Pacote java.io

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

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.

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!

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.

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

A classe InputStream é abstrata e FileInputStream uma de suas filhas concretas. FileInputStream vai procurar o arquivo no diretório em que a JVM fora 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.

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

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

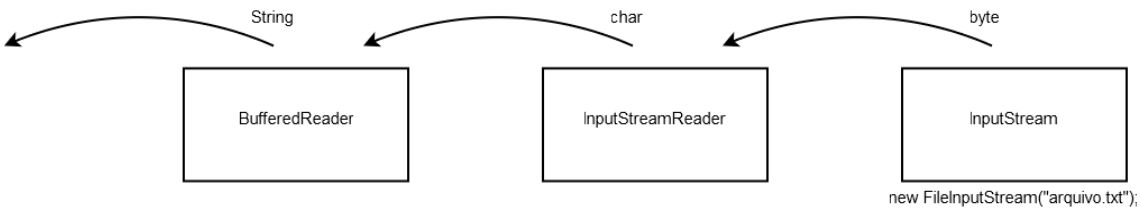
InputStreamReader é filha da classe abstrata Reader, que possui diversas outras filhas - são classes que manipulam chars.

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:

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

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:



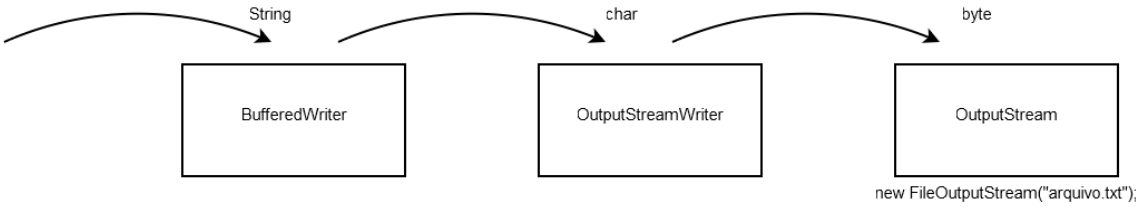
Esse padrão de composição é bastante utilizado e conhecido. É o **Decorator Pattern**.

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:

```
class TestaEntrada {
    public static void main(String[] args) throws IOException {
        InputStream is = new FileInputStream("arquivo.txt");
        InputStreamReader isr = new InputStreamReader(is);
        BufferedReader br = new BufferedReader(isr);

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

        while (s != null) {
            System.out.println(s);
            s = br.readLine();
        }

        br.close();
    }
}
```

A ANALOGIA PARA A ESCRITA: OUTPUTSTREAM

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




```
class TestaSaida {
    public static void main(String[] args) throws IOException {
        OutputStream os = new FileOutputStream("saida.txt");
        OutputStreamWriter osw = new OutputStreamWriter(os);
        BufferedWriter bw = new BufferedWriter(osw);

        bw.write("caelum");

        bw.close();
    }
}
```

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 FileInputStream/OutputStream, ou ainda chamando o close do BufferedReader/Writer. Nesse último caso, o close será cascateado 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.