Guia Completo: Prova de Programação Orientada a Objetos

Prova: Quinta-feira, 25 de Setembro de 2025

Conteúdos Específicos da Prova

Este guia aborda em detalhes todos os tópicos que serão cobrados na sua prova:

1. **Arquivos de Texto** - Leitura e escrita em formato texto
2. **Arquivos Binários** - Leitura e escrita em formato binário
3. **Arquivos JSON** - Leitura e escrita em formato JSON
4. **Serialização de Objetos** - Interface Serializable e transformação em bytes
5. **Streams** - Manipulação funcional de coleções (filter, map, sorted, reduce)
6. **Padrões Arquiteturais e de Projeto** - MVC, Singleton, Strategy

---

1. ARQUIVOS DE TEXTO - COMUNICAÇÃO LEGÍVEL

1.1 Conceito Fundamental

Arquivos de texto armazenam informações usando caracteres que podem ser lidos tanto por humanos quanto por programas. Esta característica os torna ideais para dados que precisam ser inspecionados, editados manualmente, ou compartilhados entre diferentes sistemas. Quando você salva um número como "123" em um arquivo de texto, está armazenando três caracteres ('1', '2', '3') que ocupam 3 bytes, diferentemente da representação binária que ocuparia 4 bytes mas seria ilegível.

1.2 Vantagens dos Arquivos de Texto

Legibilidade Universal: Qualquer editor de texto pode abrir e exibir o conteúdo, facilitando depuração e manutenção. Isso é especialmente valioso durante desenvolvimento quando você precisa verificar se os dados estão sendo salvos corretamente.

Portabilidade Entre Sistemas: Arquivos de texto podem ser lidos em qualquer sistema operacional e linguagem de programação, tornando-os ideais para intercâmbio de dados entre diferentes plataformas.

Facilidade de Edição Manual: Usuários podem modificar configurações, corrigir dados, ou fazer ajustes usando qualquer editor de texto, sem necessidade de ferramentas

especializadas.

Transparência e Auditoria: O conteúdo é completamente visível, permitindo auditoria manual e verificação de integridade dos dados.

1.3 Limitações dos Arquivos de Texto

Ineficiência de Espaço: Números grandes ocupam mais espaço como texto. O número 1000000 ocupa 7 bytes como texto, mas apenas 4 bytes como inteiro binário.

Performance de Conversão: Cada leitura/escrita requer conversão entre representação interna (binária) e textual, consumindo tempo de processamento.

Limitações de Precisão: Números de ponto flutuante podem perder precisão quando convertidos para texto e de volta para binário.

1.4 Implementação Prática

Classes Fundamentais

BufferedWriter: Otimiza escrita agrupando múltiplas operações em buffers internos, reduzindo chamadas ao sistema operacional.

BufferedReader: Otimiza leitura carregando blocos de dados em memória, permitindo acesso mais rápido linha por linha.

FileWriter/FileReader: Classes base que conectam com o sistema de arquivos, mas menos eficientes quando usadas diretamente.

Escrita de Arquivos de Texto

Java

```
public static void salvarRelatorio(List<String> linhas, String nomeArquivo) {
    // Try-with-resources garante fechamento automático do arquivo
    try (BufferedWriter writer = new BufferedWriter(new
FileWriter(nomeArquivo))) {

        // Escreve cabeçalho com timestamp
        writer.write("=== RELATÓRIO GERADO EM " + LocalDateTime.now() + "
===");
        writer.newLine();  // Quebra de linha compatível com o SO
        writer.newLine();  // Linha em branco para separação

        // Escreve cada linha do relatório
        for (int i = 0; i < linhas.size(); i++) {
            writer.write((i + 1) + ". " + linhas.get(i));
            writer.newLine();
```

```
        }

        // Escreve rodapé
        writer.newLine();
        writer.write("=== FIM DO RELATÓRIO ===");

        System.out.println("Relatório salvo com sucesso: " + nomeArquivo);

    } catch (IOException e) {
        System.err.println("Erro ao salvar relatório: " + e.getMessage());
        // Em aplicação real, considere re-lançar ou logar o erro
    }
}
```

Leitura de Arquivos de Texto

Java

```
public static List<String> carregarConfiguracoes(String nomeArquivo) {
    List<String> configuracoes = new ArrayList<>();

    try (BufferedReader reader = new BufferedReader(new
FileReader(nomeArquivo))) {
        String linha;
        int numeroLinha = 1;

        // Lê arquivo linha por linha até o fim
        while ((linha = reader.readLine()) != null) {

            // Remove espaços em branco das extremidades
            linha = linha.trim();

            // Ignora linhas vazias e comentários
            if (!linha.isEmpty() && !linha.startsWith("#")) {
                configuracoes.add(linha);
                System.out.println("Configuração " + numeroLinha + ": " +
linha);
            }

            numeroLinha++;
        }

        System.out.println("Total de configurações carregadas: " +
configuracoes.size());

    } catch (FileNotFoundException e) {
        System.err.println("Arquivo não encontrado: " + nomeArquivo);
```

```
    } catch (IOException e) {
        System.err.println("Erro ao ler arquivo: " + e.getMessage());
    }

    return configuracoes;
}
```

Anexação vs Sobrescrita

Java

```
// SOBRESCRITA (padrão) - cria novo arquivo ou substitui existente
public static void criarNovoLog(String mensagem) {
    try (BufferedWriter writer = new BufferedWriter(new
FileWriter("aplicacao.log"))) {
        writer.write("[NOVO LOG] " + LocalDateTime.now() + ": " + mensagem);
    } catch (IOException e) {
        System.err.println("Erro ao criar log: " + e.getMessage());
    }
}

// ANEXAÇÃO - adiciona ao final do arquivo existente
public static void adicionarAoLog(String mensagem) {
    try (BufferedWriter writer = new BufferedWriter(new
FileWriter("aplicacao.log", true))) {
        writer.write("[" + LocalTime.now() + "] " + mensagem);
        writer.newLine();
    } catch (IOException e) {
        System.err.println("Erro ao adicionar ao log: " + e.getMessage());
    }
}
```

1.5 Casos de Uso Ideais

Arquivos de Configuração: Parâmetros que usuários podem modificar manualmente (config.txt, settings.ini).

Logs de Sistema: Registros que administradores precisam ler para diagnóstico e auditoria.

Relatórios: Saídas que serão lidas por humanos ou importadas por outras ferramentas.

Dados de Intercâmbio: Informações compartilhadas entre sistemas diferentes que precisam de formato universal.

---

2. ARQUIVOS BINÁRIOS - EFICIÊNCIA MÁXIMA

2.1 Conceito Fundamental

Arquivos binários armazenam dados na mesma representação que o computador usa internamente, sem conversão para formato legível por humanos. Quando você escreve o inteiro 1000000 em um arquivo binário, os 4 bytes que representam esse número na memória são copiados diretamente para o arquivo. Esta abordagem elimina overhead de conversão e maximiza eficiência tanto de espaço quanto de velocidade.

2.2 Vantagens dos Arquivos Binários

Eficiência de Espaço: Dados ocupam o mínimo espaço possível. Um double sempre ocupa exatamente 8 bytes, independentemente do valor.

Velocidade de Processamento: Não há conversão entre representações - os dados são copiados diretamente da memória para o arquivo e vice-versa.

Preservação de Precisão: Números de ponto flutuante mantêm precisão exata, sem perdas por conversão textual.

Estruturas Complexas: Podem armazenar eficientemente grandes volumes de dados estruturados, como registros de banco de dados ou arrays multidimensionais.

2.3 Limitações dos Arquivos Binários

Ilegibilidade Humana: Impossível inspecionar conteúdo sem ferramentas especializadas, dificultando depuração.

Dependência de Arquitetura: Ordem de bytes (endianness) pode variar entre diferentes arquiteturas de processador.

Fragilidade de Formato: Qualquer alteração na estrutura de dados pode tornar arquivos existentes ilegíveis.

Falta de Portabilidade: Arquivos criados em uma plataforma podem não ser legíveis em outra.

2.4 Implementação Prática

Classes Fundamentais

DataOutputStream: Fornece métodos específicos para escrever cada tipo primitivo (writeInt, writeDouble, writeUTF, etc.).

DataInputStream: Fornece métodos correspondentes para ler cada tipo primitivo (readInt, readDouble, readUTF, etc.).

Ordem Crítica: Dados devem ser lidos na exata mesma ordem em que foram escritos.

Escrita de Arquivos Binários

Java

```java
public static void salvarDadosEmpresa(String nomeArquivo, List<Funcionario>
funcionarios) {
    try (DataOutputStream dos = new DataOutputStream(new
FileOutputStream(nomeArquivo))) {

        // Escreve cabeçalho do arquivo
        dos.writeUTF("DADOS_EMPRESA_V1.0");  // Identificador e versão
        dos.writeLong(System.currentTimeMillis());  // Timestamp de criação
        dos.writeInt(funcionarios.size());  // Quantidade de registros

        // Escreve dados de cada funcionário
        for (Funcionario func : funcionarios) {
            dos.writeInt(func.getId());           // 4 bytes - ID único
            dos.writeUTF(func.getNome());         // Tamanho variável - nome
            dos.writeDouble(func.getSalario());   // 8 bytes - salário
            dos.writeInt(func.getIdade());        // 4 bytes - idade
            dos.writeBoolean(func.isAtivo());     // 1 byte - status ativo
            dos.writeUTF(func.getDepartamento()); // Tamanho variável - depto
        }

        // Escreve checksum simples para verificação de integridade
        dos.writeInt(funcionarios.size() * 12345);  // Valor de verificação

        System.out.println("Dados salvos: " + funcionarios.size() + "
funcionários");

    } catch (IOException e) {
        System.err.println("Erro ao salvar dados binários: " +
e.getMessage());
    }
}
```

Leitura de Arquivos Binários

Java

```java
public static List<Funcionario> carregarDadosEmpresa(String nomeArquivo) {
    List<Funcionario> funcionarios = new ArrayList<>();

    try (DataInputStream dis = new DataInputStream(new
FileInputStream(nomeArquivo))) {

        // Lê e valida cabeçalho
```

```
        String identificador = dis.readUTF();
        if (!identificador.equals("DADOS_EMPRESA_V1.0")) {
            throw new IOException("Formato de arquivo inválido: " +
identificador);
        }

        long timestampCriacao = dis.readLong();
        int quantidadeFuncionarios = dis.readInt();

        System.out.println("Arquivo criado em: " + new
Date(timestampCriacao));
        System.out.println("Funcionários a carregar: " +
quantidadeFuncionarios);

        // Lê dados de cada funcionário na MESMA ORDEM da escrita
        for (int i = 0; i < quantidadeFuncionarios; i++) {
            int id = dis.readInt();                    // Primeiro: ID
            String nome = dis.readUTF();               // Segundo: nome
            double salario = dis.readDouble();         // Terceiro: salário
            int idade = dis.readInt();                 // Quarto: idade
            boolean ativo = dis.readBoolean();         // Quinto: status
            String departamento = dis.readUTF();       // Sexto: departamento

            Funcionario func = new Funcionario(id, nome, salario, idade,
ativo, departamento);
            funcionarios.add(func);
        }

        // Verifica integridade com checksum
        int checksumEsperado = funcionarios.size() * 12345;
        int checksumLido = dis.readInt();

        if (checksumLido != checksumEsperado) {
            System.err.println("AVISO: Checksum não confere - arquivo pode
estar corrompido");
        }

    } catch (EOFException e) {
        System.err.println("Fim inesperado do arquivo - dados podem estar
incompletos");
    } catch (IOException e) {
        System.err.println("Erro ao carregar dados binários: " +
e.getMessage());
    }

    return funcionarios;
}
```

Tratamento de Fim de Arquivo

Java

```java
public static List<Double> lerTodosNumeros(String arquivo) {
    List<Double> numeros = new ArrayList<>();

    try (DataInputStream dis = new DataInputStream(new
FileInputStream(arquivo))) {

        // Loop infinito - sai pela EOFException
        while (true) {
            try {
                double numero = dis.readDouble();
                numeros.add(numero);
            } catch (EOFException e) {
                // Fim do arquivo alcançado - comportamento normal
                System.out.println("Fim do arquivo. Total de números lidos:
" + numeros.size());
                break;
            }
        }

    } catch (IOException e) {
        System.err.println("Erro de I/O: " + e.getMessage());
    }

    return numeros;
}
```

2.5 Casos de Uso Ideais

Grandes Volumes de Dados Numéricos: Sensores, medições científicas, dados financeiros históricos.

Performance Crítica: Aplicações que processam milhões de registros onde cada milissegundo importa.

Dados Estruturados Internos: Caches, índices, estruturas de dados que só serão lidas pelo mesmo programa.

Aplicações Embarcadas: Sistemas com limitações de memória e processamento onde eficiência é crucial.

---

3. ARQUIVOS JSON - INTERCÂMBIO UNIVERSAL

3.1 Conceito Fundamental

JSON (JavaScript Object Notation) é um formato de intercâmbio de dados baseado em texto que usa sintaxe simples e legível para representar estruturas de dados complexas. Embora tenha origem no JavaScript, JSON tornou-se o padrão universal para comunicação entre sistemas devido à sua simplicidade, legibilidade, e suporte amplo em todas as linguagens de programação modernas.

3.2 Filosofia e Princípios do JSON

Simplicidade Radical: JSON usa apenas seis tipos de dados (string, number, boolean, null, object, array) para representar qualquer estrutura de dados complexa.

Legibilidade Humana: Estrutura hierárquica clara com indentação e nomes descritivos torna dados facilmente compreensíveis.

Interoperabilidade Universal: Qualquer linguagem pode processar JSON, tornando-o ideal para APIs, configurações, e intercâmbio de dados.

Autodescrição: Dados incluem seus próprios metadados através de nomes de campos, eliminando necessidade de esquemas externos.

3.3 Estrutura e Sintaxe JSON

Tipos de Dados Fundamentais

String: Texto entre aspas duplas, com escape para caracteres especiais.
Number: Números inteiros ou decimais, sem aspas.
Boolean: Valores true ou false (sem aspas).
Null: Valor nulo explícito.
Object: Coleção de pares chave-valor entre chaves {}.
Array: Lista ordenada de valores entre colchetes [].

Exemplo Complexo de Estrutura JSON

JSON

```
{
    "empresa": {
        "nome": "TechCorp Solutions",
        "fundacao": 2010,
        "ativa": true,
        "endereco": {
            "rua": "Av. Paulista, 1000",
            "cidade": "São Paulo",
            "cep": "01310-100",
```

```
            "coordenadas": {
                "latitude": -23.5505,
                "longitude": -46.6333
            }
        },
        "departamentos": [
            {
                "nome": "Desenvolvimento",
                "funcionarios": 25,
                "tecnologias": ["Java", "Python", "JavaScript"],
                "projetos": [
                    {
                        "nome": "Sistema ERP",
                        "status": "em_andamento",
                        "prazo": "2024-12-31",
                        "orcamento": 150000.00
                    },
                    {
                        "nome": "App Mobile",
                        "status": "concluido",
                        "prazo": "2024-06-30",
                        "orcamento": 80000.00
                    }
                ]
            },
            {
                "nome": "Marketing",
                "funcionarios": 8,
                "campanhas_ativas": 3,
                "orcamento_mensal": 25000.00
            }
        ],
        "contato": {
            "telefone": "+55-11-3000-0000",
            "email": "contato@techcorp.com",
            "website": "https://www.techcorp.com"
        }
    }
}
```

3.4 Vantagens do JSON

Legibilidade e Depuração: Desenvolvedores podem ler, entender e modificar dados JSON usando qualquer editor de texto.

Suporte Universal: Bibliotecas JSON existem para todas as linguagens principais, garantindo interoperabilidade.

Flexibilidade de Esquema: Estrutura pode evoluir sem quebrar compatibilidade - campos podem ser adicionados sem afetar código existente.

Integração Web Nativa: JavaScript processa JSON nativamente, tornando-o perfeito para aplicações web.

Facilidade de Validação: Estrutura hierárquica clara facilita validação e verificação de integridade.

3.5 Limitações do JSON

Tipos Limitados: Não suporta tipos específicos como Date, BigDecimal, ou tipos customizados - tudo deve ser representado com tipos básicos.

Sem Comentários: JSON puro não permite comentários inline, dificultando documentação.

Sem Referências: Não pode representar referências circulares ou compartilhadas entre objetos.

Overhead de Tamanho: Nomes de campos são repetidos, aumentando tamanho comparado a formatos binários.

Parsing Obrigatório: Sempre requer parsing para converter entre texto e objetos, consumindo processamento.

3.6 Implementação Prática em Java

Escrita Manual de JSON

Java

```
public static void salvarEmpresaJSON(Empresa empresa, String nomeArquivo) {
    try (BufferedWriter writer = new BufferedWriter(new
FileWriter(nomeArquivo))) {

        // Constrói JSON manualmente com indentação adequada
        StringBuilder json = new StringBuilder();
        json.append("{\n");
        json.append("  \"empresa\": {\n");
        json.append("    \"nome\":
\"").append(empresa.getNome()).append("\",\n");
        json.append("    \"fundacao\":
").append(empresa.getAnoFundacao()).append(",\n");
        json.append("    \"ativa\":
").append(empresa.isAtiva()).append(",\n");
        json.append("    \"funcionarios\": [\n");
```

```
        // Adiciona array de funcionários
        List<Funcionario> funcionarios = empresa.getFuncionarios();
        for (int i = 0; i < funcionarios.size(); i++) {
            Funcionario func = funcionarios.get(i);
            json.append("      {\n");
            json.append("        \"id\":
").append(func.getId()).append(",\n");
            json.append("        \"nome\":
\"").append(func.getNome()).append("\",\n");
            json.append("        \"salario\":
").append(func.getSalario()).append(",\n");
            json.append("        \"departamento\":
\"").append(func.getDepartamento()).append("\"\n");
            json.append("      }");

            // Adiciona vírgula se não for o último elemento
            if (i < funcionarios.size() - 1) {
                json.append(",");
            }
            json.append("\n");
        }

        json.append("    ],\n");
        json.append("    \"timestamp_criacao\":
\"").append(LocalDateTime.now()).append("\"\n");
        json.append("  }\n");
        json.append("}");

        // Escreve JSON no arquivo
        writer.write(json.toString());
        System.out.println("Dados JSON salvos: " + nomeArquivo);

    } catch (IOException e) {
        System.err.println("Erro ao salvar JSON: " + e.getMessage());
    }
}
```

Leitura e Parsing Manual de JSON

Java

```
public static Empresa carregarEmpresaJSON(String nomeArquivo) {
    try (BufferedReader reader = new BufferedReader(new
FileReader(nomeArquivo))) {

        // Lê todo o conteúdo do arquivo
        StringBuilder jsonContent = new StringBuilder();
```

```
        String linha;
        while ((linha = reader.readLine()) != null) {
            jsonContent.append(linha).append("\n");
        }

        String json = jsonContent.toString();
        System.out.println("JSON carregado:\n" + json);

        // Parsing manual simples (em produção, use biblioteca como
Jackson/Gson)
        Empresa empresa = new Empresa();

        // Extrai nome da empresa (busca padrão "nome": "valor")
        String nome = extrairValorString(json, "nome");
        empresa.setNome(nome);

        // Extrai ano de fundação
        int fundacao = extrairValorInteiro(json, "fundacao");
        empresa.setAnoFundacao(fundacao);

        // Extrai status ativo
        boolean ativa = extrairValorBoolean(json, "ativa");
        empresa.setAtiva(ativa);

        System.out.println("Empresa carregada: " + empresa.getNome());
        return empresa;

    } catch (IOException e) {
        System.err.println("Erro ao carregar JSON: " + e.getMessage());
        return null;
    }
}

// Métodos auxiliares para parsing manual (simplificados)
private static String extrairValorString(String json, String chave) {
    String padrao = "\"" + chave + "\": \"";
    int inicio = json.indexOf(padrao) + padrao.length();
    int fim = json.indexOf("\"", inicio);
    return json.substring(inicio, fim);
}

private static int extrairValorInteiro(String json, String chave) {
    String padrao = "\"" + chave + "\": ";
    int inicio = json.indexOf(padrao) + padrao.length();
    int fim = json.indexOf(",", inicio);
    if (fim == -1) fim = json.indexOf("\n", inicio);
    return Integer.parseInt(json.substring(inicio, fim).trim());
}
```

```java
private static boolean extrairValorBoolean(String json, String chave) {
    String padrao = "\"" + chave + "\": ";
    int inicio = json.indexOf(padrao) + padrao.length();
    int fim = json.indexOf(",", inicio);
    if (fim == -1) fim = json.indexOf("\n", inicio);
    return Boolean.parseBoolean(json.substring(inicio, fim).trim());
}
```

3.7 Casos de Uso Ideais

APIs Web: Comunicação entre frontend JavaScript e backend Java.

Arquivos de Configuração: Configurações complexas que usuários podem editar manualmente.

Intercâmbio Entre Sistemas: Dados compartilhados entre diferentes tecnologias e plataformas.

Logs Estruturados: Registros que precisam ser processados por ferramentas de análise.

Dados Semi-estruturados: Informações que não têm esquema rígido e podem evoluir.

---

4. SERIALIZAÇÃO DE OBJETOS - PERSISTÊNCIA NATIVA

4.1 Conceito Fundamental

Serialização é o processo de converter objetos Java complexos em uma sequência de bytes que pode ser armazenada em arquivos, enviada pela rede, ou mantida em memória para uso posterior. Este mecanismo vai muito além da simples escrita de dados primitivos - ele preserva a estrutura completa do objeto, incluindo suas relações com outros objetos, hierarquia de herança, e estado interno completo.

4.2 Filosofia da Serialização

Preservação de Identidade: Quando você serializa um objeto, não está apenas salvando dados - está preservando a "identidade" completa do objeto, incluindo sua classe, estado, e relacionamentos.

Transparência: O processo é amplamente automático - você marca uma classe como `Serializable` e o Java cuida dos detalhes complexos de conversão.

Integridade de Grafo: Se um objeto contém referências para outros objetos, todo o grafo de objetos é serializado recursivamente, mantendo todas as conexões.

Reconstrução Fiel: Durante a deserialização, objetos são reconstruídos com exatamente o mesmo estado que tinham quando foram serializados.

4.3 Interface Serializable - Marcador de Intenção

A interface `Serializable` é um marcador que não define métodos - sua presença simplesmente informa ao sistema de serialização do Java que é seguro e intencional converter objetos desta classe em bytes. Esta abordagem de "opt-in" garante que apenas classes explicitamente projetadas para serialização sejam processadas.

Implementação Básica

Java

```
public class Produto implements Serializable {
    // Controle de versão - FUNDAMENTAL para compatibilidade
    private static final long serialVersionUID = 1L;

    // Campos que serão serializados
    private String nome;
    private double preco;
    private int quantidadeEstoque;
    private Date dataCadastro;
    private List<String> categorias;

    // Campos que NÃO serão serializados
    private transient String observacoesInternas;  // Dados temporários
    private transient Connection conexaoBanco;     // Recursos não
serializáveis
    private transient double margemCalculada;      // Valores calculados

    public Produto(String nome, double preco, int quantidade) {
        this.nome = nome;
        this.preco = preco;
        this.quantidadeEstoque = quantidade;
        this.dataCadastro = new Date();
        this.categorias = new ArrayList<>();

        // Campos transient são inicializados normalmente
        this.observacoesInternas = "Produto criado";
        this.margemCalculada = preco * 0.3; // 30% de margem
    }

    // Getters e setters...

    /**
     * Método chamado APÓS deserialização para recriar campos transient
```

```
     */
    private void readObject(ObjectInputStream ois) throws IOException,
ClassNotFoundException {
        // Deserialização padrão primeiro
        ois.defaultReadObject();

        // Recria campos transient que precisam ser recalculados
        this.margemCalculada = this.preco * 0.3;
        this.observacoesInternas = "Produto deserializado em " + new Date();

        System.out.println("Produto deserializado: " + this.nome);
    }
}
```

4.4 SerialVersionUID - Controle de Compatibilidade

O `serialVersionUID` é um identificador único que Java usa para verificar compatibilidade entre a versão da classe que foi serializada e a versão atual da classe durante deserialização. Este mecanismo é crucial para evolução de software onde classes podem ser modificadas após objetos terem sido serializados.

Estratégias de Versionamento

UID Explícito Fixo: Mantém o mesmo valor durante mudanças compatíveis (adicionar campos, métodos).

UID Explícito Evolutivo: Altera o valor quando mudanças são incompatíveis (remover campos, alterar tipos).

UID Automático: Java gera baseado na estrutura da classe, mas pode quebrar com mudanças mínimas.

Java

```
public class ContaBancaria implements Serializable {
    // Versão 1.0 - estrutura inicial
    private static final long serialVersionUID = 1L;

    private String numero;
    private double saldo;
    private String titular;

    // Versão 2.0 - campos adicionados (compatível - mesmo serialVersionUID)
    private Date dataAbertura;      // Campo novo - será null em objetos
antigos
    private String agencia;         // Campo novo - será null em objetos
antigos
```

```
    // Construtor que funciona com objetos antigos e novos
    public ContaBancaria(String numero, String titular, double saldo) {
        this.numero = numero;
        this.titular = titular;
        this.saldo = saldo;
        this.dataAbertura = new Date();  // Valor padrão para novos objetos
        this.agencia = "0001";           // Valor padrão
    }

    /**
     * Método para lidar com objetos deserializados de versões antigas
     */
    private void readObject(ObjectInputStream ois) throws IOException,
ClassNotFoundException {
        ois.defaultReadObject();

        // Inicializa campos que podem ser null em versões antigas
        if (this.dataAbertura == null) {
            this.dataAbertura = new Date();  // Data atual como fallback
        }
        if (this.agencia == null) {
            this.agencia = "0001";  // Agência padrão
        }
    }
}
```

4.5 Palavra-chave Transient - Controle Granular

O modificador `transient` permite excluir campos específicos do processo de serialização. Esta funcionalidade é essencial para campos que não devem ou não podem ser persistidos.

Casos de Uso para Transient

Dados Sensíveis: Senhas, tokens, informações confidenciais que não devem ser persistidas.

Recursos do Sistema: Conexões de banco, streams, sockets que não podem ser serializados.

Valores Calculados: Campos derivados de outros campos que podem ser recalculados.

Caches Temporários: Dados mantidos em memória para performance que não precisam ser preservados.

Referências Circulares: Quebrar ciclos que poderiam causar problemas na serialização.

Java

```java
public class Usuario implements Serializable {
    private static final long serialVersionUID = 1L;

    // Dados persistidos
    private String nome;
    private String email;
    private Date dataCadastro;
    private List<String> permissoes;

    // Dados NÃO persistidos
    private transient String senha;              // Segurança - nunca serializar senhas
    private transient Connection conexao;        // Recurso - não pode ser serializado
    private transient boolean logado;            // Estado temporário
    private transient Map<String, Object> cache; // Cache - pode ser recriado

    public Usuario(String nome, String email, String senha) {
        this.nome = nome;
        this.email = email;
        this.senha = senha;
        this.dataCadastro = new Date();
        this.permissoes = new ArrayList<>();
        this.logado = false;
        this.cache = new HashMap<>();
    }

    /**
     * Reconstrói campos transient após deserialização
     */
    private void readObject(ObjectInputStream ois) throws IOException, ClassNotFoundException {
        ois.defaultReadObject();

        // Recria campos transient com valores padrão seguros
        this.senha = null;           // Força nova autenticação
        this.logado = false;         // Usuário não está logado após deserialização
        this.cache = new HashMap<>(); // Cache vazio
        this.conexao = null;         // Conexão deve ser reestabelecida

        System.out.println("Usuário " + nome + " deserializado - requer nova autenticação");
    }
}
```

4.6 Processo de Serialização e Deserialização

Serialização - Convertendo Objetos em Bytes

Java

```
public static void salvarUsuarios(List<Usuario> usuarios, String arquivo) {
    try (ObjectOutputStream oos = new ObjectOutputStream(new
FileOutputStream(arquivo))) {

        // Escreve metadados do arquivo
        oos.writeUTF("USUARIOS_V1.0");  // Identificador
        oos.writeLong(System.currentTimeMillis());  // Timestamp
        oos.writeInt(usuarios.size());  // Quantidade

        // Serializa cada usuário
        for (Usuario usuario : usuarios) {
            oos.writeObject(usuario);  // Serialização automática completa
            System.out.println("Usuário serializado: " + usuario.getNome());
        }

        System.out.println("Total de usuários salvos: " + usuarios.size());

    } catch (IOException e) {
        System.err.println("Erro ao serializar usuários: " + e.getMessage());
    }
}
```

Deserialização - Reconstruindo Objetos

Java

```
public static List<Usuario> carregarUsuarios(String arquivo) {
    List<Usuario> usuarios = new ArrayList<>();

    try (ObjectInputStream ois = new ObjectInputStream(new
FileInputStream(arquivo))) {

        // Lê e valida metadados
        String identificador = ois.readUTF();
        if (!identificador.equals("USUARIOS_V1.0")) {
            throw new IOException("Formato inválido: " + identificador);
        }

        long timestamp = ois.readLong();
        int quantidade = ois.readInt();
```

```
        System.out.println("Arquivo criado em: " + new Date(timestamp));
        System.out.println("Usuários a carregar: " + quantidade);

        // Deserializa cada usuário
        for (int i = 0; i < quantidade; i++) {
            try {
                Usuario usuario = (Usuario) ois.readObject();  // Cast
necessário
                usuarios.add(usuario);
                System.out.println("Usuário carregado: " +
usuario.getNome());
            } catch (ClassNotFoundException e) {
                System.err.println("Classe Usuario não encontrada: " +
e.getMessage());
            }
        }

    } catch (IOException e) {
        System.err.println("Erro ao deserializar usuários: " +
e.getMessage());
    }

    return usuarios;
}
```

4.7 Serialização de Coleções Complexas

Java

```
public static void salvarSistemaCompleto(SistemaEmpresa sistema, String
arquivo) {
    try (ObjectOutputStream oos = new ObjectOutputStream(new
FileOutputStream(arquivo))) {

        // Serializa objeto complexo com múltiplas coleções
        oos.writeObject(sistema);  // Todo o grafo de objetos é serializado
automaticamente

        System.out.println("Sistema completo serializado:");
        System.out.println("- Funcionários: " +
sistema.getFuncionarios().size());
        System.out.println("- Departamentos: " +
sistema.getDepartamentos().size());
        System.out.println("- Projetos: " + sistema.getProjetos().size());

    } catch (IOException e) {
```

```
            System.err.println("Erro ao serializar sistema: " + e.getMessage());
        }
    }

    public class SistemaEmpresa implements Serializable {
        private static final long serialVersionUID = 1L;

        private String nomeEmpresa;
        private List<Funcionario> funcionarios;      // Coleção serializada
    automaticamente
        private Map<String, Departamento> departamentos;  // Map serializado
    automaticamente
        private Set<Projeto> projetos;               // Set serializado
    automaticamente

        // Dados não persistidos
        private transient DatabaseConnection db;     // Conexão recriada após
    deserialização
        private transient Map<String, Object> cache; // Cache recriado vazio

        public SistemaEmpresa(String nome) {
            this.nomeEmpresa = nome;
            this.funcionarios = new ArrayList<>();
            this.departamentos = new HashMap<>();
            this.projetos = new HashSet<>();
            this.cache = new HashMap<>();
        }

        private void readObject(ObjectInputStream ois) throws IOException,
    ClassNotFoundException {
            ois.defaultReadObject();

            // Recria recursos transient
            this.cache = new HashMap<>();
            this.db = null;  // Será reconectado quando necessário

            System.out.println("Sistema " + nomeEmpresa + " deserializado com:");
            System.out.println("- " + funcionarios.size() + " funcionários");
            System.out.println("- " + departamentos.size() + " departamentos");
            System.out.println("- " + projetos.size() + " projetos");
        }
    }
```

4.8 Casos de Uso Ideais

Caches de Objetos Complexos: Salvar objetos processados para evitar recálculos custosos.

Estados de Aplicação: Preservar estado completo de aplicações entre execuções.

Comunicação Entre JVMs: Enviar objetos complexos pela rede entre aplicações Java.

Implementação de Undo/Redo: Salvar estados anteriores de objetos para funcionalidades de desfazer.

Clonagem Profunda: Criar cópias independentes de objetos complexos com múltiplas referências.

---

5. STREAMS - PROGRAMAÇÃO FUNCIONAL

5.1 Conceito Fundamental e Mudança de Paradigma

Streams representam uma revolução na forma como processamos coleções em Java, introduzindo conceitos de programação funcional que transformam código imperativo verboso em expressões declarativas concisas e expressivas. Esta mudança não é apenas sintática - representa uma forma fundamentalmente diferente de pensar sobre processamento de dados, focando no "o que fazer" ao invés de "como fazer".

5.2 Paradigma Imperativo vs Funcional

Abordagem Imperativa Tradicional

Na programação imperativa, você descreve explicitamente cada passo do processo, controlando manualmente loops, condições, e estado mutável. Este estilo é verboso e propenso a erros porque requer gerenciamento manual de muitos detalhes de baixo nível.

Java

```java
// IMPERATIVO - descreve COMO fazer passo a passo
public List<String> processarNomesImperativo(List<Pessoa> pessoas) {
    List<String> resultado = new ArrayList<>();  // Estado mutável

    for (Pessoa pessoa : pessoas) {              // Loop manual
        if (pessoa.getIdade() >= 18) {           // Condição manual
            String nome = pessoa.getNome();      // Extração manual
            String nomeUpper = nome.toUpperCase(); // Transformação manual
            resultado.add(nomeUpper);            // Adição manual
        }
    }

    // Ordenação manual
    Collections.sort(resultado);
```

```
    return resultado;
}
```

Abordagem Funcional com Streams

Na programação funcional, você descreve declarativamente o que quer fazer, usando operações de alto nível que expressam intenção de forma clara e concisa. O sistema cuida automaticamente dos detalhes de implementação.

Java

```
// FUNCIONAL - descreve O QUE fazer declarativamente
public List<String> processarNomesFuncional(List<Pessoa> pessoas) {
    return pessoas.stream()                        // Cria pipeline de
processamento
        .filter(pessoa -> pessoa.getIdade() >= 18)  // O QUE filtrar
        .map(Pessoa::getNome)                      // O QUE extrair
        .map(String::toUpperCase)                  // O QUE transformar
        .sorted()                                  // O QUE ordenar
        .collect(Collectors.toList());             // O QUE coletar
}
```

5.3 Anatomia de um Stream Pipeline

Componentes Fundamentais

Fonte (Source): Origem dos dados - coleção, array, gerador, ou I/O.
Operações Intermediárias: Transformações que produzem novos Streams.
Operação Terminal: Produz resultado final e dispara execução.

Java

```
// Estrutura conceitual de um pipeline
fonte.stream()                    // 1. Criação do Stream
    .operacaoIntermediaria1()     // 2. Transformação (lazy)
    .operacaoIntermediaria2()     // 3. Outra transformação (lazy)
    .operacaoIntermediaria3()     // 4. Mais transformação (lazy)
    .operacaoTerminal();          // 5. Execução e resultado final
```

Lazy Evaluation - Avaliação Preguiçosa

Uma característica fundamental dos Streams é que operações intermediárias não são executadas imediatamente - elas apenas constroem uma descrição do processamento

desejado. A execução real só acontece quando uma operação terminal é invocada, permitindo otimizações importantes.

Java

```java
List<String> nomes = Arrays.asList("Ana", "Bruno", "Carlos", "Diana",
"Eduardo");

Stream<String> pipeline = nomes.stream()
    .filter(nome -> {
        System.out.println("Filtrando: " + nome);  // NÃO executa ainda
        return nome.length() > 4;
    })
    .map(nome -> {
        System.out.println("Transformando: " + nome);  // NÃO executa ainda
        return nome.toUpperCase();
    });

System.out.println("Pipeline criado, mas nada foi executado ainda!");

// Somente agora a execução acontece
List<String> resultado = pipeline.collect(Collectors.toList());
```

5.4 Operações Intermediárias - Transformações de Dados

Filter - Seleção Condicional Inteligente

`filter()` seleciona elementos que satisfazem uma condição específica, permitindo múltiplos filtros encadeados para refinamento progressivo dos dados.

Java

```java
List<Produto> produtos = Arrays.asList(
    new Produto("Notebook", 2500.00, "Eletrônicos", true),
    new Produto("Mouse", 50.00, "Eletrônicos", true),
    new Produto("Livro Java", 80.00, "Livros", false),
    new Produto("Smartphone", 1200.00, "Eletrônicos", true),
    new Produto("Cadeira", 300.00, "Móveis", true)
);

// Filtros progressivos - cada um refina mais o resultado
List<Produto> produtosFiltrados = produtos.stream()
    .filter(p -> p.isDisponivel())           // Primeiro: apenas disponíveis
    .filter(p -> p.getCategoria().equals("Eletrônicos"))  // Segundo: apenas
eletrônicos
    .filter(p -> p.getPreco() > 100.00)      // Terceiro: apenas acima de R$
```

```
100
    .collect(Collectors.toList());

System.out.println("Produtos eletrônicos disponíveis acima de R$ 100:");
produtosFiltrados.forEach(p -> System.out.println("- " + p.getNome() + ": R$
" + p.getPreco()));
```

Map - Transformação de Elementos

`map()` transforma cada elemento aplicando uma função, criando um novo Stream com elementos do tipo resultante da transformação.

Java

```
List<String> frases = Arrays.asList(
    "programação orientada a objetos",
    "streams em java",
    "padrões de projeto",
    "arquivos e serialização"
);

// Transformações encadeadas
List<String> frasesProcessadas = frases.stream()
    .map(String::toUpperCase)                // String -> String (maiúscula)
    .map(frase -> frase.replace(" ", "_"))   // String -> String (substitui
espaços)
    .map(frase -> "[" + frase + "]")         // String -> String (adiciona
colchetes)
    .collect(Collectors.toList());

// Transformação de tipo - String -> Integer
List<Integer> comprimentos = frases.stream()
    .map(String::length)                     // String -> Integer
    .collect(Collectors.toList());

// Transformação complexa - String -> Objeto customizado
List<FraseInfo> informacoes = frases.stream()
    .map(frase -> new FraseInfo(
        frase,
        frase.length(),
        frase.split(" ").length,
        frase.toUpperCase()
    ))
    .collect(Collectors.toList());
```

Sorted - Ordenação Flexível

`sorted()` ordena elementos usando comparação natural ou Comparators customizados, permitindo ordenações complexas e múltiplos critérios.

Java

```java
List<Funcionario> funcionarios = Arrays.asList(
    new Funcionario("Ana Silva", 5000.00, 28, "TI"),
    new Funcionario("Bruno Costa", 4500.00, 32, "RH"),
    new Funcionario("Carlos Oliveira", 5500.00, 29, "TI"),
    new Funcionario("Diana Santos", 4800.00, 26, "Marketing")
);

// Ordenação natural por nome (se Funcionario implementa Comparable)
List<Funcionario> porNome = funcionarios.stream()
    .sorted()
    .collect(Collectors.toList());

// Ordenação por salário (crescente)
List<Funcionario> porSalario = funcionarios.stream()
    .sorted(Comparator.comparing(Funcionario::getSalario))
    .collect(Collectors.toList());

// Ordenação por salário (decrescente)
List<Funcionario> porSalarioDesc = funcionarios.stream()
    .sorted(Comparator.comparing(Funcionario::getSalario).reversed())
    .collect(Collectors.toList());

// Ordenação complexa: primeiro por departamento, depois por salário
decrescente
List<Funcionario> ordenacaoComplexa = funcionarios.stream()
    .sorted(Comparator.comparing(Funcionario::getDepartamento)
                      .thenComparing(Funcionario::getSalario,
Comparator.reverseOrder()))
    .collect(Collectors.toList());
```

Distinct - Remoção de Duplicatas

`distinct()` remove elementos duplicados baseado no método `equals()`, útil para garantir unicidade em coleções.

Java

```java
List<String> palavrasComDuplicatas = Arrays.asList(
    "java", "stream", "java", "filter", "map", "stream", "collect", "java"
);

// Remove duplicatas mantendo ordem de primeira ocorrência
```

```java
List<String> palavrasUnicas = palavrasComDuplicatas.stream()
    .distinct()
    .collect(Collectors.toList());

// Combinando distinct com outras operações
List<String> palavrasProcessadas = palavrasComDuplicatas.stream()
    .map(String::toUpperCase)        // Transforma para maiúscula
    .distinct()                      // Remove duplicatas
    .sorted()                        // Ordena alfabeticamente
    .collect(Collectors.toList());

// Distinct em objetos customizados (baseado em equals/hashCode)
List<Produto> produtosUnicos = produtos.stream()
    .distinct()  // Remove produtos com mesmo nome/categoria (se equals
implementado)
    .collect(Collectors.toList());
```

Limit e Skip - Controle de Fluxo

`limit()` restringe o número de elementos processados, enquanto `skip()` pula elementos iniciais, permitindo implementar paginação e controle de fluxo.

Java

```java
List<Integer> numeros = Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12,
13, 14, 15);

// Primeiros 5 elementos
List<Integer> primeiros5 = numeros.stream()
    .limit(5)
    .collect(Collectors.toList());

// Elementos do meio (pula 5, pega próximos 5)
List<Integer> meio = numeros.stream()
    .skip(5)
    .limit(5)
    .collect(Collectors.toList());

// Implementação de paginação
public List<Produto> obterPagina(List<Produto> produtos, int numeroPagina,
int tamanhoPagina) {
    return produtos.stream()
        .skip(numeroPagina * tamanhoPagina)  // Pula páginas anteriores
        .limit(tamanhoPagina)                // Limita ao tamanho da página
        .collect(Collectors.toList());
}
```

```java
// Top 3 produtos mais caros
List<Produto> top3Caros = produtos.stream()
    .sorted(Comparator.comparing(Produto::getPreco).reversed())
    .limit(3)
    .collect(Collectors.toList());
```

5.5 Operações Terminais - Produção de Resultados

Collect - Coleta Versátil

`collect()` é a operação terminal mais poderosa, permitindo coletar elementos em diferentes estruturas de dados usando Collectors pré-definidos ou customizados.

Java

```java
List<Funcionario> funcionarios = obterFuncionarios();

// Coleta em diferentes tipos de coleção
List<String> nomes = funcionarios.stream()
    .map(Funcionario::getNome)
    .collect(Collectors.toList());

Set<String> departamentosUnicos = funcionarios.stream()
    .map(Funcionario::getDepartamento)
    .collect(Collectors.toSet());

// Coleta em Map (nome -> salário)
Map<String, Double> nomeSalario = funcionarios.stream()
    .collect(Collectors.toMap(
        Funcionario::getNome,    // Chave: nome
        Funcionario::getSalario  // Valor: salário
    ));

// Agrupamento por departamento
Map<String, List<Funcionario>> porDepartamento = funcionarios.stream()
    .collect(Collectors.groupingBy(Funcionario::getDepartamento));

// Agrupamento com transformação
Map<String, List<String>> nomesPorDepartamento = funcionarios.stream()
    .collect(Collectors.groupingBy(
        Funcionario::getDepartamento,
        Collectors.mapping(Funcionario::getNome, Collectors.toList())
    ));

// Particionamento (divisão em dois grupos)
Map<Boolean, List<Funcionario>> salarioAlto = funcionarios.stream()
    .collect(Collectors.partitioningBy(f -> f.getSalario() > 5000));
```

```java
// Estatísticas agregadas
DoubleSummaryStatistics estatisticas = funcionarios.stream()
    .collect(Collectors.summarizingDouble(Funcionario::getSalario));

System.out.println("Salário médio: " + estatisticas.getAverage());
System.out.println("Salário máximo: " + estatisticas.getMax());
System.out.println("Salário mínimo: " + estatisticas.getMin());
```

Reduce - Redução a Valor Único

`reduce()` combina elementos usando uma operação associativa, permitindo calcular valores agregados customizados.

Java

```java
List<Integer> numeros = Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8, 9, 10);

// Soma usando reduce
Optional<Integer> soma = numeros.stream()
    .reduce((a, b) -> a + b);

// Soma com valor inicial (mais seguro)
Integer somaSegura = numeros.stream()
    .reduce(0, (a, b) -> a + b);

// Produto de todos os números
Optional<Integer> produto = numeros.stream()
    .reduce((a, b) -> a * b);

// Maior valor
Optional<Integer> maior = numeros.stream()
    .reduce(Integer::max);

// Concatenação de strings
List<String> palavras = Arrays.asList("Java", "é", "uma", "linguagem",
"poderosa");
String frase = palavras.stream()
    .reduce("", (a, b) -> a + " " + b)
    .trim();

// Reduce complexo - soma de salários de funcionários de TI
double somaSalariosTI = funcionarios.stream()
    .filter(f -> f.getDepartamento().equals("TI"))
    .map(Funcionario::getSalario)
    .reduce(0.0, Double::sum);
```

ForEach - Execução de Ações

`forEach()` executa uma ação para cada elemento, útil para efeitos colaterais como impressão, logging, ou atualizações.

Java

```java
List<Produto> produtos = obterProdutos();

// Impressão simples
produtos.stream()
    .filter(p -> p.getPreco() > 100)
    .forEach(System.out::println);

// Ação mais complexa
produtos.stream()
    .filter(p -> p.isDisponivel())
    .forEach(produto -> {
        System.out.println("Processando: " + produto.getNome());
        produto.aplicarDesconto(0.1);  // 10% de desconto
        System.out.println("Novo preço: R$ " + produto.getPreco());
    });

// Logging estruturado
funcionarios.stream()
    .filter(f -> f.getSalario() > 8000)
    .forEach(f -> {
        String mensagem = String.format("Funcionário de alto salário: %s
(%s) - R$ %.2f",
            f.getNome(), f.getDepartamento(), f.getSalario());
        logger.info(mensagem);
    });
```

Match Operations - Verificações Booleanas

As operações `anyMatch()`, `allMatch()`, e `noneMatch()` verificam condições sobre elementos do Stream, retornando resultados booleanos.

Java

```java
List<Produto> produtos = obterProdutos();

// Verifica se existe algum produto caro
boolean temProdutoCaro = produtos.stream()
    .anyMatch(p -> p.getPreco() > 1000);

// Verifica se todos os produtos estão disponíveis
```

```java
boolean todosDisponiveis = produtos.stream()
    .allMatch(Produto::isDisponivel);

// Verifica se nenhum produto está em falta
boolean nenhumEmFalta = produtos.stream()
    .noneMatch(p -> p.getEstoque() == 0);

// Verificações mais complexas
boolean temEletronicoCaro = produtos.stream()
    .filter(p -> p.getCategoria().equals("Eletrônicos"))
    .anyMatch(p -> p.getPreco() > 2000);

// Validação de regras de negócio
boolean funcionariosValidados = funcionarios.stream()
    .allMatch(f -> f.getSalario() > 0 && f.getNome() != null &&
!f.getNome().isEmpty());
```

5.6 Streams Especializados - Otimização para Primitivos

IntStream, LongStream, DoubleStream

Java fornece streams especializados para tipos primitivos que evitam boxing/unboxing e oferecem operações matemáticas específicas.

Java

```java
// Criação de IntStream
IntStream numeros = IntStream.range(1, 11);  // 1 a 10
IntStream numerosInclusivo = IntStream.rangeClosed(1, 10);  // 1 a 10
inclusivo

// Operações matemáticas específicas
List<String> palavras = Arrays.asList("Java", "Stream", "Programming",
"Functional");

int somaComprimentos = palavras.stream()
    .mapToInt(String::length)    // Stream<String> -> IntStream
    .sum();                      // Soma direta sem Optional

double mediaComprimentos = palavras.stream()
    .mapToInt(String::length)
    .average()                   // OptionalDouble
    .orElse(0.0);

int maiorComprimento = palavras.stream()
    .mapToInt(String::length)
    .max()                       // OptionalInt
```

```java
    .orElse(0);

// Estatísticas completas
IntSummaryStatistics stats = palavras.stream()
    .mapToInt(String::length)
    .summaryStatistics();

System.out.println("Soma: " + stats.getSum());
System.out.println("Média: " + stats.getAverage());
System.out.println("Máximo: " + stats.getMax());
System.out.println("Mínimo: " + stats.getMin());
System.out.println("Contagem: " + stats.getCount());

// Geração de sequências
IntStream.iterate(0, n -> n + 2)  // Números pares
    .limit(10)
    .forEach(System.out::println);

// Números aleatórios
new Random().ints(5, 1, 101)     // 5 números entre 1 e 100
    .forEach(System.out::println);
```

5.7 Expressões Lambda e Method References

Sintaxe Lambda Progressiva

Java

```java
// Evolução da sintaxe lambda

// 1. Classe anônima tradicional (verbosa)
produtos.stream().filter(new Predicate<Produto>() {
    @Override
    public boolean test(Produto p) {
        return p.getPreco() > 100;
    }
});

// 2. Lambda com tipos explícitos
produtos.stream().filter((Produto p) -> {
    return p.getPreco() > 100;
});

// 3. Lambda com inferência de tipos
produtos.stream().filter(p -> {
    return p.getPreco() > 100;
});
```

```java
// 4. Lambda com expressão simples
produtos.stream().filter(p -> p.getPreco() > 100);

// 5. Method reference (mais conciso quando possível)
produtos.stream().map(Produto::getNome);
```

Tipos de Method References

Java

```java
// Método estático
Function<String, Integer> parseInteger = Integer::parseInt;
// Equivale a: s -> Integer.parseInt(s)

// Método de instância de objeto específico
Predicate<String> isEmpty = String::isEmpty;
// Equivale a: s -> s.isEmpty()

// Método de instância de tipo arbitrário
Function<String, String> toUpper = String::toUpperCase;
// Equivale a: s -> s.toUpperCase()

// Construtor
Supplier<ArrayList<String>> listSupplier = ArrayList::new;
// Equivale a: () -> new ArrayList<>()

// Uso prático em streams
List<String> numeros = Arrays.asList("1", "2", "3", "4", "5");

List<Integer> inteiros = numeros.stream()
    .map(Integer::parseInt)      // Method reference para método estático
    .collect(Collectors.toList());

List<String> maiusculas = Arrays.asList("java", "stream", "lambda")
    .stream()
    .map(String::toUpperCase)    // Method reference para método de instância
    .collect(Collectors.toList());
```

5.8 Padrões Avançados e Boas Práticas

FlatMap - Achatamento de Estruturas

Java

```java
// Problema: List<List<String>> -> List<String>
List<List<String>> listaDeListas = Arrays.asList(
    Arrays.asList("Java", "Python"),
    Arrays.asList("JavaScript", "TypeScript", "C#"),
    Arrays.asList("Go", "Rust")
);

// Solução com flatMap
List<String> todasLinguagens = listaDeListas.stream()
    .flatMap(List::stream)       // Achata List<List<String>> em
Stream<String>
    .collect(Collectors.toList());

// Exemplo mais complexo: extrair palavras de frases
List<String> frases = Arrays.asList(
    "Java é uma linguagem poderosa",
    "Streams facilitam processamento de dados",
    "Programação funcional é elegante"
);

List<String> todasPalavras = frases.stream()
    .flatMap(frase -> Arrays.stream(frase.split(" ")))  // Frase ->
Stream<Palavra>
    .map(String::toLowerCase)
    .distinct()
    .sorted()
    .collect(Collectors.toList());

// FlatMap com objetos complexos
List<Departamento> departamentos = obterDepartamentos();

List<Funcionario> todosFuncionarios = departamentos.stream()
    .flatMap(dept -> dept.getFuncionarios().stream())  // Dept ->
Stream<Funcionario>
    .collect(Collectors.toList());
```

Collectors Avançados

Java

```java
// Joining - concatenação com delimitadores
String nomesJuntos = funcionarios.stream()
    .map(Funcionario::getNome)
    .collect(Collectors.joining(", ", "[", "]"));

// Agrupamento com contagem
```

```java
Map<String, Long> funcionariosPorDepartamento = funcionarios.stream()
    .collect(Collectors.groupingBy(
        Funcionario::getDepartamento,
        Collectors.counting()
    ));

// Agrupamento com soma
Map<String, Double> salarioTotalPorDepartamento = funcionarios.stream()
    .collect(Collectors.groupingBy(
        Funcionario::getDepartamento,
        Collectors.summingDouble(Funcionario::getSalario)
    ));

// Agrupamento multinível
Map<String, Map<Boolean, List<Funcionario>>> agrupamentoComplexo =
funcionarios.stream()
    .collect(Collectors.groupingBy(
        Funcionario::getDepartamento,
        Collectors.partitioningBy(f -> f.getSalario() > 5000)
    ));
```

5.9 Casos de Uso Práticos para Prova

Processamento de Dados de Vendas

Java

```java
public class RelatorioVendas {

    public void gerarRelatorio(List<Venda> vendas) {
        // Top 5 produtos mais vendidos
        List<String> top5Produtos = vendas.stream()
            .collect(Collectors.groupingBy(
                Venda::getProduto,
                Collectors.summingInt(Venda::getQuantidade)
            ))
            .entrySet().stream()
            .sorted(Map.Entry.<String, Integer>comparingByValue().reversed())
            .limit(5)
            .map(Map.Entry::getKey)
            .collect(Collectors.toList());

        // Vendas por mês
        Map<String, Double> vendasPorMes = vendas.stream()
            .collect(Collectors.groupingBy(
                v -> v.getData().getMonth().toString(),
                Collectors.summingDouble(Venda::getValorTotal)
```

```
            ));

        // Vendedores com performance acima da média
        double mediaVendas = vendas.stream()
            .mapToDouble(Venda::getValorTotal)
            .average()
            .orElse(0.0);

        List<String> vendedoresDestaque = vendas.stream()
            .filter(v -> v.getValorTotal() > mediaVendas)
            .map(Venda::getVendedor)
            .distinct()
            .sorted()
            .collect(Collectors.toList());
    }
}
```

6. PADRÕES ARQUITETURAIS E DE PROJETO

6.1 MVC (Model-View-Controller) - SEPARAÇÃO DE RESPONSABILIDADES

Conceito Fundamental

O padrão MVC é uma arquitetura que organiza aplicações separando responsabilidades em três componentes distintos e interconectados. Esta separação não é apenas organizacional, mas representa uma estratégia fundamental para criar sistemas maintíveis, testáveis, e flexíveis. Cada componente tem uma responsabilidade específica e bem definida, promovendo baixo acoplamento entre camadas e alta coesão dentro de cada camada.

Filosofia da Separação

Princípio da Responsabilidade Única: Cada componente do MVC tem uma única razão para mudar - Model muda quando regras de negócio mudam, View muda quando interface muda, Controller muda quando fluxo de controle muda.

Baixo Acoplamento: Componentes dependem de abstrações, não de implementações concretas, permitindo que sejam modificados independentemente.

Alta Coesão: Elementos dentro de cada componente trabalham juntos para uma única responsabilidade bem definida.

MODEL - Coração dos Dados e Regras de Negócio

O Model encapsula dados e lógica de negócio, sendo responsável por manter a integridade dos dados e implementar as regras que governam o domínio da aplicação. É o componente mais importante porque contém a essência do que a aplicação faz.

Java

```java
public class ContaBancaria {  // MODEL - dados + regras de negócio
    private String numero;
    private String titular;
    private double saldo;
    private List<Transacao> historico;
    private boolean ativa;

    public ContaBancaria(String numero, String titular, double saldoInicial)
{
        this.numero = numero;
        this.titular = titular;
        this.saldo = saldoInicial;
        this.historico = new ArrayList<>();
        this.ativa = true;

        // Registra transação inicial
        registrarTransacao("ABERTURA", saldoInicial, "Abertura da conta");
    }

    // REGRA DE NEGÓCIO: Saque com validações
    public boolean sacar(double valor, String descricao) {
        // Validação 1: Conta deve estar ativa
        if (!ativa) {
            return false;
        }

        // Validação 2: Valor deve ser positivo
        if (valor <= 0) {
            return false;
        }

        // Validação 3: Saldo deve ser suficiente
        if (valor > saldo) {
            return false;
        }

        // Executa operação
        saldo -= valor;
        registrarTransacao("SAQUE", -valor, descricao);

        return true;  // Operação bem-sucedida
```

```
    }

    // REGRA DE NEGÓCIO: Depósito com validações
    public boolean depositar(double valor, String descricao) {
        if (!ativa || valor <= 0) {
            return false;
        }

        saldo += valor;
        registrarTransacao("DEPOSITO", valor, descricao);

        return true;
    }

    // REGRA DE NEGÓCIO: Transferência entre contas
    public boolean transferir(ContaBancaria contaDestino, double valor,
String descricao) {
        if (this.sacar(valor, "Transferência para " +
contaDestino.getNumero())) {
            if (contaDestino.depositar(valor, "Transferência de " +
this.numero)) {
                return true;
            } else {
                // Reverte saque se depósito falhar
                this.depositar(valor, "Reversão de transferência");
                return false;
            }
        }
        return false;
    }

    private void registrarTransacao(String tipo, double valor, String
descricao) {
        Transacao transacao = new Transacao(
            LocalDateTime.now(),
            tipo,
            valor,
            saldo,  // Saldo após transação
            descricao
        );
        historico.add(transacao);
    }

    // Getters - Model NÃO conhece View ou Controller
    public String getNumero() { return numero; }
    public String getTitular() { return titular; }
    public double getSaldo() { return saldo; }
    public List<Transacao> getHistorico() { return new ArrayList<>
```

```
(historico); }
    public boolean isAtiva() { return ativa; }
}
```

VIEW - Interface com o Mundo Exterior

A View é responsável pela apresentação dos dados ao usuário e pela captura da entrada do usuário. Deve ser uma camada "burra" que não contém lógica de negócio - apenas traduz dados em representação visual e eventos do usuário em chamadas para o Controller.

Java

```
public class ContaBancariaView {  // VIEW - apenas interface
    private Scanner scanner;

    public ContaBancariaView() {
        this.scanner = new Scanner(System.in);
    }

    // APRESENTA dados do Model
    public void exibirDadosConta(ContaBancaria conta) {
        System.out.println("\n=== DADOS DA CONTA ===");
        System.out.println("Número: " + conta.getNumero());
        System.out.println("Titular: " + conta.getTitular());
        System.out.printf("Saldo: R$ %.2f%n", conta.getSaldo());
        System.out.println("Status: " + (conta.isAtiva() ? "ATIVA" :
"INATIVA"));
    }

    // APRESENTA histórico de transações
    public void exibirHistorico(List<Transacao> transacoes) {
        System.out.println("\n=== HISTÓRICO DE TRANSAÇÕES ===");
        if (transacoes.isEmpty()) {
            System.out.println("Nenhuma transação encontrada.");
            return;
        }

        for (Transacao t : transacoes) {
            System.out.printf("%s | %s | R$ %+.2f | Saldo: R$ %.2f | %s%n",

t.getDataHora().format(DateTimeFormatter.ofPattern("dd/MM/yyyy HH:mm")),
                t.getTipo(),
                t.getValor(),
                t.getSaldoApos(),
                t.getDescricao()
            );
        }
```

```
    }

    // CAPTURA entrada do usuário
    public int exibirMenuPrincipal() {
        System.out.println("\n=== MENU PRINCIPAL ===");
        System.out.println("1. Ver dados da conta");
        System.out.println("2. Sacar");
        System.out.println("3. Depositar");
        System.out.println("4. Ver histórico");
        System.out.println("5. Transferir");
        System.out.println("0. Sair");
        System.out.print("Escolha uma opção: ");

        return scanner.nextInt();
    }

    public double pedirValor(String operacao) {
        System.out.printf("Digite o valor para %s: R$ ", operacao);
        return scanner.nextDouble();
    }

    public String pedirDescricao() {
        scanner.nextLine();  // Limpa buffer
        System.out.print("Digite uma descrição (opcional): ");
        String descricao = scanner.nextLine();
        return descricao.isEmpty() ? "Sem descrição" : descricao;
    }

    public String pedirContaDestino() {
        System.out.print("Digite o número da conta de destino: ");
        return scanner.next();
    }

    // APRESENTA mensagens de feedback
    public void exibirMensagem(String mensagem) {
        System.out.println("\n>>> " + mensagem);
    }

    public void exibirErro(String erro) {
        System.err.println("\n!!! ERRO: " + erro);
    }

    // View NÃO contém lógica de negócio - apenas apresenta e captura
}
```

CONTROLLER - Maestro da Orquestração

O Controller coordena interações entre Model e View, controlando o fluxo da aplicação. Recebe eventos da View, interpreta esses eventos, invoca operações no Model, e atualiza a View com os resultados.

Java

```java
public class ContaBancariaController {  // CONTROLLER - coordenação
    private ContaBancaria model;        // Referência ao MODEL
    private ContaBancariaView view;     // Referência à VIEW
    private Map<String, ContaBancaria> contas;  // Simulação de banco de
dados

    public ContaBancariaController(ContaBancaria conta, ContaBancariaView
view) {
        this.model = conta;
        this.view = view;
        this.contas = new HashMap<>();

        // Registra conta no "banco de dados"
        contas.put(conta.getNumero(), conta);
    }

    // COORDENA fluxo principal da aplicação
    public void executar() {
        view.exibirMensagem("Bem-vindo ao Sistema Bancário!");

        boolean continuar = true;
        while (continuar) {
            try {
                int opcao = view.exibirMenuPrincipal();

                switch (opcao) {
                    case 1:
                        processarConsultaDados();
                        break;
                    case 2:
                        processarSaque();
                        break;
                    case 3:
                        processarDeposito();
                        break;
                    case 4:
                        processarConsultaHistorico();
                        break;
                    case 5:
                        processarTransferencia();
                        break;
```

```
                    case 0:
                        continuar = false;
                        view.exibirMensagem("Obrigado por usar nosso
sistema!");
                        break;
                    default:
                        view.exibirErro("Opção inválida!");
                }
            } catch (Exception e) {
                view.exibirErro("Erro inesperado: " + e.getMessage());
            }
        }
    }

    // COORDENA operação de saque
    private void processarSaque() {
        view.exibirDadosConta(model);           // View apresenta dados do
Model

        double valor = view.pedirValor("saque"); // View captura entrada
        String descricao = view.pedirDescricao();

        boolean sucesso = model.sacar(valor, descricao);  // Model aplica
regra de negócio

        if (sucesso) {
            view.exibirMensagem("Saque realizado com sucesso!");
            view.exibirDadosConta(model);       // View apresenta estado
atualizado
        } else {
            view.exibirErro("Não foi possível realizar o saque. Verifique
saldo e dados.");
        }
    }

    // COORDENA operação de depósito
    private void processarDeposito() {
        double valor = view.pedirValor("depósito");
        String descricao = view.pedirDescricao();

        boolean sucesso = model.depositar(valor, descricao);

        if (sucesso) {
            view.exibirMensagem("Depósito realizado com sucesso!");
            view.exibirDadosConta(model);
        } else {
            view.exibirErro("Não foi possível realizar o depósito.");
        }
```

```
    }

    // COORDENA operação de transferência
    private void processarTransferencia() {
        view.exibirDadosConta(model);

        String numeroDestino = view.pedirContaDestino();
        ContaBancaria contaDestino = contas.get(numeroDestino);

        if (contaDestino == null) {
            view.exibirErro("Conta de destino não encontrada!");
            return;
        }

        double valor = view.pedirValor("transferência");
        String descricao = view.pedirDescricao();

        boolean sucesso = model.transferir(contaDestino, valor, descricao);

        if (sucesso) {
            view.exibirMensagem("Transferência realizada com sucesso!");
            view.exibirDadosConta(model);
        } else {
            view.exibirErro("Não foi possível realizar a transferência.");
        }
    }

    private void processarConsultaDados() {
        view.exibirDadosConta(model);
    }

    private void processarConsultaHistorico() {
        List<Transacao> historico = model.getHistorico();
        view.exibirHistorico(historico);
    }

    // Controller COORDENA mas não implementa regras de negócio nem
apresentação
}
```

Vantagens do Padrão MVC

Separação Clara de Responsabilidades: Cada componente tem uma função específica, facilitando manutenção e compreensão.

Reutilização de Componentes: Models podem ser usados com diferentes Views, Views podem apresentar diferentes Models.

Testabilidade Independente: Cada componente pode ser testado isoladamente usando mocks ou stubs.

Flexibilidade de Interface: Múltiplas interfaces (console, web, mobile) podem usar o mesmo Model.

Manutenibilidade: Mudanças em um componente têm impacto mínimo nos outros.

6.2 SINGLETON - GARANTIA DE INSTÂNCIA ÚNICA

Conceito Fundamental

O padrão Singleton garante que uma classe tenha exatamente uma instância durante toda a execução do programa e fornece um ponto de acesso global a essa instância. Este padrão é útil quando você precisa coordenar ações através do sistema usando um objeto único, como configurações globais, sistemas de logging, ou gerenciadores de recursos compartilhados.

Motivação e Casos de Uso

Recursos Únicos: Alguns recursos do sistema devem ser únicos por natureza - como um gerenciador de impressora, pool de conexões, ou sistema de cache.

Coordenação Global: Quando múltiplas partes do sistema precisam acessar e modificar um estado compartilhado de forma coordenada.

Controle de Acesso: Quando você quer controlar rigorosamente como e quando uma instância é criada e acessada.

Implementação Básica Thread-Safe

Java

```java
public class ConfiguracaoSistema {
    // Instância única - volatile garante visibilidade entre threads
    private static volatile ConfiguracaoSistema instancia;

    // Dados da configuração
    private Properties configuracoes;
    private String ambiente;
    private Date dataInicializacao;
    private Map<String, String> parametros;

    // Construtor PRIVADO - impede criação externa
    private ConfiguracaoSistema() {
        System.out.println("🔧 Inicializando configuração do sistema...");
```

```
        // Inicialização custosa simulada
        try {
            Thread.sleep(100);  // Simula carregamento de arquivo
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
        }

        this.configuracoes = new Properties();
        this.parametros = new HashMap<>();
        this.dataInicializacao = new Date();

        // Carrega configurações padrão
        carregarConfiguracoesPadrao();

        System.out.println("✅ Configuração do sistema inicializada!");
    }

    // Método para obter instância única - THREAD-SAFE
    public static ConfiguracaoSistema getInstance() {
        // Double-checked locking para performance
        if (instancia == null) {                    // Primeira verificação
(sem lock)
            synchronized (ConfiguracaoSistema.class) {  // Sincronização
apenas quando necessário
                if (instancia == null) {            // Segunda verificação
(com lock)
                    instancia = new ConfiguracaoSistema();
                }
            }
        }
        return instancia;
    }

    private void carregarConfiguracoesPadrao() {
        parametros.put("max_conexoes", "100");
        parametros.put("timeout", "30000");
        parametros.put("debug", "false");
        parametros.put("log_level", "INFO");
        ambiente = "DESENVOLVIMENTO";
    }

    // Métodos de negócio
    public String getParametro(String chave) {
        return parametros.get(chave);
    }

    public void setParametro(String chave, String valor) {
        parametros.put(chave, valor);
```

```
        System.out.println("📝 Parâmetro atualizado: " + chave + " = " +
valor);
    }

    public String getAmbiente() {
        return ambiente;
    }

    public void setAmbiente(String ambiente) {
        this.ambiente = ambiente;
        System.out.println("🌍 Ambiente alterado para: " + ambiente);
    }

    public Date getDataInicializacao() {
        return new Date(dataInicializacao.getTime());  // Cópia defensiva
    }

    public void exibirStatus() {
        System.out.println("\n📊 STATUS DA CONFIGURAÇÃO:");
        System.out.println("   Ambiente: " + ambiente);
        System.out.println("   Inicializada em: " + dataInicializacao);
        System.out.println("   Parâmetros configurados: " +
parametros.size());
        parametros.forEach((chave, valor) ->
            System.out.println("     " + chave + " = " + valor));
    }
}
```

Singleton com Enum (Mais Seguro)

Java

```
public enum LoggerSistema {
    INSTANCE;  // A única instância

    // Atributos do singleton
    private List<String> logs;
    private String nomeArquivo;
    private DateTimeFormatter formatoData;

    // Construtor do enum (chamado apenas uma vez automaticamente)
    LoggerSistema() {
        this.logs = new ArrayList<>();
        this.nomeArquivo = "sistema.log";
        this.formatoData = DateTimeFormatter.ofPattern("yyyy-MM-dd
HH:mm:ss");
```

```
        adicionarLog("SISTEMA", "Logger inicializado");
        System.out.println("📝 Logger do sistema inicializado!");
    }

    // Métodos de negócio
    public void adicionarLog(String nivel, String mensagem) {
        String timestamp = LocalDateTime.now().format(formatoData);
        String logCompleto = String.format("[%s] %s: %s", timestamp, nivel,
mensagem);

        logs.add(logCompleto);
        System.out.println(logCompleto);  // Também exibe no console

        // Em implementação real, escreveria em arquivo
    }

    public void info(String mensagem) {
        adicionarLog("INFO", mensagem);
    }

    public void erro(String mensagem) {
        adicionarLog("ERRO", mensagem);
    }

    public void debug(String mensagem) {
        adicionarLog("DEBUG", mensagem);
    }

    public void exibirTodosLogs() {
        System.out.println("\n📜 TODOS OS LOGS:");
        logs.forEach(log -> System.out.println("   " + log));
    }

    public int getTotalLogs() {
        return logs.size();
    }

    public List<String> getLogsRecentes(int quantidade) {
        int inicio = Math.max(0, logs.size() - quantidade);
        return new ArrayList<>(logs.subList(inicio, logs.size()));
    }
}
```

Teste e Demonstração do Singleton

Java

```java
public class TesteSingleton {

    public static void main(String[] args) {
        System.out.println("=== TESTE DO PADRÃO SINGLETON ===\n");

        // Teste 1: Verificar instância única
        testeInstanciaUnica();

        // Teste 2: Teste em ambiente multi-thread
        testeMultiThread();

        // Teste 3: Uso prático do Logger Enum
        testeLoggerEnum();
    }

    private static void testeInstanciaUnica() {
        System.out.println("1. 🧪 TESTE DE INSTÂNCIA ÚNICA:");

        // Obtém duas "instâncias"
        ConfiguracaoSistema config1 = ConfiguracaoSistema.getInstance();
        ConfiguracaoSistema config2 = ConfiguracaoSistema.getInstance();

        // Verifica se são a mesma instância
        System.out.println("   config1 == config2: " + (config1 == config2));
        System.out.println("   HashCode config1: " + config1.hashCode());
        System.out.println("   HashCode config2: " + config2.hashCode());

        // Testa compartilhamento de estado
        config1.setParametro("teste", "valor1");
        System.out.println("   Parâmetro via config2: " +
config2.getParametro("teste"));

        config1.exibirStatus();
    }

    private static void testeMultiThread() {
        System.out.println("\n2. 🧵 TESTE MULTI-THREAD:");

        // Cria múltiplas threads para testar thread safety
        Thread[] threads = new Thread[5];

        for (int i = 0; i < threads.length; i++) {
            final int threadId = i;
            threads[i] = new Thread(() -> {
                ConfiguracaoSistema config =
ConfiguracaoSistema.getInstance();
                config.setParametro("thread_" + threadId, "valor_" +
```

```
threadId);
                System.out.println("   Thread " + threadId + " obteve
instância: " +
                                   config.hashCode());
            }, "TestThread-" + i);
        }

        // Inicia todas as threads
        for (Thread thread : threads) {
            thread.start();
        }

        // Aguarda todas terminarem
        for (Thread thread : threads) {
            try {
                thread.join();
            } catch (InterruptedException e) {
                Thread.currentThread().interrupt();
            }
        }

        // Verifica estado final
        ConfiguracaoSistema.getInstance().exibirStatus();
    }

    private static void testeLoggerEnum() {
        System.out.println("\n3. 📝 TESTE LOGGER ENUM:");

        // Acesso direto à instância
        LoggerSistema logger = LoggerSistema.INSTANCE;

        // Testa diferentes tipos de log
        logger.info("Sistema iniciado com sucesso");
        logger.debug("Carregando configurações...");
        logger.erro("Falha na conexão com banco de dados");
        logger.info("Tentando reconectar...");
        logger.info("Conexão reestabelecida");

        // Verifica se é realmente singleton
        LoggerSistema logger2 = LoggerSistema.INSTANCE;
        System.out.println("   logger == logger2: " + (logger == logger2));

        // Exibe estatísticas
        System.out.println("   Total de logs: " + logger.getTotalLogs());

        // Exibe logs recentes
        System.out.println("   Últimos 3 logs:");
        logger.getLogsRecentes(3).forEach(log ->
```

```
            System.out.println("     " + log));
    }
}
```

Vantagens e Desvantagens do Singleton

Vantagens:

- Controle rigoroso sobre instanciação
- Economia de memória (apenas uma instância)
- Acesso global coordenado
- Inicialização lazy (sob demanda)

Desvantagens:

- Pode dificultar testes (estado global)
- Viola princípio da responsabilidade única
- Pode criar dependências ocultas
- Problemas de concorrência se mal implementado

6.3 STRATEGY - ALGORITMOS INTERCAMBIÁVEIS

Conceito Fundamental

O padrão Strategy define uma família de algoritmos, encapsula cada um deles, e os torna intercambiáveis. Este padrão permite que o algoritmo varie independentemente dos clientes que o usam, promovendo flexibilidade, reutilização, e facilidade de manutenção.

Aplicação Específica: Estratégias de Persistência

Para a sua prova, o Strategy será aplicado especificamente para encapsular diferentes métodos de persistência (texto, binário, JSON), permitindo trocar entre eles em tempo de execução.

Java

```
// INTERFACE STRATEGY - Define contrato para persistência
public interface EstrategiaPersistencia {
    /**
     * Salva dados usando a estratégia específica
     * @param dados Dados a serem salvos
     * @param nomeArquivo Nome do arquivo de destino
     * @return true se salvou com sucesso, false caso contrário
     */
```

```
    boolean salvar(Object dados, String nomeArquivo);

    /**
     * Carrega dados usando a estratégia específica
     * @param nomeArquivo Nome do arquivo de origem
     * @param tipoClasse Classe dos dados a serem carregados
     * @return Dados carregados ou null se houver erro
     */
    <T> T carregar(String nomeArquivo, Class<T> tipoClasse);

    /**
     * Retorna nome descritivo da estratégia
     */
    String getNomeEstrategia();

    /**
     * Retorna extensão de arquivo recomendada
     */
    String getExtensaoArquivo();
}
```

Estratégia Concreta: Persistência em Texto

Java

```
public class EstrategiaTexto implements EstrategiaPersistencia {

    @Override
    public boolean salvar(Object dados, String nomeArquivo) {
        try (BufferedWriter writer = new BufferedWriter(new
FileWriter(nomeArquivo))) {

            // Converte objeto para representação textual
            if (dados instanceof List) {
                List<?> lista = (List<?>) dados;
                writer.write("LISTA_DADOS_TEXTO_V1.0");
                writer.newLine();
                writer.write("TOTAL_ITENS=" + lista.size());
                writer.newLine();
                writer.newLine();

                for (int i = 0; i < lista.size(); i++) {
                    writer.write("ITEM_" + i + "=" +
lista.get(i).toString());
                    writer.newLine();
                }
            } else {
```

```
                writer.write("OBJETO_TEXTO_V1.0");
                writer.newLine();
                writer.write("DADOS=" + dados.toString());
                writer.newLine();
            }

            System.out.println("📄 Dados salvos em formato texto: " +
nomeArquivo);
            return true;

        } catch (IOException e) {
            System.err.println("❌ Erro ao salvar em texto: " +
e.getMessage());
            return false;
        }
    }

    @Override
    @SuppressWarnings("unchecked")
    public <T> T carregar(String nomeArquivo, Class<T> tipoClasse) {
        try (BufferedReader reader = new BufferedReader(new
FileReader(nomeArquivo))) {

            String cabecalho = reader.readLine();
            if (cabecalho == null) {
                return null;
            }

            if (cabecalho.equals("LISTA_DADOS_TEXTO_V1.0")) {
                // Carrega lista
                String totalLine = reader.readLine();
                int total = Integer.parseInt(totalLine.split("=")[1]);
                reader.readLine(); // Linha em branco

                List<String> lista = new ArrayList<>();
                for (int i = 0; i < total; i++) {
                    String linha = reader.readLine();
                    String valor = linha.split("=", 2)[1];
                    lista.add(valor);
                }

                System.out.println("📄 Lista carregada do texto: " + total +
" itens");
                return (T) lista;

            } else if (cabecalho.equals("OBJETO_TEXTO_V1.0")) {
                // Carrega objeto simples
                String dadosLine = reader.readLine();
```

```
                String dados = dadosLine.split("=", 2)[1];

                System.out.println("📄 Objeto carregado do texto: " + dados);
                return (T) dados;
            }

        } catch (IOException | NumberFormatException e) {
            System.err.println("❌ Erro ao carregar texto: " +
e.getMessage());
        }

        return null;
    }

    @Override
    public String getNomeEstrategia() {
        return "Persistência em Texto";
    }

    @Override
    public String getExtensaoArquivo() {
        return ".txt";
    }
}
```

Estratégia Concreta: Persistência Binária

Java

```
public class EstrategiaBinario implements EstrategiaPersistencia {

    @Override
    public boolean salvar(Object dados, String nomeArquivo) {
        try (DataOutputStream dos = new DataOutputStream(new
FileOutputStream(nomeArquivo))) {

            // Escreve cabeçalho binário
            dos.writeUTF("DADOS_BINARIO_V1.0");
            dos.writeLong(System.currentTimeMillis());

            if (dados instanceof List) {
                List<?> lista = (List<?>) dados;
                dos.writeUTF("LISTA");
                dos.writeInt(lista.size());

                for (Object item : lista) {
                    dos.writeUTF(item.toString());
```

```
                }
            } else {
                dos.writeUTF("OBJETO");
                dos.writeUTF(dados.toString());
            }

            System.out.println("💾 Dados salvos em formato binário: " +
nomeArquivo);
            return true;

        } catch (IOException e) {
            System.err.println("❌ Erro ao salvar binário: " +
e.getMessage());
            return false;
        }
    }

    @Override
    @SuppressWarnings("unchecked")
    public <T> T carregar(String nomeArquivo, Class<T> tipoClasse) {
        try (DataInputStream dis = new DataInputStream(new
FileInputStream(nomeArquivo))) {

            // Lê e valida cabeçalho
            String cabecalho = dis.readUTF();
            if (!cabecalho.equals("DADOS_BINARIO_V1.0")) {
                System.err.println("❌ Formato binário inválido");
                return null;
            }

            long timestamp = dis.readLong();
            String tipo = dis.readUTF();

            if (tipo.equals("LISTA")) {
                int tamanho = dis.readInt();
                List<String> lista = new ArrayList<>();

                for (int i = 0; i < tamanho; i++) {
                    lista.add(dis.readUTF());
                }

                System.out.println("💾 Lista carregada do binário: " +
tamanho + " itens");
                System.out.println("   Criada em: " + new Date(timestamp));
                return (T) lista;

            } else if (tipo.equals("OBJETO")) {
                String dados = dis.readUTF();
```

```
                System.out.println("💾 Objeto carregado do binário: " +
dados);
                return (T) dados;
            }

        } catch (IOException e) {
            System.err.println("❌ Erro ao carregar binário: " +
e.getMessage());
        }

        return null;
    }

    @Override
    public String getNomeEstrategia() {
        return "Persistência Binária";
    }

    @Override
    public String getExtensaoArquivo() {
        return ".dat";
    }
}
```

Estratégia Concreta: Persistência JSON

Java

```
public class EstrategiaJSON implements EstrategiaPersistencia {

    @Override
    public boolean salvar(Object dados, String nomeArquivo) {
        try (BufferedWriter writer = new BufferedWriter(new
FileWriter(nomeArquivo))) {

            StringBuilder json = new StringBuilder();
            json.append("{\n");
            json.append("  \"formato\": \"DADOS_JSON_V1.0\",\n");
            json.append("  \"timestamp\":
\"").append(LocalDateTime.now()).append("\",\n");

            if (dados instanceof List) {
                List<?> lista = (List<?>) dados;
                json.append("  \"tipo\": \"lista\",\n");
                json.append("  \"dados\": [\n");

                for (int i = 0; i < lista.size(); i++) {
```

```
                    json.append("
\"").append(lista.get(i).toString()).append("\"");
                    if (i < lista.size() - 1) {
                        json.append(",");
                    }
                    json.append("\n");
                }

                json.append("  ]\n");
            } else {
                json.append("  \"tipo\": \"objeto\",\n");
                json.append("  \"dados\":
\"").append(dados.toString()).append("\"\n");
            }

            json.append("}");

            writer.write(json.toString());
            System.out.println("🌐 Dados salvos em formato JSON: " +
nomeArquivo);
            return true;

        } catch (IOException e) {
            System.err.println("❌ Erro ao salvar JSON: " + e.getMessage());
            return false;
        }
    }

    @Override
    @SuppressWarnings("unchecked")
    public <T> T carregar(String nomeArquivo, Class<T> tipoClasse) {
        try (BufferedReader reader = new BufferedReader(new
FileReader(nomeArquivo))) {

            StringBuilder jsonContent = new StringBuilder();
            String linha;
            while ((linha = reader.readLine()) != null) {
                jsonContent.append(linha).append("\n");
            }

            String json = jsonContent.toString();

            // Parsing manual simples (em produção, use biblioteca)
            if (json.contains("\"tipo\": \"lista\"")) {
                List<String> lista = new ArrayList<>();

                // Extrai itens do array JSON (parsing simplificado)
                String dadosSection =
```

```
json.substring(json.indexOf("\"dados\": [") + 10);
                dadosSection = dadosSection.substring(0,
dadosSection.indexOf("]"));

                String[] itens = dadosSection.split(",");
                for (String item : itens) {
                    String valor = item.trim().replaceAll("\"", "");
                    if (!valor.isEmpty()) {
                        lista.add(valor);
                    }
                }

                System.out.println("🌐 Lista carregada do JSON: " +
lista.size() + " itens");
                return (T) lista;

            } else if (json.contains("\"tipo\": \"objeto\"")) {
                // Extrai valor do objeto
                String dadosLine = json.substring(json.indexOf("\"dados\":
\"") + 10);
                String dados = dadosLine.substring(0,
dadosLine.indexOf("\""));

                System.out.println("🌐 Objeto carregado do JSON: " + dados);
                return (T) dados;
            }

        } catch (IOException e) {
            System.err.println("❌ Erro ao carregar JSON: " +
e.getMessage());
        }

        return null;
    }

    @Override
    public String getNomeEstrategia() {
        return "Persistência JSON";
    }

    @Override
    public String getExtensaoArquivo() {
        return ".json";
    }
}
```

Contexto: Gerenciador de Persistência

Java

```java
public class GerenciadorPersistencia {
    private EstrategiaPersistencia estrategia;
    private String diretorioBase;

    public GerenciadorPersistencia() {
        this.estrategia = new EstrategiaTexto();  // Estratégia padrão
        this.diretorioBase = "./dados/";
    }

    public GerenciadorPersistencia(EstrategiaPersistencia estrategia) {
        this.estrategia = estrategia;
        this.diretorioBase = "./dados/";
    }

    // MÉTODO PRINCIPAL - Troca estratégia em tempo de execução
    public void setEstrategia(EstrategiaPersistencia novaEstrategia) {
        String estrategiaAnterior = this.estrategia.getNomeEstrategia();
        this.estrategia = novaEstrategia;

        System.out.println("🔄 Estratégia alterada:");
        System.out.println("   De: " + estrategiaAnterior);
        System.out.println("   Para: " + novaEstrategia.getNomeEstrategia());
    }

    public boolean salvarDados(Object dados, String nomeBase) {
        String nomeCompleto = diretorioBase + nomeBase +
estrategia.getExtensaoArquivo();

        System.out.println("💾 Salvando com " +
estrategia.getNomeEstrategia());
        return estrategia.salvar(dados, nomeCompleto);
    }

    public <T> T carregarDados(String nomeBase, Class<T> tipoClasse) {
        String nomeCompleto = diretorioBase + nomeBase +
estrategia.getExtensaoArquivo();

        System.out.println("📂 Carregando com " +
estrategia.getNomeEstrategia());
        return estrategia.carregar(nomeCompleto, tipoClasse);
    }

    public void exibirEstrategiaAtual() {
        System.out.println("🎯 Estratégia atual: " +
estrategia.getNomeEstrategia());
        System.out.println("   Extensão: " +
```

```
estrategia.getExtensaoArquivo());
    }

    // Métodos de conveniência para trocar estratégias
    public void usarTexto() {
        setEstrategia(new EstrategiaTexto());
    }

    public void usarBinario() {
        setEstrategia(new EstrategiaBinario());
    }

    public void usarJSON() {
        setEstrategia(new EstrategiaJSON());
    }
}
```

Demonstração Completa do Strategy

Java

```
public class TesteStrategy {

    public static void main(String[] args) {
        System.out.println("=== TESTE DO PADRÃO STRATEGY ===\n");

        // Dados de teste
        List<String> produtos = Arrays.asList(
            "Notebook Dell", "Mouse Logitech", "Teclado Mecânico",
            "Monitor 4K", "Webcam HD"
        );

        // Cria gerenciador com estratégia padrão
        GerenciadorPersistencia gerenciador = new GerenciadorPersistencia();

        // Teste com todas as estratégias
        testarTodasEstrategias(gerenciador, produtos);

        // Demonstração de flexibilidade
        demonstrarFlexibilidade(gerenciador, produtos);
    }

    private static void testarTodasEstrategias(GerenciadorPersistencia
gerenciador,
                                               List<String> dados) {
        System.out.println("1. 🧪 TESTANDO TODAS AS ESTRATÉGIAS:\n");
```

```
        // Teste com Texto
        gerenciador.usarTexto();
        gerenciador.exibirEstrategiaAtual();
        gerenciador.salvarDados(dados, "produtos");

        List<String> dadosTexto = gerenciador.carregarDados("produtos",
List.class);
        System.out.println("   Dados carregados: " + dadosTexto.size() + "
itens\n");

        // Teste com Binário
        gerenciador.usarBinario();
        gerenciador.exibirEstrategiaAtual();
        gerenciador.salvarDados(dados, "produtos");

        List<String> dadosBinario = gerenciador.carregarDados("produtos",
List.class);
        System.out.println("   Dados carregados: " + dadosBinario.size() + "
itens\n");

        // Teste com JSON
        gerenciador.usarJSON();
        gerenciador.exibirEstrategiaAtual();
        gerenciador.salvarDados(dados, "produtos");

        List<String> dadosJSON = gerenciador.carregarDados("produtos",
List.class);
        System.out.println("   Dados carregados: " + dadosJSON.size() + "
itens\n");
    }

    private static void demonstrarFlexibilidade(GerenciadorPersistencia
gerenciador,
                                                List<String> dados) {
        System.out.println("2. 🎯 DEMONSTRAÇÃO DE FLEXIBILIDADE:\n");

        // Cenário: Sistema que precisa trocar formato baseado em contexto

        // Para desenvolvimento: usar texto (fácil depuração)
        System.out.println("📝 Modo Desenvolvimento:");
        gerenciador.usarTexto();
        gerenciador.salvarDados(dados, "dev_produtos");

        // Para produção: usar binário (eficiência)
        System.out.println("\n⚡ Modo Produção:");
        gerenciador.usarBinario();
        gerenciador.salvarDados(dados, "prod_produtos");
```

```
        // Para API: usar JSON (interoperabilidade)
        System.out.println("\n🌐 Modo API:");
        gerenciador.usarJSON();
        gerenciador.salvarDados(dados, "api_produtos");

        // Demonstra que o mesmo código funciona com qualquer estratégia
        System.out.println("\n🔄 Testando intercambialidade:");

        EstrategiaPersistencia[] estrategias = {
            new EstrategiaTexto(),
            new EstrategiaBinario(),
            new EstrategiaJSON()
        };

        for (EstrategiaPersistencia estrategia : estrategias) {
            gerenciador.setEstrategia(estrategia);

            // Mesmo código, comportamento diferente
            boolean sucesso = gerenciador.salvarDados("Teste " +
estrategia.getNomeEstrategia(),
                                                      "teste_intercambio");
            System.out.println("   Resultado: " + (sucesso ? "✅ Sucesso" :
"❌ Falha"));
        }
    }
}
```

Vantagens do Padrão Strategy

Flexibilidade: Algoritmos podem ser trocados em tempo de execução sem modificar código cliente.

Extensibilidade: Novas estratégias podem ser adicionadas sem modificar código existente.

Eliminação de Condicionais: Remove necessidade de if/else ou switch complexos para escolher algoritmos.

Testabilidade: Cada estratégia pode ser testada independentemente.

Reutilização: Estratégias podem ser reutilizadas em diferentes contextos.

Princípio Aberto/Fechado: Aberto para extensão (novas estratégias), fechado para modificação (código existente).

---

7. RESUMO EXECUTIVO PARA PROVA

7.1 Conceitos Fundamentais que Você DEVE Dominar

Arquivos de Texto

- **Quando usar:** Dados legíveis, configurações, logs, depuração
- **Como implementar:** BufferedReader/Writer com try-with-resources
- **Fragmento essencial:** `while ((linha = reader.readLine()) != null)`

Arquivos Binários

- **Quando usar:** Grandes volumes, performance crítica, dados numéricos
- **Como implementar:** DataInputStream/OutputStream
- **Fragmento essencial:** Ordem de leitura = ordem de escrita

JSON

- **Quando usar:** APIs, intercâmbio entre sistemas, dados semi-estruturados
- **Estrutura:** Objetos {}, arrays [], strings "", números, booleans, null
- **Vantagem:** Legibilidade + interoperabilidade universal

Serialização

- **Quando usar:** Objetos complexos, caches, comunicação entre JVMs
- **Como implementar:** `implements Serializable` + ObjectInputStream/OutputStream
- **Conceitos-chave:** serialVersionUID, transient, preservação de grafo de objetos

Streams

- **Filosofia:** Programação declarativa vs imperativa
- **Pipeline:** fonte.stream().intermediárias().terminal()
- **Operações essenciais:** filter, map, collect, reduce
- **Lazy evaluation:** Execução só acontece na operação terminal

MVC

- **Model:** Dados + regras de negócio (não conhece View/Controller)
- **View:** Interface + apresentação (não conhece Model, só Controller)
- **Controller:** Coordenação + fluxo (conhece Model e View)

Singleton

- **Objetivo:** Garantir instância única + acesso global
- **Implementação:** Construtor privado + getInstance() + controle de thread safety
- **Casos de uso:** Configurações, logger, recursos únicos

Strategy

- **Objetivo:** Algoritmos intercambiáveis em tempo de execução
- **Estrutura:** Interface + implementações concretas + contexto
- **Aplicação na prova:** Estratégias de persistência (texto/binário/JSON)

7.2 Fragmentos de Código para Memorizar

Java

```java
// Try-with-resources (FUNDAMENTAL)
try (BufferedReader reader = new BufferedReader(new
FileReader("arquivo.txt"))) {
    String linha;
    while ((linha = reader.readLine()) != null) {
        // processar linha
    }
} catch (IOException e) {
    // tratar erro
}

// Stream básico
List<String> resultado = lista.stream()
    .filter(item -> condicao)
    .map(String::toUpperCase)
    .collect(Collectors.toList());

// Singleton thread-safe
public static MinhaClasse getInstance() {
    if (instancia == null) {
        synchronized (MinhaClasse.class) {
            if (instancia == null) {
                instancia = new MinhaClasse();
            }
        }
    }
    return instancia;
}
```

```java
// Strategy básico
public interface Estrategia {
    void executar();
}

public class Contexto {
    private Estrategia estrategia;
    public void setEstrategia(Estrategia e) { this.estrategia = e; }
    public void executar() { estrategia.executar(); }
}
```

7.3 Erros Críticos para Evitar

❌ **Não usar try-with-resources** para arquivos
❌ **Ordem incorreta** em leitura/escrita binária
❌ **Esquecer @Override** em implementações
❌ **Confundir operações intermediárias com terminais** em Streams
❌ **Não tratar IOException** em manipulação de arquivos
❌ **Singleton sem thread safety** em ambiente concorrente
❌ **Model conhecendo View** no padrão MVC

7.4 Dicas para Respostas na Prova

Para questões conceituais:

1. Defina o conceito claramente
2. Explique QUANDO usar (casos de uso)
3. Mencione vantagens E limitações
4. Dê exemplo prático

Para questões de código:

1. Estrutura básica primeiro (imports, try-catch)
2. Lógica principal
3. Comentários explicativos
4. Tratamento de erros

Demonstre compreensão profunda:

- Explique o "porquê" por trás das decisões
- Compare alternativas
- Mencione trade-offs

- Use terminologia técnica precisa

Boa sorte na sua prova! 🍀