Aula Prática: Orientação a Objetos com Java, Python e TypeScript Construindo um Banco Digital com Docker

Agenda

- **O que é Programação Orientada a Objetos (POO)?**
  Por que usar este paradigma?
- **Os 4 Pilares Fundamentais da POO**
  Encapsulamento, Abstração, Herança e Polimorfismo.
- **Projeto Prático: O "Banco Digital"**
  Apresentação do nosso case de estudo.
- **Hands-On: Comparando as Linguagens**
  Analisando a implementação em Java, Python e TypeScript.
  Executando tudo com Docker.

O que é Programação Orientada a Objetos?

É um paradigma de programação baseado no conceito de "objetos".

Pense em objetos do mundo real: um **carro**, uma **pessoa**, uma **conta bancária**. Eles têm:

- **Atributos** (**características**): cor, modelo, saldo, nome.
- **Métodos** (**ações**): acelerar, andar, sacar, depositar.

A POO nos ajuda a trazer essa lógica do mundo real para o código, tornando-o mais organizado e intuitivo.

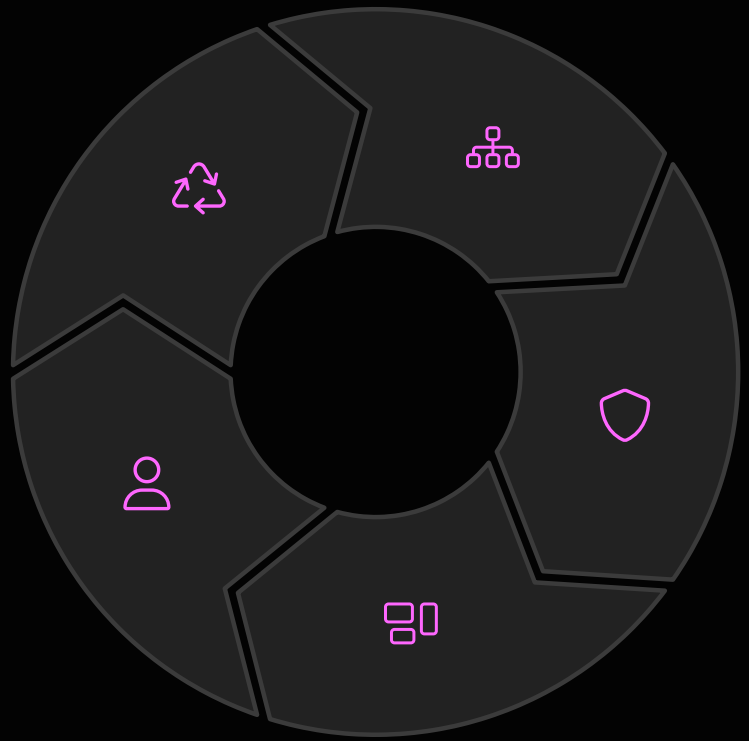
Por que usar POO?

Reutilização de Código

A Herança permite que uma classe aproveite atributos e métodos de outra.

Trabalho em Equipe

Classes bem definidas são como "peças de Lego" que diferentes desenvolvedores podem criar e juntar.



Organização

Agrupa dados e comportamentos relacionados, facilitando a manutenção.

Segurança

O Encapsulamento protege os dados de manipulações indevidas.

Flexibilidade

O Polimorfismo permite que nosso sistema se adapte a novos tipos de objetos facilmente.

Os 4 Pilares da POO

Estes são os conceitos que sustentam todo o paradigma. Vamos ver cada um deles.

Encapsulamento

A ideia:

Agrupar dados (atributos) e os métodos que os manipulam dentro de um único objeto, protegendo os dados de acesso externo direto.

Analogia:

Uma cápsula de remédio. Você não acessa os componentes químicos diretamente. Você simplesmente "usa" a cápsula (toma o remédio), e ela faz o trabalho internamente.

No nosso projeto:

O saldo da conta é protegido. Não podemos simplesmente fazer conta.saldo = -1000. Temos que usar os métodos depositar() e sacar().

Abstração

A ideia:

Ocultar a complexidade interna e expor apenas as funcionalidades essenciais de um objeto.

Analogia:

O painel de um carro. Você não precisa saber como o motor de combustão funciona para dirigir. Você só precisa do volante, pedais e da marcha. A complexidade está abstraída.

No nosso projeto:

A classe Conta é abstrata. Ela define o que toda conta deve ter (um titular, um saldo, um método sacar), mas não diz como cada saque funciona. Isso é responsabilidade das classes mais específicas.

Herança

A ideia:

Permitir que uma classe (filha) herde atributos e métodos de outra classe (pai). Isso promove a reutilização de código.

Analogia:

Uma árvore genealógica. Você herda características dos seus pais, mas também tem as suas próprias características únicas.

No nosso projeto:

ContaCorrente e ContaPoupanca herdam de Conta. Ambas têm titular e saldo, mas cada uma tem suas próprias regras (cheque especial, rendimento).

Polimorfismo

A ideia:

"Muitas formas". É a capacidade de um objeto ser referenciado de múltiplas maneiras. Permite que chamemos o mesmo método em objetos diferentes e cada um responda de sua forma específica.

Analogia:

Uma porta USB. Você pode conectar um mouse, um teclado ou um pendrive. A "porta" (o método) é a mesma, mas o comportamento de cada dispositivo conectado é diferente.

No nosso projeto:

Podemos ter uma lista de Conta e chamar sacar(100) em todos os itens. A ContaCorrente usará o limite se precisar, a ContaPoupanca não. O mesmo comando, comportamentos diferentes.

Projeto Prático: 🏦 Banco Digital

Vamos aplicar esses conceitos em um projeto real!

- Cenário: Um sistema bancário simples.
- Linguagens: Veremos implementações idênticas em Java, Python e TypeScript.
- Ambiente: Tudo roda dentro de Docker, então não é preciso instalar nada além dele!

Estrutura do Projeto e Execução

Usamos uma estrutura de monorepo, gerenciada pelo docker-compose.yml.

```
banco-digital-multi-linguagem/
├── java-banco-digital/
├── python-banco-digital/
├── typescript-banco-digital/
└── docker-compose.yml <-- O Maestro!
```

Para rodar TUDO de uma vez:

```
docker-compose up --build
```

Para rodar apenas UMA linguagem (ex: Python):

```
docker-compose up --build python
```

Código em Ação: Abstração e Herança

A classe Conta é o nosso contrato abstrato. ContaCorrente herda dela.

Java

```
public abstract class Conta { ... }
public class ContaCorrente extends Conta { ... }
```

Python

```
from abc import ABC
class Conta(ABC): ...
class ContaCorrente(Conta): ...
```

TypeScript

```
export abstract class Conta { ... }
export class ContaCorrente extends Conta { ... }
```

Encapsulamento

O saldo é protegido. Em Java, usamos private/protected. Em Python/TS, a convenção _ indica que o atributo não deve ser acessado diretamente.

Java

```
protected double saldo;
public double getSaldo() { return this.saldo;}
```

Python

```
def __init__(self):
    self._saldo = 0

@property
def saldo(self):
    return self._saldo
```

TypeScript

```
protected _saldo: number;

public getSaldo(): number { return this._saldo; }
```

Encapsulamento

Criamos uma lista de Conta e chamamos o mesmo método sacar() para todos. Cada objeto executa sua própria versão do método.

Java

```
List contas = ...;
for (Conta conta : contas) {
    conta.sacar(100);
}
```

Python

```
contas = [...]
for conta in contas:
    conta.sacar(100)
```

TypeScript

```
const contas: Conta[] = ...;
contas.forEach(conta => {
 conta.sacar(100);
});
```

Conclusão e Principais Lições

POO é Universal

Os conceitos são os mesmos, não importa a linguagem. Apenas a sintaxe muda.

Organização é Chave

POO nos força a pensar em uma estrutura lógica antes de sair codificando.

Docker Facilita a Vida

Ambientes de desenvolvimento consistentes são essenciais para o aprendizado e para o trabalho em equipe.

A Prática Leva à Perfeição

Explore o código, modifique, crie novos tipos de conta (ex: ContaSalario) e veja como o sistema se adapta!