

CAPÍTULO 15

Pacote java.io

"A benevolência é sobretudo um vício do orgulho e não uma virtude da alma."
— Doantien Alphonse François (Marquês de Sade)

Ao término desse capítulo, você será capaz de:

- ler e escrever bytes, caracteres e Strings de/para a entrada e saída padrão;
- ler e escrever bytes, caracteres e Strings de/para arquivos;
- utilizar buffers para agilizar a leitura e escrita através de fluxos;
- usar Scanner e PrintStream.

15.1 - CONHECENDO UMA API

Vamos passar a conhecer APIs do Java. `java.io` e `java.util` possuem as classes que você mais comumente vai usar, não importando se seu aplicativo é desktop, web, ou mesmo para celulares.

Apesar de ser importante conhecer nomes e métodos das classes mais utilizadas, o interessante aqui é que você enxergue que todos os conceitos previamente estudados são aplicados a toda hora nas classes da biblioteca padrão.

Não se preocupe em decorar nomes. Atenha-se em entender como essas classes estão relacionadas e como elas estão tirando proveito do uso de interfaces, polimorfismo, classes abstratas e encapsulamento. Lembre-se de estar com a documentação (javadoc) aberta durante o contato com esses pacotes.

Veremos também threads e sockets em capítulos posteriores, que ajudarão a condensar nosso conhecimento, tendo em vista que no exercício de sockets utilizaremos todos conceitos aprendidos, juntamente com as várias APIs.

15.2 - ORIENTAÇÃO A OBJETOS NO JAVA.IO

Assim como todo o resto das bibliotecas em Java, a parte de controle de entrada e saída de dados (conhecido como **io**) é orientada a objetos e usa os principais conceitos mostrados até agora: interfaces, classes abstratas e polimorfismo.

A ideia atrás do polimorfismo no pacote `java.io` é de utilizar fluxos de entrada (`InputStream`) e de saída (`OutputStream`) para toda e qualquer operação, seja ela relativa a um **arquivo**, a um campo **blob** do banco de dados, a uma conexão remota via **sockets**, ou até mesmo às **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 em Java: em um fluxo de entrada, é possível ler bytes e, no fluxo de saída, escrever bytes.

A grande vantagem dessa abstração pode ser mostrada em um método qualquer que utiliza um `OutputStream` recebido como argumento para escrever em um fluxo de saída. Para onde o método está escrevendo? Não se sabe e não importa: quando o sistema precisar escrever em um arquivo ou em uma socket, basta chamar o mesmo método, já que ele aceita qualquer filha de `OutputStream`!

Nova editora Casa do Código com livros de uma forma diferente



Editoras tradicionais pouco ligam para ebooks e novas tecnologias. Não conhecem programação para revisar os livros tecnicamente a fundo. Não têm anos de experiência em didáticas com cursos.

Conheça a **Casa do Código**, uma editora diferente, com curadoria da **Caelum** e obsessão por livros de qualidade a preços justos.

[Casa do Código, ebook com preço de ebook.](http://www.caelum.com.br/apostila-java-orientacao-objetos/pacote-java-io/)

15.3 - INPUTSTREAM, INPUTSTREAMREADER E BUFFEREDREADER

Para ler um byte de um arquivo, vamos usar o leitor de arquivo, o `FileInputStream`. Para um `FileInputStream` conseguir ler um byte, ele precisa saber de onde ele deverá ler. 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.

```

1 class TestaEntrada {
2     public static void main(String[] args) throws IOException {
3         InputStream is = new FileInputStream("arquivo.txt");
4         int b = is.read();
5     }
6 }
```

A classe `InputStream` é abstrata e `FileInputStream` uma de suas filhas concretas. `FileInputStream` vai procurar o arquivo no diretório em que a JVM for invocada (no caso do Eclipse, vai ser a partir de dentro do diretório do projeto). Alternativamente você pode usar um caminho absoluto.

Quando trabalhamos com `java.io`, diversos métodos lançam `IOException`, que é uma exception do tipo checked - o que nos obriga a tratá-la ou declará-la. Nos exemplos aqui, estamos declarando `IOException` através da clausula `throws` do `main` apenas para facilitar o exemplo. Caso a exception ocorra, a JVM vai parar, mostrando a stacktrace. Esta não é uma boa prática em uma aplicação real: trate suas exceptions para sua aplicação poder abortar elegantemente.

`InputStream` tem diversas outras filhas, como `ObjectInputStream`, `AudioInputStream`, `ByteArrayInputStream`, entre outras.

Para recuperar um caractere, precisamos traduzir os bytes com o encoding dado para o respectivo código unicode, isso pode usar um ou mais bytes. Escrever esse decodificador é muito complicado, quem faz isso por você é a classe `InputStreamReader`.

```

1 class TestaEntrada {
2     public static void main(String[] args) throws IOException {
3         InputStream is = new FileInputStream("arquivo.txt");
4         InputStreamReader isr = new InputStreamReader(is);
5         int c = isr.read();
6     }
7 }
```

O construtor de `InputStreamReader` pode receber o encoding a ser utilizado como parâmetro, se desejado, tal como `UTF-8` ou `ISO-8859-1`.

Encodings

Devido a grande quantidade de aplicativos internacionalizados de hoje em dia, é imprescindível que um bom programador entenda bem o que são os character encodings e o Unicode. O blog da Caelum possui um bom artigo a respeito:

<http://blog.caelum.com.br/2006/10/22/entendendo-unicode-e-os-character-encodings/>

`InputStreamReader` é filha da classe abstrata `Reader`, que possui diversas outras filhas - são classes que manipulam chars.

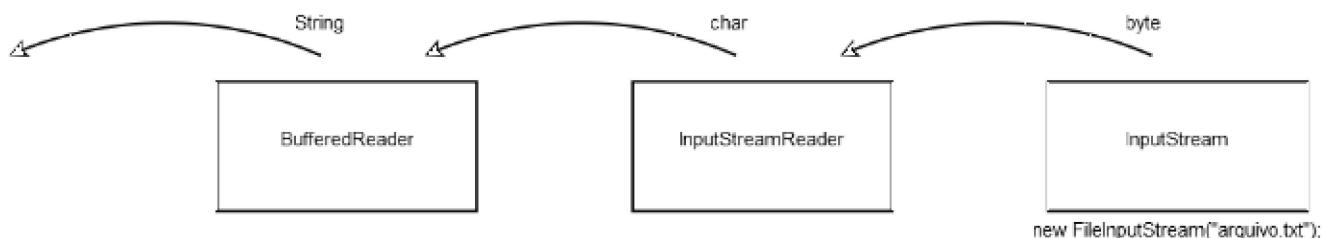
Apesar da classe abstrata `Reader` já ajudar no trabalho de manipulação de caracteres, ainda seria difícil pegar uma `String`. A classe `BufferedReader` é um `Reader` que recebe outro `Reader` pelo construtor e concatena os diversos chars para formar uma `String` através do método `readLine`:

```

1 class TestaEntrada {
2     public static void main(String[] args) throws IOException {
3         InputStream is = new FileInputStream("arquivo.txt");
4         InputStreamReader isr = new InputStreamReader(is);
5         BufferedReader br = new BufferedReader(isr);
6         String s = br.readLine();
7     }
8 }
```

Como o próprio nome diz, essa classe lê do `Reader` por pedaços (usando o buffer) para evitar realizar muitas chamadas ao sistema operacional. Você pode até configurar o tamanho do buffer pelo construtor.

É essa a composição de classes que está acontecendo:



Esse padrão de composição é bastante utilizado e conhecido. É o **Decorator Pattern**.

Aqui, lemos apenas a primeira linha do arquivo. O método `readLine` devolve a linha que foi lida e muda o cursor para a próxima linha. Caso ele chegue ao fim do Reader (no nosso caso, fim do arquivo), ele vai devolver `null`. Então, com um simples laço, podemos ler o arquivo por inteiro:

```

1 class TestaEntrada {
2     public static void main(String[] args) throws IOException {
3         InputStream is = new FileInputStream("arquivo.txt");
4         InputStreamReader isr = new InputStreamReader(is);
5         BufferedReader br = new BufferedReader(isr);
6
7         String s = br.readLine(); // primeira linha
8
9         while (s != null) {
10             System.out.println(s);
11         }
12     }
13 }
```

```

11     s = br.readLine();
12 }
13
14     br.close();
15 }
16 }
```

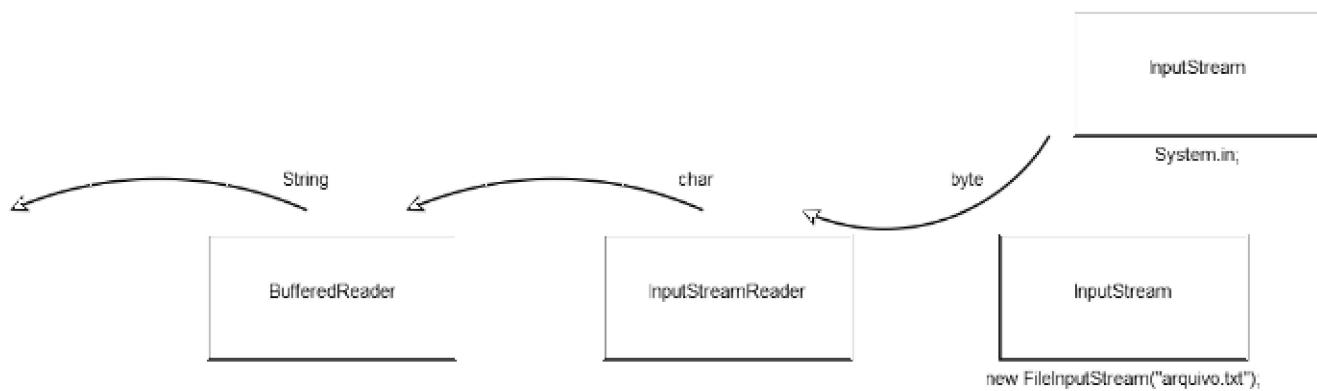
15.4 - LENDO STRINGS DO TECLADO

Com um passe de mágica, passamos a ler do teclado em vez de um arquivo, utilizando o `System.in`, que é uma referência a um `InputStream` o qual, por sua vez, lê da entrada padrão.

```

1 class TestaEntrada {
2     public static void main(String[] args) throws IOException {
3         InputStream is = System.in;
4         InputStreamReader isr = new InputStreamReader(is);
5         BufferedReader br = new BufferedReader(isr);
6         String s = br.readLine();
7
8         while (s != null) {
9             System.out.println(s);
10            s = br.readLine();
11        }
12    }
13 }
```

Apenas modificamos a quem a variável `is` está se referindo. Podemos receber argumentos do tipo `InputStream` e ter esse tipo de abstração: não importa exatamente de onde estamos lendo esse punhado de bytes, desde que a gente receba a informação que estamos querendo. Como na figura:

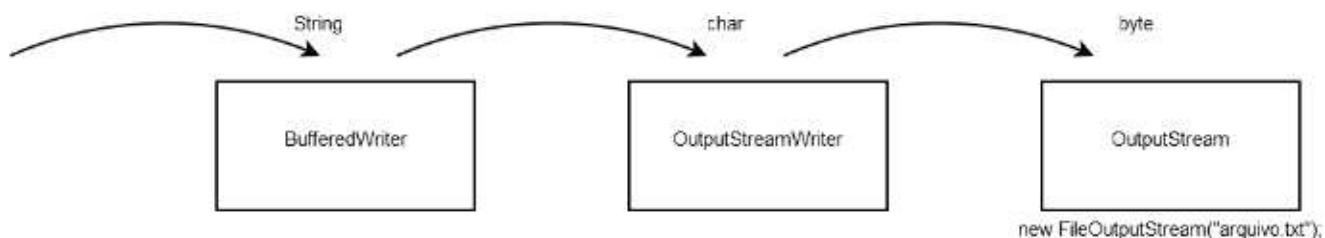


Repare que a ponta da direita poderia ser qualquer `InputStream`, seja `ObjectInputStream`, `AudioInputStream`, `ByteArrayInputStream`, ou a nossa `FileInputStream`. Polimorfismo! Ou você mesmo pode criar uma filha de `InputStream`, se desejar.

Por isso é muito comum métodos receberem e retornarem `InputStream`, em vez de suas filhas específicas. Com isso, elas desacoplam as informações e escondem a implementação, facilitando a mudança e manutenção do código. Repare que isso vai ao encontro de tudo o que aprendemos durante os capítulos que apresentaram classes abstratas, interfaces, polimorfismo e encapsulamento.

15.5 - A ANALOGIA PARA A ESCRITA: `OutputStream`

Como você pode imaginar, escrever em um arquivo é o mesmo processo:



```

1 class TestaSaida {
2     public static void main(String[] args) throws IOException {
3         OutputStream os = new FileOutputStream("saida.txt");
4         OutputStreamWriter osw = new OutputStreamWriter(os);
5         BufferedWriter bw = new BufferedWriter(osw);
6
7         bw.write("caelum");
8
9         bw.close();
10    }
11 }
    
```

Lembre-se de dar *refresh* (clique da direita no nome do projeto, refresh) no seu projeto do Eclipse para que o arquivo criado apareça. O `FileOutputStream` pode receber um booleano como segundo parâmetro, para indicar se você quer reescrever o arquivo ou manter o que já estava escrito (`append`).

O método `write` do `BufferedWriter` não insere o(s) caractere(s) de quebra de linha. Para isso, você pode chamar o método `newLine`.

Fechando o arquivo com o `finally` e o `try-with-resources`

É importante sempre fechar o arquivo. Você pode fazer isso chamando diretamente o método `close` do `InputStream/OutputStream`, ou ainda chamando o `close` do `BufferedReader/Writer`. Nesse último caso, o `close` será cascaneado para os objetos os quais o `BufferedReader/Writer` utiliza para realizar a leitura/escrita, além dele fazer o **flush** dos buffers no caso da escrita.

É comum e fundamental que o `close` esteja dentro de um bloco `finally`. Se um arquivo for esquecido aberto e a referência para ele for perdida, pode ser que ele seja fechado pelo *garbage collector*, que veremos mais a frente, por causa do `finalize`. Mas não é bom você se prender a isso. Se você esquecer de fechar o arquivo, no caso de um programa minúsculo como esse, o programa vai terminar antes que o tal do garbage collector te ajude, resultando em um arquivo não escrito (os bytes ficaram no buffer do `BufferedWriter`). Problemas similares podem acontecer com leitores que não forem fechados.

No Java 7 há a estrutura *try-with-resources*, que já fará o `finally` cuidar dos recursos declarados dentro do `try()`, invocando `close`. Pra isso, os recursos devem implementar a interface `java.lang.AutoCloseable`, que é o caso dos Readers, Writers e Streams estudados aqui:

```
try (BufferedReader br = new BufferedReader(new File("arquivo.txt"))) {
    // com exceção ou não, o close() do br sera invocado
}
```

Já conhece os cursos online Alura?



A **Alura** oferece dezenas de **cursos online** em sua plataforma exclusiva de ensino que favorece o aprendizado com a **qualidade** reconhecida da Caelum. Você pode escolher um curso nas áreas de Java, Ruby, Web, Mobile, .NET e outros, com uma **assinatura** que dá acesso a todos os cursos.

[Conheça os cursos online Alura.](#)

15.6 - UMA MANEIRA MAIS FÁCIL: SCANNER E PRINTSTREAM

A partir do Java 5, temos a classe `java.util.Scanner`, que facilita bastante o trabalho de ler de um `InputStream`. Além disso, a classe `PrintStream` possui um construtor que já recebe o nome de um arquivo como argumento. Dessa forma, a leitura do teclado com saída para um arquivo ficou muito simples:

```
Scanner s = new Scanner(System.in);
PrintStream ps = new PrintStream("arquivo.txt");
while (s.hasNextLine()) {
    ps.println(s.nextLine());
}
```

Nenhum dos métodos lança `IOException`: `PrintStream` lança

`FileNotFoundException` se você o construir passando uma `String`. Essa exceção é filha de `IOException` e indica que o arquivo não foi encontrado. O `Scanner` considerará que chegou ao fim se uma `IOException` for lançada, mas o `PrintStream` simplesmente engole exceptions desse tipo. Ambos possuem métodos para você verificar se algum problema ocorreu.

A classe `Scanner` é do pacote `java.util`. Ela possui métodos muito úteis para trabalhar com `Strings`, em especial, diversos métodos já preparados para pegar números e palavras já formatadas através de expressões regulares. Fica fácil parsear um arquivo com qualquer formato dado.

System.out

Como vimos no capítulo passado, o atributo `out` da classe `System` é do tipo `PrintStream` (e, portanto, é um `OutputStream`).

15.7 - UM POUCO MAIS...

- Existem duas classes chamadas `java.io.FileReader` e `java.io.FileWriter`. Elas são atalhos para a leitura e escrita de arquivos.
- O `do { ... } while(condicao);` é uma alternativa para se construir um laço. Pesquise-o e utilize-o no código para ler um arquivo, ele vai ficar mais sucinto (você não precisará ler a primeira linha fora do laço).

15.8 - EXERCÍCIOS: JAVA I/O

1. Crie um projeto novo chamado `teste-io`. E, nele, crie um programa (simplesmente uma classe com um `main`) que leia da entrada padrão. Para isso, você vai precisar de um `BufferedReader` que leia do `System.in` da mesma forma como fizemos.

Não digite esses nomes de classes complicados! Lembre-se de fazer como o instrutor e escrever primeiro a parte depois do igual. Então, use o **ctrl + 1** para que o Eclipse crie a variável para você! Assim mesmo, enquanto você escreve a parte da direita, abuse do **ctrl + espaço** porque além de te ajudar com o nome, ele colocará o import no devido lugar.

Cuidado: existe mais de uma classe chamada `InputStream`: queremos a do pacote

`java.io.`

```

1 public class TestaEntrada {
2
3     public static void main(String[] args) {
4         InputStream is = System.in;
5         InputStreamReader isr = new InputStreamReader(is);
6         BufferedReader br = new BufferedReader(isr);
7
8         System.out.println("Digite sua mensagem:");
9         String linha = br.readLine(); // primeira linha
10
11        while (linha != null) {
12            System.out.println(linha);
13            linha = br.readLine();
14        }
15    }
16 }
```

O compilador vai reclamar que você não está tratando algumas exceções (como `java.io.IOException`). Utilize a cláusula `throws` para deixar "escapar" a exceção pelo seu `main`, ou use os devidos `try/catch`. Utilize o *quick fix* do Eclipse para isso (**ctrl + 1**).

Vale lembrar que deixar todas as exceptions passarem despercebidas não é uma boa prática! Você pode usar aqui, pois estamos focando apenas no aprendizado da utilização do `java.io`.

Rode sua aplicação. Através da *View Console* você pode digitar algumas linhas para que seu programa as capture.

EOF

Quando rodar sua aplicação, para encerrar a entrada de dados do teclado, é necessário enviarmos um sinal de fim de stream. É o famoso **EOF**, isto é, *end of file*.

No Linux/Mac/Solaris/Unix você faz isso com o `ctrl + D`. No Windows, use o `ctrl + Z`.

2. Quando trabalhamos com recursos que falam com a parte externa à nossa aplicação, é preciso que avisemos quando acabarmos de usar esses recursos. Por isso, é **importantíssimo** lembrar de fechar os canais com o exterior que abrimos.

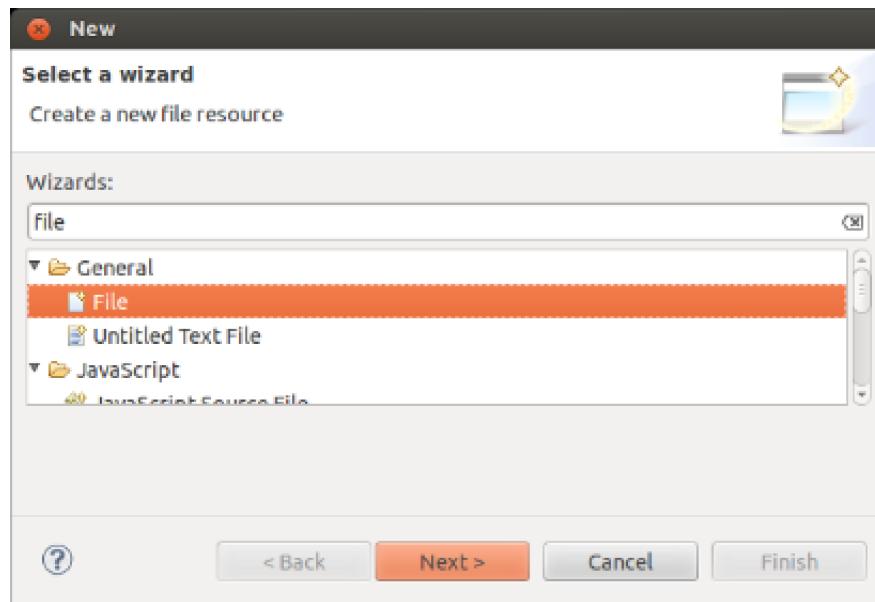
Felizmente para nós, não é necessário fechar cada um dos canais de comunicação que abrimos (`is`, `isr` e `br`) porque, ao fechar a comunicação com o `BufferedReader`, ele mesmo já trata de fechar os recursos dos quais ele depende.

Então, basta adicionarmos a seguinte instrução, depois que recebermos a última informação pelo Reader:

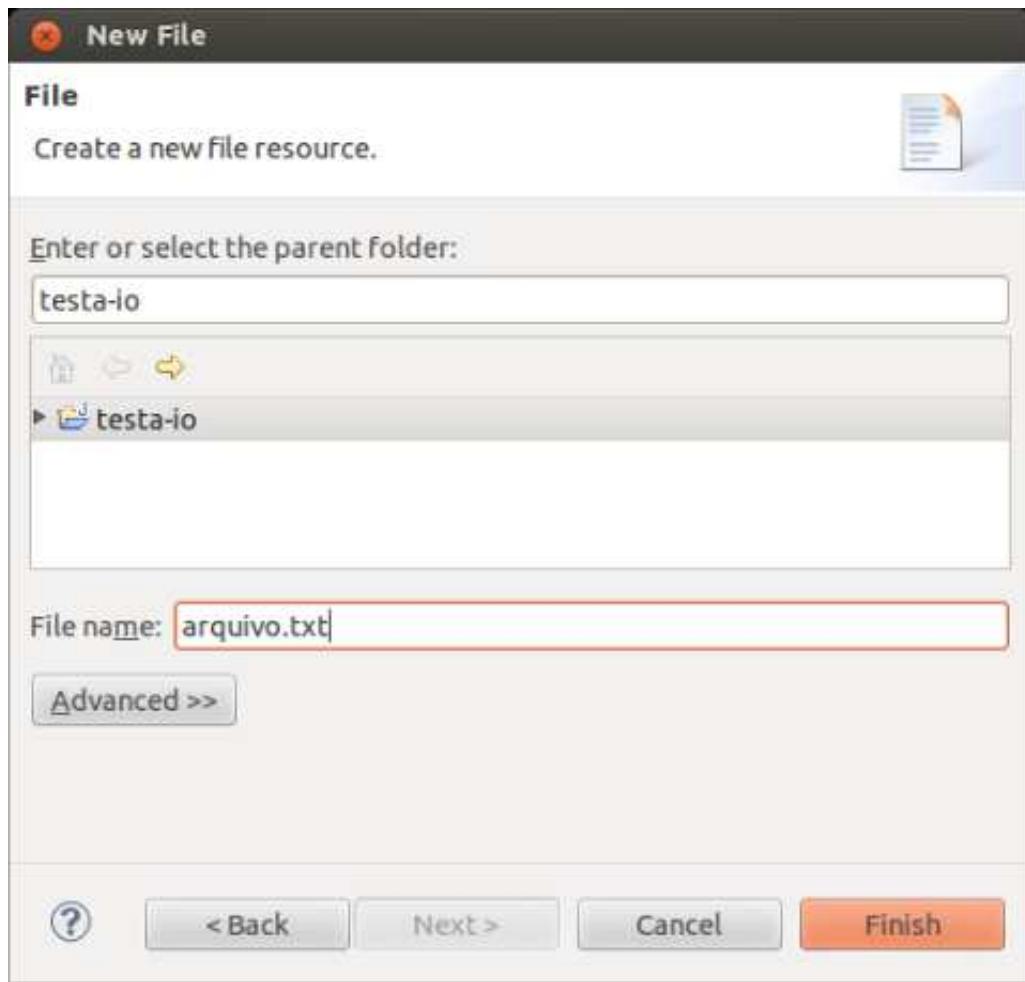
```
br.close();
```

3. Vamos ler de um arquivo, em vez do teclado. Antes, vamos criar o arquivo que será lido pelo programa:

- Use o **ctrl + N** para criar um novo arquivo e comece a digitar `File`. Use as setinhas para chegar na opção `File` e dê **enter** para escolhê-la:



- Com o nome do projeto selecionado, digite o nome `arquivo.txt` no campo `File name`:



c. Troque na classe TestaEntrada O System.in por um new FileInputStream:

```
InputStream is = new FileInputStream("arquivo.txt");
```

4. (conceitual) Seu programa lê todas as linhas desse arquivo. Repare na utilização do polimorfismo. Como ambos são InputStream, isso faz com que eles se encaixem no InputStreamReader.

Além da FileInputStream, que outras classes poderiam tomar seu lugar? Olhe na documentação!

5. Repare que, no final, só usamos mesmo o BufferedReader. As referências para InputStream e para InputStreamReader são apenas utilizadas temporariamente. Portanto, é comum encontrarmos o seguinte código nesses casos:

```
BufferedReader br = new BufferedReader(
    new InputStreamReader(
        new FileInputStream("arquivo.txt")));
```

```
String linha = br.readLine(); // primeira linha
```

Claro que, principalmente em linguagens de alto nível como o Java, preferimos legibilidade em vez de um código mais curto, mas este código em particular é bem comum e aceitável. Faça a alteração no seu programa!

6. Utilize a classe Scanner do Java 5 para ler de um arquivo e colocar na tela. O código vai ficar incrivelmente pequeno.

```
public class EntradaDeUmArquivo {
    public static void main(String[] args) throws IOException {
        InputStream is = new FileInputStream("arquivo.txt");
        Scanner entrada = new Scanner(is);

        System.out.println("Digite sua mensagem:");
        while (entrada.hasNextLine()) {
            System.out.println(entrada.nextLine());
        }
        entrada.close();
    }
}
```

Depois troque a variável `is` para que ela se refira ao `System.in`. Agora você está lendo do teclado!

7. (opcional) Altere seu programa para que ele leia do arquivo e, em vez de jogar na tela, jogue em um outro arquivo. Você vai precisar, além do código anterior para ler de um arquivo, do código para escrever em um arquivo. Para isso, você pode usar o `BufferedWriter` ou o `PrintStream`. Este último é de mais fácil manipulação.

Se for usar o `BufferedWriter`, fazemos assim parar abri-lo:

```
OutputStream os = new FileOutputStream("saida.txt");
OutputStreamWriter osw = new OutputStreamWriter(os);
BufferedWriter bw = new BufferedWriter(osw);
```

Dentro do loop de leitura do teclado, você deve usar `bw.write(x)`, onde `x` é a linha que você leu. Use `bw.newLine()` para pular de linha. Não se esqueça de, no término do loop, dar um `bw.close()`. Você pode seguir o modelo:

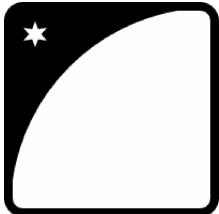
```
while (entrada.hasNextLine()) {
    String linha = entrada.nextLine();
    bw.write(linha);
    bw.newLine();
}
bw.close();
```

Após rodar seu programa, dê um refresh no seu projeto (clique da direita no nome do projeto, refresh) e veja que ele criou um arquivo `saida.txt` no diretório.

8. (opcional) Altere novamente o programa para ele virar um pequeno editor: lê do teclado e escreve em arquivo. Repare que a mudança a ser feita é mínima!

9. (opcional) A classe `Scanner` é muito poderosa! Consulte seu javadoc para saber sobre o `delimiter` e os outros métodos `next`.

Você não está nessa página a toa



Você chegou aqui porque a Caelum é referência nacional em cursos de Java, Ruby, Agile, Mobile, Web e .NET.

Faça curso com **quem escreveu essa apostila**.

[Consulte as vantagens do curso Java e Orientação a Objetos.](#)

15.9 - DISCUSSÃO EM AULA: DESIGN PATTERNS E O TEMPLATE METHOD

Aplicar bem os conceitos de orientação a objetos é sempre uma grande dúvida. Sempre queremos encapsular direito, favorecer a flexibilidade, desacoplar classes, escrever código elegante e de fácil manutenção. E ouvimos falar que a Orientação a Objetos ajuda em tudo isso.

Mas, onde usar herança de forma saudável? Como usar interfaces? Onde o polimorfismo me ajuda? Como encapsular direito? Classes abstratas são usadas em que situações?

Muitos anos atrás, grandes nomes do mundo da orientação a objetos perceberam que criar bons designs orientados a objetos era um grande desafio para muitas pessoas. Perceberam que muitos problemas de OO apareciam recorrentemente em vários projetos; e que as pessoas já tinham certas soluções para esses problemas clássicos (nem sempre muito elegantes).

O que fizeram foi criar **soluções padrões para problemas comuns** na orientação a objetos, e chamaram isso de **Design Patterns**, ou Padrões de Projeto. O conceito vinha da arquitetura onde era muito comum ter esse tipo de solução. E, em 1994, ganhou grande popularidade na computação com o livro *Design Patterns: Elements of Reusable Object-Oriented Software*, um catálogo com várias dessas soluções escrito por Erich Gamma, Ralph Johnson, Richard Helm e John Vlissides (a Gangue dos Quatro, GoF).

Design Patterns tornou-se referência absoluta no bom uso da orientação a objetos. Outros padrões surgiram depois, em outras literaturas igualmente consagradas. O conhecimento dessas técnicas é imprescindível para o bom programador.

Discuta com o instrutor como Design Patterns ajudam a resolver problemas de modelagem em sistemas orientados a objetos. Veja como Design Patterns são aplicados em muitos lugares do próprio Java.

O instrutor comentará do Template Method e mostrará o código fonte do método `read()` da classe `java.io.InputStream`:

```

1 public int read(byte b[], int off, int len) throws IOException {
2     if (b == null) {
3         throw new NullPointerException();
4     } else if (off < 0 || len < 0 || len > b.length - off) {
5         throw new IndexOutOfBoundsException();
6     } else if (len == 0) {
7         return 0;
8     }
9
10    int c = read();
11    if (c == -1) {
12        return -1;
13    }
14
15    b[off] = (byte) c;
16
17    int i = 1;
18    try {
19        for (; i < len ; i++) {
20            c = read();
21            if (c == -1) {
22                break;
23            }
24            b[off + i] = (byte)c;
25        }
26    } catch (IOException ee) {
27    }
28    return i;
29 }
```

Discuta em aula como esse método aplica conceitos importantes da orientação a objetos e promove flexibilidade e extensibilidade.

CAPÍTULO ANTERIOR:

[O pacote java.lang](#)

PRÓXIMO CAPÍTULO:

[Collections framework](#)

Você encontra a Caelum também em:

Blog Caelum

Cursos Online

Facebook

Newsletter

Casa do Código

Twitter