# JavaScript

CJR

# Aula 5

# JavaScript assíncrono

referência ↗

# Lançamento de exceções

- Podemos usar a palavra reservada throw para lançar uma exceção (um erro).
- A execução da função atual vai parar (as instruções após o throw não serão executadas), e o controle será passado para o primeiro bloco catch na pilha de chamadas. Se nenhum bloco catch existe entre as funções "chamadoras", o programa vai terminar.

```
function sum(a, b) {
    a = Number(a)
    b = Number(b)

    if (Number.isNaN(a) || Number.isNaN(b)) throw 'Invalid numbers'

    const sum = a + b
    console.log('sum:', sum)

    return sum
}

sum(1, 2) // 3
sum('2', '3') // 5
sum('a', 3) // Uncaught Invalid numbers
```

- É possível (e mais comum) lançar objetos de erro criados com o construtor Error.

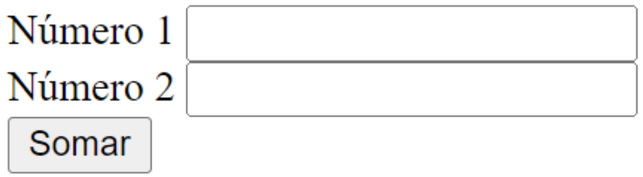
```
function sum(a, b) {
    a = Number(a)
    b = Number(b)

    if (Number.isNaN(a) || Number.isNaN(b)) throw new Error('Invalid numbers')

    const sum = a + b
    console.log('sum:', sum)

    return sum
}

sum(1, 2) // 3
sum('2', '3') // 5
sum('a', 3) // Uncaught Error: Invalid numbers
```

# Tratamento de exceções

- Podemos usar um bloco try...catch para capturar exceções. Se o código dentro do try lançar uma exceção, a execução é transferida para o bloco catch.

- Podemos acessar o erro que causou a exceção através do parâmetro que é passado para o bloco catch.

```html
<form>
    <label for="number1"> Número 1 </label>
    <input id="number1" type="text" />
    <label for="number2"> Número 2 </label>
    <input id="number2" type="text" />
    <button type="submit">Somar</button>
</form>
<p>resultado: <span id="result"></span></p>
```




```javascript
const form = document.querySelector('form')
const resultContainer = document.getElementById('result')
const input1 = document.getElementById('number1')
const input2 = document.getElementById('number2')

function showResult(event) {
    event.preventDefault()

    try {
        resultContainer.textContent = sum(input1.value, input2.value)
    } catch (error) {
        resultContainer.textContent = error.message
    }
}

form.addEventListener('submit', showResult)
```

- Também é possível adicionar a cláusula finally ao final de um bloco try...catch. Ela é executada após a execução do bloco try ou do catch, independente se der error ou não.

```
const form = document.querySelector('form')
const resultContainer = document.getElementById('result')
const input1 = document.getElementById('number1')
const input2 = document.getElementById('number2')

function showResult(event) {
    event.preventDefault()

    try {
        resultContainer.textContent = sum(input1.value, input2.value)
    } catch (error) {
        resultContainer.textContent = error.message
    } finally {
        event.target.reset() // limpa o formulário
    }
}

form.addEventListener('submit', showResult)
```

# Temporizadores

# JavaScript assíncrono

- JavaScript é single-thread e síncrono, isso significa que ele executa somente uma instrução por vez, uma seguida da outra.

- Apesar disso, APIs fornecidas pelos ambientes de execução (browser e node) podem executar códigos e realizar tarefas paralelamente a execução principal. Isso evita que o código fique travado em tarefas que demandem muito tempo.

- Algumas dessas APIs são:

    - addEventListener
    - setTimeout e setInterval
    - fetch

# setTimeout

- O método global **setTimeout()** define um cronômetro que executa uma função ou trecho de código especificado assim que o cronômetro expirar.
- O primeiro parâmetro é a função a ser executada e o segundo é o tempo em milissegundos que vai demorar até a função ser executada.
- **setTimeout()** é uma função assíncrona, o que significa que a função timer não pausará a execução de outras funções na pilha de funções.

```
setTimeout(() => {
    console.log('Executado depois de 2 segundos')
}, 2000)

console.log('Executado primeiro')
```

# setInterval

- O método global **setInterval()** repete chamadas de funções com um tempo de espera fixo entre cada chamada.
- Igual o **setTimeout**, **setInterval** recebe como primeiro parâmetro uma função e como segundo parâmetro o tempo de espera em milissegundos entre cada chamada da função.

```
let currentTime = 0

function incrementTime() {
    currentTime++
    console.log('currentTime:', currentTime)
}

setInterval(incrementTime, 1000)
```

# Cancelando timers

- Tanto o **setTimeout** quanto o **setInterval** retornam um identificador que pode ser usado para cancelar o timer.
- Para cancelar um **setTimeout**, utiliza-se o **clearTimeout().**
- Para cancelar um **setInterval**, utiliza-se o **clearInterval().**

```
let currentTime = 0

function incrementTime() {
    currentTime++
}

const intervalId = setInterval(incrementTime, 1000)

const button = document.querySelector('button')
button.addEventListener('click', () => clearInterval(intervalId))
```

# Promises

# async callbacks

- Antigamente, quando se trabalhava com JavaScript assíncrono, utilizava-se callbacks para executar uma ação após uma tarefa demorada ter sido finalizada.
- Exemplos de callback assíncrono é o callback pasado para o addEventListener e para o setTimeout.
- Outro exemplo é o callback passado para a função fs.readFile() do node.

```
const fs = require('fs')

fs.readFile('./file1.txt', 'utf8', (err, data) => {
    if (err) throw err
    console.log('data:', data)
})
```

# Problemas dos async callbacks

- Callback Hell.
- Dificulta a leitura e entendimento do código.
- Inversão de controle, você perde o controle de como e quando o callback será executado quando passada para uma biblioteca de terceiros.
- Tratamento de erros precisa ser feito individualmente em cada callback, criando repetição de código.

# Problemas dos async callbacks

```
try {
    fs.readFile('./file1.txt', 'utf8', (err, data) => {
        if (err) throw err
        fs.readFile(data, 'utf8', (err, data) => {
            if (err) throw err
            fs.readFile(data, 'utf8', (err, data) => {
                if (err) throw err
                fs.readFile(data, 'utf8', (err, data) => {
                    if (err) throw err
                    console.log('data:', data)
                })
            })
        })
    })
} catch (error) {
    console.log(error)
}
```

# Promises

- Promises resolvem todos esses problemas!
- Promise **é um objeto** usado para processamento assíncrono. Um Promise (de "promessa") representa um valor que pode estar disponível agora, no futuro ou nunca.
- Um Promise está em um destes estados:
  - pending (pendente): Estado inicial, que não foi realizada nem rejeitada.
  - fulfilled (realizada): sucesso na operação.
  - rejected (rejeitado): falha na operação.

```
const fs = require('fs/promises')

const data = fs.readFile('./file1.txt', 'utf8')

console.log(data) // Promise { <pending> }
```

# then

- Promises são objetos, isso significa que elas tem propriedades e métodos.
- Um desses métodos e o método **then**, que é chamado quando a promise é resolvida (quando há sucesso).
- Ele recebe uma função e passa pra ela o dado retornado pela promise como parâmetro.
- Ele pode ser pensado como a função de callback passada para o addEventListener.

```
const fs = require('fs/promises')

fs.readFile('./file1.txt', 'utf8').then((data) => console.log(data))
```

# then encadeados

- O then retorna uma promisse, que quando resolvida tem o valor do retorno da função passada como callback, isso permite encadear thens.

```
const fs = require('fs/promises')

fs.readFile('./file1.txt', 'utf8')
    .then((data) => {
        return `data: ${data}`
    })
    .then((data) => console.log(data))
```

```
const fs = require('fs/promises')

fs.readFile('./file1.txt', 'utf8')
    .then((data) => fs.readFile(data, 'utf8'))
    .then((data) => fs.readFile(data, 'utf8'))
    .then((data) => fs.readFile(data, 'utf8'))
    .then((data) => console.log(data))
```

# catch

- Outro método das promises e o **catch**. Ele recebe uma função de callback e a executa quando a promise (ou promises encadeadas) for rejeitada.

```
const fs = require('fs/promises')

fs.readFile('./file1.txt', 'utf8')
    .then((data) => fs.readFile(data, 'utf8'))
    .then((data) => fs.readFile(data, 'utf8'))
    .then((data) => fs.readFile(data, 'utf8'))
    .then((data) => console.log(data))
    .catch((error) => console.log(error))
```

# finally

- Também temos o **finally** que é executado após a promise ser resolvida ou rejeitada.

```
const fs = require('fs/promises')

fs.readFile('./file1.txt', 'utf8')
    .then((data) => fs.readFile(data, 'utf8'))
    .then((data) => fs.readFile(data, 'utf8'))
    .then((data) => fs.readFile(data, 'utf8'))
    .then((data) => console.log(data))
    .catch((error) => console.log(error))
    .finally(() => console.log('finally'))
```

# async/await

- Outro modo de se trabalhar com promises é usando as declarações async...await.
- O operador await é utilizado para esperar por uma Promise. Ele faz a execução de uma função async pausar, para esperar pelo retorno da Promise, e resume a execução da função async quando o valor da Promise é resolvido.
- Sempre quando se utiliza await, tem que se utilizar a palavra async na função que envolve o código com o await.
- Se a Promise for rejeitada, a expressão await invoca uma Exception com o valor rejeitado. Então devemos usar um bloco try...catch em volta dela para capturar a exceção.
- O operador async transforma o retorno da função em uma promise.
- O await faz o código dentro da função parecer síncrono, mas a função como um todo continua sendo assíncrona, pois retorna uma promise.

# async/await

```
const fs = require('fs/promises')

async function readFiles() {
    try {
        const file1 = await fs.readFile('./file1.txt', 'utf8')
        const file2 = await fs.readFile(file1, 'utf8')
        const file3 = await fs.readFile(file2, 'utf8')
        const file4 = await fs.readFile(file3, 'utf8')
        console.log(file4)
    } catch (error) {
        console.log(error)
    }
}

readFiles()
```

# Referências


- https://youtu.be/7Bs4-rqbC0c
- https://youtu.be/li7FzDHYZpc

# Últimos avisos

- Estudem sobre o **spread operator**, **rest operator** e **destructuring assignment**, super importantes!!
- Estudem sobre o Date
- Pratiquem
  - https://github.com/florinpop17/app-ideas
  - https://devchallenge.com.br/challenges
  - https://github.com/coding-horror/basic-computer-games

# É isso, valeu ✌️