Java script (ES6)

Ao definir funções dentro de objetos em ES5, nós temos de usar a palavra-chave `function` como se segue:

```
const person = {
  name: "Taylor",
  sayHello: function() {
    return `Hello! My name is ${this.name}.`;
  }
};
```

Com ES6, você pode remover a palavra-chave `function` e dois pontos ao definir funções em objetos. Aqui está um exemplo dessa sintaxe:

```
const person = {
  name: "Taylor",
  sayHello() {
    return `Hello! My name is ${this.name}.`;
  }
};
```

Exercicio = Refatore a função `setGear` dentro do objeto `bicycle` para usar a sintaxe curta descrita acima.

```
// Altere apenas o código abaixo desta linha

const bicycle = {

  gear: 2,

  setGear(newGear) {

    "use strict";

    this.gear = newGear;

  }

};
```

```
// Altere apenas o código acima desta linha

bicycle.setGear(3);

console.log(bicycle.gear);
```

Usar a sintaxe de classe para criar uma função construtora

ES6 fornece uma nova sintaxe para criar objetos, usando a palavra-chave *class*.

Deve ser notado que a sintaxe `class` é apenas sintaxe, um *syntatical sugar*. JavaScript ainda não oferece suporte completo ao paradigma orientado a objetos, ao contrário do que acontece em linguagens como Java, Python, Ruby, etc.

No ES5, geralmente definimos uma função construtora (`constructor` function) e usamos a palavra-chave `new` para instanciar um objeto.

```
var SpaceShuttle = function(targetPlanet){
  this.targetPlanet = targetPlanet;
}
var zeus = new SpaceShuttle('Jupiter');
```

A sintaxe `class` simplesmente substitui a criação da função construtora (`constructor`):

```
class SpaceShuttle {
  constructor(targetPlanet) {
    this.targetPlanet = targetPlanet;
  }
}
const zeus = new SpaceShuttle('Jupiter');
```

Deve ser notado que a palavra-chave `class` declara uma nova função, para qual um construtor é adicionado. Esse construtor é invocado quando `new` é chamado na criação de um novo objeto.

Observação: UpperCamelCase (observe a primeira letra de cada palavra em maiúsculo) deve ser usado por convenção para nomes de classe no ES6, como em `SpaceShuttle` usado acima.

O método `constructor` é um método especial usado para inicializar um objeto criado com uma classe. Você aprenderá mais sobre isso na seção Programação Orientada a Objetos da Certificação de Algoritmos e Estruturas de Dados JavaScript.

Exercicio = Use a palavra-chave `class` e declare o método `constructor` para criar a classe `Vegetable`.

A classe `Vegetable` permite criar um objeto vegetal com a propriedade `name` que é passada ao `constructor`.

```
// Altere apenas o código abaixo desta linha
class Vegetable {
  constructor(name) {
    this.name = name;
  }
}


// Altere apenas o código acima desta linha


const carrot = new Vegetable('carrot');
console.log(carrot.name); // Deve exibir 'carrot'
```

Usar getters e setter para controlar acesso a um objeto

Você pode obter valores de um objeto e definir o valor da propriedade dentro de um objeto.

Esses são classicamente chamados de *getters* e *setters*.

Funções getter tem a finalidade de simplesmente retornar (get) o valor de uma variável privada de um objeto para o usuário sem que o usuário acesse diretamente a variável privada.

Funções setter tem a finalidade de modificar, ou definir (set), o valor de uma variável privada de um objeto baseado no valor passado dentro da função setter. Essa mudança poderia envolver cálculos, ou até sobrescrever completamente o valor anterior.

```
class Book {
  constructor(author) {
    this._author = author;
  }
  // getter
  get writer() {
    return this._author;
  }
  // setter
  set writer(updatedAuthor) {
    this._author = updatedAuthor;
  }
}
const novel = new Book('anonymous');
console.log(novel.writer);
novel.writer = 'newAuthor';
console.log(novel.writer);
```

O console exibirá as strings `anonymous` e `newAuthor`.

Note a sintaxe usada para invocar o getter e setter. Eles nem sequer se parecem com funções. Getters e setters são importantes porque escondem os detalhes internos da implementação.

Observação: é uma convenção preceder o nome de uma variável privada com um underscore (_). No entanto, essa prática por si só não torna uma variável privada.

Exercicio =Use a palavra-chave `class` para criar a classe `Thermostat`. O `constructor` aceita uma temperatura Fahrenheit.

Na classe, crie um `getter` para obter a temperatura em Celsius e um `setter` para definir a temperatura em Celsius.

Lembre-se de que `C = 5/9 * (F - 32)` e `F = C * 9.0 / 5 + 32`, aonde `F` é o valor da temperatura em Fahrenheit e `C` é o valor da mesma temperatura em Celsius.

Observação: quando você implementa isso, você vai rastrear a temperatura dentro da classe em uma escala, ou Fahrenheit ou Celsius.

Esse é o poder de um getter e um setter. Você está criando uma API para outro uso, que pode pegar o resultado correto independente de qual está rastreando.

Em outras palavras, você está abstraindo detalhes de implementação do usuário.

```
// Altere apenas o código abaixo desta linha
class Thermostat {
  constructor(fahrenheit) {
    this.fahrenheit = fahrenheit;
  }

  get temperature() {
    return (5 / 9) * (this.fahrenheit - 32);
  }

  set temperature(celsius) {
    this.fahrenheit = (celsius * 9.0) / 5 + 32;
  }
}
```

```
// Altere apenas o código acima desta linha
```

```
const thermos = new Thermostat(76); // Definição na escala Fahrenheit
let temp = thermos.temperature; // 24,44 em Celsius
thermos.temperature = 26;
temp = thermos.temperature; // 26 em Celsius
```

Separar seus scripts em módulos

O JavaScript nasceu com com o objetivo de cumprir um pequeno papel em uma web onde tudo era, na maior parte, HTML. Hoje, o JavaScript é gigante. Para se ter noção, alguns websites são construídos quase que inteiramente em JavaScript. A fim de tornar o JavaScript mais modular, limpo e passível de manutenção, a versão ES6 introduziu uma forma mais simples de compartilhar códigos entre arquivos JavaScript. Dessa forma, você consegue exportar partes de um arquivo e usá-los em arquivos externos bem como importar as partes de que você precisa. Para tirar proveito dessa funcionalidade, você precisa criar uma tag script com o atributo `type` de valor `module` no seu documento HTML. Exemplo:

```
<script type="module" src="filename.js"></script>
```

O script do exemplo acima agora é um módulo (`module`) e pode usar os recursos `import` e `export` (você aprenderá sobre eles nos próximos desafios).

Exercicio = Adicione uma tag script ao documento HTML do tipo `module` e dê a ela o caminho do arquivo `index.js`

```
<html>
  <body>
```

```
    <!-- Altere apenas o código abaixo desta linha -->
<script type="module" src= "index.js"> </script>
    <!-- Altere apenas o código acima desta linha -->
  </body>
</html>
```

Usar a exportação para compartilhar um bloco de código

Imagine um arquivo chamado `math_functions.js` que contém várias funções relacionadas a operações matemáticas. Uma delas é armazenada em uma variável, `add`, que recebe dois números e retorna a soma deles. Você quer usar essa função em diversos arquivos JavaScript diferentes. Para compartilhá-las com esses outros arquivos, você primeiro precisa exportá-las (`export`).

```
export const add = (x, y) => {
  return x + y;
}
```

Acima está uma forma comum de exportar uma única função, mas você pode alcançar a mesma coisa da seguinte forma:

```
const add = (x, y) => {
  return x + y;
}

export { add };
```

Quando você exporta uma variável ou função, você pode importá-las em outro arquivo e usá-las sem ter de rescrever o código. Você pode exportar várias coisas ao repetir o primeiro exemplo para cada coisa que você queira

exportar, ou ao colocar todas elas em uma instrução de export do segundo exemplo, da seguinte forma:

```
export { add, subtract };
```

EXERCICIO = Há duas funções relacionadas a string no editor. Exporte ambas usando o método de sua escolha.

```
export const uppercaseString = (string) => {
  return string.toUpperCase();
}
```

```
 export const lowercaseString = (string) => {
  return string.toLowerCase()
}
```

OBS : SÓ COLOCAR EXPORT NA FRENTE!

Reutilizar código JavaScript usando import

`import` te permite escolher quais partes carregar de um arquivo ou módulo. Na lição anterior, os exemplos exportaram a função `add` do arquivo `math_functions.js`. Você pode importá-la e usá-la em outro arquivo assim:

```
import { add } from './math_functions.js';
```

Aqui, `import` encontrará a função `add` no arquivo `math_functions.js`, importar apenas essa função e ignorar o resto. O `./` diz ao import para procurar pelo arquivo `math_functions.js` no mesmo diretório que o arquivo atual. O caminho relativo do arquivo (`./`) e a extensão do arquivo (`.js`) são necessários ao usar import dessa forma.

Você pode importar mais de um item do arquivo ao adicioná-los na instrução `import` dessa forma:

```
import { add, subtract } from './math_functions.js';
```

Exercicio : Adicione a instrução `import` apropriada que permitirá o arquivo atual usar as funções `uppercaseString` e `lowercaseString` que você exportou na lição anterior. As funções estão em um arquivo chamado `string_functions.js`, o qual está no mesmo diretório que o arquivo atual.

```
import { uppercaseString, lowercaseString } from
'./string_functions.js';

// Altere apenas o código acima desta linha


uppercaseString("hello");

lowercaseString("WORLD!");
```

Usar * para importar tudo de um arquivo

Suponha que você tem um arquivo e deseja importar todo o conteúdo dele no arquivo atual. Isso pode ser feito com a sintaxe `import * as`. Aqui está um exemplo onde todo o conteúdo de um arquivo chamado `math_function.js` é importado em um arquivo no mesmo diretório:

```
import * as myMathModule from "./math_functions.js";
```

A instrução `import` acima criará um objeto chamado `myMathModule`. Esse nome é totalmente arbitrário. Você pode escolher qualquer outro nome que seja apropriado para sua aplicação. O objeto conterá todas as exportações do arquivo `math_functions.js`. Dessa forma, você pode acessar as funções exportadas da mesma forma que você acessa as propriedades de um objeto. Aqui está um exemplo de como você pode usar as funções `add` e `subtract` que foram importadas:

```
myMathModule.add(2,3);
myMathModule.subtract(5,3);
```

Exercicio = O código nesse desafio requer o conteúdo do arquivo: `string_functions.js`, o qual está no mesmo diretório que o arquivo atual. Use a sintaxe `import * as` para importar tudo do arquivo em um objeto chamado `stringFunctions`.

```
import * as stringFunctions from "./string_functions.js";

// Altere apenas o código acima desta linha



stringFunctions.uppercaseString("hello");

stringFunctions.lowercaseString("WORLD!");
```

Exportar apenas um valor com export default

Na lição de `export` você aprendeu sobre a sintaxe que chamamos de *exportação nomeada*. Naquela lição você exportou múltiplas funções e variáveis que ficaram disponíveis para utilização em outros arquivos.

Há outra sintaxe para `export` que você precisa saber, conhecida como *exportação padrão*. Você usará essa sintaxe quando apenas um valor estiver sendo exportado de um arquivo ou módulo. Essa sintaxe também é usada para exportar um valor substituto caso o valor original não possa ser exportado.

Abaixo estão exemplos utilizando a sintaxe `export default`:

```
export default function add(x, y) {
  return x + y;
}

export default function(x, y) {
  return x + y;
```

```
}
```

O primeiro exemplo é uma função nomeada e o segundo é uma função anônima.

A sintaxe `export default` é usada para exportar um único valor de um arquivo ou módulo. Tenha em mente que você não pode usar o `export default` com `var`, `let` ou `const`

Exercicio = A função a seguir deve ser o único valor a ser exportado. Adicione o código necessário para que apenas um valor seja exportado.

```
export default function subtract(x, y) {
  return x - y;
}
```

Importar uma exportação padrão

No último desafio, você aprendeu sobre `export default` e seus usos. Para importar uma exportação padrão, você precisa usar uma sintaxe diferente de `import`. No exemplo a seguir, `add` é a exportação padrão do arquivo `math_functions.js`. Veja como importá-la:

```
import add from "./math_functions.js";
```

A sintaxe é diferente em apenas um ponto. O valor importado, `add`, não está rodeado por chaves (`{}`). Aqui, `add` é simplesmente uma palavra qualquer que vai ser usada para identificar a variável sendo exportada do arquivo `math_functions.js`. Você pode usar qualquer nome ao importar algo que foi exportado como padrão.

Exercicio = No código a seguir, importe a exportação padrão do arquivo `math_functions.js` encontrado no mesmo diretório do arquivo que foi usado como exemplo. Dê a importação o nome `subtract`.

```
import subtract from "./math_functions.js";

// Altere apenas o código acima desta linha


subtract(7,4);
```

Criar uma promessa em JavaScript

Uma promessa em JavaScript é exatamente o que parece - você faz a promessa de que vai fazer uma tarefa, geralmente de forma assíncrona. Quando a tarefa é finalizada, ou você cumpriu a promessa ou falhou ao tentar. Por ser uma função construtora, você precisa utilizar a palavra-chave `new` para criar uma `Promise`. Ela recebe uma função, como seu argumento, com dois parâmetros - `resolve` e `reject`. Esses métodos são usados para determinar o resultado da promessa. A sintaxe se assemelha a:

```
const myPromise = new Promise((resolve, reject) => {

});
```

Exercicio = Crie uma nova promessa chamada `makeServerRequest`. No construtor da promessa, passe uma função com os parâmetros `resolve` e `reject`.

```
const makeServerRequest = new Promise((resolve, reject) => {
```

```
});
```

Concluir uma promessa com resolve e reject

Uma promessa possui três estados: pendente (`pending`), cumprida (`fulfilled`) e rejeitada (`rejected`). A promessa que você criou no desafio anterior está presa no estado `pending` para sempre porque você não adicionou uma forma de concluir a promessa. Os parâmetros `resolve` e `reject` passados para o argumento da promessa servem para este propósito. `resolve` é utilizado quando a promessa for bem-sucedida, enquanto `reject` é utilizado quando ela falhar. Ambos são métodos que recebem apenas um argumento, como no exemplo abaixo.

```
const myPromise = new Promise((resolve, reject) => {
  if(condition here) {
    resolve("Promise was fulfilled");
  } else {
    reject("Promise was rejected");
  }
});
```

O exemplo acima usa strings como argumento desses métodos, mas você pode passar qualquer outro tipo de dado. Geralmente, é passado um objeto para esses métodos. Assim você pode acessar as propriedades deste objeto e usá-las em seu site ou em qualquer outro lugar.

Exercicio = Adapte a promessa para ambas as situações de sucesso e fracasso. Se `responseFromServer` for `true`, chame o método `resolve` para completar a promessa com sucesso. Passe a string `We got the data` como argumento para o método `resolve`. Se `responseFromServer` for `false`, passe a string `Data not received` como argumento para o método `reject`.

```
const makeServerRequest = new Promise((resolve, reject) => {
```

```
  // responseFromServer representa uma resposta de um
servidor
  let responseFromServer;



  if(responseFromServer) {
     resolve ("We got the data"); // Altere esta linha
  } else {
    reject ("Data not received");// Altere esta linha
  }
});
```

Manipular uma promessa cumprida usando o then

Promessas são úteis quando você tem um processo que leva uma quantidade de tempo desconhecido para ser finalizado (ou seja, algo assíncrono). Muitas vezes, uma requisição a um servidor. Fazer uma requisição a um servidor leva tempo, e após a requisição ser finalizada, você geralmente quer fazer algo com a resposta retornada. Isso pode ser feito usando o método `then`. O método `then` é executado imediatamente após a promessa ser cumprida com `resolve`. Exemplo:

```
myPromise.then(result => {

});
```

O parâmetro `result` vem do argumento dado ao método `resolve`.

Exercicio = Adicione o método `then` à sua promessa. Use `result` como parâmetro de sua função de callback e exiba `result` no console.

```
const makeServerRequest = new Promise((resolve, reject) => {
  // responseFromServer está definido como verdadeiro para representar uma resposta de sucesso de um servidor
  let responseFromServer = true;

  if(responseFromServer) {
    resolve("We got the data");
  } else {
    reject("Data not received");
  }
});
makeServerRequest.then(result => {console.log(result); });
```

Manipular uma promessa rejeitada usando o catch

`catch` é o método usado quando a promessa é rejeitada. Ele é executado imediatamente após o método `reject` da promessa ser chamado. Aqui está a sintaxe:

```
myPromise.catch(error => {

});
```

O parâmetro `error` é o argumento passado para o método `reject`.

Exercicio = Adicione o método `catch` à sua promessa. Use `error` como parâmetro de sua função de callback e exiba o valor de `error` no console.

```
const makeServerRequest = new Promise((resolve, reject) => {
```

```
  // responseFromServer está definido como falso para
representar uma resposta sem sucesso de um servidor
  let responseFromServer = false;


  if(responseFromServer) {
    resolve("We got the data");
  } else {
    reject("Data not received");
  }
});


makeServerRequest.then(result => {
  console.log(result);
});


makeServerRequest.catch(error => {
  console.log(error);
});
```