USCS
UNIVERSIDADE DE SÃO CAETANO DO SUL

Linguagem de Programação

Prof. Renato Carioca Duarte

USCS
UNIVERSIDADE DE SÃO CAETANO DO SUL

Unidade 02

Programação Funcional

Programação Funcional

- A Programação Funcional é um paradigma de programação que trata a computação como uma avaliação de funções matemáticas e evita mudanças de estado e dados mutáveis.

- Nesse estilo de programação, as funções são tratadas como cidadãos de primeira classe, o que significa que podem ser atribuídas a variáveis, passadas como argumentos para outras funções e retornadas como resultados de outras funções.

Programação Funcional

- A programação funcional incentiva a criação de funções puras, que não têm efeitos colaterais e sempre retornam o mesmo resultado para os mesmos argumentos.

- JavaScript suporta programação funcional e é considerado uma linguagem multiparadigma, o que significa que pode ser usada tanto para programação funcional quanto para outros paradigmas, como a programação orientada a objetos.

Funções de primeira classe

- Funções que podem ser atribuídas a variáveis, passadas como argumentos e retornadas como resultados.

```
// Exemplo de função de primeira classe
const multiplicar = function(a, b) {
  return a * b;
};
```

Funções de ordem superior

- Funções que podem receber outras funções como argumentos ou retornar outras funções.

```
// Exemplo de função de ordem superior
// ou seja, que recebe uma função como argumento
function aplicarOperacao(a, b, operacao) {
  return operacao(a, b);
}
```

Callback

- Callback é uma função que é passada como argumento para outra função e é executada após a conclusão de uma operação assíncrona ou em um determinado momento dentro da função que a recebeu.

- Em JavaScript, as funções são tratadas como cidadãos de primeira classe, o que significa que elas podem ser tratadas como qualquer outra variável.
- Essa característica permite que funções sejam passadas como argumentos para outras funções, tornando-as callbacks.

Callback

- C é uma função de callback

```
function calcularSoma(a, b, c) {
    var resultado = a + b;
    c(resultado);
}

function mostrarResultado(resultado) {
    console.log('O resultado é: ' + resultado);
}

calcularSoma(5, 20, mostrarResultado);
```

Callback

- Neste exemplo, a função calcularSoma recebe dois números e uma função como argumentos.
- Ela calcula a soma dos dois números e passa o resultado para a função de callback (mostrarResultado), que é chamada com o resultado como argumento.
- Isso é outro exemplo de como uma função pode ser usada como um callback em JavaScript.

Callback

```
function saudacao(nome, callback) {
  const mensagem = "Olá, " + nome + "!";
  callback(mensagem);
}

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

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

Função Anônima

- Uma função anônima é uma função que não possui um nome associado a ela.
- Em JavaScript, as funções anônimas são comuns quando são utilizadas como argumentos em outras funções ou quando precisam ser definidas em um contexto onde um nome de função não é necessário ou não é relevante.

```
const minhaFuncao = function() {
    console.log('Olá, mundo!');
}

minhaFuncao();
```

Função Anônima

- As funções anônimas são frequentemente usadas em callbacks (funções que são passadas como argumentos para outras funções) e em situações onde precisamos criar uma função rapidamente sem precisar atribuir-lhe um nome específico.

```
function saudar(nome, callback) {
    console.log('Olá, ' + nome + '!');
    callback();
}

// Usando uma função anônima como callback
saudar('João', function() {
    console.log('Espero que você esteja tendo um ótimo dia!');
});
```

Função Anônima

- Neste exemplo, temos uma função chamada saudar que recebe um nome e uma função de callback.
- A função saudar exibe uma saudação no console com o nome fornecido e, em seguida, chama a função de callback.
- Ao chamar a função saudar com o nome "João" e uma função anônima como callback, teremos a seguinte saída no console:

```
Olá, João!
Espero que você esteja tendo um ótimo dia!
```

Função Seta

- Arrow functions, também conhecidas como funções de seta ou fat arrow functions, são uma forma mais concisa de escrever funções em JavaScript.
- Elas foram introduzidas no ECMAScript 6 (ES6) e são uma adição poderosa à sintaxe da linguagem.
- A principal característica das arrow functions é a sintaxe reduzida, tornando a escrita de funções mais simples e legível.
- Elas são especialmente úteis em contextos onde funções anônimas são comumente usadas, como callbacks e funções que retornam outras funções.

Função Seta

- Sintaxe básica de uma arrow function:

```
const nomeDaFuncao = (param1, param2, ... param3) => {
    // corpo da função
    return valor;
};
```

Características das Funções Seta

- **Concisão sintática**:

Se a função tem apenas uma expressão como retorno, não é necessário escrever o bloco de código ou usar a palavra-chave return. O valor retornado é implícito.

- **Não pode ser usada como construtor**:

Ao contrário das funções regulares, as arrow functions não podem ser usadas com o operador new para criar objetos.

Características das Funções Seta

- **Valor do this**:

As arrow functions têm um comportamento especial em relação ao valor do this.

Elas não têm seu próprio this e herdam o this do contexto em que são definidas.

Isso pode ser útil em situações onde o this pode mudar de contexto, como dentro de funções aninhadas ou métodos de objetos.

Função Seta

- **Exemplo básico:**

```
const saudacao = () => {
    return "Olá, mundo!";
};

console.log(saudacao()); // Saída: "Olá, mundo!"
```

Função Seta

- **Exemplo com retorno implícito:**

```
const dobrar = (numero) => numero * 2;

console.log(dobrar(5)); // Saída: 10
```

Função Seta

- **Exemplo com parâmetros:**

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

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

Função Seta

- **Exemplo com uso de this:**

```
const objeto = {
  valor: 10,
  multiplicar: function (x) {
    return x * this.valor;
  },
  multiplicarArrow: (x) => x * this.valor, // this não funciona aqui!
};

console.log(objeto.multiplicar(5)); // Saída: 50
console.log(objeto.multiplicarArrow(5)); // Saída: NaN (valor de this é diferente)
```

Closure

- Closure, em JavaScript, é um conceito que se refere à capacidade de uma função "lembrar" do escopo em que foi criada, mesmo que essa função seja executada fora desse escopo.
- Isso permite que a função tenha acesso às variáveis e parâmetros do escopo onde foi definida, mesmo que esse escopo já não exista mais.
- Um closure ocorre quando uma função é definida dentro do escopo de outra função e, em seguida, a função interna (a função aninhada) é retornada ou passada como callback para outras partes do código.

Closure

```
function criarContador() {
    let contador = 0;

    function incrementar() {
      contador++;
      console.log(contador);
    }

    return incrementar;
}

const contador1 = criarContador();
contador1(); // Saída: 1
contador1(); // Saída: 2

const contador2 = criarContador();
contador2(); // Saída: 1
contador2(); // Saída: 2
```

Closure

- Neste exemplo, temos uma função chamada criarContador, que retorna uma função chamada incrementar.
- A função incrementar é uma closure, pois ela "lembra" do escopo onde foi criada e tem acesso à variável contador, que é definida no escopo da função criarContador.
- Ao chamar criarContador, obtemos duas instâncias independentes da função incrementar em contador1 e contador2.
- Cada instância de incrementar mantém seu próprio escopo e variável contador, permitindo que as contagens sejam independentes entre elas.

Dúvidas

??????

Questionário

Questionário Unidade 2 no classroom

Exercícios

1.Defina uma função que receba uma função como argumento e retorne o resultado da função.

2. Defina uma função que receba uma função e retorne uma nova função que executa a função original apenas se uma condição for satisfeita.

3. Definir uma função de primeira classe que recebe outra função como parâmetro e a executa.

4. Criar uma função chamada add que recebe dois números e um callback. A função deve somar os dois números e passar o resultado para o call-back.

5. Defina uma função anônima que imprima o valor de uma variável.

6. Defina uma função seta que imprima o valor de uma variável.