Professores
Celso Rodrigo Giusti
Daniel Manoel Filho
Marlon Palata Fanger
Rodrigues

POO

SENAI

PROGRAMAÇÃO BACK-END

Paradigmas

Paradigma é uma maneira ou abordagem para pensar e resolver problemas dentro de um determinado campo. Na programação o termo paradigma refere-se a um **estilo** ou modelo de **organização** do **código** e da **lógica** de um programa.

PROGRAMAÇÃO BACK-END

Paradigma Imperativo

Um dos paradigmas mais tradicionais e amplamente utilizados.

No paradigma imperativo, você escreve um conjunto de instruções que descrevem passo a passo como o computador deve realizar a tarefa.

Aqui, o foco está em como fazer as coisas, ou seja, você dá ao computador uma sequência de comandos para executar

```
let contador = 0;  // Estado inicial
contador += 1;     // Mudança de estado
console.log(contador);  // Saída: 1
```

PROGRAMAÇÃO BACK-END

Paradigma Declarativo

No paradigma declarativo você descreve o que quer fazer.

Você define o resultado desejado sem especificar explicitamente as etapas para alcançá-lo.

Exemplo de código declarativo (usando SQL):

Aqui não estamos dizendo ao banco de dados COMO buscar os alunos. Estamos apenas declarando O QUE queremos.

```sql
SELECT nome FROM alunos WHERE idade > 18;
```

PROGRAMAÇÃO BACK-END

Paradigma Funcional

No paradigma funcional, o foco está em **funções** puras e **transformações** de dados.

Em vez de modificar o estado do programa ou usar variáveis mutáveis, o paradigma funcional enfatiza a criação de funções que retornam novos valores a partir de entradas, sem modificar o estado global.

```
const soma = (a, b) => a + b;

const dobro = (x) => x * 2;


const resultado = dobro(soma(2, 3));  // Saída: 10
```

PROGRAMAÇÃO BACK-END

Paradigma Orientado a Objetos

- O paradigma orientado a objetos (OO) é baseado no conceito de objetos, que são instâncias de classes.
- Em OO, você modela o mundo em termos de objetos que interagem entre si. Cada objeto possui propriedades e métodos, e você organiza o código em torno desses objetos.
- Este paradigma possui diversas características, que veremos daqui a pouco.

PROGRAMAÇÃO BACK-END

O que é um Objeto?

Um objeto é uma **coleção de propriedades e métodos** que representam um conceito ou entidade do **mundo real** dentro de um programa.

- **Atributos:** são dados que descrevem o objeto;
- **Métodos:** são funções associadas ao objeto que realizam ações ou comportamentos relacionados ao objeto.

Objetos são usados para representar coisas reais, como carros, pessoas, animais, etc.

PROGRAMAÇÃO BACK-END

Objeto Literal

Um objeto literal é definido por um par de chaves, onde definimos diretamente um objeto sem utilizar um modelo (classe).

```
const pessoa = {
  nome: "João",
  idade: 30,
  falar: function() {
    console.log(`Olá, meu nome é ${this.nome}`);
  }
};

pessoa.falar();
```

PROGRAMAÇÃO BACK-END

Objeto Instanciado (Classe)

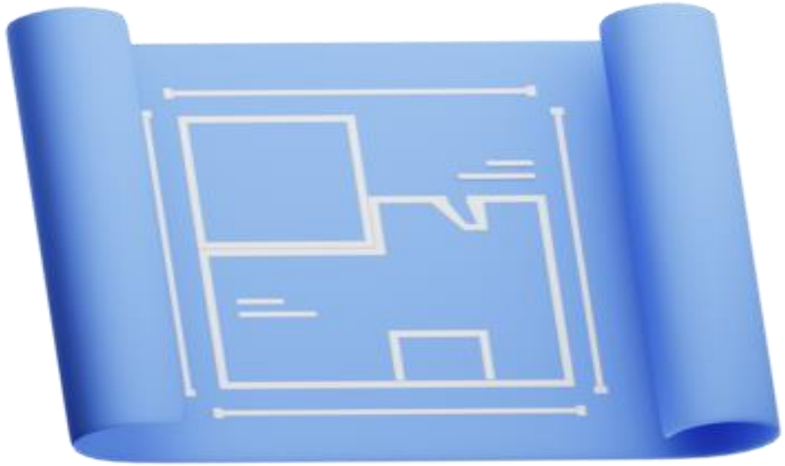
Um objeto instanciado é definido a partir de um modelo (classe), nele é possível herdar propriedades e métodos, reutilizar código, e manipular o objeto.

PROGRAMAÇÃO BACK-END

Objeto Instanciado (Classe)

A classe é o modelo, onde vamos criar nosso objeto. É nela que iremos definir as propriedades e métodos que os objetos criados a partir dela terão.

Mas não confunda, a classe em si não é um objeto real, ela é como um projeto para criar os objetos.



PROGRAMAÇÃO BACK-END

O que é uma Instância?

Uma instância é um objeto criado a partir de uma classe. Quando você cria um objeto a partir de uma classe, esse objeto é uma instância dessa classe.

Então a ordem é:

- Cria o modelo do Objeto (CLASSE);
- Cria o Objeto (INSTÂNCIA A CLASSE);

PROGRAMAÇÃO BACK-END

Criando uma Classe em JS

Para criarmos uma classe em JavaScript devemos seguir a seguinte estrutura:

```
class Pessoa {}
```

Atenção: O nome da classe sempre deve ser em PascalCase, ou seja, deve começar com letra maiúscula sempre que tiver uma nova palavra.

Exemplo: “ class ClasseExemplo { } ”

PROGRAMAÇÃO BACK-END

Construtor

O construtor é um método especial da Classe que permite a inicialização do Objeto e também a declaração dos atributos.

Quando a classe é instanciada, o construtor é chamado automaticamente, para que os atributos sejam passados e a classe seja construída a partir dos atributos.

Para definirmos quais atributos serão passados utilizaremos a palavra-chave “this” que faz referência ao objeto e nos permite acessar suas propriedades.

PROGRAMAÇÃO BACK-END

Definindo um Construtor

Sendo um método, é necessário passar os parâmetros, que são os atributos do nosso objeto, e depois usar o THIS para fazer referência aos nossos atributos:

```
class Pessoa {
  constructor(nome, idade) {
    this.nome = nome;
    this.idade = idade;
  }
```

PROGRAMAÇÃO BACK-END

Métodos

São basicamente funções que pertencem aos Objetos que geralmente determinam uma ação.

Dentro dos métodos também é possível acessar os atributos através da palavra-chave “this”

```
class Pessoa {
  constructor(nome, idade) {
    this.nome = nome;
    this.idade = idade;
  }

  saudacao() {
    return `Olá, meu nome é ${this.nome} e tenho ${this.idade} anos.`;
  }
}
```

PROGRAMAÇÃO BACK-END

Métodos

```
class Pessoa {
  constructor(nome, idade) {
    this.nome = nome;
    this.idade = idade;
  }

  saudacao() {
    return `Olá, meu nome é ${this.nome} e tenho ${this.idade} anos.`;
  }
}
```

PROGRAMAÇÃO BACK-END

Instanciando uma Classe

Vamos ao exemplo de como instanciar uma classe e criar o objeto em questão:

```
// Criando um objeto a partir da classe Pessoa
const pessoa1 = new Pessoa("João", 25);

// Chamando o método do objeto
pessoa1.saudacao();
```

PROGRAMAÇÃO BACK-END

Os 4 pilares da Orientação a Objetos

PROGRAMAÇÃO BACK-END

Abstração

A abstração é a ideia de ocultar os detalhes complexos e mostrar apenas o que é relevante para o usuário.

Exemplo: Imagine que você tem um objeto “Carro”. Você não precisa se preocupar com como o motor funciona internamente, só precisa saber que pode ligá-lo ou desligá-lo.

```
class Carro {

  ligaMotor(){
    console.log('Motor Ligado')
  }

  desligarMotor(){
    console.log('Motor Desligado')
  }
```

PROGRAMAÇÃO BACK-END

Encapsulamento

Encapsulamento significa **proteger** os dados internos de um objeto, permitindo que eles sejam acessados e **modificados** apenas por **métodos da própria classe** e métodos específicos, conhecidos como getters e setters.

Encapsulamento – Atributo Privado

Para privar um atributo no JavaScript basta colocar um “#” antes do nome do atributo. Dessa forma ele será um atributo privado.

```
class ContaBancaria {
    #saldo
    numeroConta

    constructor(saldo, numeroConta){
        this.#saldo = saldo
        this.numeroConta = numeroConta
    }
```

Encapsulamento – Acessando Atributo Privado

No mesmo arquivo, abaixo da nossa classe, vamos instanciar a nossa classe criando um novo objeto, e vamos tentar acessar o atributo privado.

Observem que ao tentar acessar o atributo privado, um erro é exibido. Para acessar esse atributo privado iremos definir um getter para ele.

```
let conta = new ContaBancaria(100,'R2-D2')

console.log(conta.numeroConta)
console.log(conta.#saldo)    Property '#saldo' is not accessible outside class
```

PROGRAMAÇÃO BACK-END

Getter

O Getter é um método especial, usado para acessar (ler) o valor de um atributo de forma controlada.

Ele é geralmente usado para retornar um valor.

Com o getter também é possível realizar cálculos ou transformar os dados antes de retorná-los

Definindo Getter

Na nossa Classe “ContaBancaria”, vamos definir o getter para acessar nosso atributo privado.

```
class ContaBancaria {
    #saldo
    numeroConta

    constructor({saldo, numeroConta}){
        this.#saldo
        this.numeroConta
    }

    get getSaldo() {return this.#saldo}
```

Acessando atributo via Getter

Para acessar o nosso atributo vamos utilizar a nossa classe que já está instanciada, e acessar o getter como se fosse um atributo:

```
let conta = new ContaBancaria(1250, "R2D2")

console.log(conta.numeroConta)
console.log(conta.getSaldo)
```

Getter não altera valores

Se tentarmos utilizar o getter para alterar o valor do atributo privado, diferente do Dart, o Javascript não retornará um erro de sintaxe, porém não irá alterar o valor armazenado no getSaldo.

Para alterar o valor do atributo, teremos que utilizar um Setter

```
console.log(conta.getSaldo)
conta.getSaldo = 150
console.log(`saldo alterado pelo get: ${conta.getSaldo}`)
```

Retorno:

```
saldo get: 1250
saldo alterado pelo get: 1250
```

PROGRAMAÇÃO BACK-END

Setter

O Setter é um método especial, usado para **alterar ou validar** o valor de um atributo de forma controlada.

Ele permite validar ou transformar os dados antes de armazená-los, garantindo que apenas valores corretos sejam atribuídos ao atributo.

PROGRAMAÇÃO BACK-END

Definindo Setter

Primeiro vamos criar um Setter simples dentro da nossa classe ContaBancaria, que nos permita alterar o valor do atributo, sem validações.

```
set setSaldo(value) {
    return this.#saldo = value
}
```

Alterando valor do saldo

Agora onde instanciamos a nossa classe, iremos chamar o método set e ver a alteração do valor.

```
let conta = new ContaBancaria(1250, "R2D2")

console.log(`saldo get: ${conta.getSaldo}`)

conta.setSaldo = 150

console.log(`saldo alterado pelo set: ${conta.getSaldo}`)
```

Retorno:

```
saldo get: 1250
saldo alterado pelo set: 150
```

PROGRAMAÇÃO BACK-END

Adicionando validações

Vamos implementar validações extras no setter da nossa classe, para validar se o valor inserido pelo setter é válido.

```
set setSaldo(value) {

    if(value != null && value > 0){
        this.#saldo = value
    } else {
        console.log('⚠ Saldo
        Inválido')
    }
}
```

Alterando valor do saldo

Agora onde instanciamos a nossa classe, iremos chamar o método set e ver a alteração do valor. Observe que estamos passando um valor negativo para o saldo, para forçar o erro.

```
let conta = new ContaBancaria(1250, "R2D2")

console.log(`saldo get: ${conta.getSaldo}`)

conta.setSaldo = -6

console.log(`saldo alterado pelo set: ${conta.getSaldo}`)
```

Retorno:

```
saldo get: 1250
⚠Saldo Inválido
saldo alterado pelo set: 1250
```

Herança

No exemplo abaixo temos uma classe Animal, e uma classe Cachorro.

A classe animal contém um método “fazerSom”, a classe Cachorro precisa utilizar esse método e para reaproveitar o método que já existe basta a classe Cachorro EXTENDER a classe Animal, tendo assim, acesso ao método “fazerSom”.

```
class Animal {
    fazerSom(){
        console.log('Emite um som genérico')
    }
}

class Cachorro extends Animal {}
```

Herança

Observe que mesmo a classe Cachorro estando vazia, ela consegue acessar os métodos e atributos da classe pai (Classe Animal)

```
let cachorro = new Cachorro();
cachorro.fazerSom();
```

PROGRAMAÇÃO BACK-END

Herança

Além de conseguir reutilizar os métodos já prontos da classe pai, também é possível alterar esses métodos de acordo com o que a classe filha precisa.

Para alterar basta criarmos uma nova função com o mesmo nome da classe pai.

```
class Animal {
    fazerSom(){
        console.log('Emite um som genérico')
    }
}

class Cachorro extends Animal {
    fazerSom(){
        console.log('Auau')
    }
}
```

PROGRAMAÇÃO BACK-END

Super (Classe pai)

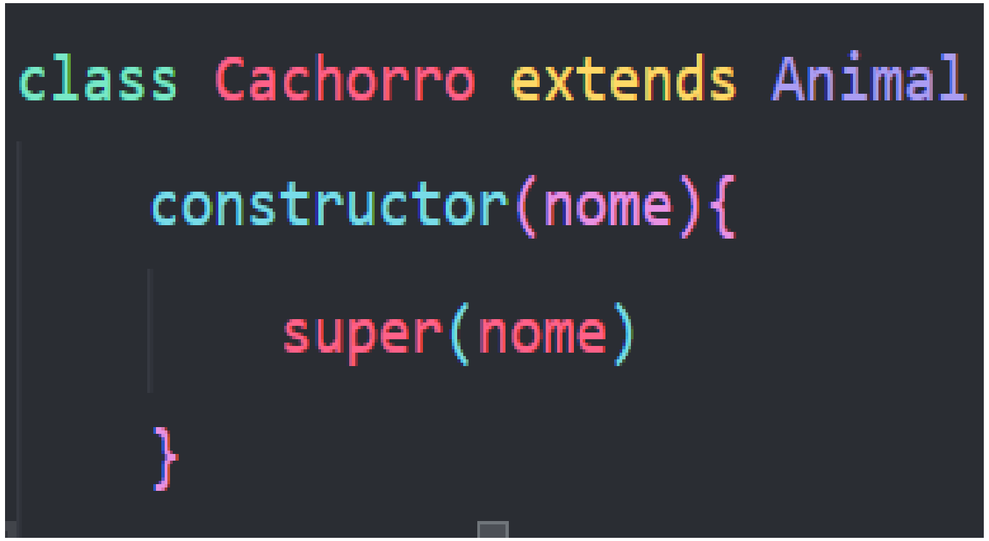
O super é uma palavra reservada que é usada dentro de classes filhas para chamar métodos ou construtores da classe pai.
Por exemplo, na nossa classe pai “Animal”, vamos definir um construtor e passar um atributo “nome”.

```
class Animal {

    nome

    constructor(nome){

        this.nome = nome

    }
```

PROGRAMAÇÃO BACK-END

Super (Classe filha)

Agora na classe filha “Cachorro” que estende a classe “Animal” vamos criar um construtor para utilizar o atributo “nome” da classe pai.



PROGRAMAÇÃO BACK-END

Super (Utilização)

Agora vamos instanciar a nossa classe e utilizar o atributo nome.

Repare que não criamos um atributo nome na classe Cachorro, mas sim na classe Animal, isso só é possível pois estamos utilizando o construtor da Classe Pai através do super()

```
let cachorro = new Cachorro('Rex')

console.log(`Nome do Cachorro: ${cachorro.nome}`)
```

PROGRAMAÇÃO BACK-END

Polimorfismo

Polimorfismo é um conceito fundamental na programação orientada a objetos, que significa "muitas formas".

Em termos simples, o polimorfismo permite que um objeto seja tratado como um tipo mais genérico, mas ainda assim execute comportamentos específicos de sua classe concreta.

Ele permite que diferentes classes implementem um mesmo método de maneiras distintas.

PROGRAMAÇÃO BACK-END

Override

Para aplicar o polimorfismo, podemos utilizar a sobrescrita de métodos:

- Sobrescrita (ou overriding): Quando uma classe filha reimplementa um método de sua classe pai, ou seja, ela “substitui” o comportamento do método herdado.

Para realizarmos uma sobrescrita, basta chamar o método da classe pai e modificá-lo.

Exemplo de Sobrescrita

Neste exemplo temos a classe PAI animal, que possui um método de fazerSom. E temos uma classe filha cachorro que HERDA da classe Animal, tendo acesso ao método fazerSom.

Dessa forma “criamos” novamente um método com o mesmo nome, para que o polimorfismo seja aplicado.

```
class Cachorro extends Animal {

    fazerSom(){

        console.log('Auau')

    }

}
```

PROGRAMAÇÃO BACK-END

Overload

Polimorfismo de **sobrecarga** acontece quando um mesmo método tem **várias versões**, diferenciadas pelo número ou tipo de parâmetros.

O Polimorfismo de Sobrecarga é um tema que gera muita dúvida em **JavaScript**, porque ele **não funciona** da mesma forma que em linguagens fortemente tipadas como **Java** ou **C#**.

PROGRAMAÇÃO BACK-END

Overload

Exemplo em **JAVA**

```java
class Calculadora {
    int soma(int a, int b) {
        return a + b;
    }

    double soma(double a, double b) {
        return a + b;
    }

    int soma(int a, int b, int c) {
        return a + b + c;
    }
}
```

PROGRAMAÇÃO BACK-END

Overload

Exemplo em **JavaScript**

```
class Calculadora {
    soma(a, b) {
        return a + b
    }

    soma(a, b, c) {
        return a + b + c
    }
}

let calc = new Calculadora()
console.log(calc.soma(2, 3))
```

```
class Calculadora {
    soma(a, b, c) {
        if (c !== undefined) {
            return a + b + c
        }
        return a + b
    }
}

let calc = new Calculadora()
console.log(calc.soma(2, 3))
console.log(calc.soma(2, 3, 4))
```

Escola SENAI "Italo Bologna"

Av. Goiás, 139 – Itu/SP

Telefone
(11) 2396-1999

Instagram
@senai.itu

Facebook
/senai.itu

Site
https://sp.senai.br/unidade/itu/