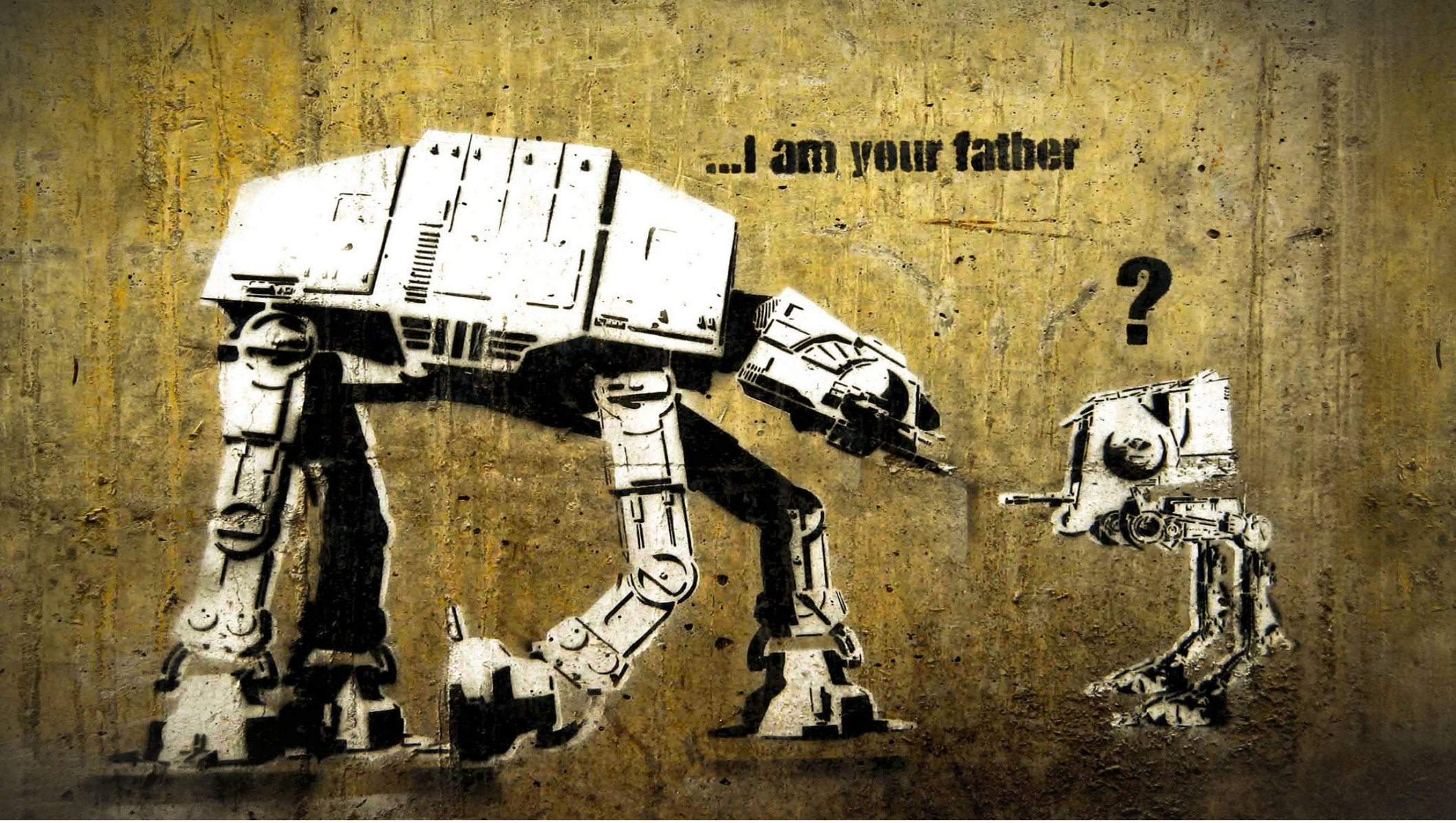


14.Arquivos e Serialização

Prof. Alexandre Krohn

Roteiro

- Arquivos
  - Arquivos Texto
  - Arquivos Binários
    - Arquivos de Objetos
      - Serialização
  - Acessando o sistema de arquivos

Arquivos

- Arquivos são a “metáfora” utilizada por sistemas operacionais para representar o mecanismo de **armazenamento de informações em meio persistente**

Arquivos

- **Meio persistente** é todo aquele que preserva os dados mesmo quando desconectado da energia elétrica
- Ex.: Disco rígido (HDD), pendrive, CD/DVD/Bluray, Discos de estado sólido (SSD)

Arquivos e o SO

- Note que nunca manipulamos um arquivo diretamente, nem sabemos em qual parte do meio persistente o mesmo está gravado
- Isso cabe ao sistema operacional, que faz isso usando tabelas de alocação de arquivos. Ex.: FAT, NTFS, ext4, etc... e comandos específicos para acesso.

Arquivos

- Arquivos são portanto, cópias armazenadas dos dados que estavam na memória, prontos para serem trazidos de volta à memória e/ou enviados(copiados) para outro local.

Tipos de Arquivos

- Há dois tipos básicos de arquivos:
  - Arquivos texto
  - Arquivos binários

Roteiro

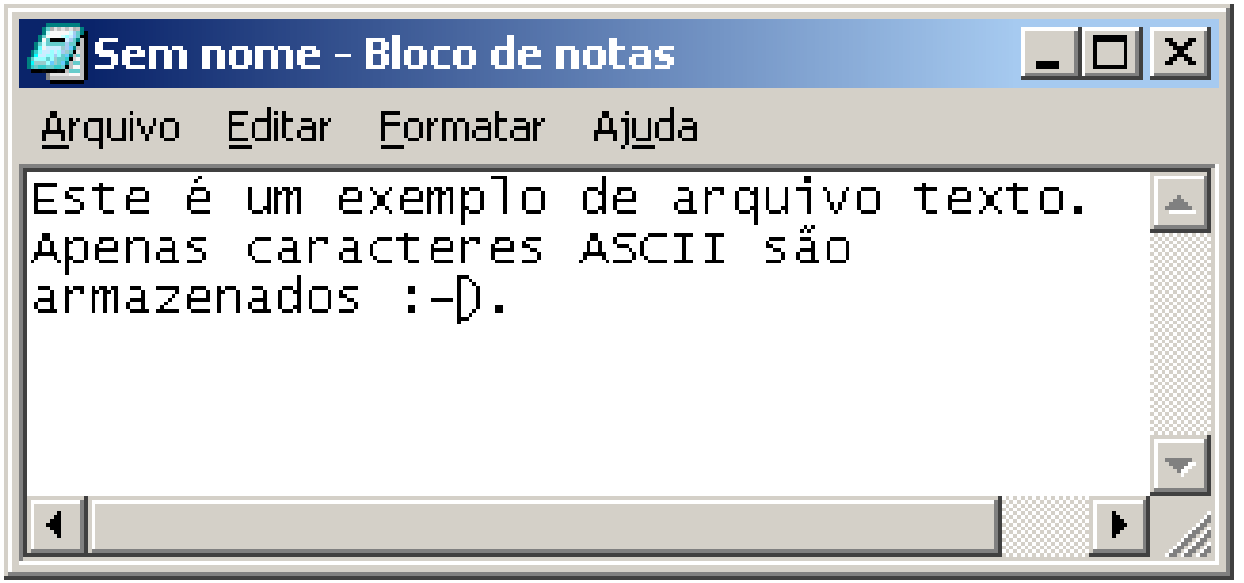
- Arquivos
  - Arquivos Texto
  - Arquivos Binários
    - Arquivos de Objetos
      - Serialização
  - Acessando o sistema de arquivos

Arquivos Texto

- Nos arquivos de texto os dados são armazenados em disco em byte por byte (um caracter ASCII).
- Arquivos de texto podem ser abertos por qualquer editor de texto
- Exemplos : arquivos .txt. .c, .cpp, .java, .py, códigos fonte em geral, .xml, .html

Arquivos Texto

- Arquivos texto são visualizados facilmente por editores como notepad, VI, notepad++, Gedit, Sublime...



Arquivos e Java

- As classes para acesso à aqruivos em Java estão no pacote ***java.io***
- **Sempre** que trabalhamos com arquivos, há a possibilidade de ocorrência de exceções, por isso sempre usa-se ***try/catch***

Java.io.IOException

- Todas as exceções de manipulação de arquivos herdam de ***IOException***

```
try {

    // manipulação dos arquivos

} catch (IOException ioe) {
    ioe.printStackTrace();
}
```

Arquivos Texto e Java

- Em Java, podemos manipular arquivos texto de maneiras diferentes
  - Lendo/escrevendo byte-a-byte
  - Usando buffers
- Seja qual a forma escolhida, precisaremos dos ***Streams***

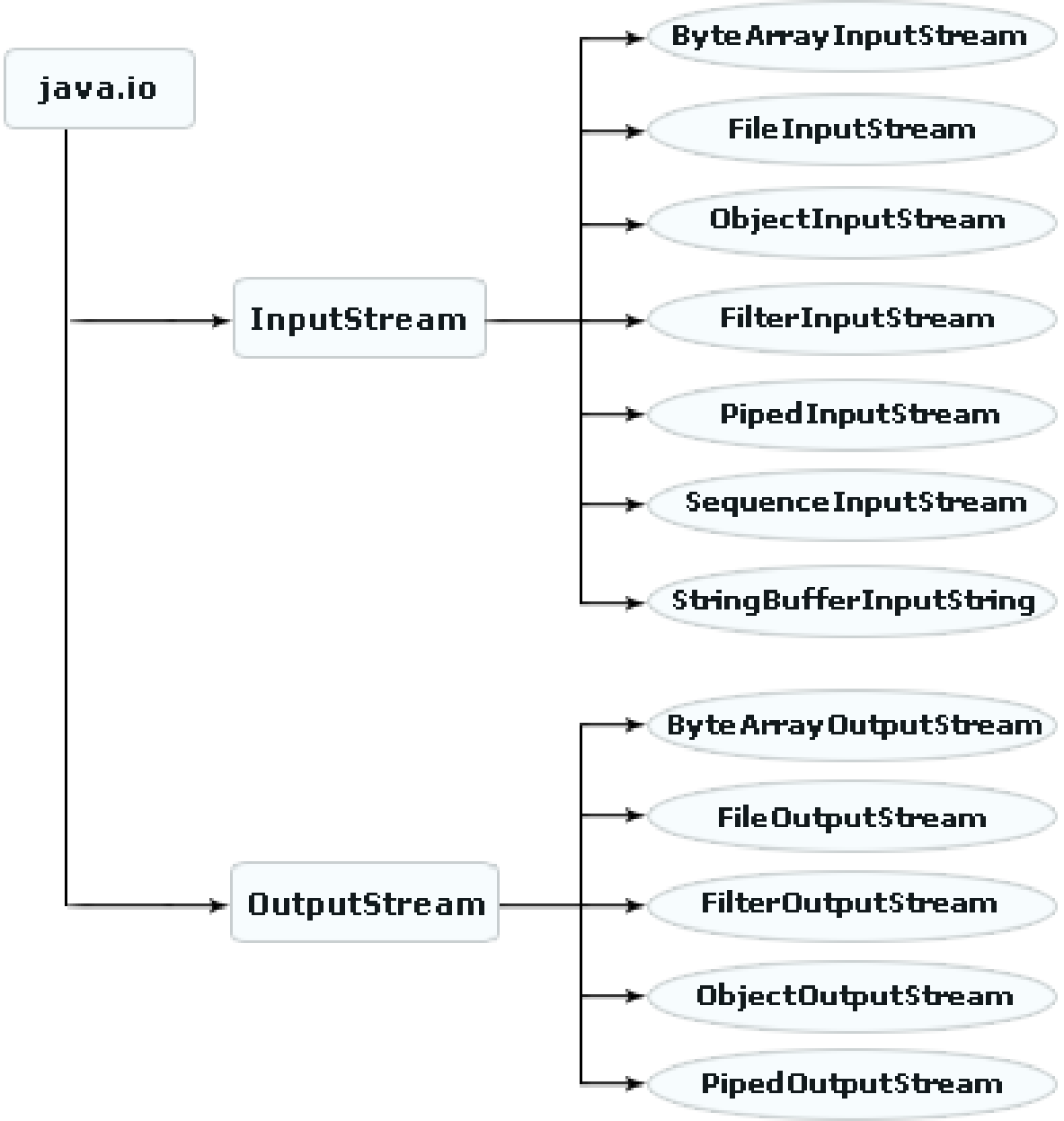
Streams de dados

- Um ***Stream*** é um fluxo de dados. Quando queremos ler ou gravar dados usamos um Stream.
- Há Streams de entrada e de saída
  - ***InputStream*** : dados entrando na memória, sendo lidos de uma fonte
  - ***OutputStream*** : dados saindo da memória para um outro meio

Streams : Implementações

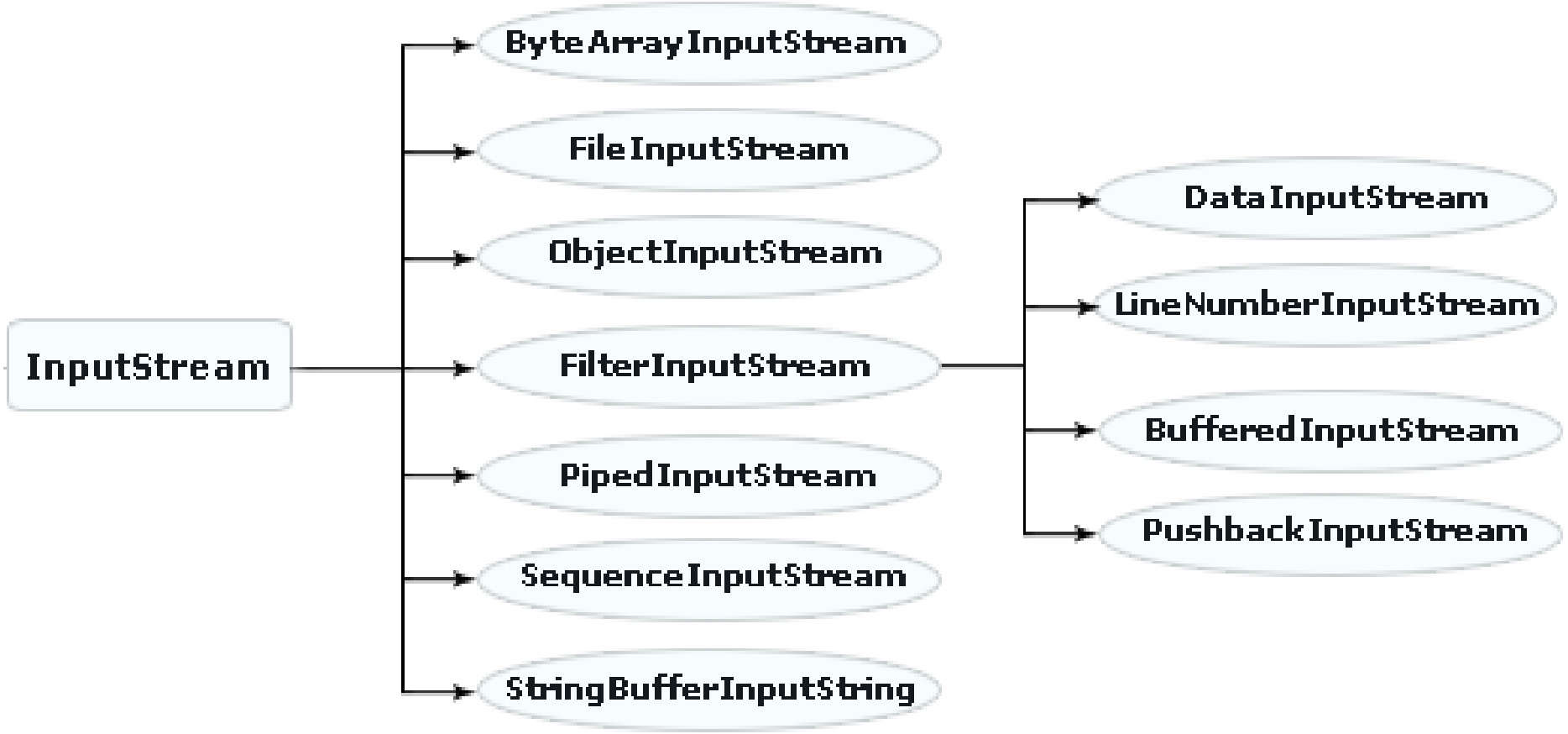
- ***InputStream*** e ***OutputStream*** são classes abstratas
- Sempre usaremos implementações específicas
- Há tipos de Streams diferentes para **tipos de informações** e **tipos de transportes** diferentes

Streams : Implementações



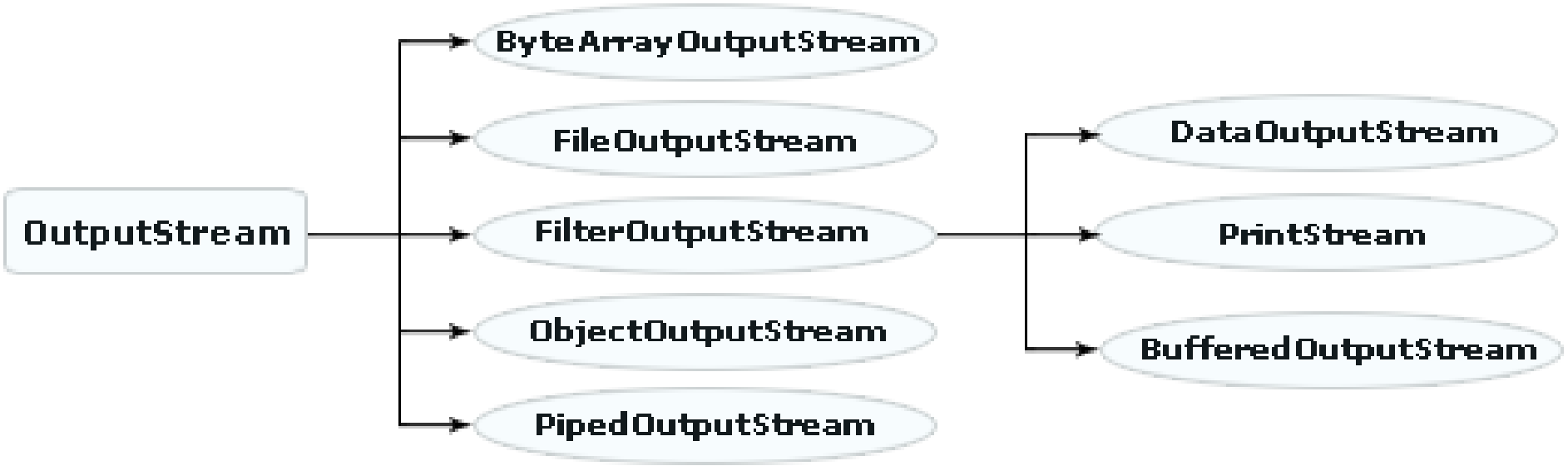
InputStream

- Para **ler** dados



OutputStream

- Para **escrever** dados



Arquivos Texto byte-a-byte

- Exemplo de **leitura** de arquivo texto, byte por byte, colocando-se cada byte lido em uma String
- Ou seja, o texto é inteiramente lido para uma String
- Usaremos um ***InputStream***

Arquivos Texto byte-a-byte

LEITURA

```
StringBuilder sb = new StringBuilder();

try {
    File f = new File("arquivo.txt");
    FileInputStream fi = new FileInputStream(f);
    int i = 0;
    while (i != -1) {
        i = fi.read();
        char c = (char) i;
        sb.append(c);
    }
    fi.close();
} catch (Exception e) {
    e.printStackTrace();
}
```

Arquivos Texto byte-a-byte

LEITURA

```
StringBuilder sb = new StringBuilder();

try {
    File f = new File("arquivo.txt");
    FileInputStream fi = new FileInputStream(f);
    int i = 0;
    while (i != -1) {
        i = fi.read();
        char c = (char) i;
        sb.append(c);
    }
    fi.close();
} catch (Exception e) {
    e.printStackTrace();
}
```

Começa-se obtendo a referência para o arquivo a ser manipulado. Para isso, usa-se a classe **File**

Arquivos Texto byte-a-byte

LEITURA

```
StringBuilder sb = new StringBuilder();

try {
    File f = new File("arquivo.txt");
    FileInputStream fi = new FileInputStream(f);
    int i = 0;
    while (i != -1) {
        i = fi.read();
        char c = (char) i;
        sb.append(c);
    }
    fi.close();
} catch (Exception e) {
    e.printStackTrace();
}
```

Depois, abre-se uma **FileInputStream** a partir do arquivo referenciado

Arquivos Texto byte-a-byte

LEITURA

```
StringBuilder sb = new StringBuilder();

try {
    File f = new File("arquivo.txt");
    FileInputStream fi = new FileInputStream(f);
    int i = 0;
    while (i != -1) {
        i = fi.read();
        char c = (char) i;
        sb.append(c);
    }
    fi.close();
} catch (Exception e) {
    e.printStackTrace();
}
```

O próximo passo
é fazer um **loop** sobre
os dados do Stream, que
é lido com o método
read()

Arquivos Texto byte-a-byte

LEITURA

```
StringBuilder sb = new StringBuilder();

try {
    File f = new File("arquivo.txt");
    FileInputStream fi = new FileInputStream(f);
    int i = 0;
    while (i != -1) {
        i = fi.read();
        char c = (char) i;
        sb.append(c);
    }
    fi.close();
} catch (Exception e) {
    e.printStackTrace();
}
```

Por fim, o Stream deve ser fechado, ou o arquivo fica “preso” e o SO não permite mais o seu acesso

Arquivos Texto byte-a-byte

LEITURA

```
StringBuilder sb = new StringBuilder();

try {
    File f = new File("arquivo.txt");
    FileInputStream fi = new FileInputStream(f);
    int i = 0;
    while (i != -1) {
        i = fi.read();
        char c = (char) i;
        sb.append(c);
    }
    fi.close();
} catch (Exception e) {
    e.printStackTrace();
}
```

Sem esquecer que todo o código deve ser escrito dentro de um bloco **try / catch**

Arquivos Texto byte-a-byte

- Para escrever no arquivo, iteraremos sobre os bytes do String e os enviaremos ao mesmo usando um ***OutputStream***
- O procedimento é o mesmo, apenas a direção do fluxo de dados é invertida

Arquivos Texto byte-a-byte

ESCRITA

```
try {
    File f = new File("arquivo.txt");
    FileOutputStream fos = new FileOutputStream(f);

    byte[] bytes = texto.getBytes();

    for(int i = 0; i < bytes.length; i++) {
        fos.write(bytes[i]);
    }

    fos.close();
} catch (Exception e) {
    e.printStackTrace();
}
```

Perceba-se aqui o uso de **FileOutputStream** para estabelecer o fluxo de saída dos dados e do método **write(…)** para escrevê-los no arquivo

Operações bufferizadas

- Os sistemas operacionais não lêem e escrevem do disco em bytes.
  - A performance não seria boa
- Os SO lêem pedaços maiores de informação de uma vez só, e as armazenam em buffers, que são áreas temporárias.

Arquivos Texto Bufferizados

LEITURA

```
try {
    FileInputStream fi = new FileInputStream("arquivo.txt");
    InputStreamReader ir = new InputStreamReader(fi);
    BufferedReader br = new BufferedReader(ir);
    String linha;
    while ((linha = br.readLine()) != null) {
        System.out.println(linha);
    }
    fi.close();
} catch (Exception e) {
    e.printStackTrace();
}
```

Para leitura bufferizada, usa-se **FileInputStream** para estabelecer o fluxo de entrada dos dados

Arquivos Texto Bufferizados

LEITURA

```
try {
    FileInputStream fi = new FileInputStream("arquivo.txt");
    InputStreamReader ir = new InputStreamReader(fi);
    BufferedReader br = new BufferedReader(ir);
    String linha;
    while ((linha = br.readLine()) != null) {
        System.out.println(linha);
    }
    fi.close();
} catch (Exception e) {
    e.printStackTrace();
}
```

Na sequência, é necessário um objeto de **InputStreamReader** para ler os dados do arquivo

Arquivos Texto Bufferizados

LEITURA

```
try {
    FileInputStream fi = new FileInputStream("arquivo.txt");
    InputStreamReader ir = new InputStreamReader(fi);
    BufferedReader br = new BufferedReader(ir);
    String linha;
    while ((linha = br.readLine()) != null) {
        System.out.println(linha);
    }
    fi.close();
} catch (Exception e) {
    e.printStackTrace();
}
```

Um objeto **BufferedReader** servirá para o buffer dos dados lidos

Arquivos Texto Bufferizados

LEITURA

```
try {
    FileInputStream fi = new FileInputStream("arquivo.txt");
    InputStreamReader ir = new InputStreamReader(fi);
    BufferedReader br = new BufferedReader(ir);
    String linha;
    while ((linha = br.readLine()) != null) {
        System.out.println(linha);
    }
    fi.close();
} catch (Exception e) {
    e.printStackTrace();
}
```

Os textos são lidos então, linha por linha, através do método **readLine()** do buffer

Arquivos Texto Bufferizados

ESCRITA

- Para **escrita** do arquivo texto usando buffers, envia-se todos os bytes de uma única vez!

Arquivos Texto Bufferizados

ESCRITA

```
try {
    File f = new File("arquivo.txt");
    FileOutputStream fo = new FileOutputStream(f);
    fo.write(texto.getBytes());
    fo.close();
    System.out.println("Arquivo gravado com sucesso");
} catch (Exception e) {
    e.printStackTrace();
}
```

O método **write(…)** é sobrecarregado e consegue enviar um vetor de bytes em uma única operação

Arquivos Texto Bufferizados

ESCRITA

- Outra opção é utilizar a classe ***FileWriter***, que já se encarrega de fazer a conversão dos Strings para char (bytes) automaticamente.

FileWriter

```
try {
    FileWriter fileWriter = new FileWriter("file1.txt");
    fileWriter.write("Dado gravado com FileWriter");
    fileWriter.close();
} catch (Exception e) {
    e.printStackTrace();
}
```

FileWriter com PrintWriter

- Nesse caso, o ***FileWriter*** é ”decorado” com a classe ***PrintWriter***, e podemos usar os métodos ***print()*** e ***println()***

FileWriter com PrintWriter

```
try {
    FileWriter fileWriter = new FileWriter("file2.txt");
    PrintWriter printWriter = new PrintWriter(fileWriter);
    printWriter.println("Dado gravado com PrintWriter");
    printWriter.close();
    fileWriter.close();
} catch (Exception e) {
    e.printStackTrace();
}
```

Roteiro

- Arquivos
  - Arquivos Texto
  - Arquivos Binários
    - Arquivos de Objetos
      - Serialização
  - Acessando o sistema de arquivos

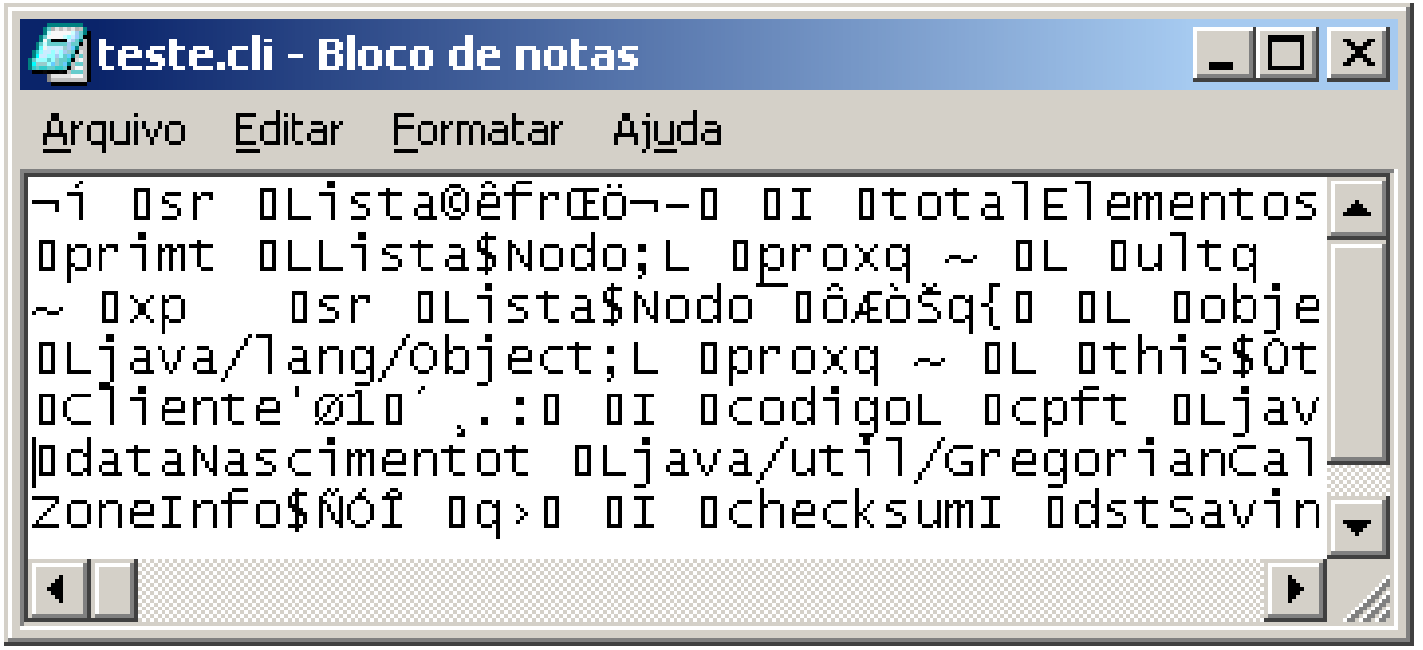
Arquivos binários

- Nos arquivos binários, os dados podem ser gravados utilizando:
  - O mesmo formato dos dados na memória (raw)
  - Formatos específicos de arquivo, que descrevem a organização interna de seus conteúdos. Ex.: .doc, .pdf, .exe

Arquivos Binários

- Arquivos binários **NÃO** podem ser lidos através de editores de texto
- Eles necessitam de **programas específicos** capazes de "interpretar" seus conteúdos

Arquivos Binários



- Um arquivo binario aberto em um editor de texto geralmente apresenta um conteúdo ilegível para humanos

Arquivos Binários

- Arquivos binários em Java tambem podem ser manipuladas através de duas formas:
  - Por acesso randômico
  - Por serialização de objetos

Roteiro

- Arquivos
  - Arquivos Texto
  - Arquivos Binários
    - Arquivos de Objetos
      - Serialização
  - Acessando o sistema de arquivos

Arquivos de Objetos

- Em Java, podemos gravar objetos em arquivos.
- Esse processo é conhecido como **serialização**

Serialização

- Para que um objeto possa ser gravado em arquivo (ou enviado pela rede) ele precisa implementar a interface ***Serializable***

Serializable

```
import java.io.Serializable;

public class Aluno implements Serializable{

    private static final long serialVersionUID = 1L;

    private int codigo;
    private String nome;

    public Aluno() {
    }
```

A interface **Serializable** não possui nenhum método. ela somente serve para indicar que a classe pode ser gravada

Gravando um Objeto

```
Aluno aluno = new Aluno(1,"Mickey");

try {
    FileOutputStream fo = new FileOutputStream("mickey.dat");
    ObjectOutputStream ou = new ObjectOutputStream(fo);
    ou.writeObject(aluno);
    ou.close();
    fo.close();
} catch (IOException e) {
    e.printStackTrace();
}
```

Note-se que para gravar um objeto, foram utilizados **ObjectOutputStream** e o método **writeObject()**

Lendo um objeto

```
Aluno a = null;

try {
    FileInputStream fi = new FileInputStream("mickey.dat");
    ObjectInputStream oi = new ObjectInputStream(fi);
    Object o = oi.readObject();
    a = (Aluno) o;
    oi.close();
    fi.close();
} catch (IOException e) {
    e.printStackTrace();
} catch (ClassNotFoundException e) {
    e.printStackTrace();
}
```

Para ler um objeto, foram utilizados **ObjectInputStream** e o método **readObject()**

Lendo um objeto

```
Aluno a = null;

try {
     FileInputStream fi = new FileInputStream("mickey.dat");
     ObjectInputStream oi = new ObjectInputStream(fi);
     Object o = oi.readObject();
     a = (Aluno) o;
     oi.close();
     fi.close();
} catch (IOException e) {
     e.printStackTrace();
} catch (ClassNotFoundException e) {
     e.printStackTrace();
}
```

Como todos osobjetos são lidos como Object, é necessário usar ***cast*** para convertê-los para a classe desejada

Serialização

- Um problema apresentado na serialização é que só é possível serializar um tipo de objeto em um arquivo, e um objeto por arquivo

Gravação de múltiplos objetos

- Quando se necessita gravar coleções de objetos, uma técnica bastante utilizada é gravar uma lista de objetos

Lendo listas

```
List<Aluno> lista = new ArrayList<>();

File f = new File("alunos.dat");

if (f.exists()) {

    try {
        FileInputStream fi = new FileInputStream("alunos.dat");
        ObjectInputStream oi = new ObjectInputStream(fi);
        Object o = oi.readObject();
        lista = (List<Aluno>) o;
        oi.close();
        fi.close();
    } catch (Exception e) {
        e.printStackTrace();
    }
}
```

Esse exemplo é igual ao anterior, mudando apenas o tipo do objeto lido e acrescentando a verificação de existência do arquivo

Escrevendo listas

```
List<Aluno> alunos = new ArrayList<>();

alunos.add(new Aluno(1,"Donald"));
alunos.add(new Aluno(2,"Mickey"));
alunos.add(new Aluno(3,"Pateta"));

try {
    FileOutputStream fo = new FileOutputStream("alunos.dat");
    ObjectOutputStream ou = new ObjectOutputStream(fo);
    ou.writeObject(alunos);
    ou.close();
    fo.close();
} catch (Exception e) {
    e.printStackTrace();
}
```

Roteiro

- Arquivos
  - Arquivos Texto
  - Arquivos Binários
    - Arquivos de Objetos
      - Serialização
  - Acessando o sistema de arquivos

Acesso ao sistema de arquivos

- O acesso ao sistema de arquivos também é feito através de classes do pacote ***java.io***
- A classe principal é a classe ***File***
- Quando executamos :

```
File f = new File(“teste.txt”);
```

- Não estamos criando um novo arquivo, apenas obtendo uma referência para um arquivo chamado “teste.txt”

Classe File : Métodos

- boolean **exists()** // Retorna true se o arquivo existir
- boolean **isFile()** // Retorna true se for um arquivo
- boolean **isDirectory()** // Retorna true se for um diretório
- String **getName()** // Retorna nome do arquivo/diretório
- int **length()** // Retorna o número de bytes (tamanho) de um arquivo
- String **getPath()** // Retorna o do caminho do arquivo
- boolean **delete()** // Apaga o arquivo
- boolean **renameTo(f2)** // Renomeia o arquivo para f2
- boolean **createNewFile()** // Cria o arquivo

Exemplo

Sempre inverta as barras nos nomes de arquivos

- Listando os conteúdos de um diretório

```
File diretorio = new File("c:/teste");

if (diretorio.isDirectory()) {
    for (String nomeDoArquivo : diretorio.list()) {
        String caminho = diretorio.getPath();
        File arquivo = new File(caminho + "/" + nomeDoArquivo);
        if (arquivo.isFile()) {
            System.out.print(arquivo.getName() + " - ");
            long tamanhoEmMB = arquivo.length() / 1024;
            System.out.println(tamanhoEmMB + "MB");
        }
    }
}
```

Dúvidas?

Atividades

- Execute as atividades presentes no documento

  14.Lista.de.Exercícios.POO.pdf

Próximos passos

- Acesso Randômico