UNIVERSIDADE ESTÁCIO DE SÁ

CAMPUS POLO CENTRO - MARICÁ - RJ

ALUNO: LUIZ CARLOS MARINHO JUNIOR

MATRÍCULA: 2023.11.17557-1

CURSO: DESENVOLVIMENTO FULL STACK

SEMESTRE LETIVO: 2024.4

DISCIPLINA: NÍVEL 1: INICIANDO O CAMINHO PELO JAVA

TÍTULO: 1º PROCEDIMENTO | CRIAÇÃO DAS ENTIDADES E SISTEMA DE PERSISTÊNCIA

RIO DE JANEIRO

2024

LUIZ CARLOS MARINHO JUNIOR

1º PROCEDIMENTO | CRIAÇÃO DAS ENTIDADES E SISTEMA DE PERSISTÊNCIA

Trabalho prático para aprovação na disciplina de Nível 1: Iniciando o Caminho Pelo Java.

Tutora: Prof. Maria B.

RIO DE JANEIRO

2024

SUMÁRIO

1. OBJETIVOS 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, terá implementado um sistema cadastral em Java, utilizando os recursos da programação orientada a objetos e a persistência em arquivos binários.

2. CÓDIGOS SOLICITADOS

2.1. APRESENTAÇÃO

Todos os códigos que serão aqui apresentados, estão disponíveis de forma completa no repositório do Github. O desenvolvimento do código está separado por branch, e no caso desse 1ª Procedimento, o código se encontra na branch PrimeiroProcedimento.

O **link para o repositório do Github** desse 1ª Procedimento se encontra logo abaixo: https://github.com/luizmarinhojr/cadastro-poo/tree/PrimeiroProcedimento.

2.2. PRINCIPAIS INSTRUÇÕES DO 1ª PROCEDIMENTO

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.

Implementando o procedimento:

```
public abstract class Pessoa {
```

```
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 Pessoa() {}

public abstract String exibir();

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;
}
```

```
    @Override
    public String toString() {
        return "Pessoa{" + "id=" + id + ", 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.

Implementando o procedimento:

```
public class PessoaFisica extends Pessoa {
    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 PessoaFisica() {}

    @Override
    public String exibir(){
        return """
                Id: %d
                Nome: %s
                CPF: %s
                Idade: %d
```

```
            """.formatted(getId(), getNome(), getCpf(), getIdade());
    }

    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 String toString() {
        return "\n" + "Id: " + getId() + "\n" +
                "Nome: " + getNome() + "\n" +
                "CPF: " + getCpf() + "\n" +
                "Idade: " + getIdade();
    }
}
```

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

Implementando o procedimento:

```
public class PessoaJuridica extends Pessoa {
   private static final long serialVersionUID = 1L;
   private String cnpj;

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

   public PessoaJuridica() {}

   @Override
   public String exibir() {
      return """
            Id: %d
            Nome: %s
            CNPJ: %s
            """.formatted(getId(), getNome(), getCnpj());
   }

   public String getCnpj() {
      return this.cnpj;
   }

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

```
    }

    @Override
    public String toString() {
        return "\n" + "Id: " + getId() + "\n" +
                "Nome: " + getNome() + "\n" +
                "CNPJ: " + getCnpj();
    }
}
```

d. Adicionar interface Serializable em todas as classes

Implementando o procedimento:

```
public abstract class Pessoa implements Serializable {
    ...
}
```

```
public class PessoaFisica extends Pessoa implements Serializable {
    ...
}
```

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

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.

Implementando o procedimento:

```
public class PessoaFisicaRepo {


   private List<PessoaFisica> pessoasFisicas;


   public PessoaFisicaRepo() {
      pessoasFisicas = new ArrayList();
   }


   public void inserir(PessoaFisica... pessoasFisicas) {
      for (PessoaFisica pessoaFisica: pessoasFisicas) {
         if (this.pessoasFisicas.stream().anyMatch((pessoa) -> pessoa.getId() ==
pessoaFisica.getId())) {
            System.out.println("**** A pessoa física %s não foi inserida, pois o ID %d
a que ela se refere já se encontra inserido no Array. ****"
                  .formatted(pessoaFisica.getNome(), pessoaFisica.getId()));
         } else {
            this.pessoasFisicas.add(pessoaFisica);
         }
      }
   }


   public void alterar(PessoaFisica pessoaFisicaAtual, PessoaFisica pessoaFisicaNova)
{
      pessoaFisicaAtual.setId(pessoaFisicaNova.getId());
      pessoaFisicaAtual.setNome(pessoaFisicaNova.getNome());
      pessoaFisicaAtual.setCpf(pessoaFisicaNova.getCpf());
      pessoaFisicaAtual.setIdade(pessoaFisicaNova.getIdade());
   }


   public void excluir(PessoaFisica pessoaFisica) {
```

```
    pessoasFisicas.remove(pessoaFisica);
    System.out.println("\n**** Pessoa Física excluída com sucesso ****");
}

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


public String obterTodos() {
    this.pessoasFisicas.sort((p1, p2) -> Integer.compare(p1.getId(), p2.getId()));
    String pessoasFisicas = "";
    if (!this.pessoasFisicas.isEmpty()) {
        for (PessoaFisica pessoaFisica: this.pessoasFisicas) {
            pessoasFisicas += pessoaFisica.exibir();
        }
    } else {
        pessoasFisicas += "\nNão há pessoas físicas inseridas\n";
    }
    return pessoasFisicas;
}

public void persistir(String prefixo) throws Exception {
    String nomeArquivo = prefixo + ".fisica.bin";
    FileOutputStream fos = new FileOutputStream(nomeArquivo);
    ObjectOutputStream ous = new ObjectOutputStream(fos);
    ous.writeObject(this.pessoasFisicas);
```

```
        this.pessoasFisicas.clear();
        System.out.println("Dados de Pessoa Física Armazenados.");
    }
}
```

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 .

Implementando o procedimento:

```
public class PessoaJuridicaRepo {

    private List<PessoaJuridica> pessoasJuridicas;

    public PessoaJuridicaRepo() {
        pessoasJuridicas = new ArrayList();
    }

    public void inserir(PessoaJuridica... pessoasJuridicas) {
        for (PessoaJuridica pessoaJuridica: pessoasJuridicas) {
            if (this.pessoasJuridicas.stream().anyMatch((pessoa) -> pessoa.getId() ==
pessoaJuridica.getId())) {
                System.out.println("**** A pessoa jurídica %s não foi inserida, pois o
ID %d a que ela se refere já se encontra inserido no Array. ****"
                        .formatted(pessoaJuridica.getNome(), pessoaJuridica.getId()));
            } else {
                this.pessoasJuridicas.add(pessoaJuridica);
            }
        }
    }
```

```
    public void alterar(PessoaJuridica pessoaJuridicaAtual, PessoaJuridica
pessoaJuridicaNova) {
        pessoaJuridicaAtual.setId(pessoaJuridicaNova.getId());
        pessoaJuridicaAtual.setNome(pessoaJuridicaNova.getNome());
        pessoaJuridicaAtual.setCnpj(pessoaJuridicaNova.getCnpj());
    }

    public void excluir(PessoaJuridica pessoaJuridica) {
        this.pessoasJuridicas.remove(pessoaJuridica);
        System.out.println("\n**** Pessoa Jurídica excluída com sucesso ****");
    }

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

    public String obterTodos() {
        this.pessoasJuridicas.sort((p1, p2) -> Integer.compare(p1.getId(), p2.getId()));
        String pessoasJuridicas = "";
        if (!this.pessoasJuridicas.isEmpty()) {
            for (PessoaJuridica pessoa: this.pessoasJuridicas) {
                pessoasJuridicas += pessoa.exibir();
            }
        } else {
            pessoasJuridicas += "\nNão há pessoas jurídicas inseridas\n";
        }
```

```
        return pessoasJuridicas;
    }
}
```

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

Implementando o procedimento:

```
public class PessoaFisicaRepo {
    ...
    public void persistir(String prefixo) throws Exception {
        String nomeArquivo = prefixo + ".fisica.bin";
        FileOutputStream fos = new FileOutputStream(nomeArquivo);
        ObjectOutputStream ous = new ObjectOutputStream(fos);
        ous.writeObject(this.pessoasFisicas);
        this.pessoasFisicas.clear();
        System.out.println("Dados de Pessoa Física Armazenados.");
    }
}
```

```
public class PessoaJuridicaRepo {
    ...
    public void persistir(String prefixo) throws Exception {
        String nomeArquivo = prefixo + ".juridica.bin";
        FileOutputStream fos = new FileOutputStream(nomeArquivo);
        ObjectOutputStream ous = new ObjectOutputStream(fos);
        ous.writeObject(this.pessoasJuridicas);
        this.pessoasJuridicas.clear();
```

```
        System.out.println("Dados de Pessoa Jurídica Armazenados.");
    }
}
```

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

Implementando o procedimento:

```
public class PessoaFisicaRepo {
    ...
    public void recuperar(String prefixo) throws Exception {
        String nomeArquivo = prefixo + ".fisica.bin";
        if (new File(nomeArquivo).exists()) { // Verifica se o arquivo existe no diretório
            FileInputStream fis = new FileInputStream(nomeArquivo);
            ObjectInputStream ois = new ObjectInputStream(fis);
            this.pessoasFisicas = (List<PessoaFisica>) ois.readObject();
            System.out.println("Dados de Pessoa Física Recuperados.");
        } else {
            System.out.println("**** ATENÇÃO: Dados não recuperados, pois não existe arquivo persistido com o prefixo " + prefixo + " ****");
        }
    }
}
```

```
public class PessoaJuridicaRepo {
    ...
    public void recuperar(String prefixo) throws Exception {
        String nomeArquivo = prefixo + ".juridica.bin";
        if (new File(nomeArquivo).exists()) {
            FileInputStream fis = new FileInputStream(nomeArquivo);
```

```
            ObjectInputStream ois = new ObjectInputStream(fis);
            this.pessoasJuridicas = (List<PessoaJuridica>) ois.readObject();
            System.out.println("Dados de Pessoa Jurídica Recuperados.");
        } else {
            System.out.println("**** ATENÇÃO: Dados não recuperados, pois não
existe arquivo persistido com o prefixo " + prefixo + " ****");
        }
    }
}
```

e. Os métodos persistir e recuperar devem ecoar (throws) exceções.

Implementando o procedimento:

```
public class PessoaFisicaRepo {
    ...
    public void persistir(String prefixo) throws Exception {
        ...
    }


    public void recuperar(String prefixo) throws Exception {
        ...
    }
}
```

```
public class PessoaJuridicaRepo {
    ...
    public void persistir(String prefixo) throws Exception {
        ...
```

```
    }

    public void recuperar(String prefixo) throws Exception {

        ...

    }
}
```

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

Implementando o procedimento:

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

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

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

Implementando o procedimento:

```
public void inserir(PessoaFisica... pessoasFisicas) {
    for (PessoaFisica pessoaFisica: pessoasFisicas) {
        if (this.pessoasFisicas.stream().anyMatch((pessoa) -> pessoa.getId() ==
pessoaFisica.getId())) {
```

```
            System.out.println("**** A pessoa física %s não foi inserida, pois o ID %d
a que ela se refere já se encontra inserido no Array. ****"
                    .formatted(pessoaFisica.getNome(), pessoaFisica.getId()));
        } else {
            this.pessoasFisicas.add(pessoaFisica);
        }
    }
}
```

```
    public void alterar(PessoaFisica pessoaFisicaAtual, PessoaFisica pessoaFisicaNova)
{
        pessoaFisicaAtual.setId(pessoaFisicaNova.getId());
        pessoaFisicaAtual.setNome(pessoaFisicaNova.getNome());
        pessoaFisicaAtual.setCpf(pessoaFisicaNova.getCpf());
        pessoaFisicaAtual.setIdade(pessoaFisicaNova.getIdade());
    }
```

```
    public void inserir(PessoaJuridica... pessoasJuridicas) {
        for (PessoaJuridica pessoaJuridica: pessoasJuridicas) {
            if (this.pessoasJuridicas.stream().anyMatch((pessoa) -> pessoa.getId() ==
pessoaJuridica.getId())) {
                System.out.println("**** A pessoa jurídica %s não foi inserida, pois o
ID %d a que ela se refere já se encontra inserido no Array. ****"
                        .formatted(pessoaJuridica.getNome(), pessoaJuridica.getId()));
            } else {
                this.pessoasJuridicas.add(pessoaJuridica);
            }
        }
    }
```

```
public void alterar(PessoaJuridica pessoaJuridicaAtual, PessoaJuridica
pessoaJuridicaNova) {
    pessoaJuridicaAtual.setId(pessoaJuridicaNova.getId());
    pessoaJuridicaAtual.setNome(pessoaJuridicaNova.getNome());
    pessoaJuridicaAtual.setCnpj(pessoaJuridicaNova.getCnpj());
}
```

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

Implementando o procedimento:

```
public void excluir(PessoaFisica pessoaFisica) {
    pessoasFisicas.remove(pessoaFisica);
    System.out.println("\n**** Pessoa Física excluída com sucesso ****");
}
```

```
public void excluir(PessoaJuridica pessoaJuridica) {
    this.pessoasJuridicas.remove(pessoaJuridica);
    System.out.println("\n**** Pessoa Jurídica excluída com sucesso ****");
}
```

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

Implementando o procedimento:

```
public String obterTodos() {
    this.pessoasFisicas.sort((p1, p2) -> Integer.compare(p1.getId(), p2.getId()));
    String pessoasFisicas = "";
    if (!this.pessoasFisicas.isEmpty()) {
        for (PessoaFisica pessoaFisica: this.pessoasFisicas) {
```

```
            pessoasFisicas += pessoaFisica.exibir();
        }
    } else {
        pessoasFisicas += "\nNão há pessoas físicas inseridas\n";
    }
    return pessoasFisicas;
}
```

```
public String obterTodos() {
    this.pessoasJuridicas.sort((p1, p2) -> Integer.compare(p1.getId(), p2.getId()));
    String pessoasJuridicas = "";
    if (!this.pessoasJuridicas.isEmpty()) {
        for (PessoaJuridica pessoa: this.pessoasJuridicas) {
            pessoasJuridicas += pessoa.exibir();
        }
    } else {
        pessoasJuridicas += "\nNão há pessoas jurídicas inseridas\n";
    }
    return pessoasJuridicas;
}
```

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

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

Implementando o procedimento:

```
PessoaFisicaRepo repo1 = new PessoaFisicaRepo();
```

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

Implementando o procedimento:

```
PessoaFisica pessoaFisica1 = new PessoaFisica(
        1, "Ana", "11111111111", 25
);
PessoaFisica pessoaFisica2 = new PessoaFisica(
        2, "Carlos", "22222222222", 52
);
repo1.inserir(pessoaFisica1, pessoaFisica2);
```

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

Implementando o procedimento:

```
try {
    repo1.persistir("controle");
} catch (Exception ex) {
    System.out.println("Houve um erro ao armazenar os dados.\n" + "Erro: " +
                       ex.getMessage());
}
```

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

Implementando o procedimento:

```
PessoaFisicaRepo repo2 = new PessoaFisicaRepo();
```

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

Implementando o procedimento:

```
try {
```

```
        repo2.recuperar("controle");
    } catch (Exception ex) {
        System.out.println("Houve um erro ao armazenar os dados.\n" + "Erro: " +
        ex.getMessage());
    }
```

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

Implementando o procedimento:

```
    System.out.print(repo2.obterTodos());
```

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

Implementando o procedimento:

```
    PessoaJuridicaRepo repo3 = new PessoaJuridicaRepo();
```

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

Implementando o procedimento:

```
    PessoaJuridica pessoaJuridica1 = new PessoaJuridica(
            3, "XPTO Sales", "33333333333333"
    );

    PessoaJuridica pessoaJuridica2 = new PessoaJuridica(
            4, "XPTO Solutions", "44444444444444"
    );

    repo3.inserir(pessoaJuridica1, pessoaJuridica2);
```

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

Implementando o procedimento:

```
try {
    repo3.persistir("financeiro");
} catch (Exception ex) {
    System.out.println("Houve um erro ao armazenar os dados.\n" + "Erro: " +
    ex.getMessage());
}
```

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

Implementando o procedimento:

```
PessoaJuridicaRepo repo4 = new PessoaJuridicaRepo();
```

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

Implementando o procedimento:

```
try {
    repo4.recuperar("financeiro");
} catch(Exception ex) {
    System.out.println("Houve um erro ao armazenar os dados.\n" + "Erro: " +
    ex.getMessage());
}
```

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

Implementando o procedimento:

```
System.out.print(repo4.obterTodos());
```

3. RESULTADO DA EXECUÇÃO DOS CÓDIGOS

```
Dados de Pessoa Física Armazenados.
Dados de Pessoa Física Recuperados.
Id: 1
Nome: Ana
CPF: 11111111111
Idade: 25
Id: 2
Nome: Carlos
CPF: 22222222222
Idade: 52
Dados de Pessoa Jurídica Armazenados.
Dados de Pessoa Jurídica Recuperados.
Id: 3
Nome: XPTO Sales
CNPJ: 33333333333333
Id: 4
Nome: XPTO Solutions
CNPJ: 44444444444444
BUILD SUCCESSFUL (total time: 0 seconds)
```

4. ANÁLISE E CONCLUSÃO

4.1. QUAIS AS VANTAGENS E DESVANTAGENS DO USO DE HERANÇA?

- **Vantagens da Herança:**
  1. Reutilização de código: Evita a duplicação de código, tornando o código mais conciso e de mais fácil manutenção.
  2. Hierarquia de classes: Organiza as classes em uma estrutura hierárquica, e dessa forma, facilita a compreensão do sistema e a identificação de relações entre as classes.
  3. Polimorfismo: Permite que objetos de classes distintas sejam tratados de forma polimórfica, aumentando a flexibilidade e a usabilidade do código.

4. Especialização: Permite criar classes mais específicas a partir de classes mais genéricas, adicionando novos comportamentos ou modificando comportamentos existentes.

- **Desvantagens da Herança:**

1. Acoplamento: Cria um forte acoplamento entre a superclasse e suas subclasses, tornando o código mais complicado de modificar. Possíveis alterações na superclasse podem afetar todas as subclasses.
2. Rigidez: A hierarquia de classes pode se tornar rígida e difícil de modificar, especialmente em sistemas complexos.
3. Fragilidade: Alterações na superclasse podem levar a erros inesperados nas subclasses, tornando o sistema mais frágil.
4. Complexidade: O uso excessivo de herança pode levar a hierarquias de classes complexas e difíceis de entender e manter.
5. Violação do princípio do Liskov Substitution: Se uma subclasse não puder ser substituída por sua superclasse sem que o comportamento do programa seja alterado, a herança está sendo usada incorretamente.

4.2. POR QUE A INTERFACE SERIALIZABLE É NECESSÁRIA AO EFETUAR PERSISTÊNCIA EM ARQUIVOS BINÁRIOS?

Porque a interface Serializable indica ao compilador que os objetos da classe que a implementa podem ser serializados. Quando uma classe implementa a interface Serializable, a JVM (Máquina Virtual Java) utiliza um mecanismo interno para percorrer o objeto e seus atributos, transformando cada valor em uma representação binária, e dessa forma, facilita com que os objetos sejam persistidos, transmitidos pela rede ou compartilhados entre aplicativos.

4.3. COMO O PARADIGMA FUNCIONAL É UTILIZADO PELA API STREAM NO JAVA?

O paradigma funcional oferece uma maneira mais concisa e expressiva de processar dados, inspirando-se em linguagens funcionais como Haskell e Scala. Através da API

Stream, introduzida no Java 8, trouxe uma nova forma de trabalhar com coleções, adotando conceitos desse paradigma funcional.

Alguns dos principais conceitos do paradigma funcional aplicados na Stream API:

1. Imutabilidade: Os elementos de um stream não são modificados diretamente. Em vez disso, novas streams são criadas a partir de transformações. Isso evita efeitos colaterais indesejados e torna o código mais previsível.
2. Funções como cidadãos de primeira classe: Funções podem ser atribuídas a variáveis, passadas como argumentos e retornadas por outras funções. As lambdas e as referências a métodos são exemplos disso em Java.
3. Mapeamento: Transforma cada elemento de um stream em outro elemento, aplicando uma função a cada um.
4. Filtragem: Retorna um novo stream contendo apenas os elementos que satisfazem um determinado predicado.
5. Redução: Reduz um stream a um único valor, como a soma de todos os elementos, o maior valor, etc.

4.4. QUANDO TRABALHAMOS COM JAVA, QUAL PADRÃO DE DESENVOLVIMENTO É ADOTADO NA PERSISTÊNCIA DE DADOS EM ARQUIVOS?

Ao trabalhar com a persistência de dados em arquivos em Java, não existe um padrão de projeto específico e universalmente adotado, como acontece com a persistência em bancos de dados relacionais (onde o padrão ORM, como Hibernate, é amplamente utilizado). No entanto, alguns princípios e técnicas são comumente aplicados para organizar e estruturar o código de forma eficiente e reutilizável.

Principais abordagens e considerações:

- **Serialização**:

A forma mais simples de persistir objetos em arquivos é utilizando a interface Serializable, pois essa abordagem permite converter o estado de um objeto em uma sequência de bytes e gravar em um arquivo.

Vantagens: Simples de implementar, suporta hierarquias de objetos.

Desvantagens: Pode gerar arquivos grandes e complexos, que pode não ser tão ideal para alguns formatos de dados específicos (como JSON ou XML).

- **Formatos de dados:**

JSON: Um formato leve e humano-legível, amplamente utilizado para troca de dados. Bibliotecas como Jackson e Gson facilitam a serialização e desserialização de objetos Java em JSON.

XML: Outro formato estruturado, mais verboso que o JSON, mas com um esquema mais rígido. Bibliotecas como JAXB permitem mapear objetos Java para XML.

Propriedades: Formato simples, baseado em pares chave-valor, ideal para configurações e dados simples.

- **Bibliotecas e frameworks:**

Apache Commons IO: Fornece classes utilitárias para trabalhar com arquivos e streams, simplificando operações de leitura e escrita.

Google Gson: Uma biblioteca popular para converter objetos Java em JSON e vice-versa.

Jackson: Outra biblioteca para serialização/desserialização JSON, com funcionalidades avançadas.

JAXB: Para trabalhar com XML e mapear objetos Java para esquemas XML.

REFERÊNCIAS

ORACLE, Java™ Platform, Standard Edition 8 API Specification. Disponível em: <https://docs.oracle.com/javase/8/docs/api/>. Acesso em 09 de dezembro de 2024.

BAELDUNG, How to Serialize a Singleton in Java. Disponível em: <https://www.baeldung.com/java-serialize-singleton>. Acesso em 09 de dezembro de 2024.

CODEHS, Data Persistence in Java. Disponível em: <https://codehs.com/tutorial/david/data-persistence-in-java>. Acesso em 09 de dezembro de 2024.

MEDIUM, Programação Funcional no Java. Disponível em: ≤https://medium.com/@nvieirarafael/programa%C3%A7%C3%A3o-funcional-no-java-2a005964cb20>. Acesso em 09 de dezembro de 2024.