ARQUIVOS

4.1 - Introdução

O que fazer para armazenar informações e não perdê-las ao finalizar um programa?

É possível utilizar arquivos ou banco de dados, pois ambos contêm informações que são armazenadas em meio físico (HD, CD-ROM, disquete, etc.).

A principal diferença entre esses dois tipos de armazenamento consiste em sua organização, no acesso e recuperação dos dados.

Neste material, serão abordadas maneiras de acesso somente a arquivos. E, como existem vários tipos de arquivo (arquivo de imagem, de som, de textos...), será visto somente arquivos que armazenam textos.

Para manipular arquivos em Java, é necessário importar o pacote **`java.io`**, como veremos adiante.

4.2 – ARQUIVO-TEXTO

Um arquivo do tipo texto pode guardar qualquer informação como um conjunto de caracteres. Se, por um lado são arquivos que podem ser lidos, entendidos e alterados com mais facilidade, por outro, são arquivos cujo acesso é mais demorado por ser um acesso seqüencial, não permitindo um acesso direto a elementos do arquivo.

Ao trabalhar com um registro dentro de um arquivo, é necessário lembrar que os registros são organizados por campo e, em cada campo é que devem ser inseridas as informações. Assim, os campos devem aparecer todos em uma única linha do arquivo e, cada linha deve conter apenas um registro. Olhe como exemplo, a figura 1 abaixo:

Nome	Nota1	Nota2	Media
Fulano de Tal	7,0	8,0	7,5
Ciclano de Souza	6.0	6.5	6.25
Beltrano Silva	8.0	5.0	6.5

Existem quatro operações básicas de manipulação de arquivo:

- **Inserção**
- **Consulta**
- **Alteração**
- **Exclusão**

Lembre-se que ao manipular um arquivo, a primeira operação a ser feita é a **abertura** do arquivo e última operação é o **fechamento** do mesmo. Todas essas operações serão estudadas, passo-a-passo a seguir.

4.3 – ARQUIVO DE ACESSO SEQUENCIAL

Arquivos de acesso seqüencial são arquivos que armazenam informações como uma seqüência de caracteres e, têm como desvantagens:

- As informações são lidas na mesma ordem em que foram inseridas;
- É necessário percorrer o arquivo linha por linha até encontrar a informação desejada.

- As informações não podem ser alteradas diretamente no arquivo. Para isso, é necessário copiar todo o conteúdo do arquivo para um auxiliar, com as modificações já realizadas.
- O acesso aos dados é muito lento.
- Não é possível abrir um arquivo desse tipo para ler e escrever ao mesmo tempo.

4.3.1 – Preparação

Para incluir, consultar, alterar ou excluir um registro de um arquivo é necessário adotar um processo padrão como descrito abaixo:

1) Criar uma classe para definir o tipo do registro que será manipulado

Na classe de manipulação de arquivo:

2) Criar uma variável do tipo registro declarado em (1)
3) Criar uma variável do tipo arquivo, mencionando o tipo de arquivo que será manipulado (de leitura ou escrita)
4) Abrir o arquivo de forma adequada
5) Manipular as informações
6) Fechar o arquivo

4.3.2 – Tratamento de Exceções

Trabalhar com arquivo pode provocar várias exceções na hora da execução, por exemplo:

- ✓ Não encontrar o arquivo
- ✓ Os dados do arquivo não estão compatíveis com os dados que serão lidos
- ✓ Arquivo está vazio,
- ✓ Etc.

No caso de trabalhar com arquivos, alguns tipos de exceções que podem ocorrer são:

- **`FileNotFoundException`** ➔ Ocorre esta exceção quando o programa não encontra o arquivo que se quer abrir. Deve-se importar a classe **`java.io.FileNotFoundException`**.
- **`NoSuchElementException`** ➔ Ocorre esta exceção quando o programa lê no arquivo uma quantidade de elementos diferente do esperado. Por exemplo, espera-se ler três elementos em uma determinada linha e encontram-se lá apenas dois elementos. Deve-se importa a classe **`java.util.NoSuchElementException`**.
- **`IOException`** ➔ Ocorre esta exceção quando há um erro ao abrir ou fechar um arquivo. Por exemplo, quando um arquivo é aberto para gravação em uma unidade com tamanho insuficiente ou quando um arquivo de leitura é aberto para gravação. Deve-se importar a classe **`java.io.IOException`**
- **`NumberFormatException`** ➔ Ocorre esta exceção quando o programa lê no arquivo um valor de um tipo diferente do tipo do esperado. Por exemplo, espera-se ler um inteiro e no arquivo há uma String.

4.3.3 – Classes FileWriter e PrintWriter

Para manipular um arquivo com o objetivo de gravar alguma informação, optou-se por trabalhar com as classes ***`FileWriter`*** e ***`PrintWriter`***, presentes no pacote ***`java.io.FileWriter`*** e ***`java.io.PrintWriter`***.

A classe para gravação de arquivo ficaria:

```
package io;
import java.io.*;
/**
 * @author Cinthia
 *
 * TODO Esta classe cria um objeto que e um arquivo para gravacao
 */
public class GravaArquivo {
  private FileWriter writer;
  // objeto que representara o "gravador" de caracteres.
  private PrintWriter saida;
  // objeto que possibilita escrever Strings no arquivo
  // utilizando os métodos print() e println().
  /**
   * Construtor da classe
   * @param nome => nome do arquivo que sera aberto para gravacao
   * @throws IOException => Excecao se houver algum problema se o
   *                        arquivo nao puder ser aberto para gravacao
   */
  public GravaArquivo (String nome) throws IOException{
    try{
      // false significa que se o arquivo ja existir, ele sera
      // sobrescrito caso queira apenas acrescentar dados ao final
      // do arquivo, deve usar true. Se o arquivo nao existir, cria um.
      writer = new FileWriter(new File(nome), true);
      // esse objeto significa que significando que 0 arquivo poderá
      // sofrer inclusão de dados. O segundo argumento (opcional) indica
      // (true) que os dados serão enviados para o arquivo a toda
      // chamada do método println(), caso contrário, os dados só são
      // enviados quando voce enviar uma quebra de linha, fechar o
      // arquivo ou mandar ele atualizar as mudanças (modo autoflush).
      saida = new PrintWriter (writer);
    }
    catch (IOException e){
      throw new IOException ("ARQUIVO NAO PODE SER ABERTO PARA"+
                             "GRAVACAO");
    }
  }
  /**
   * Este metodo grava uma String qualquer em um arquivo tipo texto
   * @param str => String a ser gravada no arquivo
   */
  public void gravaArquivo (String str)     {
      this.saida.print(str);
  }
  /**
   * Metodo para fechar o arquivo de gravacao
   * @throws IOException => Excecao, se ocorrer erro ao fechar o
   *                        arquivo.
   */
  public void fechaArquivo ()throws IOException{
    try{
      this.saida.close();
      this.writer.close();
    }
    catch (IOException e){
```

```
        throw new IOException ("ERRO AO FECHAR O ARQUIVO");
      }
    }
}
```

4.3.4 – Inclusão

A inclusão de dados em um arquivo sequencial é feita sempre ao final do arquivo, então, o elemento que será inserido será o último elemento do arquivo.

Para incluir um elemento em um arquivo deve-se:

Para incluir um elemento em um arquivo, de forma organizada para facilitar uma posterior leitura, deve-se:

1) Ler todos os campos do registro
2) Criar um arquivo para armazenamento (escrita) de dados
3) Escrever cada campo do registro no arquivo
4) Fechar o arquivo

4.3.5 – Classe Scanner

Para a leitura de um arquivo, optou-se por trabalhar com a classe ***Scanner***, presente no pacote ***java.util.Scanner***. Esta classe oferece facilidades na leitura de dados e ainda, pode receber no seu construtor uma String, um File, um InputStream, etc.

Vamos supor que se tenha um arquivo (.txt) como na linha abaixo:

```
28/08/2025;MIRIAN ALVES OLIVEIRA;85013090363;774.5;true
```

Ou seja, cada linha contem data, nome, cpf, valor de uma promissória e a informação de que ela está paga ou não. Todas as informações estão separadas por “;”. O exemplo abaixo mostra a sequência de passos para ler esse arquivo:

```
package io;
/**
 * @author Cinthia
 *
 * TODO Classe para criar um objeto do tipo arquivo de leitura
 */
import java.io.*;
import java.util.*;
import item.CadPromissoria;
import item.Promissoria;
public class LeArquivoPromissoria {
  private Scanner entrada;
  // objeto do tipo Scanner para realizar a leitura dos dados
  // do arquivo.
  /**
   * Construtor
   * @param nome => Nome do arquivo que sera aberto para leitura
   * @throws FileNotFoundException => Excecao se nao encontrar o arquivo
   */
  public LeArquivoPromissoria(String nome) throws FileNotFoundException{
    try{
      this.entrada = new Scanner (new File (nome));
      //Instanciamento do objeto do tipo Scanner, tendo como argumento
```

```
      // File que será o arquivo que será lido
    }
    catch (FileNotFoundException e){
      throw new FileNotFoundException ("ARQUIVO NAO ENCONTRADO");
    }
  }
  /**
   * Metodo para ler os dados contidos no arquivo
   * @param alunos => Vetor de alunos que sera preenchido durante a
   *                              leitura do arquivo
   * @throws IllegalStateException => Excecao se houver erro ao ler o
   * arquivo
   */
  public CadPromissoria leArquivo(int tam)throws NoSuchElementException,
                                        ArrayIndexOutOfBoundsException{
    CadPromissoria cadastro = new CadPromissoria(tam);
    String linha;
    try{
      while (this.entrada.hasNext()){
      // A função hasNext() indica se ainda existe uma String
      // para ser lida.
           linha = this.entrada.nextLine();
           // A função nextLine() devolve a próxima linha como
           // uma String.
           cadastro.insere(separaDados(linha));
      }
      return cadastro;
    }
    catch (ArrayIndexOutOfBoundsException e){
      throw new ArrayIndexOutOfBoundsException("Arquivo corrompido");
    }
  }
  /**
   * Metodo para transformar uma linha do arquivo em um objeto
   * do tipo Promissoria
   * @param linha => String contendo a linha do arquivo que sera
                transformada
   * @return => a promissoria criada a partir do linha passada
   * @throws NoSuchElementException => Excecao causada por elementos
   *              insuficientes na String, durante a transformacao
   */
  private Promissoria separaDados(String linha)throws
                                              NoSuchElementException{
    String[] dados;
    String nome, cpf, aux;
    Calendar venc;
    double valor;
    boolean paga;

    try{
      dados = linha.split(";");
      // O método split quebra uma String em várias substrings a partir
      // do caracter definido como argumento,nesse caso ";", cria
      // um vetor de String e armazena cada substring em um posicao
      aux = dados[0];
      venc = montaData(aux);
      nome = dados[1];
      cpf = dados[2];
```

```
      valor  = Double.parseDouble(dados[3]);
      paga = Boolean.parseBoolean(dados[4]);
      return (new Promissoria(nome, cpf, valor, venc, paga));
    }
    catch (NoSuchElementException erro){
      throw new NoSuchElementException ("ARQUIVO DIFERENTE DO REGISTRO");
    }
  }
  /**
   * Metodo para separar o dia, mes e o ano de uma data.
   * Eles estao separados por / de cria uma data com
   * esses inteiros.
   * @param str => String contendo dd/mm/aa
   * @return um objeto do tipo Calendar com a data.
   */
  protected static Calendar montaData (String str){
    int dia, mes, ano;
    String[] aux;
    Calendar data = Calendar.getInstance();
    aux = str.split("/");
    dia = Integer.parseInt(aux[0]);
    mes = Integer.parseInt(aux[1])-1;
    ano = Integer.parseInt(aux[2]);
    data.set(ano, mes, dia);
    return data;
  }
  /**
   * Metodo para fechar o arquivo de leitura
   * @throws IllegalStateException => Excecao causada se nao conseguir
   *                                                fechar o arquivo.
   */
  public void fechaArquivo ()throws IllegalStateException{
    try{
      this.entrada.close();
    }
    catch (IllegalStateException e){
      throw new IllegalStateException ("ERRO AO FECHAR O ARQUIVO");
    }
  }
}
```