JS-12: JavaScript Moderno

Aula 5: Funções de Callback, Promises e Async/Await

Funções de Callback

No documento correspondente à aula 2 fomos introduzidos ao conceito de **High-Order Functions** e vimos algumas das disponíveis para manipulação de Arrays como *reduce*, *filter* e *map*.

Para ajudar a refrescar sua memória uma **High-Order Function** nada mais é *que uma função que recebe como parâmetro outra função e/ou retorna uma função como resultado de sua operação*.

Vamos dar uma olhada nos exemplos que mencionamos anteriormente. Se você prestar bem atenção, todas essas funções recebem outra função como parâmetro (seja uma *function* ou uma *arrow function*).

Olha só:

```
const listaNumeros = [ 3, 4, 5, 9, 12, 6 ];
let valorInicial = 0;
const somaNumeros = listaNumeros.reduce((acumulador, valor) => {
        return acumulador + valor;
}, valorInicial);
```

A função *reduce()* dos Arrays recebe uma função como primeiro parâmetro contendo um acumulador e o valor de cada item do array. O segundo parâmetro é o valor inicial do acumulador (neste caso 0). Sendo assim, *reduce()* é uma **High-Order Function** e a função passada como parâmetro é chamada de **Callback Function**. Por quê? *Funções de Callback são funções que são passadas como argumentos para serem "chamadas", ou seja, executadas mais tarde pela função que as recebe (geralmente quando algo específico acontecer)*.

No caso da função *reduce()*, um looping é executado internamente que pega cada valor do array e o valor atual do acumulador e passa para a função de callback que especificamos. Para cada rodada do looping interno da função *reduce()* nossa função de callback é executada com os valores atualizados e junto de sua execução, toda a lógica que especificamos para ela fazer (ou seja, pegar cada valor passado e somar ao acumulador). Viu só? *Funções de Callback são funções que são executadas DEPOIS*; quando for necessário; quando algo acontecer. No caso da reduce, a cada rodada do looping interno da função.

Mas vejamos outro exemplo que talvez deixe o conceito ainda mais claro. Vimos alguns exemplos onde adicionamos eventos a elementos da interface como botões e caixas de texto. Um evento muito comum para botões é o evento “CLICK”.

Para adicionar eventos a objetos no JavaScript, temos a função *addEventListener()* e o que ela recebe como parâmetros? O primeiro argumento é o nome do evento (‘click’, ‘mouseenter’, ‘keyup’, etc) e o segundo argumento é UMA FUNÇÃO, ou seja, *aquilo que deve ser executado quando evento acontecer*. Logo, estamos aqui *passando uma função que será executada DEPOIS*. Quando? *Quando algo acontecer, e esse algo é o evento que está sendo monitorado*.

Sendo assim, a função *addEventListener()* é uma **High-Order Function** (porque ela recebe uma função como parâmetro) e a função que passamos para ser executada quando o evento ocorrer é uma **Função de Callback**.

```
$meuBotao.addEventListener( “click”, function() {
        // algo a ser executado quando o click acontecer no $meuBotão
} );
```

Fácil, né? :)

ES6 e a introdução das Promises

Há situações em que dependemos do resultado de uma operação ser terminado para podermos executar outro. Isso é muito comum quando consultamos serviços externos (uma API qualquer) e precisamos fazer algo com o resultado obtido da consulta a essa API. O grande problema desse tipo de cenário é que, como dependemos de um recurso externo, não temos certeza de quanto tempo irá demorar até que o servidor onde está a API responda a nossa solicitação.

Outro problema é que operações síncronas (executadas uma após a outra na sequência em que foram escritas) travam a execução do programa caso haja alguma lógica que leve muito tempo para ser executada. Olhando para o cenário apresentado acima, ao realizarmos uma consulta a uma API externa de forma síncrona e o servidor demorar 4 minutos para responder, isso quer dizer que a sua aplicação ficará 4 minutos congelada até que a resposta seja recebida e processada pelo JS. Isso não é muito legal!!!

O ideal para lidar com operações cujo tempo de execução é desconhecido é realizá-las de forma ASSÍNCRONA, ou seja, como se estivessem fora da thread principal de execução. Até a versão 5 do ECMAScript lidávamos com esse tipo de situação utilizando **Funções Callback** e eventos. Por exemplo, é muito comum utilizarmos AJAX para realizar consultas em APIs diversas e através do evento “load” existente no objeto XMLHttpRequest (utilizado para realizar

operações AJAX) sabíamos quando os dados carregados estavam disponíveis. Desse modo, quando o evento “load” do AJAX fosse executado, a **Função de Callback** seria executada e nela recebíamos os dados enviados pela API.

```
const ajax = new XMLHttpRequest();
… toda a configuração do Ajax necessária…
ajax.send(); // realiza a requisição para carregamento de dados externos
ajax.addEventListener(“load”, function(dadosCarregados) {
        // função callback usada quando os dados são carregados da API (load)
});
```

Assim, operações assíncronas foram feitas durante muito tempo através da combinação de eventos e funções de callback.

Contudo o ES6 trouxe uma solução bem mais sofisticada e muito aguardada: *as Promises*.

Promises é uma maneira bem mais sofisticada para lidar com operações assíncronas no JavaScript, pois além de deixar o código muito mais legível e organizado ela evita um grande problema criado pelo uso excessivo das funções de callback amplamente conhecido na área de programação como **CALLBACK HELL**: *quando temos múltiplas operações assíncronas, cada uma dependendo do resultado da anterior e, sendo organizadas em callbacks, obriga o programador a colocar uma função de callback dentro da outra, tornando o código um verdadeiro inferno e extremamente difícil de ser mantido e entendido*.

Dica.: *para mais informações sobre Callback Hell visite* http://callbackhell.com

Uma *Promise* nada mais é que um objeto JavaScript *que armazena em si o estado de uma operação assíncrona qualquer com a promessa de que em algum momento a operação que está sendo executada terá um desfecho, isto é, de que uma resposta será retornada, seja ela uma resposta positiva (status “resolved”) ou negativa (status “rejected”, associado a um erro de execução que pode ter ocorrido)*.

Dependendo do resultado obtido (“resolved” ou “rejected”), geralmente queremos que uma função seja executada. Ex: ao carregarmos os dados de uma API queremos executar alguma operação que utilize esses dados de alguma forma (mostrar numa tabela ou coisa do tipo), ou, caso algum erro ocorra no processo, queremos executar outra função que mostre uma mensagem de erro ao usuário.

Para isso, os objetos do tipo Promise disponibilizam dois métodos: para resultados positivos (“resolved”) o método *then()* deve ser utilizado. E para resultados negativos (“rejected”) o método *catch()* será executado.

Vejamos um exemplo:

```
// carregarDadosApi() retorna um objeto PROMISE contendo a possibilidade
// de 2 respostas: a que contém os dados da Climatempo ou a que contém um erro
carregarDadosApi("http://api.climatempo.com.br/")
    .then( function(dadosClimatempo) {
        // OK, usamos os dados da climatempo
    } )
    .catch( function(erro) {
        // se algo deu errado, o método catch() será executado
        // e podemos fazer alguma coisa com o erro reportado
    } );
```

Se uma função passada para o método then() ou para o método catch() também retornar uma Promise como valor (embora no caso do catch() isso seja incomum), podemos encadear vários métodos then(). Isso é frequente quando temos várias operações assíncronas que devem ser executadas exatamente uma após a outra.

Veja um exemplo:

```
// função que realiza a reserva de uma cadeira no avião
// essa função retorna uma Promise
realizarReservaCadeira(55)
    .then( salvarDadosComprador() ) // também retorna uma Promise
    .then( realizarPagamento() ) // também retorna uma Promise
    .then( enviarComprovantePorEmail() ) // NÃO retorna uma promise
    .catch( (erro) => {
        // se algum erro acontecer em qualquer um dos then()
        // o catch() será executado.
        alert("Erro ao realizar operação: " + erro);
    } );
```

Veja bem: cada operação depende que a anterior termine antes de ser executada. Só que não temos como saber quanto tempo cada operação irá levar. *salvarDadosComprador()* pode levar vários minutos, assim como *realizarPagamento()*. Além disso, temos que ter certeza de que uma será executada imediatamente após a outra, pois só podemos realizar o pagamento se a reserva da cadeira já foi feita e se os dados do comprador já foram salvos, senão, teremos um erro no processo. Ainda mais importante: NÃO PODEMOS BLOQUEAR A EXECUÇÃO DO PROGRAMA por conta dessas operações! Logo, elas precisam ser assíncronas, isto é, precisam executar de forma "paralela" ao restante do programa.

Tudo isso já é resolvido pelas Promises. Cada *then()* será executado após o término da operação anterior e tudo isso de forma assíncrona e com código ordenado e organizado (sem

funções de callback uma dentro da outra e eventos “load”). Temos assim um código assíncrono organizado, fácil de ler e de manter. :)

Para exemplos de *Promises*, consulte os arquivos de exemplo comentados e utilizados em sala de aula. ;)

Async/Await: código assíncrono que tem cara de “síncrono”.

Pois bem, ainda estamos falando de Promises e assincronicidade em JavaScript. O uso dos métodos then() e catch() já ajudaram bastante na organização do código se comparados com as funções de callback (e o temido callback hell). Porém, quando temos diversas operações assíncronas, uma após a outra, ainda que dê pra entender o código, ele ainda pode parecer um pouco “feio”.

Veja:

```
operacao1()
  .then(operacao2())
  .then(operacao3())
  .then(operacao4())
  .then(operacao5())
  .catch(erro => console.log(erro));
```

Ficar colocando vários then()... then()... then() pode deixar nosso código estranho. Legível, porém estranho.

E se pudéssemos escrever um código *assíncrono* que parece *síncrono*? Bom, sabemos que um código *síncrono* é basicamente aquele em que cada instrução é executada uma após a outra e que se uma determinada operação demorar muito para ser executada, todo o programa ficará congelado até que a operação se complete.

Num código síncrono, a operação acima ficaria mais ou menos assim:

```
function processarReserva() {
        operacao1();
        operacao2();
        operacao3();
        operacao4();
        operacao5();
}
```

```
processarReserva(); // executamos a função
```

Certo… mas sabemos que a tela ficaria congelada até que cada uma das operações fosse finalizada.

Daria pra escrever um código com essas características (uma instrução após a outra, do jeito como estamos acostumados a escrever) e ainda assim fazer com que ele seja executado de forma assíncrona (ou seja, sem travar a execução da thread principal)?

A resposta é SIM!!! Para esses casos foram introduzidas as palavras chave **async** e **await**.

O comando **async** é utilizado na declaração de uma função (ou arrow function) que deve ser executada de forma assíncrona. A palavra **await** é utilizada na frente da execução de funções que retornam PROMISES dentro de uma função do tipo **async**. Essa palavra indica que o programa deve "esperar" (await) a Promise ser processada antes de ir para a próxima linha de execução. Ou seja, é uma forma diferente de escrever o then().

***Obs.: async** e **await** são SEMPRE utilizadas em conjunto. SEMPRE*.

Veja como o código anterior ficaria com o uso de async/await:

```
async function processarReserva() {
    await operacao1();
    await operacao2();
    await operacao3();
    await operacao4();
    await operacao5();
}

processarReserva(); // executamos a função
```

Para tratar erros utilizamos **try… catch** em funções **async**. Exatamente como faríamos no tratamento normal de erros.

Nosso código para realizar reserva de cadeiras ficaria assim numa versão usando async/await:

```
async function processarReserva() {
    try {
        await realizarReservaCadeira(55);
        await salvarDadosComprador();
        await realizarPagamento();
        // como não retorna uma promise, não precisamos do await
```

```
            enviarComprovantePorEmail();
        }
        catch(erro) {
            alert("Erro ao realizar operação: " + erro);
        }
}
```

Situações mais comuns para operações assíncronas:

- Requisições Ajax: carregamento ou envio de informações pela internet;
- Leitura ou manipulação de arquivos locais;
- Acesso e realização de operações em banco de dados no browser ou no servidor;

Diversas funções no JS hoje retornam promises. Por exemplo, a função *fetch()* disponível nos navegadores modernos possibilita a realização de chamadas AJAX (para carregamento ou envio de informações do/para o back-end) e já trabalha com PROMISES possibilitando o uso de then()/catch() ou async/await.

Para maiores informações sobre a função fetch(), visite:
https://developer.mozilla.org/pt-BR/docs/Web/API/Fetch_API/Using_Fetch