JavaScript assíncrono

Um código síncrono tem sua execução na ordem comum das linhas, por outro lado o código assíncrono ocorre quando uma linha é executada foram da ordem comum.

Isso ocorre bastante em situações onde lidamos com tarefas que exigem leituras externas ao código como operações com banco de dados, acesso a APIs ou leitura de outros arquivos.

Mas também pode ser causada de forma planejada como na configuração de um tempo de espera para executar uma função ou espera de um evento HTML acontecer para executar outra função.

Callback, Promise e async await

As formas mais eficazes e com sintaxe mais limpa para tratar eventos assíncronos no JavaScript são as Promises (com uso do .then para usarmos o retorno da ação) e o uso de await em funções criadas com uso da palavra async.

Uma forma que ainda é necessária em algumas funções nativas, mas que acaba poluindo mais o código é o uso de callbacks.

Callbacks

Callbacks podem ser resumidas como funções que são passadas como argumentos para outras funções

Elas podem ser chamadas por alguma ação como um click em um botão HTML, um período de tempo definido, ou a execução de uma função pré-definida.

Temos algumas funções do JavaScript que possuem seus callbacks configurados nativamente, como os métodos de array: filter, find, every, some, reduce.

Na sua criação foram definidos como:
Array.filter(callback)

Uso de Callbacks

No momento de utilizar a função que espera um callback, nós trocamos o parâmetro callback por uma função específica:

Array.filter(callback)

lista.filter((item) => item % 2 == 0)

Callback relacionado a evento HTML

A função addEventListener permite que sejam associados eventos a elementos HTML e executar algo no JavaScript quando o evento ocorrer.

Exemplo: Criar uma função que será executada quando um item for clicado

```
const botao = document.querySelector("button")

botao.addEventListener("click", () => {
     alert("botão clicado")
})
```

Alguns eventos legais para combinar com addEventListener

click: elemento é clicado com o botão esquerdo do mouse.
mouseover: ponteiro do mouse entra no elemento.
mouseout: ponteiro do mouse sai do elemento.
mousedown: botão do mouse é pressionado sobre o elemento.
mouseup: botão do mouse é liberado sobre o elemento.
keydown: a tecla do teclado é pressionada.
keyup: a tecla do teclado é liberada.
keypress: a tecla é pressionada e liberada.

submit: formulário é enviado.
change: valor de um elemento de formulário é alterado.
input: valor de um campo de entrada de texto é alterado.
load: página da web é completamente carregada.
resize: janela do navegador é redimensionada.
scroll: janela é rolada

Callback relacionado a evento HTML

```
const input = document.querySelector('input')
input.addEventListener('mouseover', () => alert("Tira o mouse de mim!"))
```

```
addEventListener('resize', () => alert("Para de me apertar!!"))
```

```
addEventListener('scroll', () => alert("Ta rolando algo aqui!"))
```

Callback depois de um tempo - setTimeout

O setTimeout permite que uma função espere algum tempo em milissegundos para ser executada.

Isso acaba tirando a execução dela do fluxo comum do JavaScript. Sendo assim, as linhas abaixo dele vão ser executadas, enquanto a função vai ser chamada após o tempo definido.

```
setTimeout( callback, tempo)
```

setTimeout - Exemplo

```
console.log('1')

setTimeout(function depoisDeCincoSegundos() {
  console.log('2')
}, 5000)

console.log('3')
```

```
1
3
2
```

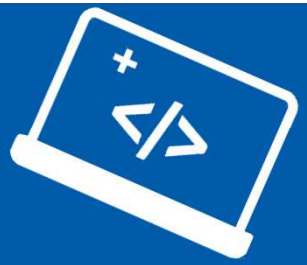
Callback de tempos em tempos - setInterval

A função setInterval repete trechos de código, com um tempo de espera fixo entre cada chamada.

setInterval(callback, tempo)

```
setInterval( () => console.log("rodando código em 2 segundos"), 2000 )
```

Atividades (callbacks)



- Crie uma função dentro de um setTimeout que exibe um alerta com a mensagem “Acesso expirado” após 10 segundos.

- Crie uma função dentro de um setInterval que exibe um prompt perguntando “Tem alguém ai?” a cada 3 segundos.

Criação de função com Callback

Podemos criar nossa função que espera um callback (não é um costume)

```
function nome(parametro, callback) {
    // código necessário
    console.log(`Olá ${parametro}`)
    callback()
}
```

Qual é o valor do callback?

Na hora da criação nós não sabemos, mas quem chamar a função vai passar o que é.

Uso de Callbacks

O que for passado como segundo parâmetro vai substituir a linha 4.

```
function nome(parametro, callback) {
    // código necessário
    console.log(`Olá ${parametro}`)
    callback()
}
```

```
nome("mundo", () => console.log("executando o callback") )
```

Na prática a função vira o seguinte:

```
function nome("mundo", callback) {
  // código necessário
  console.log(`Olá mundo`)
  console.log("executando o callback")
}
```

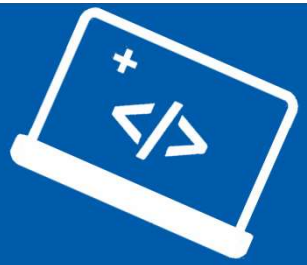
Uso de Callbacks com parâmetros

Podemos passar parâmetros na execução do callback também:

```
function processaAposReceberNome(callback) {
    let nome = prompt("Qual é o seu nome?")
    callback(nome)
}

processaAposReceberNome(nomeQueVaiReceberNaExecucao => {
    alert(`Olá ${nomeQueVaiReceberNaExecucao}`)
})
```

Atividades (callbacks)



- Crie uma função que recebe um callback como argumento:

  Dentro dela crie uma variável nome para receber um nome via prompt e em seguida execute o callback.

  Realize 3 chamadas para essa função passando callbacks diferentes em cada uma, por exemplo:
  1- Passar uma função que exibe um alerta com `Olá ${nome}`
  2- Passar uma função que exibe um alerta com `Feliz aniversário ${nome}`
  3- Passar uma função que exibe um alerta com `Feliz natal ${nome}`

Outros eventos assíncronos e callback hell

Um evento assíncrono muito importante é a chamada para APIs externas.

Quando precisamos utilizar dados de uma ação assíncrona em várias outras ações assincronas o uso de callback torna a sintaxe bastante poluída, podendo formar um inferno de callbacks.

```
inserirBanco(novoUsuario, (resultadoInsercao) => {
    console.log(resultadoInsercao)
    consultarBanco(resultadoInsercao, (resultadoConsulta) => {
        console.log(resultadoConsulta)
        atualizarBanco(resultadoConsulta, (resultadoAtualizacao) => {
            console.log(resultadoAtualizacao)
            consultarBancoNovamente(resultadoAtualizacao, (resultadoConsultaNova) => {
                alert(resultadoConsultaNova)
            })
        })
```

Como uma alternativa para esse problema, foi criado um conceito para tornar as operações assíncronas mais legíveis: as Promises

Um aperitivo das Promises com async await

```
inserirBanco(novoUsuario, (resultadoInsercao) => {
    console.log(resultadoInsercao)
    consultarBanco(resultadoInsercao, (resultadoConsulta) => {
        console.log(resultadoConsulta)
        atualizarBanco(resultadoConsulta, (resultadoAtualizacao) => {
            console.log(resultadoAtualizacao)
            consultarBancoNovamente(resultadoAtualizacao, (resultadoConsultaNova) => {
                alert(resultadoConsultaNova)
            })
        })
    })
})
```

```
const resultadoInsercao = await inserirBanco(novoUsuario)
const resultadoConsulta = await consultarBanco(resultadoInsercao)
const resultadoAtualizacao = await atualizarBanco(resultadoConsulta)
const resultadoConsultaNova = await consultarBancoNovamente(resultadoAtualizacao)
alert(resultadoConsultaNova)
```

Promises

Uma promise é um objeto retornado por uma função assíncrona, contendo o estado atual da operação.

Ela pode ser criada (forma menos comum) ou ser resultado de uma operação utilizando APIs nativas ou bibliotecas de terceiros (forma mais comum).

Como operação assíncrona não é finalizada instantaneamente, o objeto da promise oferece métodos/funções para tratar o possível sucesso ou falha da operação no momento da sua finalização.

Promises – Exemplo 1

Para chamar uma API externa no JavaScript podemos utilizar a função nativa fetch, que retorna uma promise.

```
const promessa = fetch("link-de-uma-API")
console.log(promessa)
```

```
const promessa = fetch("https://jsonplaceholder.typicode.com/albums")
console.log(promessa)
```

A operação de fetch inicialmente retorna: Promise {<pending>}, nos dizendo que o objeto Promise tinha estado com valor "pending" ("pendente") naquele momento.

Possíveis estados de uma Promise

pending: Promise foi criada e ainda não foi concluída.

fulfilled: a função assíncrona foi concluída com sucesso, podendo acionar a função then()

rejected: a função assíncrona falhou, podendo acionar a função catch()

settled: completa, independente de ter sido fulfilled ou rejected.

Promises – capturar o retorno com sucesso

Para conseguirmos um retorno após a conclusão da Promise, utilizamos a função then().

Quando a operação tiver conclusão, receberemos um objeto com a resposta da API.

```
fetch("https://jsonplaceholder.typicode.com/albums")
.then(dados => console.log(`Retorno com sucesso, status: ${dados.status}`))

OU

const promessa = fetch("https://jsonplaceholder.typicode.com/albums")
promessa.then(dados => console.log(`Retorno com sucesso, status: ${dados.status}`))
```

Promises – capturar o retorno com erro

A API fetch() pode lançar um erro por vários motivos (por exemplo, porque não havia conexão de rede ou a API não foi encontrada).

Para lidarmos com esses erros de promise, utilizaremos uma função além do then(), que é o catch(), servindo para realizarmos uma tratativa quando a operação assíncrona falhar.

```
const promessa = fetch("https://jsonplaceholder.typicode.com/albums")

promessa
.then(dados => console.log(`Retorno com sucesso, status: ${dados.status}`))
.catch(erroFetch => console.log(`Erro na resposta: ${erroFetch}`))
```

Promises – realizar uma ação depois do then ou catch

Caso seja necessário dar um retorno após o then e o também o catch, independente de qual ocorrer, podemos utilizar a função finally ao final:

```
const promessa = fetch("https://jsonplaceholder.typicode.com/albums")

promessa
.then(dados => console.log(`Retorno com sucesso, status: ${dados.status}`))
.catch(erroFetch => console.log(`Erro na resposta: ${erroFetch}`))
.finally(() => console.log('Resposta final independente'))
```

Promises – combinação de promises

No retorno da promessa do fetch, teremos uma propriedade body, que para ser lido precisa outra função: json(), que também retorna uma promise, dessa forma teremos que utilizar um segundo then()

```
const promessa = fetch("https://jsonplaceholder.typicode.com/albums")

promessa
.then(retornoFetch => retornoFetch.json())
.then(retornoJson => console.log(retornoJson))
```

No primeiro then, iremos utilizar a resposta da primeira promessa (retorno de fetch) e executar a função resposta.json() que é a segunda promessa.
Com isso, teremos o segundo then que possuirá o resultado da segunda promessa.

Async await – Promises mais legíveis

Podemos trocar o then para os retornos das promises pela palavra await e usá-lo em uma função que utiliza o fetch, utilizando o termo async function em vez de function.

```
async function minhaFuncao() {
    const retornoFetch = await fetch("https://jsonplaceholder.typicode.com/albums")
    const retornoJson = await retornoFetch.json()
    console.log(retornoJson)
}

minhaFuncao()
```

Async await - lidando com erros

Para capturarmos erros continuamos utilizando o catch, porém precisamos incluir todo o código que conterá a execução comum dentro de um bloco chamado try e o tratamento do catch em um outro bloco.

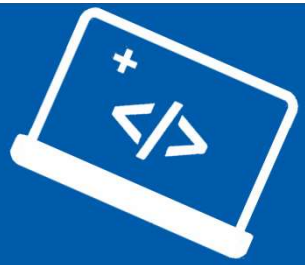
```
async function minhaFuncao() {
  try {
    const retornoFetch = await fetch("https://jsonplaceholder.typicode.com/albums")
    const retornoJson = await retornoFetch.json()
    console.log(retornoJson)
  } catch (erro) {
    console.error(`Ocorreu um erro:`, erro)
  }
}
minhaFuncao();
```

Async await - lidando com erros

Podemos incluir um terceiro bloco de finally para realizar uma operação após a execução do try e/ou do catch.

```
async function minhaFuncao() {
  try {
    const retornoFetch = await fetch("https://jsonplaceholder.typicode.com/albums")
    const retornoJson = await retornoFetch.json()
    console.log(retornoJson)
  } catch (erro) {
    console.error(`Ocorreu um erro:`, erro)
  } finally {
    console.log(`Função finalizada`)
  }
}
minhaFuncao();
```

Atividades (promises e async await)



- Use o fetch para buscar a lista de usuários (https://jsonplaceholder.typicode.com/users) e, em seguida, use o método map para armazenar os nomes dos usuários em um Array.

- Utilize o fetch para buscar uma lista de tarefas na API https://jsonplaceholder.typicode.com/todos
  Após isso, utilize o método filter para encontrar as tarefas incompletas. Em seguida, use o método length para contar quantas tarefas estão incompletas.

- Busque a lista de álbuns de fotos de um usuário na API https://jsonplaceholder.typicode.com/albums
  Use o método filter para encontrar os álbuns de um usuário de sua escolha pelo userId e depois map para exibir o título de cada álbum.

Métodos de Promises

Promise.all(): Quando é necessário que várias promises sejam cumpridas, mas elas não dependem umas das outras. Inicia todas juntas após o cumprimento de todas retorna um array com os retornos das promises em uma única promise (precisamos usar o then para acessar).

```
const fetchPromise1 = fetch("API 1")
const fetchPromise2 = fetch("API 2")
const fetchPromise3 = fetch("API 3")
Promise.all([fetchPromise1, fetchPromise2, fetchPromise3])
.then(respostas => console.log(respostas))
.catch(erro => console.error(`Falha ao buscar: ${erro}`))
```

Se alguma das promessas estiver com erro, o retorno será o catch com o erro da primeira promessa com erro.

Métodos de Promises

Promise.allSettled(): Semelhante ao Promise.all, porém ele nunca será rejeitado (catch).
No Promise.all, quando uma das promessas é rejeitada, a função catch é acionada, já no Promise.allSettled, é armazenado que o status da promessa foi rejeitada, mas o retorno continua sendo pela função then, no formato de array da seguinte forma:

```
[
        { status: "fulfilled", value: "valor da promessa 1"},
        { status: "fulfilled", value: "valor da promessa 2"},
        { status: "rejected", reason: "erro da promessa 3"}
]
```

Métodos de Promises

Promise.allSettled(): A sintaxe é semelhante a do Promise.all:

```
const fetchPromise1 = fetch("API 1")
const fetchPromise2 = fetch("API 2")
const fetchPromise3 = fetch("API 3")
Promise.allSettled([fetchPromise1, fetchPromise2, fetchPromise3])
.then(respostas => console.log(respostas))
.catch(erro => console.error(`Falha ao buscar: ${erro}`))
```

Métodos de Promises

Promise.any(): Quando é necessário captar o resultado de uma promessa qualquer de um conjunto, assim que houver a conclusão de uma promessa, irá retornar essa única conclusão como then ou catch

```
const fetchPromise1 = fetch("API 1")
const fetchPromise2 = fetch("API 2")
const fetchPromise3 = fetch("API 3")
Promise.any([fetchPromise1, fetchPromise2, fetchPromise3])
.then(respostas => console.log(respostas))
.catch(erro => console.error(`Falha ao buscar: ${erro}`))
```