Desenvolvimento Front-end

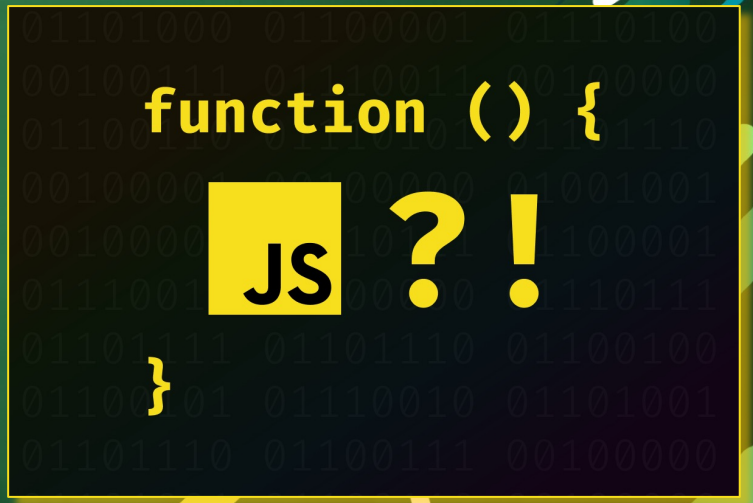
Disciplina

Cássia Perego

Professora

Dione Ferrari

Professor



Unoeste | EAD

Aula: Introdução às Funções em TypeScript

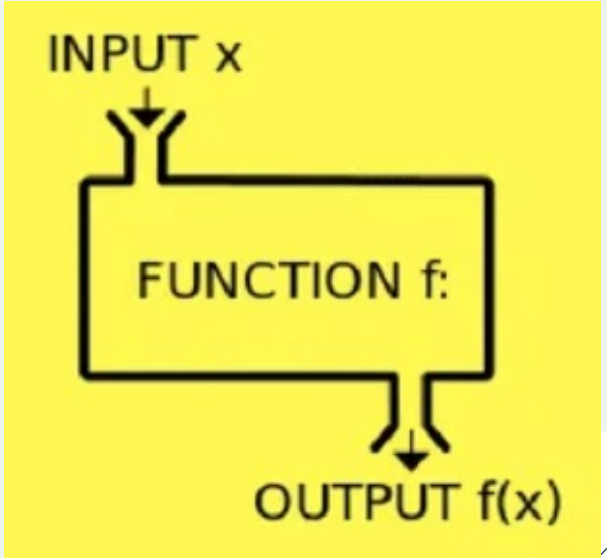
Objetivo da Aula:

Nesta aula, vamos **explorar o conceito de funções** em JavaScript/TypeScript. Vamos aprender como declarar funções, passar parâmetros, definir tipos de retorno e utilizar funções de ordem superior. Além disso, veremos exemplos práticos para consolidar o entendimento sobre o tema.

I. Introdução

B. Por que as funções são importantes em programação

- As **funções** são blocos de código reutilizáveis que podem ser definidos uma vez e executados em qualquer lugar do programa.
- Elas são uma parte fundamental da linguagem e são usadas para agrupar instruções relacionadas em uma unidade lógica.
- As funções podem aceitar parâmetros, realizar operações e retornar valores.



I. Introdução

B. Por que as funções são importantes em programação

- As funções são essenciais na programação porque proporcionam **organização**, **reutilização**, **abstração**, **testabilidade**, **facilitam a manutenção do código e permitem a colaboração eficaz entre desenvolvedores**.

- Elas são uma das principais construções que permitem aos programadores criar software complexo e eficiente.

I. Introdução

B. Por que as funções são importantes em programação

1. Reutilização de Código:

Funções permitem que você escreva um bloco de código que pode ser reutilizado em diferentes partes do programa. Em vez de repetir o mesmo código várias vezes, você pode encapsulá-lo em uma função e chamá-la sempre que precisar realizar uma tarefa específica. Isso promove a modularidade e facilita a manutenção do código.

2. Organização e Estruturação:

Usar funções ajuda a organizar o código em partes lógicas e estruturadas. Em programas grandes e complexos, dividir o código em funções facilita a compreensão, o debug e a colaboração entre os membros da equipe. Cada função pode se concentrar em realizar uma tarefa específica, tornando o código mais legível e fácil de entender.

I. Introdução

B. Por que as funções são importantes em programação

3. Abstração:

Funções permitem abstrair detalhes complexos do código. Ao criar uma função bem nomeada e documentada, outros desenvolvedores podem entender o que a função faz sem precisar entender os detalhes internos. Isso simplifica o uso de funcionalidades complexas, reduzindo a complexidade percebida do programa.

4. Facilidade de Manutenção:

Quando uma função precisa ser corrigida ou atualizada, você só precisa fazer isso em um único lugar. Sempre que a função for chamada em outros lugares do código, essas chamadas serão automaticamente atualizadas, facilitando a manutenção do sistema como um todo. Isso reduz a probabilidade de introduzir erros ao corrigir ou modificar funcionalidades.

I. Introdução

B. Por que as funções são importantes em programação

5. Testabilidade:

Funções bem definidas e isoladas são mais fáceis de testar. Você pode criar casos de teste específicos para cada função e garantir que ela produza os resultados esperados para diferentes conjuntos de entradas. Isso é essencial para garantir a qualidade do código e evitar regressões à medida que o programa é modificado ou expandido.

6. Implementação de Algoritmos:

Algoritmos complexos podem ser divididos em funções menores e mais gerenciáveis. Isso torna a implementação de algoritmos complicados mais acessível, pois cada parte do algoritmo pode ser tratada separadamente, permitindo uma abordagem mais modular e menos propensa a erros.

I. Introdução

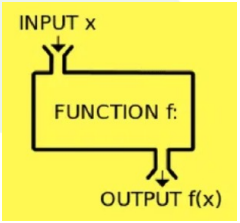
B. Por que as funções são importantes em programação

7. Facilidade de Colaboração:

Em projetos de software colaborativos, funções bem definidas e documentadas facilitam a colaboração entre diferentes desenvolvedores. Cada desenvolvedor pode trabalhar em funções específicas do sistema sem interferir no trabalho dos outros, desde que a interface da função (parâmetros e retorno) seja mantida.

II. Declarando Funções

A. Sintaxe básica para declarar funções em TypeScript

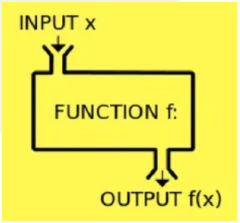


- Em TypeScript, a estrutura básica de uma função inclui o **nome da função**, uma **lista de parâmetros** com seus tipos e um **tipo de retorno**.

```typescript
function soma(a: number, b: number): number {
    // corpo da função
    return a + b;
}
```

II. Declarando Funções

A. Sintaxe básica para declarar funções em TypeScript



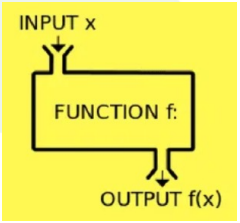
- Em TypeScript, a estrutura básica de uma função inclui o **nome da função**, uma **lista de parâmetros** com seus tipos e um **tipo de retorno**.

```
function soma(a: number, b: number): number {
    // corpo da função
    return a + b;
}
```

Nome da Função:
O nome da função é um identificador que a diferencia de outras funções no seu código.
Ele segue as mesmas regras que os identificadores em JavaScript e TypeScript, ou seja, deve começar com uma letra (maiúscula ou minúscula), sublinhado (_) ou cifrão ($), seguido por letras, dígitos ou mais sublinhados.

II. Declarando Funções

A. Sintaxe básica para declarar funções em TypeScript



- Em TypeScript, a estrutura básica de uma função inclui o **nome da função**, uma **lista de parâmetros** com seus tipos e um **tipo de retorno**.

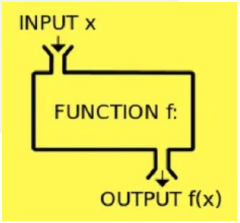
```
function soma(a: number, b: number): number {
    // corpo da função
    return a + b;
}
```

Lista de Parâmetros:
Uma função pode aceitar zero ou mais parâmetros. Os parâmetros são variáveis que representam os valores que a função espera receber quando é chamada. Cada parâmetro é seguido por dois pontos (:) e pelo tipo esperado para aquele parâmetro.

No exemplo acima, a e b são os parâmetros da função soma, ambos do tipo number.

II. Declarando Funções

A. Sintaxe básica para declarar funções em TypeScript



- Em TypeScript, a estrutura básica de uma função inclui o **nome da função**, uma **lista de parâmetros** com seus tipos e um **tipo de retorno**.

```typescript
function soma(a: number, b: number): number {
    // corpo da função
    return a + b;
}
```

Tipo de Retorno:
O tipo de retorno indica o tipo de valor que a função deve retornar após sua execução. É especificado após os parâmetros e dois pontos (:). Se uma função não retornar um valor, o tipo de retorno é void.

No exemplo da função soma, o tipo de retorno é number porque a função retorna a soma dos dois parâmetros, que são números.

II. Declarando Funções

A. Sintaxe básica para declarar funções em TypeScript

Estrutura Completa da Função em TypeScript:

```
function nomeDaFuncao(parametro1: Tipo, parametro2: Tipo): TipoDeRetorno {
    // corpo da função
    // código para processar os parâmetros e retornar um valor
}
```

Na estrutura acima:

nomeDaFuncao é o nome da função.

parametro1 e **parametro2** são os parâmetros da função, cada um com um tipo específico (Tipo).

TipoDeRetorno é o tipo de dado que a função deve retornar.

II. Declarando Funções

B. Parâmetros de função: tipos e valores padrão

Em JavaScript, as funções podem aceitar parâmetros, que são valores passados para a função quando ela é chamada.

Os parâmetros podem ser de qualquer tipo de dado, como números, strings, objetos, ou até mesmo outras funções.

Além disso, você pode definir valores padrão para os parâmetros, o que significa que se a função for chamada sem fornecer um valor para um parâmetro, o valor padrão será usado.

II. Declarando Funções

B. Parâmetros de função: tipos e valores padrão

1. Parâmetros sem Tipo Específico:

Em JavaScript, os parâmetros de função não têm tipos específicos. Eles podem aceitar qualquer tipo de dado, incluindo números, strings, arrays, objetos, ou até mesmo outras funções. Você simplesmente lista os nomes dos parâmetros entre os parênteses quando define a função.

```
function saudacao(nome) {
    console.log("Olá, " + nome + "!");
}

saudacao("João"); // Saída: "Olá, João!"
```

II. Declarando Funções

B. Parâmetros de função: tipos e valores padrão

2. Valores Padrão para Parâmetros (ES6 em diante):

A partir do ECMAScript 6 (ES6), você pode definir valores padrão para os parâmetros de função. Se um valor não for fornecido ao chamar a função, o valor padrão será usado.

```
function saudacao(nome = "Mundo") {
    console.log("Olá, " + nome + "!");
}

saudacao(); // Saída: "Olá, Mundo!"
saudacao("Alice"); // Saída: "Olá, Alice!"
```

II. Declarando Funções

B. Parâmetros de função: tipos e valores padrão

3. Argumentos de Função (arguments):

Em JavaScript, você pode acessar os argumentos passados para uma função usando a palavra-chave *arguments*.

arguments é uma variável local disponível dentro de todas as funções que contém uma lista de todos os argumentos passados para a função.

```
function soma() {
    let total = 0;
    for (let i = 0; i < arguments.length; i++) {
        total += arguments[i];
    }
    return total;
}

console.log(soma(1, 2, 3)); // Saída: 6
```

II. Declarando Funções

C. Exemplos práticos de declaração de funções simples

1. Função para Somar Dois Números:

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

let resultado = somar(3, 5);
console.log(resultado); // Saída: 8
```

II. Declarando Funções

C. Exemplos práticos de declaração de funções simples

2. Função para Verificar se um Número é Par:

```
function ehPar(numero) {
    return numero % 2 === 0;
}

console.log(ehPar(4)); // Saída: true
console.log(ehPar(7)); // Saída: false
```

II. Declarando Funções

C. Exemplos práticos de declaração de funções simples

3. Função para Calcular a Área de um Retângulo:

```
function calcularAreaRetangulo(altura, largura) {
    return altura * largura;
}

let area = calcularAreaRetangulo(4, 6);
console.log(area); // Saída: 24
```

III. Tipos de Retorno

Em JavaScript, as funções podem retornar qualquer tipo de dado, incluindo números, strings, objetos, arrays e até outras funções.

Exemplo de uma função que retorna um número:

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

let resultado = soma(3, 5);
console.log(resultado); // Saída: 8 (um número)
```

III. Tipos de Retorno

Exemplo de uma função que retorna uma string:

```
function saudacao(nome) {
    return "Olá, " + nome + "!";
}

let mensagem = saudacao("Alice");
console.log(mensagem); // Saída: "Olá, Alice!" (uma string)
```

IV. Funções Anônimas

As funções anônimas são funções que não possuem um nome associado. Elas podem ser usadas em situações em que você precisa de uma função temporária ou quando deseja passar uma função como argumento para outra função. As funções anônimas são frequentemente usadas em callbacks e em expressões imediatamente invocadas.

Exemplo de uma função anônima atribuída a uma variável:

```
let saudacao = function(nome) {
    console.log("Olá, " + nome + "!");
};

saudacao("Alice"); // Saída: "Olá, Alice!"
```

IV. Funções Anônimas

Sintaxe de Função Anônima

```
let minhaFuncao = function(parametro1, parametro2) {
    // Corpo da função
    // Código a ser executado
};
```

IV. Arrow Functions

As arrow functions (funções de flecha) são uma sintaxe mais curta e concisa para escrever funções em JavaScript. Elas foram introduzidas no ECMAScript 6 (ES6) e oferecem uma maneira mais elegante de escrever funções, especialmente para funções de ordem superior e callbacks.

Sintaxe básica de uma arrow function:

```
let quadrado = (x) => {
    return x * x;
};

console.log(quadrado(4)); // Saída: 16
```

IV. Arrow Functions

Sintaxe das Arrow Functions:

```
// Sem parâmetros
const funcaoSemParametro = () => {
    // Corpo da função
};

// Com um parâmetro
const funcaoComUmParametro = parametro => {
    // Corpo da função usando o parâmetro
};

// Com múltiplos parâmetros
const funcaoComMultiplosParametros = (parametro1, parametro2) => {
    // Corpo da função usando os parâmetros
};

// Retornando um valor implicitamente
const dobrar = numero => numero * 2;

// Com corpo de função múltiplas linhas e retorno explícito
const calcularArea = (largura, altura) => {
    let area = largura * altura;
    return area;
};
```

IV. Arrow Functions

Comparação com Funções Tradicionais:

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

// Arrow function equivalente
const somarArrow = (a, b) => a + b;
```

V. Funções de Ordem Superior

Funções de ordem superior são funções que aceitam outras funções como argumentos e/ou retornam funções.

Elas são uma parte importante do paradigma de programação funcional e permitem uma abordagem mais flexível e modular para lidar com operações em coleções de dados, como arrays.

V. Funções de Ordem Superior

Exemplo de uma função que recebe outra função como argumento:

```
function processarArray(array, callback) {
    let resultado = [];
    for (let elemento of array) {
        resultado.push(callback(elemento));
    }
    return resultado;
}

let numeros = [1, 2, 3, 4, 5];
let aoQuadrado = processarArray(numeros, function(numero) {
    return numero * numero;
});

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

V. Funções de Ordem Superior

Exemplo de uma função que retorna outra função:

```
function multiplicador(fator) {
    return function(numero) {
        return numero * fator;
    };
}

let duplicar = multiplicador(2);
let triplicar = multiplicador(3);

console.log(duplicar(4)); // Saída: 8
console.log(triplicar(4)); // Saída: 12
```

V. Funções de Ordem Superior

Funções de ordem superior são funções que aceitam outras funções como argumentos e/ou retornam funções.

Elas são uma parte importante do paradigma de programação funcional e permitem uma abordagem mais flexível e modular para lidar com operações em coleções de dados, como arrays.

VI. Exercícios Práticos

Exercício 1: Calculadora de Operações Matemáticas

Crie uma calculadora em TypeScript que seja capaz de realizar operações matemáticas básicas, como adição, subtração, multiplicação e divisão.

A calculadora deve ser implementada como uma função que aceita três parâmetros: dois números e uma string representando a operação a ser realizada ("soma", "subtração", "multiplicação" ou "divisão").

A função deve retornar o resultado da operação.

Exemplo de uso da função:

```
const resultadoSoma = calcularOperacao(5, 3, "soma"); // Deve retornar 8
const resultadoMultiplicacao = calcularOperacao(4, 2, "multiplicação");
```


Unoeste | EAD