

N_PROG SIST II_A2 - Texto de apoio

Site: [EAD Mackenzie](#)

Tema: PROGRAMAÇÃO DE SISTEMAS II {TURMA 03B} 2023/1

Livro: N_PROG SIST II_A2 - Texto de apoio

Impresso por: FELIPE BALDIM GUERRA .

Data: terça, 25 abr 2023, 00:39

Descrição

Índice

1. MANIPULAÇÃO DE ARQUIVOS NO JAVA

2. REFERÊNCIAS

1. MANIPULAÇÃO DE ARQUIVOS NO JAVA

Durante a execução de um programa, seus dados ficam na memória. Quando o programa termina, ou o computador é desligado, os dados na memória desaparecem, ou seja, são completamente apagados.

Para armazenar os dados permanentemente, você tem de colocá-los em um **arquivo**, que normalmente é armazenado em um disco rígido (HD), em um pendrive ou em um CD-ROM. Os arquivos são organizados em pasta (=diretório=folder) e, dentro de uma pasta, um arquivo é identificado por um nome e extensão únicos.

Manipular um arquivo em um programa em Java é muito parecido com trabalhar com livros: para utilizar um livro, você tem de **abri-lo**. Quando você termina, você tem de **fechá-lo**. Enquanto o livro estiver **aberto**, você pode tanto ler quanto escrever nele. Na maioria das vezes, você lê o livro inteiro em sua ordem natural, uma página após a outra, do início para fim. Em um arquivo texto, os dados são organizados como uma sequência de caracteres dividida em linhas terminadas por um caractere de fim de linha (\n).

Em um programa, a manipulação de um arquivo é realizada por meio de uma API (*Application Programming Interface*), em português Interface de Programação de Aplicações, ou seja, uma biblioteca de funções.

De modo geral, uma API é composta por uma série de funções acessíveis somente por chamadas de funções dentro do programa, sem a necessidade de se preocupar com detalhes de implementação da API.

Na linguagem Java, a API para manipulação de arquivos é chamada de **java.io** (pacote **java.io**) e, como outras bibliotecas do Java, é orientada a objetos e utiliza os principais conceitos vistos até agora: herança, polimorfismo, classes abstratas e interfaces.

A biblioteca **java.io** usa o polimorfismo para definir fluxos de entrada (**InputStream**) e de saída (**OutputStream**) para toda e qualquer operação, seja ela relativa a um arquivo, a um registro de banco de dados, a uma conexão remota via sockets ou, até mesmo, a entrada e saída padrão de um programa (normalmente o teclado e o console).

As classes **abstratas InputStream** e **OutputStream** definem, respectivamente, o comportamento padrão dos fluxos entrada e saída em Java: em um fluxo de entrada, é possível ler bytes e, no fluxo de saída, escrever bytes, independentemente da origem e destino do fluxo; com o conceito de fluxos, é possível ler/escrever em um arquivo ou em um socket, basta chamar o método correspondente de qualquer classe filha das classes **InputStream** e **OutputStream**.

Para manipular um programa em Java, precisamos realizar três etapas: **abertura do arquivo, leitura/escrita dos dados e fechamento do arquivo**; para ler um byte de um arquivo, usaremos o leitor de arquivo: a **classe FileInputStream**.

Para um objeto da **classe FileInputStream** conseguir ler um byte, ele precisa saber de onde ele deverá ler, ou seja, o nome do arquivo. Essa informação é tão importante que quem escreveu essa classe obriga você a passar o nome do arquivo pelo construtor: sem isso, o objeto não pode ser construído, a **classe InputStream** é abstrata e a **classe FileInputStream**, uma de suas filhas concretas. O construtor da **classe FileInputStream** procurará o arquivo no diretório, na **pasta raiz do projeto no NetBeans**. Veja o exemplo abaixo:

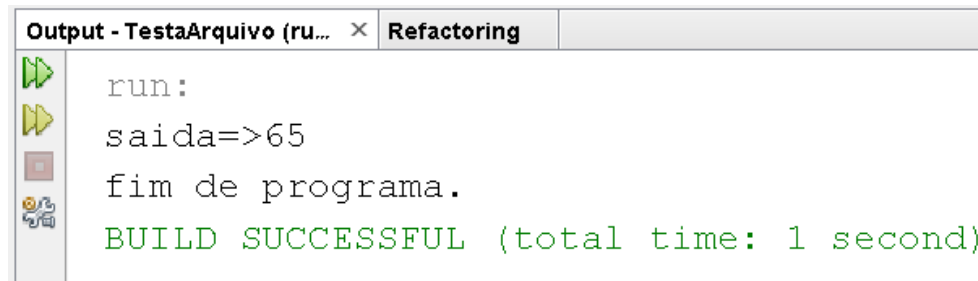
```
import java.io.*;

public class TestaArquivo {
    public static void main(String[] args) throws IOException {
        // Abre o arquivo.txt
        InputStream is = new FileInputStream("arquivo.txt");
        // Le um byte do arquivo.txt
        int b = is.read();
        // Imprime o byte lido
        System.out.println("saida=>" + b);
        // Fecha o fluxo de stream
        is.close();
        System.out.println("fim de programa.");
    }
}
```

No programa acima, logo no início, é importado o **pacote java.io**. Dentro do corpo programa, é instanciado um objeto da **classe FileInputStream** com o nome do arquivo para leitura como argumento para o construtor da classe; em seguida, é lido o primeiro byte do arquivo texto e, depois, é impresso na tela. Imagine que o **arquivo.txt** tenha a seguinte informação:

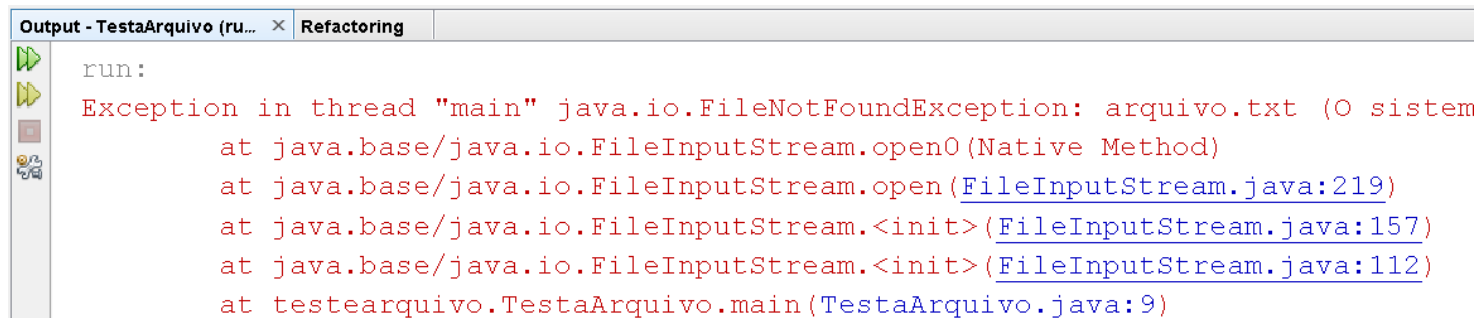
ABCD 1234

A saída do programa seria conforme abaixo, note que é impresso na janela de output do NetBeans o código Unicode da letra A lida do arquivo.txt.



```
Output - TestaArquivo (ru... x Refactoring
run:
saida=>65
fim de programa.
BUILD SUCCESSFUL (total time: 1 second)
```

Caso o arquivo não seja encontrado, é gerada uma exceção de entrada e saída pela JVM (Máquina Virtual Java) e o programa é finalizado imediatamente. Note, na tela de output do NetBeans abaixo, que a mensagem **"fim de programa."** não é apresentada, e sim o rastro da pilha (**stacktrace**) gerada pela exceção **FileNotFoundException** (arquivo não encontrado).



```
Output - TestaArquivo (ru... x Refactoring
run:
Exception in thread "main" java.io.FileNotFoundException: arquivo.txt (O sistema
    at java.base/java.io.FileInputStream.open0(Native Method)
    at java.base/java.io.FileInputStream.open(FileInputStream.java:219)
    at java.base/java.io.FileInputStream.<init>(FileInputStream.java:157)
    at java.base/java.io.FileInputStream.<init>(FileInputStream.java:112)
    at testearquivo.TestaArquivo.main(TestaArquivo.java:9)
```

A exceção **FileNotFoundException** é do tipo **IOException** que é uma **checked exceptions**. Para esse tipo exceção, o programador precisa, obrigatoriamente, tratar (**try-catch**) ou relançar (**throws**) a exceção, no exemplo acima a exceção **FileNotFoundException** foi relançada usando a cláusula **throws** no **função main()** apenas para facilitar a explicação.

Esta não é uma boa prática em uma aplicação real: **você deve sempre tratar suas exceções para sua aplicação poder abortar elegantemente.**

O tratamento de exceções é feito em blocos de códigos estruturados com as clausula **try-catch**:

```
try{
    <instruções que podem ocorrer exceções>
}catch(indica qual exceção será tratada){
    <tratamento da exceção>
}
```

O bloco **try-catch** dá uma chance para o programa se recuperar do erro inesperado, evitando que o programa finalize no momento que acontece a exceção.

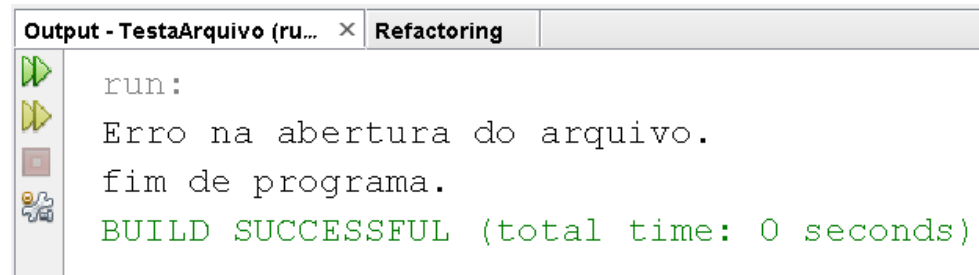
Dentro do bloco **try{...}** são colocados os códigos (instruções, chamadas de métodos, ...) que podem gerar exceções. Dessa forma, o bloco **try{...}** tentará (try) executar o bloco perigoso e, no bloco **catch(){...}**, é colocado o tratamento efetivo da exceção, que pode ser uma mensagem informativa de erro ou logado em um arquivo.

```
import java.io.*;

public class TestaArquivo {
    public static void main(String[] args){

        try {
            // Abre o arquivo.txt
            InputStream is = new FileInputStream("arquivo.txt");
            // Le um byte do arquivo.txt
            int b = is.read();
            // Imprime o byte lido
            System.out.println("saida=>" + b);
            // Fecha o fluxo de stream
            is.close();
        } catch (IOException ex) {
            System.out.println("Erro na abertura do arquivo.");
        }
        System.out.println("fim de programa.");
    }
}
```

Veja que, no programa acima, a abertura do arquivo, a leitura do byte e o fechamento do arquivo está dentro do bloco **try{...}**, apesar de o arquivo não ser encontrado no momento em que a exceção foi **caught** (pega, tratada), a execução volta ao normal a partir daquele ponto e o programa finaliza de forma tradicional.



```
run:
Erro na abertura do arquivo.
fim de programa.
BUILD SUCCESSFUL (total time: 0 seconds)
```

No exemplo, quando foi feita a leitura correta do arquivo, foi apresentado o código Unicode do caractere lido. Mas e se quiséssemos, ao invés de apresentar o código ASCII, imprimir o caractere (letra)?

Para recuperar um caractere, precisamos traduzir os bytes com o Encoding dado para o respectivo código Unicode, lembrando que um código Encondig pode usar um ou mais bytes. Escrever esse decodificador é muito complicado, quem faz isso por você é a **classe InputStreamReader**, seu construtor recebe como parâmetro um objeto da **classe FileInputStream** para funcionar. Caso seja necessário, você pode passar ao construtor da **classe InputStreamReader** o Encoding a ser utilizado como parâmetro, tal como UTF-8 ou ISO-8859-1. Para saber mais sobre Unicode e os Encodings, leia o blog no link a seguir:

<https://blog.caelum.com.br/entendendo-unicode-e-os-character-encodings/amp/>

A seguir, a implementação do programa que faz a leitura de caractere (char).

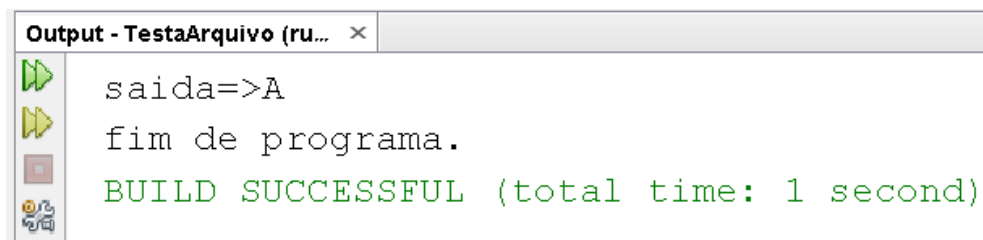

```

import java.io.*;

public class TestaArquivo {
    public static void main(String[] args){
        try {
            // Abre o arquivo.txt
            InputStream is = new FileInputStream("arquivo.txt");
            InputStreamReader isr = new InputStreamReader(is);
            // Le um char do arquivo.txt
            char c = (char) isr.read();
            // Imprime o char lido
            System.out.println("saida=>" + c);
            // Fecha o fluxo de stream
            is.close();
        } catch (IOException ex) {
            System.out.println("Erro na abertura do arquivo.");
        }
        System.out.println("fim de programa.");
    }
}

```

A saída do programa é apresentada abaixo. Note que é impresso na janela de output do NetBeans a **letra A** lida do **arquivo.txt**.



The screenshot shows the 'Output - TestaArquivo (ru...)' window in NetBeans. It contains the following text:

```

saida=>A
fim de programa.
BUILD SUCCESSFUL (total time: 1 second)

```

On the left side of the window, there are four icons: a green play button, a yellow play button, a red stop button, and a bug icon.

Apesar da **classe abstrata Reader** já ajudar no trabalho de manipulação de caracteres, ainda é difícil ler uma String no arquivo texto. A **classe BufferedReader** é um **Reader** que recebe outro **Reader** pelo construtor e concatena os diversos caracteres para formar uma **String** por meio do método **readLine()**, assim conseguimos ler toda a linha de arquivo texto, veja a seguir:

```
import java.io.*;

public class TestaArquivo {
    public static void main(String[] args){
        try {
            // Abre o arquivo.txt
            InputStream is = new FileInputStream("arquivo.txt");
            InputStreamReader isr = new InputStreamReader(is);
            BufferedReader br = new BufferedReader(isr);
            // Le uma String do arquivo.txt
            String s = br.readLine();
            // Imprime a String lida
            System.out.println("saida=>" + s);
            // Fecha o fluxo de stream
            is.close();
        } catch (IOException ex) {
            System.out.println("Erro na abertura do arquivo.");
        }
        System.out.println("fim de programa.");
    }
}
```

No exemplo acima, lemos apenas a primeira linha do arquivo, o **método readLine()** devolve a linha que foi lida e muda o cursor para a próxima linha, a saída do exemplo acima seria:

```
Output - TestaArquivo (ru... x
saída=>ABCD;1234
fim de programa.
BUILD SUCCESSFUL (total time: 1 second)
```

Caso chegue ao fim do Reader (no nosso caso, fim do arquivo), a chamada **br.readLine()** devolverá **null**. Assim, se quisermos ler todo arquivo, basta usar um laço simples (**while**) com a condição de parada o retorno de **null** para a chamada do **método readLine()**.

Agora que já conseguimos ler uma String em um arquivo texto, podemos incrementar nosso exemplo. Imagine que agora queremos quebrar a linha lida em duas partes: uma somente com as letras e outra com os dígitos. Note que a String que lida com essas duas partes são separadas pelo caractere ponto e **vírgula** “;”. Para isso, podemos usar o **método split()** que quebra uma **String** em várias substrings a partir de um caractere definido por você, veja o exemplo abaixo:

```
// Le uma String do arquivo.txt
String s = br.readLine();
// Separa a String s em duas substring
String subStrings[] = s.split(";");
// Imprime as substrings geradas pelo método split()
System.out.println("saída=>" + subStrings[0]);
System.out.println("saída=>" + subStrings[1]);
```

A partir do programa modificado temos a seguinte saída.

```
saída=>ABCD
saída=>1234
fim de programa.
BUILD SUCCESSFUL (total time: 0 seconds)
```

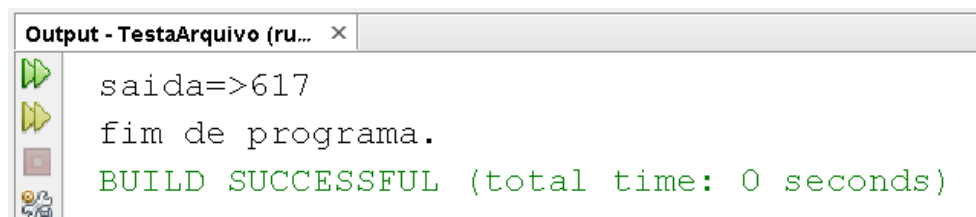
Um detalhe importante aqui é que você deverá saber previamente quantas substrings o método `split()` gerará, isso é determinado em função do formato da linha do arquivo texto. Em nosso exemplo, definimos o seguinte formato: uma sequência de letras, ponto e vírgula e uma sequência de dígitos, mas você, como desenvolvedor, pode definir outras sequências, e o caractere separador que pode ser qualquer caractere, por exemplo o espaço em branco (" ").

No programa exemplo acima, conseguimos separar a String lida em duas partes uma com somente letras ("ABCD") e outra com somente dígitos ("1234"). Mas e se precisássemos "transformar" esses dígitos em um número?

Para transformar uma String em número, utilizamos uma **classe Wrapper**, que é uma classe que representa um tipo primitivo. Por exemplo, o **Wrapper** de int é a **classe Integer**, assim, para transformar a String em um número inteiro, utilizamos o **método estático `parseInt()`** da **classe Integer**:

```
// converte a substring com somente digitos em numeros inteiros.  
int numero = Integer.parseInt(subStrings[1]);  
numero = numero / 2;  
System.out.println("saida=>" + numero);
```

A partir dessa transformação conseguimos fazer operações aritméticas com a **variável `numero`**, por exemplo dividir por dois e imprimir o resultado:



The screenshot shows an IDE output window titled "Output - TestaArquivo (ru...". It contains three lines of text: "saida=>617", "fim de programa.", and "BUILD SUCCESSFUL (total time: 0 seconds)". To the left of the text are three icons: a green play button, a yellow play button, and a red stop button.

```
Output - TestaArquivo (ru... x  
saida=>617  
fim de programa.  
BUILD SUCCESSFUL (total time: 0 seconds)
```

Além da **classe Integer**, temos outras classes **Wrapper**, tais como as **classes Double, Short, Long, Float** etc. as quais contêm o mesmo tipo de método, como **parseDouble** e **parseFloat** que retornam um **double** e **float** respectivamente.

Essas classes também são muito utilizadas para fazer o **wrapping** (embrulho) de tipos primitivos como objetos, pois referências de objetos e tipos primitivos são incompatíveis. Imagine que precisamos passar como argumento um inteiro para um guardador de objetos (**classe ArrayList**). Um int não é um Object, como fazer então?

Basta declarar os elementos da **classe ArrayList** como objetos da **classe Integer**:

```
ArrayList<Integer> guardador = new ArrayList();  
int i = 5;  
guardador.add(i);
```

Assim podemos dizer que Wrappers, como o **Integer**, são úteis quando precisamos usar nossa variável em coleções como o **ArrayList**.

O diagrama abaixo mostra como funciona a interação entre as classes envolvidas na leitura de um arquivo texto.



Analogamente, a escrita de um arquivo texto pode ser apresentada da seguinte forma:



Como exemplo, considere o programa abaixo que escreve a String "Mackenzie" no arquivo texto saída.txt.

```
import java.io.*;

public class TestaSaida {
    public static void main(String[] args){
        try {
            // Cria o arquivo saida.txt
            OutputStream os = new FileOutputStream("saida.txt");
            OutputStreamWriter osw = new OutputStreamWriter(os);
            BufferedWriter bw = new BufferedWriter(osw);
            // Escreve no arquivo saida.txt
            bw.write("Mackenzie");
            // Fecha o fluxo de stream
            bw.close();
        } catch (IOException ex) {
            System.out.println("Erro na criação do arquivo.");
        }
        System.out.println("fim de programa.");
    }
}
```

No programa, o **arquivo saída.txt** é apagado e criado toda vez que o construtor da **classe FileOutputStream** é chamado, mas classe FileOutputStream tem um outro construtor sobrecarregado que pode receber um booleano como segundo parâmetro, para indicar se você quer manter o que já estava escrito no arquivo e se as próximas escritas serão realizadas no final do arquivo (append).

O **método write()** da **classe BufferedWriter** não insere o(s) caractere(s) de quebra de linha. Para isso, você pode chamar o **método newLine()**, e é muito importante sempre fechar o arquivo, você pode fazer isso chamando diretamente o **método close()** da **classe OutputStream**, ou ainda chamando o close da **classe BufferedWriter**. Nesse último caso, o close será cascadeado aos objetos que o **BufferedWriter** utiliza para realizar a escrita, além dele fazer o flush dos buffers no caso da escrita, ou seja, os dados que estão no buffer de dados são enviados imediatamente para o arquivo de escrita.

2. REFERÊNCIAS

SILVEIRA, Paulo. **Entendendo Unicode e os Character Encodings**. Blog Caelum, 2007. Disponível em: <<https://blog.caelum.com.br/entendendo-unicode-e-os-character-encodings/amp/>>. Acesso em: 12 mar. 2021.