CURSO DE PROGRAMAÇÃO EM JAVA

Aula 16

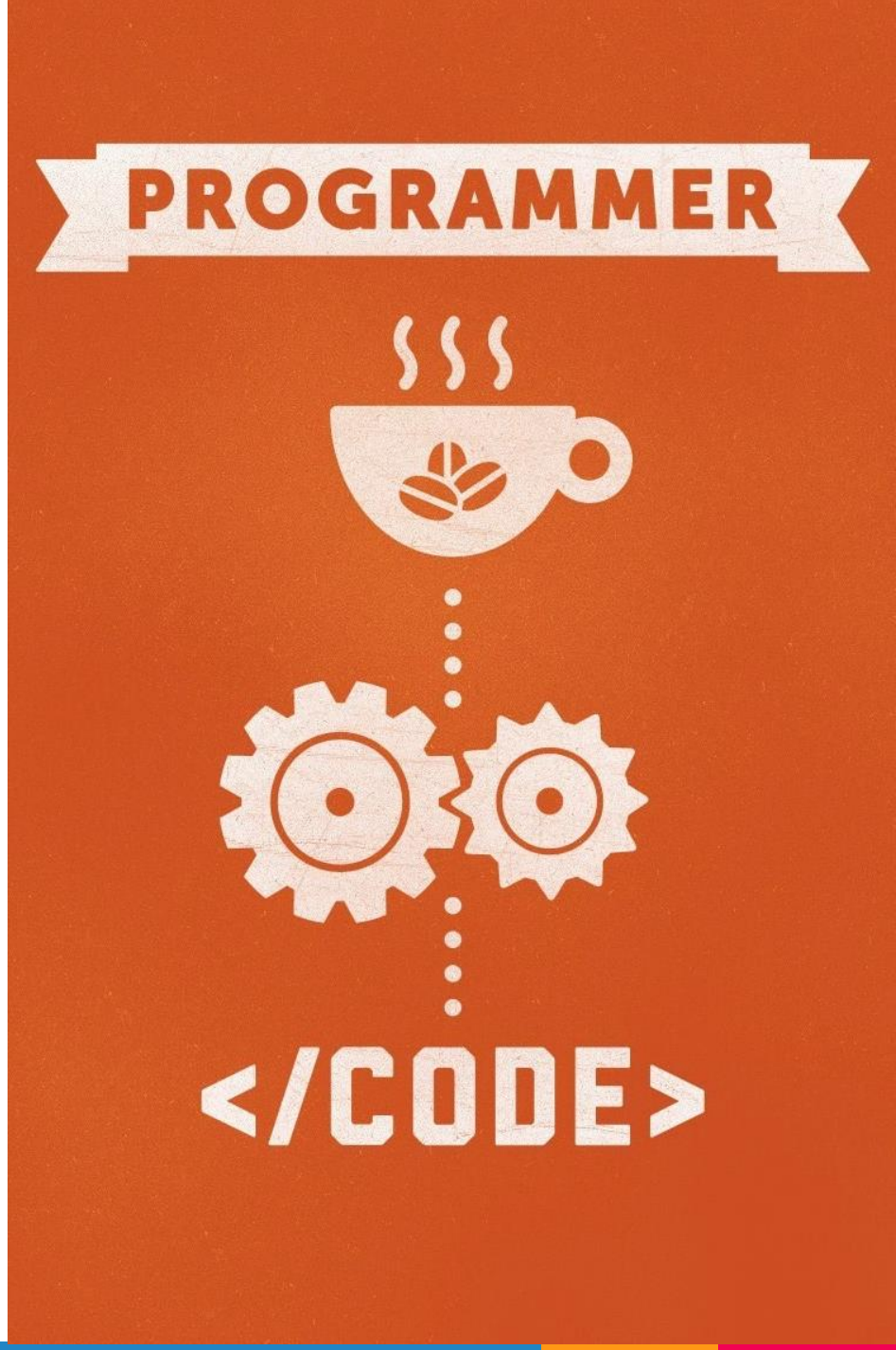
Arquivos e fluxos

1.
Arquivos e fluxos

Arquivos e fluxos

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**.

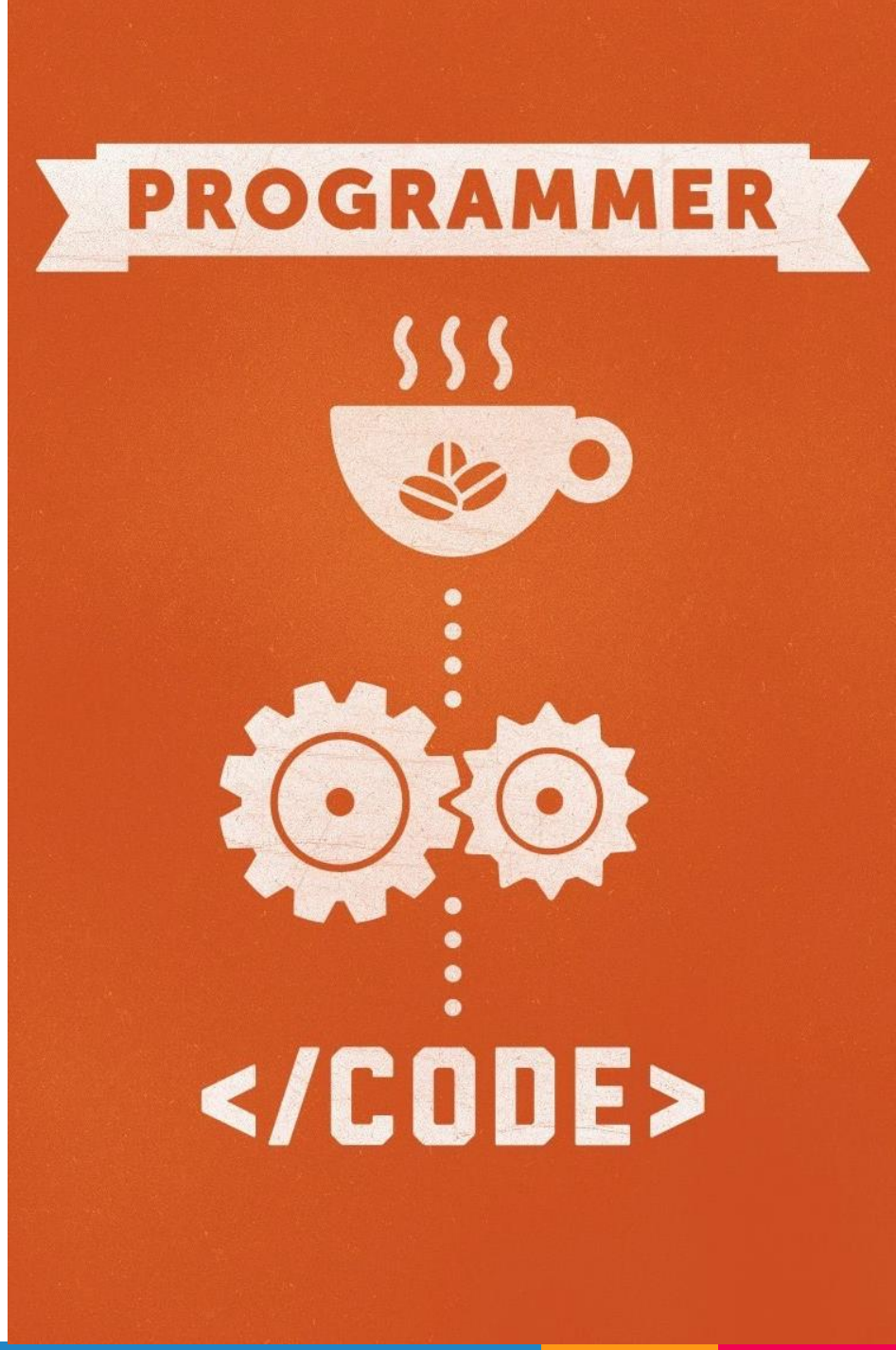
A ideia atrás do polimorfismo no pacote java.io é de utilizar fluxos de entrada (**InputStream**) e de saída (**OutputStream**) para toda e **qualquer operação**, seja ela relativa a um arquivo, a um campo blob do banco de dados, a uma conexão remota via sockets, ou até mesmo às entrada e saída padrão de um programa (normalmente o teclado e o console).



Arquivos e fluxos

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**

.



2.

InputStream, InputStreamReader e BufferedReader

FileInputStream

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

InputStreamReader

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**.

BufferedReader

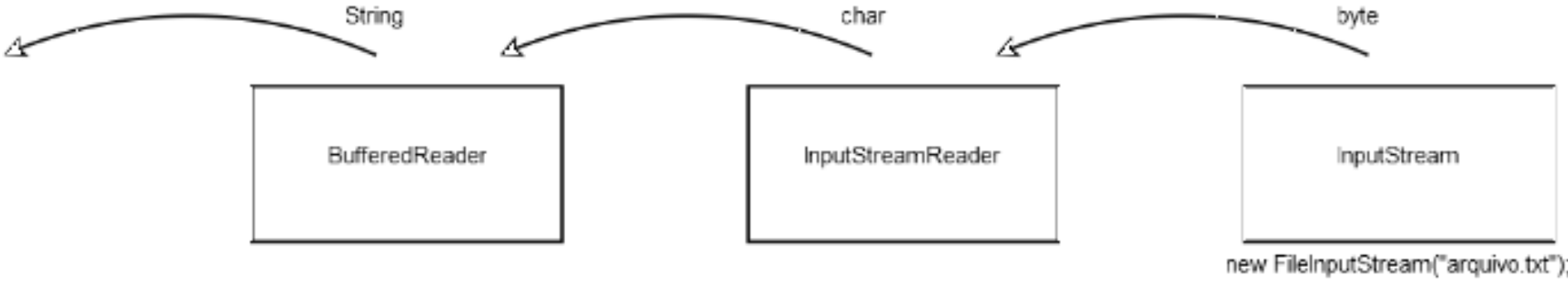
Apesar da classe **abstrata Reader** já ajudar no trabalho de manipulação de **caracteres**, ainda seria difícil pegar uma **String**. A classe **BufferedReader** é um **Reader** que recebe outro **Reader** pelo construtor e **concatena** os diversos **chars** para formar uma **String** através do método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

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.

BufferedReader

É essa a composição de classes que está acontecendo:



Exemplo

```
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();
  }
}
```

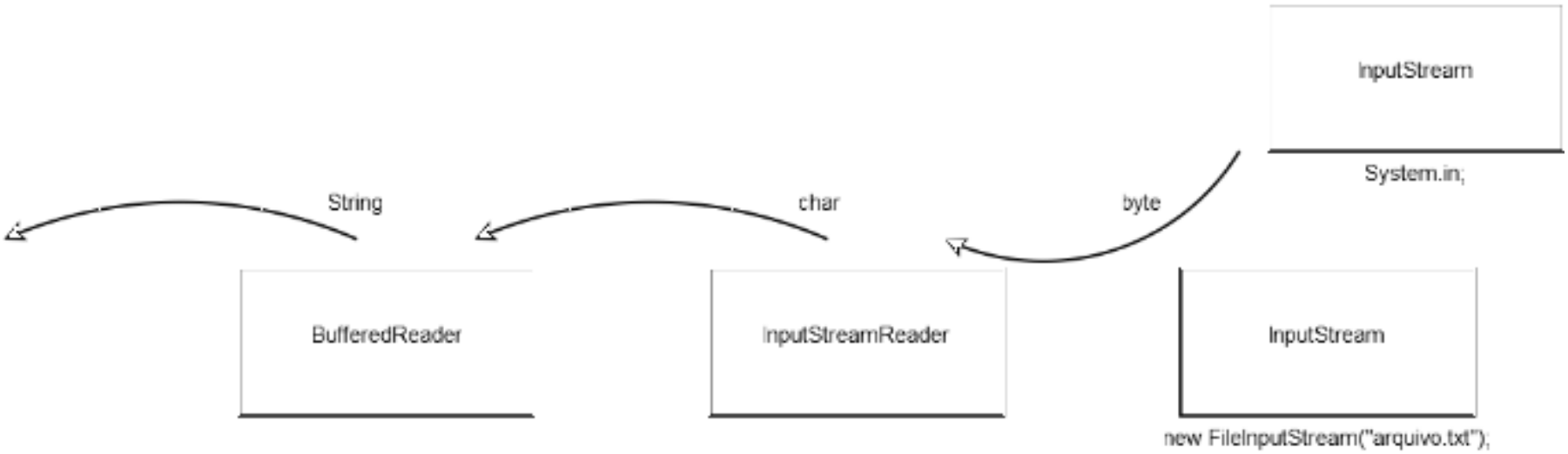
Aqui, lemos apenas a **primeira linha** do arquivo. O método **readLine** devolve a linha que foi lida e muda o cursor para a **próxima linha**. Caso ele chegue ao fim do Reader (no nosso caso, fim do arquivo), ele vai devolver **null**. Então, com um simples laço, podemos ler o arquivo por inteiro:

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(); // primeira linha

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

    br.close();
  }
}
```

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



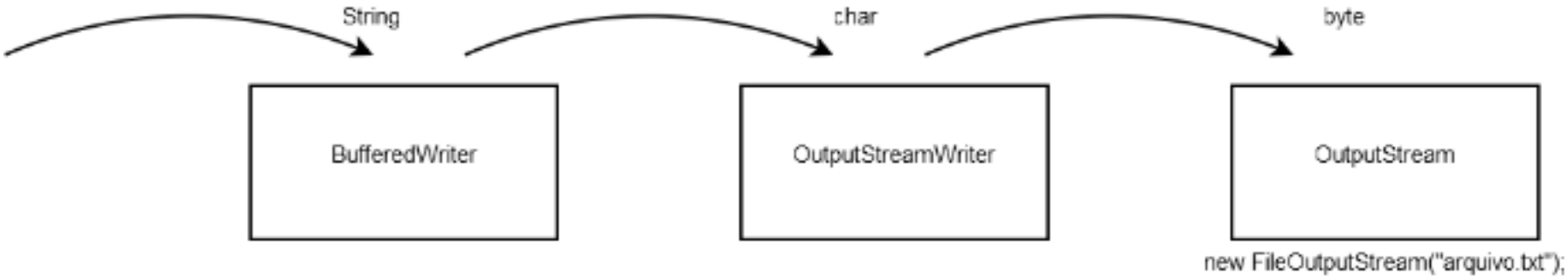
Repare que a ponta da direita poderia ser qualquer **InputStream**, seja **ObjectInputStream**, **AudioInputStream**, **ByteArrayInputStream**, ou a nossa **FileInputStream**. **Polimorfismo**! Ou você mesmo pode criar uma filha de **InputStream**, se desejar.

3.

OutputStream

OutputStream

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

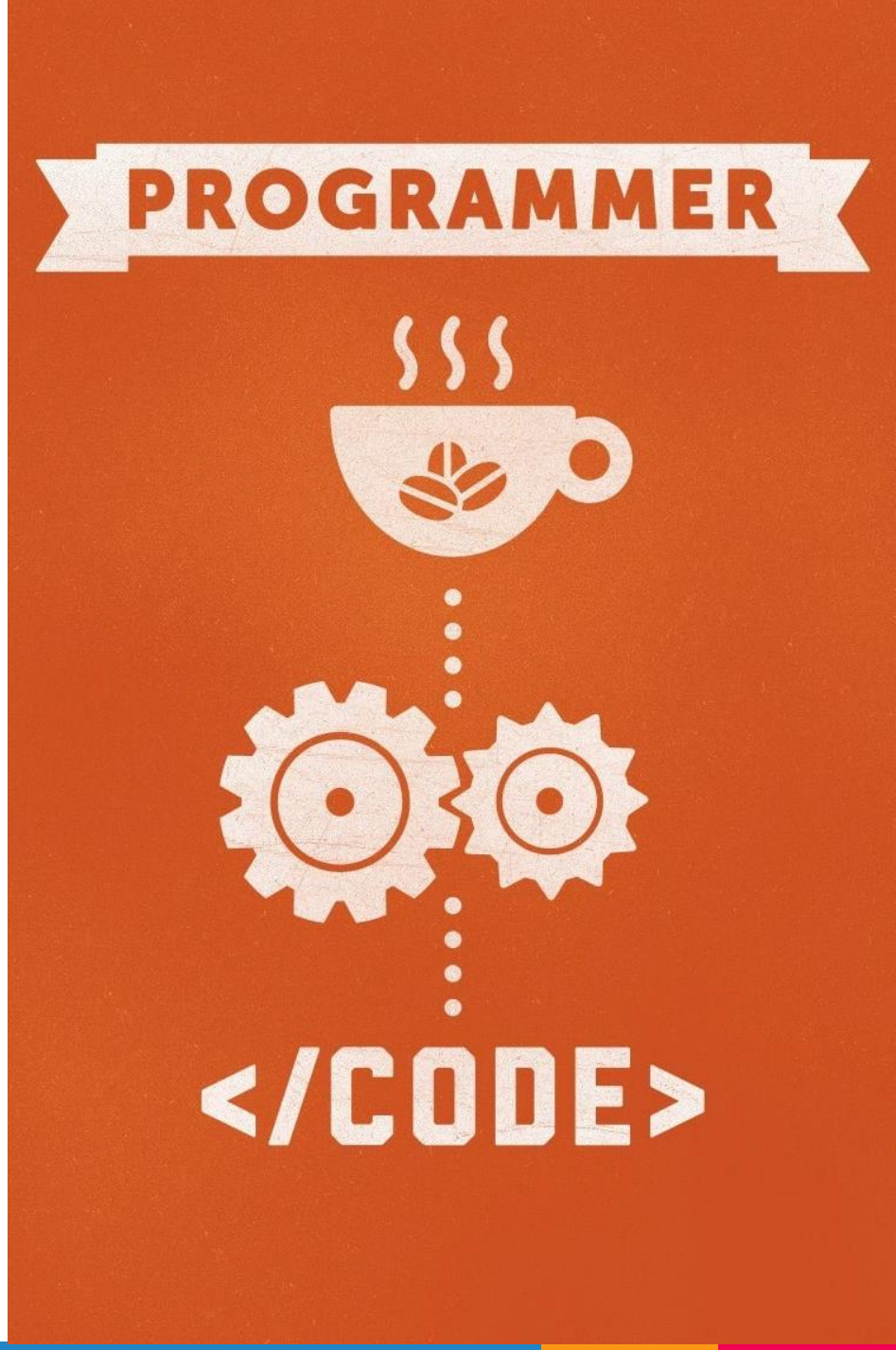



```
class TestaSaida {
   public static void main(String[] args) throws IOException {
     OutputStream os = new FileOutputStream("saida.txt");
     OutputStreamWriter osw = new OutputStreamWriter(os);
     BufferedWriter bw = new BufferedWriter(osw);

     bw.write("James");
     bw.close();
   }
 }
```

OutputStream

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).

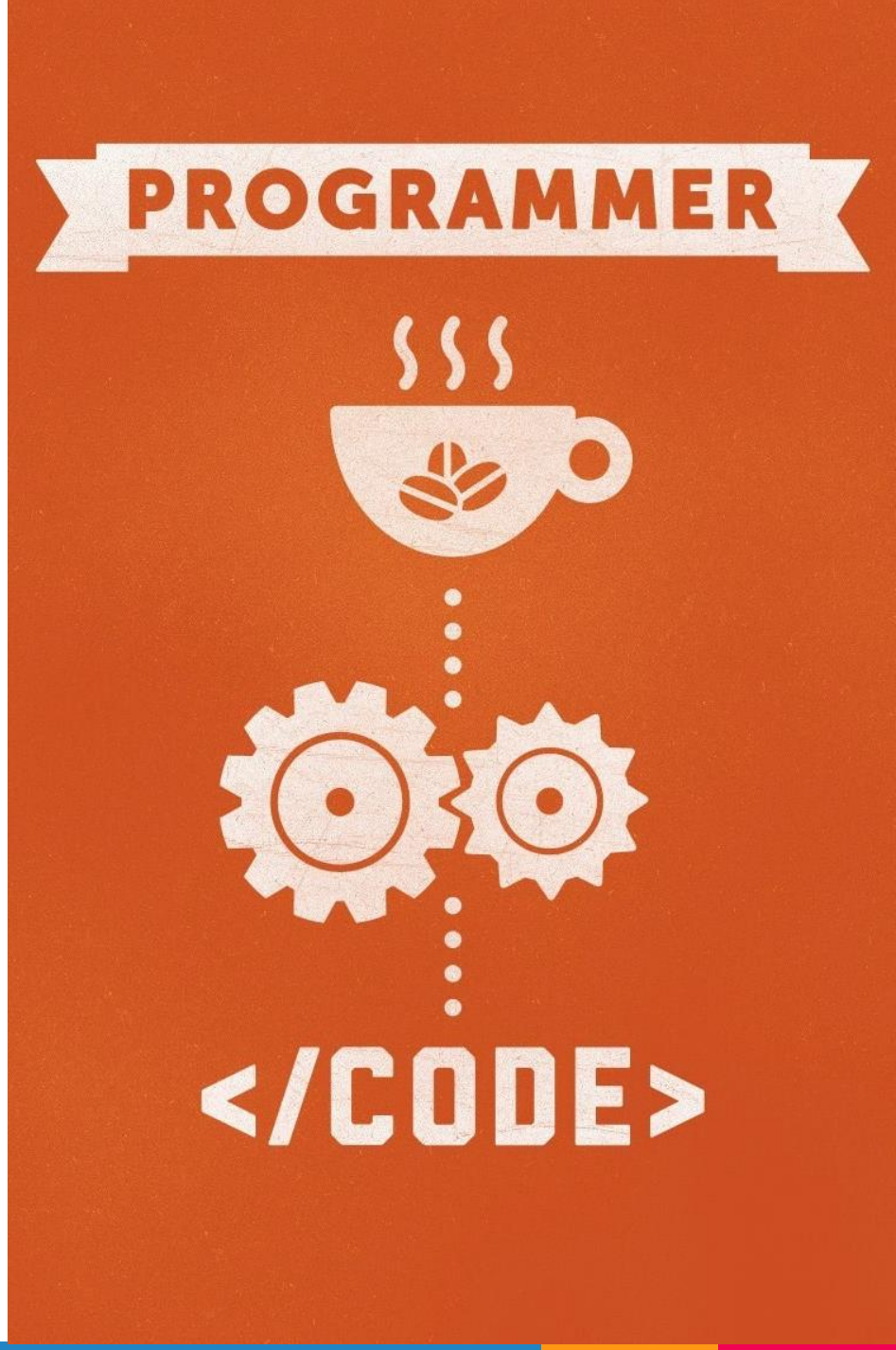
O método **write** do **BufferedWriter** não insere o(s) caractere(s) de quebra de linha. Para isso, você pode chamar o método **newLine**.



OutputStream

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**.



4.

FileReader e FileWriter

FileReader e FileWriter

- Byte Streams: Entrada e Saída de bytes (8-bits)
  - Lêem/escrevem um byte por vez
  - Nível mais baixo de E/S - se estamos lendo um arquivo texto - melhor *Character Streams*
  - Obs.: todos streams são construídos sobre byte streams
- Character Streams: Classes descendentes de Reader e Writer - atuam sobre caracteres, não bytes
  - Para escrita e leitura em arquivos - `FileReader` e `FileWriter`

FileReader e FileWriter

```
public static void copyCharacters() throws IOException {
  FileReader inputStream = null;
  FileWriter outputStream = null;
  try {
    inputStream = new FileReader("xanadu.txt");
    outputStream = new FileWriter("xanadu_output.txt");
    int c;
    while ((c = inputStream.read()) != -1)
      outputStream.write(c);
  } finally {
    if (inputStream != null)
      inputStream.close();
    if (outputStream != null)
      outputStream.close();
  }
}
```

FileReader e FileWriter

```
public static void copyLines() throws IOException {
  BufferedReader inputStream = null;
  PrintWriter outputStream = null;
  try {
    inputStream = new BufferedReader(new FileReader("xanadu.txt"));
    outputStream =
      new PrintWriter(new FileWriter("characteroutput.txt"));
    String l;
    while ((l = inputStream.readLine()) != null)
      outputStream.println(l);
  } finally {
    if (inputStream != null)
      inputStream.close();
    if (outputStream != null)
      outputStream.close();
  }
}
```

5.

Scanner

Scanner

- Métodos para converter texto para os tipos apropriados
- Particiona texto em tokens de tipos diversos
  - Exemplo: “ab*cd 12.34 253”, “ab*cd” - token do tipo String, “12.34” token do tipo double e ”253” token do tipo int
- Útil para ‘quebrar’ entradas formatadas para tokens de acordo com o seu tipo
- Dois construtores:
  - `public Scanner(String s)` e `public Scanner(InputStream source)`
- Alguns métodos:
  - `public String nextLine(); public int nextInt(); public double nextDouble()`

Scanner

```
public static void main(String[] args) throws IOException {
  Scanner s = null;
  try {
    s = new Scanner(
      new BufferedReader(new FileReader("xanadu.txt"))
    );
    while (s.hasNext())
      System.out.println(s.next());
   } finally {
    if (s != null) { s.close(); }
   }
}
```

Scanner

```
public static void main(String[] args) throws IOException {
  Scanner s = null; double sum = 0;
  try {
    s = new Scanner(
      new BufferedReader(new FileReader("usnumbers.txt"))
    );
    while (s.hasNext()) {
      if (s.hasNextDouble()) { sum += s.nextDouble(); }
      else { s.next(); }
    }
  } finally { s.close(); }
  System.out.println(sum);
}
```

DESAFIO

E aí, vamos praticar?

Combinador

Implemente um programa denominado combinador, que recebe duas strings e deve combiná-las, alternando as letras de cada string, começando com a primeira letra da primeira string, seguido pela primeira letra da segunda string, em seguida pela segunda letra da primeira string, e assim sucessivamente. As letras restantes da cadeia mais longa devem ser adicionadas ao fim da string resultante e retornada.

Combinador

Entrada: A entrada contém vários casos de teste. A primeira linha contém um inteiro N que indica a quantidade de casos de teste que vem a seguir. Cada caso de teste é composto por uma linha que contém duas cadeias de caracteres, cada cadeia de caracteres contém entre 1 e 50 caracteres inclusive.

Saída: Combine as duas cadeias de caracteres da entrada como mostrado no exemplo abaixo e exiba a cadeia resultante.

Exemplo de Entrada	Exemplo de Saída
2 Tpo oCder aa bb	TopCoder abab

Exercício

2. Faça um programa em Java que gere uma versão criptografada de um arquivo texto trocando cada caractere de código ASCII j pelo caractere de código ASCII j+k, onde k é um parâmetro especificado pelo usuário. Fique atento para não gerar códigos ASCII fora da faixa permitida

3. Faça um programa Java que leia um arquivo texto chamado “entrada.txt” e imprima, em outro arquivo texto, denominado “saida.txt”, o total de letras, vogais, consoantes, espaços em branco, palavras e o total de linhas encontradas no primeiro arquivo.

Obrigado!

Alguma pergunta?

Você pode nos contatar em:
ywassef@hotmail.com