



**UNIVERSIDADE ESTÁCIO DE SÁ**  
**POLO ALCÂNTARA – SÃO GONÇALO/RJ**  
**TECNOLOGIA EM DESENVOLVIMENTO FULL STACK**  
**RPG0014 / 9001 - INICIANDO O CAMINHO PELO JAVA**

**LUIZ IZAIAS TEIXEIRA**

**RELATÓRIO DA MISSÃO PRÁTICA**

LUIZ IZAIAS TEIXEIRA

## RELATÓRIO DA MISSÃO PRÁTICA

Relatório apresentado à disciplina RPG0014 -  
Iniciando o Caminho Pelo Java do curso de  
Tecnologia em Desenvolvimento Full Stack da  
Universidade Estácio de Sá.

Tutor(a): Prof. Josyane Souza

## 1 INTRODUÇÃO

Este relatório visa relatar e analisar a prática realizada no contexto do desenvolvimento de sistemas em Java, abordando a implementação de um cadastro de clientes em modo texto com persistência em arquivos. A missão prática propõe a aplicação de conceitos fundamentais como herança, polimorfismo, persistência de objetos em arquivos binários e tratamento de exceções na linguagem Java.

A prática busca proporcionar a oportunidade de consolidar conhecimentos teóricos por meio da execução de um projeto prático. O desenvolvimento do sistema cadastral em Java abrange a criação de entidades, implementação de gerenciadores, utilização da interface *Serializable*, além da realização de testes e ajustes para garantir o correto funcionamento do sistema.

Ao final da atividade prática, espera-se um sistema cadastral funcional, mas também compreender e aplicar as boas práticas de programação orientada a objetos, organização de código e persistência de dados. A prática, realizada individualmente, visa fortalecer as habilidades básicas necessárias para programação em Java, com ênfase na persistência em arquivos binários.

A seguir, apresentaremos os conceitos utilizados, os procedimentos realizados, os resultados obtidos e as análises pertinentes ao desenvolvimento do sistema, culminando em uma conclusão que aborda vantagens e desvantagens das técnicas utilizadas, além de reflexões sobre os conceitos aprendidos e aplicados durante a prática.

## 2 PROGRAMAÇÃO ORIENTADA A OBJETOS

A Programação Orientada a Objetos (POO) é um paradigma de programação que se baseia na concepção de objetos, que podem conter dados e códigos que operam sobre esses dados. Essa abordagem visa modelar o mundo real, representando entidades e suas interações de forma mais fiel, facilitando a compreensão e manutenção do código. (VERSOLATTO, 2023, p. 3).

### 2.1 Pilares da Programação Orientada a Objetos

De acordo com Silva (2023) e Versolatto (2023), para entender a POO,

enquanto um paradigma de programação, faz-se necessário a compreensão de seus pilares, sendo eles: a abstração, o encapsulamento, a herança e o polimorfismo.

- a) **Abstração:** A abstração refere-se à capacidade de simplificar um sistema complexo, destacando apenas os detalhes essenciais. Isso é alcançado por meio da criação de classes e objetos, que encapsulam propriedades e comportamentos relevantes.
- b) **Encapsulamento:** O encapsulamento envolve o agrupamento de dados (atributos) e métodos (comportamentos) em uma única unidade chamada classe. Essa técnica ajuda a proteger os detalhes internos de uma implementação e permite o controle do acesso aos dados.
- c) **Herança:** A herança permite que uma classe herde características (atributos e métodos) de outra. Isso promove a reutilização de código e facilita a criação de hierarquias, onde classes mais especializadas podem estender ou modificar o comportamento de classes mais genéricas.
- d) **Polimorfismo:** O polimorfismo refere-se à capacidade de um objeto assumir várias formas. Em Java, isso é frequentemente alcançado por meio de sobrecarga de métodos e interfaces. O polimorfismo permite que objetos de diferentes tipos sejam tratados de maneira uniforme.

### **3 LINGUAGEM DE PROGRAMAÇÃO JAVA**

Java é uma linguagem de programação projetada desde o início com os princípios da Programação Orientada a Objetos criada por James Gosling em 1995 quando ele trabalhava na Sun Microsystems.

A abordagem de POO oferece diversos benefícios, proporcionando um ambiente de desenvolvimento modular, reutilizável e de fácil manutenção. A seguir, destacamos aspectos-chave que evidenciam o comprometimento do Java com a POO (PALMEIRA, 2012; LEMOS, 2009):

- a) Em Java, a classe é a unidade fundamental de organização do código. Uma classe define a estrutura e o comportamento dos objetos que serão criados a partir dela. O encapsulamento é enfatizado, permitindo que os detalhes internos de uma classe sejam protegidos e acessados apenas

por meio de interfaces bem definidas.

- b) A programação em Java é centrada em objetos. Os objetos são instâncias de classes e encapsulam dados e comportamentos relacionados. Essa abordagem facilita a modelagem de entidades reais de maneira intuitiva e eficiente.
- c) Java suporta herança, permitindo que uma classe herde atributos e métodos de outra. Isso promove a reutilização de código e a criação de hierarquias de classes. A herança é fundamental para estabelecer relações entre classes e modelar conceitos como especialização e generalização.
- d) O polimorfismo é um princípio central na POO, e Java oferece suporte robusto a essa característica. O polimorfismo permite que objetos de diferentes tipos sejam tratados de maneira uniforme. Em Java, isso é alcançado por meio de interfaces, classes abstratas e sobrecarga de métodos.
- e) Java incentiva o encapsulamento, restringindo o acesso direto aos atributos de uma classe. Isso é alcançado por meio de modificadores de acesso, como *public*, *private* e *protected*, que controlam a visibilidade dos membros de uma classe.

Em resumo, Java é uma linguagem que incorpora os princípios da POO abrangentemente, proporcionando uma base sólida para o desenvolvimento de sistemas escaláveis, modulares e orientados a objetos. O uso desses conceitos na prática citada demonstra a aplicação prática dos fundamentos da POO em um ambiente de programação Java.

## 4 METODOLOGIA

Inicialmente, foi realizado um levantamento teórico sobre os conceitos-chave abordados na prática, incluindo herança, polimorfismo, persistência de objetos em arquivos binários, interface *Serializable*, e a utilização da API Stream no contexto funcional.

O roteiro da prática foi analisado cuidadosamente para entender as etapas e requisitos propostos. Cada parte do desenvolvimento foi dividida e

detalhada, identificando os pontos críticos que demandariam atenção especial.

Foi garantido que ambiente de desenvolvimento Java estivesse configurado corretamente, incluindo a instalação do JDK, sendo a versão 17.0.6 utilizada durante a implementação, e da IDE NetBeans. A escolha dessas ferramentas foi baseada na compatibilidade e facilidade de uso.

A implementação do projeto foi conduzida de forma gradual e modular. Começando com a definição das entidades e seus relacionamentos, a equipe seguiu as instruções do roteiro, implementando cada parte e realizando testes unitários para garantir o funcionamento adequado.

Após a implementação, foi dedicado tempo significativo à fase de testes. Foram realizados testes individuais para cada parte do sistema, verificando a correta persistência e recuperação de dados, o funcionamento das operações de cadastro em modo texto, e a robustez do tratamento de exceções.

O projeto foi armazenado em um repositório Git e a documentação em formato PDF foi gerada. O link do repositório e a documentação foram compartilhados com o tutor para avaliação.

## 5 RESULTADOS E ANÁLISES

O projeto foi estruturado em pacotes, com destaque para o pacote *model* que abriga as entidades e gerenciadores. A abordagem modular permite uma organização eficiente e facilita a manutenção do código.

No pacote *model*, encontramos as classes *Pessoa*, *PessoaFisica* e *PessoaJuridica*. A classe base *Pessoa* contém os campos comuns a todas as entidades, e as classes derivadas incorporam campos específicos. Além disso, a implementação da interface *Serializable*, em todas as classes, permite a persistência fácil e segura em arquivos binários. Seguindo os códigos elaborados a seguir:

### Código 1: Classe Pessoa

```
package model;
import java.io.Serializable;
public class Pessoa implements Serializable {
    private static final long serialVersionUID = 1L;
    private int id;
    private String nome;
    public Pessoa(int id, String nome) {
        this.id = id;
    }
}
```

```

        this.nome = nome;
    }
    public int getId() {
        return id;
    }
    public String getNome() {
        return nome;
    }
    public void setNome(String nome) {
        this.nome = nome;
    }
    public void setId(int id) {
        this.id = id;
    }
    public void exibir(){
        System.out.println("ID: " + this.id);
        System.out.println("Nome: " + this.nome);
    }
}

```

## Código 2: Classe PessoaFisica

```

package model;
import java.io.Serializable;
public class PessoaFisica extends Pessoa implements Serializable {
    private static final long serialVersionUID = 1L;
    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;
    }
    public String getCpf() {
        return cpf;
    }
    public void setCpf(String cpf) {
        this.cpf = cpf;
    }
    public int getIdade() {
        return idade;
    }
    public void setIdade(int idade) {
        this.idade = idade;
    }
    @Override
    public void exibir() {
        super.exibir();
        System.out.println("CPF: " + this.cpf);
        System.out.println("Idade: " + this.idade);
    }
}

```

## Código 3: Classe PessoaJuridica

```

package model;
import java.io.Serializable;
public class PessoaJuridica extends Pessoa implements Serializable {

```

```

private static final long serialVersionUID = 1L;
private String cnpj;
public PessoaJuridica(int id, String nome, String cnpj) {
    super(id, nome);
    this.cnpj = cnpj;
}
public String getCnpj() {
    return cnpj;
}
public void setCnpj(String cnpj) {
    this.cnpj = cnpj;
}
@Override
public void exibir() {
    System.out.println("CNPJ: " + this.cnpj);
}
}

```

Os gerenciadores, como PessoaFisicaRepo e PessoaJuridicaRepo, são responsáveis por manipular as entidades e realizar operações como inserção, alteração, exclusão, recuperação e persistência. A presença dos métodos persistir e recuperar, que lançam exceções, garante um controle robusto das operações de armazenamento e recuperação de dados. Como apresentados a seguir:

#### Código 4: Classe PessoaFisicaRepo

```

package model;
import java.io.*;
import java.util.ArrayList;
import java.util.List;
public class PessoaFisicaRepo {
    private final List<PessoaFisica> listaPessoasFisicas = new ArrayList<>();
    public void inserir(PessoaFisica pessoaFisica) {
        listaPessoasFisicas.add(pessoaFisica);
    }
    public void alterar(PessoaFisica pessoaFisica) {
        for (int i = 0; i < listaPessoasFisicas.size(); i++) {
            if (pessoaFisica.getId() == listaPessoasFisicas.get(i).getId()) {
                listaPessoasFisicas.set(i, pessoaFisica);
                return;
            }
        }
    }
    public void excluir(int id) {
        for (int i = 0; i < listaPessoasFisicas.size(); i++) {
            if (listaPessoasFisicas.get(i).getId() == id) {
                listaPessoasFisicas.remove(i);
                return;
            }
        }
    }
    public PessoaFisica obter(int id) {
        for (PessoaFisica pessoaFisica : listaPessoasFisicas) {
            if (pessoaFisica.getId() == id) {

```



```

        return pessoaFisica;
    }
    }
    return null;
}
public List<PessoaFisica> obterTodos() {
    return listaPessoasFisicas;
}
public void persistir(String nomeArquivo) throws IOException {
    try (ObjectOutputStream outputStream = new ObjectOutputStream(new
FileOutputStream(nomeArquivo))) {
        outputStream.writeObject(listaPessoasFisicas);
    } catch (IOException e) {
        throw e;
    }
}
public void recuperar(String nomeArquivo) throws IOException,
ClassNotFoundException {
    try (ObjectInputStream inputStream = new ObjectInputStream(new
FileInputStream(nomeArquivo))) {
        listaPessoasFisicas.clear();
        List<PessoaFisica> listaRecuperada = (List<PessoaFisica>)
inputStream.readObject();
        listaPessoasFisicas.addAll(listaRecuperada);
    } catch (IOException | ClassNotFoundException e) {
        throw e;
    }
}
}
}

```

### Código 5: Classe PessoaFisicaRepo

```

package model;
import java.io.*;
import java.util.ArrayList;
import java.util.List;
public class PessoaJuridicaRepo {
    private final List<PessoaJuridica> listaPessoasJuridicas = new
ArrayList<>();
    public void inserir(PessoaJuridica pessoaJuridica) {
        listaPessoasJuridicas.add(pessoaJuridica);
    }
    public void alterar(PessoaJuridica pessoaJuridica) {
        for (int i = 0; i < listaPessoasJuridicas.size(); i++) {
            if (pessoaJuridica.getId() == listaPessoasJuridicas.get(i).getId())
{
                listaPessoasJuridicas.set(i, pessoaJuridica);
                return;
            }
        }
    }
    public void excluir(int id) {
        for (int i = 0; i < listaPessoasJuridicas.size(); i++) {
            if (listaPessoasJuridicas.get(i).getId() == id) {
                listaPessoasJuridicas.remove(i);
                return;
            }
        }
    }
}

```

```

    }
    public PessoaJuridica obter(int id) {
        for (PessoaJuridica pessoaJuridica : listaPessoasJuridicas) {
            if (pessoaJuridica.getId() == id) {
                return pessoaJuridica;
            }
        }
        return null;
    }
    public List<PessoaJuridica> obterTodos() {
        return listaPessoasJuridicas;
    }
    public void persistir(String nomeArquivo) throws IOException {
        try (ObjectOutputStream outputStream = new ObjectOutputStream(new
        FileOutputStream(nomeArquivo))) {
            outputStream.writeObject(listaPessoasJuridicas);
        } catch (IOException e) {
            throw e;
        }
    }
    public void recuperar(String nomeArquivo) throws IOException,
    ClassNotFoundException {
        try (ObjectInputStream inputStream = new ObjectInputStream(new
        FileInputStream(nomeArquivo))) {
            listaPessoasJuridicas.clear();
            List<PessoaJuridica> listaRecuperada = (List<PessoaJuridica>)
inputStream.readObject();
            listaPessoasJuridicas.addAll(listaRecuperada);
        } catch (IOException | ClassNotFoundException e) {
            throw e;
        }
    }
}

```

O método *main* da classe principal foi implementado para instanciar repositórios, adicionar dados, persistir e recuperar esses dados (Código 6). Os resultados esperados incluem uma execução organizada e a correta persistência e recuperação dos dados.

### Código 6: Classe Main

```

import model.PessoaFisica;
import model.PessoaFisicaRepo;
import model.PessoaJuridica;
import model.PessoaJuridicaRepo;
public class Main {
    public static void main(String[] args) throws Exception {
        PessoaFisicaRepo repo1 = new PessoaFisicaRepo();
        PessoaFisica pessoa1 = new PessoaFisica(1, "Ana", "1111111111", 25);
        PessoaFisica pessoa2 = new PessoaFisica(2, "Carlos", "2222222222", 52);
        repo1.inserir(pessoa1);
        repo1.inserir(pessoa2);
        repo1.persistir("dados-pf1");
        PessoaFisicaRepo repo2 = new PessoaFisicaRepo();
        repo2.recuperar("dados-pf1");
    }
}

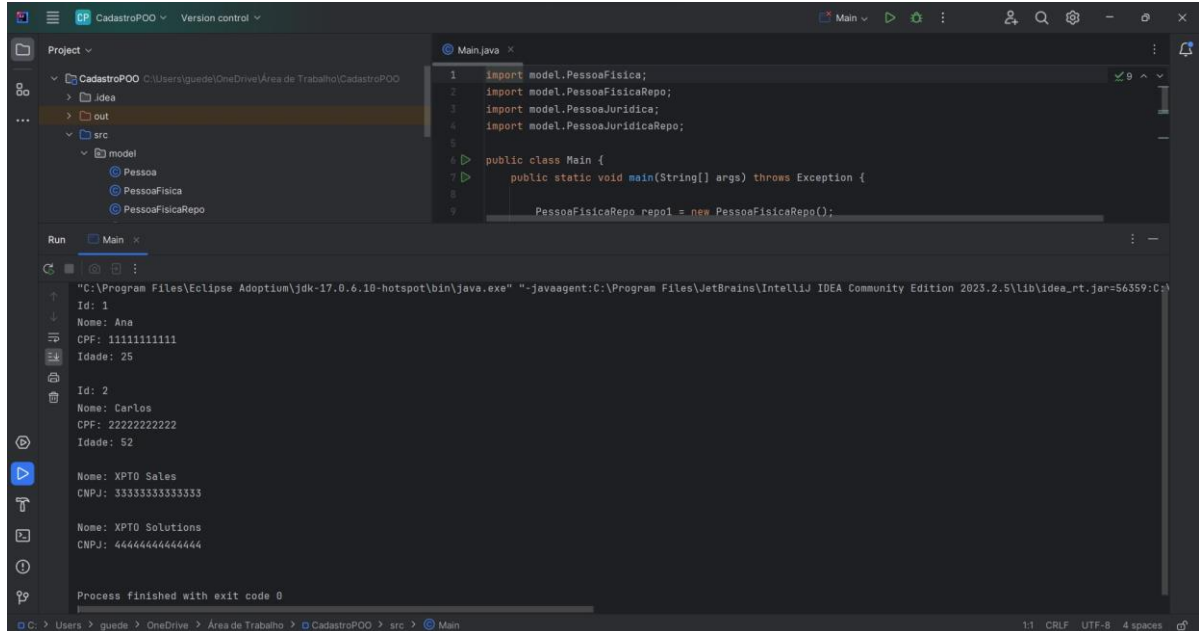
```

```

    for (PessoaFisica pessoa : repo2.obterTodos()) {
        System.out.println("Id: " + pessoa.getId());
        System.out.println("Nome: " + pessoa.getNome());
        System.out.println("CPF: " + pessoa.getCpf());
        System.out.println("Idade: " + pessoa.getIdade());
        System.out.println();
    }
    PessoaJuridicaRepo repo3 = new PessoaJuridicaRepo();
    PessoaJuridica pessoaJuridica1 = new PessoaJuridica(1, "XPTO Sales",
"3333333333333333");
    PessoaJuridica pessoaJuridica2 = new PessoaJuridica(2, "XPTO Solutions",
"4444444444444444");
    repo3.inserir(pessoaJuridica1);
    repo3.inserir(pessoaJuridica2);
    repo3.persistir("dados-pj1");
    PessoaJuridicaRepo repo4 = new PessoaJuridicaRepo();
    repo4.recuperar("dados-pj1");
    for (PessoaJuridica pessoaJuridica : repo4.obterTodos()) {
        System.out.println("Nome: " + pessoaJuridica.getNome());
        System.out.println("CNPJ: " + pessoaJuridica.getCnpj());
        System.out.println();
    }
}
}
}

```

**Figura 1 - Execução da classe Main**



A segunda parte do desenvolvimento, apresentada no método *main2*, introduz uma interação textual com o usuário. As opções incluem incluir, alterar, excluir, buscar e exibir dados, além de persistir e recuperar informações. As exceções são tratadas de maneira adequada para garantir a robustez do sistema, conforme solicitado no segundo procedimento da atividade prática (Código 7 e Figura 2).

## Código 7: Classe Main2

```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import model.PessoaFisica;
import model.PessoaFisicaRepo;
import model.PessoaJuridica;
import model.PessoaJuridicaRepo;
public class Main2 {
    public static void main(String[] args) {
        BufferedReader reader = new BufferedReader(new
InputStreamReader(System.in));
        String opcao = "";
        PessoaFisicaRepo repoFisica = new PessoaFisicaRepo();
        PessoaJuridicaRepo repoJuridica = new PessoaJuridicaRepo();
        while (!"0".equals(opcao)) {
            System.out.println("=====");
            System.out.println("1 - Incluir Pessoa");
            System.out.println("2 - Alterar Pessoa");
            System.out.println("3 - Excluir Pessoa");
            System.out.println("4 - Buscar pelo Id");
            System.out.println("5 - Exibir Todos");
            System.out.println("6 - Persistir Dados");
            System.out.println("7 - Recuperar Dados");
            System.out.println("0 - Finalizar Programa");
            System.out.println("=====");
            try {
                opcao = reader.readLine();
                switch (opcao) {
                    case "1":
                        System.out.println("F - Pessoa Física | J - Pessoa
Jurídica");
                        String tipoPessoa = reader.readLine();
                        switch (tipoPessoa) {
                            case "F":
                                PessoaFisica pf = lerDadosPessoaFisica(reader);
                                repoFisica.inserir(pf);
                                System.out.println("Pessoa Física incluída com
sucesso.");
                                break;
                            case "J":
                                PessoaJuridica pj =
lerDadosPessoaJuridica(reader);
                                repoJuridica.inserir(pj);
                                System.out.println("Pessoa Jurídica incluída com
sucesso.");
                                break;
                            default:
                                System.out.println("Tipo inválido.");
                        }
                        break;
                    case "2":
                        System.out.println("F - Pessoa Física | J - Pessoa
Jurídica");
                        String tipoPessoaAlterar = reader.readLine();
                        switch (tipoPessoaAlterar) {
                            case "F":
```

```

        alterarPessoa(repoFisica, reader);
        System.out.println("Pessoa Física alterada com
sucesso.");
        break;
    case "J":
        alterarPessoa(repoJuridica, reader);
        System.out.println("Pessoa Jurídica alterada com
sucesso.");
        break;
    default:
        System.out.println("Tipo inválido.");
    }
    break;
case "3":
    System.out.println("F - Pessoa Física | J - Pessoa
Jurídica");
    String tipoPessoaExcluir = reader.readLine();
    switch (tipoPessoaExcluir) {
        case "F":
            excluirPessoa(repoFisica, reader);
            System.out.println("Pessoa Física excluída com
sucesso.");
            break;
        case "J":
            excluirPessoa(repoJuridica, reader);
            System.out.println("Pessoa Jurídica excluída com
sucesso.");
            break;
        default:
            System.out.println("Tipo inválido.");
    }
    break;
case "4":
    System.out.println("F - Pessoa Física | J - Pessoa
Jurídica");
    String tipoPessoaBuscar = reader.readLine();
    switch (tipoPessoaBuscar) {
        case "F":
            buscarPessoa(repoFisica, reader);
            System.out.println("Pessoa Física encontrada com
sucesso.");
            break;
        case "J":
            buscarPessoa(repoJuridica, reader);
            System.out.println("Pessoa Jurídica encontrada
com sucesso.");
            break;
        default:
            System.out.println("Tipo inválido.");
    }
    break;
case "5":
    System.out.println("F - Pessoa Física | J - Pessoa
Jurídica");
    String tipoPessoaExibirTodos = reader.readLine();
    switch (tipoPessoaExibirTodos) {
        case "F":
            exibirTodasPessoas(repoFisica);

```

```

        System.out.println("Listagem de todas as Pessoas
Físicas cadastradas.");
        break;
    case "J":
        exibirTodasPessoas(repoJuridica);
        System.out.println("Listagem de todas as Pessoas
Jurídicas cadastradas.");
        break;
    default:
        System.out.println("Tipo inválido.");
    }

    break;
case "6":
    System.out.print("Qual o nome dos arquivos? ");
    String arquivoP = reader.readLine();
    try {
        repoFisica.persistir(arquivoP + ".fisica.bin");
        repoJuridica.persistir(arquivoP + ".juridica.bin");
        System.out.println("Dados salvos com sucesso.");
    } catch (IOException e) {
        System.out.println("Erro ao salvar os dados: " +
e.getMessage());
    }
    break;
case "7":
    System.out.print("Qual o nome dos arquivos? ");
    String arquivoR = reader.readLine();
    try {
        repoFisica.recuperar(arquivoR + ".fisica.bin");
        repoJuridica.recuperar(arquivoR + ".juridica.bin");
        System.out.println("Dados recuperados com
sucesso.");
    } catch (IOException | ClassNotFoundException e) {
        System.out.println("Erro ao recuperar os dados: " +
e.getMessage());
    }
    break;
case "0":
    System.out.println("Finalizando o programa...");
    break;
default:
    System.out.println("Opção inválida!");
    break;
    }
} catch (IOException e) {
    System.out.println("Erro de entrada/saída: " + e.getMessage());
}
}

private static PessoaFisica lerDadosPessoaFisica(BufferedReader reader)
throws IOException {
    System.out.println("Digite o id da pessoa: ");
    int id = Integer.parseInt(reader.readLine());
    System.out.println("Insira os dados...");
    System.out.print("Nome: ");
    String nome = reader.readLine();
    System.out.print("CPF: ");
    String cpf = reader.readLine();

```

```

        System.out.print("Idade: ");
        int idade = Integer.parseInt(reader.readLine());
        return new PessoaFisica(id, nome, cpf, idade);
    }
    private static PessoaJuridica lerDadosPessoaJuridica(BufferedReader reader)
throws IOException {
        System.out.println("Digite o id da pessoa: ");
        int id = Integer.parseInt(reader.readLine());
        System.out.println("Insira os dados...");
        System.out.print("Nome: ");
        String nome = reader.readLine();
        System.out.print("CNPJ: ");
        String cnpj = reader.readLine();
        return new PessoaJuridica(id, nome, cnpj);
    }
    private static void alterarPessoa(Object repo, BufferedReader reader) throws
IOException {
        System.out.println("Digite o id da pessoa: ");
        int id = Integer.parseInt(reader.readLine());
        if (repo instanceof PessoaFisicaRepo) {
            PessoaFisica pf = ((PessoaFisicaRepo) repo).obter(id);
            if (pf != null) {
                System.out.println("Nome atual: " + pf.getNome());
                System.out.print("Novo nome: ");
                pf.setNome(reader.readLine());
                System.out.println("CPF atual: " + pf.getCpf());
                System.out.print("Novo CPF: ");
                pf.setCpf(reader.readLine());
                System.out.println("Idade atual: " + pf.getIdade());
                System.out.print("Nova idade: ");
                pf.setIdade(Integer.parseInt(reader.readLine()));
                ((PessoaFisicaRepo) repo).alterar(pf);
            } else {
                System.out.println("Pessoa Física não encontrada.");
            }
        } else if (repo instanceof PessoaJuridicaRepo) {
            PessoaJuridica pj = ((PessoaJuridicaRepo) repo).obter(id);
            if (pj != null) {
                System.out.println("Nome atual: " + pj.getNome());
                System.out.print("Novo nome: ");
                pj.setNome(reader.readLine());
                System.out.println("CNPJ atual: " + pj.getCnpj());
                System.out.print("Novo CNPJ: ");
                pj.setCnpj(reader.readLine());
                ((PessoaJuridicaRepo) repo).alterar(pj);
            } else {
                System.out.println("Pessoa Jurídica não encontrada.");
            }
        }
    }
    private static void excluirPessoa(Object repo, BufferedReader reader) throws
IOException {
        System.out.print("Digite o Id do usuário: ");
        int id = Integer.parseInt(reader.readLine());
        if (repo instanceof PessoaFisicaRepo) {
            ((PessoaFisicaRepo) repo).excluir(id);
        } else if (repo instanceof PessoaJuridicaRepo) {
            ((PessoaJuridicaRepo) repo).excluir(id);
        }
    }
}

```

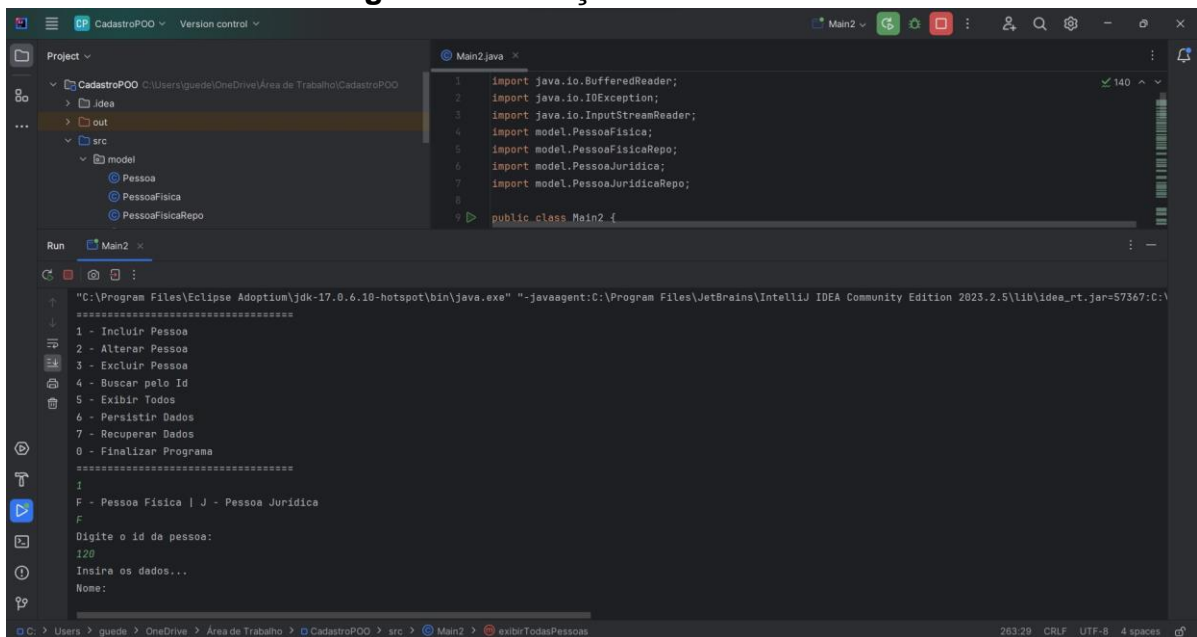
```

    }
}
private static void buscarPessoa(Object repo, BufferedReader reader) throws
IOException {
    System.out.print("Digite o id da pessoa: ");
    int id = Integer.parseInt(reader.readLine());
    if (repo instanceof PessoaFisicaRepo) {
        PessoaFisica pf = ((PessoaFisicaRepo) repo).obter(id);
        if (pf != null) {
            System.out.println("Id: " + pf.getId());
            System.out.println("Nome: " + pf.getNome());
            System.out.println("CPF: " + pf.getCpf());
            System.out.println("Idade: " + pf.getIdade());
        } else {
            System.out.println("Pessoa Física não encontrada.");
        }
    } else if (repo instanceof PessoaJuridicaRepo) {
        PessoaJuridica pj = ((PessoaJuridicaRepo) repo).obter(id);
        if (pj != null) {
            System.out.println("Id: " + pj.getId());
            System.out.println("Nome: " + pj.getNome());
            System.out.println("CNPJ: " + pj.getCnpj());
        } else {
            System.out.println("Pessoa Jurídica não encontrada.");
        }
    }
}
private static void exibirTodasPessoas(Object repo) {
    if (repo instanceof PessoaFisicaRepo) {
        for (PessoaFisica pf : ((PessoaFisicaRepo) repo).obterTodos()) {
            System.out.println("Id: " + pf.getId());
            System.out.println("Nome: " + pf.getNome());
            System.out.println("CPF: " + pf.getCpf());
            System.out.println("Idade: " + pf.getIdade());
        }
    } else if (repo instanceof PessoaJuridicaRepo) {
        for (PessoaJuridica pj : ((PessoaJuridicaRepo) repo).obterTodos()) {
            System.out.println("Id: " + pj.getId());
            System.out.println("Nome: " + pj.getNome());
            System.out.println("CNPJ: " + pj.getCnpj());
        }
    }
}
}

```



**Figura 2: Execução da classe Main2**



Durante a execução do código, foram observados resultados satisfatórios. Os repositórios de pessoas físicas e jurídicas foram instanciados, dados foram adicionados, persistidos e recuperados com sucesso. A estrutura organizada do código, aliada ao uso eficiente das funcionalidades do NetBeans, contribuiu para uma implementação bem-sucedida do sistema cadastral.

## 6 DISCUSSÕES

A prática proporcionou uma compreensão aprofundada dos conceitos teóricos discutidos, permitindo a aplicação prática desses conhecimentos na construção do sistema cadastral. As vantagens da programação orientada a objetos, aliadas à correta utilização de persistência em arquivos binários, resultaram em um código funcional.

### 6.1 Quais as vantagens e desvantagens do uso de herança?

A utilização de herança e polimorfismo foi essencial na definição de entidades no contexto da programação orientada a objetos. A classe base Pessoa serviu como um exemplo prático desses conceitos, onde PessoaFisica e PessoaJuridica herdaram características da classe mãe, permitindo a extensão e especialização do comportamento.

### 6.1.1 Vantagens

A herança permite que as classes filhas herdem os atributos e métodos da classe mãe, promovendo a reutilização de código. Além disso, proporciona uma estrutura hierárquica, facilitando a compreensão e organização do sistema. E permite o uso de polimorfismo, onde objetos de classes diferentes podem ser tratados de maneira uniforme.

### 6.1.2 Desvantagens

A herança excessiva pode levar a um alto acoplamento entre as classes, tornando o sistema mais complexo e difícil de manter. Fora isso, modificações na classe mãe podem impactar as classes filhas, tornando o sistema menos flexível.

## 6.2 Por que a interface ***Serializable*** é necessária ao efetuar persistência em arquivos binários?

A persistência de objetos em arquivos binários é um aspecto crucial na manipulação e armazenamento de dados. Por isso, a interface *Serializable* foi empregada em todas as classes envolvidas no projeto, permitindo que os objetos sejam convertidos em sequências de bytes para serem armazenados e recuperados posteriormente.

A interface *Serializable* serve como uma marcação para indicar ao Java que uma classe pode ser serializada, ou seja, convertida em bytes. Essencial para operações de persistência em arquivos binários, possibilitando a gravação e leitura eficientes de objetos.

## 6.3 Como o paradigma funcional é utilizado pela API Stream no Java?

A API Stream em Java adota o paradigma funcional para operações de processamento de dados. Utilizando expressões lambda, a API Stream permite operações poderosas e concisas em coleções, contribuindo para um código mais limpo e legível.

Expressões Lambda permitem a definição de funções anônimas concisamente. Operações de filtragem e mapeamento, como os métodos *filter* e *map* aplicam operações funcionalmente em elementos da coleção.

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

Ao trabalhar com Java, um padrão de desenvolvimento comumente adotado na persistência de dados em arquivos é a utilização da interface *Serializable*, já abordada e tendo sido utilizada durante a implementação desta atividade prática. Essa abordagem é especialmente valiosa para persistir objetos eficientemente e estruturada, mantendo a integridade dos dados.

#### **6.5 O que são elementos estáticos e qual o motivo para o método *main* adotar esse modificador?**

Elementos estáticos em Java são associados à classe, não à instância de um objeto. Isso significa que pertencem à classe na totalidade, e não a uma cópia específica do objeto. Métodos e variáveis estáticos são declarados usando a palavra-chave *static* e podem ser acessados sem a necessidade de criar uma instância da classe.

O método *main* em Java é o ponto de entrada para a execução de um programa. Ele é chamado pelo sistema quando o programa é iniciado. Por isso o modificador *static* é utilizado no método *main* na classe *Main2*, para indicar que esse método pertence à classe em vez de uma instância específica. Isso permite que o método seja chamado sem a necessidade de criar um objeto da classe, tornando-o acessível diretamente pela JVM (Java Virtual Machine).

#### **6.6 Para que serve a classe *Scanner*?**

A classe *Scanner* em Java é utilizada para obter entradas do usuário a partir do console. Ela fornece métodos simples para ler diferentes tipos de dados, como inteiros, *doubles* e *strings*, facilitando a interação com o usuário durante a

execução do programa. O Scanner é uma ferramenta valiosa para a entrada de dados dinâmica e interativa.

A classe `BufferedReader` foi utilizada para obter entradas do usuário a partir do console no método *main*. Embora não seja diretamente a classe `Scanner`, o princípio foi o mesmo - obter dados do usuário interativamente. O `BufferedReader` oferece métodos para ler diferentes tipos de dados, e sua utilização no projeto possibilitou uma interação dinâmica durante a execução do programa.

## **6.7 Como o uso de classes de repositório impactou na organização do código?**

A introdução de classes de repositório teve um impacto positivo na organização do código. Essas classes atuam como intermediárias entre o programa e os dados persistidos, encapsulando a lógica de armazenamento e recuperação. Isso resulta em um código mais modular e coeso, facilitando a manutenção e evolução do sistema. Além disso, o uso de classes de repositório segue o princípio de responsabilidade única, tornando o código mais compreensível e aderente às boas práticas de desenvolvimento.

O projeto incorporou classes de repositório, como `PessoaFisicaRepo` e `PessoaJuridicaRepo`, que desempenharam um papel fundamental na organização do código. Essas classes encapsulam a lógica de gerenciamento de entidades e a persistência em arquivos. O código principal (*Main2*) utiliza esses repositórios modularmente, invocando métodos específicos para realizar operações de CRUD (Create, Read, Update, Delete) nas entidades.

## **7 CONSIDERAÇÕES FINAIS**

O desenvolvimento desta atividade prática proporcionou uma imersão nos conceitos fundamentais da Programação Orientada a Objetos (POO) e na manipulação de persistência de dados em Java. A implementação do sistema cadastral, com enfoque nas entidades `Pessoa`, `PessoaFisica`, e `PessoaJuridica`, proporcionou uma compreensão aprofundada sobre herança, polimorfismo e persistência de objetos em arquivos binários.

Ao abordar o tema da herança, pudemos explorar as vantagens, como a reutilização de código e a promoção da estrutura hierárquica. No entanto,

identificamos desafios, como o potencial aumento da complexidade à medida que a hierarquia cresce.

A interface *Serializable* desempenhou um papel crucial na persistência de objetos em arquivos binários. Através do uso de *ObjectOutputStream* e *ObjectInputStream*, foi possível transformar objetos em bytes e armazená-los eficientemente em arquivos. A necessidade dessa interface para a serialização destacou a importância da segurança na transmissão e armazenamento de objetos.

O padrão de desenvolvimento adotado para a persistência de dados em arquivos, utilizando a interface *Serializable* e classes de repositório, proporcionou uma organização eficaz do código.

Diante disso, concluímos que a atividade prática não apenas atendeu aos objetivos propostos, mas também ofereceu uma base sólida para compreender os princípios da POO em Java e as estratégias de persistência de dados, consolidando conhecimentos essenciais para a jornada na programação orientada a objetos.

