Boas-vindas!

Esteja confortável, pegue uma água
e se acomode em um local tranquilo
que já começamos.

CODERHOUSE

Esta aula será

- gravada

Resumo
da última aula

- ✓ Familiarizar-se com estruturas e conceitos fundamentais ao programar usando JavaScript
- ✓ Conhecer as características do ECMAScript
- ✓ Aplique os conceitos incorporados no desenvolvimento do backend

Perguntas?

Aula 03. BACKEND

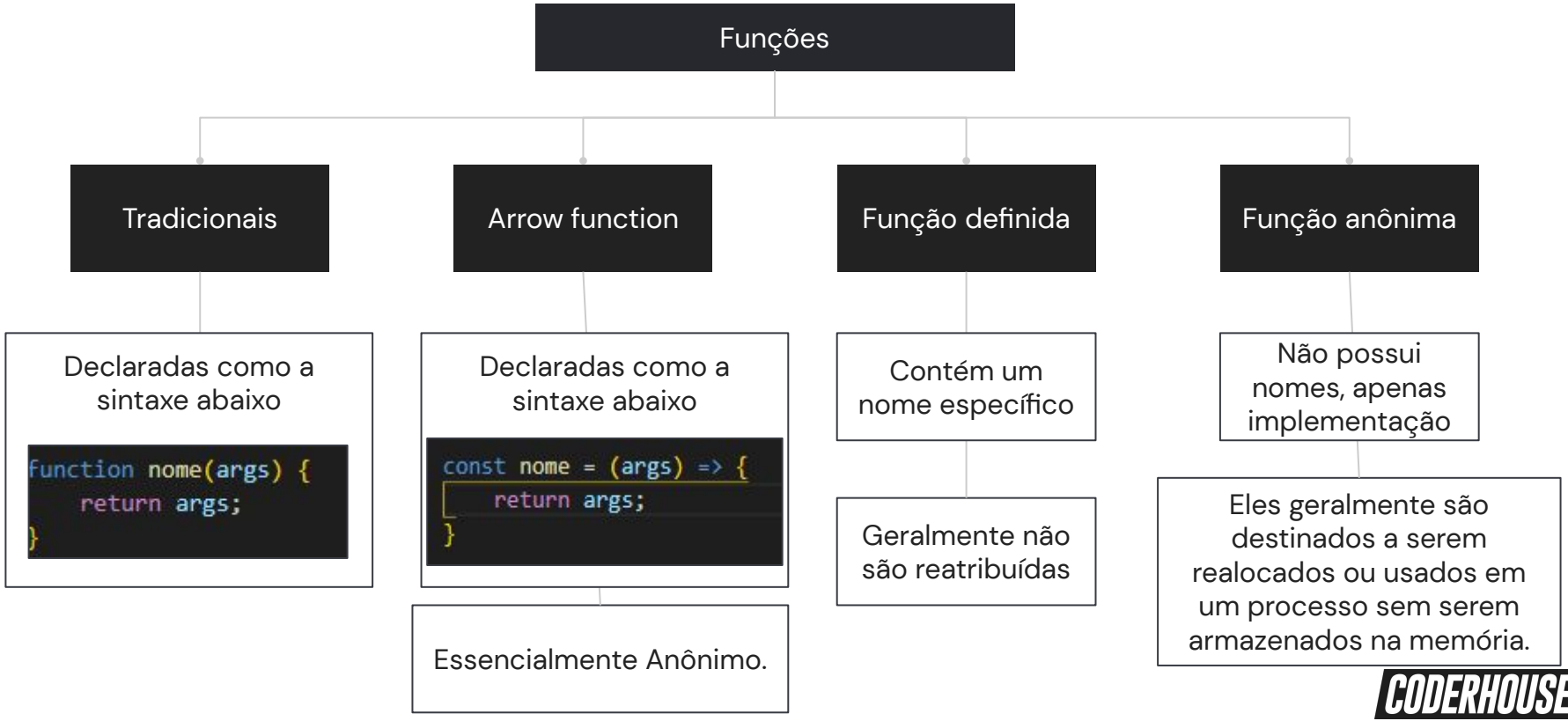
Programação Síncrona e Assíncrona

CODERHOUSE

Objetivos da aula

- Revisar funções em Javascript
- Entenda um callback e como as funções se relacionam com ele.
- Usando promessas em Javascript
- Entenda a diferença entre programação síncrona e assíncrona.

Mapa de conceitos - Javascript



CODERHOUSE

Funções em Javascript

Para recordar...

Lembre-se de que as funções são blocos de código que podem ser chamados em diferentes momentos da execução do nosso programa. Elas podem ter um identificador ao serem declaradas ou ser anônimas. Além disso, elas são **reatribuíveis**.

PARA PENSAR

Por que desejaríamos reatribuir uma função?

Por que iríamos querer usar uma função sem defini-la primeiro?

Compartilhe no chat suas opiniões

Enviando uma função para outra função - CALLBACKS

Callbacks

Um callback é uma função como qualquer outra, a diferença é que ela é passada como parâmetro (argumento) para ser usada por outra função.

Isso permite que as funções executem operações adicionais dentro delas mesmas.

Quando passamos um callback, é porque nem sempre sabemos o que queremos que seja executado em cada caso da nossa função.

Callbacks

Alguns exemplos em que você usou callbacks (embora não acredite) são:

- ✔ O método onClick no frontend
- ✔ O método forEach
- ✔ O método map ou filter

Exemplo de map com callback

Utilizaremos a função **map**, enfatizando o callback, para entendê-lo de forma convencional. Será explicado o funcionamento interno da função map para analisar quando o callback é utilizado.

callback.js > ...

```
1
2   //vamos usar este array de teste
3   let valoresOriginais = [1,2,3,4,5];
4   //Estamos acostumados a ler uma função de mapa da seguinte maneira:
5   let novosValores = valoresOriginais.map(x=>x+1); //novos valores terão: [2,3,4,5,6]
6   //*Porém, o que colocamos dentro da função map é uma função (seta, mais especificamente), que indica que o valor * do número que está no array é somado a 1.
7   //*Temos sempre que somar 17? Não! Podemos entrar com a operação que quisermos, e o map irá executá-la internamente!
8   //"
9
10  let outrosValores = valoresOriginais.map(x=>x*2); // outrosValores terão: [2,4,6,8,10]
11  let maisValores = valoresOriginais.map(x=>"a"); //maisValores terão: ["a","a","a","a","a"]
12  /* Notamos que, não importa o quantas vezes mude função que você está colocando dentro do map, map está feita PARA RECEBER UMA FUNÇÃO COMO PARÂMETRO e pode executá-la quando julgar apropriado. Agora. Se estruturarmos o callback por fora. */
13  const funcionCallback = (value) => { //Função que avalia se o valor do array é um número par
14      if(value%2===0){
15          return value;
16      } else {
17          return "não é par";
18      }
19  }
20  const verificaPares = valoresOriginais.map(funcionCallback); //Estou passando a função inteira como um argumento para o mapa de função
21  console.log(verificaPares) // o resultado será: ["não é par",2,"não é par",4,"não é par" ];
22
```

CODERHOUSE

Exemplo de decomposição de função de map

A função do map será decomposta para poder analisá-la por dentro. O objetivo é localizar em que ponto a função "map" chamaria internamente o retorno de chamada.

JS callback2.js > Array > minhaPropriaFuncionMap

```
1  //Usaremos el arreglo de prueba
2  let arrayDeProva = [1,2,3,4,5];
3  const minhaFuncionMap = (array,callback) => {
4      let novoArray = [];
5      for(let i=0;i<array.length;i++){
6          let novoValor = callback(array[i]); // Observe que o callback recebido por parâmetro está sendo executado nesta linha
7          novoArray.push(novoValor);
8      }
9      return novoArray;
10 }
11
12
13 //vamos comparar nossa nova função COM UM CALLBACK e a função map
14 let novoArrayProprio = minhaFuncionMap(arrayDeProva,x=>x*2); //O novo array será: (2,4,6,8,10)
15 let novoArrayComMap = arrayDeProva.map(x=>x*2); //O array será: [2,4,6,8,10]||
16 /**
17 * Note que não há diferença. Acabamos de recriar a função map para entender seu funcionamento interno e ver ONDE ela está usando o * callback que enviamos como parâmetro
18 */
19
20 /**
21  * EXTRA: Se quisermos que a função seja executada no mesmo array e não tenhamos que passá-la como parâmetro, devemos adicionar nossa nova função * no protótipo do objeto Array
22  */
23
24 Array.prototype.minhaPropriaFuncionMap = function(callback) {
25     let novoArray = [];
26     for(let i=0;i<array.length;i++){
27         let novoValor = callback(array[i]); // Observe que o callback recebido por parâmetro está sendo executado nesta linha
28         novoArray.push(novoValor);
29     }
30     return novoArray;
31 }
32
33 let arrayProva = [1,2,3,4,5];
34 let novosValores = arrayProva.minhaPropriaFuncionMap(x=>x+1);
```

Exemplo de callback com operações

Quatro funções serão criadas**: somar, subtrair**, **multiplicar** e **dividir**.
Além disso, será fornecida outra função de operação, que receberá como callback qualquer uma dessas três funções para executá-la.

JS callback3.js > realizarOperacao

```js
const somar = (numero1, numero2) =>numero1+numero2;
const subtrair = (numero1, numero2) => numero1-numero2;
const multiplicar = (numero1, numero2) => numero1*numero2;
const dividir = (numero1, numero2) => numero1/numero2;

const realizarOperacao = (numero1, numero2, callback) => {
    console.log("Vou fazer uma operação, não sei qual, mas vou!");
    let resultado = callback(numero1, numero2);
    /**
    * Não sabemos qual das 4 funções será, mas isso não nos importa, apenas executamos e retornamos o resultado.
    */
    console.log(`O resultado da operação, que eu não sabia o que era, é: ${resultado}`);
}

realizarOperacao(2,5, somar); // O resultado da operação, que eu não sabia o que era, é: 7
realizarOperacao(2,5, subtrair);// O resultado da operação, que eu não sabia o que era, é: -3
realizarOperacao(2,5, multiplicar); // O resultado da operação, que eu não sabia o que era, é: 10
realizarOperacao(2,5,dividir) //O resultado da operação, que eu não sabia o que era, é: 0.4

/**
* Vamos analisar, performOperation recebe uma função de callback e executa ela dentro, maaaaaaaassss... Ele não tem ideia do que a função faz, apenas
* o executa! Portanto devemos sempre ter muito cuidado com o que passamos como callback, pois no caso de passar uma função que não é
* compatível com os valores com os quais a função está trabalhando, poderíamos quebrar o código para o qual passamos o callback.
*/
```

Callbacks: algumas convenções

- ✓ O callback é sempre o último parâmetro.
- ✓ O callback geralmente é uma função que recebe dois parâmetros.
- ✓ A função chama o callback quando termina de executar todas as suas operações.
- ✓ Se a operação **foi bem-sucedida**, a função chamará o callback passando null como primeiro parâmetro e se gerou algum resultado será passado como segundo parâmetro.
- ✓ Se a operação **resultar em erro**, a função chamará o callback passando o erro retornado como primeiro parâmetro.

documentação de Callback

Exemplo de convenção

Do lado do **callback**, essas funções devem saber como lidar com os parâmetros. Por esse motivo, encontraremos com frequência esta estrutura.

```
const exemploDeCallback = (error, result) => {
    if (error) {
        // faz algo com o erro
    } else {
        // faz algo com o result
    }
}
```

CODERHOUSE

Callbacks aninhados

Callbacks aninhados

Em algum momento, o mundo do trabalho exige que você faça mais do que apenas uma soma ou subtração. Encontraremos processos que exigem operações de várias etapas.

Se trabalharmos com callbacks, podemos encadear um conjunto de operações sequenciais.

Assim, um callback pode chamar outro callback, e este pode chamar outro callback, e assim por diante...

```
// Callback Hell


a(function (resultsFromA) {
    b(resultsFromA, function (resultsFromB) {
        c(resultsFromB, function (resultsFromC) {
            d(resultsFromC, function (resultsFromD) {
                e(resultsFromD, function (resultsFromE) {
                    f(resultsFromE, function (resultsFromF) {
                        console.log(resultsFromF);
                    })
                })
            })
        })
    })
});

```

CODERHOUSE

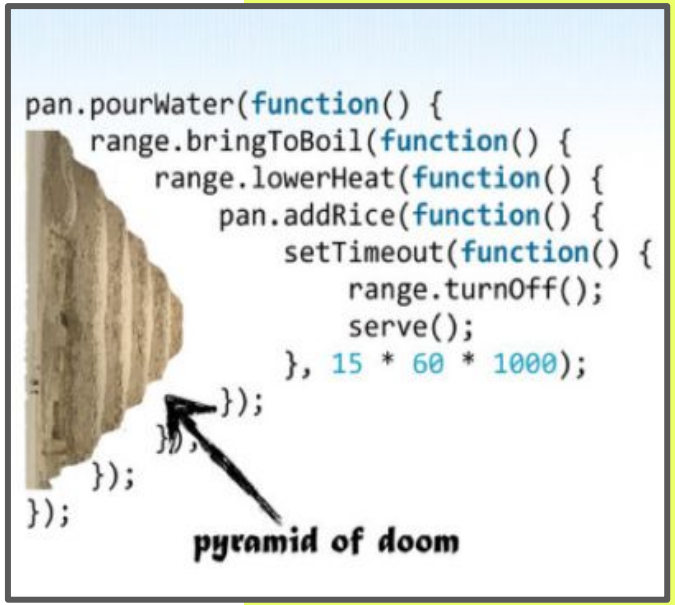
Exemplo de callback animado

```
const copiarArquivo = (nomeArquivo, callback) => {
    buscarArquivo(nomeArquivo, (error, arquivo) => {
        if (error) {
            callback(error);
        } else {
            lerArquivo(nomeArquivo, 'utf-8', (error, texto) => {
                if (error) {
                    callback(error);
                } else {
                    const nomeCopia = nomeArquivo + '.copy';
                    escrevaArquivo(nomeCopia, texto, (error) => {
                        if (error) {
                            callback(error);
                        } else {
                            callback(null);
                        }
                    })
                }
            })
        }
    })
}
```

CODERHOUSE

⚠️ Importante!

Quanto mais callbacks aninharmos (dependendo do tamanho do processo), mais formamos uma pirâmide horizontal. Em nosso código, isso é conhecido como **CALLBACK HELL** (também conhecido como Pyramid of Doom por sua forma).
Se você está trabalhando com callbacks e seu código começa a assumir essa forma... Muito cuidado, é preciso mudar de estratégia!



CODERHOUSE

PARA PENSAR

Se callbacks podem apresentar esse callback Hell, então você não deveria usá-los?

Quando usamos callbacks?

Compartilhe no chat suas opiniões

Agora sim, vamos conhecer uma estratégia diferente para resolver o problema do Callback Hell.

CODERHOUSE

Promessas

Promessas

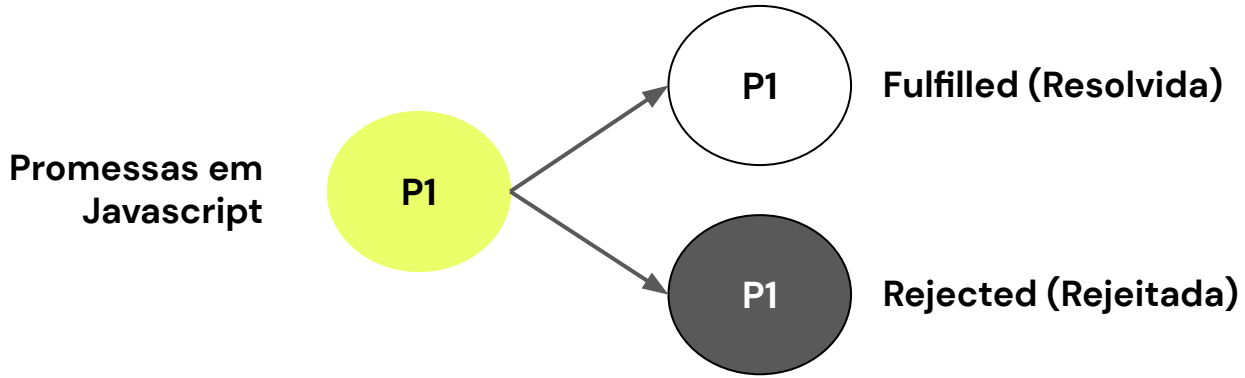
É un objeto especial que **nos permitirá encapsular uma operação**, que reagirá a duas possíveis situações dentro de uma promessa:

- ✔ O que devo fazer se a promessa for cumprida?
- ✔ O que devo fazer se a promessa não for cumprida?

Estados de uma promessa

Uma promessa funciona de forma muito semelhante ao mundo real.

Quando prometemos algo, é uma promessa em estado pendente (**pending**), não sabemos quando essa promessa será cumprida. No entanto, quando chega o momento, somos notificados se a promessa foi cumprida **(Fulfilled**, também encontramos como **Resolvido**) ou talvez, apesar do tempo, no final somos notificados de que a promessa não pôde ser cumprida, foi rejeitada (**Rejected**).



Exemplo de criação de uma promessa

Uma promessa será criada, enfatizando resolver casos e rejeitar casos.

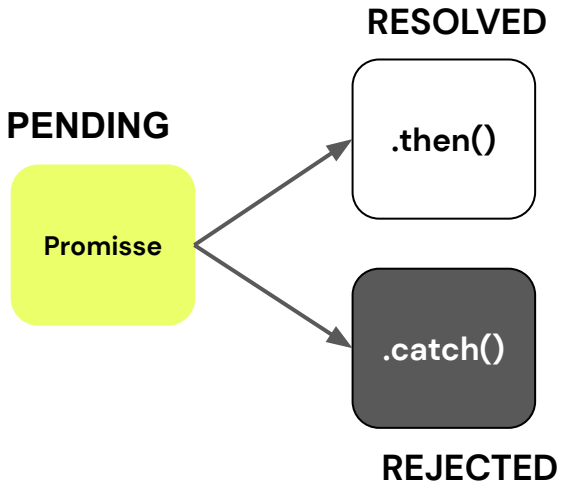
promises1.js > dividir

```
const dividir = (dividendo,divisor) => {
    return new Promise((resolve, reject)=>{//Observe que ao criar uma promessa, estamos passando um callback com dois parâmetros: resolver e rejeitar
        if(divisor===0) {
            reject('Não pode dividir por zero')
        /**
        * Rejeitamos a operação porque não é possível trabalhar com divisão por zero, não posso cumprir a promessa que
        * fiz ao usuário sobre a divisão de seus números.
        */
        } else {
            resolve(dividendo/divisor);
            /**
            * Se os valores forem válidos, então sim posso cumprir a promessa que fiz ao usuário de dividir seus números, portanto,
            * usaremos o valor
            */
        }
    })
}
```

Usando promessas

Agora que entendemos que existem duas maneiras de resolver uma promessa (Resolvida/Cumprida ou Rejeitada), precisamos aprender como usar esses dois estados.

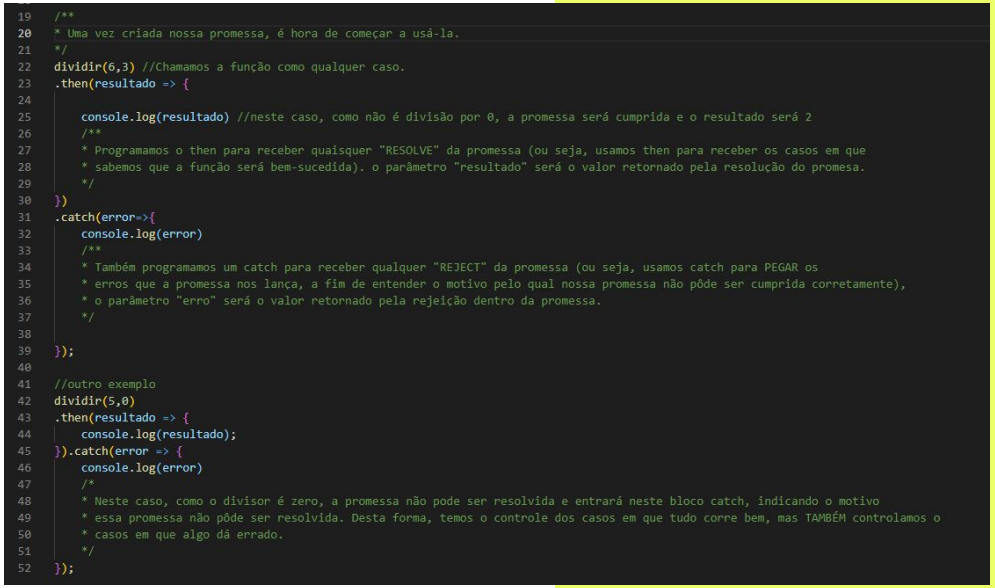
- ✔ Vamos executar a função que acabamos de criar para que a promessa seja executada.
- ✔ Usaremos o operador .then para receber o caso em que a promessa for cumprida
- ✔ Usaremos o operador .catch para receber o caso em que a promessa não foi cumprida.



Documentação de Promisses

Exemplo de uso de una promessa

Usaremos a função “dividir” que foi criada no exemplo anterior, para nos aprofundarmos nos operadores .then e .catch



```
/**
* Uma vez criada nossa promessa, é hora de começar a usá-la.
*/
dividir(6,3) //Chamamos a função como qualquer caso.
.then(resultado => {

    console.log(resultado) //neste caso, como não é divisão por 0, a promessa será cumprida e o resultado será 2
    /**
    * Programamos o then para receber quaisquer "RESOLVE" da promessa (ou seja, usamos then para receber os casos em que
    * sabemos que a função será bem-sucedida). o parâmetro "resultado" será o valor retornado pela resolução do promesa.
    */
})
.catch(error=>{
    console.log(error)
    /**
    * Também programamos um catch para receber qualquer "REJECT" da promessa (ou seja, usamos catch para PEGAR os
    * erros que a promessa nos lança, a fim de entender o motivo pelo qual nossa promessa não pôde ser cumprida corretamente),
    * o parâmetro "erro" será o valor retornado pela rejeição dentro da promessa.
    */

});

//outro exemplo
dividir(5,0)
.then(resultado => {
    console.log(resultado);
}).catch(error => {
    console.log(error)
    /*
    * Neste caso, como o divisor é zero, a promessa não pode ser resolvida e entrará neste bloco catch, indicando o motivo
    * essa promessa não pôde ser resolvida. Desta forma, temos o controle dos casos em que tudo corre bem, mas TAMBÉM controlamos o
    * casos em que algo dá errado.
    */
});
```

CODERHOUSE

Promessa encadeada

Sempre que colocamos um return dentro de um **“.then()”**, o resultado se torna automaticamente outra promessa e pode ser encadeado com outro **.then()**, e assim sucessivamente até o final do processo.

Se em algum dos “**.then()**” der errado, basta um catch para pegá-lo.

```
new Promise(function (resolve, reject) {
    setTimeout(() => resolve(1), 1000); //(*)
})
.then(result => { // (**)
    console.log(result); // 1
    return result * 2;
}).then(result => { // (***)
    console.log(result); // 4
    return result * 2;
}).then(result => { // (****)
    console.log(result); // 8
    return result * 2;
});

//1) A promessa inicial se resolve em 1 segundo (*)
//2) Então é chamado o controlador .then (**)
//3) E o valor devolvido é passado para o próximo controlador .then (***)
```

CODERHOUSE

Break

5 minutos e voltamos!

CODERHOUSE

Break

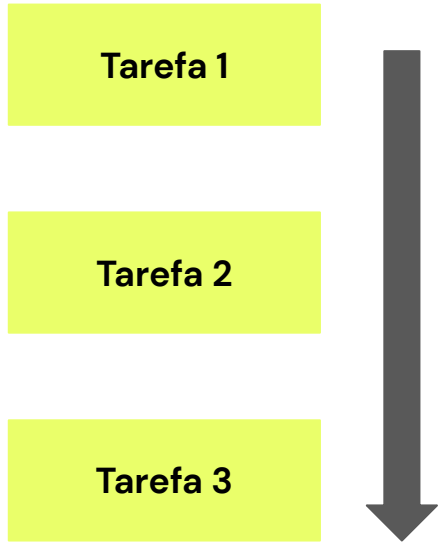
10 minutos e voltamos!

CODERHOUSE

Sincronismo
X
Assincronismo

Sincronismo

Há poucos dias, quando você foi ensinado a programar, você entendeu que as **instruções eram executadas em cascata**, ou seja, que a tarefa 1 tinha que terminar para que a execução da tarefa 2 pudesse começar, e a tarefa 2 terminar para executar a tarefa 3. , etc



Síncrono

Exemplo execução síncrona

A qualquer momento, apenas as instruções para uma das funções estão sendo executadas por vez. Ou seja, você deve terminar uma função antes de continuar com a outra.
O fim de uma função marca o início da próxima, e o fim desta, o início da seguinte, e assim por diante, descrevendo uma sequência que ocorre em uma única linha do tempo.

```
function funA() {
    console.log(1);
    funB();
    console.log(2);
};
function funB() {
    console.log(3);
    funC();
    console.log(4);
};
function funC() {
    console.log(5);
};

funA();

//Ao executar a função funA()
//o console.log irá exibir o resultado da seguinte maneira
1
3
5
4
2
```

CODERHOUSE

⚠️ Importante!

As operações síncronas são bloqueantes, isso significa que as **outras tarefas não podem começar a ser executadas até que a primeira termine de ser executada.**

Documentação sobre sincronismo e assincronismo

Assincronismo

Se o que procuramos é que as tarefas funcionem "em paralelo", então devemos encontrar uma maneira de programar instruções assíncronas, o que significa que cada uma seguirá o fio de resolução que considera seu ritmo.

Você deve ter cuidado ao usá-los, pois:

- ✓ Não controlamos quando vai acabar, apenas quando começa.
- ✓ Se uma tarefa depender do resultado de outra, haverá problemas, pois ela aguardará sua execução em paralelo

Exemplo de execução assíncrona

No exemplo, a execução normal do programa não é bloqueada e é permitido continuar executando.
A execução da operação de escrita "começa" e imediatamente passa o controle para a próxima instrução, que escreve a mensagem de conclusão na tela.
Quando a operação de escrita termina, ele executa o callback que informará na tela que a escrita foi bem-sucedida.

```
const escreverArquivo = require('./escreveArquivo.js');

console.log('Inicio do programa');

// o criador desta função a definiu
// como não bloqueante, recebe um callback que
// executará apenas após escrever o arquivo
escreverArquivo('Olá mundo', () => {
    console.log('terminou de escrever o arquivo');
});

console.log('final do programa');

// o resultado exposto na tela será:
// > Inicio do programa
// > final do programa
// > terminou de escrever o arquivo
```

⚠️ Importante!

As operações assíncronas são sem bloqueio, o que significa que **as tarefas podem ser executadas em paralelo e não esperar por outras tarefas.** Assim, a tarefa número 3 poderia terminar antes mesmo da tarefa número 1.

Documentação sobre sincronismo e assincronismo

Async / Await

Problemas com .then e .catch

Quando precisamos de mais de uma operação para poder executar algo assíncrono, o uso de uma promessa por si só não é suficiente, mas precisamos de um ambiente completo para poder executar essas operações. Então, neste caso, serve apenas para encadear o promessas e obter seus resultados, mas não nos permite um ambiente assíncrono completo para trabalhar, então nos obriga a trabalhar TUDO dentro desse escopo.

Além disso, o principal problema com .then() e .catch() é seu encapsulamento excessivo, impedindo ou limitando nosso acesso aos recursos de alguns resultados, variáveis, etc.

Async / Await

Surgiu então o **suporte para Async - Await em Javascript**, algumas palavras reservadas que, trabalhando juntas, permitem gerenciar um ambiente assíncrono, resolvendo as limitações de .then() e .catch()

- ✓ Async será colocado no início de uma função, indicando que todo o corpo dessa função deve ser executado de forma assíncrona.
- ✓ Await servirá (como o próprio nome indica) para aguardar o resultado da promessa e extrair seu resultado.

Como essas são operações que podem funcionar bem, **mas também erradas**, é importante incluir o corpo em um bloco try {} catch {}.

Documentação sobre Async e Await

Documentação sobre Async e Await

Async / Await

Explicação sobre como usar uma função assíncrona aplicando async/await.
Usaremos a mesma promessa de divisão com a qual trabalhamos durante a aula.

```
const dividir = (dividendo,divisor) => {
    return new Promise((resolve, reject)=>{//Observe que ao criar uma promessa, estamos passando um callback com dois parâmetros: resolver e rejeitar
        if(divisor===0) {
            reject('Não pode dividir por zero')
        /**
        * Rejeitamos a operação porque não é possível trabalhar com divisão por zero, não posso cumprir a promessa que
        * fiz ao usuário sobre a divisão de seus números.
        */
        } else {
            resolve(dividendo/divisor);
            /**
            * Se os valores forem válidos, então sim posso cumprir a promessa que fiz ao usuário de dividir seus números, portanto,
            * usaremos o valor
            */
        }
    })
}

const funcaoAssincrona = async() =>{
    /**
    * Estamos inicializando um ambiente assíncrono completo, tudo dentro das chaves da função se comportará sem bloqueio com
    * o exterior.
    */
    try{
        //Incluímos a operação a ser realizada em um bloco try, porque sendo uma promessa, PODERIA NÃO SER CUMPRIDA, portanto devemos estar preparados
        let resultado = await dividir(10,5) //Não há mais .then, agora é só ESPERAR pelo resultado da promessa.
        console.log(resultado);
    } catch(error) {
        //O bloco catch é obrigatório após um try{} e funciona da mesma forma que .catch, para capturar erros.
        console.log(error);
    }
}

funcaoAssincrona(); //Como o ambiente de execução assíncrona reside em uma função. você tem que executá-lo no final.
```

CODERHOUSE

Hands on Lab

Duração: **15 minutos**

CODERHOUSE

ATIVIDADE EM SALA

Hands on Lab

Calculadora positiva com promises

Como fazemos? Será criado um conjunto de funções gerenciadas por promessas e um ambiente ASYNCHRONOUS onde podemos testá-las

Defina a função soma:

- ✓ Deve retornar uma promessa que resolva, desde que nenhum parcela seja 0
- ✓ Caso alguma parcela seja 0, rejeite a promessa indicando "Operação desnecessária".
- ✓ Caso a soma seja negativa, rejeite a promessa afirmando “A calculadora deve retornar apenas valores positivos”

ATIVIDADE EM SALA

Hands on Lab

Defina a função de subtração:

- ✓ Deve retornar uma promessa que resolve desde que nenhum dos valores seja 0
- ✓ Se o minuendo ou o subtraendo for 0, rejeite a promessa indicando "Operação inválida”
- ✓ Caso o valor da subtração seja menor que 0, rejeite a promessa indicando "A calculadora só pode retornar valores positivos"

ATIVIDADE EM SALA

Hands on Lab

Defina uma função de multiplicação:

- ✓ Deve retornar uma promessa que resolve desde que nenhum dos fatores seja negativo
- ✓ Se o produto for negativo, rejeite a oferta afirmando "A calculadora só pode retornar valores positivos

Defina a mesma função de divisão usada nesta classe.

Definir uma função assíncrona “cálculos” e realizar testes usando async/await e try/catch

Perguntas?

Como foi a aula?

1

Que bom

O que foi super legal na aula e podemos sempre trazer para as próximas?

2

Que pena

O que você acha que não funcionou bem e precisamos melhorar?

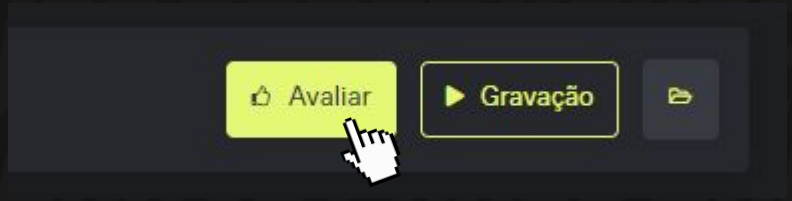
3

Que tal

Qual sugestão deveríamos tentar em próximas aulas?

CODERHOUSE

O que você achou da aula?



Seu feedback vale pontos para o Top 10!! 😎

Deixe sua opinião!

1. Acesse a plataforma
2. Vá na aula do dia
3. Clique em Avaliar

CODERHOUSE

Resumo
da aula de hoje:

- ✓ Familiarizar-se com estruturas e conceitos fundamentais ao programar usando JavaScript
- ✓ Conhecer as características do ECMAScript
- ✓ Aplique os conceitos incorporados no desenvolvimento do backend

Obrigado por estudar conosco!