SCC0504 - Programação Orientada a Objetos

Arquivos - Parte 01

Prof.: Leonardo Tórtoro Pereira

leonardop@usp.br

Arquivos

- Em Java, existem 2 pacotes principais para leitura e escrita de dados. A leitura pode ser dividida em Streams e File
  - I/O Streams (java.io)
  - File I/O (java.nio e em menor parte na java.io)

I/O Streams

I/O Streams

- Uma I/O Stream representa uma fonte de entrada ou destino de saída
- A *Stream* pode representar vários tipos diferentes de fontes e destinos, incluindo:
  - Arquivos de disco
  - Dispositivos
  - Outros programas
  - Arrays de memória

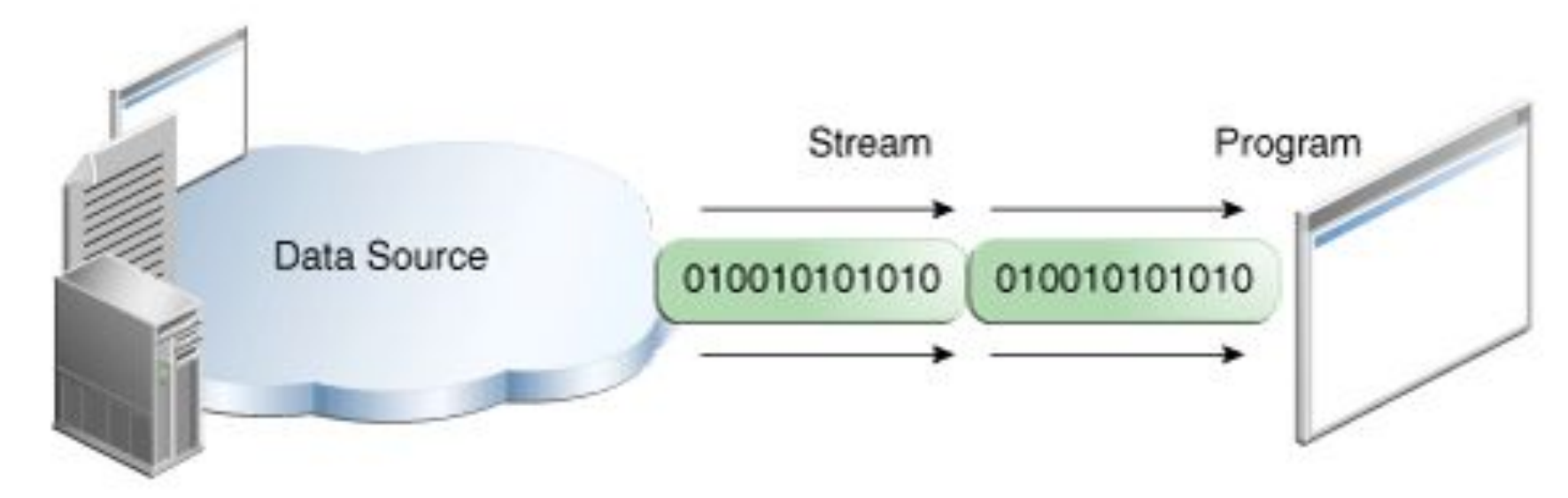
I/O Streams

- Ela também suporta diferentes tipos de dados, como:
  - Bytes simples
  - Tipos primitivos de dados
  - Caracteres localizados
  - Objetos
- Algumas *streams* apenas repassam dados
- Outras manipulam e transformam os dados de maneiras úteis

I/O Streams

- Independente do funcionamento interno da *stream*, ela apresenta o mesmo modelo simples para os programas que a usam:
  - Uma sequência de dados
- Um programa usa uma *input stream* para ler dados de uma fonte, um item por vez

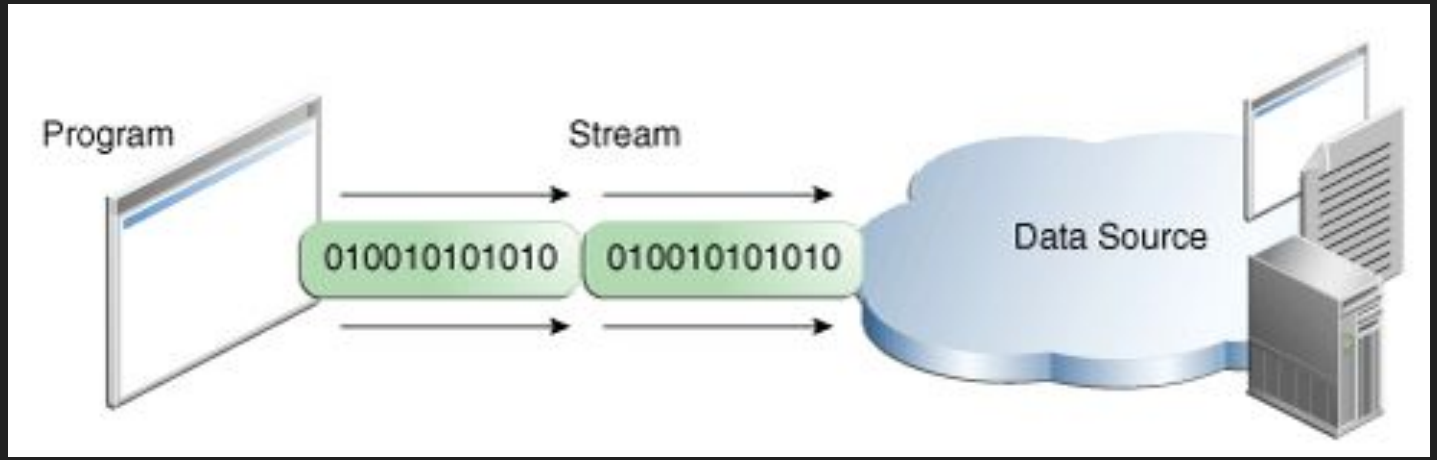
I/O Streams



Lendo dados de um programa. Fonte:
https://docs.oracle.com/javase/tutorial/essential/io/streams.html

I/O Streams

➔ Um programa usa um *output stream* para escrever dados em um destino, um item por vez



Escrevendo dados de um programa.

Fonte: https://docs.oracle.com/javase/tutorial/essential/io/streams.html

Byte Streams

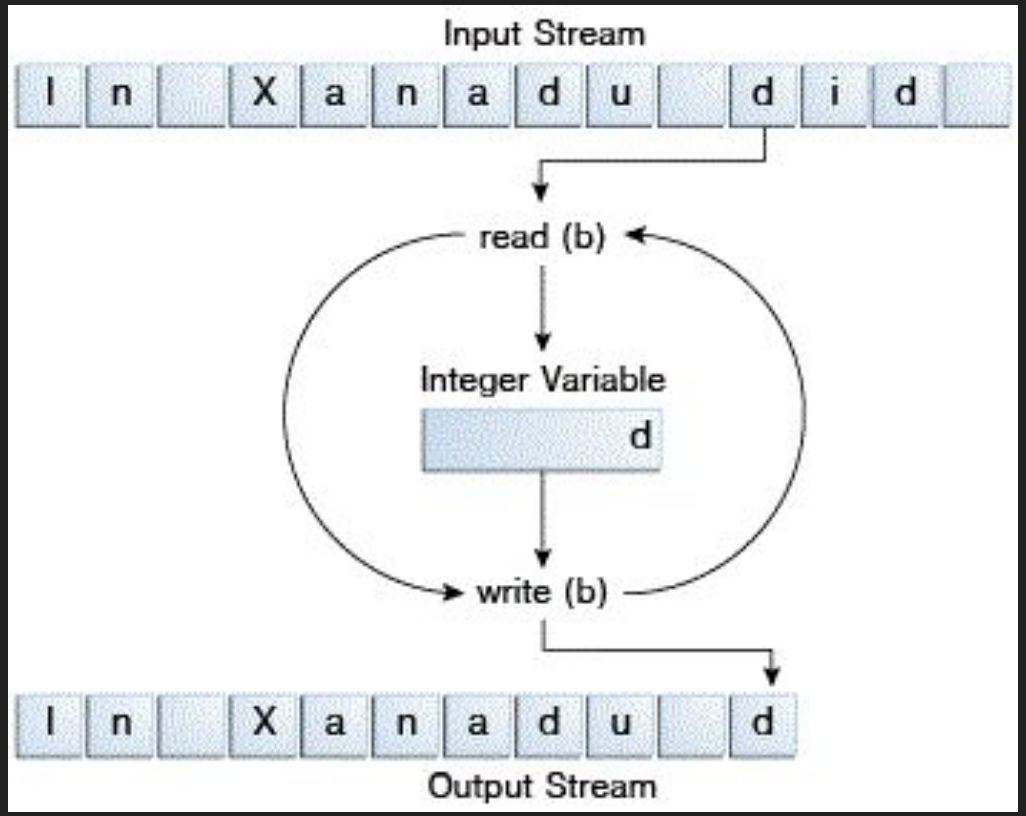
Byte Streams

- Realiza operações com bytes (8-bits). Todas as classes de *byte stream* herdam de *InputStream* e *OutputStream*
  - Existem várias classes de *byte stream*
  - Vamos ver um exemplo usando as da *file I/O*
    - *FileInputStream* e *FileOutputStream*
- Vamos copiar um arquivo de entrada, um byte por vez :)

Exemplo ByteStream

```
FileInputStream in = null;
FileOutputStream out = null;

try {
    in = new FileInputStream("aizen.txt");
    out = new FileOutputStream("outagain.txt");
    int c;
    while ((c = in.read()) != -1) {
        out.write(c);
    }
}
finally {
    if (in != null) { in.close(); }
    if (out != null) { out.close(); }
}
```



Fluxo de leitura e escrita. Fonte:
https://docs.oracle.com/javase/tutorial/essential/io/bytestreams.html

Byte Streams

- Feche sempre as *streams*!
  - *nomeDaStream.close()*
- Evita o vazamento de recursos!
- Um jeito bom de fazer isso é usando a palavra-chave *finally*, que veremos melhor nas aulas de exceções
- Leitura de *bytes* é bem baixo nível. Se o que for lido/escrito for algo melhor definido, evite o *byte*

Character Streams

Character Streams

- Os caracteres em Java são salvos usando a formatação Unicode
- Uma *Character Stream* automaticamente traduz essa formatação interna *de* e *para* o conjunto de caracteres locais
  - No ocidente, normalmente é um superconjunto de 8-bit do ASCII

Character Streams

- Na maioria dos casos, usar a *Character* é tão simples quanto usar a *Byte Stream*
  - As entradas e saídas são automaticamente traduzidas ao conjuntos de caracteres locais
  - O programa fica preparado para internacionalização sem nenhum trabalho extra do programador

Character Streams

- As classes de *character stream* herdam de *Reader* e *Writer*
  - Nesse caso, usamos a *FileReader* e a *FileWriter*

Exemplo *Character*Stream

```
FileReader inputStream = null;
FileWriter outputStream = null;

try {
    inputStream = new FileReader("data/aizen.txt");
    outputStream = new FileWriter("data/characteroutput.txt");

    int c;
    while ((c = inputStream.read()) != -1) {
        outputStream.write(c);
    }
}
finally {
    if (inputStream != null) { inputStream.close(); }
    if (outputStream != null) { outputStream.close(); }
}
```

Character Streams

- Os exemplos são bem parecidos, mudando principalmente as classes que lidam com os arquivos
- Os *character streams* atuam como "*wrappers*" de *byte streams*, usando esta para a parte física da entrada e saída, enquanto a *stream* de caracteres lida com a transição entre os tipos de dado.

Character Streams

- ➔ É possível ler os dados por linhas inteiras ao invés de caractere por caractere!
- ➔ Para isso, é preciso usar as classes *BufferedReader* e *PrintWriter*

Exemplo *Character*Stream

```
BufferedReader inputStream = null;
PrintWriter outputStream = null;

try {
    inputStream = new BufferedReader(new FileReader("data/aizen.txt"));
    outputStream = new PrintWriter(new FileWriter("data/linesoutput.txt"));
    String l;
    while ((l = inputStream.readLine()) != null) {
        outputStream.println(l);
    }
} finally {
    if (inputStream != null) { inputStream.close(); }
    if (outputStream != null) { outputStream.close(); }
}
```

Character Streams

- *readLine* retorna uma linha de texto
- Cada linha é escrita usando *println* que escreve a linha e adiciona ao final o terminador de linha do sistema operacional em questão
  - Pode ser diferente do terminador do arquivo lido!

Buffered Streams

Buffered Streams

- ➔ Em uma operação de I/O que não usa um *buffer*, cada requisição de leitura e escrita é manipulada diretamente pelo SO
- ➔ Isso pode reduzir a eficiência do programa, já que cada requisição normalmente requer um acesso a disco, atividade de rede ou outra operação relativamente cara

Buffered Streams

- Ao usarmos *buffered streams* reduzimos esses problemas.
- Eles lêem/escrevem dados de uma área de memória conhecida como *buffer*
  - A API de entrada/saída nativa só é chamada quando o *buffer* esvazia (no caso da entrada) ou enxe (no caso da saída).

Buffered Streams

- É possível converter uma *stream* sem *buffer* para usar *buffer* com empacotadores
  - É preciso passar o objeto da *stream* sem *buffer* para o construtor de uma classe *buffered stream*

```
inputStream = new BufferedReader(new FileReader("input.txt"));
outputStream = new BufferedWriter(new FileWriter("output.txt"));
```

Buffered Streams

- ➔ Existem 4 classes de *buffereds stream* para empacotar outras *streams*
  - Para bytes:
    - *BufferedInputStream*
    - *BufferedOutputStream*
  - Para caracteres:
    - *BufferedReader*
    - *BufferedWriter*

Buffered Streams

- Alguns comandos podem esvaziar o *buffer* antes dele estar cheio, conhecido como *autoflush*.
- Um *PrintWriter* com *autoflush* esvazia o *buffer* a cada chamada de *println*
- Para fazer isso manualmente, é possível chamar o método *flush()*

Data Streams

Data Streams

- Suportam I/O binária de tipos primitivos de dado e Strings
- Implementam a interface *DataInput* ou a *DataOutput*
  - As mais famosas são *DataInputStream* e *DataOutputStream*
- Vamos ver um exemplo usando a tabela seguinte

Order in record	Data type	Data description	Output Method	Input Method	Sample Value
1	`double`	Item price	`DataOutputStream.writeDouble`	`DataInputStream.readDouble`	`19.99`
2	`int`	Unit count	`DataOutputStream.writeInt`	`DataInputStream.readInt`	`12`
3	`String`	Item description	`DataOutputStream.writeUTF`	`DataInputStream.readUTF`	`"Java T-Shirt"`

Fonte:
https://docs.oracle.com/javase/tutorial/essential/io/datastreams.html

Exemplo DataStream

```
static final String dataFile = "invoicedata";

static final double[] prices = { 19.99, 9.99, 15.99, 3.99,
4.99 };
static final int[] units = { 12, 8, 13, 29, 50 };
static final String[] descs = {
    "Java T-shirt",
    "Java Mug",
    "Duke Juggling Dolls",
    "Java Pin",
    "Java Key Chain"
};
```

Exemplo DataStream

```
DataOutputStream out = null;

try {
    out = new DataOutputStream(new
            BufferedOutputStream(new FileOutputStream(dataFile)));

    for (int i = 0; i < prices.length; i ++) {
        out.writeDouble(prices[i]);
        out.writeInt(units[i]);
        out.writeUTF(descs[i]);
    }
} finally { out.close(); }
```

Exemplo DataStream

```
DataInputStream in = null;
double total = 0.0;
try {
    in = new DataInputStream(new
            BufferedInputStream(new
FileInputStream(dataFile)));

    double price;
    int unit;
    String desc;
```

Exemplo DataStream

```
try {
    while (true) {
        price = in.readDouble();
        unit = in.readInt();
        desc = in.readUTF();
        System.out.format("You ordered %d units of %s at
$%.2f%n",
                unit, desc, price);
        total += unit * price;
    }
} catch (EOFException e) { }
```

Data Streams

- É possível detectar uma condição de fim de arquivo com a EOFException
  - Vamos ver sobre exceções em breve :)
- É dever do programador fazer com que cada *read* e *write* especializado (com o tipo de dado correto) corresponda ao que está no arquivo
  - O *input stream* consiste em dados binários, sem um indicador do tipo dos valores ou onde eles começam

Object Streams

Object Streams

- Suporta I/O de objetos
- A maioria das classes padrões suporta a serialização dos objetos
  - São marcadas com a interface *Serializable*
- As classes são *ObjectInputStream* e *ObjectOutputStream*
  - Implementam *ObjectInput* e *ObjectOutput*
    - Que são subinterfaces de *DataInput* e *DataOutput*

Object Streams

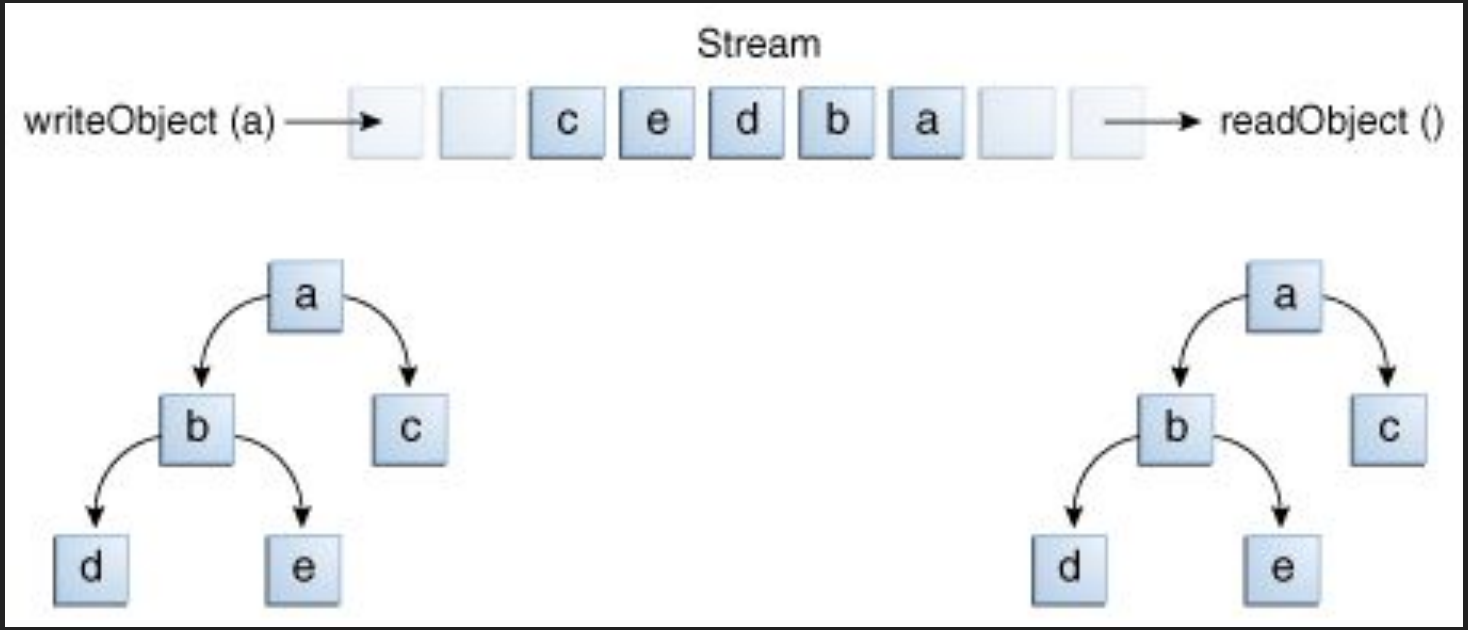
- Ou seja, todos os métodos para tipos primitivos vistos anteriormente em *Data Streams* são válidos
  - Pode haver mistura entre tipos primitivos e objetos
- Se *readObject()* não retornar o tipo esperado de objeto, tentar dar *cast* ao tipo correto pode dar uma exceção de *ClassNotFoundException*.

Object Streams

- Os métodos *readObject* e *writeObject* são simples de usar, mas contém uma lógica de manuseamento complexa.
- Para objetos com referências para outros, isso é importante.
- Se *readObject* precisa reconstruir um objeto de uma *stream*, ele precisa reconstruir TODOS os objetos que o original possui referência.

Object Streams

- E se esses objetos possuem referências, estas também precisa reconstruí-las e assim por diante...
- Nessa situação, *writeObject* atravessa toda a cadeia de referências a objetos e escreve todos os objetos dessa cadeia em uma *stream*.
- Uma simples chamada a *writeObject* pode causar a escrita de uma grande quantidade de objetos na *stream*



Fonte:
https://docs.oracle.com/javase/tutorial/essential/io/objectstreams.html

Object Streams

- Caso você escreva duas vezes o mesmo objeto em uma mesma *stream*, ele só irá escrever uma vez.
  - Ou seja, ao ler ambos os objetos, as duas referências apontarão pro mesmo objeto.
- Mas caso seja escrito em *streams* separadas, cada uma retornará um objeto diferente

Referências

1. https://docs.oracle.com/javase/tutorial/essential/io/streams.html