Sintaxe e Organização do Javascript

1. Funções e Métodos

Agora que entendemos escopos, devemos ver mais sobre funções, suas variantes e mais detalhes.

Funções

E a partir daqui entram as funções, que servem como procedimentos em Javascript criando um escopo. Elas são fragmentos de código que podem ser chamados por outros códigos, por si mesmo ou até mesmo uma variável que se refere à função.

Existem diferentes tipos de funções

- Anônimas

```
function () {};
() => {}; // arrow notation
```

- Nomeadas

```
function foo() {};
const foo = () => {}; // arrow notation
```

- Internas

  São funções dentro de outra função (`square` nesse caso)

```
function addSquares(a,b) {
   function square(x) {
      return x * x;
   }
   return square(a) + square(b);
};

const addSquares = (a,b) => { // arrow notation
   const square = x => x*x;
   return square(a) + square(b);
};
```

- Recursivas

  São funções que invocam a si mesma. A recursão é usada para resolver problemas que contêm subproblemas menores. Uma função recursiva pode receber dois inputs: um caso base que termina a recursão ou um caso recursivo que a chama de novo.

```
function loop(x) {
   if (x >= 10) return;
```

```
    loop(x + 1);
};

const loop = x => { // arrow notation
    if (x >= 10) return;
    loop(x + 1);
};
```

- Invocada Imediatamente

  Como o nome sugere, são chamadas logo após o código ser compilado pelo navegador. São funções uteis para separar o escopo global do navegador da aplicação. Assim variáveis que são declaradas dentro desse contexto são criadas sem a possibilidade de outro código às acessarem. A maneira de identificar um IIFE é localizar os parênteses extra esquerdo e direito no final da declaração da função.

```
(function foo() {
    console.log("Hello Foo");
}());

(() => { // arrow notation
    console.log("Hello Food");
})();
```

Argumentos e Parâmetros

Logo no começo dessa sessão vamos ter um padrão de entendimento:

> 💡 **Argumentos** são valores enviados para funções e **parâmetros** são variáveis recebidas pela chamada da função e usadas dentro de seu escopo.

Logo:

```
function foo(param1, param2) {
  console.log(param1, param2);
}

const arg1 = 1
const arg2 = 2

foo(arg1, arg2) // 1, 2
```

Arrow Functions

Essas são as queridinhas de quase todo desenvolvedor web, são fáceis de entender e possuem uma sintaxe parecida com a criação de variáveis comuns. Como acabamos escrevendo menos caiu no uso e esta sendo de longe usada por grande parte das pessoas.

```
const newFunction1 = () => 0 // newFunction retorna 0, essa sintaxe é igual que:
const newFunction2 = () => {
```

```
  return 0
}
```

Elas possuem algumas diferenças que são muito importantes em relação às funções convencionais. Uma dessas coisas é que elas não identificam o uso do `this`:

```
const me = {
  name: "Gustavo Vasconcellos",
  thisInArrow: () => {
    console.log("My name is " + this.name);
  },
  thisInRegular: function() {
    console.log("My name is " + this.name);
  }
}

me.thisInArrow() // "My name is "
me.thisInRegular() // "My name is Gustavo Vasconcellos"
```

Outra coisa que falta nelas é a variável de escopo de funções `arguments` que lista todos os argumentos que são enviados nela:

```
const foo = {
  argumentsInRegular() { // Outra forma de declarar funções em objetos
    console.log(arguments)
  },
  argumentsInArrow: () => {
    console.log(arguments)
  }
}

foo.argumentsInRegular(1,2,3,4) // [1,2,3,4, ...]
foo.argumentsInArrow(1,2,3,4) // ReferenceError: arguments is not defined
```

Por mais que ***arrow functions*** possuam essas limitações elas são usadas na grande maioria das vezes pelos desenvolvedores sendo hoje em dia uma grande forma de padronização do código de projetos.

2. Callbacks

Esse é um assunto que muitos desenvolvedores tem cuidado ao falar sobre, a maioria até sabe o que é mas não entende a diferença entre funções normais e *callbacks*. Então vamos começar falando que eles são necessários em fluxo de funções **assíncronas**, onde uma função precisa esperar outra função terminar para ser executada, como **esperar um arquivo carregar.**

Antes de aprofundar sobre *Callbacks,* vamos falar sobre **Assíncrono e Síncrono.**

Assíncrono e Síncrono

Normalmente, o código de uma aplicação é executada com uma coisa acontecendo por vez. Se uma função depende do resultado de outra função, ela tem que esperar o retorno e até que isso aconteça, o programa inteiro praticamente “para de funcionar” da perspectiva do usuário.

```
function expensiveOperation() {
  for(let i = 0; i < 1000000; i++) {
    ctx.fillStyle = 'rgba(0,0,255, 0.2)';
    ctx.beginPath();
    ctx.arc(random(0, canvas.width), random(0, canvas.height), 10, degToRad(0), degToRad(360), false);
    ctx.fill()
  }
}

fillBtn.addEventListener('click', expensiveOperation);

alertBtn.addEventListener('click', () =>
  alert('You clicked me!')
);
```

O código acima simula 1 milhão de alterações em um elemento `canvas`, adicionando vários círculos nele, como tem um botão que executa uma função que exige bastante do navegador a função atrelada ao outro botão que abre um alerta simplesmente não abre até que a tarefa anterior termine. Pois Javascript tem apenas uma Thread de execução.

Em poucas palavras, **síncrono** ocorre em tempo real como as funções que vimos até agora que são executadas e após terminarem continuam a executar o código. **Assíncrono** é o contrario, ou melhor, é uma instrução que espera em "segundo plano" que algo esteja pronto antes de executar, você pode pensar nos eventos do navegador como um exemplo de forma assíncrona.

> ⚠️ Atenção sempre ao enviar funções via parâmetro, elas devem ir sempre sem os parêntesis, se você enviar com eles você estará executando a função e na verdade enviando o valor de retorno dela.

3. Promises

Este é o assunto mais importante do nosso curso antes de entrarmos em React, isso porque Promises são basicamente as formas que mais utilizaremos e veremos sendo usadas em React, isso porque… Bom, vamos deixar isso para depois.

Por enquanto vamos nos atentar a sintaxe das Promises, e vamos usar esse código para os próximos exemplos também:

```
const promise = new Promise((resolve, reject) => {
  console.log('Isso é uma Promise');

  resolve();
})

const funcPromise = (condition) => {
  return new Promise((resolve, reject) => {
    console.log('Isso é uma Promise');

    if (condition) {
      resolve();
    }
    reject();
```

```
  })
}
```

Basicamente essas são as formas de criar uma Promise, mas isso não importa tanto, a verdade é que é raro no dia a dia termos que criar uma, mas claro que precisamos passar por isso para entender como funcionam.

Promise é uma estrutura assíncrona e assim temos dois casos a se atentar, `resolve` e `reject`. Resolve é uma função de resposta que significa que ela foi terminada sem erros, reject é outra função que para a execução da Promise e devolve um erro. Como são funções, podemos enviar os argumentos que forem necessários para elas e assim utiliza-los posteriormente.

Mas usar aonde? Para isso quando uma Promise é chamada ela é automaticamente executada e depois temos outros dois carinhas `then` e `catch`.

```
promise.then(() => console.log('then'))

funcPromise(true)
  .then(() => console.log('then'))
  .catch(() => console.log('catch'))
```

A primeira linha exibirá apenas o log do "then", isso acontece porque a Promise ja foi executada na atribuição da variável e a única coisa que ficamos é com o resultado do `resolve`. Ja a execução da `funcPromise` é diferente porque a Promise esta sendo criada sempre que chamamos a função, isso faz com que ela sempre mostre o texto “Isso é uma Promise” e além disso, condicionalmente, se mandarmos o valor `true` ou `false` ela vai executar o `then` ou o `catch` da Promise.

Geralmente usamos Promises que foram implementadas da segunda forma, onde uma função retorna uma Promise e podemos passar argumentos para ela.

Async/Await

E quando não queremos que a nossa Promise seja executada dessa forma? Quando na verdade queremos que ela execute como uma função normal, linha por linha? Afinal de contas, as Promises sempre vão executar o `then` quando terminar a execução. Para isso existe o `await`.

4. Loops e Iterações

Vamos começar falando de funções puras, são essas as que **NÃO ALTERAM** a variável diretamente. Já as impuras são funções que **ALTERAM** diretamente a variável. Vamos considerar o array abaixo para todos os exemplos abaixo.

```
const array = [1, 2, 3, 4, 5]
```

E a partir disso vamos seguir sobre esses tipos de funções:

```
// Adicionar um item no array de forma PURA
const newArray = array.concat([6])
// array -> [1, 2, 3, 4, 5]
// newArray -> [1, 2, 3, 4, 5, 6]

// Adicionar um item no array de forma IMPURA
array.push(6) // Mesma coisa que: array = array.concat([6])
// array -> [1, 2, 3, 4, 5, 6]
```

Podemos ver com isso que `concat` e `push` possuem resultados parecidos, enquanto `concat` cria e retorna um array novo modificado `push` vai alterar diretamente a variável que chamou o método mas retorna o valor adicionado.

Embora os dois métodos sejam parecidos e façam a mesma coisa - adicionar ao final do array - eles são dois divisores de código. Escrever métodos com funções puras são na maioria das vezes mais complexos de se escrever mas acabam sendo mais fáceis de serem lidos por outros desenvolvedores. Em contrapartida escrever código com funções impuras acaba sendo mais simples por simplesmente nos preocuparmos com apenas uma variável mas cria uma complexidade no futuro de manutenção e as vezes de entendimento por outros desenvolvedores.

Dito isso, vamos entrar em métodos puros de arrays. Esses métodos recebem um ***callback*** que é executado **para cada posição do array** e cada um funciona de uma forma:

.map()

Método utilizado para **transformar** os itens do array, é muito utilizado em React para transformar itens que são objetos, strings e etc em ***Componentes.*** Se acalme, logo estaremos falando de React mas por enquanto foquemos em JS. O retorno dele é o novo array:

Parâmetros:

- `current` → referencia **o valor** da posição
- `index` → referencia **a posição**
- `array` → referencia **o array inicial** sem alterações

```
const duplicatedArray = array.map((current, index, array) => {
  return current * 2
}) // [2, 4, 6, 8, 10]
```

.filter()

Método para **filtrar** um array, obtendo **o mesmo tipo de array** porem com menos (ou a mesma quantidade) de posições. Diferente do `map`, invés de retornarmos algum valor no ***callback*** devemos retornar `true` (caso essa posição deva existir no novo array) ou `false` (caso essa posição não deva existir no novo array).

Parâmetros:

- `current` → referencia **o valor** da posição
- `index` → referencia **a posição**
- `array` → referencia **o array inicial** sem alterações

```
const evenArray = array.filter((current) => {
  return current % 2 === 0
}) // [2, 4]
```

.some() e .find()

Ambos métodos servem para **encontrar um valor** dentro do array mas o que importa aqui é o **retorno** de cada um. O `some` retornará um valor ***booleano*** - `true` ou `false` - já o `find` retornará **o valor**.

O ***callback*** desses métodos precisa que o retorno dele seja um valor ***booleano*** (ou melhor, valores verdadeiros como um objeto ou um numero diferente de zero é entendido como verdadeiro) e quando encontrar o primeiro valor verdadeiro o método será parado e retornado.

Para definir quando usar um ou outro defina bem **o que você precisa**, se apenas quiser saber **se algo existe** então siga com `some`, se você precisar **obter o valor especifico** de alguma posição do array use `find`.

Parâmetros:

- `current` → referencia **o valor** da posição
- `index` → referencia **a posição**
- `array` → referencia **o array inicial** sem alterações

```
const thereIsTwoOrFive = array.some((current) => {
  return current === 2 || current === 5
}) // true

const four = array.find((current) => {
  return current === 4
}) // 4
```

.reduce()

Esse aqui é complexo, mas é colírio para os olhos quando bem usado. Como o nome sugere, ele busca **reduzir** um array iterando por cada elemento dessa lista com o objetivo de gerar um único valor.

Esse método diferente dos outros, em que enviamos apenas o ***callback,*** possui dois parâmetros: o ***callback*** e um **valor inicial**. Esse valor inicial é referenciado pelo primeiro parâmetro do nosso ***callback***:

- `initial` → referencia **o acumulador (ou valor anterior)** do método
- `current` → referencia **o valor** da posição

- `index` → referencia **a posição**
- `array` → referencia **o array inicial** sem alterações

```
const sumArray = array.reduce((initial, current) => {
  return initial + current
}, 0) // 15

const sumOnlyEvenValues = array.reduce((initial, current) => {
  if (current % 2 !== 0) {
    return initial + current
  }
  return initial
}, 0) // 9
```

Perceba que no ***callback*** do `reduce` precisamos sempre retornar o `initial`, isso acontece pois o valor retornado em uma iteração será usado na próxima. Logo, vamos pensar no **teste de mesa** da função `sumOnlyEvenValues`.

iteração (`index`)	initial	current	current % 2 !== 0	return
0	0	1	`true`	1
1	1	2	`false`	1
2	1	3	`true`	4
3	4	4	`false`	4
4	4	5	`true`	9

Se existisse alguma condição que não retornasse valor no ***callback*** estaríamos basicamente criando um comportamento para que ele desse o valor `undefined` para o `initial` na próxima iteração e dependendo do nosso método ele daria algum erro ou o valor final não seria o que esperássemos.

5. Uso de Formulários

Formulários são usados para obter dados do usuário e com isso temos uma série de funcionalidades e tags HTML para conhecer então vamos começar com alguns elementos HTML importantes:

<input>

Este elemento é responsável por receber o valor que o usuário der para ser utilizado posteriormente. A partir dessa tag é possivel gerar campos de texto, caixas de seleção e até botões, tudo pelo atributo `type`:

Tipo	**Descrição**	Exemplo
<input type="text">	Exibe um campo de texto de uma linha	Campo de nome e sobrenome

Tipo	Descrição	Exemplo
<input type="radio">	Exibe botões para selecionar **uma** de varias opções	Botões para escolher o sexo
<input type="checkbox">	Exibe um botão de seleção para verificar algo	Botão para concordar com os termos de uso
<input type="submit">	Exibe um botão com a função de enviar um formulário	Botão “Enviar” em formulários
<input type="button">	Exibe um botão	Botão “Adicionar ao carrinho” em lojas virtuais

<label>

Elemento para definir um rótulo, um nome, para um campo de um formulário. Esse elemento é importante pois é a partir dele que leitores de tela vão identificar campos de uma formulário. Vamos falar mais de Acessibilidade no futuro, mas tratar bem formulários é o segundo passo para entrarmos nesse assunto, o primeiro passo é HTML Semântico, mas também veremos no futuro.

Ele também serve para ajudar cliques em elementos pequenos de mais como caixas de seleção ou verificação mas para que isso funcione devemos usar o atributo `for` que recebe o `id` do `input` que será vinculado com ele.

```
<form>
  <label for="fname">First name:</label><br>
  <input type="text" id="fname" name="fname"><br>
  <label for="lname">Last name:</label><br>
  <input type="text" id="lname" name="lname">

  <label>What do you prefer?</label><br>
  <input type="radio" id="html" name="fav_language" value="HTML">
  <label for="html">HTML</label><br>
  <input type="radio" id="css" name="fav_language" value="CSS">
  <label for="css">CSS</label><br>

  <label>What do you prefer?</label>
  <input type="checkbox" id="agree" name="agree">
  <label for="agree">Do you agree?</label>

  <input type="submit" value="Submit">
</form>
```

O `input` tem diversos atributos que manipulam os valores que são colocados e o comportamento dos campos, para ver mais sobre eles acesse a referencia com a mesma ideia de que é importante saber que existe e não decora-los:

https://www.w3schools.com/html/html_form_attributes.asp

<form>

Agora sim, com um conceito básico podemos usar para esse elemento. A complexidade dele se da pela diversas formas que podemos utiliza-lo. Em aula vimos como o atributo `action` se

comporta junto do `method` e que também podemos usar o evento `submit` para poder manipular como quisermos o formulário.

Action e Method

Utilizar essa funcionalidade demanda que utilizemos um formulário muito bem tratado com os elementos e atributos HTML para garantir o funcionamento dele. Por mais que possamos utilizar os eventos de `submit` juntos pode gerar uma complexidade de entendimento e de manutenção.

- `action` → Define um caminho para enviar os dados que foram colocados no formulário
- `method` → Define o método da requisição ao enviar o formulário

Submit event

Este é mais flexível e customizável, serve perfeitamente quando temos visuais específicos para cada situação dos campos como erros, validações e exibições. Porem utilizar o evento de `submit` traz a responsabilidade de tratar no código esses erros e validações - se algum campo for obrigatório mas não tiver uma condição que trate isso erros acontecerão no teu formulário.

E lembre-se de utilizar o método `event.preventDefault()` que eventos possuem para evitar o recarregamento da página quando o usuário enviar o formulário.