Orientação a Objetos Classica

Namom Alves Alencar

Javadoc

- Como vamos saber o que cada classe tem no Java? Quais são seus métodos, o que eles fazem?

- Acesse através do link: http://docs.oracle.com/javase/8/docs/api/

Pacote java.io

- 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.

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.

Conhecendo uma API

- 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(quando farlarmos de web) um pouco mais a frente, que ajudarão a condensar nosso conhecimento, tendo em vista que no exercício de web utilizaremos todos conceitos aprendidos, juntamente com as várias APIs.

Orientação a objetos no java.io

- 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();
  } }
```

InputStream, InputStreamReader e BufferedReader

- 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.

InputStream, InputStreamReader e BufferedReader

- 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();
  } }
```

InputStream, InputStreamReader e BufferedReader

- 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.*

InputStream, InputStreamReader e BufferedReader

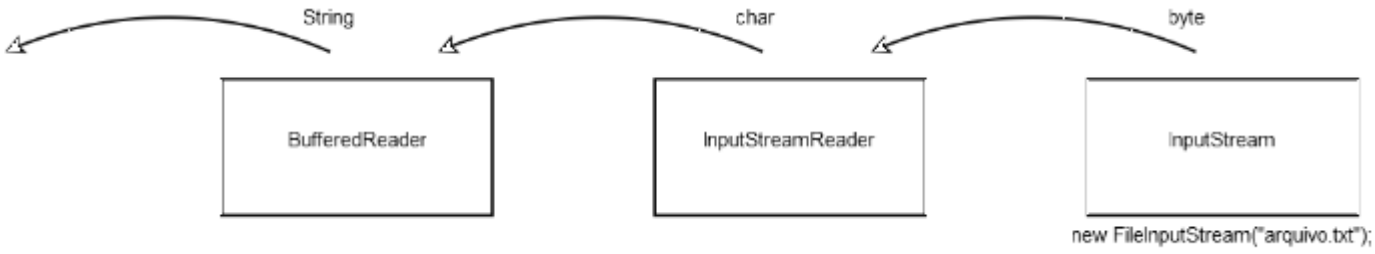
- Apesar da classe abstrata Reader já ajudar no trabalho de manipulação de caracteres, ainda seria difícil pegar uma String. A classe BufferedReader é um Reader que recebe outro Reader pelo construtor e concatena os diversos chars para formar uma String através do método readLine:

```
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();
  } }
```

InputStream, InputStreamReader e BufferedReader

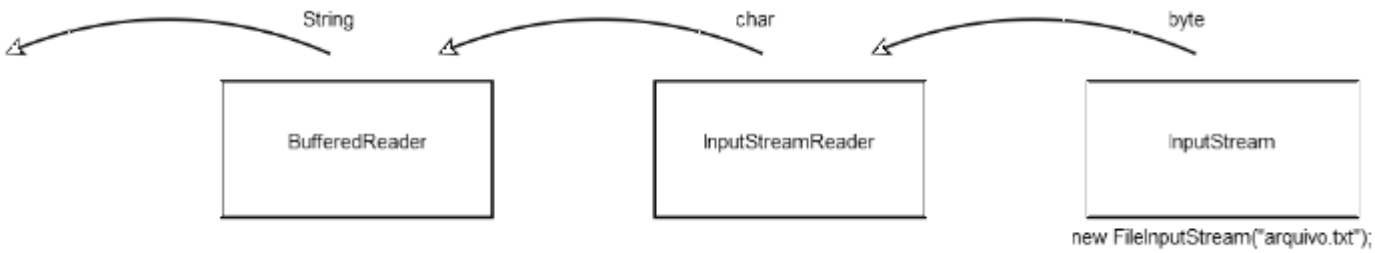
- Como o próprio nome diz, essa classe lê do Reader por pedaços (usando o buffer) para evitar realizar muitas chamadas ao sistema operacional. Você pode até configurar o tamanho do buffer pelo construtor.

- É essa a composição de classes que está acontecendo:



InputStream, InputStreamReader e BufferedReader

- 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**.

InputStream, InputStreamReader e BufferedReader

- 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:

InputStream, InputStreamReader e BufferedReader

```
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();
  }  }
```

Lendo Strings do teclado

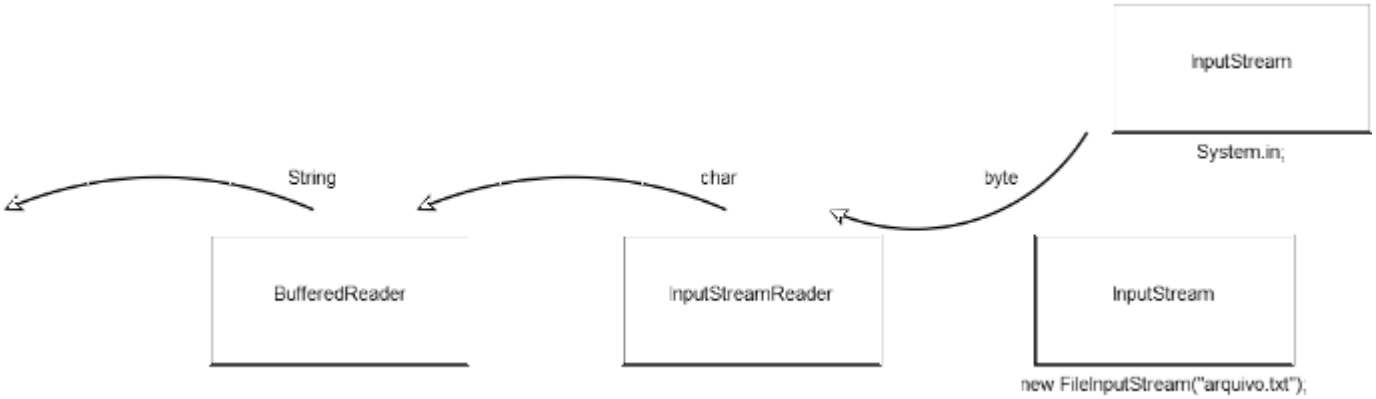
- Com um passe de mágica, passamos a ler do teclado em vez de um arquivo, utilizando o System.in, que é uma referência a um InputStream o qual, por sua vez, lê da entrada padrão.

Lendo Strings do teclado

```
class TestaEntrada {
  public static void main(String[] args) throws IOException {
    InputStream is = System.in;
    InputStreamReader isr = new InputStreamReader(is);
    BufferedReader br = new BufferedReader(isr);
    String s = br.readLine();
    while (s != null) {
      System.out.println(s);
      s = br.readLine();
    }   }  }
```

Lendo Strings do teclado

- Apenas modificamos a quem a variável *is(input stream)* 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:



Lendo Strings do teclado

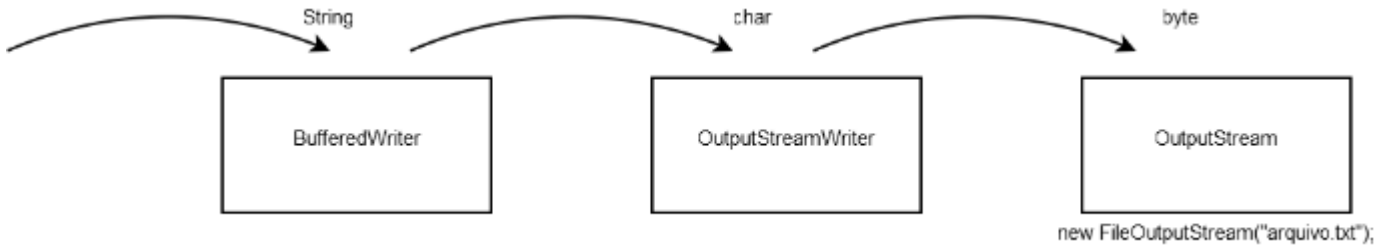
- Repare que a ponta da direita poderia ser qualquer InputStream, seja ObjectInputStream, AudioInputStream, ByteArrayInputStream, ou a nossa FileInputStream. Polimorfismo! Ou você mesmo pode criar uma filha de InputStream, se desejar.

- 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.

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("cursoJava");
     bw.close();
  } }
```



A analogia para a escrita: OutputStream

- Despois da criação do arquivo, 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.

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

- 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.

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

- 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:

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

```
finally {
    if (out != null) {
        System.out.println("Closing PrintWriter");
        out.close();
    } else {
        System.out.println("PrintWriter not open");
    }
}
===============
try (BufferedReader br = new BufferedReader(new File("arquivo.txt"))) {
  // com exceção ou não, o close() do br sera invocado
}
```

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());
}
```

Uma maneira mais fácil: Scanner e PrintStream

- 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.

Exercicios

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.

Exercicios

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.

Exercicios

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();
```

Exercicios

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

a) Use o ctrl + N para criar um novo arquivo e comece a digitar File.

b) Com o nome do projeto selecionado, digite o nome arquivo.txt no campo File

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

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

Exercicios

3. 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!

Exercicios

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

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

Exercicios para casa

5. 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.

Exercicios para casa

6 . 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!

7 . A classe Scanner é muito poderosa! Consulte seu javadoc para saber sobre o delimiter e os outros métodos next.

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).

Bons Estudos

Namom Alves Alencar