



POLO CHAPADA - MANAUS - AM/UNIVERSIDADE ESTÁCIO DE SÁ

Curso: Desenvolvimento Full Stack

Disciplina Nível 1: RPG0014 - Iniciando o caminho pelo Java

Número da Turma: 2024.2

Semestre Letivo: Mundo-3

Aluno: Gilvan Júnior Nascimento Gonçalves

Matrícula: 202304560188

Missão Prática | Nível 1 | Mundo 3

Objetivo da Prática:

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

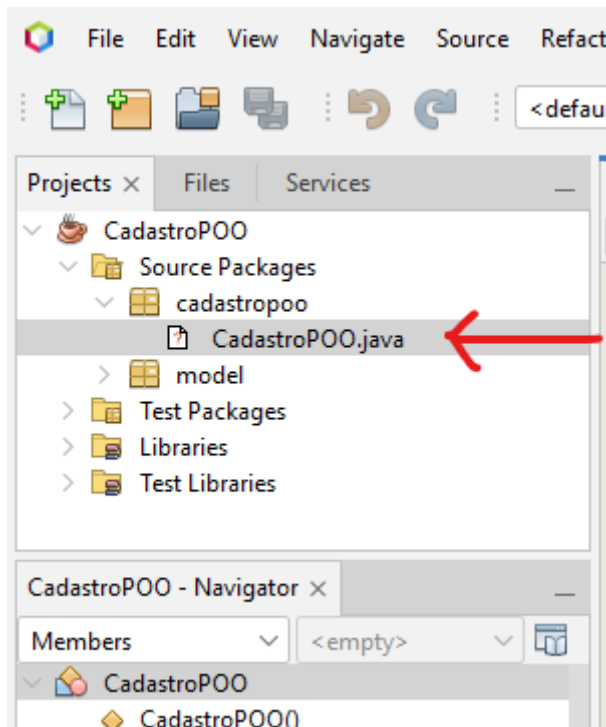
No final do projeto, o aluno terá implementado um sistema cadastral em Java, utilizando os recursos da programação orientada a objetos e a persistência em arquivos binários.

Todos os códigos solicitados neste roteiro de aula.

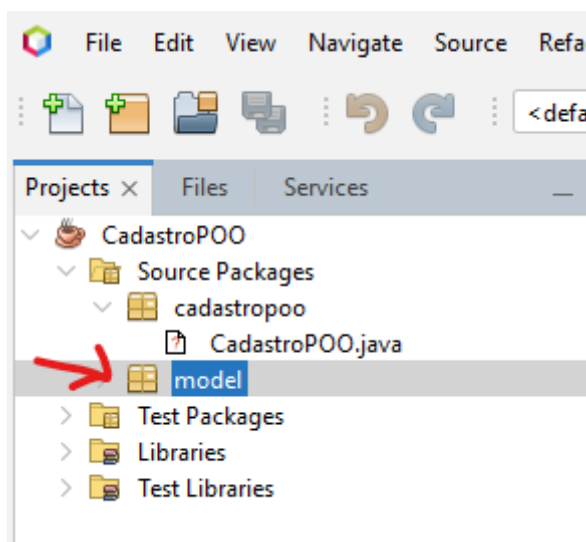
(Os resultados da execução dos códigos também devem ser apresentados)

1º Procedimento | Criação das Entidades e Sistema de Persistência

1. Criar um projeto do tipo Ant..Java Application no NetBeans, utilizando o nome CadastroPOO para o projeto:



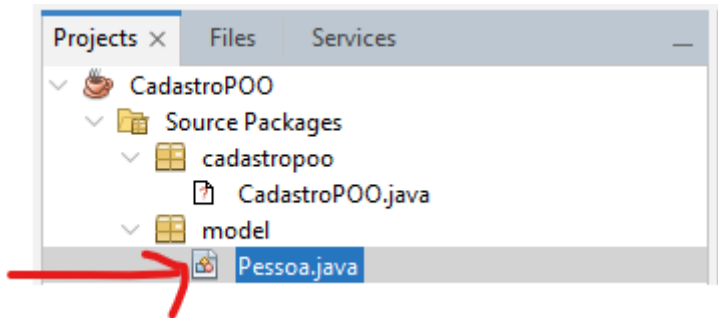
2. Criar um pacote com o nome "model", para as entidades e gerenciadores:



3. No pacote model criar as entidades, com as seguintes características:

a. Classe **Pessoa**, com os campos *id* (inteiro) e *nome* (texto), método *exibir*, para impressão dos dados, construtor padrão e completo, além de getters e setters para todos os campos:

obs: Adicionar interface *Serializable* em todas as classes:

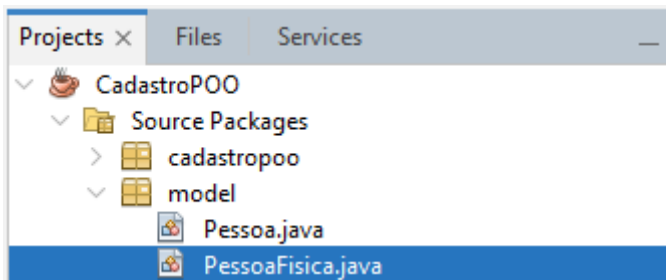


```
package model;
```

```
import java.io.Serializable;
```

```
public class Pessoa implements Serializable {  
    // Campos  
    private int id;  
    private String nome;  
  
    // Construtor padrão  
    public Pessoa() {  
    }  
    // Construtor completo  
    public Pessoa(int id, String nome) {  
        this.id = id;  
        this.nome = nome;  
    }  
    // Método para exibir os dados da pessoa  
    public void exibir() {  
        System.out.println("ID: " + id);  
        System.out.println("Nome: " + nome);  
    }  
  
    public int getId() {  
        return id;  
    }  
    public void setId(int id) {  
        this.id = id;  
    }  
    public String getNome() {  
        return nome;  
    }  
    public void setNome(String nome) {  
        this.nome = nome;  
    }  
}
```

b. Classe *PessoaFisica*, herdando de *Pessoa*, com o acréscimo dos campos *cpf* (texto) e *idade* (inteiro), método *exibir* polimórfico, construtores, getters e setters.



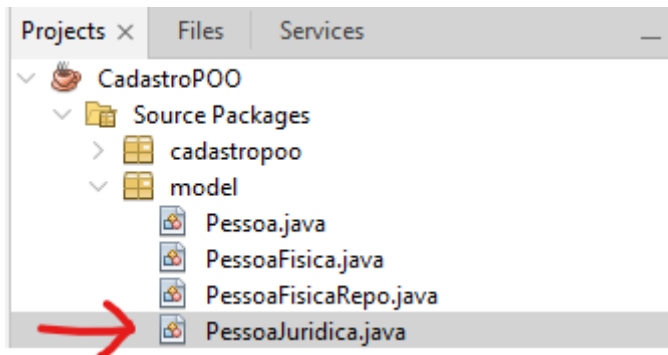
```
package model;
```

```
import java.io.Serializable;
```

```
public class PessoaFisica extends Pessoa implements Serializable {
    private String cpf;
    private int idade;

    // Construtor padrão
    public PessoaFisica() {
        super(); // Chamando o construtor padrão da classe pai
    }
    // Construtor completo
    public PessoaFisica(int id, String nome, String cpf, int idade) {
        super(id, nome); // Chamando o construtor completo da classe pai
        this.cpf = cpf;
        this.idade = idade;
    }
    // Método para exibir os dados da pessoa física (sobrescrito)
    @Override
    public void exibir() {
        super.exibir(); // Chamando o método exibir da classe pai
        System.out.println("CPF: " + cpf);
        System.out.println("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;
    }
}
```

c. Classe **PessoaJuridica**, herdando de *Pessoa*, com o acréscimo do campo *cnpj* (texto), método *exibir* polimórfico, construtores, getters e setters.



```
package model;
```

```
import java.io.Serializable;
```

```
public class PessoaJuridica extends Pessoa implements Serializable {
```

```
    private String cnpj;
```

```
    public PessoaJuridica() {
```

```
        super(); // Chamando o construtor padrão da classe pai
```

```
    }
```

```
    public PessoaJuridica(int id, String nome, String cnpj) {
```

```
        super(id, nome); // Chamando o construtor completo da classe pai
```

```
        this.cnpj = cnpj;
```

```
    }
```

```
    @Override
```

```
    public void exibir() {
```

```
        super.exibir(); // Chamando o método exibir da classe pai
```

```
        System.out.println("CNPJ: " + cnpj);
```

```
    }
```

```
    public String getCnpj() {
```

```
        return cnpj;
```

```
    }
```

```
    public void setCnpj(String cnpj) {
```

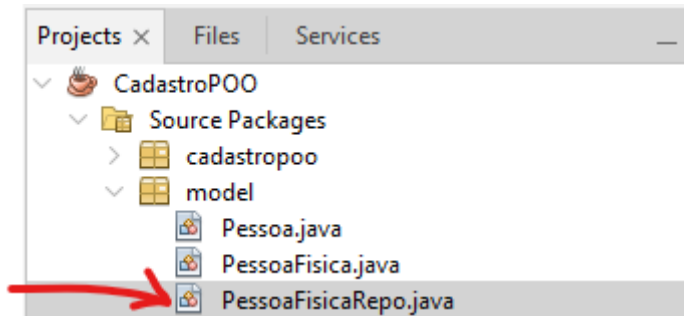
```
        this.cnpj = cnpj;
```

```
    }
```

```
}
```

4. No pacote model criar os gerenciadores, com as seguintes características:

a. Classe **PessoaFisicaRepo**, contendo um `ArrayList` de `PessoaFisica`, nível de acesso privado, e métodos públicos `inserir`, `alterar`, `excluir`, `obter` e `obterTodos`, para gerenciamento das entidades contidas no `ArrayList`.



```
package model;
```

```
import java.io.*;
import java.util.ArrayList;
import java.util.List;
```

```
public class PessoaFisicaRepo implements Serializable {
```

```
    private List<PessoaFisica> listaPessoasFisicas;
```

```
    public PessoaFisicaRepo() {
        listaPessoasFisicas = new ArrayList<>();
    }
```

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

```
    public void alterar(PessoaFisica pessoaFisica) {
        for (int i = 0; i < listaPessoasFisicas.size(); i++) {
            PessoaFisica pf = listaPessoasFisicas.get(i);
            if (pf.getId() == pessoaFisica.getId()) {
                listaPessoasFisicas.set(i, pessoaFisica);
                break;
            }
        }
    }
```

```
    public void excluir(int id) {
        for (int i = 0; i < listaPessoasFisicas.size(); i++) {
            PessoaFisica pf = listaPessoasFisicas.get(i);
            if (pf.getId() == id) {
                listaPessoasFisicas.remove(i);
                break;
            }
        }
    }
}
```

```

public PessoaFisica obter(int id) {
    for (PessoaFisica pf : listaPessoasFisicas) {
        if (pf.getId() == id) {
            return pf;
        }
    }
    return null;
}

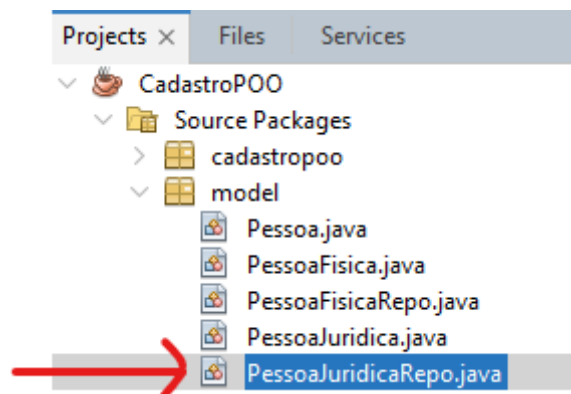
public List<PessoaFisica> obterTodos() {
    return listaPessoasFisicas;
}

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

public void recuperar(String nomeArquivo) throws IOException, ClassNotFoundException {
    try (ObjectInputStream inputStream = new ObjectInputStream(new
        FileInputStream(nomeArquivo))) { listaPessoasFisicas = (List<PessoaFisica>)
        inputStream.readObject(); }
}
}

```

b. Classe `PessoaJuridicaRepo`, com um `ArrayList` de `PessoaJuridica`, nível de acesso privado, e métodos públicos **inserir**, **alterar**, **excluir**, **obter** e **obterTodos**, para gerenciamento das entidades contidas no `ArrayList`.



package model;

```

import java.io.*;
import java.util.ArrayList;
import java.util.List;
import java.io.Serializable;

```

```

public class PessoaJuridicaRepo implements Serializable {
    // ArrayList para armazenar objetos PessoaJuridica
    private List<PessoaJuridica> listaPessoasJuridicas;
}

```

```

// Construtor da classe
public PessoaJuridicaRepo() {
    listaPessoasJuridicas = new ArrayList<>();
}
public void inserir(PessoaJuridica pessoaJuridica) {
    listaPessoasJuridicas.add(pessoaJuridica);
}
public void alterar(PessoaJuridica pessoaJuridica) {
    for (int i = 0; i < listaPessoasJuridicas.size(); i++) {
        PessoaJuridica pj = listaPessoasJuridicas.get(i);
        if (pj.getId() == pessoaJuridica.getId()) {
            listaPessoasJuridicas.set(i, pessoaJuridica);
            break;
        }
    }
}

public void excluir(int id) {
    for (int i = 0; i < listaPessoasJuridicas.size(); i++) {
        PessoaJuridica pj = listaPessoasJuridicas.get(i);
        if (pj.getId() == id) {
            listaPessoasJuridicas.remove(i);
            break;
        }
    }
}

public PessoaJuridica obter(int id) {
    for (PessoaJuridica pj : listaPessoasJuridicas) {
        if (pj.getId() == id) {
            return pj;
        }
    }
    return null;
}

public List<PessoaJuridica> obterTodos() {
    return listaPessoasJuridicas;
}

```

c. Em ambos os gerenciadores adicionar o método público **persistir**, com a recepção do nome do arquivo, para armazenagem dos dados no disco.

(Obs) Os métodos persistir e recuperar devem ecoar (throws) exceções:

```

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

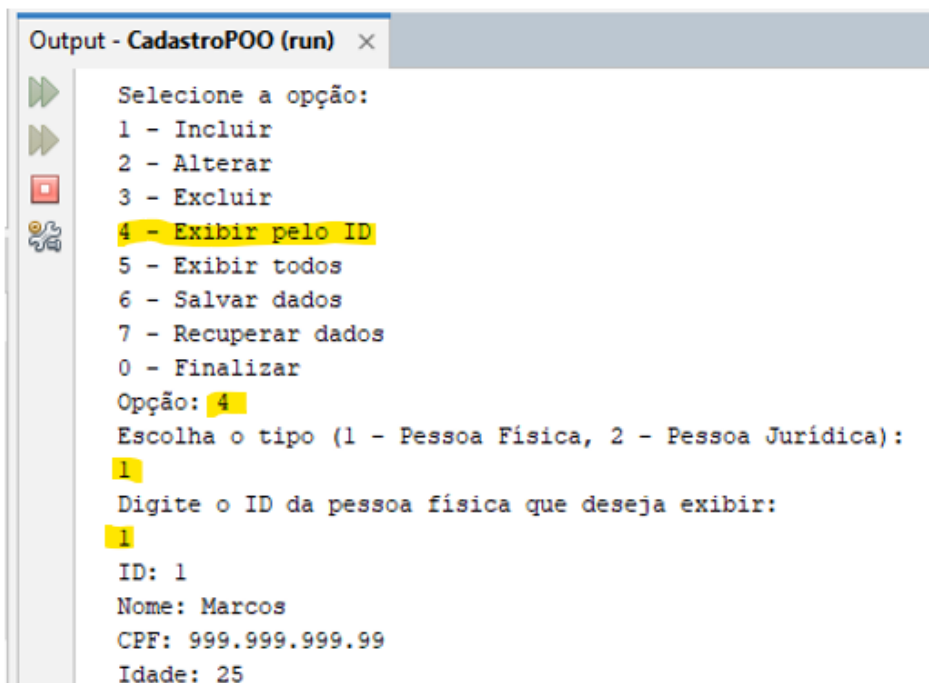
```


d. Em ambos os gerenciadores adicionar o método público **recuperar**, com a recepção do nome do arquivo, para recuperação dos dados do disco.

e(Obs) Os métodos **persistir** e **recuperar** devem **ecoar (throws) exceções**:

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

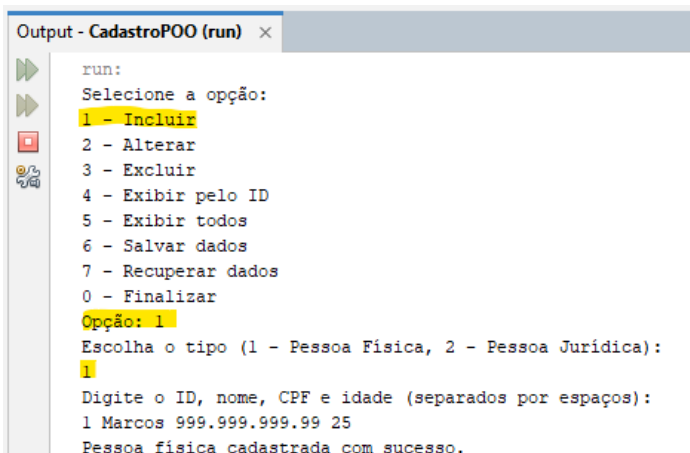
f. O método **obter** deve retornar uma entidade a partir do id.



```
Output - CadastroPOO (run) x
Selezione a opção:
1 - Incluir
2 - Alterar
3 - Excluir
4 - Exibir pelo ID
5 - Exibir todos
6 - Salvar dados
7 - Recuperar dados
0 - Finalizar
Opção: 4
Escolha o tipo (1 - Pessoa Física, 2 - Pessoa Jurídica):
1
Digite o ID da pessoa física que deseja exibir:
1
ID: 1
Nome: Marcos
CPF: 999.999.999.99
Idade: 25
```

g. Os métodos **inserir** e **alterar** devem ter entidades como parâmetros.

Incluir



```
Output - CadastroPOO (run) x
run:
Selezione a opção:
1 - Incluir
2 - Alterar
3 - Excluir
4 - Exibir pelo ID
5 - Exibir todos
6 - Salvar dados
7 - Recuperar dados
0 - Finalizar
Opção: 1
Escolha o tipo (1 - Pessoa Física, 2 - Pessoa Jurídica):
1
Digite o ID, nome, CPF e idade (separados por espaços):
1 Marcos 999.999.999.99 25
Pessoa física cadastrada com sucesso.
```

Alterar

```
Output - CadastroPOO (run) x
Selecione a opção:
1 - Incluir
2 - Alterar
3 - Excluir
4 - Exibir pelo ID
5 - Exibir todos
6 - Salvar dados
7 - Recuperar dados
0 - Finalizar
Opção: 2
Escolha o tipo (1 - Pessoa Física, 2 - Pessoa Jurídica):
1
Digite o ID da pessoa física que deseja alterar:
1
Dados atuais da pessoa física:
ID: 1
Nome: Marcos
CPF: 999.999.999.99
Idade: 25
Digite os novos dados (nome, CPF, idade):
Marcos2 999.999.999.53 26
Pessoa física alterada com sucesso.
```

h. O método excluir deve receber o id da entidade para exclusão

```
Output - CadastroPOO (run) x
Selecione a opção:
1 - Incluir
2 - Alterar
3 - Excluir
4 - Exibir pelo ID
5 - Exibir todos
6 - Salvar dados
7 - Recuperar dados
0 - Finalizar
Opção: 3
Escolha o tipo (1 - Pessoa Física, 2 - Pessoa Jurídica):
1
Digite o ID da pessoa física que deseja excluir:
2
Pessoa física excluída com sucesso.
```

- i. O método **obterTodos** deve retornar o conjunto completo de entidades.

Obter Todos – Exibir Todos - Pessoa Física

```
Output - CadastroPOO (run) x
Selezione a opção:
1 - Incluir
2 - Alterar
3 - Excluir
4 - Exibir pelo ID
5 - Exibir todos
6 - Salvar dados
7 - Recuperar dados
0 - Finalizar
Opção: 5
Escolha o tipo (1 - Pessoa Física, 2 - Pessoa Jurídica):
1
Pessoas Físicas:
ID: 1
Nome: Marcos
CPF: 123.456.789.00
Idade: 25
```

Obter Todos – Exibir Todos - Pessoa Jurídica

```
Output - CadastroPOO (run) x
Selezione a opção:
1 - Incluir
2 - Alterar
3 - Excluir
4 - Exibir pelo ID
5 - Exibir todos
6 - Salvar dados
7 - Recuperar dados
0 - Finalizar
Opção: 5
Escolha o tipo (1 - Pessoa Física, 2 - Pessoa Jurídica):
2
Pessoas Jurídicas:
ID: 1
Nome: Francisca
CNPJ: 42.560.747.0001/57
```

5. Alterar o método “main” da classe principal para testar os repositórios:

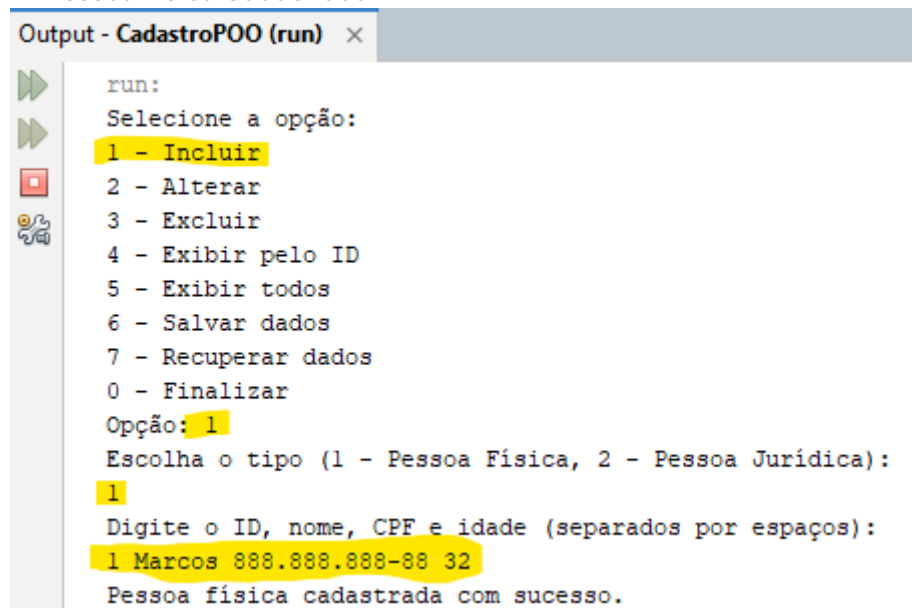
a) Instanciar um repositório de pessoas físicas (repo1).

```
public class CadastroPOO {  
  
    private static final Scanner scanner = new Scanner(System.in);  
    private static final PessoaFisicaRepo repoPessoaFisica = new PessoaFisicaRepo();  
    private static final PessoaJuridicaRepo repoPessoaJuridica = new PessoaJuridicaRepo();  
  
    /**
```

b) Adicionar duas pessoas físicas, utilizando o construtor completo.

```
private static void incluirPessoaFisica() {  
    System.out.println("Digite o ID, nome, CPF e idade (separados por espaços):");  
    int id = scanner.nextInt();  
    String nome = scanner.next();  
    String cpf = scanner.next();  
    int idade = scanner.nextInt();  
    repoPessoaFisica.inserir(new PessoaFisica(id, nome, cpf, idade));  
    System.out.println("Pessoa física cadastrada com sucesso.");  
}
```

1ª Pessoa Física Cadastrada.



Output - CadastroPOO (run) x

```
run:  
Selecione a opção:  
1 - Incluir  
2 - Alterar  
3 - Excluir  
4 - Exibir pelo ID  
5 - Exibir todos  
6 - Salvar dados  
7 - Recuperar dados  
0 - Finalizar  
Opção: 1  
Escolha o tipo (1 - Pessoa Física, 2 - Pessoa Jurídica):  
1  
Digite o ID, nome, CPF e idade (separados por espaços):  
1 Marcos 888.888.888-88 32  
Pessoa física cadastrada com sucesso.
```

2º Pessoa Física Cadastrada.

```
Output - CadastroPOO (run) x
Selezione a opção:
1 - Incluir
2 - Alterar
3 - Excluir
4 - Exibir pelo ID
5 - Exibir todos
6 - Salvar dados
7 - Recuperar dados
0 - Finalizar
Opção: 1
Escolha o tipo (1 - Pessoa Física, 2 - Pessoa Jurídica):
1
Digite o ID, nome, CPF e idade (separados por espaços):
2 Maria 999.999.999-99 22
Pessoa fisica cadastrada com sucesso.
```

- c) Invocar o método de persistência em repo1, fornecendo um nome de arquivo fixo, através do código.

```
Output - CadastroPOO (run) x
run:
Selezione a opção:
1 - Incluir
2 - Alterar
3 - Excluir
4 - Exibir pelo ID
5 - Exibir todos
6 - Salvar dados
7 - Recuperar dados
0 - Finalizar
Opção: 7
Digite o prefixo dos arquivos para recuperar:
repo1
Dados recuperados com sucesso!
```

- d) Instanciar outro repositório de pessoas físicas (repo2).

```
Selezione a opção:
1 - Incluir
2 - Alterar
3 - Excluir
4 - Exibir pelo ID
5 - Exibir todos
6 - Salvar dados
7 - Recuperar dados
0 - Finalizar
Opção: 6
Digite o prefixo dos arquivos para salvar:
repo2
Dados salvos com sucesso.
```

- e) Invocar o método de recuperação em repo2, fornecendo o mesmo nome de arquivo utilizado anteriormente.

```
Selecione a opção:
1 - Incluir
2 - Alterar
3 - Excluir
4 - Exibir pelo ID
5 - Exibir todos
6 - Salvar dados
7 - Recuperar dados
0 - Finalizar
Opção: 7
Digite o prefixo dos arquivos para recuperar:
repo2
Dados recuperados com sucesso!
```

- f) Exibir os dados de todas as pessoas físicas recuperadas.

```
Output - CadastroPOO (run) x
Selecione a opção:
1 - Incluir
2 - Alterar
3 - Excluir
4 - Exibir pelo ID
5 - Exibir todos
6 - Salvar dados
7 - Recuperar dados
0 - Finalizar
Opção: 5
Escolha o tipo (1 - Pessoa Física, 2 - Pessoa Jurídica):
1
Pessoas Físicas:
ID: 1
Nome: Marcos
CPF: 888.888.888-88
Idade: 32

ID: 2
Nome: Maria
CPF: 999.999.999-99
Idade: 22
```

g) Instanciar um repositório de pessoas jurídicas (repo3).

```
Selecione a opção:
1 - Incluir
2 - Alterar
3 - Excluir
4 - Exibir pelo ID
5 - Exibir todos
6 - Salvar dados
7 - Recuperar dados
0 - Finalizar
Opção: 6
Digite o prefixo dos arquivos para salvar:
repo3
Dados salvos com sucesso.
```

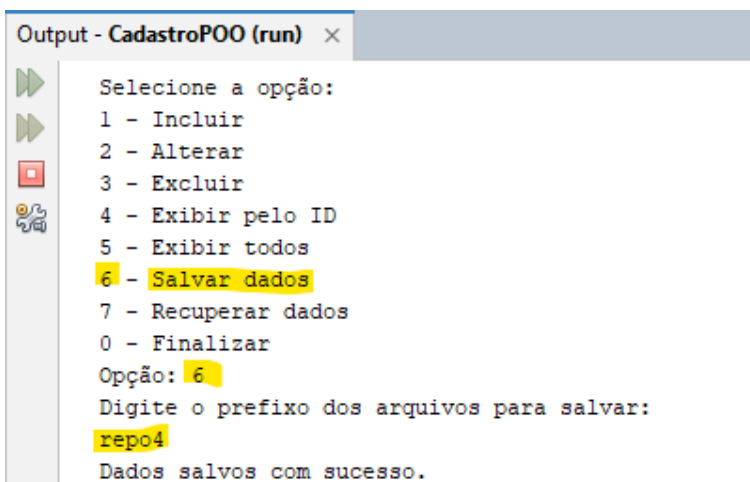
h) Adicionar duas pessoas jurídicas, utilizando o construtor completo.

```
Output - CadastroPOO (run) x
Selecione a opção:
1 - Incluir
2 - Alterar
3 - Excluir
4 - Exibir pelo ID
5 - Exibir todos
6 - Salvar dados
7 - Recuperar dados
0 - Finalizar
Opção: 1
Escolha o tipo (1 - Pessoa Física, 2 - Pessoa Jurídica):
2
Digite o ID, nome e CNPJ (separados por espaços):
1 GDTECH 48.456.748/0001-48
Pessoa jurídica cadastrada com sucesso.
Selecione a opção:
1 - Incluir
2 - Alterar
3 - Excluir
4 - Exibir pelo ID
5 - Exibir todos
6 - Salvar dados
7 - Recuperar dados
0 - Finalizar
Opção: 1
Escolha o tipo (1 - Pessoa Física, 2 - Pessoa Jurídica):
2
Digite o ID, nome e CNPJ (separados por espaços):
2 COMPUTEC 42.545.999/0001-01
Pessoa jurídica cadastrada com sucesso.
```

- i) Invocar o método de persistência em repo3, fornecendo um nome de arquivo fixo, através do código.

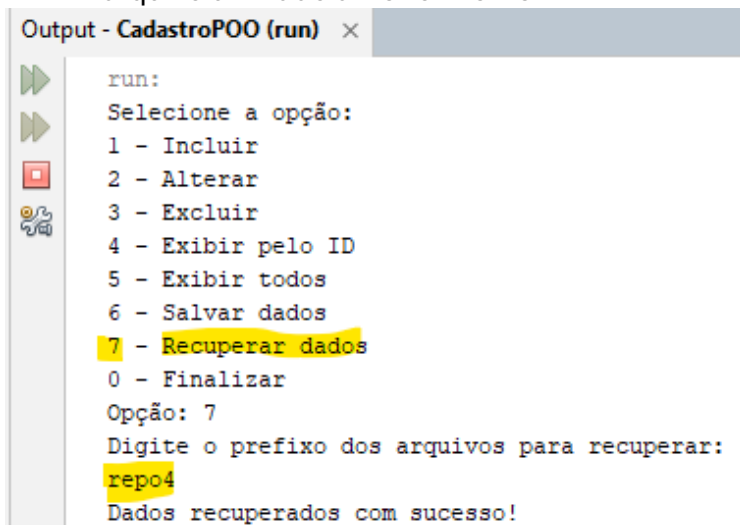
```
Selecione a opção:
1 - Incluir
2 - Alterar
3 - Excluir
4 - Exibir pelo ID
5 - Exibir todos
6 - Salvar dados
7 - Recuperar dados
0 - Finalizar
Opção: 6
Digite o prefixo dos arquivos para salvar:
repo3
Dados salvos com sucesso.
```

- j) Instanciar outro repositório de pessoas jurídicas (repo4).



```
Output - CadastroPOO (run) x
Selecione a opção:
1 - Incluir
2 - Alterar
3 - Excluir
4 - Exibir pelo ID
5 - Exibir todos
6 - Salvar dados
7 - Recuperar dados
0 - Finalizar
Opção: 6
Digite o prefixo dos arquivos para salvar:
repo4
Dados salvos com sucesso.
```

- k) Invocar o método de recuperação em repo4, fornecendo o mesmo nome de arquivo utilizado anteriormente.



```
Output - CadastroPOO (run) x
run:
Selecione a opção:
1 - Incluir
2 - Alterar
3 - Excluir
4 - Exibir pelo ID
5 - Exibir todos
6 - Salvar dados
7 - Recuperar dados
0 - Finalizar
Opção: 7
Digite o prefixo dos arquivos para recuperar:
repo4
Dados recuperados com sucesso!
```


l) Exibir os dados de todas as pessoas jurídicas recuperadas.

```
Output - CadastroPOO (run) x
Selecione a opção:
1 - Incluir
2 - Alterar
3 - Excluir
4 - Exibir pelo ID
5 - Exibir todos
6 - Salvar dados
7 - Recuperar dados
0 - Finalizar
Opção: 5
Escolha o tipo (1 - Pessoa Física, 2 - Pessoa Jurídica):
2
Pessoas Jurídicas:
ID: 1
Nome: CAMABRAS
CNPJ: 64.444.471/0001-71

ID: 2
Nome: CASAS-BAHIA
CNPJ: 54.741.244/0001-54
```

RELATÓRIO DISCENTE DE ACOMPANHAMENTO – Parte 1

ANÁLISE E CONCLUSÃO:

a) Quais as vantagens e desvantagens do uso de herança?

A herança em programação é um conceito fundamental, especialmente em linguagens orientadas a objetos. Abaixo segue com mais detalhes sobre as vantagens e desvantagens do uso de herança:

Vantagens:

1. **Reutilização de código:** Com a herança, você pode criar classes que herdam atributos e métodos de outras classes, permitindo reutilização de código. Isso economiza tempo e esforço, pois você não precisa reescrever o mesmo código várias vezes.
2. **Organização hierárquica:** A herança permite organizar as classes em uma hierarquia, onde classes mais específicas herdam características de classes mais genéricas. Isso torna o código mais fácil de entender e manter.
3. **Polimorfismo:** O polimorfismo é a capacidade de tratar objetos de diferentes classes de maneira uniforme. Com a herança, você pode criar métodos em classes base que podem ser substituídos por métodos nas classes derivadas, permitindo o polimorfismo.
4. **Flexibilidade e extensibilidade:** A herança permite estender e modificar o comportamento de classes existentes sem modificar seu código original. Isso torna o código mais flexível e adaptável a mudanças futuras.

Desvantagens:

1. **Acoplamento forte:** A herança pode levar a um acoplamento forte entre as classes, o que significa que uma mudança em uma classe base pode ter um grande impacto em todas as classes derivadas. Isso pode tornar o código mais difícil de manter e modificar.
2. **Hierarquias complicadas:** Às vezes, hierarquias de herança complexas podem surgir, tornando o código difícil de entender e debugar. Uma má hierarquia de herança pode levar a problemas de design conhecidos como "herança excessiva" ou "herança múltipla".
3. **Herança múltipla:** Algumas linguagens de programação suportam herança múltipla, onde uma classe pode herdar de múltiplas classes. Isso pode levar a problemas de ambiguidade e complexidade, e muitas vezes é evitado em favor de outras abordagens, como interfaces ou composição.
4. **Perda de encapsulamento:** Se não for usado corretamente, o uso de herança pode violar o princípio de encapsulamento, expondo detalhes de implementação desnecessários e tornando o código mais frágil.

Em resumo, a herança é uma ferramenta poderosa em programação, mas deve ser usada com cuidado para evitar problemas de design e manutenção.

b) Por que a interface `Serializable` é necessária ao efetuar persistência em arquivos binários?

A interface `Serializable` em Java é necessária ao efetuar persistência em arquivos binários por alguns motivos:

1. Serialização e desserialização: A interface `Serializable` é usada para indicar que uma classe pode ser serializada, ou seja, pode ser convertida em uma sequência de bytes para ser gravada em um arquivo ou transmitida pela rede. Isso é essencial ao realizar a persistência de objetos em arquivos binários, pois permite que os objetos sejam gravados em disco em um formato que possa ser recuperado posteriormente.

2. Persistência de estado de objetos: Ao gravar objetos em arquivos binários, geralmente estamos interessados em persistir seu estado, ou seja, os valores de seus atributos. A serialização permite que todo o estado do objeto seja gravado em disco, incluindo seus atributos e a estrutura de seus objetos internos.

3. Facilidade de uso: A serialização em Java é uma maneira conveniente de realizar a persistência de objetos em arquivos binários. Ao implementar a interface `Serializable`, a maioria das tarefas relacionadas à serialização é tratada automaticamente pela máquina virtual Java (JVM), simplificando o processo para os desenvolvedores.

4. Compatibilidade entre plataformas: A serialização em Java é independente de plataforma, o que significa que objetos serializados podem ser lidos e gravados em qualquer máquina virtual Java, independentemente do sistema operacional ou arquitetura de hardware. Isso facilita a interoperabilidade entre diferentes sistemas e ambientes.

No entanto, é importante observar que a serialização em Java tem algumas limitações e considerações de segurança que os desenvolvedores precisam estar cientes. Por exemplo, a serialização pode resultar em vulnerabilidades de segurança se os dados serializados forem manipulados por partes não confiáveis. Além disso, nem todos os tipos de objetos podem ser facilmente serializados em Java; por exemplo, objetos que mantêm referências a recursos externos podem encontrar problemas ao serem serializados.

c) Como o paradigma funcional é utilizado pela API stream no Java?

Na API Stream do Java, o paradigma funcional é amplamente utilizado para realizar operações de processamento de dados de forma declarativa e concisa. Abaixo segue algumas maneiras pelas quais o paradigma funcional é aplicado na API Stream:

1. Operações de alto nível: A API Stream fornece operações de alto nível, como `map`, `filter`, `reduce`, `collect`, entre outras, que permitem realizar transformações e operações em coleções de elementos de forma funcional. Por exemplo, o método `map` permite aplicar uma função a cada elemento da stream, enquanto `filter` permite filtrar os elementos com base em um predicado.

2. Lambda expressions ou expressões lambdas: O uso de expressões lambda é fundamental na API Stream. Expressões lambda permitem passar comportamentos como argumentos para os métodos da API Stream de forma concisa e expressiva. Por exemplo, podemos usar uma expressão lambda para definir a função de mapeamento em um método “map”.

3. Pipeline de operações: A API Stream permite encadear várias operações em um pipeline de operações. Isso permite a construção de fluxos de processamento de dados de forma modular e legível. Cada operação no pipeline é geralmente uma função que opera em um ou mais elementos da stream.

4. Operações sem efeito colateral: As operações na API Stream são projetadas para serem sem efeito colateral, o que significa que elas não modificam a fonte de dados original. Em vez disso, elas produzem um novo fluxo de dados com os resultados das operações aplicadas. Isso está alinhado com os princípios do paradigma funcional, onde as funções são puras e não têm efeitos colaterais.

5. Imutabilidade: Embora não seja estritamente aplicada pela API Stream, a ideia de imutabilidade é promovida. A maioria das operações na API Stream produz um novo fluxo de dados em vez de modificar a fonte de dados original, o que é consistente com o princípio de imutabilidade.

Em resumo, a API Stream do Java utiliza fortemente o paradigma funcional para fornecer uma maneira poderosa e expressiva de processar e manipular coleções de dados de forma eficiente e concisa. Isso torna o código mais legível, modular e menos propenso a erros.

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

Ao lidar com a persistência de dados em arquivos binários em Java, um padrão comum de desenvolvimento é o padrão de projeto DAO (Data Access Object), assim como na persistência em arquivos em geral. No entanto, a implementação específica pode variar dependendo dos requisitos e das nuances do sistema.

Logo abaixo segue uma visão geral de como o padrão DAO pode ser aplicado na persistência de dados em arquivos binários de texto em Java:

1. Interface DAO: Define uma interface que especifica os métodos de acesso aos dados, como inserção, atualização, exclusão e recuperação. Por exemplo, um DAO para manipular dados em arquivos de texto pode ter métodos como “save”, “update”, “delete”, “findAll”, entre outros.

2. Implementação do DAO: Fornece uma implementação concreta da interface DAO. Esta implementação contém a lógica para interagir com os arquivos de texto binários, realizando operações como leitura, gravação, atualização e exclusão de dados. Isso pode envolver o uso de classes Java para manipulação de arquivos, como `FileInputStream`, `FileOutputStream`, `BufferedReader`, `BufferedWriter`, entre outros. A diferença principal é que os dados são tratados como bytes em vez de texto puro.

Como exemplo foi usado na classe *PessoaFisicaRepo* nos métodos *Persistir()* e *Recuperar()*:

```
public void persistir(String nomeArquivo) throws IOException {
    try (ObjectOutputStream outputStream = new ObjectOutputStream(new FileOutputStream(nomeArquivo))) {
        outputStream.writeObject(listaPessoasFisicas);
    }
}

@SuppressWarnings("unchecked")
public void recuperar(String nomeArquivo) throws IOException, ClassNotFoundException {
    try (ObjectInputStream inputStream = new ObjectInputStream(new FileInputStream(nomeArquivo))) {
        listaPessoasFisicas = (List<PessoaFisica>) inputStream.readObject();
    }
}
```

3. Modelo de dados: Define as classes de modelo que representam os dados a serem armazenados e manipulados. Ao lidar com arquivos binários de texto, é importante converter os dados para e de bytes conforme necessário. Por exemplo, pode-se usar classes Java como `ObjectOutputStream` e `ObjectInputStream` para serializar objetos Java em bytes e gravá-los em arquivos de texto binários.

Como exemplo foi usado na classe *PessoaFisicaRepo* no método *Recuperar()*

```
@SuppressWarnings("unchecked")
public void recuperar(String nomeArquivo) throws IOException, ClassNotFoundException {
    try (ObjectInputStream inputStream = new ObjectInputStream(new FileInputStream(nomeArquivo))) {
        listaPessoasFisicas = (List<PessoaFisica>) inputStream.readObject();
    }
}
```

4. Gerenciamento de recursos: Assim como na persistência de arquivos em geral, é importante garantir que os recursos sejam abertos e fechados adequadamente ao manipular arquivos binários de texto. O padrão DAO pode encapsular a lógica para abrir e fechar os recursos necessários para acessar os arquivos de forma segura.

5. Testabilidade: O padrão DAO facilita a testabilidade do código, permitindo que a lógica de acesso aos dados seja isolada em sua própria camada. Isso simplifica a escrita de testes unitários e a realização de testes de integração, pois os testes podem se concentrar apenas na lógica de negócios, sem se preocupar com os detalhes de acesso aos dados.

Em resumo, ao aplicar o padrão DAO na persistência de dados em arquivos binários de texto em Java, é possível criar um código mais modular, organizado e fácil de manter, seguindo boas práticas de desenvolvimento de software.

RELATÓRIO DISCENTE DE ACOMPANHAMENTO – Parte 2

ANÁLISE E CONCLUSÃO:

a) O que são elementos estáticos e qual o motivo para o método “*main*” adotar esse modificador?

Em Java, elementos estáticos são aqueles associados à classe em si, em vez de instâncias individuais da classe. Isso significa que eles são compartilhados por todas as instâncias da classe e podem ser acessados sem a necessidade de criar um objeto da classe.

Os elementos estáticos em Java incluem variáveis estáticas (também conhecidas como variáveis de classe) e métodos estáticos (também conhecidos como métodos de classe). Eles são definidos usando a palavra-chave “static”.

A razão pela qual o método “main” é estático em Java é para permitir que ele seja chamado pelo ambiente de execução Java (JVM) sem a necessidade de criar uma instância da classe que contém o método “main”. Quando você inicia um programa Java, a JVM procura o método `main` na classe especificada como ponto de entrada e o invoca diretamente, sem precisar criar um objeto dessa classe. Isso facilita a inicialização do programa e é uma convenção de design em Java.

b) Para que serve a classe Scanner?

A classe Scanner em Java é usada para ler entradas de dados a partir de várias fontes, como o teclado (System.in), arquivos ou strings. Ela fornece métodos para ler diferentes tipos de dados, como inteiros, ponto flutuantes, booleanos, entre outros, de forma fácil e conveniente.

A classe Scanner é muito útil para interações com o usuário e para processar entradas de dados em programas Java.

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

O uso de classes de repositório em Java pode ter um grande impacto na organização do código, especialmente em projetos que seguem o padrão de arquitetura de software conhecido como “Arquitetura em Camadas” (Layered Architecture) ou “Arquitetura de Três Camadas” (Three-Tier Architecture).

As classes de repositório são frequentemente usadas para abstrair o acesso aos dados do banco de dados em uma camada separada da lógica de negócios da aplicação. Isso promove uma separação de preocupações e torna o código mais modular e fácil de manter. Segue abaixo algumas maneiras pelas quais o uso de classes de repositório pode impactar a organização do código em Java:

1. Separação de preocupações: *As classes de repositório encapsulam toda a lógica de acesso aos dados, isolando-a da lógica de negócios da aplicação. Isso permite que as preocupações relacionadas aos dados sejam tratadas separadamente das preocupações relacionadas à lógica de negócios, resultando em um código mais organizado e coeso.*

2. Facilidade de manutenção: Ao centralizar o acesso aos dados em classes de repositório, as mudanças na lógica de acesso aos dados podem ser feitas em um local centralizado, em vez de serem distribuídas por toda a aplicação. Isso torna mais fácil manter e modificar o código relacionado ao acesso aos dados.

3. Reutilização de código: As classes de repositório podem ser reutilizadas em toda a aplicação para acessar diferentes entidades de dados. Isso promove a reutilização de código e evita a duplicação de lógica de acesso aos dados em vários lugares da aplicação.

4. Testabilidade: O código que utiliza classes de repositório tende a ser mais fácil de testar, pois a lógica de acesso aos dados pode ser isolada e testada separadamente da lógica de negócios. Isso permite a criação de testes unitários mais eficazes e facilita a adoção de práticas de desenvolvimento orientado a testes (TDD).

Em resumo, o uso de classes de repositório pode impactar positivamente a organização do código em Java, promovendo uma separação de preocupações, facilitando a manutenção, promovendo a reutilização de código e aumentando a testabilidade da aplicação.