Valderi Leithardt, Dr.

IPW (Class notes for the fourth week)

Summary

- Review of previous class *
- Express.JS
- Asynchronous Programming
- Exercises

The Secret Life of Objects (*Review previous class)

A maioria das histórias de programação, começa com um problema de complexidade. A teoria é de que a complexidade pode ser administrada separando-a em pequenos compartimentos isolados um do outro. Esses compartimentos acabaram ganhando o nome de objetos. (sistemas modulares…)

De acordo com https://eloquentjavascript.net/06_object.html

Um objeto é um “escudo” que esconde a complexidade dentro dele, e nos apresenta pequenos conectores (como métodos) que apresentam uma interface para utilizarmos o objeto. A ideia é que a interface seja relativamente simples e toda as coisas complexas que vão dentro do objeto possam ser ignoradas enquanto se trabalha com ele.

No ECMAScript 2015 foi introduzida uma sintaxe reduzida para definição de métodos em inicializadores de objetos. É uma abreviação para uma função atribuída ao nome do método. Material adaptado com base em MDN

Métodos são propriedades simples que comportam valores de funções. Isso é um método simples:

```
var coelho = {};
coelho.diz = function(linha) {
  console.log("O coelho diz '" + linha + "'");
};

coelho.diz("Estou vivo.");
// → O coelho diz 'Estou vivo.'
```

Definição de Método (*Review previous class)

7. filter()
O método filter gera um novo array como os elementos que estão na condição da função que foram inseridos como parâmetro para o método.

```
const words = ['spray', 'limit', 'elite', 'exuberant', 'destruction', 'present'];

const result = words.filter((word) => word.length > 6);

console.log(result);
// Expected output: Array ["exuberant", "destruction", "present"]
```

8. sort()
Este método é utilizado para arranjar ou ordenar os itens de um array de maneira ascendente ou descendente.

```
const arr = [2,4,6,8];
const alpha = ['d', 'c', 'b', 'a'];
const descendente = arr.sort((a, b) => a > b ? -1 : 1);
console.log(descendente);
const ascendente = alpha.sort((a, b) => a > b ? 1 : -1);
console.log(ascendente);
```

Definição de Protótipo (*Review previous class)

- Em JavaScript um objeto é uma coleção de propriedades, sendo cada propriedade definida como uma sintaxe de par chave : valor. A chave pode ser uma string e o valor pode ser qualquer informação.

- Cada objeto em **JavaScript** tem uma referência interna a outro objeto, chamado de "**protótipo**". Rever conceitos sobre objetos, tutorial JS*

- Quando uma propriedade ou método é acessado em um objeto, o **JavaScript** primeiro verifica se essa propriedade ou método existe no próprio objeto.

- Se não for encontrado, o **JavaScript** procurará no **protótipo** do objeto.

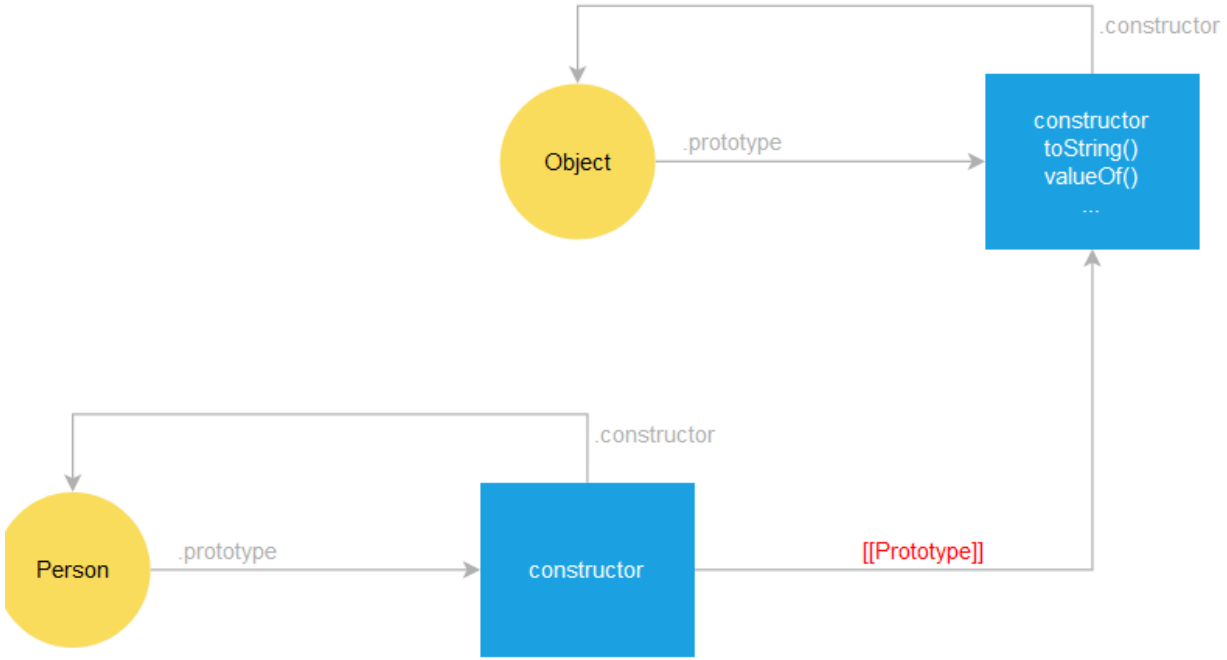
- Protótipos são os mecanismos pelo qual objetos JavaScript herdam recursos uns dos outros.

Material adaptado conforme descrito em MDN

Construtores (*Review previous class)

- Em JavaScript, chamar uma função precedida pela palavra-chave **new** faz com que ela seja tratada como um construtor.
- O construtor terá sua variável **this** relacionada a um novo objeto, e a menos que explicite o retorno do valor de outro objeto, esse novo objeto será retornado a partir da chamada.
- Um objeto criado com **new** é chamado de instância do construtor.

*Material adaptado conforme descrito em livro PT




```
function Coelho(tipo) {
  this.tipo = tipo;
}

var coelhoAssassino = new Coelho("assassino");
var coelhoPreto = new Coelho("preto");
console.log(coelhoPreto.tipo);
// → preto
```

**Outros exemplos em tutoriais JS

JSON (*Review previous class)

JavaScript Object Notation

Diferenças:

- JSON não é uma linguagem de marcação. Não possui tag de abertura e muito menos de fechamento!
- JSON representa as informações de forma mais compacta.
- JSON não permite a execução de instruções de processamento, algo possível em XML.
- JSON é tipicamente destinado para a troca de informações, enquanto XML possui mais aplicações.
- Por exemplo: nos dias atuais existem bancos de dados inteiros armazenados em XML e estruturados em SGBD's XML nativo.

Material Adaptado conforme reference w3s

Node.JS (*Review previous class)

Tutorial NODE.JS para iniciantes

https://learn.microsoft.com/pt-br/windows/dev-environment/javascript/nodejs-beginners-tutorial

Tutorial e documentação

https://www.tutorialspoint.com/nodejs/index.htm

https://www.tutorialspoint.com/nodejs/nodejs_tutorial.pdf

Tutorial W3S

https://www.w3schools.com/nodejs/

Express.js

Express é o framework Node mais popular, é a biblioteca subjacente para uma série de outros frameworks do Node. O Express oferece soluções para:

✓ Gerir requisições de diferentes verbos HTTP em diferentes URLs;

--> Verbos HTTP: GET, POST, DELETE, PUT, PATCH

Material adaptado conforme descrito em MDN, 2023

✓ Integrar "view engines" para inserir dados nos templates;
✓ Definir as configurações comuns da aplicação web, como a porta a ser usada para conexão e a localização dos modelos que são usados para renderizar a resposta;
✓ Adicionar novos processos de requisição por meio de "middleware" em qualquer ponto da "fila" de requisições.

Tutoriais de instalação

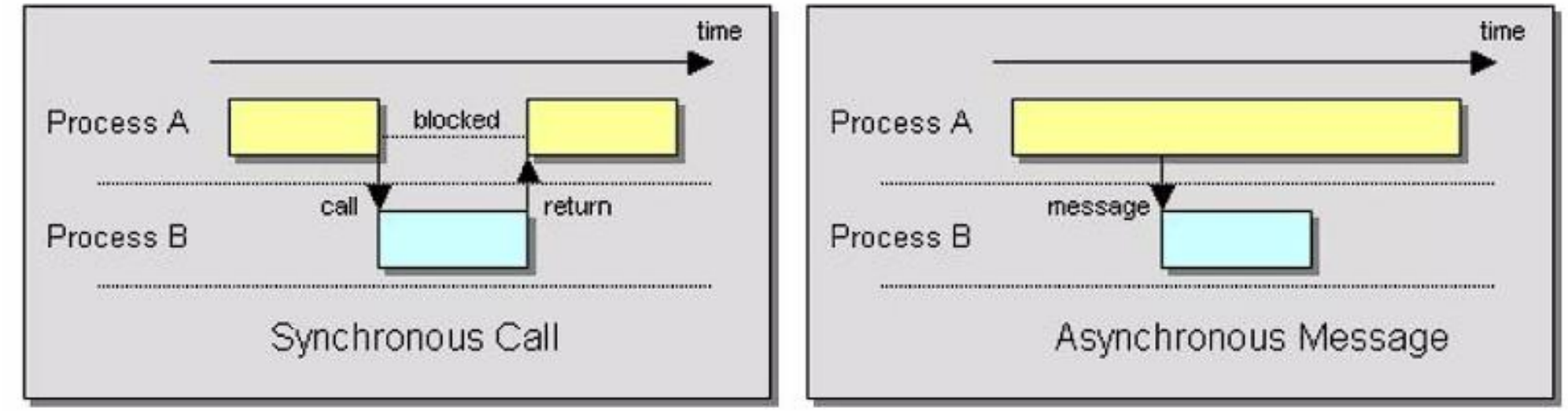
https://github.com/programadriano/node-express

https://github.com/expressjs/express

Programação assíncrona (cont)…

É um conceito essencial em JavaScript que permite que seu código seja executado em segundo plano sem bloquear a execução de outro código. Os desenvolvedores podem criar aplicativos mais eficientes e responsivos usando recursos como retornos de chamada, async/await e promises.

Material adaptado disponível em: * google 2023



Em códigos síncronos, todas as funções e requisições trabalham em sincronia, em um contato direto, do início ao fim da comunicação. Dessa maneira, esse código só permite uma requisição por vez.

Programação assíncrona vs. sincrona

```
}

function funcao1(){
    console.log("Executa 1...")
}

function funcao2(){
    console.log("... Executa 2...")
}

function funcao3(){
    console.log("...Execução 3!")
}

funcao1()
funcao2()
funcao3()
```

A **programação síncrona** é aquela que ocorre simultaneamente, ao mesmo tempo.

Isso significa que o código precisa estar em "sincronia", de modo a executar funções e receber respostas a essas funções logo em seguida. Então, podemos dizer que o código é executado "na ordem", Por exemplo:

```
function executaDe1a4(funcao) {
    funcao(1);
    funcao(2);
    funcao(3);
    funcao(4);
  }

  executaDe1a4(function (x) {
console.log('IPW Turma ' + x); });
```

Programação assíncrona

- Na programação assíncrona as funções não são executadas em ordem. Com o assincronismo podemos interromper do código para conseguirmos alguma outra informação necessária para continuar a execução.
- Isso significa que o código espera por um outro trecho de código e, enquanto espera, executa as demais partes.
- Para exemplificar esse efeito, sem usar Promises, podemos usar o método setTimeout() do JavaScript, dessa forma:

```
function funcao1(){
    console.log("Execução  assincrona!")
}

function funcao2(){
    setTimeout((funcao1) => {
        console.log("O que vai aparecer
primeiro?")
    }, 1200);
}
funcao2()
```

Programação assíncrona

Dessa forma a String “assincrona - Funcionou?” é impressa depois dos segundos que especificamos pelo timeOut(). Executar uma terceira função que não seja marcada pelo assincronismo, veremos o que será impressa primeiro.

```
function funcao1(){
    console.log("Teste assincrono!")
}

function funcao2(){
    setTimeout((funcao1) => {
        console.log("assincrona - Funcionou? ")
    }, 1200);
}

function funcao3(){
    console.log("Vai ser executada primeiro, pois não esta marcada por
assincrona!")
}
//funcao1()
funcao2()
funcao3()
```

Programação assíncrona

Foi utilizado o setTimeout junto de uma Arrow Function, a String que especificamos em funcao1() foi “esquecida”. Podemos modificar isso para exibir as três Strings dessa forma:

```
function funcao01(){
    console.log("Teste assincrono!") //e por último essa daqui!
}
function funcao02(){
    setTimeout(funcao01, 1200);
    console.log("assincrona - Funciona? ") //não é uma execução
assíncrona, então imprime primeiro!
    console.log("...")// depois essa...
}
funcao02()
```

Programação assíncrona

Um método de callback é uma rotina que é passada como parâmetro para outro método. É esperado então que o método execute o código do argumento em algum momento. A invocação do trecho pode ser imediata, como em um (callback síncrono), ou em outro momento (callback assíncrono). Conforme descrito em: [1]

Os meios em que os callbacks são suportados em diferentes linguagens de programação diferem, porém eles são normalmente implementados com sub-rotinas, expressões lambda, blocos de código ou ponteiros de funções.

Uma função que recebe outra função como argumento é considerada uma função de ordem maior, em contraste à uma função de primeira ordem. Há de se notar no entanto que existem outros tipos de funções de ordem maior, como as que retornam uma nova função sem receber uma como argumento.

Dois exemplos clássicos de funções de callback são as **funções setTimeout()** e **setInterval()**, que são nativas do JavaScript. Ambas esperam uma função de retorno.

Programação assíncrona

O efeito “Callback Hell”

O “callback hell” ocorre quando você tem múltiplos níveis de aninhamento de callbacks, o que torna o código difícil de ler e entender. Para evitar isso, utilize técnicas como promisses, async/await ou bibliotecas que facilitam o tratamento de operações assíncronas de maneira mais clara. (*Conforme descrito em google 2023).

As Promises também são úteis para resolver o “callback hell”, que acontece quando usamos várias callbacks dentro de outras callbacks para termos retornos assíncronos de várias funções, deixando o código poluído e confuso.

```
setTimeout(function(){
    console.log("Executando Callback...")

    setTimeout(function (){
        console.log("Executando
Callback...")

        setTimeout(function(){
            console.log("Executando
Callback...")
        }, 2000)
    }, 2000)
}, 2000)
```

(*Definições e conceitos callbackhell, 2023).

(*Figura extraída do google (2023)).

Programação assíncrona (cont...)

O código anterior fica um pouco excessivo e complicado. Por outro lado, quando usamos Promises o resultado parece ser mais simples:

```
function esperarPor(tempo = 2000){
    return new Promise(function(resolve){
        setTimeout(function(){
            console.log("Executando Promise...")
            resolve() //ao chamar o resolve, o then vai ser chamado
        }, tempo)
    })
}

esperarPor() //se não passar nenhum valor, ele espera o valor padrão de 2s
    .then(() => esperarPor())
    .then(esperarPor)
```

Programação assíncrona

Conceitos Promises

- É importante: Promises são objetos. Consequentemente, sempre que instanciarmos uma nova Promise, ela deverá estar acompanhada da palavra new.

- Utilizamos Promises para processamento assíncrono.
- Promisses recebem duas funções callback, que são funções que chamam outras funções: a resolve e a reject.
- Usamos essas funções, respetivamente, quando a Promises é resolvida e quando ela é rejeitada, ou ocorre erro. Qualquer uma das duas situações chama uma função correspondente.

- Promises contam também com diversos métodos, sendo o foco aqui o método then() e o método catch().
- **O método then()** possui dois argumentos, que também são funções callback resolve e reject.
- Enquanto o then() lida com ambos casos (tanto se a Promise for resolvida quanto rejeitada), o catch() lida apenas com casos de rejeição.

- Por sempre retornarem Promises, podemos encadear tanto o then() quanto o catch(), embora geralmente utilizamos o catch() por último na linha de código, pois ele trata dos erros → depois de vários then's, o catch é utilizado para "coletar" todos os erros.

Programação assíncrona

Conceitos async/await

- A declaração async function define uma função assíncrona, que retorna um objeto AsyncFunction. (Conforme descrito em MDN 2023).

- Quando uma função assíncrona é chamada, ela retorna uma Promise. Quando a função assíncrona retorna um valor, a Promise será resolvida com o valor retornado. Quando a função assíncrona lança uma exceção ou algum valor, a Promise será rejeitada com o valor lançado.

- Uma função assíncrona pode conter uma expressão await, que pausa a execução da função assíncrona e espera pela resolução da Promise passada, e depois retoma a execução da função assíncrona e retorna o valor resolvido.

Example

```
async function myFunction() {
  return "Hello";
}
```

Is the same as:

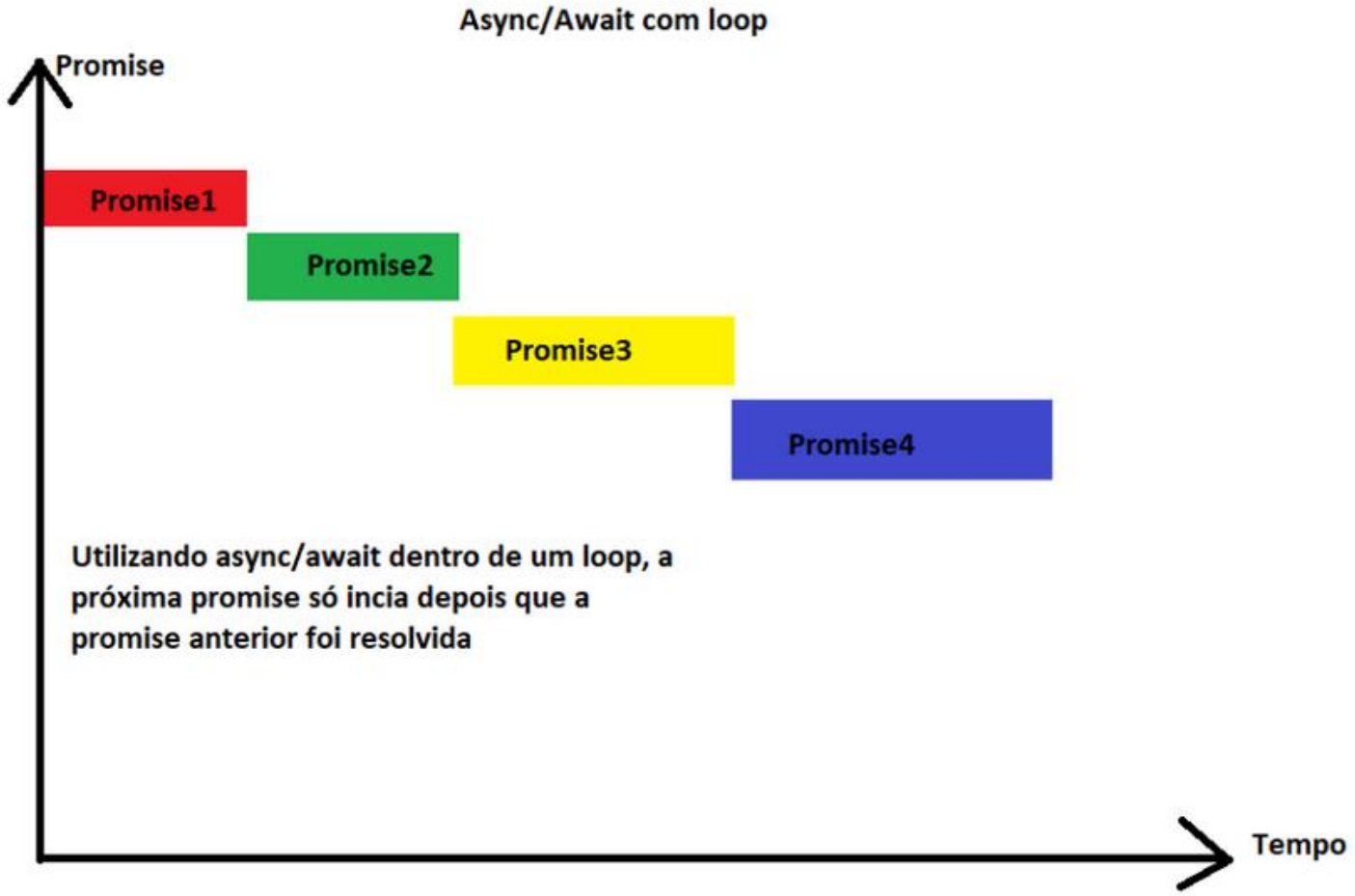
```
function myFunction() {
  return Promise.resolve("Hello");
}
```

(Figura extraída de W3S 2023)

Programação assíncrona

Conceitos async/await

- Uma função assíncrona pode conter uma expressão await, que pausa a execução da função assíncrona e espera pela resolução da Promise passada, e depois retoma a execução da função assíncrona e retorna o valor resolvido.



Programação assíncrona

Conceitos async/await

- O **método Promise.all()** recebe um iterável de promesses como entrada e retorna uma única Promesse que resolve para uma matriz dos resultados das promesses de entrada.

- Essa promise retornada será cumprida quando todas as promises de entrada forem cumpridas ou se o iterável de entrada não contiver promises.

- Então rejeita imediatamente após qualquer uma das promises de entrada rejeitando ou não promises lançando um erro, e irá rejeitar com esta primeira mensagem/erro de rejeição.

- Em resumo: **o método Promise.all** recebe um array de promises e retorna uma única promise que só irá ser resolvida quando todas as promises do array que foi passado forem resolvidas.

* Outros exemplos e definições (2023).

Programação assíncrona

Principais estados de uma Promise

- **Pending**

Como a própria tradução da palavra, o estado da Promise ainda é pendente. Ou seja, é quando a sua Promise não passou pelo processo de ser fulfilled (sucesso) ou rejected (rejeitada).

- **fulfilled**

No inglês, também é conhecida como “resolved”, ou seja, é quando nossa Promise foi resolvida com sucesso.

- **rejected**

Nesse estado, a Promise é rejeitada, ou seja, a operação falha.

- **Settled**

Essa é a etapa final da Promise, uma vez que se percebe a conclusão e saberemos se ela foi resolved (realizada) ou rejected (rejeitada).

Programação assíncrona

Quando utilizar uma promise?

- ✓ Na estruturação de um código, a criação de funções assíncronas auxiliam no fluxo do código. A exemplo disso, ela pode ser utilizada no momento de processamento de imagens no programa.

- ✓ Nesse sentido, manter uma padronização no código também é fundamental para manter a organização, uma vez que em uma implementação que demanda o auxílio de diferentes pessoas a legibilidade torna-se essencial.

Propriedades e Métodos das promises

1) Propriedades

- ✓ Promise.length

A propriedade length é responsável pelo número de argumentos do construtor, o valor sempre será 1.

- ✓ Promise.prototype

Propriedade protótipo responsável pelo método construtor da Promise.

Programação assíncrona

2) Métodos

* No livro há outros exemplos disponíveis

✓ **Promise.all**

Esse método retornará todas as promisses que estiverem presentes no argumento. Por exemplo: uma promisse.all(lista), o argumento será responsável por retornar sempre que a Promise for resolvida ou rejeitada. Nesse sentido, se Promise for resolvida, será aplicado um array para mostrar as promises dessa lista.

✓ **Promise.race**

Esse método retornará o valor de uma Promise ou motivo pelo qual ela está sendo resolvida ou rejeitada.

✓ **Promise.reject**

Esse método é responsável por retornar os objetos promises que foram rejeitados e o seu motivo.

✓ **Promise.resolve**

Esse método é responsável por retornar os objetos do promises que foram resolvidos e o seu valor. Uma curiosidade legal na construção do seu código é que você pode consultar se o valor é uma promise.

Exercises

- 1º Trabalho disponibilizado em:

https://github.com/isel-leic-ipw/2324i-IPW-LEIC31D/wiki/IPW_IP-2324-1-A1

Cap 11 – livro ENG: https://eloquentjavascript.net/11_async.html

* Concluir exercícios da aulas anteriores.

References

- https://eloquentjavascript.net/11_async.html
- https://www.javascripttutorial.net/
- https://developer.mozilla.org/pt-BR/docs/Learn/JavaScript/
- https://www.w3schools.com/js/default.asp
- Material adaptado de Raquel Fontenelle 2023.
- Também foram realizadas adaptações e modificações com base no material disponibilizado por Professor Luís Falcão, acesso online em: https://github.com/isel-leic-ipw/
- Aulas gravadas Professor Falcão:
- https://videoconf-colibri.zoom.us/rec/share/A9TGbxJpYSBUzl7YwZeeMTT3NdyEEhRkjeY73FgD95g7ayv-2gJp3ftjvzTC3Qij.zQh622OYbtvBSpaz
- https://videoconf-colibri.zoom.us/rec/share/PFbUkFuNhvNEjF3RCbOVR1MPRwPd4ZjwtetQuhe6BGVe0SLpfgF__61tXOJ2MGte.iOrjlJ-i3LYMfdrG
- https://videoconf-colibri.zoom.us/rec/share/13QF--PBrOpXWJ_bPBkdPDDzN2o_J15sIV2N3W6rWcbkmILeML5D9D8DHXM1psVk.rPO6wQysUXy1pHkY
- Part 1: https://videoconf-colibri.zoom.us/rec/share/Eq1oe-3PrY0AlgZYr73zFklnQ1JKYjYUW6yEPMmMKTaR2XUjMJFN8EC1Wecz6MJE.a2RysTo8b7MGtHqP
- Part 2: https://videoconf-colibri.zoom.us/rec/share/dvJUsndNkc1krQS7d-mDPwBwEkrKJvsJUBVRhOpnvjeyFLLf1cX8iK0rHIJWk-HN.V6vNOr6wxMdvbGFz
- Part 3: https://videoconf-colibri.zoom.us/rec/share/UcN_r6j6VWkC6MzCfXtT1jJgIkbBK0lAZUO51I0PL11dSKahV7J7YnjInGlJ6MLt._3RozFUJ_JwsdmrX
- https://videoconf-colibri.zoom.us/rec/share/XUBZUISQJmtNGwbDoN09ExwritlM7qimhueHokLpZABwPrtSJwNDIPi_4inXUdDm.T0aW2Tj2VbbCH27i

Valderi Leithardt, Dr.

Professor IPW

valderi.leithardt@isel.pt