MÓDULO VII

Introdução à Programação I

A Educação é o primeiro passo para um futuro melhor...

MÓDULO VII

Funções em JavaScript:

Se você está começando a explorar esse universo incrível da programação, já deve ter ouvido falar sobre funções. Elas são como super poderes do código: permitem organizar, reutilizar e tornar tudo mais eficiente

O que são funções?

Funções em *JavaScript* são blocos de código que realizam uma tarefa específica. Elas são como robôs programados para realizar ações. Você cria a função uma vez e depois pode chamá-la sempre que precisar. Isso torna o código mais organizado, além de evitar repetição desnecessária.

MÓDULO VII

Exemplo básico de uma *função*

Vamos começar criando uma *função* bem simples que imprime uma saudação:

```
function saudacao() {
  console.log('Olá, dev! Bem-vindo ao mundo do JavaScript!');
}
```

Agora, sempre que você quiser ver essa mensagem, basta chamar a *função*:

```
saudacao(); // Saída: Olá, dev! Bem-vindo ao mundo do JavaScript!
```

Simples, né? Você define o que a função faz e depois pode usá-la quantas vezes quiser.

Criando funções no *JavaScript*

Existem diferentes formas de criar funções em *JavaScript*, e cada uma tem suas particularidades. Se esses conceitos estão parecendo confusos à primeira vista, não se preocupe! É super normal achar tudo isso um pouco complicado no começo. Mas, com a prática e o tempo, essas ideias vão se tornar cada vez mais naturais. O importante é continuar praticando, errar faz parte do processo.

MÓDULO VII

Declaração de função

A forma mais comum de criar uma função é usando a palavra-chave *function*:

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

Aqui, *somar* é o nome da função, *a* e *b* são os parâmetros (valores que a função vai receber) e *return* devolve o resultado da soma. Para chamar a função e ver o resultado, basta fazer:

```
console.log(somar(2, 3)); // Saída: 5
```

console.log?

O *console.log* é uma ferramenta muito útil para visualizar o que está acontecendo no seu código. Ele imprime no console do navegador (*ou terminal, no caso de Node.js*) o que você pedir. Nesse exemplo, ele exibe o resultado da *função somar(2, 3)* no console, que é *5*. Isso ajuda bastante a testar e depurar seu código enquanto você desenvolve!

MÓDULO VII

📌 Exemplo 1: Função Básica (Saudação)

```
// Função que diz "Oi" para o usuário
function cumprimentar(nome) {
  return `Oi, ${nome}! Tudo bem? 👋`;
}

// Usando a função:
console.log(cumprimentar("Maria")); // Saída: "Oi, Maria! Tudo bem? 👋"
console.log(cumprimentar("João"));  // Saída: "Oi, João! Tudo bem? 👋"
```

✔️ O que aprendemos:

- *function* define a *função*.
- nome é um ***parâmetro*** (variável que recebe o valor).
- *return* devolve o resultado.

MÓDULO VII

📌 Exemplo 2: Função com Cálculo (Tabuada)

```
// Função que mostra a tabuada de um número
function tabuada(numero) {
  for (let i = 0; i <= 10; i++) {
    console.log(`${numero} x ${i} = ${numero * i}`);
  }
}

// Usando a função:
tabuada(5); // Mostra a tabuada do 5
tabuada(7); // Mostra a tabuada do 7
```

✔️ O que aprendemos:

- Funções podem ter **loops** dentro.
- Podem ser chamadas várias vezes com valores diferentes.

MÓDULO VII

📌 Exemplo 3: Função com Condicional (Maior de Idade)

```
// Função que verifica se a pessoa é maior de idade
function verificarIdade(idade) {
  if (idade >= 18) {
    return "Pode dirigir! 🚗";
  } else {
    return "Ainda não pode dirigir! 🚸";
  }
}

// Usando a função:
console.log(verificarIdade(20)); // Saída: "Pode dirigir! 🚗"
console.log(verificarIdade(15)); // Saída: "Ainda não pode dirigir! 🚸"
```

✔️ O que aprendemos:

- *Funções* podem usar *if/else*.
- Retornam resultados diferentes conforme a condição

MÓDULO VII

📌 Exemplo 4: Função com Array (Lista de Compras)

```
// Função que adiciona itens a uma lista de compras
let listaCompras = ["maçã", "leite"];

function addItem(item) {
  listaCompras.push(item); // Adiciona o item ao array
  console.log(`✅ ${item} foi adicionado! Lista atual: ${listaCompras}`);
}

// Usando a função:
addItem("pão");    // Saída: "✅ pão foi adicionado! Lista atual: maçã,leite,pão"
addItem("queijo"); // Saída: "✅ queijo foi adicionado! Lista atual: maçã,leite,pão,queijo"
```

✔️ O que aprendemos:

- *Funções* podem **modificar *arrays*** com métodos como *push()*.
- Podem acessar variáveis de fora (como *listaCompras*).

MÓDULO VII

📌 Exemplo 5: Função com Template String (Bio de Rede Social)

```
// Função que gera uma bio estilizada
function criarBio(nome, idade, hobby) {
  return `
  ✨ ${nome.toUpperCase()} ✨
  Idade: ${idade} anos
  Hobby: ${hobby}
  `;
}

// Usando a função:
console.log(criarBio("Carlos", 16, "jogar futebol"));
/* Saída:
✨ CARLOS ✨
Idade: 16 anos
Hobby: jogar futebol
*/
```

✔️ O que aprendemos:

- *Funções* podem formatar textos com **template strings**.
- *toUpperCase()* transforma texto em maiúsculas.

MÓDULO VII

🎮 **Analogia com Videogame:**

Pense em *funções* como **poderes especiais** em um jogo:

- **Nome do poder**: Nome da *função* (*cumprimentar, tabuada*).
- **Energia necessária**: *Parâmetros* (*nome, numero*).
- **Efeito**: O que a *função* faz (*return* ou *console.log*).

❓ **Perguntas para Fixar:**

1. Como criar uma *função* que calcule a média de 3 notas?

```
function media(nota1, nota2, nota3) {
  return (nota1 + nota2 + nota3) / 3;
}

console.log(media(4, 5, 7));
// Saída:
// 5.333333333333333
```

2. **O que acontece se esquecer o *return*?**
   → A *função* retorna undefined.

MÓDULO VII

🛠️ Desafio Prático:

Crie uma *função* ***verificarAprovacao*** que:

- Receba uma nota (0 a 10).
- Retorne "Aprovado 🎉" se a nota for ≥ *6*.
- Retorne "Reprovado 😢" se for < *6*.

Solução:

```
const verificarAprovacao = (nota) => nota >= 6 ? "Aprovado 🎉" : "Reprovado 😢";
console.log(verificarAprovacao(7)); // "Aprovado 🎉"
```

MÓDULO VII

Funções anônimas

Outra maneira de criar *funções* é utilizando *funções* anônimas.
Elas são chamadas assim porque, diferente das *funções* tradicionais, não têm um nome específico.
Em vez disso, são atribuídas a uma variável, o que permite chamá-las quando necessário.

```
const multiplicar = function(a, b) {
  return a * b;
};
```

Aqui, a *função* foi atribuída à variável multiplicar.
Para chamar essa *função*, você usa o nome da variável, como faria com qualquer outra *função*:

```
console.log(multiplicar(2, 4)); // Saída: 8
```

Funções anônimas são muito úteis quando você precisa passar uma *função* como argumento para outra *função* (*como em callbacks*) ou criar uma *função* rapidamente, sem se preocupar em nomeá-la. Elas são bastante comuns em situações onde o foco está mais na ação que a *função* executa do que no nome que ela tem.

MÓDULO VII

Ao trabalhar com funções anônimas, você pode facilmente criar lógica dinâmica e flexível dentro do seu código, tornando o desenvolvimento mais fluido.

Callbacks?

Callbacks são *funções* que são passadas como argumento para outra *função* e executadas depois que algum processo é concluído. Em outras palavras, elas permitem que você defina uma *função* para ser chamada no futuro, após o término de uma operação.

MÓDULO VII

📌 Exemplo 1: Função Anônima Básica (Atribuída a uma Variável)

```
// Função anônima armazenada numa variável (como um "poder sem nome" em um jogo)
const cumprimentar = function(nome) {
  return `E aí, ${nome}! Beleza? 👍`;
};

// Usando a função:
console.log(cumprimentar("Lucas")); // Saída: "E aí, Lucas! Beleza? 👍"
console.log(cumprimentar("Ana"));   // Saída: "E aí, Ana! Beleza? 👍"
```

✔️ O que aprendemos:

- A *função* não tem nome depois do *function* (por isso é anônima).
- Ela é atribuída à variável cumprimentar e pode ser chamada por ela.

MÓDULO VII

📌 Exemplo 2: Função Anônima em um *Array* (*.forEach*)

```
// Lista de games favoritos
const games = ["Minecraft", "Fortnite", "Roblox"];

// Função anônima para mostrar cada game (como um "modo replay automático")
games.forEach(function(game) {
  console.log(`Jogando: ${game} 🎮`);
});

/* Saída:
Jogando: Minecraft 🎮
Jogando: Fortnite 🎮
Jogando: Roblox 🎮
```

✔️ O que aprendemos:

- *Funções* anônimas podem ser usadas direto em métodos como *forEach*.
- game é o item atual do *array* em cada volta do loop.

MÓDULO VII

📌 Exemplo 3: Função Anônima como Argumento (SetTimeout)

```
// Função anônima como argumento (como um "alarme secreto")
setTimeout(function() {
  console.log("Hora do lanche! 🍕");
}, 3000); // Executa após 3 segundos (3000 milissegundos)
```

✔️ O que aprendemos:

- Aqui, a *função* anônima é passada direto para *setTimeout*.
- Ela será executada **após um tempo determinado** (3 segundos).

MÓDULO VII

Exemplo 4: Arrow Function Anônima (Versão Moderna)

```
// Arrow function anônima (ainda mais curta!)
const dobrar = function(numero) {
  return numero * 2;
};

// Versão arrow function:
const triplicar = (numero) => numero * 3;

// Usando as funções:
console.log(dobrar(5));    // Saída: 10
console.log(triplicar(5)); // Saída: 15
```

✔️ O que aprendemos:

- Arrow functions (=>) são uma forma moderna de escrever *funções* anônimas.
- Se tiver só uma linha, o *return* é implícito.

MÓDULO VII

🎮 Analogia com TikTok:

Pense em *funções* anônimas como **vídeos temporários** que você grava e usa uma vez:

- Não tem nome fixo (são "*anônimos*").
- São úteis para **ações rápidas** (como um story que some depois de 24h).

❓ Perguntas para Fixar:

1. **Qual a diferença entre *função* nomeada e anônima?**
   → *Funções* nomeadas têm identificador (ex: *function soma()*), enquanto anônimas não (mas podem ser atribuídas a variáveis).
2. **Posso usar *arrow functions* como anônimas?**
   → Sim! Exemplo:

```
setTimeout(() => console.log("Hello!"), 2000);
```

MÓDULO VII

🛠️ Desafio Prático:

Crie uma *função* anônima que:

- Receba um *array* de números.
- Use *map* para dobrar cada número.
- Retorne o novo *array*.

Solução:

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

MÓDULO VII

Arrow functions

As *arrow functions* foram pensadas como uma maneira mais curta e simples de escrever *funções*.
Elas eliminam a necessidade da palavra-chave *function* e têm uma sintaxe mais enxuta:

```
const dividir = (a, b) => a / b;
```

Além da sintaxe curta, uma das principais vantagens é que elas não alteram o valor do *this*, mantendo o contexto léxico da *função* onde foi criada. Isso as torna especialmente úteis em *callbacks* e *funções* simples:

```
const numeros = [1, 2, 3, 4];
const dobrados = numeros.map(n => n * 2);
```

No entanto, como não possuem seu próprio this, elas não são ideais para métodos de objetos que dependem desse contexto.

MÓDULO VII

📌 1. Arrow Function Básica (Postagem em Rede Social)

```
// Versão tradicional
const postar = function(mensagem) {
  return `📢 ${mensagem}`;
};

// Versão Arrow Function (simplificada!)
const postar = mensagem => `📢 ${mensagem}`;

// Usando:
console.log(postar("Aprendendo Arrow Functions!"));
// Saída: 📢 Aprendendo Arrow Functions!
```

✔️ **O que aprendemos:**

- Quando tem **só 1 *parâmetro***, os *()* são opcionais.
- Quando tem **só 1 linha**, o *return* é implícito.

MÓDULO VII

📌 2. Arrow Function sem Parâmetros (Alerta de Loading)

```
// Arrow Function sem parâmetros (usamos parênteses vazios)
const carregando = () => console.log("⏳ Carregando...");

// Usando:
carregando();
// Saída: ⏳ Carregando...
```

✔️ O que aprendemos:

- Sem parâmetros, os () são **obrigatórios**.

MÓDULO VII

📌 3. Arrow Function em Callbacks (Lista de Games)

```
const games = ["Minecraft", "Fortnite", "Roblox"];

// Arrow Function como callback no .map()
const gamesUpper = games.map(game => game.toUpperCase());

console.log(gamesUpper);
// Saída: ["MINECRAFT", "FORTNITE", "ROBLOX"]
```

✔️ O que aprendemos:

- Arrow Functions são **perfeitas para callbacks curtas**.

MÓDULO VII

🎮 Analogia com Emojis:

- ***Função*** **Tradicional** => 🧙 function() { ... } (mágico com varinha).
- ***Arrow Function*** => 🏹 () => ... (flecha rápida e direta).

❓ Perguntas para Fixar:

1. **Quando usar {} em Arrow Functions?**
   → Quando a função tem **mais de uma linha** ou precisa de return explícito.
2. **Posso usar *Arrow Functions* para tudo?**
   → Quase! Evite em **métodos de objeto** ou ***constructors*** (onde this importa).
3. **Como transformar isto em *Arrow Function*?**

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

Resposta:

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

MÓDULO VII

🛠️ **Desafio Prático:**

Transforme esta *função* tradicional em *Arrow Function*:

```
function criarPowerUp(nome, nivel) {
  return `✨ ${nome} (Nível ${nivel})`;
}
```

Solução:

```
const criarPowerUp = (nome, nivel) => `✨ ${nome} (Nível ${nivel})`;
console.log(criarPowerUp("Invencibilidade", 5));
// Saída: ✨ Invencibilidade (Nível 5)
```

MÓDULO VII

Contexto léxico, *this*?

O contexto léxico basicamente significa "*onde*" uma *função* foi criada. Ele define o que a *função* pode acessar em termos de variáveis e objetos. Por exemplo, se você cria uma *função* dentro de outra, essa *função* "*lembra*" do ambiente onde foi criada e pode acessar variáveis daquele escopo.

Agora, o this é uma palavra que usamos para referir ao "*dono*" da *função*, ou seja, a quem a *função* pertence. Em *funções* normais, o *this* pode mudar dependendo de como a *função* é chamada.

Isso pode causar confusão, porque o valor de *this* pode variar.

Com as *arrow functions*, o valor de *this* não muda! Ele é "*preso*" ao contexto onde a *função* foi criada. Isso significa que, dentro de uma *arrow function*, o *this* sempre se refere ao que está fora dela, o que facilita muito o trabalho em certos casos.

Resumindo: o contexto léxico define o que uma função pode acessar, e o this diz "quem" está chamando essa função. As arrow functions mantêm esse this fixo no contexto onde foram criadas, tornando o código mais previsível!

MÓDULO VII

Parâmetros e argumentos

As *funções* podem receber parâmetros, que são valores passados para a *função* executar sua tarefa.

Por exemplo, uma *função* que recebe um nome e imprime uma saudação personalizada:

```
function saudacaoComNome(nome) {
  console.log(`Olá, ${nome}!`);
}

saudacaoComNome('Rocketseat'); // Saída: Olá, Rocketseat!
```

Os argumentos são os valores passados quando chamamos a função, como no exemplo acima, onde o argumento é 'Rocketseat'.

MÓDULO VII

📌 1. Conceito Básico (Lanche no Ifood)

```
// PARÂMETROS são como o "cardápio" da função (variáveis que ela espera)
function fazerPedido(lanche, bebida, quantidade) { // parâmetros
  console.log(`🍔 ${quantidade}x ${lanche} com ${bebida} saindo!`);
}

// ARGUMENTOS são os "itens reais" que você pede
fazerPedido("X-Burger", "Coca", 2); // argumentos
/* Saída: 🍔 2x X-Burger com Coca saindo! */
```

✔️ O que aprendemos:

- **Parâmetros**: Variáveis na definição da função (lanche, bebida).
- **Argumentos**: Valores reais passados na chamada ("X-Burger", "Coca").

MÓDULO VII

📌 2. Valores Padrão (Configuração de Wi-Fi)

```
// Parâmetro com valor padrão (senha = "1234")
function conectarWifi(rede, senha = "1234") {
  console.log(`📶 Conectado a ${rede}. Senha: ${senha}`);
}

// Chamadas:
conectarWifi("Casa"); // Usa senha padrão
/* Saída: 📶 Conectado a Casa. Senha: 1234 */

conectarWifi("Escola", "Aula@2023"); // Sobrescreve a senha
/* Saída: 📶 Conectado a Escola. Senha: Aula@2023 */
```

✔️ O que aprendemos:

- *Parâmetros* podem ter **valores padrão** (*senha = "1234"*).
- Se omitido, o *argumento* usa o valor padrão.

MÓDULO VII

📌 3. Argumentos Dinâmicos (Lista de Amigos)

```
// Função com parâmetro REST (...amigos)
function marcarEncontro(local, ...amigos) {
  console.log(`📍 ${local}: Encontro com ${amigos.join(", ")}`);
}

// Chamada com múltiplos argumentos
marcarEncontro("Shopping", "Ana", "Lucas", "Bia");
/* Saída: 📍 Shopping: Encontro com Ana, Lucas, Bia */
```

✔️ O que aprendemos:

- ...amigos agrupa **todos os *argumentos* extras** em um *array*.
- Útil para listas variáveis.

MÓDULO VII

🎮 Analogia com Videogame:

- ***Parâmetros*** = Slots vazios no inventário do jogo.
- ***Argumentos*** = Itens que você coloca nesses slots.

Slot do Inventário (Parâmetro)	Item Equipado (Argumento)
arma	"Espada Lendária"
pocao	"Cura Total"

❓ Perguntas para Fixar:

1. **Posso pular argumentos?**
   → Sim, mas serão undefined (ou use valores padrão).

```
function teste(a, b) { console.log(a, b) }
teste(1); // Saída: 1 undefined
```

2. **Quantos argumentos posso passar?**
   → Quantos quiser! Use ...rest para capturar todos.
3. **Posso passar funções como argumentos?**
   → Claro! Isso são **callbacks**:

```
function calcular(a, b, operacao) {
  return operacao(a, b);
}
calcular(2, 3, (x, y) => x + y); // 5
```

MÓDULO VII

🛠️ Desafio Prático:

Crie uma *função montarPersonagem* que:

- Aceite 3 *parâmetros*: *classe, arma, habilidade.*
- Retorne uma string no formato:
  "⚔️ [*classe*] com [*arma*] (Habilidade: [*habilidade*])".

Solução:

```
function montarPersonagem(classe, arma, habilidade) {
  return `⚔️ ${classe} com ${arma} (Habilidade: ${habilidade})`;
}

console.log(montarPersonagem("Guerreiro", "Machado", "Golpe Crítico"));
/* Saida: ⚔️ Guerreiro com Machado (Habilidade: Golpe Crítico) */
```

MÓDULO VII

Argumentos padrão

Você também pode definir valores padrão para os parâmetros da *função*.
Se o *argumento* não for fornecido, a *função* usa o valor padrão:

```
function saudacaoComNome(nome = 'dev') {
  console.log(`Olá, ${nome}!`);
}

saudacaoComNome(); // Saída: Olá, dev!
```

Retorno de valores

Funções podem retornar valores usando a palavra-chave *return*.

Isso é útil quando você quer usar o resultado da *função* em outra parte do código:

```
function multiplicar(a, b) {
  return a * b;
}

const resultado = multiplicar(3, 4);
console.log(resultado); // Saída: 12
```

MÓDULO VII

📌 1. Função com Argumentos (Sistema de XP em Jogo)

```
// Função que calcula XP ganho após uma missão
function ganharXP(monstro, xpBase) {
  const xpTotal = xpBase * 2; // XP dobrada para desafio!
  console.log(`🎉 Você derrotou ${monstro} e ganhou ${xpTotal} XP!`);
  return xpTotal; // Retorna o valor para uso futuro
}

// Usando:
const xp = ganharXP("Dragão", 500); // Passando dois argumentos
console.log(`XP acumulado: ${xp}`);

/* Saída:
🎉 Você derrotou Dragão e ganhou 1000 XP!
XP acumulado: 1000
*/
```

✔️ O que aprendemos:

- monstro e *xpBase* são **argumentos** (dados que a *função* recebe).
- *return* envia o resultado de volta para quem chamou a *função*.

MÓDULO VII

📌 2. Múltiplos Argumentos (Customização de Personagem)

```
// Função que cria um personagem com várias propriedades
function criarPersonagem(nome, classe, nivel = 1) { // valor padrão para nível
  return {
    nome: nome,
    classe: classe,
    nivel: nivel,
    info: `🧙 ${nome} (${classe}) - Nível ${nivel}`
  };
}

// Usando:
const mago = criarPersonagem("Gandalf", "Mago", 10);
const guerreiro = criarPersonagem("Aragorn", "Guerreiro"); // Nível 1 por padrão

console.log(mago.info);     // Saída: 🧙 Gandalf (Mago) - Nível 10
console.log(guerreiro.info); // Saída: ⚔️ Aragorn (Guerreiro) - Nível 1
```

✔️ O que aprendemos:

- *Argumentos* podem ter **valores padrão** (*nivel = 1*).
- *Funções* podem retornar **objetos complexos**.

MÓDULO VII

📌 3. Retorno Condicional (Verificação de Passe de Batalha)

```
// Função que verifica se o jogador pode entrar em um evento
function verificarPasse(idade, temPasse) {
  if (idade >= 16 && temPasse) {
    return "✅ Acesso liberado ao Torneio!";
  } else {
    return "❌ Nível ou passe insuficientes!";
  }
}

// Usando:
console.log(verificarPasse(15, true));  // Saída: ❌ Nível ou passe insuficientes!
console.log(verificarPasse(17, true));  // Saída: ✅ Acesso liberado ao Torneio!
```

✔️ O que aprendemos:

- *return* pode estar dentro de condicionais *(if/else)*.
- A *função* para de executar quando encontra um *return*.

MÓDULO VII

🎮 Analogia com Videogame:

- ***Argumentos***: Como **cheats** que você digita para mudar o jogo (*god mode on*).
- ***Retorno***: Como o **resultado** na tela após o comando ("*Vida infinita ativada!*").

❓ Perguntas para Fixar:

1. **Posso passar uma *função* como *argumento*?**
   → Sim! Isso se chama ***callback*** (ex: *array.map(função)*).
2. **E se esquecer o *return*?**
   → A *função* retorna undefined.
3. **Como retornar múltiplos valores?**
   → Use um ***array*** ou ***objeto***:

```
function stats() { return [10, 20]; }
const [ataque, defesa] = stats();
```

MÓDULO VII

🛠️ Desafio Prático:

Crie uma *função combinarItens* que:

1. Receba 2 itens de jogo (*strings*).
2. Retorne um novo item combinado (ex: "Espada + Poção = Espada Mágica").

Solução:

```
function combinarItens(item1, item2) {
  const combinacoes = {
    "Espada + Poção": "Espada Mágica 🔮",
    "Arco + Flecha": "Arco Potente 🏹"
  };
  const chave = `${item1} + ${item2}`;
  return combinacoes[chave] || "Combinação desconhecida!";
}

console.log(combinarItens("Espada", "Poção")); // "Espada Mágica 🔮"
```

MÓDULO VII

Escopo das funções

Um conceito importante em funções é o escopo, que define onde as variáveis estão disponíveis. As variáveis declaradas dentro de uma função não são acessíveis fora dela:

Escopo local

Variáveis declaradas dentro de uma função não são acessíveis fora dela.

```
function exemploEscopo() {
  let mensagem = 'Dentro da função';
  console.log(mensagem);
}

exemploEscopo(); // Saída: Dentro da função
console.log(mensagem); // Erro: mensagem não está definida
```

Se tentar acessar mensagem fora da função, dará erro.

MÓDULO VII

Escopo global

Variáveis declaradas fora de funções são acessíveis em qualquer parte do código.

```
let mensagemGlobal = 'Fora da função';

function mostrarMensagem() {
  console.log(mensagemGlobal);
}

mostrarMensagem(); // Saída: Fora da função
```

MÓDULO VII

1. Escopo Global vs. Local (Analogia com um Jogo RPG)

Imagine que seu código é um **mapa de RPG**:

- **Variáveis globais**: Como **itens no seu inventário** (todo mundo vê).
- **Variáveis locais**: Como **itens dentro de um baú** (só a *função* atual enxerga).

Código Exemplo:

```
// ✨ Variável GLOBAL (como um item no inventário)
let ouro = 100;

function entrarNaCaverna() {
  // 🔑 Variável LOCAL (como um baú secreto)
  let tesouro = 50;
  console.log(`Você achou ${tesouro} moedas na caverna!`);
  ouro += tesouro; // Pega o tesouro e adiciona ao ouro global
}

entrarNaCaverna(); // Saída: Você achou 50 moedas na caverna!
console.log(`Ouro total: ${ouro}`); // Saída: Ouro total: 150
// console.log(tesouro); // ❌ ERRO! tesouro não existe fora da caverna
```

✔️ O que aprendemos:

- ouro é ***global***: Pode ser acessada em qualquer lugar.
- tesouro é ***local***: Só existe dentro da *função entrarNaCaverna*().

MÓDULO VII

📌 2. Escopo em Blocos (*if/for*)

Variáveis com *let* e *const* têm escopo de **bloco** (*dentro de {}*).

Código Exemplo:

```
function batalha() {
  let vida = 100; // 🔵 Escopo da função

  if (vida > 0) {
    let golpe = 20; // 🔴 Escopo do bloco if
    console.log(`Você deu ${golpe} de dano!`);
    vida -= golpe;
  }

  console.log(`Vida restante: ${vida}`);
  // console.log(golpe); // ❌ ERRO! golpe não existe aqui
}

batalha();

/* Saída:
Você deu 20 de dano!
Vida restante: 80
*/
```

✔️ **O que aprendemos:**

- golpe só existe dentro do *if*.
- vida existe em toda a *função* batalha().

MÓDULO VII

📌 3. Var vs. Let (Vazamento de Escopo)

var vaza o escopo de blocos (*como um balão furado* 🎈), enquanto *let* e *const* não.

Código Exemplo:

```
function exemploVar() {
  if (true) {
    var item = "Poção"; // vaza para a função toda
    let arma = "Espada"; // só existe no bloco
  }
  console.log(item); // ✅ "Poção" (var vazou)
  // console.log(arma); // ❌ ERRO! arma não existe
}

exemploVar();
```

✔️ O que aprendemos:

- **Evite *var***: Pode causar bugs inesperados.
- **Prefira *let/const***: São mais seguras.

MÓDULO VII

❓ Perguntas para Fixar

1. **Como proteger uma variável para não ser global?**
   → Use *let/const* dentro de *funções* ou blocos.
2. **O que acontece se duas *funções* usarem o mesmo nome de variável local?**
   → Cada uma terá sua **cópia independente** (não conflitam).
3. **Como uma *função* acessa uma variável global?**
   → Diretamente pelo nome (ouro no Exemplo 1).

🎮 Analogia com Minecraft

- **Escopo global**: Como itens no seu inventário (todos os biomas enxergam).
- **Escopo local**: Como itens dentro de um baú (só aquele chunk acessa).
- **Escopo de bloco**: Como uma redstone que só funciona num circuito.

MÓDULO VII

🛠️ Desafio Prático

Crie uma *função* cofre que:

1. Tenha uma variável local segredo com um número.
2. Retorne uma *função* que **adicione** um valor ao segredo.
3. Mostre como o escopo protege o segredo de ser acessado diretamente.

Solução:

```
function cofre() {
  let segredo = 0; // 🔒 Variável local

  return function(valor) {
    segredo += valor;
    console.log(`Segredo atual: ${segredo}`);
  };
}

const guardar = cofre();
guardar(10); // Segredo atual: 10
guardar(5);  // Segredo atual: 15
// console.log(segredo); // ❌ ERRO! segredo está protegido
```

MÓDULO VII

Funções de alta ordem

No *JavaScript*, as *funções* são tratadas como "*primeira classe*", ou seja, podem ser passadas como argumentos ou retornadas por outras *funções*. *Funções* que fazem isso são chamadas de *funções* de alta ordem. Elas são extremamente poderosas, pois permitem criar um código mais flexível e dinâmico.

Um exemplo simples de *função* de alta ordem é uma *função* que recebe outra *função* como argumento:

```
function executarFuncao(funcao) {
  funcao();
}

executarFuncao(() => {
  console.log('Função passada como argumento!');
});
```

Isso é muito útil em várias situações, como em eventos de clique em uma página web, onde você passa uma função que será executada quando o usuário clicar em um botão, ou em operações assíncronas, como carregar dados de uma API.

MÓDULO VII

📌 O que são Funções de Alta Ordem?

São funções que:

1. **Recebem outras funções como argumentos** (callbacks),
   OU
2. **Retornam funções** (como uma "fábrica de funções").

Vamos aos exemplos:

MÓDULO VII

🎮 Exemplo 1: Função que Recebe Callback (Sistema de Batalha em RPG)

```
// Função de alta ordem (executa uma habilidade do jogador)
function usarHabilidade(jogador, habilidade) {
  console.log(`⚔️ ${jogador} usou:`);
  habilidade(); // Executa a callback
}

// Callbacks (habilidades)
function ataqueBasico() {
  console.log("Golpe de espada! 💥 -10 HP");
}

function magiaDeGelo() {
  console.log("Rajada de gelo! ❄️ -25 HP");
}

// Usando:
usarHabilidade("Guerreiro", ataqueBasico);
usarHabilidade("Mago", magiaDeGelo);

/* Saída:
⚔️ Guerreiro usou:
Golpe de espada! 💥 -10 HP
⚔️ Mago usou:
Rajada de gelo! ❄️ -25 HP
*/
```

✔️ O que aprendemos:

- *usarHabilidade* é uma ***função*** **de alta ordem** porque recebe uma *função* (habilidade) como argumento.

MÓDULO VII

🛠️ Exemplo 2: Função que Retorna Outra Função (Criadora de Armas)

```
// Função de alta ordem (fábrica de armas)
function criarArma(tipo) {
  return function(dano) {
    console.log(`🚀 Arma criada: ${tipo} (${dano} de dano) 💥`);
  };
}

// Gerando armas:
const espada = criarArma("Espada Flamejante");
const arco = criarArma("Arco Arcano");

// Usando as armas:
espada(15); // Saída: 🚀 Arma criada: Espada Flamejante (15 de dano) 💥
arco(12);   // Saída: 🚀 Arma criada: Arco Arcano (12 de dano) 💥
```

✔️ O que aprendemos:

- *criarArma* **retorna uma *função*** personalizada para cada tipo de arma.

MÓDULO VII

📌 Exemplo 3: Função que Filtra Dados (Moderação de Conteúdo)

```
// Função de alta ordem (filtra posts ofensivos)
function moderarPosts(posts, filtro) {
  return posts.filter(filtro); // .filter() é outra função de alta ordem embutida
}

// Callback (verifica se o post é seguro)
function ehSeguro(post) {
  const palavrasProibidas = ["ódio", "xingamento"];
  return !palavrasProibidas.some(palavra => post.includes(palavra));
}

// Usando:
const posts = [
  "Amo programar!",
  "Isso é um xingamento!",
  "Vamos jogar?"
];

const postsAprovados = moderarPosts(posts, ehSeguro);
console.log(postsAprovados); // Saída: ["Amo programar!", "Vamos jogar?"]
```

✔️ **O que aprendemos:**

- *moderarPosts* usa *Array.filter()* com uma *função* (filtro) para decidir quais posts manter.

MÓDULO VII

🌟 Por que isso é útil?

- **Reutilização de código**: Escreva a lógica uma vez (ex: *moderarPosts*) e mude o comportamento passando funções diferentes.
- **Abstração**: Esconda detalhes complexos (como loops) dentro de *funções* como *map*, *filter*.
- **Flexibilidade**: Crie "*fábricas*" de *funções* (*criarArma*) para gerar comportamentos personalizados.

🎯 Desafio Prático

Crie uma *função* de alta ordem *criarInimigo* que:

1. Receba um **tipo de inimigo** (ex: "*zumbi*", "*esqueleto*").
2. Retorne uma *função* que **ataca** com um dano específico.

Solução:

```
function criarInimigo(tipo) {
  return function(dano) {
    console.log(`👾 ${tipo} atacou! Causou ${dano} de dano.`);
  };
}

const zumbi = criarInimigo("Zumbi Sangrento");
zumbi(8); // Saída: 👾 Zumbi Sangrento atacou! Causou 8 de dano.
```

MÓDULO VII

Callback functions

Um tipo muito comum de *função* de alta ordem são as *callbacks*, que são *funções* passadas como *argumentos* para outras *funções* e executadas posteriormente. Elas são amplamente usadas em operações assíncronas, como ao lidar com APIs ou eventos de usuário.

Vamos a um exemplo:

```
function saudacaoPersonalizada(nome, callback) {
  const mensagem = `Olá, ${nome}!`;
  callback(mensagem);
}

saudacaoPersonalizada('Dev', (msg) => {
  console.log(msg);
});
```

Nesse caso, a função callback é executada depois que a função saudacaoPersonalizada processa a mensagem, tornando o código mais flexível e modular.

MÓDULO VII

📌 Exemplo 1: Callback Simples (Like no Instagram)

```
// Função principal (postar foto)
function postarFoto(legenda, callback) {
  console.log(`📸 Foto postada: "${legenda}"`);
  callback(); // Chama a função de like depois
}

// Callback (função de like)
function darLike() {
  console.log("❤️ Você curtiu esta foto!");
}

// Usando:
postarFoto("Praia no verão", darLike);

/* Saída:
📸 Foto postada: "Praia no verão"
❤️ Você curtiu esta foto!
*/
```

✔️ O que aprendemos:

- *postarFoto* recebe uma legenda e uma *função* (*callback*).
- Depois de postar, executa o "like" (*darLike*).

MÓDULO VII

📌 Exemplo 2: Callback com Array (Playlist do Spotify)

```
// Função principal (tocar playlist)
function tocarPlaylist(playlist, callback) {
  console.log("🎶 Tocando playlist:");
  playlist.forEach(musica => console.log(musica));
  callback(playlist.length); // Chama callback com o total de músicas
}

// Callback (mostrar resumo)
function resumoPlaylist(totalMusicas) {
  console.log(`🎧 Total de músicas: ${totalMusicas}`);
}

// Usando:
const minhaPlaylist = ["Samba", "Funk", "Trap"];
tocarPlaylist(minhaPlaylist, resumoPlaylist);

/* Saída:
🎶 Tocando playlist:
Samba
Funk
Trap
🎧 Total de músicas: 3
*/
```

✔️ O que aprendemos:

- *Callbacks* podem **receber argumentos** (*totalMusicas*).
- São úteis para ações pós-processamento.

MÓDULO VII

📌 Exemplo 3: Callback em Temporizador (Alarme do Despertador)

```
// Função principal (programar despertador)
function despertar(tempo, callback) {
  setTimeout(() => {
    console.log(`⏰ Alarmando após ${tempo} segundos!`);
    callback();
  }, tempo * 1000);
}

// Callback (ação pós-alarme)
function tomarCafe() {
  console.log("☕ Hora do café da manhã!");
}

// Usando:
despertar(5, tomarCafe); // Alarme em 5 segundos

/* Saída (após 5 segundos):
⏰ Alarmando após 5 segundos!
☕ Hora do café da manhã!
*/
```

✔️ O que aprendemos:

- *setTimeout* usa uma ***arrow function anônima*** como *callback*.
- A *função tomarCafe* é executada depois do alarme.

MÓDULO VII

🎮 Analogia com Videogame:

Pense em *callbacks* como **missões secundárias** em um RPG:

1. Você completa a missão principal (*postarFoto*).
2. O jogo **automaticamente** ativa a missão extra (darLike).

❓ Perguntas para Fixar:

1. **Como saber quando usar *callbacks*?**
   → Quando uma ação **depende** do resultado de outra (ex: carregar dados → mostrar na tela).
2. **Posso usar *arrow functions* como *callbacks*?**
   → Sim! Exemplo moderno:

```
document.addEventListener("click", () => console.log("Clicou!"));
```

MÓDULO VII

Métodos de objetos

Métodos são *funções* que são associadas a um *objeto*.

Quando uma *função* é declarada dentro de um *objeto*, ela passa a ser chamada de *método* desse *objeto*. Um *método* pode acessar as propriedades do próprio *objeto* usando a palavra-chave *this*, que faz referência ao *objeto* ao qual o *método* pertence.

Aqui está um exemplo:

```
const pessoa = {
  nome: 'João',
  saudacao() {
    console.log(`Olá, meu nome é ${this.nome}`);
  },
};

pessoa.saudacao(); // Saída: Olá, meu nome é João
```

MÓDULO VII

O que está acontecendo?

- A *função saudacao* foi definida dentro do *objeto pessoa*, tornando-a um *método* desse *objeto*.
- O *this* dentro da *função* se refere ao próprio *objeto pessoa*. Dessa forma, t*his.nome* acessa o valor da propriedade nome do *objeto*, que é *'João'*.

Os métodos são uma forma eficiente de associar comportamentos a objetos e permitem criar funcionalidades reutilizáveis no contexto dos dados que o objeto contém.

MÓDULO VII

Boas práticas para criar *funções*

Seguir boas práticas ao criar *funções* ajuda a manter o código mais legível, fácil de manter e menos propenso a erros. Aqui estão algumas dicas essenciais que refletem as práticas atualizadas.

- **Responsabilidade única**: cada *função* deve realizar apenas uma tarefa específica. Isso torna o código mais modular e fácil de entender. *Funções* com muitas responsabilidades tendem a ser confusas e difíceis de depurar.
- **Nomes significativos**: dê nomes que descrevam claramente o que a *função* faz. Isso facilita a leitura e a manutenção do código. Em vez de calcular, por exemplo, prefira algo mais descritivo como calcularTotalComDesconto.
- **Evite efeitos colaterais (side effects)**: *funções* devem ser previsíveis, ou seja, devem depender apenas dos parâmetros passados e não alterar variáveis externas ou o estado do sistema de forma inesperada. Isso garante que, ao chamar a *função*, você saiba exatamente o que ela fará.

MÓDULO VII

- **Limite de parâmetros**: prefira *funções* com poucos parâmetros. Idealmente, não mais que três. Se sua função precisa de muitos parâmetros, considere usar um objeto para agrupá-los ou dividir a *função* em partes menores.
- **Adote a sintaxe moderna**: utilize *arrow functions* e *desestruturação* de *objetos* e *arrays* nos seus códigos para refletir as práticas atuais.

Manipulação de arrays

Funções são amplamente utilizadas na manipulação de *arrays*, especialmente com *métodos* como *.map()*, *.filter()* e *.reduce()*. Esses *métodos* permitem transformar, filtrar ou reduzir os *arrays* usando *funções* anônimas ou *arrow functions*.

MÓDULO VII

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

const quadrados = numeros.map((num) => num * num);

console.log(quadrados); // Saída: [1, 4, 9, 16, 25]
```

A função passada para .map() transforma cada número do array no seu respectivo quadrado. Esse tipo de operação é muito comum quando você precisa transformar ou processar dados em uma lista.

MÓDULO VII

EXERCÍCIOS...

A Educação é o primeiro passo para um futuro melhor...

MÓDULO VII

FIM...

A Educação é o primeiro passo para um futuro melhor...