



Estácio

Estácio - Unidade São Pedro

RJ 140 Km 2, 512 loja 1, São Pedro da Aldeia - RJ, 28941-182

Desenvolvimento Full Stack

Classe: Missão Prática | Nível 1 | Mundo 3

3º Semestre

Marvin de Almeida Costa

Título da Prática: Material de orientações para desenvolvimento da missão prática do 1º nível de conhecimento.

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.
5. 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:

- **CadastroPOO.java**

```
package cadastropoo;
```

```
/**
```

```
 *
```

```
 * @author Marvin
```

```
 */
```

```
import java.io.IOException;
```

```

public class CadastroPOO {

    public static void main(String[] args) {
        PessoaFisicaRepo repo1 = new PessoaFisicaRepo();
        PessoaFisica pessoa1 = new PessoaFisica(1, "Ana", "11111111111", 25);
        PessoaFisica pessoa2 = new PessoaFisica(2, "Carlos", "22222222222", 52);

        repo1.inserir(pessoa1);
        repo1.inserir(pessoa2);
        System.out.println("Dados de Pessoa Fisica Armazenados.");

        try {
            repo1.persistir("pessoasFisicas.dat");
        } catch (IOException e) {
            e.printStackTrace();
        }

        PessoaFisicaRepo repo2 = new PessoaFisicaRepo();
        try {
            repo2.recuperar("pessoasFisicas.dat");
            System.out.println("Dados de Pessoa Fisica Recuperados.");
        } catch (IOException | ClassNotFoundException e) {
            e.printStackTrace();
        }
        for (PessoaFisica pf : repo2.obterTodos()) {
            pf.exibir();
        }

        PessoaJuridicaRepo repo3 = new PessoaJuridicaRepo();
        PessoaJuridica pessoa3 = new PessoaJuridica(3, "XPTO Sales", "33333333333");
        PessoaJuridica pessoa4 = new PessoaJuridica(4, "XPTO Solutions", "44444444444");

        repo3.inserir(pessoa3);
        repo3.inserir(pessoa4);
        System.out.println("Dados de Pessoa Juridica Armazenados.");

        try {
            repo3.persistir("pessoasJuridicas.dat");
        } catch (IOException e) {
            e.printStackTrace();
        }

        PessoaJuridicaRepo repo4 = new PessoaJuridicaRepo();
        try {

```

```

        repo4.recuperar("pessoasJuridicas.dat");
        System.out.println("Dados de Pessoa Juridica Recuperados.");
    } catch (IOException | ClassNotFoundException e) {
        e.printStackTrace();
    }
    for (PessoaJuridica pf : repo4.obterTodos()) {
        pf.exibir();
    }
}
}

```

- **Pessoa.java**

```

package cadastropoo;

import java.io.Serializable;

/**
 *
 * @author Marvin
 */

public class Pessoa implements Serializable {
    private int id;
    private String nome;

    public Pessoa() {}

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

    public void exibir() {
        System.out.println("Id: " + id);
        System.out.println("Nome: " + nome);
    }

    // Getters e Setters
    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;
    }
}

```

- **PessoaFisica.java**

```

package cadastrapoo;

/**
 *
 * @author Marvin
 */
import java.io.Serializable;

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

    public PessoaFisica() {
        super();
    }

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

    @Override
    public void exibir() {
        super.exibir();
        System.out.println("CPF: " + cpf);
        System.out.println("Idade: " + idade);
    }
}

```

```
}
```

- **PessoaFisicaRepo.java**

```
package cadastrapoo;

/**
 *
 * @author Marvin
 */
import java.io.*;
import java.util.ArrayList;
import java.util.List;

public class PessoaFisicaRepo {
    private List<PessoaFisica> pessoasFisicas = new ArrayList<>();

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

    public void alterar(PessoaFisica pessoaFisica) {

    }

    public void excluir(int id) {

    }

    public PessoaFisica obter(int id) {

        return null;
    }

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

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

```

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

```

- **PessoaJuridica.java**

```

package cadastropoo;

/**
 *
 * @author Marvin
 */
import java.io.Serializable;

public class PessoaJuridica extends Pessoa implements Serializable {
    private String cnpj;

    public PessoaJuridica() {
        super();
    }

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

    @Override
    public void exibir() {
        super.exibir();
        System.out.println("CNPJ: " + cnpj);
    }
}

```

- **PessoaJuridicaRepo.java**

```

package cadastropoo;

/**

```

```

*
* @author Marvin
*/
import java.io.*;
import java.util.ArrayList;
import java.util.List;

public class PessoaJuridicaRepo {
    private List<PessoaJuridica> pessoasJuridicas = new ArrayList<>();

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

    public void alterar(PessoaJuridica pessoaJuridica) {

    }

    public void excluir(int id) {

    }

    public PessoaJuridica obter(int id) {

        return null;
    }

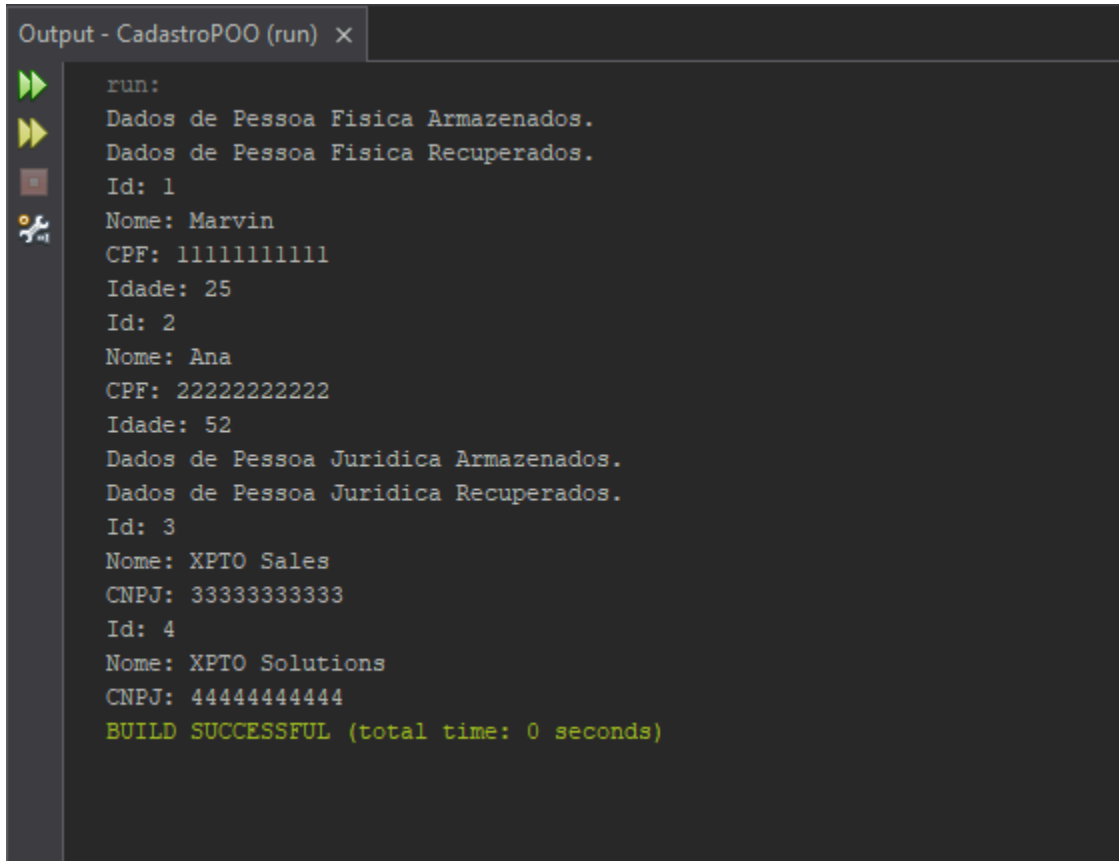
    public List<PessoaJuridica> obterTodos() {
        return new ArrayList<>(pessoasJuridicas);
    }

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

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

```

Os resultados da execução dos códigos também devem ser apresentados;



```
run:
Dados de Pessoa Fisica Armazenados.
Dados de Pessoa Fisica Recuperados.
Id: 1
Nome: Marvin
CPF: 111111111111
Idade: 25
Id: 2
Nome: Ana
CPF: 222222222222
Idade: 52
Dados de Pessoa Juridica Armazenados.
Dados de Pessoa Juridica Recuperados.
Id: 3
Nome: XPTO Sales
CNPJ: 333333333333
Id: 4
Nome: XPTO Solutions
CNPJ: 444444444444
BUILD SUCCESSFUL (total time: 0 seconds)
```

Análise e Conclusão:

Quais as vantagens e desvantagens do uso de herança?

Vantagens da herança

- Reutilização de código: permite que as classes herdem atributos e métodos de uma classe base, evitando assim a duplicação de código.
- Extensibilidade: facilita a extensão de uma classe existente para criar novas classes que herdam todos os recursos da classe base e, em seguida, adicionam ou modificam seus próprios recursos.
- Coerência e consistência: agrupa recursos comuns em uma classe base, garantindo que todas as classes derivadas mantenham uma estrutura e um comportamento coerentes.

- Abstração e generalização: identifica características comuns em classes relacionadas, abstrai essas características em uma classe base que captura a essência comum dessas classes, incentivando a generalização.

-

Desvantagens da herança

Embora a herança tenha muitas vantagens, ela também tem algumas desvantagens que devem ser consideradas:

- Acoplamento rígido: as classes filhas são fortemente acopladas às suas classes pai, o que pode limitar a flexibilidade do projeto e dificultar a substituição das classes pai por outras que não sejam diretamente compatíveis.
- Problemas de design: se a classe base for alterada com frequência, isso pode exigir alterações significativas em todas as classes derivadas, o que pode levar a problemas de manutenção e escalabilidade.
- Risco de quebra do princípio de substituição de Liskov: se as classes derivadas introduzirem um comportamento que quebre as expectativas definidas pela classe base, isso pode levar a erros de tempo de execução.
- Complexidade adicional: a herança introduz complexidade adicional no design do sistema, especialmente quando vários níveis de herança são usados, o que pode dificultar a compreensão e a manutenção do código.

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

A necessidade da interface "Serializable" ao persistir arquivos binários em Java se deve a vários fatores importantes relacionados à serialização e ao armazenamento de objetos:

- Conversão de objeto em fluxo de bytes: a serialização permite que o estado de um objeto seja convertido em um fluxo de bytes ("byte stream"). Esse processo é essencial para armazenar o estado de um objeto em um arquivo ou transmiti-lo por uma rede. Depois de serializado, o objeto pode ser salvo em um arquivo e, posteriormente, recuperado por um processo chamado desserialização, que reconstrói o objeto a partir do fluxo de bytes.
- Independência de plataforma: A serialização em Java cria um fluxo de bytes que é independente de plataforma. Isso significa que um objeto serializado em uma plataforma específica pode ser adequadamente desserializado em qualquer outra plataforma compatível com Java, facilitando a portabilidade de dados entre diferentes sistemas operacionais e arquiteturas.

- Uso de interfaces de marcador: a interface "Serializable" é uma interface de marcador, o que significa que não contém métodos a serem implementados. Sua única finalidade é informar à máquina virtual Java (JVM) que o objeto é elegível para serialização. Ao implementar essa interface, estamos informando à JVM que queremos que nosso objeto possa ser convertido em um fluxo de bytes e, posteriormente, recriado.
- Persistência de dados: A serialização é comumente usada para manter o estado de um objeto. Depois que um objeto é serializado e gravado no disco, ele pode ser lido e desserializado posteriormente, reconstruindo o objeto original com seu estado intacto. Isso é fundamental para aplicativos que precisam salvar sessões de usuário, configurações ou outros dados dinâmicos entre execuções.
- Compatibilidade e segurança: a serialização Java inclui mecanismos para gerenciar a versão dos objetos serializados por meio do uso de "serialVersionUID". Esse identificador de versão ajuda a garantir que os objetos serializados sejam compatíveis entre diferentes versões de um aplicativo, evitando problemas de desserialização caso os objetos tenham sido alterados de forma incompatível. Além disso, somente os objetos que implementam a interface "Serializable" podem ser serializados, o que proporciona um nível de segurança ao evitar a serialização acidental de objetos confidenciais.

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

A API Java Stream, introduzida no Java SE 8, incorpora princípios do paradigma funcional, permitindo que os desenvolvedores processem sequências de elementos de forma concisa e expressiva. Veja a seguir os detalhes de como esse paradigma é integrado:

- "Laziness" (Preguiça).

A API de fluxo opera com base no princípio da "preguiça", em que as operações em um fluxo são executadas somente quando uma operação terminal é chamada. Isso permite o processamento eficiente de grandes conjuntos de dados sem computação desnecessária.

- Operações intermediárias e terminais

As operações em fluxos se enquadram em duas categorias: operações intermediárias e terminais. As operações intermediárias, como "filter()", "map()" e "sorted()", transformam os elementos de um fluxo e retornam um novo fluxo. As operações terminais, como "forEach()", "collect()" e "reduce()", acionam a execução de operações intermediárias e produzem um resultado final.

- Paralelismo

A API de streaming em Java integra-se ao conceito de paralelismo, permitindo que os desenvolvedores aproveitem os processadores de vários núcleos para processamento simultâneo. Ao invocar o método "parallel()" em um fluxo, as operações podem ser executadas em paralelo, melhorando o desempenho de tarefas computacionalmente intensas.

- Funções de ordem superior e expressões Lambda

A API Stream faz uso extensivo de funções de ordem superior e expressões lambda, que são fundamentais para a programação funcional. As funções de ordem superior são funções que podem receber outras funções como argumentos ou retorná-las como resultados. As expressões lambda em Java permitem que as funções anônimas sejam definidas de forma concisa, facilitando a criação de funções de ordem superior.

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

Ao trabalhar com Java e a necessidade de reter dados em arquivos, vários padrões e técnicas são adotados, dependendo do contexto e dos requisitos específicos do projeto. Algumas práticas comuns são destacadas aqui:

- Uso da classe "Properties" (Propriedades)

Uma maneira simples de armazenar dados permanentemente em Java é por meio do uso da classe "Properties". Essa classe permite armazenar pares de valores-chave em um arquivo ".properties", o que é especialmente útil para configurações ou dados simples, como PINs de usuários e números de contas.

- Serialização e arquivos XML

Para casos mais complexos, em que é necessário serializar objetos Java em um formato que possa ser armazenado em arquivos, é possível usar métodos padrão de serialização Java ou converter objetos em XML. A serialização permite que o estado de objetos complexos seja armazenado em um arquivo, enquanto o XML fornece uma representação textual que pode ser facilmente lida e modificada por humanos. No entanto, a serialização pode ser lenta e propensa a erros se as classes forem alteradas com frequência.

- Armazenamento no Android

Para aplicativos Android, o armazenamento de dados segue diferentes convenções, incluindo o uso de diretórios específicos para armazenar arquivos somente para o aplicativo ("getFilesDir()"), compartilhamento de arquivos com outros aplicativos ("getExternalFilesDir()"),

preferências compartilhadas para dados primitivos ("SharedPreferences") e bancos de dados privados usando a biblioteca Room para dados estruturados. Essas opções oferecem diferentes níveis de segurança, acessibilidade e eficiência, dependendo do tipo de dados e do uso pretendido.