Desenvolvimento Full Stack

CadastroPoo

Matheus José Ribeiro de Moura

Iniciando o caminho pelo Java
Campus Rua Teresa
2024.3 FLEX
Turma 9001

Repositório GIT: https://github.com/MatheusJRM/CadastroPOO.git

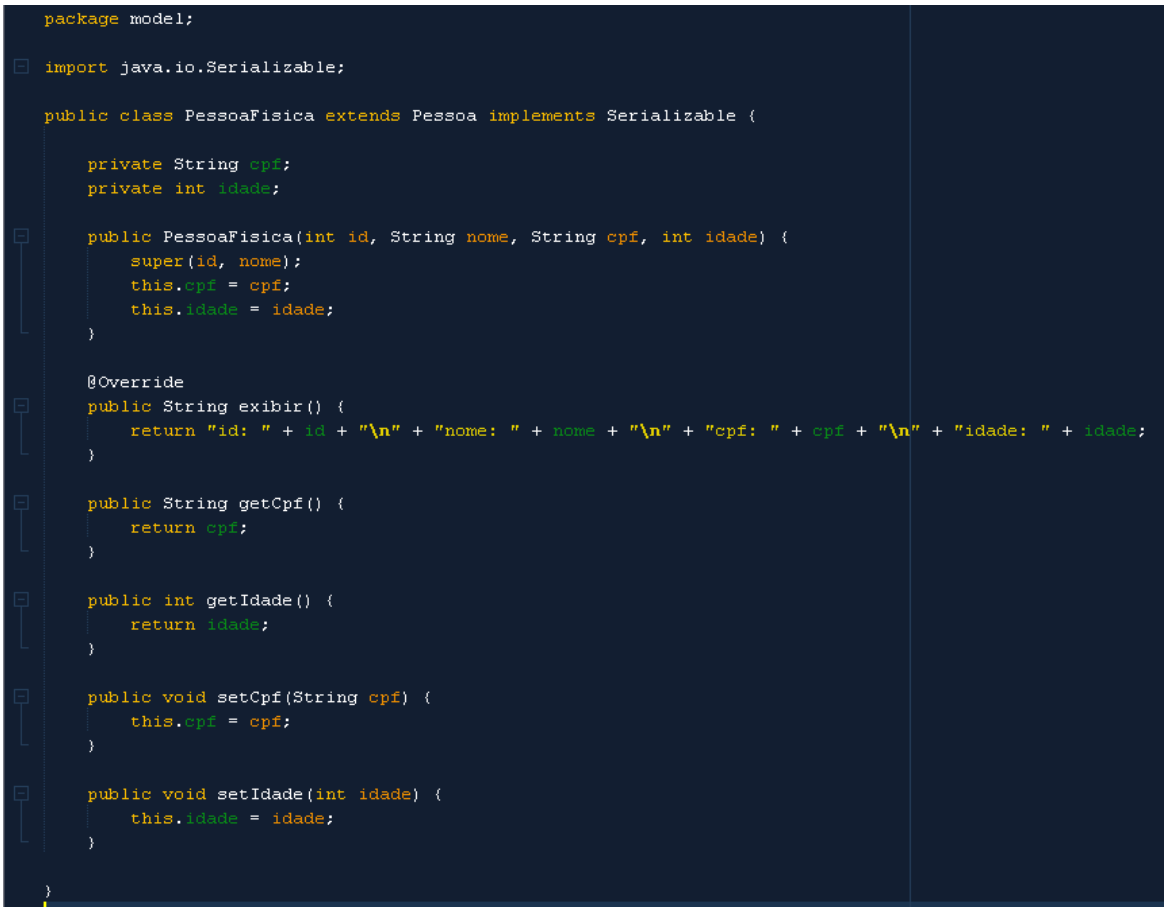
- Objetivo da Prática:

  1. Utilizar herança e polimorfismo na definição de entidades.
  2. Utilizar persistência de objetos em arquivos binários.
  3. Implementar uma interface cadastral em modo texto.
  4. Utilizar o controle de exceções da plataforma Java.

- Códigos Fonte:
  - Model Pessoa

```java
package model;

import java.io.Serializable;

public class Pessoa implements Serializable {

    protected int id;
    protected String nome;

    public Pessoa(int id, String nome) {
        super();
        this.id = id;
        this.nome = nome;
    }

    protected String exibir() {
        return id + " " + nome;
    }

    public int getId() {
        return id;
    }

    public void setId(int id) {
        this.id = id;
    }

    public void setNome(String nome) {
        this.nome = nome;
    }

    public String getNome() {
        return nome;
    }

}
```

- Model PessoaFisica



```java
package model;

import java.io.Serializable;

public class PessoaFisica extends Pessoa implements Serializable {

    private String cpf;
    private int idade;

    public PessoaFisica(int id, String nome, String cpf, int idade) {
        super(id, nome);
        this.cpf = cpf;
        this.idade = idade;
    }

    @Override
    public String exibir() {
        return "id: " + id + "\n" + "nome: " + nome + "\n" + "cpf: " + cpf + "\n" + "idade: " + idade;
    }

    public String getCpf() {
        return cpf;
    }

    public int getIdade() {
        return idade;
    }

    public void setCpf(String cpf) {
        this.cpf = cpf;
    }

    public void setIdade(int idade) {
        this.idade = idade;
    }

}
```

- Model PessoaJuridica

```java
package model;

import java.io.Serializable;

public class PessoaJuridica extends Pessoa implements Serializable {

    private String cnpj;

    public PessoaJuridica(int id, String nome, String cnpj) {
        super(id, nome);
        this.cnpj = cnpj;
    }

    public String exibir() {
        return "id: " + id + "\n" + "nome: " + nome + "\n" + "cnpj: " + cnpj;
    }

    public String getCnpj() {
        return cnpj;
    }

    public void setCnpj(String cnpj) {
        this.cnpj = cnpj;
    }

}
```

- Model PessoaFisicaRepo

```java
package model;

import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;
import java.util.ArrayList;
import java.util.Optional;

public class PessoaFisicaRepo {

    private ArrayList<PessoaFisica> pessoasFisicas = new ArrayList<>();

    public PessoaFisicaRepo() {
        this.pessoasFisicas = new ArrayList<>();
    }

    public void inserir(PessoaFisica pessoaFisica) {
        pessoasFisicas.add(pessoaFisica);
    }

    public void alterar(PessoaFisica pessoaAlterada) {
        for (int i = 0; i < pessoasFisicas.size(); i++) {
            PessoaFisica pessoaExistente = pessoasFisicas.get(i);
            if (pessoaExistente.getId() == pessoaAlterada.getId()) {
                pessoaExistente.setNome(pessoaAlterada.getNome());
                pessoaExistente.setCpf(pessoaAlterada.getCpf());
                pessoaExistente.setIdade(pessoaAlterada.getIdade());
            }
        }

    }

    public boolean excluir(int id) {
        return pessoasFisicas.removeIf(pessoa -> {
            return id == pessoa.getId();
        });
    }

    public Optional<PessoaFisica> obter(int id) {
        return pessoasFisicas.stream().filter(pessoa -> id == pessoa.getId()).findFirst();
    }

    public ArrayList<PessoaFisica> obterTodos() {
        return new ArrayList<>(pessoasFisicas);
    }

    public void persistir(String nomeArquivo) throws IOException {
        try (ObjectOutputStream oos = new ObjectOutputStream(new FileOutputStream(nomeArquivo))) {
            oos.writeObject(pessoasFisicas);
        }
    }

    public void recuperar(String nomeArquivo) throws IOException, ClassNotFoundException {
        try (ObjectInputStream ois = new ObjectInputStream(new FileInputStream(nomeArquivo))) {
            pessoasFisicas = (ArrayList<PessoaFisica>) ois.readObject();
        }
    }
}
```

- Model PessoaJuridicaRepo

```java
package model;

import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;
import java.util.ArrayList;
import java.util.Optional;

public class PessoaJuridicaRepo {

    private ArrayList<PessoaJuridica> pessoasJuridicas = new ArrayList<>();

    public PessoaJuridicaRepo() {
        this.pessoasJuridicas = new ArrayList<>();
    }

    public void inserir(PessoaJuridica pessoaJuridica) {
        pessoasJuridicas.add(pessoaJuridica);
    }

    public void alterar(PessoaJuridica pessoaAlterada) {
        for (int i = 0; i < pessoasJuridicas.size(); i++) {
            PessoaJuridica pessoaExistente = pessoasJuridicas.get(i);
            if (pessoaExistente.getId() == pessoaAlterada.getId()) {
                pessoaExistente.setNome(pessoaAlterada.getNome());
                pessoaExistente.setCnpj(pessoaAlterada.getCnpj());
            }
        }

    }

    public boolean excluir(int id) {
        return pessoasJuridicas.removeIf(pessoa -> id == pessoa.getId());
    }

    public Optional<PessoaJuridica> obter(int id) {
        return pessoasJuridicas.stream().filter(pessoa -> id == pessoa.getId()).findFirst();
    }

    public ArrayList<PessoaJuridica> obterTodos() {
        return new ArrayList<>(pessoasJuridicas);
    }

    public void persistir(String nomeArquivo) throws IOException {
        try (ObjectOutputStream oos = new ObjectOutputStream(new FileOutputStream(nomeArquivo))) {
            oos.writeObject(pessoasJuridicas);
        }
    }

    public void recuperar(String nomeArquivo) throws IOException, ClassNotFoundException {
        try (ObjectInputStream ois = new ObjectInputStream(new FileInputStream(nomeArquivo))) {
            pessoasJuridicas = (ArrayList<PessoaJuridica>) ois.readObject();
        }
    }
}
```

- Classe Main (1° Procedimento):

```java
package main;

import java.io.IOException;
import model.PessoaFisica;
import model.PessoaFisicaRepo;
import model.PessoaJuridica;
import model.PessoaJuridicaRepo;

public class Main {

    public static void main(String[] args) {
        PessoaFisicaRepo repo1 = new PessoaFisicaRepo();

        PessoaFisica pessoa1 = new PessoaFisica(1, "Matheus Jose", "12345678900", 30);
        PessoaFisica pessoa2 = new PessoaFisica(2, "Isabela Arruda", "98765432100", 25);
        repo1.inserir(pessoa1);
        repo1.inserir(pessoa2);

        try {
            System.out.println("Dados de Pessoa Fisica Armazenados.");
            repo1.persistir("pessoas_fisicas.dat");
        } catch (IOException e) {
            System.out.println("Erro ao salvar os dados: " + e.getMessage());
        }

        PessoaFisicaRepo repo2 = new PessoaFisicaRepo();

        try {
            System.out.println("Dados de Pessoa Fisica Recuperados.");
            repo2.recuperar("pessoas_fisicas.dat");
            repo2.obterTodos().forEach(pessoa -> System.out.println(pessoa.exibir()));
        } catch (IOException | ClassNotFoundException e) {
            System.out.println("Erro inesperado ao recuperar os dados: " + e.getMessage());
        }

        PessoaJuridicaRepo repo3 = new PessoaJuridicaRepo();

        PessoaJuridica empresa1 = new PessoaJuridica(3, "Empresa ABC", "11111111000111");
        PessoaJuridica empresa2 = new PessoaJuridica(4, "Empresa XYZ", "22222222000122");
        repo3.inserir(empresa1);
        repo3.inserir(empresa2);

        try {
            System.out.println("\nDados de Pessoa Juridica Armazenados.");
            repo3.persistir("pessoas_juridicas.dat");
        } catch (IOException e) {
            System.out.println("Erro ao salvar os dados: " + e.getMessage());
        }

        PessoaJuridicaRepo repo4 = new PessoaJuridicaRepo();

        try {
            System.out.println("Dados de Pessoa Juridica Recuperados.");
            repo4.recuperar("pessoas_juridicas.dat");
            repo4.obterTodos().forEach(pessoa -> System.out.println(pessoa.exibir()));
        } catch (IOException | ClassNotFoundException e) {
            System.out.println("Erro inesperado ao recuperar os dados: " + e.getMessage());
        }
    }
```

- Código Resultante 1° Procedimento:

```
run:
Dados de Pessoa Fisica Armazenados.
Dados de Pessoa Fisica Recuperados.
id: 1
nome: Matheus Jose
cpf: 12345678900
idade: 30
id: 2
nome: Isabela Arruda
cpf: 98765432100
idade: 25

Dados de Pessoa Juridica Armazenados.
Dados de Pessoa Juridica Recuperados.
id: 3
nome: Empresa ABC
cnpj: 11111111000111
id: 4
nome: Empresa XYZ
cnpj: 22222222000122
BUILD SUCCESSFUL (total time: 0 seconds)
```

- Classe Main 2° Procedimento:
  Link (Transformado em link pelo tamanho da imagem)

- Código Resultante 2° Procedimento:

```
run:

Opcoes do Programa:
1 - Incluir
2 - Alterar
3 - Excluir
4 - Exibir pelo ID
5 - Exibir todos
6 - Salvar dados
7 - Recuperar dados
0 - Sair

Escolha uma opcao: 1

F - Pessoa Fisica | J - Pessoa Juridica: F
ID: 1
Nome: Matheus Jose
CPF(APENAS DIGITOS): 12752985754
Idade: 24
Pessoa Fisica inserida com sucesso!

Opcoes do Programa:
1 - Incluir
2 - Alterar
3 - Excluir
4 - Exibir pelo ID
5 - Exibir todos
6 - Salvar dados
7 - Recuperar dados
0 - Sair

Escolha uma opcao:
```

- Análise e Conclusão:
  - Vantagens e Desvantagens do Uso de Herança:

    1. **Vantagens**:
       1.1. Reutilização de Código: Herança permite que classes compartilhem código comum, reduzindo a duplicação e facilitando a manutenção.
       1.2. Hierarquia de Classes: Cria uma estrutura hierárquica clara, organizando classes de maneira lógica, o que melhora a legibilidade do código.
       1.3. Extensibilidade: Novas funcionalidades podem ser facilmente adicionadas a uma classe base e propagadas para as subclasses.
       1.4. Polimorfismo: Permite que objetos de diferentes subclasses sejam tratados como objetos de uma classe base comum, facilitando a generalização de código.
    2. **Desvantagens**:
       2.1. Aumento da Complexidade: Em sistemas grandes, uma hierarquia de herança complexa pode se tornar difícil de gerenciar e entender.
       2.2. Acoplamento Rígido: A herança cria um forte acoplamento entre a classe base e as subclasses, dificultando mudanças na classe base sem impactar as subclasses.
       2.3. Problemas de Manutenção: Modificações na classe base podem ter efeitos colaterais imprevisíveis em subclasses, dificultando a manutenção.
       2.4. Fragilidade do Código: Mudanças na classe base podem quebrar a funcionalidade nas subclasses, especialmente quando a herança é usada de forma imprópria.

  - Por que a interface Serializable é necessária ao efetuar persistência em arquivos binários?

- A interface Serializable é necessária ao persistir objetos em arquivos binários porque ela permite que o objeto seja convertido em uma sequência de bytes. Essa conversão, chamada de serialização, é essencial para armazenar o estado de um objeto em um arquivo ou transmiti-lo através de uma rede. Serializable permite que o Java verifique em tempo de execução se uma classe pode ser serializada, evitando erros de execução.

- Como o paradigma funcional é utilizado pela API stream no Java?
  - A API Stream do Java não altera a coleção original, em vez disso, elas retornam um novo stream ou resultado. Funções passadas para métodos como map, filter, e reduce devem ser puras, ou seja, sem efeitos colaterais. Isso facilita a composição de operações. Operações intermediárias em um stream, como map ou filter, são avaliadas de forma preguiçosa, o que significa que elas só são executadas quando uma operação terminal (como collect ou reduce) é invocada. Streams permitem a composição de várias operações funcionais, criando pipelines de transformação de dados que são fáceis de entender e manter.

- Quando trabalhamos com Java, qual padrão de desenvolvimento é adotado na persistência de dados em arquivos?

  - O padrão de desenvolvimento comumente adotado para persistência de dados em arquivos no Java é o DAO (Data Access Object). O padrão DAO abstrai a lógica de acesso a dados do restante do sistema, permitindo uma separação clara entre a lógica de negócios e a lógica de persistência. DAOs podem ser

reutilizados por diferentes partes do sistema, centralizando a lógica de acesso a dados. Alterações na forma como os dados são armazenados (por exemplo, mudar de arquivos binários para banco de dados) podem ser feitas alterando apenas a implementação do DAO, sem impactar o restante do sistema.

- O que são elementos estáticos e qual o motivo para o método main adotar esse modificador?
  - Elementos estáticos são membros de uma classe (como variáveis, métodos ou blocos de inicialização) que pertencem à própria classe, e não a uma instância específica dessa classe. Isso significa que eles podem ser acessados diretamente pela classe, sem a necessidade de criar um objeto dessa classe.

  - O método main é o ponto de entrada de um programa Java. Quando o programa é iniciado, o ambiente de execução da JVM (Java Virtual Machine) invoca o método main sem criar uma instância da classe que o contém. Como o main é o primeiro método executado, ele precisa ser acessível sem a criação de um objeto. Torná-lo estático permite que a JVM o chame diretamente usando apenas o nome da classe. Isso evita a necessidade de criar uma instância da classe apenas para iniciar a execução do programa, simplificando o processo de inicialização.

- Para que serve a classe Scanner?
  - A classe Scanner é usada para ler dados de várias fontes de entrada, como a entrada padrão do console (System.in), arquivos, strings, entre outros. A classe pode ler diferentes tipos de dados, como inteiros, strings, e números de ponto flutuante, facilitando a

captura de entrada do usuário e a conversão para o tipo apropriado.

- Como o uso de classes de repositório impactou na organização do código?

1. **Separa a Lógica de Negócios e Persistência**: Classes de repositório, como PessoaFisicaRepo e PessoaJuridicaRepo, ajudam a separar a lógica de persistência de dados da lógica de negócios. Isso significa que a lógica para acessar e manipular dados é isolada em uma camada específica, tornando o código mais organizado e modular.
2. **Facilidade de Manutenção**: Se for necessário alterar a forma como os dados são persistidos (por exemplo, de arquivos para um banco de dados), as mudanças podem ser feitas apenas nas classes de repositório, sem impactar o restante do sistema.
3. **Reutilização de Código**: Com as classes de repositório, a lógica de acesso a dados pode ser reutilizada em várias partes do sistema sem duplicação de código. Isso promove a manutenção do código, pois alterações no acesso aos dados podem ser feitas em um único lugar.