrow function* na soma de dois números.

```
function somar(a, b) {  
    return a + b;  
}  
  
console.log(somar(5, 3)); // Saída: 8
```

```
const somar = (a, b) => a + b;  
  
console.log(somar(5, 3)); // Saída: 8
```

O que mudou?

# Gabarito

E aí, acertou?

```
function somar(a, b) {  
    return a + b;  
}  
  
console.log(somar(5, 3)); // Saída: 8
```

```
const somar = (a, b) => a + b;  
  
console.log(somar(5, 3)); // Saída: 8
```

Observe que, na função de flecha, podemos omitir as chaves {} e a palavra-chave *return*, porque há apenas uma expressão de retorno. Já na função normal, usamos a palavra-chave *return* para explicitamente retornar o valor calculado.

# Bloco 3

---

Explorando a função construtora.

# Você consegue diferenciar?

Pense em um objeto aleatório, como um **carro**.

A partir disso, você saberia dizer o que define as suas **propriedades** e os seus **métodos**?



VS



# Gabarito

Propriedades

Motor

Câmbio

Rodas

VS

Métodos

Ligar motor

Acender faróis

Acelerar a velocidade

# Como isso se relaciona com a criação de um código?

JavaScript é uma linguagem orientada a **objetos** e semelhante a um objeto da vida real. Ela possui propriedades que são como características do objeto, assim como métodos que são como ações que o objeto pode executar.

As propriedades são semelhantes às variáveis vinculadas ao objeto e podem armazenar **valores**.



Você pode acessar as propriedades de um objeto usando a notação de ponto, como `nomeObjeto.nomePropriedade`



# Exemplo prático de criação de um objeto

Vamos criar um carro fictício em um código:

```
var carro = {  
    marca: "Volkswagen",  
    modelo: "Kombi",  
    cor: "Amarela",  
    ano: 1975}
```

Podemos acessar ou alterar as propriedades de um objeto em JavaScript usando [] (colchetes):

```
carro ["marca"] = "Golf"
```



# Função construtora

A **função construtora** permite a criação de múltiplas instâncias de objetos com a mesma estrutura e comportamento.

Observe como ela seria utilizada no exemplo do carro:

```
function Carro(marca, modelo, cor, ano) {  
    this.marca = marca;  
    this.modelo = modelo;  
    this.cor = cor;  
    this.ano = ano;
```



# Criando um objeto com uma função construtora

Agora que temos a função construtora definida, é hora de criar o objeto **carro**:

```
var meuCarro = new Carro("Volkswagen", "Fusca",  
"Vermelho", 1977)
```



# Bloco 4

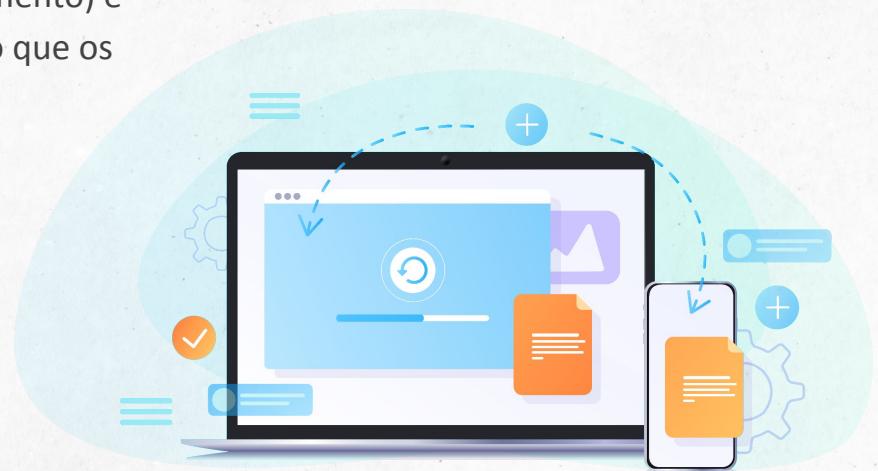
---

Descobrindo o *Document Object Model* (DOM).

# **Document Object Model (DOM)**

Imagine que uma página da *web* é como uma árvore de elementos. Esses elementos podem ser muitas coisas, como títulos, parágrafos, imagens, botões e mais.

O DOM (em tradução livre, Modelo de Objetos de Documento) é uma maneira de representar essa árvore em um formato que os computadores sejam capazes de entender.



# Árvore de elementos

Pense em páginas da *web* como árvores de Natal.

Cada enfeite, luz e galho na árvore é como um elemento em uma página da *web*. Agora, imagine que você está de pé ao lado da árvore e pode ver todos esses enfeites e luzes.

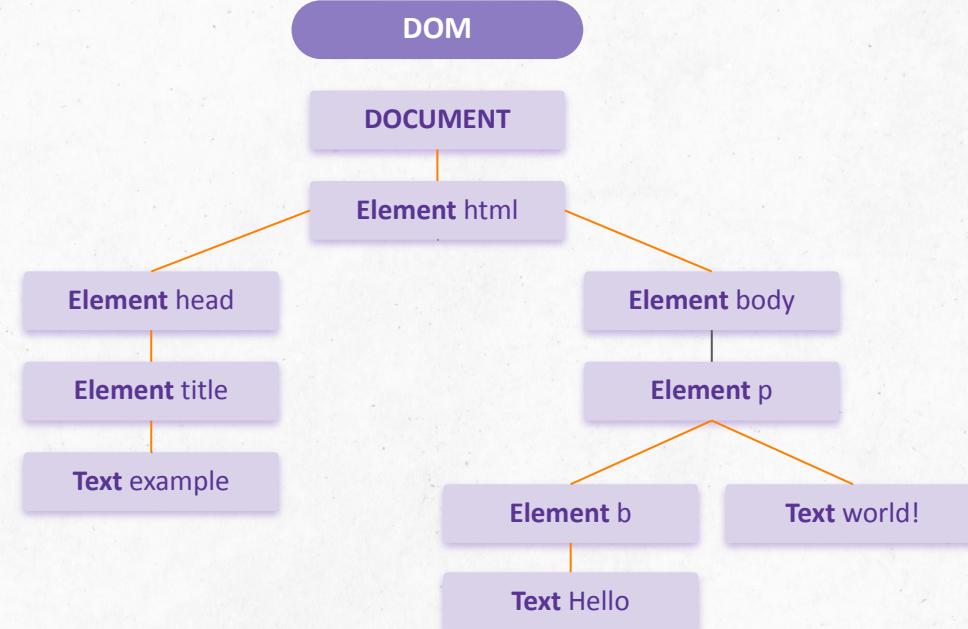
- Você pode tocar em um enfeite específico e movê-lo;
- Você pode ligar e desligar luzes individuais;
- Você pode até adicionar ou remover enfeites conforme desejar.



# Exemplo de um DOM

Página HTML

```
<!DOCTYPE html>
<html>
  <head>
    <title>Example</title>
  </head>
  <body>
    <p><b>Hello</b> world!</p>
  </body>
</html>
```



# Funcionamento do DOM

1

Quando você carrega uma página da *web* no seu navegador, o navegador lê o código HTML que compõe a página;

2

Ele, então, cria uma estrutura em forma de árvore com base nesse código HTML. Cada elemento HTML se torna um “nó” nessa árvore;

3

Ele inclui todos os elementos da página, como cabeçalhos, parágrafos, *links*, imagens e muito mais;

4

Com o DOM, você pode encontrar elementos específicos, alterar o conteúdo de um parágrafo, mover elementos pela página ou até mesmo adicionar novos elementos.

# Manipulação do DOM

Envolve a interação com esses nós e utiliza JavaScript para realizar ações, como:

**Acessar elementos** – usando seletores (como getElementById, querySelector, getElementsByClassName etc.), você pode acessar elementos específicos da página;

**Alterar conteúdo** – você pode alterar o texto, atributos e conteúdo HTML dos elementos;

**Manipular estilos** – é possível modificar as propriedades de estilo (como cor, tamanho, margens etc.) dos elementos;

**Adicionar e remover elementos** – você pode adicionar novos elementos à página ou remover elementos existentes;

**Responder a eventos** – você pode definir manipuladores de eventos para executar ações quando eventos, como cliques ou submissões de formulários, ocorrerem.

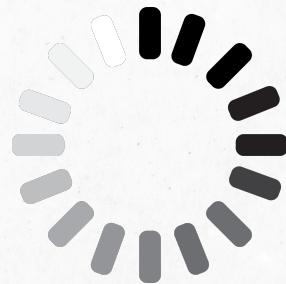
# Bloco 5

---

Explorando os eventos e *closures* de páginas web.

## Avaliação do humor

De 1 a 6, o quanto a imagem a seguir lhe causa ansiedade?



O ícone de carregamento pode ser motivo de grande estresse para pessoas que desejam acessar uma página web. Do ponto de vista da programação, ele está relacionado a um elemento que estudaremos mais profundamente: os eventos e *closures*.

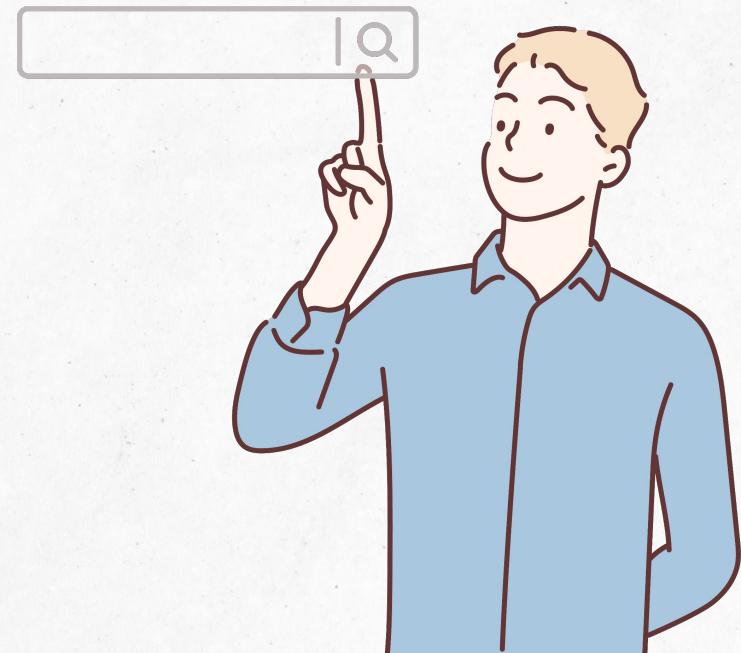


# Eventos

Um evento é um conjunto de ações que são realizadas dentro de um determinado elemento de uma página.

Quando pressionamos uma tecla sobre um elemento, por exemplo, é disparado o evento desse elemento, chamado “**keydown**”.

Quando um usuário clica em um botão, é disparado um evento deste elemento, chamado “**click**”.



# Analisando um exemplo de eventos

Aqui, utilizamos o onEvent id “click” e onEvent id “input”.

```
1  onEvent("text_input1", "input", function() {
2      setText("label2", "o número escolhido foi " + getText("text_input1"));
3  });
4  onEvent("buttonVerificar", "click", function() {
5      if (getText("text_input1") % 2 == 0) {
6          setText("labelresposta", "O número escolhido é Par");
7      } else {
8          setText("labelresposta", "O número escolhido é Ímpar");
9      }
10 });

```

# Principais eventos em JavaScript

onBlur	Remove o foco do elemento.
onChange	Muda o valor do elemento.
onClick	O elemento é clicado pelo usuário.
onFocus	O elemento é focado.
onKeyPress	O usuário pressiona uma tecla sobre o elemento.
onLoad	Carrega o elemento por completo.
onMouseOver	Define ação quando o usuário passa o <i>mouse</i> sobre o elemento.
onMouseOut	Define ação quando o usuário retira o <i>mouse</i> sobre o elemento.

## onChange

São responsáveis pelas ações que o usuário realiza com o *mouse* quando ele está em cima do elemento e quando está fora, respectivamente.

Um exemplo é um código no qual temos um elemento <input> de tipo "color" que permite ao usuário selecionar uma cor. Quando a cor é selecionada, o evento "onchange" é acionado e a função mudarCor() é chamada. A função obtém o valor da cor selecionada e muda a cor do parágrafo com o id "mensagem" para a cor selecionada.



## onMouseOver e onMouseOut

É usado para que determinada ação seja realizada após alguma mudança no documento.

Imagine um código que, quando o *mouse* passa sobre essa *div* (evento *onMouseOver*), a função *destacarElemento()* é chamada e altera a cor de fundo da *div* para "*lightgreen*", destacando-a. Quando o mouse sai da *div* (evento *onMouseOut*), a função *restaurarElemento()* é chamada e restaura a cor de fundo original para "*lightblue*".



# onMouseDown, onMouseUp e onClick

Com a combinação desses eventos, podemos trabalhar com o clique do *mouse* e quando soltar o clique.

Imagine uma página web com o id "meuBotao". Quando você interage com esse botão usando o *mouse*, acontecem alguns eventos.

**onMouseDown:** esse evento é acionado quando o botão do *mouse* é pressionado enquanto o cursor está sobre o botão;

**onMouseUp:** esse evento é acionado quando o botão do *mouse* é liberado após ser pressionado enquanto o cursor está sobre o botão;

**onClick:** esse evento é acionado quando o botão do *mouse* é pressionado e liberado enquanto o cursor está sobre o botão.

## Closures

Ocorre quando uma função é definida dentro de outra função e mantém acesso ao escopo da função externa, mesmo após a função externa ter sido executada e o seu escopo ter sido encerrado.

```
function criarSaudacao(saudacao) {  
    return function(nome) {  
        console.log(saudacao + " " + nome + "!");  
    };  
}  
  
const saudacaoEmPortugues = criarSaudacao("Olá");  
const saudacaoEmIngles = criarSaudacao("Hello");  
  
saudacaoEmPortugues("Maria"); // Exibe "Olá Maria!"  
saudacaoEmIngles("John"); // Exibe "Hello John!"
```



## Analisando o exemplo anterior

No exemplo anterior, a função **criarSaudacao** recebe um argumento **saudacao** e retorna uma função interna. Essa função interna é um *closure* que pode usar a saudação específica fornecida quando foi criada.

Depois, são criadas duas instâncias da *closure*, uma para saudações em português e outra para saudações em inglês. Quando chamamos essas funções com nomes diferentes, elas combinam a saudação com o nome e exibem uma mensagem de saudação personalizada.



# Características das *closures*

**Função interna:** a função que é definida dentro de outra função é chamada de função interna ou função fechada (*closure function*).

**Função externa:** a função que envolve a função interna é chamada de função externa ou função envolvente (*outer function*).

**Escopo:** a função interna tem acesso ao escopo da função externa, incluindo as variáveis, parâmetros e outros elementos definidos neste escopo.

**Retenção de escopo:** uma *closure* retém o escopo da função externa mesmo após a função externa ter terminado a execução.



## Alteração de escopo

Se refere à capacidade de uma função interna (função aninhada) acessar e manipular as variáveis definidas no escopo de sua função externa (função envolvente), mesmo após a função externa ter terminado.

Vamos observar:

Ao usar *closures*, você pode criar funções que operam com dados específicos, sem expor esses dados a todo o escopo global.

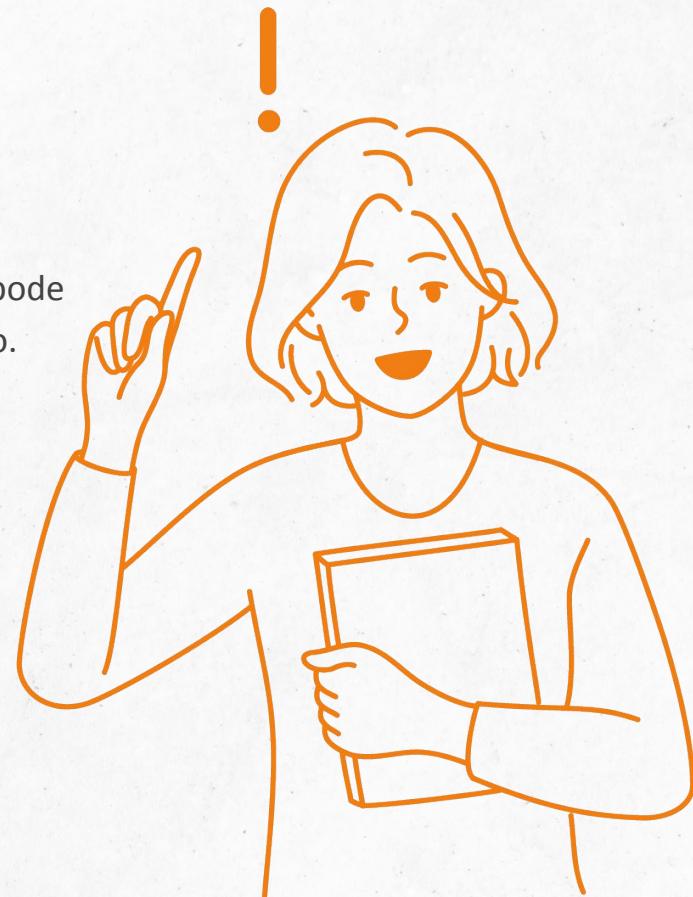
# Hora de analisar

Observe o exemplo:

```
1 function externa() { }
2     var x = 10;
3
4     function interna() {
5         console.log(x); // A função interna acessa a variável 'x' da função externa
6     }
7
8
9
10    return interna;
11}
12 var minhaFuncao = externa();
13 minhaFuncao(); // Irá imprimir 10
14
```

## Gabarito

No exemplo anterior, a função interna é retornada como resultado da função externa. Embora a função externa tenha terminado a sua execução e a variável `x` esteja fora de escopo, a função interna ainda pode acessar a variável `x`, porque ela mantém uma referência a esse escopo.



# Bloco 6

---

Hora de praticar!

## Rotações por estações

Vamos realizar um exercício um pouco diferente?

Nesta aula, haverá três espaços, cada um com uma atividade distinta. Os alunos devem separar-se em três grupos para realizar o desafio de cada estação e, ao final do tempo, deverão inverter a ordem, de modo que, ao final, cada aluno tenha passado pelos três espaços.



# Fechamento

O que eu já sabia?

O que eu não sabia?

O que quero aprender?

# Referências Bibliográficas

PROZ EDUCAÇÃO. *Apostila de Desenvolvimento para Dispositivos Móveis I*, 2023.

CN BLOGS. *DOM遍历*. CN Blogs, 10 fev. 2017. Disponível em:

<https://www.cnblogs.com/xiaohuochai/p/6387570.html>. Acesso em: 7 out. 2023.