



Aula 3

Scripts web e métodos de classe

Sumário

- 1 Scripts Web
- 2 Recapitulando: Objetos, classes, construtores e atributos
- 3 Métodos de classe
- 4 Prática
- 5 Materiais de aula

Scripts Web

Um **navegador** nada mais é do que um programa capaz de **interpretar alguns arquivos**, sendo 3 formatos principais:

- HTML
- CSS
- Javascript

HTML

Indica a **estrutura** do site:

- Configurações do site
- Quais elementos ele possui
- Acesso a alguns arquivos externos

HTML

Indica a estrutura do site:

- **Configurações do site**
- Quais elementos ele possui
- Acesso a alguns arquivos externos

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta http-equiv="X-UA-Compatible" content="IE=edge">
  <meta name="viewport" content="width=device-width, initial-
scale=1.0">
  <title>Conceitos - Javascript & OOP</title>
</head>
<body>
  <h1>Conceitos</h1>
  <script async src="review1Objects.js"></script>
  <script async src="review2Classes.js"></script>
  <script async src="review3Constructors.js"></script>
  <script async src="review4Attributes.js"></script>
  <script async src="classMethods.js"></script>
</body>
</html>
```

HTML

Indica a estrutura do site:

- Configurações do site
- **Quais elementos ele possui**
- Acesso a alguns arquivos externos

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta http-equiv="X-UA-Compatible" content="IE=edge">
  <meta name="viewport" content="width=device-width, initial-
scale=1.0">
  <title>Conceitos - Javascript & OOP</title>
</head>
<body>
  <h1>Conceitos</h1>
  <script async src="review10objects.js"></script>
  <script async src="review2Classes.js"></script>
  <script async src="review3Constructors.js"></script>
  <script async src="review4Attributes.js"></script>
  <script async src="classMethods.js"></script>
</body>
</html>
```


HTML

Indica a estrutura do site:

- Configurações do site
- Quais elementos ele possui
- **Acesso a alguns arquivos externos**

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta http-equiv="X-UA-Compatible" content="IE=edge">
  <meta name="viewport" content="width=device-width, initial-
scale=1.0">
  <title>Conceitos - Javascript & OOP</title>
</head>
<body>
  <h1>Conceitos</h1>
  <script async src="review1Objects.js"></script>
  <script async src="review2Classes.js"></script>
  <script async src="review3Constructors.js"></script>
  <script async src="review4Attributes.js"></script>
  <script async src="classMethods.js"></script>
</body>
</html>
```


A tag `<script></script>` serve para carregarmos **códigos de Javascript** em uma página Web

Quando aplicamos essa tag, basta colocar qual é o caminho do arquivo que queremos carregar

```
<script async src="arquivo.js"></script>
```

```
<script async src="arquivo.js"></script>
```

O atributo **async** indica que o arquivo Javascript será carregado em **paralelo** e, **assim que disponível**, executado

1. A página começa a ser interpretada e carregada
2. O arquivo começa a ser baixado junto com o carregamento da página
3. Assim que o arquivo é carregado, ele é executado
4. Restante da página é carregada

```
<script defer src="arquivo.js"></script>
```

O atributo **async** indica que o arquivo Javascript será carregado em **paralelo** e só será executado quando a **página terminar de carregar**

1. A página começa a ser interpretada e carregada
2. O arquivo começa a ser baixado junto com o carregamento da página
3. A página termina de carregar
4. O script é executado

```
<script src="arquivo.js"></script>
```

Se **nenhum** desses atributos estiverem presentes, o arquivo é **carregado e executado imediatamente**, bloqueando o carregamento da página

1. A página começa a ser interpretada e carregada
2. Ao chegar na linha do HTML em que a tag script está, o arquivo é carregado e executado
3. O restante da página volta a ser carregado

Recapitulando

Objetos, classes, construtores e
atributos

Objetos

Recapitulando

Objetos são estruturas de programação que representam dados de uma forma mais próxima à realidade

Eles são uma das estruturas principais da **P OO** (Programação Orientada a Objetos) e **TAD** (Tipos abstratos de dados)

Objetos

Recapitulando

```
const bank = {  
  name: 'Santander',  
  description: 'A solução completa para o seu negócio',  
  departments: ['Crédito', 'Contas a pagar', 'Atendimento ao cliente'],  
  clients: [  
    { id: 1, name: 'Ada', account: '12345', digit: 'X'}  
  ]  
}
```

Classes

Recapitulando

Classes, dentro da programação, são usadas para criar objetos a partir de **estruturas pré-definidas**

Objetos são **instâncias** de classes

Para criarmos objetos a partir de classes, usamos a *keyword* **new**

Assim como os objetos, elas são uma das estruturas principais da **P00** (Programação Orientada a Objetos) e **TAD** (Tipos abstratos de dados)

Classes

Recapitulando

```
class Bank {  
  name  
  departments  
  description  
  clients  
}  
  
const bank = new Bank()
```

Construtores

Recapitulando

Construtores são “funções” dentro da classes que são rodadas toda vez que usamos a keyword new

Elas permitem padronizar a forma como objetos são instanciados a partir de classes, já que definem tudo que uma classe precisa para ser instanciada

Construtores

Recapitulando

```
class Bank {  
  name  
  description  
  departments  
  clients  
  
  constructor(name, description, departments, clients) {  
    this.name = name  
    this.description = description  
    this.departments = departments  
    this.clients = clients  
  }  
}
```

Construtores

Recapitulando



```
const name = 'Santander'
const description = 'A solução completa para o seu negócio'
const departments = ['Crédito', 'Contas a pagar', 'Atendimento ao cliente']
const clients = [
  { id: 1, name: 'Ada', account: '12345', digit: 'X' }
]

const bank = new Bank(name, description, departments, clients)
console.log('#Review3: ', bank)
```


Atributos

Recapitulando

Atributos são elementos de uma classe que podem assumir os tipos primitivos (string, number, array, etc)

Em javascript, os atributos podem ter encapsulamento de **public** ou **private**

Atributos

Recapitulando

Atributos **public** são acessíveis fora da classe (tanto para leitura como para escrita)

```
class Bank {  
    #name  
    description  
    #departments  
    #clients  
}
```

Atributos

Recapitulando

Atributos **public** são acessíveis fora da classe (tanto para leitura como para escrita)



```
console.log(bank.description)
```

Atributos

Recapitulando

Atributos **private** **não** são acessíveis fora da classe (nem para leitura nem para escrita)

Em javascript, são definidos com um **#** na frente

```
class Bank {  
  #name  
  description  
  #departments  
  #clients  
}
```


Atributos

Recapitulando

Atributos **private** **não** são acessíveis fora da classe (nem para leitura nem para escrita)



```
bank.#name           // erro  
bank.#departments    // erro  
bank.#clients         // erro
```

Atributos

Recapitulando

Atributos **private** **não** são acessíveis fora da classe (nem para leitura nem para escrita)

Para ler os atributos privados, usamos os **getters**


```
get name() {  
    return this.#name  
}
```

Atributos

Recapitulando

Atributos **private** **não** são acessíveis fora da classe (nem para leitura nem para escrita)

Para alterar os atributos privados, usamos os **setters**



```
set name(value) {  
    this.#name = value  
}
```



Métodos

Métodos

Métodos são funções definidas dentro das classes que permitem dar mais interatividade para elas



```
addDepartment(department) {  
    this.#departments.push(department)  
}
```



```
removeDepartment(index) {  
  this.#departments.splice(index, 1)  
}
```

Prática

Parte 1: Criando uma classe para Cliente

1. Criem uma nova classe para os clientes desse banco
2. Ela deve possuir os atributos `id`, `name`, `account` (conta) e `digit` (digito da conta). Vocês podem criar quantos atributos a mais vocês quiserem

Parte 2: Criando métodos na classe Bank

1. Criem um método que busca um determinado cliente a partir do seu id no banco
2. Criem um método que remove um cliente a partir do seu id no banco
3. Criem um método que adiciona um novo cliente no banco. Antes de criar, vocês devem validar se: (a) já existe um cliente com esse id; (b) se já existe um cliente com o nome em questão. Caso já exista, não adicionem o novo cliente

Materias de Aula

1. Repositório do Github com exemplos de código:
<https://github.com/joaogolias/FE-JS-003-PROGRAMACAO-ORIENTADA-A-OBJETOS-v1>

Obrig.ada



Aula 4

Herança e Polimorfismo

Sumário

- 1 Introdução
- 2 Herança
- 3 Polimorfismo
- 4 Prática
- 5 Materiais de aula

Introdução

O que é herança na vida real?

O que vocês acham que é Polimorfismo?

Herança

Herança

Herança (Inheritance) é uma característica dentro da POO que permite que classes herdem o construtor, os atributos e os métodos de outras

Fazemos isso com o auxílio da *keyword* **extends**



```
class BankAccount {  
    #number  
    #digit  
    #amount  
  
    get amount() {  
        return this.#amount  
    }  
  
    addAmount(amount) {  
        this.#amount += amount  
    }  
}
```

Herança

A classe **SavingsBankAccount** herda o construtor, os atributos e os métodos definidos na classe **BankAccount**

SavingsBankAccount é chamada de “classe filha” (*child class*)

BankAccount é chamada de “classe mãe” ou “classe pai” (*parent class* ou *super class*)



```
class SavingsBankAccount extends BankAccount {}
```

Herança



```
class SavingsBankAccount extends BankAccount {}  
  
const savingsBankAccount = new SavingsBankAccount('123', '1', 200)  
savingsBankAccount.addAmount(100)  
console.log('savingsBankAccount: ', savingsBankAccount.amount)
```

Herança

Atributos **privados** na classe mãe **não são acessíveis** na classe filha

```
class SavingsBankAccount extends BankAccount {  
  printAccount() {  
    console.log(this.#account) // Dá um erro  
    console.log(this.account) // Funciona, porque é um método herdado  
  }  
}
```


Herança

Podemos acessar o construtor e métodos da classe mãe usando a *keyword* **super**



```
class SavingsBankAccount extends BankAccount {  
    constructor(number, digit, amount) {  
        super(number, digit, amount)  
    }  
}
```

Herança

Podemos **sobrescrever (override)** métodos da classe mãe, alterando seus comportamentos

```
class SavingsBankAccount extends BankAccount {  
  addAmount(amount) {  
    console.log('Adding savings')  
    super.addAmount(amount)  
  }  
}
```

Polimorfismo

Polimorfismo

Polimorfismo (polymorphism) é uma característica dentro da POO que diz que:

Se uma classe é filha de outra, então ela também é do mesmo tipo da classe pai

Etimologia:

- Poli, do grego, “muitas”
- Morphos, do grego, “formas”

Ou seja: Polimorfismo é a característica que diz que uma classe pode “assumir muitas formas”, desde que herde “outras formas” de outras classes

Polimorfismo

A keyword **instanceof** permite verificar se um objeto é uma instância de uma classe



```
const bankAccount = new BankAccount('123', '1', 1000)
const savingsBankAccount = new SavingsBankAccount('123', '1', 200)

console.log(bankAccount instanceof BankAccount) // True
console.log(savingsBankAccount instanceof SavingsBankAccount) // True

console.log(savingsBankAccount instanceof BankAccount) // True
console.log(bankAccount instanceof SavingsBankAccount) // False
```

Prática

Parte 1: Alterações na classe `BankAccount`

1. Na classe `BankAccount`, adicione um método que representa o “saque” de um valor, que deve:
 - a. Descontar o valor que estamos tentando sacar
 - b. Retornar o novo valor da conta

Parte 2: Especificidades da SavingsBankAccount

1. Na classe `SavingsBankAccount`, adicione uma propriedade para representar o “valor mínimo da conta”
2. Na classe `SavingsBankAccount`, altere o método de “saque” para que ele não permita realizar o saque caso o novo valor armazenado da conta passe a ser menor do que o “valor mínimo da conta”

Exemplo: Vamos considerar uma conta que tem o valor mínimo de R\$100 e possui R\$150,00 como o valor armazenado.

- Se o cliente tentar fazer um saque de R\$50,00, o valor final seria R\$100. Então esse saque é permitido.
- Se o cliente tentar fazer um saque de R\$51,00, o valor final seria R\$99,00. Então esse saque não pode ser executado

Parte 3: Criação do ProfitableBankAccount

1. Criem uma nova classe chamada `ProfitableBankAccount` (conta rentável) que é filha da classe `BankAccount`
2. Toda vez que adicionarmos um novo valor nessa conta, ela deve:
 - a. Adicionar 1% em relação ao valor adicionado
 - b. Descontar uma comissão de R\$3,00Exemplo: ao adicionar R\$500 reais, na verdade, ela deve adicionar:

$$R\$500,00 + 1\% \times R\$500,00 (R\$5,00) - R\$3,00 = R\$502,00$$

3. Toda vez que realizamos um saque nesta conta, ela deve descontar R\$0,50 do valor armazenado. Altere o método de saque para isso.

Parte 4: Alteração na classe Client

1. Criem um método que permite adicionar uma conta ao `Client`
2. O banco que estamos simulando só permite que clientes tenham contas de poupança ou rentáveis. Então, alterem o `Client` para que:
 - No construtor, não permita adicionar contas do tipo `BankAccount`, apenas de suas filhas.
 - No método de adicionar uma conta, não permita adicionar contas do tipo `BankAccount`, apenas de suas filhas.

Materias de Aula

Materiais de Aula

1. Os códigos da aula estão no seguinte repositório do Github na pasta *class4*:
<https://github.com/joaogolias/FE-JS-003-PROGRAMACAO-ORIENTADA-A-OBJETOS-v1>

Obrig.ada