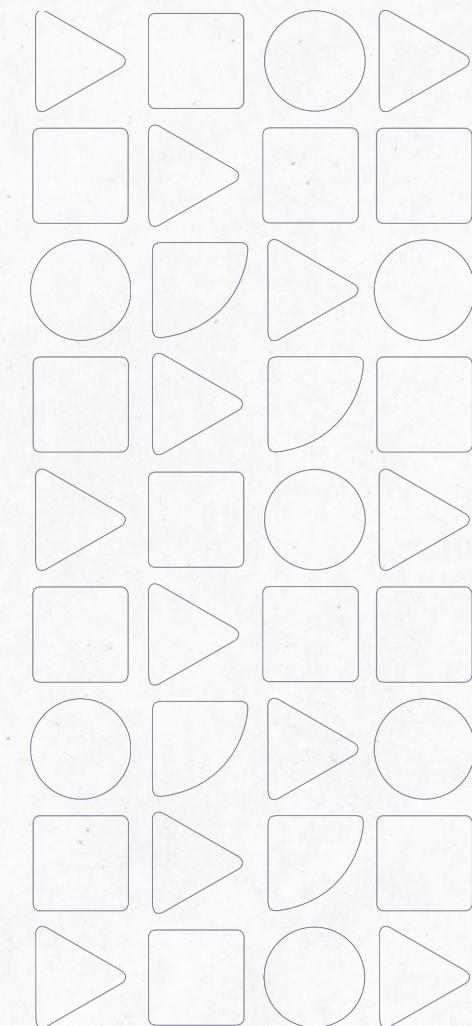


Funções Assíncronas

Disciplina: Linguagem de Programação



Conteúdos:

Funções assíncronas.

Habilidade(s):

- Utilizar ambientes de desenvolvimento de *software* para aplicativos móveis;
- Aplicar técnicas corretivas em aplicativos e sistemas;
- Conhecer a estrutura das linguagens de programação de alto nível.

Bloco 1

Explorando as funções assíncronas.

Quem é você?

De acordo com as fotos abaixo, quem é você quando o assunto é uma **reunião assíncrona**?



1



2



3



4



O que define uma reunião assíncrona é o fato dos participantes não estarem presentes nela ao mesmo tempo.

Mas... qual é a sua semelhança com as funções assíncronas?

Funções assíncronas

São funções que permitem a execução de tarefas em segundo plano, sem bloquear a execução do programa principal.

São baseadas em ***promises***, recebendo o retorno tanto se forem executadas ou não.

Essa abordagem é usada para lidar com operações que podem levar algum tempo para serem realizadas, como fazer solicitações de rede, ler arquivos grandes ou executar cálculos demorados.



Sintaxe das funções assíncronas

```
async function minhaFuncaoAssincrona() {  
    // Código síncrono aqui  
  
    // Usando await para esperar uma promise  
    const resultado = await algumaOperacaoAssincrona();  
  
    // Mais código síncrono  
}
```

async: é usada antes da palavra-chave “*function*” para indicar que a função é assíncrona.

nomeDaFuncao: deve ser um identificador válido e seguir as regras de nomenclatura de variáveis em JavaScript.

await: pausa a execução da função até que a *promise* seja resolvida ou rejeitada.

O que são promises?

São uma maneira de lidar com operações assíncronas de forma mais organizada e legível. Elas representam um valor que pode estar disponível **agora, no futuro ou nunca**. São representadas pelos três estados abaixo:

- | | |
|---------------------------------------------------------------------------------------------|-------------------------------------------------------------|
|  Pendente | ▶ Estado inicial, ou seja, não foi realizada nem rejeitada. |
|  Realizada | ▶ Sucesso na operação. |
|  Rejeitada | ▶ Falha na operação. |

Vamos analisar um exemplo?

Ao executar a **segundaFuncao()**, a última linha só irá rodar quando a resposta da **primeiraFunção()** for exibida.

```
1 function primeiraFuncao() {  
2   console.log("Esperou isso")  
3 }async function segundaFuncao() {  
4   console.log("Iniciou")  
5   await primeiraFuncao()  
6   ...  
7   console.log("Agora executou isso!")  
8 }  
segundaFuncao()
```

CONSOLE X

```
Iniciou  
Esperou isso  
Agora executou isso!
```

Funcionamento da *promise*

- Quando uma *promise* pendente se torna realizada com sucesso, ou seja, quando a operação assíncrona é bem-sucedida, o método **then()** é acionado;
- Se a *promise* for rejeitada por algum motivo, o método **catch()** ou o segundo argumento do **then()**, que é usado para manipular erros, será acionado.



Sintaxe da *promise*

```
new Promise ((resolve: Function, reject: Function) => void)
```

resolve: função para retornar o resultado da *promise*.

reject: função para retornar o erro da *promise*.

Elementos da *promise*

Propriedade

Constructor: função construtora que recebe um *callback*, criando uma função assíncrona.

Métodos

Then: permite definir o bloco executado mediante o cumprimento de uma *promise*.

Catch: permite definir o bloco executado mediante a rejeição de uma *promise*.

Funções

Resolve: cria uma *promise* resolvida.

Reject: cria uma *promise* rejeitada com o resultado igual ao argumento recebido.

All: faz a união de várias *promises* em um *array* e retorna o valor quando todas tiverem sido resolvidas.

Observando na prática

Aqui, a mensagem é mostrada após cinco segundos da execução da função, com o disparo da função '`timeout`'.

```
1 const timeout = (duration) => {
2   return new Promise((resolve, reject) => { setTimeout(resolve, duration)
3   })
4 }
5 timeout(5000)
6 .then(function() { // executa o bloco após 5 segundos
7   console.log('Passou 5 segundos')
8 })
```

CONSOLE 

```
Passou 5 segundos
```

Bloco 2

Conhecendo os temporizadores.

Vamos recapitular?

Explicando para uma criança

Imagine que você está tentando ensinar **funções assíncronas** para o seu primo de oito anos. Aqui, você deve utilizar as melhores palavras e analogias para que ele possa compreender o que é dito corretamente.

Desafio

Tente fazer isso em um minuto!



Temporizadores

Você já usou um **cronômetro**?

O temporizador funciona de maneira semelhante: são funções usadas para o retorno após um **determinado tempo**.

Aqui, temos duas funções principais:

setTimeout()

setInterval()

setTimeout()

Executa um bloco específico uma vez depois de determinado tempo.

Possui os seguintes parâmetros:

- zero ou mais valores que representam quaisquer parâmetros que você quiser passar para a função assim que ela for executada;
- uma função a ser executada ou uma referência de uma função definida em outro lugar;
- um número representando o intervalo de tempo em milissegundos. 1000 milissegundos equivalem a um segundo para esperar antes de executar o código.

Exemplo de um *setTimeout()*

Com esse código, espera-se dois segundos para aparecer uma janela de alerta com a mensagem “Seja bem-vindo”.

```
1 let saudacao = setTimeout(function() {  
2   console.log("Seja bem-vindo!");  
3 }, 2000)
```

CONSOLE ×

Seja bem-vindo!

setInterval()

Executa um bloco específico e com um intervalo fixo entre cada chamada. Os parâmetros podem ser encontrados no seguinte código:

```
1 // Função para usar o timeout
2 function saudacao(nome){
3   console.log("Bem-vindo" + nome);
4 }
5 // Criando uma variável e passando os parâmetros(função, tempo, nome)
6 let msg = setTimeout(saudacao, 2000, " José de Assis");
```

CONSOLE ×

Bem-vindo José de Assis

Parando o tempo

Aqui, foi possível adicionar um botão para dar o *stop* no relógio.

```
1 <html>
2 <body>
3 <h2>Relógio com botão de parada</h2>
4 <p id="relogio"></p>
5 <button onclick="clearInterval(intervalo)">Parar relógio</button>
6 <script>
7 let intervalo = setInterval(meutimer ,1000); function meuTimer() {
8 const d = new Date();
9 document.getElementById("relogio").innerHTML = d.toLocaleTimeString();
10 }
11 </script>
12 </body>
13 </html>
```

WEBSITE VIEW 

Relógio com botão de parada

22:38:00

Parar relógio

Bloco 3

Explorando o *WebStorage*.

Se você pudesse escolher...

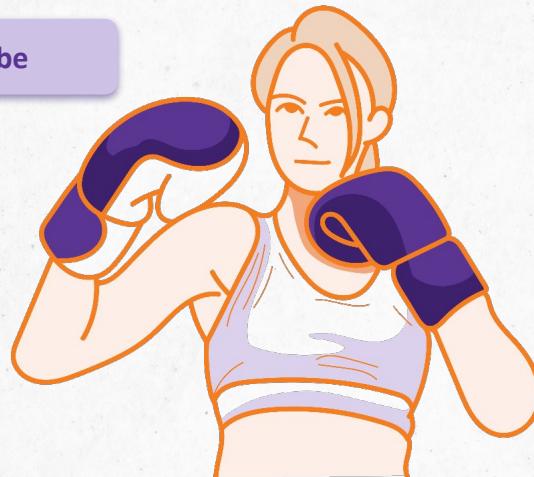
Entre mostrar o seu histórico do navegador e o seu histórico do YouTube, o que você preferiria?



Navegador

VS

YouTube



WebStorage

O histórico dos navegadores está envolvido com o **armazenamento** temporário de dados.

Com o lançamento do HTML5, o armazenamento de dados tornou-se bem mais prático e eficiente. Foram adicionados dois objetos para o controle dessa função:

localStorage

Armazena dados sem data de expiração.

sessionStorage

Armazena apenas para a sessão atual.

Características do *WebStorage*

Propriedade

Storage.length: retorna um número inteiro e que é representado pelo número de itens de dados armazenados no objeto *storage*.

Métodos

Storage.key(): é usado quando passado um número “n”. Ele retorna o nome da enésima chave no dado objeto;

Storage.getItem(): é usado para adicionar a chave passada, caso ela já exista, alterar essa chave;

Storage.RemoveItem(): remove a chave do armazenamento;

Storage.clear(): limpa todos os itens do armazenamento.

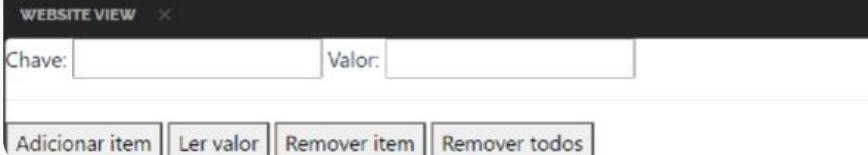
Observando de forma prática

1

O primeiro passo é inserir uma chave e um valor nos respectivos campos do documento.

O usuário deve inserir a chave que está buscando no primeiro campo e clicar no botão **Ler valor**.

```
1 <html>
2 <head>
3 <script type="text/javascript">
4 //inserir as funções aqui, uma abaixo da outra
5 </script>
6 </head>
7 <body>
8 Chave: <input type="text" id="txtChave"/>
9 Valor: <input type="text" id="txtValor"/>
10 <hr/>
11 <button id="btnAdicionar" onclick="adicionar()">Adicionar item</button>
12 <button id="btnLer" onclick="ler()">Ler valor</button>
13 <button id="btnRemover" onclick="remover()">Remover item</button>
14 <button id="btnLimpar" onclick="limpar()">Remover todos</button>
15 </body>
16 </html>
```



Observando de forma prática

2

Se o item existir, o seu valor será exibido. Caso não exista uma mensagem, será mostrada pelo '*alert*'.

```
function ler(){
var obj = localStorage.getItem(txtChave.value);
if(obj != null){
alert(obj)
} else{
alert("O item procurado não existe.")
}}
```

3

Depois, a partir da chave digitada no campo, há a remoção de um item específico.

```
function remover()
{
localStorage.removeItem(txtChave.value);
alert("Item removido.");
}
```

Observando de forma prática

4

Há a função `clear()`, que serve para excluir todos os itens do *storage*.

```
function limpar(){
localStorage.clear();
alert("Itens Removidos Com Sucesso!!");
}
```

5

Com um *script*, podemos verificar se o navegador suporta a funcionalidade *WebStorage*. Em caso negativo, as operações que fariam uso dela são abortadas. Observe ao lado:

```
1 if(typeof(Storage)!=="undefined"){
2 //o browser suporta HTML 5 Web Storage
3 }else{
4 //o browser não suporta HTML 5 Web Storage
5 }
```

Cuidados e limitações ao usar o *localStorage*

- 1 Você só pode usar *strings* no *localStorage*.
- 2 O uso do *localStorage* é síncrono.
- 3 Em qualquer navegador, o *localStorage* é limitado a guardar somente 5Mb de dados.
- 4 Não use o *localStorage* para guardar dados sensíveis e confidenciais.
- 5 Não pode ser usado por *web workers*.
- 6 Os dados que estão gravados não têm camada nenhuma de proteção de acesso.

Bloco 4

Conhecendo algumas funcionalidades do ES6.



Vamos realizar um desafio?

Separem-se em grupos e distribuem-se em *notebooks* e computadores. Abram o navegador do Google Chrome e, em seguida, apertem o atalho F12. Discutam o que conseguem observar e qual é a relação do que é visto com o *WebStorage*.

ECMAScript (ES)

??

Você sabe o que é ECSMAScript?

É o padrão que define a linguagem JavaScript, indicando como a linguagem deve funcionar e quais recursos devem ser suportados.

A seguir, iremos conhecer alguns.



Palavras-chave

'*let*' e '*const*' são usadas para declarar variáveis. Desse modo, *let* serve para variáveis **mutáveis** e *const* para variáveis com valores **constantes**.

Diferentemente de *var*, *let* e *const* possuem **escopo de bloco**.

let

```
let contador = 0;  
contador = contador + 1;  
console.log(contador);
```

const

```
const pi = 3.14159;  
console.log(pi)
```

VS

Funções helper para arrays

Funções como **'forEach'**, **'map'**, **'filter'**, **'find'**, **'every'** e **'reduce'** facilitam a manipulação de *arrays*, permitindo realizar operações em cada elemento ou criar novos *arrays* com base em critérios específicos.



Map

Cria um *array* contendo o mesmo número de elementos de um determinado *array*.

```
1 var cores = ['vermelho', 'verde', 'violeta']
2 function maiusculas(v){
3   return v.toUpperCase();
4 }var coresMaiusculas = cores.map(maiusculas)
5 console.log(coresMaiusculas);
```

CONSOLE 

```
(3) [ "VERMELHO", "VERDE", "VIOLETA" ]
```

Filter

Cria um *array* contendo subconjunto dos elementos do *array* original.

```
1 var valores = [2, 45, 67, 54, 12, 5]
2 // função que retorna valores menores que 50
3 function novaFuncao(v) {
4   return v < 50
5 }
6 var valorMenor50 = valores.filter(novaFuncao)
7 console.log(valorMenor50);
```

CONSOLE ×

```
(4) [2, 45, 12, 5]
```

Find

Encontra o primeiro elemento de um *array* que satisfaz determinada condição imposta por uma função que retorna verdadeiro (*true*) ou falso (*false*).

```
1  var galera = [
2    {nome: 'Afonso', idade: 8},
3    {nome: 'Thiago', idade: 31},
4    {nome: 'Diego', idade: 29},
5    {nome: 'Lucia', idade: 15},
6    {nome: 'Monique', idade: 24}
7  ]
8  function jovem(pessoa) {
9    return pessoa.idade > 10 && pessoa.idade < 18
10 }
11 var primeiraPessoa = galera.find(jovem)
12 console.log('Primeiro adolescente encontrado:',primeiraPessoa.nome, 'com', primeiraPessoa.idade, 'anos.')
```

CONSOLE ×

Primeiro adolescente encontrado: Lucia com 15 anos.

Every

Faz a varredura em cada elemento de um *array*.

```
1  var galera = [
2    {nome: 'Afonso', idade: 8},
3    {nome: 'Thiago', idade: 31},
4    {nome: 'Diego', idade: 29},
5    {nome: 'Lucia', idade: 15},
6    {nome: 'Monique', idade: 24}
7  ]
8  function jovem(pessoa) {
9    return pessoa.idade > 10 && pessoa.idade < 18
10 }
11 var todosSaoAdolescentes = galera.every(jovem)
12 console.log('Todo mundo é adolescente.', todosSaoAdolescentes)
```

CONSOLE ×

Todo mundo é adolescente. **false**

Reduce

Permite o uso de dois parâmetros e se destina a interação dos elementos de um *array*.

```
1 var array = [1, 2, 3, 4]
2 function soma(somar, valor) {
3   return somar + valor
4 }
5 function produto(mult, valor) {
6   return mult * valor
7 }
8 var somaDosArrays = array.reduce(soma, 0)
9 var produtoDosArrays = array.reduce(produto, 1)
10 console.log('A soma dos elementos do array', array, 'é ', somaDosArrays)
11 console.log('O produto dos elementos do array', array, 'é', produtoDosArrays)
```

CONSOLE

A soma dos elementos do array

(4) [1, 2, 3, 4]
é 10

O produto dos elementos do array

(4) [1, 2, 3, 4]
é 24

Arrows functions

São uma forma mais concisa de criar funções em ES6, úteis para simplificar a sintaxe e melhorar a legibilidade do código.

```
1 var array = [1, 2, 3, 4]
2 function soma(somar, valor) {
3 | return somar + valor
4 }
5 function produto(mult, valor) {
6 | return mult * valor
7 }
8 var somaDosArrays = array.reduce(soma, 0)
9 var produtoDosArrays = array.reduce(produto, 1)
10 console.log('A soma dos elementos do array', array, 'é ', somaDosArrays)
11 console.log('O produto dos elementos do array', array, 'é', produtoDosArrays)
```

CONSOLE ×

A soma dos elementos do array

(4) [1, 2, 3, 4]

é 10

O produto dos elementos do array

(4) [1, 2, 3, 4]

é 24

Objetos literais incrementados

ES6 tornou mais fácil criar objetos literais com campos de variáveis de mesmo nome.

```
1 var array = [1, 2, 3, 4]
2 var somaDosArrays = array.reduce((somar, valor) => somar + valor, 0)
3 var produtoDosArrays = array.reduce((produto, valor) => produto * valor, 1)
4 console.log('A soma dos elementos do array', array, 'é', somaDosArrays)
5 console.log('O produto dos elementos do array', array, 'é', produtoDosArrays)
```

CONSOLE ×

```
A soma dos elementos do array
(4) [1, 2, 3, 4]
é 10
O produto dos elementos do array
(4) [1, 2, 3, 4]
é 24
```

Bloco 5

Vamos explorar o VS Code?

O que é o VS Code?

Desenvolvido pela Microsoft, o Visual Studio Code é um ambiente de desenvolvimento integrado (IDE) leve e altamente configurável.

É uma ferramenta amplamente usada por programadores.





Navegue pelo conhecimento!



Vamos explorar o VSCode?

A rectangular window with rounded corners and a thin orange border. The title bar is orange and contains the text "Navegue pelo conhecimento!". The main area is white. In the center is a large orange globe icon with a white cursor arrow pointing towards it. Below the globe is a purple button with the text "Vamos explorar o VSCode?". The window has standard OS X-style controls (minimize, maximize, close) in the top right corner.

Bloco 6

Hora de praticar!

Quem quer ser um programador?

Separem-se em cinco grupos.

Nesta atividade, que se assemelha ao programa “Quem quer ser um milionário?”, o(a) professor(a) fará algumas perguntas para os grupos. Caso saibam as respostas, os membros devem levantar as mãos rapidamente.

Cada resposta correta fornecerá um ponto. A equipe que acumular o maior número de pontos será declarada a vencedora.



Fechamento

O que eu já sabia?

O que eu não sabia?

O que quero aprender?

Referências Bibliográficas

PROZ EDUCAÇÃO. *Apostila de Desenvolvimento para Dispositivos Móveis I.* 2023.