Aula 7 - Funções - Parte 2

SECITECI
SECRETARIA DE ESTADO DE CIÊNCIA E TECNOLOGIA E INOVAÇÃO

GOVERNO DE MATO GROSSO

<Fic_Dev>
Programador de Sistemas

Roteiro

1. Desafio do Dia
2. Introdução
3. Funções Anônimas
4. Arrow function - Funções de Seta
5. Promisses (Promessas)
6. ASYNC/AWAIT
7. Conclusão

<Fic_Dev>
Programador de Sistemas

Desafio do Dia

Desenvolva um programa para registrar a compra de Criptomoedas.
Primeiramente armazene as cotações atuais para as seguintes moedas: Bitcoin (BTC), Ethereum (ETH) e Litecoin (LTC).
Então, solicite do usuário quanto deseja investir em cada cripto e apresente a quantidade que será adquirida.
Após a fase acima, solicite do usuário uma expectativa de valorização para cada moeda acima e recalcule os valores investidos, conforme as informações já coletadas. Por fim, apresente o valor investido inicialmente e o valor total com expectativa de valorização.
Um Plus: calcule o % de valorização total (Valor total esperado X Valor investido inicial).
Obs.: Use funções anônimas e de seta para os cálculos e Promessas para garantir que a expectativa de valorização seja maior que zero.

<Fic_Dev>
Programador de Sistemas

Introdução

- Vimos que as funçãos são blocos de código que executam uma tarefa específica e visam aumentar as chances de reutilização de código.
- São também conhecidas como Módulos ou Modularização
- Praticamente todas as linguagens de programação possuem a capacidade de modularização.

Introdução

- Importante comentar que as funções podem ou não possuir retorno. Para isso, o uso do comando **return** é fundamental.
- O comando **return** provoca a saída da função, entregando um resultado ou simplesmente encerrando sua execução.

Introdução

- Lembrando ainda que as funções podem receber parâmetros, que são valores enviados para as funções para que executem as tarefas.
- Como visto na aula passada, é importante que haja a prevenção a falhas, tratando as possível exceções por meio do comando **try...catch...finally**.

Funções Anônimas

- Conhecidas como funções sem nome ou funções lambda, pois não possuem um nome identificador.
- São usadas principalmente quando precisamos de uma função temporária ou quando queremos passar uma função como argumento para outra função.
- É como se passássemos um bloco de código para que alguma outra função execute.

Funções Anônimas

- Um exemplo bastante claro, é quando usamos a função **setTimeout** que é uma função que serve para executar um determinado bloco de código após um determinado tempo.

Sintaxe:

```
setTimeout(function() {
console.log("Executado após 3 segundos");
}, 3000);
```

Funções Anônimas

Vamos executar o código acima. Abra seu navegador e teste. Verá que a função será executada após 3 segundos e imprimirá a mensagem "Executado após 3 segundos"no console.

Funções Anônimas

- As funções anônimas são definidas usando a palavra-chave function sem especificar um nome após ela.
- Em vez disso, a função é atribuída a uma variável ou passada diretamente como argumento para outra função.

Aqui está um outro exemplo de uma função anônima:
Sintaxe:

```
var resultado = (function(a, b) {
return a + b;
})(3, 4);
console.log(resultado); // Output: 7
```

Arrow function

- São uma sintaxe alternativa, mais concisa, para escrever funções em JavaScript.
- Oferecem algumas vantagens em relação às funções tradicionais

Sintaxe:

```
const minhaFuncao = (param1, param2, ...) => {
// corpo da função
// retorno opcional
};
```

Arrow function

- A principal diferença entre as Arrow Functions e as funções tradicionais é o uso da seta => após a lista de parâmetros, indicando que a função será definida dessa forma.
- A palavra-chave function não é necessária e o retorno da função é implícito, a menos que seja usado um bloco de código **{ }**, caso em que o retorno deve ser explicitamente definido.

Arrow function

Aqui estão alguns exemplos para ilustrar o uso de Arrow Functions:

```
// Exemplo 1: Função que retorna o quadrado de um número
const quadrado = (num) => num * num;
// Exemplo 2: Função que soma dois números
const soma = (a, b) => a + b;
// Exemplo 3: Função que verifica se um número é par
const ehPar = (num) => {
return num % 2 === 0;
};
// Exemplo 4: Função que imprime uma mensagem no console
const saudacao = () => {
console.log('Olá!');
};
```

Arrow function

- Nesses exemplos, as Arrow Functions são usadas para definir funções que realizam tarefas específicas.
- A função quadrado retorna o quadrado de um número, a função soma realiza a soma de dois números, a função ehPar verifica se um número é par e a função saudacao imprime uma mensagem no console.

Arrow function - Concluindo

- As Arrow Functions são uma adição útil ao JavaScript moderno, oferecendo uma sintaxe mais concisa e legível para a criação de funções.
- Elas são amplamente utilizadas em muitos projetos e podem ajudar a melhorar a produtividade do desenvolvedor.

Promisses - Introdução

- As Promises (promessas) são um recurso importante em JavaScript para lidar com operações assíncronas.
- Elas fornecem uma maneira mais organizada e legível de lidar com o resultado ou o erro de uma operação assíncrona, como uma requisição HTTP, leitura de arquivo ou consulta a um banco de dados.

Promisses - Introdução

- É um objeto que representa a eventual conclusão (ou falha) de uma operação assíncrona e seu resultado.
- A ideia básica é que, em vez de esperar que uma operação assíncrona seja concluída antes de continuar a execução do código, podemos criar uma promessa que será resolvida posteriormente com o resultado da operação.

Promisses - Estados

Uma Promise representa um valor que pode estar disponível agora, no futuro ou nunca. Ela possui três estados possíveis:

- Pending: O estado inicial de uma Promise, indicando que a operação ainda está em andamento e o resultado não está disponível.
- Fulfilled: A operação assíncrona foi concluída com sucesso e o resultado está disponível. Nesse estado, a Promise é considerada "resolvida".
- Rejected: A operação assíncrona falhou e ocorreu um erro. Nesse estado, a Promise é considerada "rejeitada".

Promisses - Sintaxe

A sintaxe básica de uma Promise é a seguinte: const minhaPromise = new Promise((resolve, reject) => {
// Lógica assíncrona
});

Promisses - Exemplo

```
function multiplicaPorDois(valor) {
return new Promise((resolve, reject) => {
if (typeof valor === 'number') {
resolve(valor * 2); } else {
reject(new Error('O argumento não é um número.'));
}
});
}
multiplicaPorDois(4).then(result => {
console.log(result);}).catch(error => {
console.error(error.message);
});
```

Obs 1: cuidado com as aspas

Obs 2: altere a chamada passando um texto

<Fic_Dev>
Programador de Sistemas

Promisses - Exemplo

- O método then() é usado quando a Promise é resolvida com sucesso e recebe uma função de retorno de chamada que será executada com o resultado.
- O método catch() é usado para tratar erros e recebe uma função de retorno de chamada que será executada quando ocorrer um erro.

Promisses - Fetch

Um exemplo prático de uso de Promises é uma requisição assíncrona a uma API usando o **fetch()**:
Sintaxe:

```
fetch('https://api.example.com/data')
.then((response) => response.json())
.then((data) => {
// Tratar os dados recebidos
})
.catch((error) => {
// Lidar com o erro da requisição
});
```

Obs: esse exemplo fará sentido mais adiante na disciplina...

Promisses - Fetch - Concluindo

- O fetch() retorna uma Promise que é encadeada com then() para processar a resposta da requisição no formato JSON. Em seguida, outro then() é usado para tratar os dados obtidos. Se ocorrer algum erro durante a requisição, o catch() captura o erro e lida com ele.
- As Promises fornecem uma maneira mais estruturada e eficiente de lidar com operações assíncronas em JavaScript, permitindo um código mais legível e manutenível. Elas são amplamente utilizadas em aplicações modernas para lidar com fluxos assíncronos e facilitar a programação assíncrona.

ASYNC/AWAIT - Introdução

- O **async/await** é uma construção sintática introduzida no ECMAScript 2017 (ES8) para lidar com programação assíncrona de forma mais síncrona e legível em JavaScript.
- Ela é construída em cima do sistema de Promises e oferece uma maneira mais fácil e intuitiva de escrever código assíncrono.
- Ao utilizar **async/await**, você pode escrever código assíncrono como se estivesse escrevendo código síncrono, sem a necessidade de encadear várias chamadas de then() ou catch().
- Ele funciona em conjunto com funções assíncronas e retorna uma Promise, permitindo aguardar a conclusão de operações assíncronas antes de prosseguir.

ASYNC/AWAIT

- A palavra-chave async é usada para declarar uma função assíncrona, que pode conter uma ou várias operações assíncronas.
- Essa função sempre retorna uma Promise. Aqui está um exemplo de uma função assíncrona simples:

Sintaxe:

```
async function exemplo() {
try {
const resultado1 = await operacaoAssincrona1();
const resultado2 = await operacaoAssincrona2(resultado1);
// Realizar mais operações...
return resultadoFinal;
} catch (erro) {
// Lidar com erros
}
}
```

ASYNC/AWAIT

- Nesse exemplo, a função exemplo() é declarada como assíncrona. Ela utiliza await para aguardar a conclusão de cada operação assíncrona antes de prosseguir para a próxima linha.
- Dessa forma, o código é executado de maneira sequencial e parecida com código síncrono, facilitando o entendimento.
- É importante notar que o uso de await só é permitido dentro de uma função assíncrona. Além disso, a expressão após o await deve ser uma Promise.
- Se a Promise for resolvida com sucesso, o valor resolvido será atribuído à variável. Se a Promise for rejeitada, uma exceção será lançada e pode ser capturada usando um bloco try/catch.

ASYNC/AWAIT

Em resumo, async/await é uma forma mais elegante e fácil de ler e escrever código assíncrono em JavaScript. Algumas situações comuns em que você pode querer usar async/await em vez de callbacks ou promessas são:

- Fazer requisições assíncronas de dados em uma API;
- Ler ou gravar arquivos em um sistema de arquivos;
- Acessar bancos de dados;
- Executar tarefas assíncronas em segundo plano;
- Executar tarefas assíncronas complexas que dependem de outras operações assíncronas.

Conclusão

- Chegamos ao final dessa aula, que complementou a aula anterior sobre Funções. Vimos as funções anônimas, de seta e promisses.
- Além disso, também tratamos do ASYNC/AWAIT que são uma forma de se escrever código assíncrono em JS.
- Todos estes recursos nos darão suporte para a continuidade da disciplina, principalmente na construção de Web API's.
- **Vamos ao Desafio do Dia!**

<Fic_Dev>
Programador de Sistemas