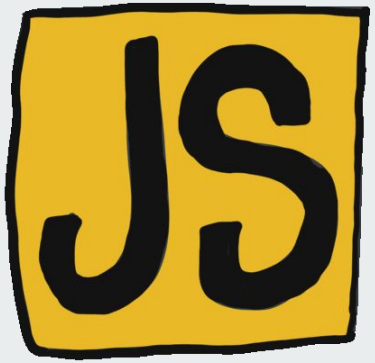
INTRODUÇÃO AO JAVASCRIPT



Introdução ao Desenvolvimento Web - Técnico Integrado em Redes de Computadores

Profº: Me. Lucas Ferreira Mendes

ROTEIRO

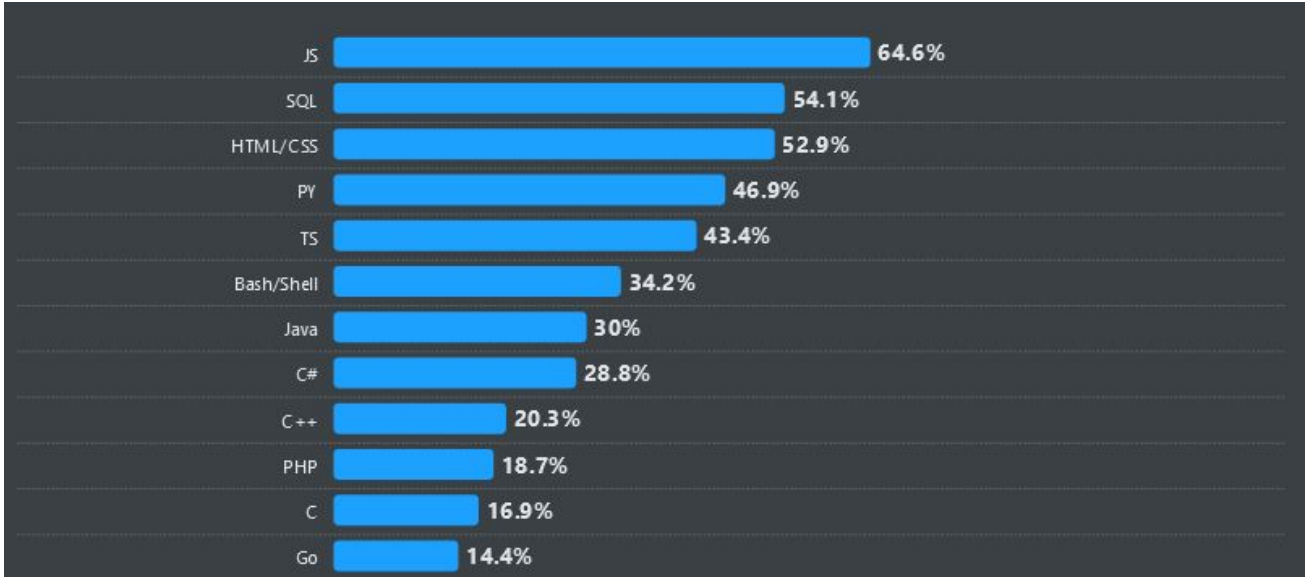
- Conceitos iniciais
- Adicionando JavaScript no documento HTML
- Comentários e saída de dados
- Statements e sintaxe
- Variáveis, operadores e tipos de dados
- Condicionais
- Estruturas de Repetição
- Funções

O que é JavaScript?

- Linguagem de programação **mais popular** do mundo atualmente
- Linguagem de programação da Web
- Linguagem leve, interpretada e baseada em objetos
- Além disso, é uma linguagem de script para a Web que é executada no lado cliente por um navegador
- Recentemente tem sido bastante utilizada também como linguagem de programação no lado servidor, através de plataformas como o Node.js
- O JavaScript surge em 1995 e torna-se um padrão ECMA em 1997, tendo como nome oficial **ECMAScript**

O que é JavaScript? (Popularidade)

- Ranking de Linguagens de Programação, Script e de Marcação do Stack Overflow (2024 Developer Survey)



O que é possível fazer com o JavaScript?

- Alterar o conteúdo dos elementos HTML
- Modificar conteúdo de uma página Web
- Modificar valores de atributos em elementos HTML
- Alterar estilos CSS aplicados em elementos HTML
- Reagir a eventos, como o clique em um botão
- Validar dados em um formulário antes de submetê-los
- E muitas outras possibilidades…

Adicionando JavaScript no documento HTML

- Dentro de uma tag **<script>** no HTML

index.html

```
<script>
      document.getElementById("demo").innerHTML = "My First JavaScript";
</script>
```

- Arquivos JS externos

myScript.js

```
function myFunction() {
      document.getElementById("demo").innerHTML = "Paragraph changed.";
}
```

index.html

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

Adicionando JavaScript no documento HTML

- Adicionar código diretamente nos atributos

index.html

```
<button onclick="console.log('JS na tag!')">Clique aqui!</button>
```

não recomendado!

Adicionando JavaScript no documento HTML

- Problemas e estratégias para o carregamento de Scripts
  - A ordem de carregamento pode impactar no resultado e causar problemas
  - Scripts que dependem de outros scripts devem ser carregados na sequência
  - Scripts que manipulam elementos da página devem ser carregados após o HTML ser completamente carregado
  - Soluções modernas: *async* e *defer*

index.html

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

Adicionando JavaScript no documento HTML

- Problemas e estratégias para o carregamento de Scripts
  - A ordem de carregamento pode impactar no resultado e causar problemas
  - Scripts que dependem de outros scripts devem ser carregados na sequência
  - Scripts que manipulam elementos da página devem ser carregados após o HTML ser completamente carregado
  - Soluções modernas: *async* e *defer*

index.html

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

Comentários em JavaScript

- Assim como em outras linguagens de programação, é possível escrever comentários dentro do código JS, que serão ignorados pelo navegador
- Há dois tipos de comentários em JS: *comentário de linha* e *comentário de múltiplas linhas*

myScript.js

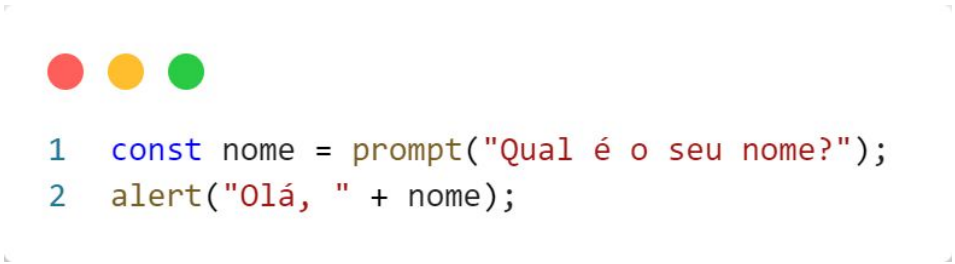
```
// Eu sou um comentário de uma única linha
```

myScript.js

```
/*
      Eu sou um comentário
      de múltiplas linhas
*/
```

Entrada de dados

- O JavaScript pode trabalhar com entrada de dados de diferentes maneiras,a depender da aplicação. Em um ambiente Web tradicional, as formas mais comuns de receber dados são através de formulários ou prompts.
    - método `prompt()` - permite capturar dados diretamente do usuário através de um pop-up na tela.



```
1  const nome = prompt("Qual é o seu nome?");
2  alert("Olá, " + nome);
```

Entrada de dados

- **entrada via formulários** - o JavaScript pode manipular esses dados com eventos como submit, change ou input.

```html
<form id="meuForm">
      <input type="text" id="nome" placeholder="Digite seu nome" />
      <button type="submit">Enviar</button>
    </form>

    <script>
      document
        .getElementById("meuForm")
        .addEventListener("submit", function (event) {
          event.preventDefault(); // Evita o comportamento padrão de enviar o formulário
          const nome = document.getElementById("nome").value;
          alert("Nome enviado: " + nome);
        });
    </script>
```

Saídas de dados

- O JavaScript pode “exibir” dados de diferentes maneiras:
  - escrevendo dentro de um elemento HTML com a propriedade `innerHTML`
  - escrevendo na saída do documento HTML usando `document.write()`
  - escrevendo dentro de uma caixa de alerta usando `window.alert()`
  - escrevendo no console do *browser* usando `console.log()`

JavaScript Statements

- Statements (declarações de instruções) no JavaScript são compostas de: valores, operadores, expressões, palavras reservadas e comentários
- O ponto e vírgula (;) separa as instruções em JavaScript, embora possamos codar sem utilizá-lo

myScript.js

```
let a, b, c;
a = 5;
b = 6;
c = a + b;

let x
x = c + 2
```

JavaScript Statements

- Instruções JavaScript podem ser agrupadas em blocos de códigos demarcados por chaves **{ }**

myScript.js

```
function myFunction() {
      document.getElementById("demo1").innerHTML = "Hello Dolly!";
      document.getElementById("demo2").innerHTML = "How are you?";
}
```

JavaScript Statements

- Instruções geralmente iniciam com uma palavra-chave (palavra reservada) para indicar a ação que deve ser executada
- Algumas das palavras reservadas do JS, são:

Palavra reservada	Descrição
var	Declara uma variável
let	Declara uma variável de bloco
const	Declara uma constante de bloco
if	Indica um bloco de instruções que serão executadas a depender de uma condição
switch	Indica um bloco de instruções que serão executadas em diferentes casos
for	Indica um bloco de instruções que serão executadas em um loop
function	Declara uma função
return	Finaliza uma função
try	Implementa um gerenciador de erro para um bloco de instruções

Sintaxe JavaScript: detalhes importantes

- Regras gerais para nomear uma variável ou função:
  - um nome deve começar com uma letra (**A-Z** ou **a-z**), um símbolo de cifrão (**$**) ou um underline (**_**)
  - números não são permitidos para iniciar o nome de uma variável ou função
  - os caracteres subsequentes podem ser letras, digitos, underline ou símbolos de cifrão
- JavaScript é **Case Sensitive**:
  - as variáveis ***lastName*** e ***lastname***, são variáveis distintas

Sintaxe JavaScript: detalhes importantes

- JavaScript não define um padrão para representar nomes compostos para variáveis ou funções, porém é muito comum a adoção do estilo ***Camel Case*** para esse propósito
- Algumas formas de juntar palavras para nomear variáveis e funções:
    - ***Hífens (Kebab Case):*** last-name, first-name
    - ***Underscore (Snake Case):*** first_name, last_name
    - ***Upper Camel Case***: FirstName, LastName
    - ***Lower Camel Case:*** firstName, lastName

Variáveis

- Variáveis em JavaScript podem ser declaradas de 4 formas diferentes:
  - automaticamente
  - usando `var`
  - usando `let`
  - usando `const`

Variáveis

- Variáveis em JavaScript podem ser declaradas de 4 formas diferentes:
  - automaticamente
  - usando `var`
  - usando `let`
  - usando `const`

myScript.js

```
x = 5;
y = 6;
z = x + y;
```

- **Variáveis não declaradas, são declaradas automaticamente em seu primeiro uso!**

Variáveis

- Variáveis em JavaScript podem ser declaradas de 4 formas diferentes:
  - automaticamente
  - usando **`var`**
  - usando `let`
  - usando `const`

myScript.js

```
var x = 5;
var y = 6;
var z = x + y;
```

- O var só deve ser usado em códigos escritos para navegadores antigos
- O gerenciamento de escopo é “confuso”
- Usado antes do ES6

Variáveis

- Variáveis em JavaScript podem ser declaradas de 4 formas diferentes:
  - automaticamente
  - usando `var`
  - usando `let`
  - usando `const`

myScript.js

```
let x = 5;
let y = 6;
let z = x + y;
```

- Pensado para trazer o conceito de escopo de bloco
- Agora uma variável declarada em um bloco específico tem seu escopo limitado a ele
- Variáveis declaradas com let não podem ser re-declaradas
- Devem ser declaradas antes do uso

Variáveis

- Variáveis em JavaScript podem ser declaradas de 4 formas diferentes:
  - automaticamente
  - usando `var`
  - usando `let`
  - usando `const`

myScript.js

```
const x = 5;
const y = 6;
const z = x + y;
```

- Usa-se o const quando o valor não deve ser alterado
- Somente use let quando não for possível usar o const

Variáveis

- Variáveis em JavaScript podem ser declaradas de 4 formas diferentes:
  - automaticamente
  - usando `var`
  - usando `let`
  - usando `const`

myScript.js

```
const cars = ["Saab", "Volvo", "BMW"];
cars[0] = "Toyota";
cars.push("Audi");
```

- Usa-se o const quando o valor não deve ser alterado
- Somente use let quando não for possível usar o const
- const não define um valor constante, define uma referência constante

Operadores

- Operadores aritméticos

Operador	Descrição
=	Operador de atribuição
+, *, -, /	Operadores aritméticos básicos
**	Exponenciação
%	Módulo
--, ++	Decremento e Incremento

Operadores

- Operadores de comparação

Operador	Descrição
==	Igual a
===	Valor e tipo iguais
!=	Diferente
!==	Valor e tipo diferentes
>	Maior que
<	Menor que
>=	Maior ou igual
<=	Menor ou igual
?	Operador ternário

Operadores

- Operadores lógicos

Operador	Descrição
&&	AND
||	OR
!	NOT

- Operadores de tipo

Operador	Descrição
typeof	Retorna o tipo de uma variável
instanceof	Retorna verdadeiro se um objeto é uma instância de um tipo de objeto

Tipos de Dados

- O JS tem 8 tipos de dados:
  - String
  - Number
  - Bigint
  - Boolean
  - Undefined
  - Null
  - Symbol
  - Object

myScript.js

```
// Numbers:
let length = 16;
let weight = 7.5;

// Strings:
let color = "Yellow";
let lastName = "Johnson";

// Booleans
let x = true;
let y = false;
```

Tipos de Dados

- O JS tem 8 tipos de dados:
  - String
  - Number
  - Bigint
  - Boolean
  - Undefined
  - Null
  - Symbol
  - Object

- O tipo de dado *object* pode conter:
  - um objeto
  - um array
  - uma data

myScript.js

```
// Object:
const person = {firstName:"John", lastName:"Doe"};

// Array object:
const cars = ["Saab", "Volvo", "BMW"];

// Date object:
const date = new Date("2022-03-25");
```

Tipos de Dados: detalhes importantes

- Em JS uma variável pode conter qualquer tipo de dado (**tipagem dinâmica**)
- Quando adicionamos um valor *number* com uma *string*, o JS trata o número também como uma *string*
- Strings podem ser escritas com aspas simples (‘ ’) ou aspas duplas (“ ”)
- Todo número em JS é armazenado como números decimais (ponto flutuante, com 64-bit)
- Para trabalhar com números inteiros fora do intervalo de representação do tipo *number* pode-se utilizar o tipo *BigInt* (ECMA2020)

Condicionais

- Frequentemente, ao escrever um código, nos deparamos com a necessidade de executar ações que dependem de alguma condição
- No JavaScript temos algumas estruturas de código para realizar controle de fluxo condicinal:
  - `if`
  - `else`
  - `else if`
  - `switch`

Condicionais

- Usamos a estrutura **`if`** quando queremos executar um bloco de código no caso da condição testada retornar verdadeiro **`(true)`**
- Sintaxe:

myScript.js

```
if (condition) {
  //  bloco de código que será executado caso a condição seja verdadeira
}
```

Condicionais

- Usamos a estrutura **`else`** quando queremos executar um bloco de código no caso da condição testada retornar falso **`(false)`**
- Sintaxe:

myScript.js

```
if (condition) {
  //  bloco de código que será executado caso a condição seja verdadeira
} else {
  //  bloco de código que será executado caso a condição seja falsa
}
```

Condicionais

- Usamos a estrutura **`else if`** quando queremos testar uma nova condição, caso a primeira condição tenha retornada falsa
- Sintaxe:

myScript.js

```
if (condition1) {
  //  bloco de código que será executado caso a condição 1 seja verdadeira
} else if (condition2) {
  //  bloco de código que será executado caso a condição 1 seja falsa e a condição 2 seja verdadeira
} else {
  //  bloco de código que será executado caso as condições 1 e 2 sejam falsas
}
```

Condicionais

- Usamos a estrutura **`switch`** para selecionar um bloco de código, entre vários, para ser executado a depender do caso de teste
- Sintaxe:

myScript.js

```
switch(expression) {
  case x:
      // bloco de código
      break;
  case y:
      // bloco de código
      break;
  default:
    // bloco de código
}
```

myScript.js

```
switch(expression) {
  case x:
      // bloco de código
      break;
  case y:
      // bloco de código
      break;
  default:
    // bloco de código
}
```

Condicionais

- Usamos a estrutura **`switch`** para selecionar um bloco de código, entre vários, para ser executado a depender do caso de teste
- Exemplo:

myScript.js

```
switch(expression) {
  case x:
      // bloco de código
      break;
  case y:
      // bloco de código
      break;
  default:
    // bloco de código
}
```

myScript.js

```
switch(mes) {
  case 1:
      return "Janeiro";
      break;
  case 2:
      return "Fevereiro";
      break;
  default:
    return "Mês inválido";
}
```

Laços de Repetição

- Outro tipo de estruturas de controle de fluxo bastante úteis são os *loops*, ou laços de repetição
- Com essas estruturas, é possível executar um mesmo trecho de código repetidas vezes, geralmente atuando sobre dados diferentes.
- No JavaScript temos as seguintes estruturas básicas para implementar laços de repetição:
  - `for`
  - `for in`
  - `for of`
  - `while / do while`

Laços de Repetição

- Com a estrutura **`for`** percorremos um bloco de código várias vezes (geralmente uma quantidade específica de vezes)
- Sintaxe:

`myScript.js`

```
for (expression 1; expression 2; expression 3) {
  // bloco de código a ser executado
}
```

- **Expressão 1:** executada ***somente uma vez***, antes da execução do bloco de código, geralmente para inicializar uma variável de controle
- **Expressão 2:** define a ***condição*** para a execução do bloco de código
- **Expressão 3:** executada após a execução do bloco de código, geralmente para alterar o valor da variável de controle

Laços de Repetição: `for`

- Com a estrutura **`for`** percorremos um bloco de código várias vezes (geralmente uma quantidade específica de vezes)
- Exemplo:

myScript.js

```
for (let i = 0; i < 5; i++) {
  console.log("O número é " + i);
  soma += i;
}
```

Laços de Repetição: `for in`

- Com a estrutura **`for in`** conseguimos percorrer as propriedades de um objeto
- Sintaxe:

myScript.js

```
for (key in object) {
  // bloco de código a ser executado
}
```

- **key:** variável utilizada para armazenar o valor de cada propriedade percorrida dentro do objeto, em cada iteração do for
- **object:** referência ao objeto a ser percorrido

Laços de Repetição: `for in`

- Com a estrutura **`for in`** conseguimos percorrer as propriedades de um objeto
- Exemplo:

myScript.js

```
const car = {brand:"Fiat", model:"Toro", year:2024};

for (let p in car) {
  console.log(p);
}
```

Laços de Repetição: `for of`

- Com a estrutura **`for of`** conseguimos percorrer valores de um ***objeto iterável***
- Um ***Objeto Iterável*** é um objeto que pode ser percorrido (**iterado**) de forma sequencial em um laço de repetição
- Sintaxe:

`myScript.js`

```
for (variable of iterable) {
  // bloco de código a ser executado
}
```

- **variable:** variável utilizada para armazenar o valor de cada elemento individual do objeto iterável
- **iterable:** referência ao objeto a ser iterado

Laços de Repetição: `for of`

- Com a estrutura **`for of`** conseguimos percorrer valores de um ***objeto iterável***
- Exemplo:

myScript.js

```
// exemplo de iteração em array
const cars = ["Fiat", "Ford", "Toyota"];

for (let x of cars) {
  console.log(x);
}
```

myScript.js

```
// exemplo de iteração em string
let language = "JavaScript";

for (let x of language) {
  console.log(x);
}
```

Laços de Repetição: `while`

- Com a estrutura **`while`** podemos executar um bloco de código repetidas vezes, **enquanto a condição testada for verdadeira**
- Sintaxe:

`myScript.js`

```
while (condition) {
  // bloco de código a ser executado
}
```

- **condition:** condição que é sempre testada no início de cada iteração. O bloco de código vai ser executado enquanto essa condição retornar **true**.

Laços de Repetição: `while`

- Com a estrutura **`while`** podemos executar um bloco de código repetidas vezes, **enquanto a condição testada for verdadeira**
- Exemplo:

myScript.js

```
let i = 0;

while (i < 10) {
  console.log("O valor de i é: " + i);
  i++;
}
```

Laços de Repetição: `do while`

- A estrutura **`do while`** é uma variante da estrutura **`while`** e também executa um bloco de código repetidas vezes, **enquanto a condição testada for verdadeira**
- Contudo, diferentemente da estrutura while, na estrutura do while, o bloco de código é executado antes da verificação da condição
- Como consequência, o bloco de código é **executado pelo menos uma vez**
- Sintaxe:

myScript.js

```
do {
  // bloco de código a ser executado
} while (condition);
```

Laços de Repetição: `do while`

- A estrutura **`do while`** é uma variante da estrutura **`while`** e também executa um bloco de código repetidas vezes, **enquanto a condição testada for verdadeira**
- Contudo, diferentemente da estrutura while, na estrutura do while, o bloco de código é executado antes da verificação da condição
- Exemplo:

myScript.js

```
let i = 0;

do {
  console.log("O valor de i é: " + i);
  i++;
} while (i < 10);
```

Laços de Repetição: `break` e `continue`

- Em algumas situações é interessante parar a execução de um laço de repetição e sair dele, principalmente quando o objetivo inicial do laço já foi alcançado
- Em outras situações é interessante pular uma iteração do laço e passar diretamente para a iteração seguinte sem terminar de executar o código dentro do laço
- Para essas situações podem ser utilizadas as palavras reservadas break e continue, respectivamente

Laços de Repetição: `break` e `continue`

- Em algumas situações é interessante parar a execução de um laço de repetição e sair dele, principalmente quando o objetivo inicial do laço já foi alcançado
- Em outras situações é interessante pular uma iteração do laço e passar diretamente para a iteração seguinte sem terminar de executar o código dentro do laço
- Para essas situações podem ser utilizadas as palavras reservadas break e continue, respectivamente

myScript.js

```
for (let i = 1; i <= 10; i++) {
  if (i % 5 === 0 ) { break; }
  console.log(i);
}
```

myScript.js

```
for (let i = 1; i <= 10; i++) {
  if (i % 5 === 0 ) { continue; }
  console.log(i);
}
```

Exercício

- Escreva um programa que calcule a soma dos cinco primeiros números ímpares encontrados no intervalo entre dois valores informados pelo usuário. Se não houver pelo menos cinco números ímpares no intervalo informado, o programa deve alertar isso ao usuário e solicitar um novo intervalo de valores. O programa encerra quando o usuário informar um intervalo que contenha pelo menos cinco números ímpares e a soma deles seja exibida.

Funções

- Uma função nada mais é do que um bloco de código que executa uma tarefa específica quando chamado

myScript.js

```
// Função para calcular o produto de p1 e p2
function myFunction(p1, p2) {
  return p1 * p2;
}
```

- Sintaxe:
  - definida pela palavra reservada **function**, seguida pelo seu **nome** e por **parênteses ()**
  - os parênteses podem incluir nomes de **parâmetros** separados por vírgula
  - o bloco de código da função é colocado dentro de **chaves {}**

Funções

- Uma função também pode ser definida utilizando uma **expressão**:

myScript.js

```
// Função para calcular o produto de p1 e p2
const x = function (p1, p2) {return p1 * p2};
let z = x(4, 3);
```

- A abordagem acima é conhecida como **função anônima**

Funções: parâmetros com valor default

- O JavaScript permite definir valores padrão (default) para parâmetros de funções

myScript.js

```
// Se não o argumento y não é passado, então y = 10 por padrão
function myFunction(x, y = 10) {
  return x + y;
}
myFunction(5);
```

Funções: rest parameters

- O recurso de **rest parameters (...)** permite uma função tratar um número indefinido de argumentos como um array

myScript.js

```
// O parâmetro args encapsula um array com os argumentos passados
function sum(...args) {
  let sum = 0;
  for (let arg of args) sum += arg;
  return sum;
}

let x = sum(4, 9, 16, 25, 29, 100, 66, 77);
```

Funções: chamada e retorno

- O código de uma função é executado quando ela é chamada (invocada), o que pode ocorrer quando:
  - um evento ocorre (quando o usuário clica em um botão, por exemplo)
  - é chamada a partir de uma instrução de código JavaScript
  - automaticamente
- Uma função para de executar quando encontra a instrução **return**. Essa instrução é utilizada quando queremos retornar um valor computado na função para quem a chamou, como no exemplo a seguir:

myScript.js

```
// A função é chamada e o valor retornado é guardado na variável x
let x = myFunction(4, 3);

function myFunction(a, b) {
// A função retorna o produto de a e b
  return a * b;
}
```