

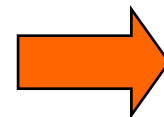
IO e Serialização em Java

Parte I

Conhecendo uma API

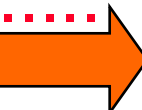
- Introdução

- Vamos passar agora a conhecer mais a APIs do Java
 - Que possuem as classes que você com certeza 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...
 - Não se preocupe em decorar nomes. Preocupe-se em entender
 - O interessante aqui é que você enxergue que todos os conceitos previamente estudados são aplicados em vários momentos nas classes da plataforma Java.



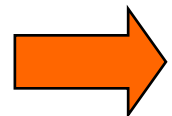
Assuntos abordados

- Introdução
 - Com isso em mente, agora vamos explorar os componentes mais importantes do pacote `java.io`
 - E outros recursos da linguagem relacionados à E/S e arquivos
 - O pacote `java.io` oferece abstrações que permitem ao programador lidar com:
 - ➡ Arquivos, diretórios e seus dados de uma maneira **independente** de plataforma. Isso é bastante interessante para a portabilidade.....
 - Mais detalhadamente podemos encontrar no `java.io`.....



O pacote java.io

-Em java.io podemos encontrar recursos para facilitar:
 - A manipulação de dados durante o processo de leitura ou gravação
 - Bytes sem tratamento
 - Caracteres Unicode
 - Dados filtrados de acordo com certo critério
 - Dados otimizados em buffers
 - Leitura/gravação automática de objetos
 - Entre outras coisitas a mais...



O pacote java.io

- Esses recursos são **mapeados em classes** como:

⇒ A classe `File`, que representa arquivos e diretórios

⇒ Objetos que implementam entrada e saída

- `InputStream` e `OutputStream`, `Readers` e `Writers`
- Compressão com GZIP streams
- `FileChannels`

⇒ Objeto que implementa arquivo de acesso aleatório

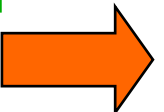
- `RandomAccessFile`

⇒ Recursos de serialização

- `Serializable`, `ObjectOutputStream` e `ObjectInputStream`

- etc.... Mas antes de utilizar as classes....

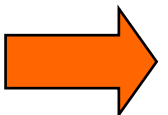
– Vamos começar entendendo um conceito muito importante de E/S em Java....



I/O em Java

- Em Java, as informações são:
 - Armazenada ou gravadas e apanhada ou lidas usando um sistema de comunicação chamado *streams*.
- É possível criar:
 - ⇒ *streams* de entrada para ler informações e
 - ⇒ *streams* de saída para armazenar informações.
-ou seja, os *streams* trabalham com o tráfego da informação
 - Seja a informação de disco, da Internet, teclado ou de outros programas.

⇒ Mas que raios são *streams*?.....



I/O em Java

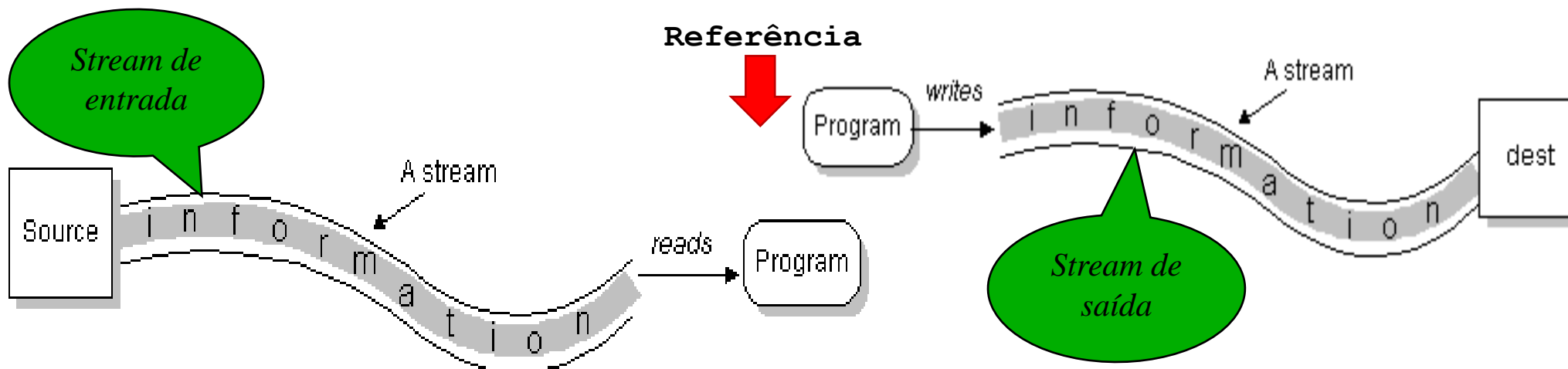
- Mas, o que são *streams*?
 - Um *stream* é o caminho atravessado pelos dados em um programa

⇒ Um *stream* de entrada

- Envia dados de uma origem para um programa

⇒ Um *stream* de saída

- Envia dados de um programa para um destino.



I/O em Java

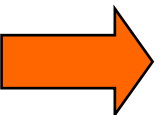
- Os *Streams* que podemos trabalhar em Java são:

Streams de bytes

- São usados para lidar com *bytes*, inteiros e outros tipos de dados simples
- Ex.: Programas executáveis, comunicações pela Internet e *bytecode* utilizam esse *stream*

Streams de caracteres

- Tratam de arquivos textos e outras fontes de textos
- Ex.: São um tipo especializado de *stream de bytes*, que trata somente de dados textuais, tais como arquivos texto, páginas Web como documentos HTML, dados do usuário, etc.



I/O em Java

- Os Streams que vamos trabalhar em Java são:

Streams de dados

- Permitem escrita e leitura de tipos primitivos diretamente (char, float, integer, double, etc)
- Ex.: Se precisar trabalhar com dados que não sejam representados como bytes ou caracteres, podemos usar os *streams* de entrada e saída de dados.
 - Estes *streams* filtram um *stream* de bytes existente de modo que os tipos primitivos possam ser lidos ou escritos diretamente do *stream*

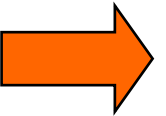
Streams de objetos

- Permite que os dados sejam representados como parte de um objeto
- Ex.: Tratam da persistência e recuperação de objetos como um todo para que um objeto seja salvo em um destino, como um arquivo de disco, por exemplo.

- Todos são implementados no pacote **java.io**

I/O em Java

- Blz,
 - Entendemos o que é um stream e os tipos que temos...
 - Mas como podemos efetivamente utilizar esses *stream*...???



I/O em Java

- Usando um *stream*

- O procedimento para usar um *stream* de *byte* ou caracteres em Java é praticamente o mesmo
 - Vamos entender o processo de criação e uso de *streams*.

➡ Para um *stream* de entrada.

- O primeiro passo é criar um objeto que esteja associado à origem de dados.
 - Ex.: Se a origem for um arquivo no disco, um objeto *FileInputStream* poderia ser associado a esse arquivo
- Depois que o objeto *stream* estiver associado, é possível ler informações a partir desse *stream*, usando um dos métodos do objeto
 - Ex.: No caso do exemplo acima, o método *read()*.

I/O em Java

- Usando um *stream*

➡ Para um *stream* de saída.

- O primeiro passo é criar um objeto que esteja associado ao destino dos dados.

– Ex.: Um objeto desse tipo pode ser o *BufferedWriter*

- Depois que o objeto *stream* estiver associado, é possível escrever informações a partir desse *stream*, usando um dos métodos do objeto

– Ex.: Método *writer()*

➡ Quando terminar de ler ou escrever os dados,

- Basta chamar o método *close()*, para indicar que terminou de usar o *stream*.

I/O em Java

- Por padrão, os algoritmos de E/S usando *streams* trabalham com a seguinte estrutura...
 - Usado tanto para obter como ler dados entre origem e destino

– Entrada de dados

Entrada:

Abrir fluxo

Enquanto houver informação

Ler informação

Fechar fluxo

– Saída de dados

Saída:

Abrir fluxo

Enquanto houver informação

Escrever informação

Fechar fluxo

I/O em Java

- Usando um *stream*

- **Resumindo:** Para usar um *stream* basta:

- ⇒ Criá-lo e

- ⇒ Chamar os seus métodos para enviar ou receber dados,

- Dependendo se é um *stream* de entrada ou saída.

I/O em Java

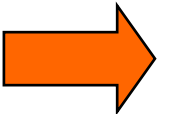
- Blz...

- Vamos começar a trabalhar.....

- Inicialmente vamos analisar alguns *stream* de bytes e de caracteres básicos...

- Para isso é necessário conhecer as classes que lidam com esses *streams*...

- Vamos ter classes específicas para lidar com *streams* específicos.....



I/O em Java

- Classes do *Stream* de *bytes*
 - Todos os *stream de bytes* são uma subclasse de *InputStream* ou *OutputStream*
 - ⇒ *InputStream*
 - Classe genérica (abstrata) para lidar com fluxos de bytes de entrada e nós de fonte (dados para leitura).
 - Método principal: *read()*
 - ⇒ *OutputStream*
 - Classe genérica (abstrata) para lidar com fluxos de bytes de saída e nós de destino (dados para gravação).
 - Método principal: *write()*
 - Essas são classes abstratas,
 - De modo que não é possível criar um *stream* diretamente dessas classes, somente das suas subclasses

I/O em Java

- Classes do *Stream* de caracteres

- As classes usadas para ler e escrever os **streams de caracteres** são todas subclasses de *Reader* ou *Writer*.

➡ Reader

- Classe abstrata para lidar com fluxos de caracteres de entrada
- Método principal: read()

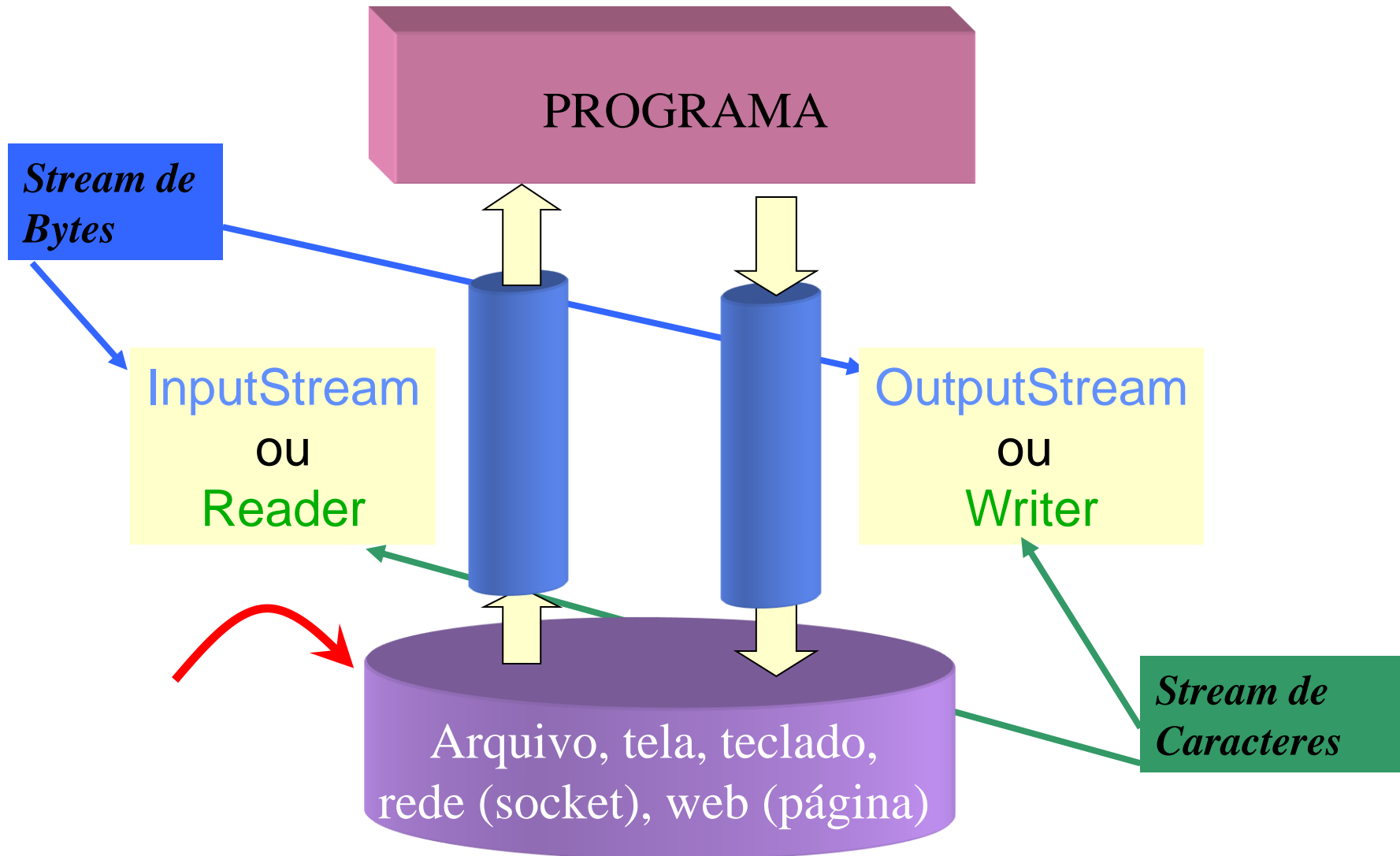
➡ Writer

- Classe abstrata para lidar com fluxos de bytes de saída:
- Método principal: write()

- Estas devem ser usadas para toda entrada de texto,
 - Em vez de lidar diretamente com *stream* de *bytes*

I/O em Java

- Resumindo: Fluxo dos dados e as classes utilizadas

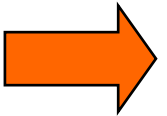


Hierarquia de Classes

- Blz,

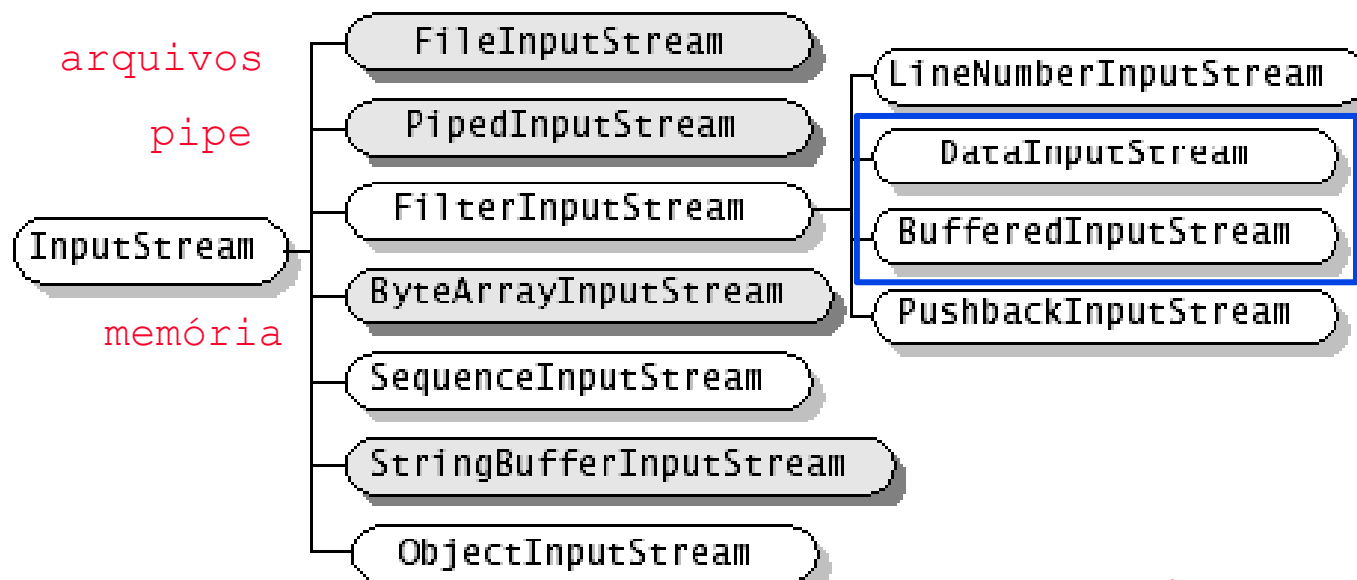
- Mas quais são as **classes concretas** com as quais a gente vai ter que trabalhar???...

- Para responder essa questão temos que analisar rapidamente a hierarquia de classes dessa api.....

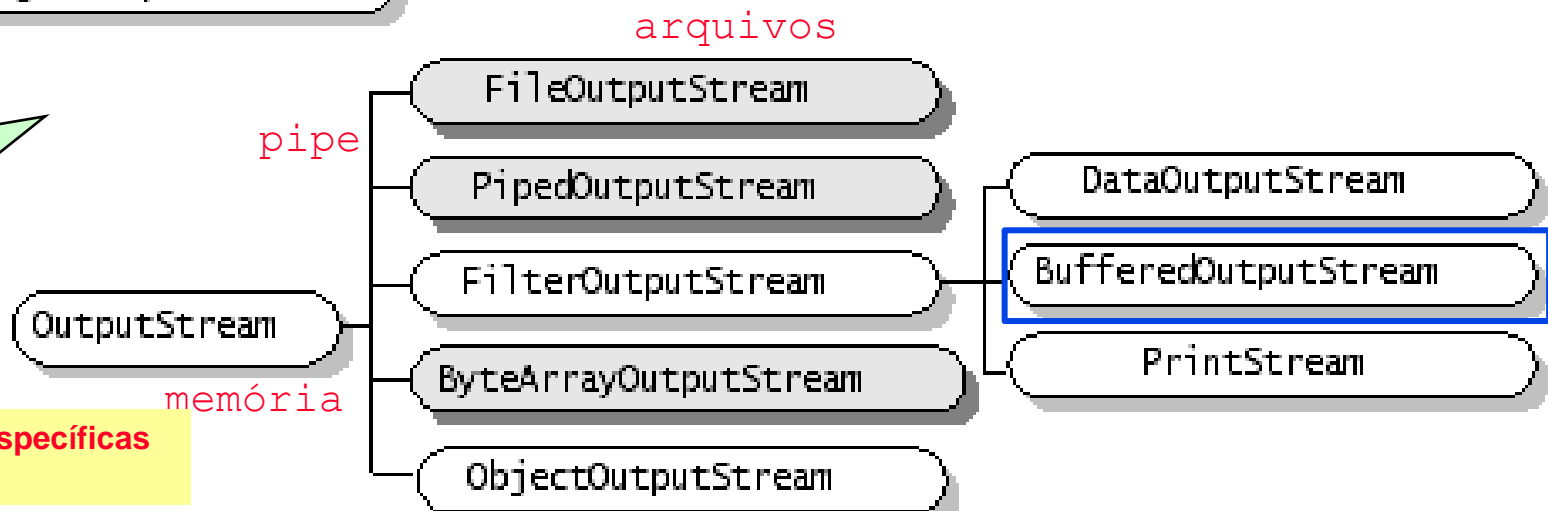


Hierarquia: InputStream, OutputStream

- Principais implementações



As classes iniciais são abstratas, logo é necessário definir as subclasses que deverão ser instanciadas, montando essa hierarquia.

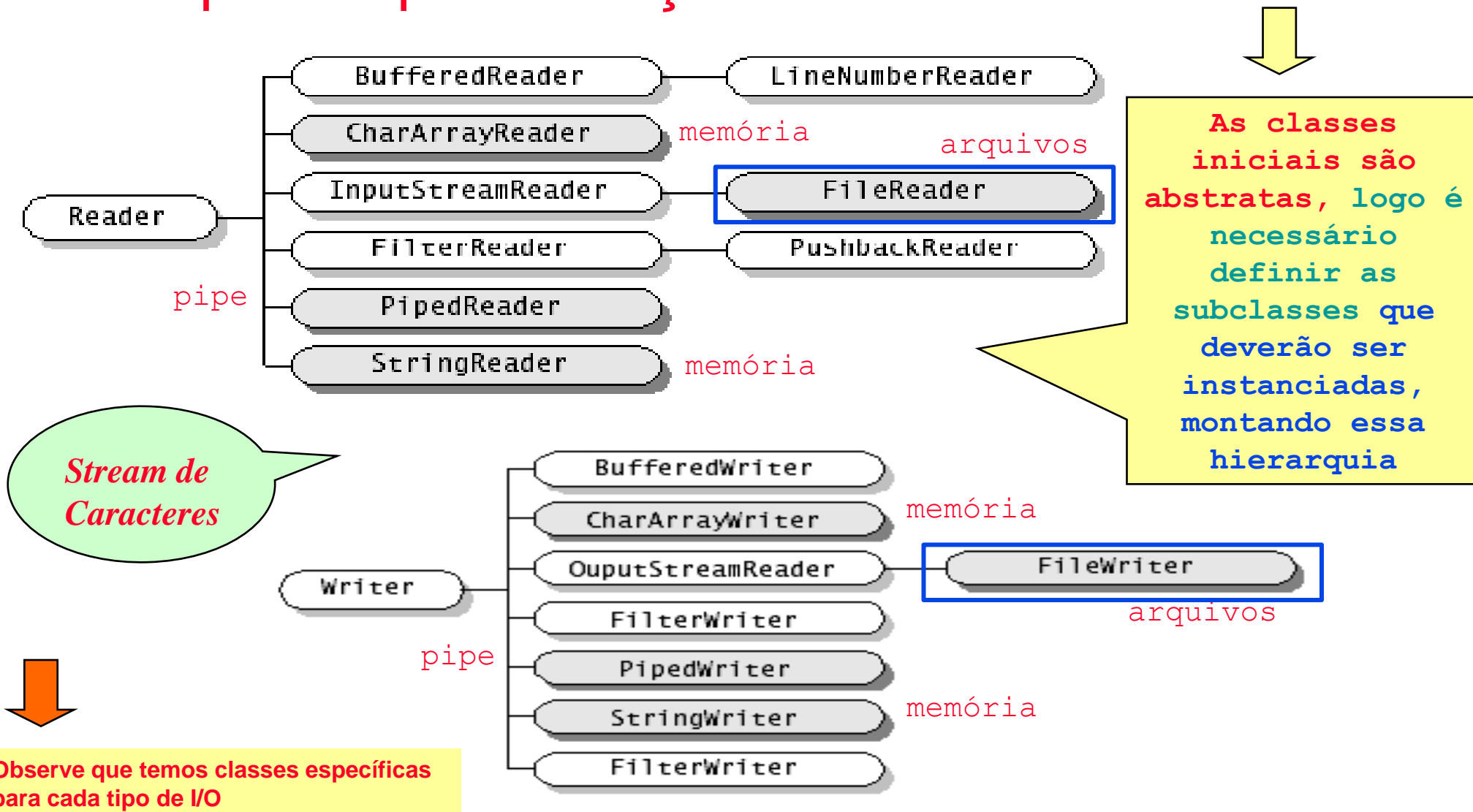


Stream de Bytes

Observe que temos classes específicas para cada tipo de I/O

Hierarquia: Reader, Writer

- Principais implementações



I/O em Java

- Resumindo: Qual *streams* utilizar???

- Temos que definir:

- ⇒ O tipo de informação

- ⇒ E trabalhar com o *stream* correto

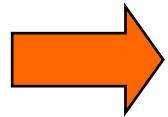
- A classe utilizada depende da necessidade de trabalhar diretamente.... Por exemplo..

- ⇒ Com *bytes* (inteiros, real, binários, etc)

- ⇒ Ou caracteres (texto)

I/O em Java

- Bom um outro conceito importante e muito utilizado em stream....
-é o de Filtro e Buffer..
- Vamos analisar esses conceitos...



I/O em Java

- Vamos analisar o conceito de *filtro*.

→ FILTRO:

- Um filtro é um tipo de *stream* que modifica o modo como um *stream* existente é tratado.

→ Pense em uma represa em um riacho na montanha. A represa regula o fluxo de água dos pontos acima para os pontos abaixo.

- A represa é um tipo de filtro – remova-o e a água fluirá de uma forma muito menos controlado.

- Implementam o padrão de projeto Decorator

→ Adição incremental de funcionalidades

→ Os filtros são concatenados em *streams* primitivos oferecendo métodos mais úteis com dados filtrados

– Por exemplo...

- Classe `FilterInputStream`:
 - Recebe fonte de bytes e oferece métodos para ler dados filtrados.
- Classe `FilterOutputStream`:
 - Recebe destino de bytes e escreve dados via filtro.

I/O em Java

- Vamos analisar o conceito de *buffer*

➡ BUFFER:

- Um *buffer* é um local de armazenamento no qual os dados podem ser mantidos antes que sejam necessários por um programa que lê ou grava esses dados.

➡ Usando *buffer*, pode-se obter dados sem estar sempre retornando à fonte original dos dados.

- Onde queremos chegar com Buffers e Filtros...

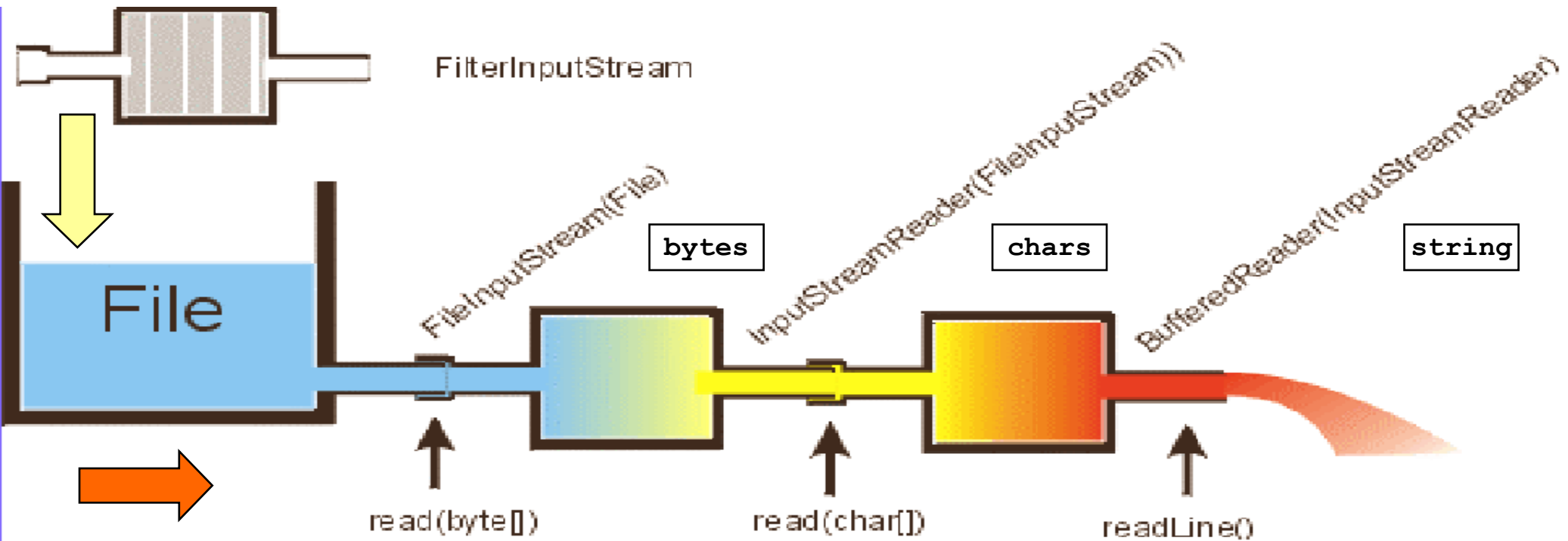
➡ **Por exemplo....** O conceito de *filtro* é utilizado pelo *stream* de entrada e saída de *bytes* para construir um *buffer*:

➡ Um *buffer* é um *stream* com *filtro* armazenando as informações que são enviadas ou lidas no *stream* de *bytes*

– Ex.: Classes *BufferedInputStream* e *BufferedOutputStream*

I/O em Java

- Buffer e Filtro
 - Exemplo detalhado de uso de filtros e buffer para ler uma linha de um arquivo



```
// objeto do tipo File
File tanque = new File("agua.txt");

// referência FileInputStream
// cano conectado no tanque
FileInputStream cano =
    new FileInputStream(tanque);
```

```
// filtro chf conectado no cano
InputStreamReader chf = new InputStreamReader(cano);
// filtro br conectado no chf
BufferedReader br = new BufferedReader(chf);
// lê linha de texto a de br
String linha = br.readLine();
```

I/O em Java

- *Stream* de entrada e saída do console

⇒ Blz, agora que entendemos as estruturas de *stream* de I/O em Java

- Inicialmente podemos analisar na prática com o console

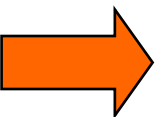
- Trabalhar inicialmente com interação com teclado....

⇒ No princípio, essa não é era tarefa muito simples em Java....

⇒ Java oferece a classe *System*, para representar os dispositivos de entrada e saída

- Essa classe faz parte do pacote `java.lang`,

⇒ É utilizada para representar e fornece facilidades como.....:



I/O em Java

➡ Entrada Padrão (exemplo: teclado)

– `System.in` -> *public static final InputStream in*

- A variável de classe *in* é um objeto *InputStream*, que recebe entrada de teclado ou outra fonte de dados especificado pelo usuário (como um arquivo, ou socket) através do *stream*
- Este *stream* já está aberto e preparado para fornecer entrada de dados
- Ex.: `System.in.read(byte[] b);`

➡ Saída Padrão - (e.g. vídeo)

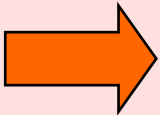
– `System.out` -> *public static final PrintStream out*

- A classe *PrintStream* herda de *OutputStream*
- Ex.: `System.out.println()`

➡ Saída Erros Padrão - (e.g. vídeo)

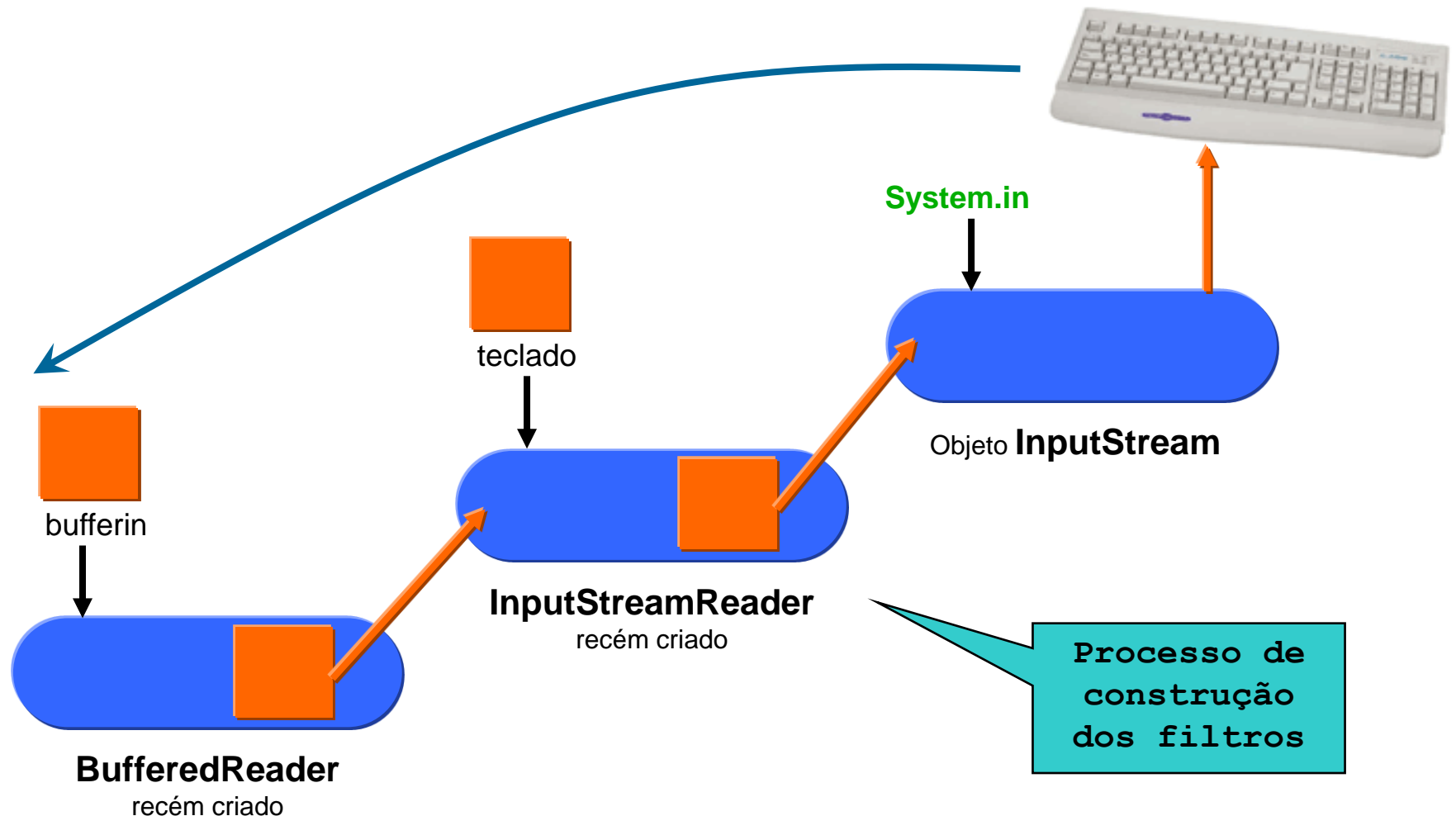
– `System.err` -> *public static final PrintStream err*

- A classe *PrintStream* herda de *OutputStream*
- Ex.: `System.err.println()`



I/O em Java

- Exemplo de Leitura do teclado utilizando *Buffer*
 - Vamos usando o *Reader*....ou seja, estamos trabalhando com caracteres



• Exemplo de Leitura do teclado utilizando *Buffer*

– Estamos usando o *Reader*

I/O em Java

```
1 import java.io.*;
2
3 public class LeituraTecladoBufferedReader {
4     public static void main(String[] arguments) {
5
6         // Primeira forma de declaração
7         System.out.print("Digite alguma coisa e tecle enter: ");
8         BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
9         String palavra;
10
11         try{
12             palavra = br.readLine();
13             System.out.println("Voce digitou: " + palavra);
14         }
15         catch(IOException e) {
16             System.out.println("Erro durante a leitura" + e);
17         }
18
19         // Segunda forma de declaração
20         String buffer;
21         InputStreamReader teclado = new InputStreamReader(System.in);
22         BufferedReader bufferin = new BufferedReader(teclado);
23
24         while (true) {
25             try{
26                 System.out.print("Digite o texto e tecle enter (exit para sair): ");
27                 buffer = bufferin.readLine();
28
29                 if( buffer.equals("exit") ) {
30                     break;
31                 }
32                 System.out.println("echo: " + buffer); // Mostra na tela
33             }
34             catch(IOException e) {
35                 System.out.println("Erro durante a leitura do echo" + e);
36             }
37         }
38     }
39 }
40
41 }
```

O que o programa está fazendo?

I/O em Java

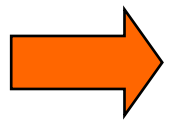
⇒ Blz....

- Agora que entendemos “essas coisas”...
 -podemos conversar sobre **manipulação de arquivos..**
 - E sobre ***stream* de entrada e saída de arquivos.....**

⇒ Esses *streams* são usados para troca de dados com arquivos

- Nas unidades de disco, CD-ROM, fita, pen-driver, etc....

⇒ Mas como isso é possível....???



I/O em Java

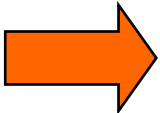
- **Simplesmente** utilizando as classes:

⇒ Para acesso via *streams*

- FileInputStream / FileOutputStream (*bytes*)
- FileReader / FileWriter (*caracteres*)

⇒ Para manipulação de arquivos

- File



I/O em Java

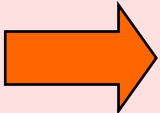
➡ Trabalhando com *stream* de *bytes* para entrada e saída bufferizados de arquivos.

- O *buffer* é utilizado para otimizar o acesso as informações
 - Usando *buffer*, o programa primeiro consulta o *buffer* antes de ir à fonte original do *stream*.

- ➡ Para isso precisamos usar o:
- *FileOutputStream* e *FileInputStream* em conjunto com
 - *BufferedInputStream* e *BufferedOutputStream*

- ➡ O nome do arquivo é usado para criar um *stream* de entrada e saída de arquivo.
- *FileInputStream* *file* = new *FileInputStream*("numeros.dat");
 - *FileOutputStream* *file* = new *FileOutputStream*("numeros.dat");

- ➡ O *stream* de arquivo criado é usado para criar um *stream* de *buffer* para a entrada e saída.
- *BufferedInputStream* *buff* = new *BufferedInputStream*(*file*);
 - *BufferedOutputStream* *buff* = new *BufferedOutputStream*(*file*);



I/O em Java

➡ Trabalhando com **stream de caracteres para entrada e saída bufferizados** de arquivos.

- O *buffer* é utilizado para otimizar o acesso as informações
 - Usando *buffer*, o programa primeiro consulta o *buffer* antes de ir à fonte original do *stream*.

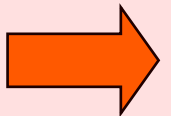
- ➡ Estamos usando o:
- *FileReader* e *FileWriter* em conjunto com
 - *BufferedReader* e *BufferedWriter*

➡ O nome do arquivo é usado para criar um *stream* de entrada e saída de arquivo.

- `FileReader fr = new FileReader("readme.txt");`
- `FileWriter fw = new FileWriter("readmeCaps.txt");`

➡ O *stream* de arquivo criado é usado para criar um *stream de buffer* para a entrada e saída.

- `BufferedReader in = new BufferedReader(fr);`
- `BufferedWriter out = new BufferedWriter(fw);`



I/O em Java

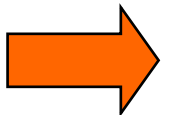
➡ Vamos analisar um exemplo de **stream de caracteres para entrada e saída** *bufferizados* de arquivos.

- Estamos usando o *FileReader* em conjunto com *BufferedReader*
 - O nome do arquivo é usado para criar um *stream* de entrada.
- O *stream* de arquivo criado é usado para criar um *stream* de *buffer* para a entrada.

➡
– `FileReader fr = new FileReader("LeFonte.java");`
– `BufferedReader in = new BufferedReader(fr);`



- Após montar essa estrutura eu posso usar o método `readLine()` de *buffer*.
 - Este método é usado para ler no arquivo texto **uma linha por vez**.
- O loop termina quando o método retorna o valor null
 - `String line = in.readLine();`



I/O em Java

- Vamos analisar um exemplo de **stream de caracteres para entrada e saída bufferizados** de arquivos.
 - Estamos usando o *FileReader* em conjunto com *BufferedReader*

```
1  import java.io.*;
2
3  public class LeFonte {
4      public static void main(String[] arguments) {
5          try {
6              FileReader file = new FileReader("LeFonte.java");
7              BufferedReader buff = new BufferedReader(file);
8              boolean eof = false;
9              while (!eof) {
10                 // Este método é usado para ler o arquivo texto uma linha por vez.
11                 // O loop termina quando o método retorna o valor null
12                 String line = buff.readLine();
13                 if (line == null)
14                     eof = true;
15                 else
16                     System.out.println(line);
17             }
18             buff.close();
19         } catch (IOException e) {
20             System.out.println("Error -- " + e.toString());
21         }
22     }
23 }
```

O que o programa está fazendo?

I/O em Java

- Exemplo de *stream* de caracteres sem buffer e uso das classes Scanner

- new Scanner(FileReader file)
- Importante observar que o conector final é o scanner
- É ele que fornece os métodos de interação

```
13 import java.io.FileReader;
14 import java.io.FileWriter;
15 import java.io.IOException;
16 import java.util.Scanner;
17
18 public class ArquivoScanner {
19     public static void main(String args[]) throws IOException {
20
21         int i;
22         double d;
23         boolean b;
24         String str;
25
26         FileWriter fout = new FileWriter("test.txt");
27         fout.write("Testing Scanner 10 12.2 one true two false");
28         fout.close();
29
30         FileReader fin = new FileReader("Test.txt");
31
32         Scanner src = new Scanner(fin);
33
34         while (src.hasNext()) {
35             if (src.hasNextInt()) {
36                 i = src.nextInt();
37                 System.out.println("int: " + i);
38             }
39             else if (src.hasNextDouble()) {
40                 d = src.nextDouble();
41                 System.out.println("double: " + d);
42             }
43             else if (src.hasNextBoolean()) {
44                 b = src.nextBoolean();
45                 System.out.println("boolean: " + b);
46             }
47             else {
48                 str = src.next();
49                 System.out.println("String: " + str);
50             }
51         }
52
53         fin.close();
54     }
55 }
```

O que o programa
está fazendo?

I/O em Java

- Exemplo de *stream* de caracteres sem buffer e uso das classes Scanner

- new Scanner(Readable source)
- Importante observar que o conector final é o scanner
- É ele que fornece os métodos de interação

O que o programa está fazendo?

```
1  /*
2  */
3  import java.io.FileReader;
4  import java.io.FileWriter;
5  import java.io.IOException;
6  import java.util.Scanner;
7
8  public class MediaScannerArquivo {
9      public static void main(String args[]) throws IOException {
10         int count = 0;
11         double sum = 0.0;
12
13         FileWriter fout = new FileWriter("test1.txt");
14         //fout.write("2, 3.4, 5,6, 7.4, 9.1, 10.5, done");
15         fout.write("2- 3,4- 5-6- 7,4- 9,1- 10,5- done");
16
17         fout.close();
18
19         FileReader fin = new FileReader("Test1.txt");
20
21         Scanner src = new Scanner(fin);
22
23         //src.useDelimiter(", *");
24         src.useDelimiter("- *"); // 0 * ignora os espaços
25         //src.useDelimiter("-");
26
27         while(src.hasNext()) {
28             if(src.hasNextDouble()) {
29                 sum += src.nextDouble();
30                 count++;
31             }
32             else {
33                 String str = src.next();
34                 if(str.equals("done")) break;
35                 else {
36                     System.out.println("File format error.");
37                     return;
38                 }
39             }
40         }
41
42         fin.close();
43         System.out.println("Media eh: " + sum / count);
44     }
45 }
46
```

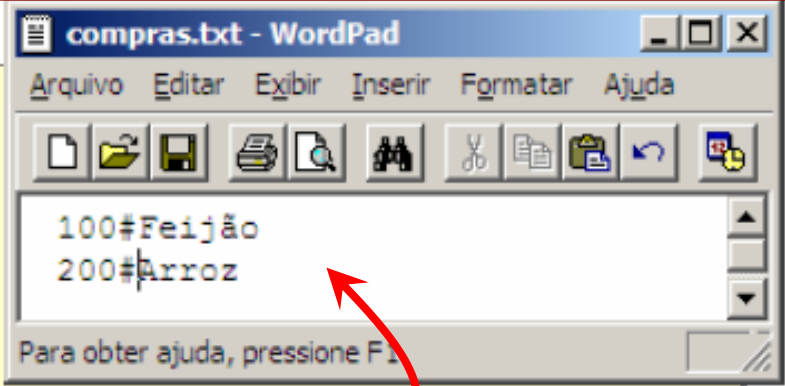
I/O em Java

➡ **Resumo importante!!!**

➡ **Lembre-se:** o controle sobre toda a formatação do arquivo (separadores, quebras de linha, etc) é tarefa do programador do sistema

```
import java.io.BufferedWriter;
import java.io.FileWriter;
import java.io.IOException;

public class GravaTexto {
    public static void main( String[] args ) {
        try {
            FileWriter fw = new FileWriter("compras.txt");
            BufferedWriter bw = new BufferedWriter(fw);
            bw.write("100"); bw.write("#"); bw.write("Feijão"); bw.write("\n");
            bw.write("200"); bw.write("#"); bw.write("Arroz"); bw.write("\n");
            bw.close();
        } catch ( IOException e ) { e.printStackTrace(); }
    }
}
```



Para obter ajuda, pressione F1

Escrita formatada do texto

I/O em Java

→ Resumo importante!!!

→ Lembre-se: o controle sobre toda a formatação do arquivo (separadores, quebras de linha, etc) é tarefa do programador do sistema

- Para recuperar o texto formatado pode-se usar o método `split()` ou a classe `StringTokenizer`.

```
import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;
public class LeTexto {
    public static void main( String[] args ) {
        try {
            FileReader fr = new FileReader("compras.txt");
            BufferedReader br = new BufferedReader(fr);
            String linha = null;
            while ( (linha = br.readLine()) != null ) {
                String[] v = linha.split("#");
                for ( String dado : v ) System.out.println(dado);
            }
            br.close();
        } catch ( IOException e ) { e.printStackTrace(); }
    }
}
```

Leitura formatada do texto

I/O em Java

- Classe *File*

→ Em todos exemplos até aqui, uma String foi utilizada para se referir ao arquivo envolvido em uma operação de *stream* de arquivo

– Isso normalmente é o bastante, mas....

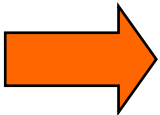
→e se houver necessidade de copiar para diretórios diferentes, renomear ou tratar de outras tarefas,

- E se tivermos Plataformas Diferentes

- Ex.: Diferentes notações para trilhas

- windows: c:/java/aplic

- unix: \usr\home\java\aplic



– Por isso é necessário utilizar um objeto *File*.!!!

I/O em Java

- Classe *File*: Características importantes!!!

⇒ Um objeto *File* representa

- Uma referência de arquivo ou pasta

⇒ É apenas uma abstração:

- **Importante:** A existência de um objeto *File* não significa a existência de um arquivo ou diretório

⇒ Contém métodos para:

- Testar a existência de arquivos, para definir permissões (nos S.O.s onde for aplicável), para apagar arquivos, criar diretórios, listar o conteúdo de diretórios, etc.

⇒ Notação multiplataforma

- Prefixo, seqüência de strings, *File.separator*

I/O em Java

- Alguns métodos da classe *File*

- `java.io.File`

- `String getAbsolutePath()`
- `String getParent()`: retorna o diretório (objeto *File*) pai
- `boolean exists()`
- `boolean isFile()`
- `boolean isDirectory()`
- `boolean delete()`: tenta apagar o diretório ou arquivo
- `long length()`: retorna o tamanho do arquivo em bytes
- `boolean mkdir()`: cria um diretório com o nome do arquivo
- `String[] list()`: retorna lista de arquivos contido no diretório

I/O em Java

- Trabalhando com **stream de caracteres** para entrada e **saída** de arquivos e utilização da classe *File*

```
1  import java.io.*;
2
3  public class CopiaUsandoFile {
4      public static void main(String[] args) throws IOException {
5
6          File inputFile = new File("origem.txt");
7          File outputFile = new File("destinoCopiaUsandoFile.txt");
8
9          FileReader in = new FileReader(inputFile);
10         FileWriter out = new FileWriter(outputFile);
11
12         if (outputFile.exists())
13             outputFile.delete();
14
15         int c;
16         while((c = in.read()) != -1)
17             out.write(c);
18
19         in.close();
20         out.close();
21     }
22 }
```

O que o programa
está fazendo?

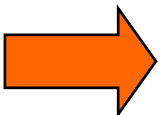
I/O em Java

- Blz, quase fechando...

- Agora que entendemos “essas coisas”... podemos conversar sobre um outro tipo de *stream*...

- Agora vamos conversar sobre *stream* de entrada e saída de dados.....

- Mas que raios é isso.....???



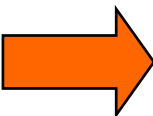
I/O em Java

- *Streams* de dados

→ Se precisar trabalhar com dados que não sejam representados como *bytes* ou caracteres.... Podemos usar os *streams* de entrada e saída de dados.

→ Este *streams* permitem escrita e leitura de tipos primitivos diretamente, tais como... *char*, *float*, *integer*, *double*, etc

- Estes *streams* filtram um *stream* de *bytes* existente
 - De modo que os tipos primitivos possam ser lidos ou escritos diretamente do *stream*
- Para trabalhar com esse *stream* temos que trabalhar com...



I/O em Java

- As classes definidas pelas interfaces...

- ***DataInput*** e ***DataOutput***

- Estas interface são Implementados por:

➡ ***DataInputStream***

- Um *stream* de entrada de dados é criado com o construtor

DataStream(InputStream)

- O argumento deve ser um *stream* de entrada existente, como um *stream* de entrada de buffer ou de arquivo


➡ ***DataOutputStream***


- Um *stream* de saída de dados é criado com o construtor

DataOutputStream(OutputStream),

- Que indica o *stream* de saída associado.

I/O em Java

- 
- Cada método de entrada retorna o tipo de dado primitivo indicado pelo nome do método.
 - O mesmo é válido para os métodos de saída.

 É possível determinar o tipo de informação que está sendo enviado dentro do stream com **métodos específicos** para os tipos de dados

– java.io.DataInput

```
public abstract boolean readBoolean() throws IOException
public abstract byte readByte() throws IOException
public abstract int readUnsignedByte() throws IOException
public abstract char readChar() throws IOException
public abstract void readFully(byte b[]) throws IOException
public abstract int skipBytes(int n) throws IOException
```

– java.io.DataOutput

```
public abstract void write(byte b[]) throws IOException
public abstract void writeBoolean(boolean v) throws IOException
public abstract void writeByte(int v) throws IOException
public abstract void writeChar(int v) throws IOException
public abstract void writeInt(int v) throws IOException
```

I/O em Java

➡ Trabalhando com *streams* de dados para saída de arquivos

- Um *stream* de dados para saída de arquivos é construído utilizando as classes *FileOutputStream* e *DataOutputStream*
 - O construtor recebe um argumento *String* que deverá ser o nome do arquivo onde os dados serão armazenados:
 - Ex.: Abrir um arquivo para escrita com *stream* de dados

➡

```
DataOutputStream out = new FileOutputStream(new  
FileOutputStream("Out.txt"));
```

- O significado do comando acima é: ↓
 - Criar um *stream* de dados de saída de arquivo a partir do arquivo "Out.txt"

➡ Após o *stream* ter sido criado, é possível escrever tipos primitivos no *stream*, chamando o método *write* do tipo específico.

- Ex.:

```
out.writeDouble(prices[i]); out.writeChar('\t'); out.writeInt(units[i]);
```

I/O em Java

- Exemplo de *streams de dados para saída de arquivos*
 - Estamos usando o *FileOutputStream* em conjunto com *DataOutputStream*

```
1 import java.io.*;
2
3 public class EscreverDados {
4
5     public static void main(String args[]) throws IOException {
6
7         // static String  getProperty(String key):
8         // Gets the system property indicated by the specified key.
9         char lineSep = System.getProperty("line.separator").charAt(0);
10
11         // Cria arquivo pedDados.txt
12         DataOutputStream out = new DataOutputStream(new FileOutputStream("pedDados.txt"));
13         double[] prices = { 19.99, 9.99, 15.99};
14         int[] units = { 12, 8, 13 };
15         String[] descs = { "Camisa", "Sapato", "Chaveiro" };
16
17         for (int i = 0; i < prices.length; i++) {
18             out.writeDouble(prices[i]);
19             out.writeChar('\t');           // tabulação (tab)
20
21             out.writeInt(units[i]);
22             out.writeChar('\t');           // tabulação (tab)
23
24             out.writeChars(descs[i]);
25             out.writeChar(lineSep);
26         }
27         out.close();
28     }
29 }
30
31
```

Observe a ordem / forma
que os dados são escritos no arquivo
Ela é relevante em algum momento???

I/O em Java

→ Trabalhando com *streams* de dados para entrada de arquivos

- Um *stream* de dados para entrada de arquivos é construído utilizando as classes *FileInputStream* e *DataInputStream*
 - O construtor recebe um argumento *String* que deverá ser o nome do arquivo onde os dados estão armazenados:
 - Ex.: Abrir um arquivo para leitura com *stream* de dados

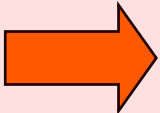
→

```
DataInputStream in = new DataInputStream(new  
FileInputStream("In.txt"));
```

- O significado do comando acima é: ↓
- Criar um *stream* de dados de entrada de arquivo a partir do arquivo "In.txt"

→ Após o *stream* ter sido criado, é possível ler tipos primitivos no *stream*, chamando o método *read* do tipo específico.

- Ex.: `in.readDouble(); in.readChar(); in.readInt();`



I/O em Java

- *Streams* de dados para entrada de arquivos
 - É importante observar que.....os diferentes métodos de leitura de um *stream* de entrada de dados.....

→ Não retornam um valor que possa ser usado como indicador de que o final do *stream* foi alcançado.

→ Como alternativa, podemos esperar que uma *EOFException* (exceção de fim de arquivo) seja gerada quando um método de leitura atingir o fim do *stream*.

I/O em Java

- Exemplo de *streams* de dados para entrada de arquivos
 - Estamos usando o *FileInputStream* em conjunto com *DataInputStream*

```
1 import java.io.*;
2
3 public class LeDados {
4
5     public static void main(String args[]) throws IOException {
6
7         // static String  getProperty(String key):
8         // Gets the system property indicated by the specified key.
9         char lineSep = System.getProperty("line.separator").charAt(0);
10
11         // Le o arquivo pedDados.txt
12         DataInputStream in = new DataInputStream(new FileInputStream("pedDados.txt"));
13         double price;
14         int unit;
15         StringBuffer desc = new StringBuffer();
16         double total = 0.0;
17         char chr;
18
19         try {
20             while (true) {
21                 // Observar que estou recuperando os dados na mesma ordem que
22                 // escrevi no arquivo
23                 price = in.readDouble();
24                 in.readChar(); // pula tabulação
25
26                 unit = in.readInt();
27                 in.readChar(); // pula tabulação
28
29                 // Lê caracter por caracter até encontrar o separador
30                 while ((chr = in.readChar()) != lineSep)
31                     desc.append(chr);
32
33                 System.out.println("Pedido " + unit + " unid. de " + desc + " a R$" + price);
34                 total = total + unit * price;
35             }
36         }
37         catch (EOFException e) { }
38
39         System.out.println("Valor total das mercadorias: R$ " + total);
40         in.close();
41     }
42 }
```

Pra casa: Como alterar o código para limpar o StringBuffer?

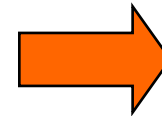
A ordem é importante...

I/O em Java

- Blz, pra fechar!!..

– Um *stream* é um objeto que transporta dados de um lugar para outro de uma origem para o programa Java ou do programa Java para o destino

– Temos vários tipos de streams, como.....



I/O em Java

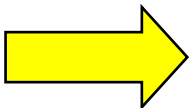
- Relembrando os *Streams* que temos em Java:
 - *Streams de bytes*
 - São usados para lidar com bytes, inteiros e outros tipos de dados simples
 - *Streams de caracteres*
 - Tratam de arquivos textos e outras fontes de textos
 - *Streams de dados*
 - Permitem escrita e leitura de tipos primitivos diretamente (char, float, integer, double, etc)



Streams de objetos

- Permite que os dados sejam representados como parte de um objeto
 - Tratam da persistência e recuperação de objetos como um todo para que um objeto seja salvo em um destino, como um arquivo de disco, por exemplo.

Como
assim??



I/O em Java

- Serializando Objetos – *Stream* de Objetos

⇒ Conceito Importante: O conceito por trás dessa *stream* é a persistência

Persistência: é a capacidade de um objeto existir e funcionar fora do programa que o criou

⇒ Mas como isso fica em OO. Vamos analisar isso em OO...

- Um objeto é complexo e armazena várias informações.....
- Quando “desligamos” o nosso sistema OO, os objetos existentes na memória são apagados....
- Como podemos guardar e restaurar estes objetos com suas informações futuramente em um programa...

I/O em Java

- Serializando Objetos – Qual é a idéia...

➡ Para que um objeto seja salvo em um destino....

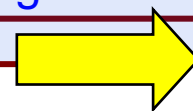
- Como um arquivo de disco, por exemplo

➡ele precisa ser convertido para a forma serial proprietária ou customizada...

- Dados seriais são enviados um de cada vez, como um fileira de carros em uma linha de montagem.

➡ Um objeto serializado é um grafo que inclui dados da classe e todas as suas dependências que podem ser persistidas

- Se a classe ou suas dependências mudar, o formato usado na serialização mudará e os novos objetos serão incompatíveis com os antigos
 - Não será mais possível recuperar arquivos gravados com a versão antiga

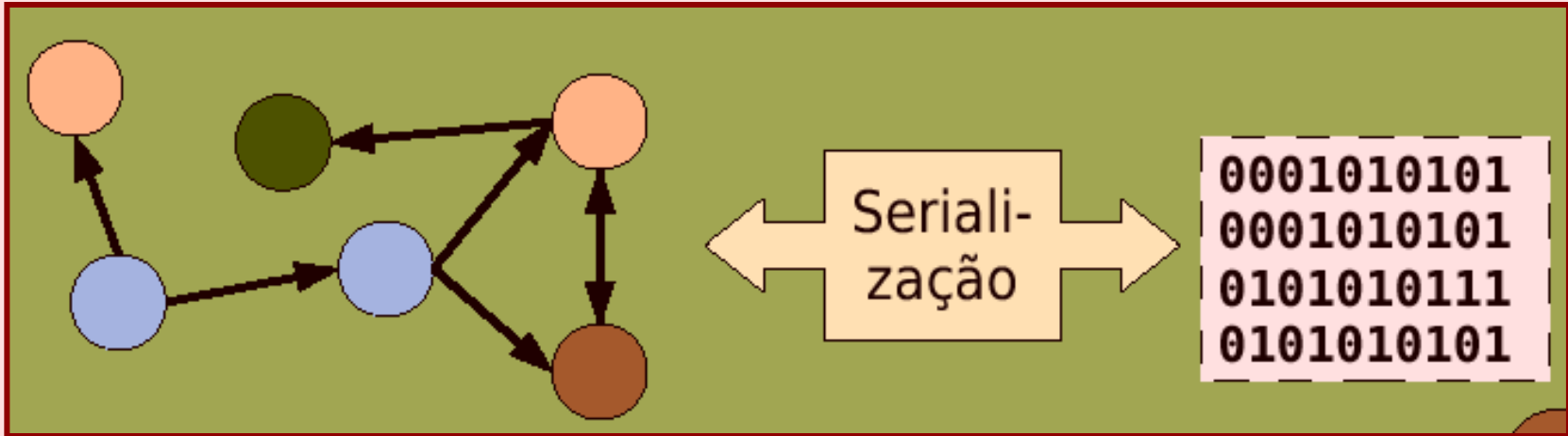


I/O em Java

- Serializando Objetos – Qual é a idéia...

➡ Quando um objeto é salvo em um *stream* de forma serial,

- Todos os objetos que podem ser persistido aos quais ele contém referencia também são salvos



➡ Um objeto que não é serializado não é persistente...

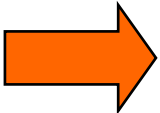
-ou seja, deixa de existir após a execução do programa

I/O em Java

- Blz,
 - Agora que entendemos “essas coisas” temos que conversar sobre 2 assuntos importantes..

⇒ 1. Como podemos aplicar isso na prática, ou seja, declarar que uma classe pode ser serializada..... ????

⇒ 2. E quais são as classes que possibilitam o uso da persistência de objeto...



I/O em Java

- 1. Serializando Objetos – Declarando que pode persistir...

➡ Um objeto indica que pode ser serializado....

-Por meio da implementação da interface **Serializable**

➡ Difere das outras interfaces...

-Pois não contém métodos que precisam ser incluídos nas classes que o implementam

➡ A única finalidade de interface **Serializable** é....

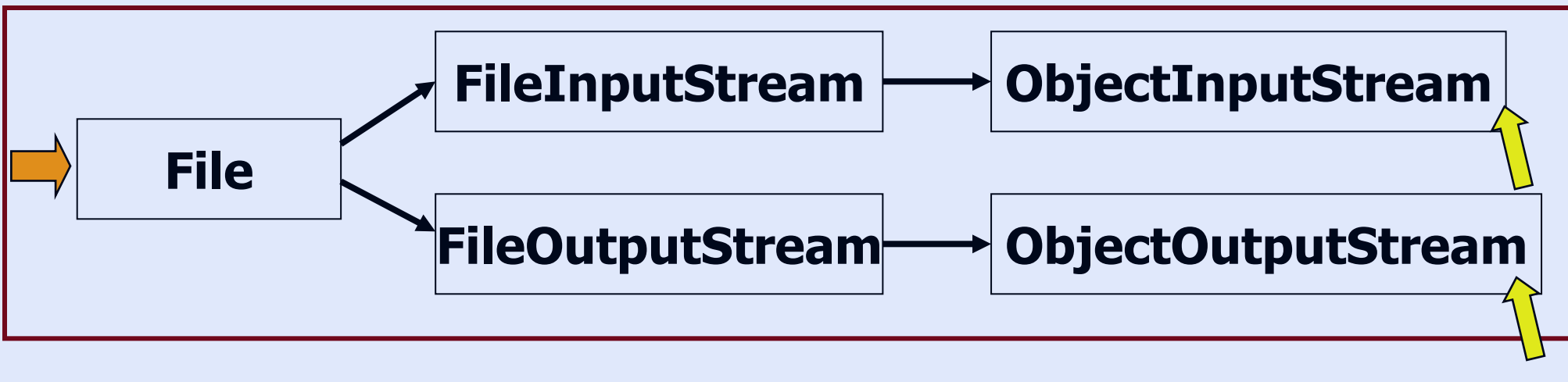
-Indicar que os objetos da classe podem ser armazenados e recuperados de forma serial

```
public class Info implements Serializable {
```

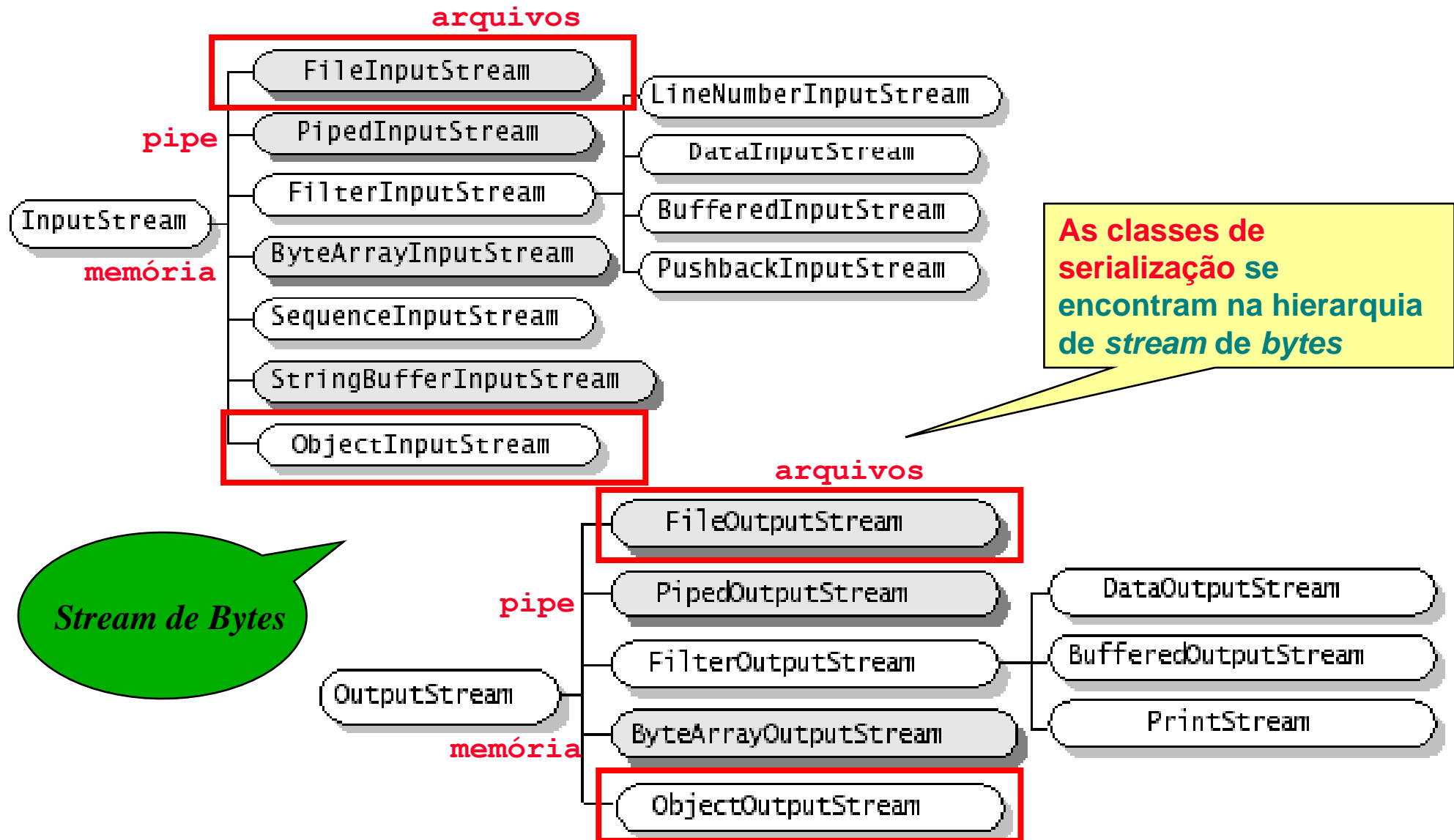
I/O em Java - Arquivos e fluxos

- 2. Serializando Objetos – *Streams* Entrada/Saída de objetos

➡ Para trabalhar com entrada e saída serializadas de informações em arquivo...é necessário utilizar as classes envolvidas para ler e escrever objetos que são:



Hierarquia: InputStream, OutputStream



I/O em Java - Arquivos e fluxos

- Trabalhando com **Serialização de Objetos** – Saída de objetos

➔ Um objeto é escrito em um *stream* por meio da classe **ObjectOutputStream**

- Para criar um *stream* de saída para arquivo e um *stream* de saída de objeto associado devemos fazer:

➔ **FileOutputStream disco = new FileOutputStream ("ObjetoSalvo.dat");**
ObjectOutputStream obj = new ObjectOutputStream(disco);

- O *stream* de saída de objeto criado nesse exemplo é o *obj*.
- Os métodos da classe *obj* podem ser usados para gravar objetos serializáveis e outras informações em um arquivo chamado *ObjetoSalvo.dat*

➔ Pode-se escrever um objeto utilizando o método **writeObject(Objeto)**

➔ **obj.writeObject(DadosUsuario)**

– Onde *DadoUsuario* precisa ser declarada como serializável.

- Objeto deve implementar a **interface java.io.Serializable**

I/O em Java - Arquivos e fluxos

- Serializando Objetos – Saída de objetos

- ObjectOutputStream

➡ Importante observar a assinatura do método writeObject:

```
public final void writeObject(Object obj)  
                        throws IOException
```

Anúncio de exceção,
Uso do try\catch no
código

I/O em Java - Arquivos e fluxos

- Serializando Objetos – Entrada de objetos

➡ Um objeto é lido de um *stream* usando a classe ***ObjectInputStream***

- Para criar um *stream* de entrada de arquivo e um *stream* de entrada de objeto associado devemos fazer:

➡ ***FileInputStream disco = new FileInputStream ("ObjetoSalvo.dat");***
ObjectInputStream obj = new ObjectInputStream(disco);

- O *stream* de entrada de objeto criado nesse exemplo é o *obj*.
- Essa *stream* de entrada de objeto é configurada para ler de um objeto que está armazenado em um arquivo chamado *ObjetoSalvo.dat*

➡ Um objeto pode ser lido do arquivo por meio do método ***readObject()***

➡ ***ClasseSerializavel nomeClasse = (ClasseSerializavel) obj.readObject()***

- Importante observar a realização do ***typecast*** do objeto recuperado
 - Isto torna possível reconhecer a estrutura do objeto que está sendo lido.

I/O em Java - Arquivos e fluxos

- Serializando Objetos – Entrada de objetos

- **ObjectInputStream**

➡ Importante observar a assinatura do método readObject:

```
public final Object readObject()  
                        throws ClassNotFoundException,  
                        IOException
```

Anúncio de exceção,
Uso do try\catch no
código

I/O em Java - Arquivos e fluxos

- Serializando Objetos – **Resumo:**

➡ Objetos podem ser persistidos...

- Usado a Serialização de objetos.

➡ Objetos podem ser serializados:

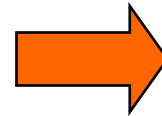
- Para o disco em uma única máquina ou
- Podem ser serializados pela rede, mesmo em caso de SO diferentes.
 - Java trabalha de modo transparente com diferentes formatos para salvar dados nesses sistemas quando os objetos são serializados
 - Ex: /n no Windos e \n no Linux

➡ O Mecanismo de serialização é responsável por:

- Converter para bytes e vice-versa;
- Fazer e desfazer a “busca” pelas referências do objeto;
- Compensar diferenças entre sistemas operacionais;
- Usar ObjectInputStream e ObjectOutputStream.

I/O em Java - Arquivos e fluxos

- Blz,
 - Vamos analisar alguns exemplos...



I/O em Java - Arquivos e fluxos

- Serializando o objeto - Modelagem / Definição

```
32 class Message implements Serializable {  
33     int lineCount;  
34     String from, to;  
35     Date when;  
36     String[] text;  
37  
38     void writeMessage(String inFrom, String inTo, Date inWhen, String[] inText) {  
39  
40         text = new String[inText.length];  
41         for (int i = 0; i < inText.length; i++)  
42             text[i] = inText[i];  
43         lineCount = inText.length;  
44         to = inTo;  
45         from = inFrom;  
46         when = inWhen;  
47     }  
48 }
```

O que o programa
está modelando?

I/O em Java - Arquivos e fluxos

- Serializando o objeto – Saída

O que o programa
está fazendo?

```
1 import java.io.*;
2 import java.util.*;
3
4 public class ObjectToDisk {
5
6     public static void main(String[] arguments) {
7
8         Message mess = new Message();
9
10        String author = "Vinicius Rosalen, Brasil";
11        String recipient = "Alunos da UVV , UVV - Boa Vista";
12        String[] letter = { "Exercicio de serializao de objetos.",
13                            "Depois daqui vamos serializar objetos para o todo mundo",
14                            "dessa forma vamos enviar objetos pelo mundo." };
15        Date now = new Date();
16
17        mess.writeMessage(author, recipient, now, letter);
18
19        try {
20            FileOutputStream fo = new FileOutputStream("Message.obj");
21            ObjectOutputStream oo = new ObjectOutputStream(fo);
22            oo.writeObject(mess);
23            oo.close();
24            System.out.println("Objeto criado com sucesso.");
25        }
26        catch (IOException e) {
27            System.out.println("Error -- " + e.toString());
28        }
29    }
30 }
```

I/O em Java - Arquivos e fluxos

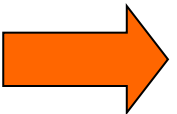
- Recuperando o objeto serializado – Entrada

O que o programa está fazendo?

```
1 import java.io.*;
2 import java.util.*;
3
4 public class ObjectFromDisk {
5
6     public static void main(String[] arguments) {
7         try {
8             ➔ FileInputStream fi = new FileInputStream("message.obj");
9             ➔ ObjectInputStream oi = new ObjectInputStream(fi);
10
11             ➔ Message mess = (Message) oi.readObject();
12             System.out.println("Mensagem:\n");
13             System.out.println("Remetente: " + mess.from);
14             System.out.println("Para: " + mess.to);
15             System.out.println("Data: " + mess.when + "\n");
16
17             for (int i = 0; i < mess.lineCount; i++)
18                 System.out.println(mess.text[i]);
19
20             oi.close();
21         }
22         catch (Exception e) {
23             System.out.println("Error -- " + e.toString());
24         }
25     }
26 }
```


Exercícios

- Blz... Agora é hora de exercitar.....
- Tente resolver ou analisar os seguintes problemas..
 - Em dupla
 - Apresentar ao professor no final da aula



Exercício

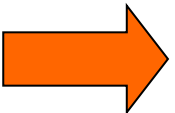
- Faça um programa que indique a existência ou não de um arquivo.
 - Implemente o valor de entrada como parâmetro da classe e verificações necessárias de erro
 - Consulte a documentação da API e verifique a classe e o método que trabalha com arquivos em Java.
 - Faça um teste com algum arquivo válido da máquina

Exercício

- Faça um programa que indica se o arquivo é um diretório.
 - Implemente o valor de entrada como parâmetro da classe e verificações necessárias de erro
 - Consulte a documentação da API e verifique a classe e o método que trabalha com arquivos em Java.
 - Faça um teste com algum diretório válido da máquina

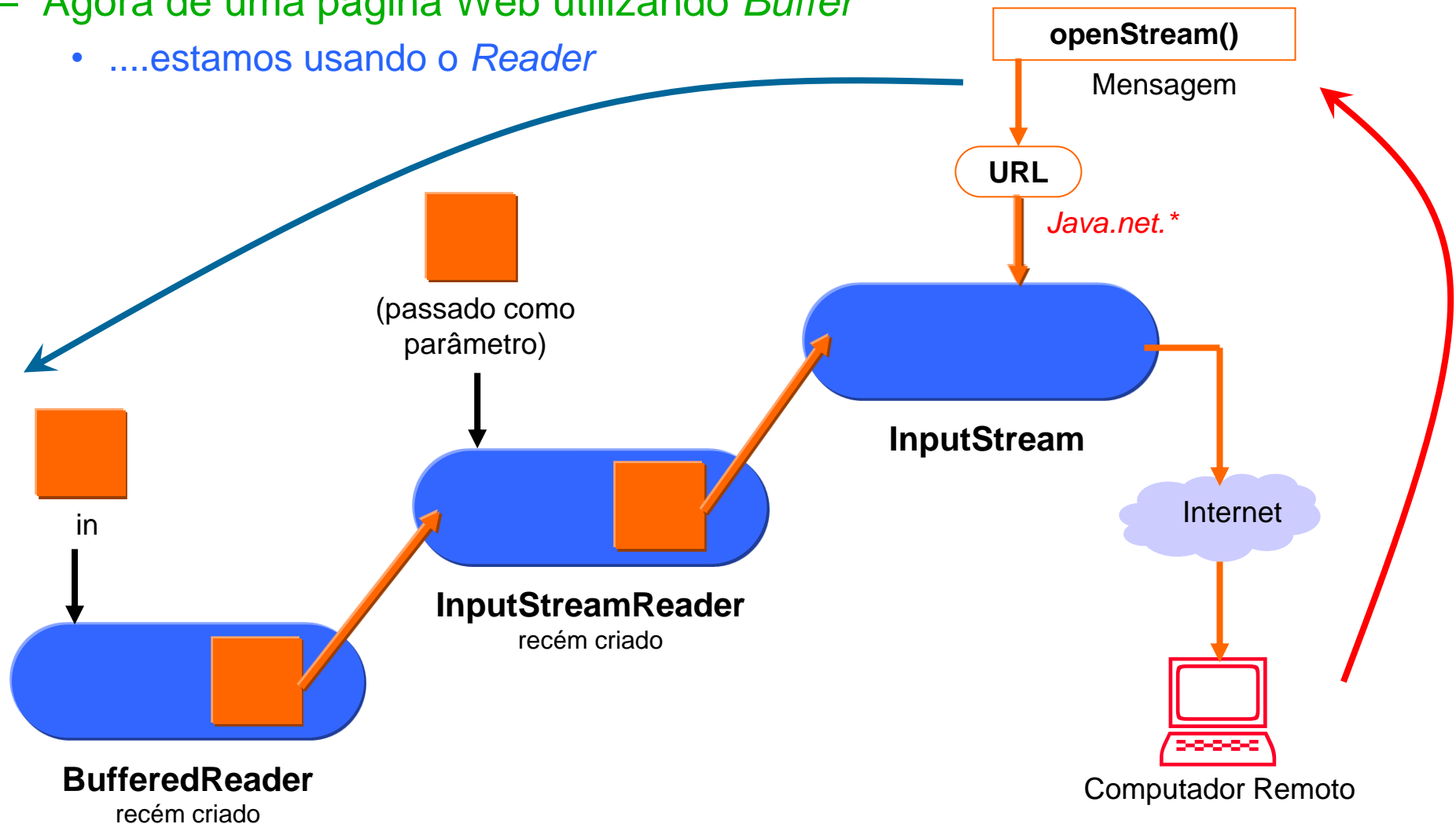
Exercício

- Desenvolva um programa que leia uma página na Web e a imprima na tela.
 - O programa deve ter as seguintes características
 - 1. Deve utilizar *stream* de caracteres com *Buffer* para obtenção dos dados
 - 2. Deve ler a linha de entrada como *String*
 - 3. Deve criar um objeto da classe URL e abrir uma conexão com a endereço <http://www.uvv.br>
 - 3.1 Qual método temos que usar para estabelecer a conexão
 - 4. Declare a exceção diretamente no método *main* ou capture com *try*.
 - Qual devemos utilizar?
 - 5. Utilize como condição de parada o retorno *null* do método *readLine()*
 - 6. Feche o *stream* ao final da computação
 - 7. Após executar o código, redirecione para um arquivo, via comando “>” no prompt



Exercício

- Exemplo de Leitura..
 - Agora de uma página Web utilizando *Buffer*
 - ...estamos usando o *Reader*



Exercício

- Desenvolva um programa que realize a soma de valores inteiros apresentados como entrada no console.
 - O programa deve ter as seguintes características
 - 0. Não vale usar `Scanner`... Nem sempre essa classe vai resolver nossos problemas de I/O...
 - 1. Deve utilizar *stream* de caracteres com *Buffer* para obtenção dos dados
 - 2. Deve ler a linha de entrada como *String*
 - 2.1. Qual método utilizar para essa leitura??
 - 2.2. O que deve ser feito para converter para a *String* para inteiro?
 - 3. Qual classe de exceção devemos utilizar no *catch*?
 - 4. Utilize como condição de parada o retorno *null* para o método utilizado em 2.1
 - Testes
 - 5. O que acontece se a entrada fornecida não for um inteiro?

Exercício

- *Streams* de dados *bufferizados* para saída de arquivos
 - Escreve um programa que:
 - Encontre e armazene, em um vetor, os N primeiros números primos.
 - N é um inteiro positivo qualquer
 - Números primos são aqueles que são divisíveis somente por 1 e por ele mesmo.
 - Escreve um método que verifique se o número é primo ou não e retorne um boolean
 - `public static boolean isPrime(int checkNumber)`
 - Escreva o resultado armazenado no vetor em um arquivo chamado primos.dat (**stream de bytes**)
 - Utilize *FileOutputStream*, *BufferedOutputStream* e *DataOutputStream*
 - A exceção *IOException* deve ser utilizada para capturar possíveis problemas nos *streams*
 - Feche o *stream* de dados com o método *close()*

Exercício

- *Streams* de dados *bufferizados* para entrada de arquivos
 - Escreve um programa que:
 - Leia o resultado armazenado no arquivo chamado *primos.dat* e imprima na tela.
 - Utilize *FileOutputStream*, *BufferedOutputStream* e *DataOutputStream*
 - Utilize a exceção *EOFException* para capturar o final do arquivo
 - É importante observar que os diferentes métodos de leitura de um *stream* de entrada de dados não retornam um valor que possa ser usado como indicador de que o final do *stream* foi alcançado.
 - Como alternativa, podemos esperar que uma *EOFException* (exceção de fim de arquivo) seja gerada quando um método de leitura atingir o fim do *stream*.
 - A exceção *IOException* deve ser utilizada para capturar possíveis problemas nos *streams*
 - Feche o *stream* de dados com o método *close()*

Exercício

- Serializando Objetos – Exercício (Parte A e B)
 - Criar 3 objetos, gravar em um arquivo, e recuperá-los, mostrando na tela
- (Parte A)
 - Modificar a classe fornecida (classe `PersistenciaDeObjetos_Pessoa_O`) para que ele possa ser serializada.
 - O que esta classe está modelando?

`PersistenciaDeObjetos_Pessoa_O.java`

Exercício

PersistenciaDeObjetos.java

- Serializando Objetos – Exercício (Parte A e B)
 - Criar 3 objetos, gravar em um arquivo, e recuperá-los, mostrando na tela
- (Parte B)
 - Para facilitar, crie uma classe programa onde todas as variáveis serão declaradas dentro do método *main()*
 - ***public static void main(String args[])***
 - Crie para escrita em arquivo as seguintes variáveis
 - ***File outfile;***
 - ***FileOutputStream outstream;***
 - ***ObjectOutputStream out;***
 - Crie para leitura de arquivo as seguintes variáveis
 - ***File infile;***
 - ***FileInputStream instream;***
 - ***ObjectInputStream in;***
 - O nome do arquivo para leitura e escrita será “Objeto.dat”
 - Use métodos da classe File para deletar e criar o arquivo “Objeto.dat”
 - Não esquecer de utilizar o *close()* para fechar os *streams*

• Outros Exemplos - Serializando o objeto

I/O em Java - Arquivos e fluxos

O que o programa está fazendo?

```
1 import java.util.Date;
2 import java.io.*;
3
4 class Info implements Serializable {
5     private String texto;
6     private float numero;
7     private Dado dado;
8     public Info(String t, float n, Dado d){
9         texto = t; numero = n; dado =d;
10    }
11    public String toString(){
12        return texto +","+numero+","+dado;
13    }
14 }
15
16 class Dado implements Serializable {
17     private Integer numero;
18     private Date data;
19     public Dado(Integer n, Date d){
20         numero = n; data = d;
21     }
22     public String toString(){
23         return "("+data +":"+numero +")";
24     }
25 }
26
27 public class TesteSerializacaoEDeserializacao {
28     public static void main(String [] args) throws Exception {
29         Info [] vetor =new Info [] {
30             new Info("Um",1.1f, new Dado(10,new Date())),
31             new Info("Dois",2.2f, new Dado(20,new Date()))
32         };
33
34         ObjectOutputStream out = new ObjectOutputStream(new FileOutputStream("objs.dat"));
35         out.writeObject("Os dados serializados foram: ");
36         out.writeObject(vetor);
37         out.close();
38
39         ObjectInputStream in =new ObjectInputStream(new FileInputStream("objs.dat"));
40         String msg =(String)in.readObject();
41         Info [] i =(Info [])in.readObject();
42         in.close();
43         System.out.println(msg +"\n"+i [0 ] +"\n"+i [1 ] );
44     }
45 }
```

Parte II

Exercício

- Valendo ponto em sala de aula....
 - Em dupla.. Entregar até o final da aula!!!!
- 1) Desenvolva o solicitado em Atividades Extra - Streams.pdf
- 2) Dado a implementação do pacote SistemaProfessor-Aluno.zip
 - Modifique a parte de persistência para que a mesma possa ser armazenada e recuperada em arquivo agora de FORMA SERIALIZADA!!!.

Anexo I

I/O em Java


- Relacionamentos importantes – Serializando Objetos e Composição/Aggregação

- Vamos analisar alguns comportamentos interessantes sobre serialização...

➡ Caso 1:

- O que acontece se temos uma classe que é serializado e dentro dele tem um atributo que é uma classe que não é serializado???

```
1  import java.io.*;
2
3  class Cat implements Serializable {
4      int quantPatas = 4;
5      Dono d = new Dono();
6  }
7
8  // Caso 1
9  class Dono {
10     String nome = "Eu";
11 }
12
13
```



I/O em Java

- Relacionamentos importantes – Serializando Objetos e Herança

- Vamos analisar alguns comportamentos interessantes sobre serialização...

➡ Caso 2:

- O que acontece se temos uma classe pai que É serializado e uma classe filha que NÃO é serializada e tentamos persistir a classe filha

➡ Caso 3:

- O que acontece se temos uma classe pai que NÃO é serializado e uma classe filha que É serializada e tentamos persistir a classe filha

- The object to be persisted must implement the Serializable interface or inherit that implementation from its object hierarchy

I/O em Java

- Relacionamentos importantes – Static

- Vamos analisar alguns comportamentos interessantes sobre serialização...

➡ Caso 4:

- Quando se diz: “A serialização não está para static” isso não quer dizer que seu código não vai compilar ou vai lançar alguma exceção caso tente serializar uma variável static.
 - Simplesmente diz que não é possível gravar o estado de uma variável static. Mas por quê?

```
1  import java.io.*;
2
3  class CatS implements Serializable {
4      |
5      |     int quantPatas = 4;
6      |     Dono d = new Dono();
7      |
8      | }
9
10
11  // Caso 1
12  class Dono implements Serializable {
13      |
14      |     static String nome = "Eu";
15      |
16      | }
```

I/O em Java

- Serializando Objetos – Variáveis transientes

➡ Vimos que é muito simples serializar e deserializar um objeto

- ...ou seja, armazenar e recuperar o seu estado, as suas informações.

➡ Essa facilidade gera alguns problemas do tipo...

➡ O que fazer em relação a segurança???

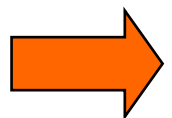
- Ex: Senhas persistidas são facilmente vistas...

➡ O que fazer em relação ao espaço de armazenamento;

- Ex: Necessidade de armazenar apenas as informações básica, descartando por exemplo, data

➡ O que fazer em relação a quanto temos necessidade de criar a variável toda vez que recuperar o objeto serializável, em vez de utilizar o que já foi criado;

- Ex: Usar sempre a data corrente



I/O em Java

- Serializando Objetos – Variáveis transientes

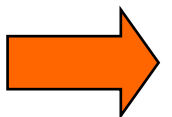
➡ Neste caso entra em ação o modificador *transient*

➡ É utilizado para evitar que uma variável de instância seja incluída na serialização,

➡ Esse modificador é incluído na instrução que cria a variável de classe.

- Ex: *transient String from, to;*

```
class Message implements Serializable {  
    int lineCount;  
    ➡ transient String from, to;  
    Date when;  
    String[] text;
```



I/O em Java

- Serializando Objetos – Variáveis transientes

➡ Pergunta que não quer calar.....

- Se os dados não são gravados, o que vai aparecer quando eles forem recuperados???

➡ O que acontece se:

- Executarmos o código ObjectToDisk.java
- Declarando “transient String from, to;” na classe Message???
- E depois executarmos o código ObjectFromDisk.java

– Vamos fazer isso!!!.....

I/O em Java - Arquivos e fluxos

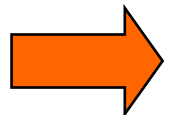
- Serializando Objetos – Variáveis transientes

➡ Outros exemplos...

- NossaClasseSerializada.java
- Serializa.java
- RecuperaSerializado.java

I/O em Java - Arquivos e fluxos

- Blz,
 - Aprendemos como podemos armazenar e recuperar os nossos objetos...
 - Um problema que pode acontecer é o arquivo de persistência ficar muito grande..
 - O que não seria legal, por exemplo, pra enviar via rede...
- ⇒ Será que tem como diminuir o tamanho dos arquivos com os objetos serializados???



I/O em Java - Arquivos e fluxos

- Fluxo com compactação de arquivos - Zip e Jar
 - ➔ Os pacotes `java.util.zip` e `java.util.jar` permitem comprimir dados e colecionar arquivos mantendo intactas as estruturas de diretórios
 - ➔ Podemos usar as classes de ZIP e JAR para coleções de arquivos
 - `ZipEntry`, `ZipFile`, `ZipInputStream`, etc.
 - ➔ Podemos usar o stream GZIP para arquivos individuais e para reduzir tamanho de dados enviados pela rede
 - `GZIPOutputStream`, `GZIPInputStream`

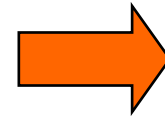
I/O em Java - Arquivos e fluxos

- Fluxo com compactação de arquivos - Zip e Jar

Vantagens

- Menor tamanho temos maior eficiência de E/S e menor espaço em disco
- Menos arquivos para transferir pela rede (também maior eficiência de E/S)

 Vamos analisar um exemplo utilizando o GZip.....



I/O em Java - Arquivos e fluxos

- Fluxo com compactação de arquivos - GZip

➡ GZIP usa o mesmo algoritmo usado em ZIP e JAR mas não agrupa coleções de arquivos

- **GZIPOutputStream** comprime dados na gravação
- **GZIPInputStream** expande dados durante a leitura

➡ É stream, logo para usá-los, basta incluí-los na cadeia de *streams*:

```
ObjectOutputStream out = new ObjectOutputStream(  
    ➡ new java.util.zip.GZIPOutputStream(  
        new FileOutputStream(armario) ) );  
  
Objeto gravado = new Objeto();  
out.writeObject(gravado);  
  
// (...)  
  
ObjectInputStream in = new ObjectInputStream(  
    ➡ new java.util.zip.GZIPInputStream(  
        new FileInputStream(armario) ) );  
  
Objeto recuperado = (Objeto)in.readObject();
```

Um *stream* para compactar e outro para descompactar...

I/O em Java - Arquivos e fluxos

Serializando o objeto compactado

```
import java.io.*;
import java.util.*;
import java.util.zip.*;

public class ArquivoCompactado {
    public static void main(String[] arguments) {
        final int QUANT_MSG = 500;
        try {
            FileOutputStream fo = new FileOutputStream("MessageNaoCompactada.obj");
            ObjectOutputStream oo = new ObjectOutputStream(fo);

            FileOutputStream foc = new FileOutputStream("MessageCompactada.obj");
            GZIPOutputStream co = new GZIPOutputStream(foc);
            ObjectOutputStream ooc = new ObjectOutputStream(co);

            System.out.println("===== ARMAZENADO OS DADOS NO ARQUIVO COMPACTADO =====");
            for (int i = 0; i < QUANT_MSG; i++){
                Message messOut = new Message();
                String author = "Vinicius Rosalen, Brasil";
                String recipient = "Alunos da UVV, UVV - Boa Vista";
                String[] letter = { "Exercicio de serializao de objetos.",
                                    "Depois daqui vamos serializar objetos para todo mundo.",
                                    "dessa forma vamos enviar objetos pelo mundo." };
                Date now = new Date();
                messOut.writeMessage(author, recipient, now, letter);
                oo.writeObject(messOut);
                ooc.writeObject(messOut);
            }
            oo.close();
            ooc.close();

            System.out.println("Objeto criado com sucesso.\n\n");
            // FileInputStream fic = new FileInputStream("MessageNaoCompactada.obj");
            FileInputStream fic = new FileInputStream("MessageCompactada.obj");
            GZIPInputStream ci = new GZIPInputStream(fic);
            ObjectInputStream oic = new ObjectInputStream(ci);

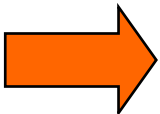
            System.out.println("===== RECUPERANDO OS DADOS DO ARQUIVO COMPACTADO =====");
            for (int i = 0; i < QUANT_MSG; i++){
                Message messIn = (Message) oic.readObject();
                System.out.println("ID: " + i);
                System.out.println("Mensagem:\n");
                System.out.println("Remetente: " + messIn.from);
                System.out.println("Para: " + messIn.to);
                System.out.println("Data: " + messIn.when + "\n");

                for (int j = 0; j < messIn.lineCount; j++)
                    System.out.println(messIn.text[j]);
            }
            oic.close();
        }
        catch (IOException e) {
            System.out.println("Error -- " + e.toString());
        }
        catch (ClassNotFoundException e) {
            System.out.println("Error -- " + e.toString());
        }
    }
}
```

O que o programa
está fazendo?

I/O em Java

- Blz,
 - Agora que entendemos e trabalhamos com vários tipos de *streams* diferentes
 - Podemos conversar sobre uns tipos de *streams* bastante utilizado...
 - Que permite um fluxo de impressão de saída de forma simplificada.....esses *streams* são os.....



I/O em Java

-*PrintStream* e *PrintWriter*

➡ O *PrintStream* e *PrintWriter* são outros conjunto de classes em Java que lidam com *stream*.

- Qual o programador vai utilizar? Depende do que é requisito do programa

➡ A classe *PrintStream* adiciona funcionalidades, para um *stream* de saída, de representar a impressão de vários valores dos dados convenientemente

- Ou seja, formatar a saída e o tipo impresso
 - Ex.: `println()`, `print(int i)`, `print (long i)`, colocar `\n` no final

➡ A classe *PrintWriter* pode ser usada em situações que requerem a escrita de caracteres em vez *bytes* puros.

- Esta classe implementa todos os métodos `print` encontrados em *PrintStream*, mas a saída sempre é em formato texto (*String*)

I/O em Java

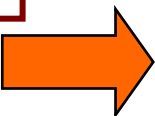
- Fluxo de impressão - *streams* utilizando *PrintStream* e *PrintWriter*
 - Métodos das classes *PrintWriter* e *PrintStream*

void print(String s)	void println(int i)
void println(String s)	void println(long l)
void println(char s[])	void println(float f)
void println(char c)	void println(double d)
void println(boolean b)	

- Todos os caracteres impressos pelo *PrintStream* são convertidos em *bytes*
 - Usando a codificação da representação da plataforma corrente.
- Podemos ver claramente a sobrecarga do método print



Resumo: Você não se preocupa com o tipo de dados que vai ser “impresso”



I/O em Java

- Fluxo de impressão - *streams* utilizando *PrintStream*
 - O que o código abaixo está fazendo?

```
1  import java.io.*;
2
3  public class EscreverPrintStream {
4
5      public static void main(String args[]) {
6          int i = 1;
7          try {
8              //Abrir um arquivo para gravacao
9              FileOutputStream argsai = new FileOutputStream("OutPrintStream.txt");
10
11              //Abrindo um fluxo de impresso - Usando PrintStream
12              PrintStream saida = new PrintStream(argsai);
13              saida.println("Joao, 23, 1000");
14              saida.println("Maria, 30, 450");
15              saida.print("i + 1 = ");
16              saida.println(i + 1);
17              saida.print(i + " + 2 = ");
18              saida.println(i + 2);
19              saida.print("valor double = ");
20              saida.println(2.45);
21              saida.print("valor boolean = ");
22              saida.println(true);
23              saida.print("valor char = ");
24              saida.println('V');
25
26
27          }
28          catch(IOException e) {
29              System.out.println("Erro: "+e.getMessage());
30          }
31      }
32  }
```

I/O em Java

- Fluxo de impressão - *streams* utilizando *PrintStream*
 - Interessante...outra forma... O que o código abaixo está fazendo?

```
try {
    System.out.print("Digite algo para guardar em arquivo:");
    Scanner s = new Scanner(System.in);
    PrintStream ps = new PrintStream("OutPrintStream2.txt");

    while(s.hasNextLine()) {
        ps.println(s.nextLine());
    }

} catch (FileNotFoundException e) {
    System.out.println("Erro 2: "+e.getMessage());
}
```

EOF

Quando rodar sua aplicação, para encerrar a entrada de dados do teclado, é necessário enviarmos um sinal de fim de stream. É o famoso EOF, End Of File.

No Linux/Mac/Solaris/Unix você faz isso com o CONTROL+D. No Windows, use o CONTROL+Z.

I/O em Java

- Fluxo de impressão - *streams* utilizando *PrintWriter*
 - O que o código abaixo está fazendo?

```
1 import java.io.*;
2
3 public class EscreverPrintWriter {
4
5     public static void main(String args[]) {
6         int i = 1;
7         try {
8             //Abrir um arquivo para gravacao
9             FileOutputStream arqsai = new FileOutputStream("OutPrintWriter.txt");
10
11             /*
12
13             // Usando PrintWriter COM flush automático
14             PrintWriter saida = new PrintWriter(arqsai, true);
15             saida.println("Joao, 23, 1000");
16             saida.println("Maria, 30, 450");
17             saida.print("valor double = ");
18             saida.println(2.45);
19             saida.print("valor boolean = ");
20             saida.println(true);
21
22             /*
23             // Usando PrintWriter SEM flush automático
24             PrintWriter saida = new PrintWriter(arqsai);
25             saida.println("Joao, 23, 1000");
26             saida.println("Maria, 30, 450");
27             saida.println(2.45);
28             saida.print("valor boolean = ");
29             saida.println(true);
30             saida.flush();
31
32             arqsai.close();
33         }
34         catch(IOException e) {
35             System.out.println("Erro: "+e.getMessage());
36         }
37     }
38 }
```

→

→

E se retirar o flush?

• Trabalhando com o Excel

```
1 import java.io.File;
2 import java.io.FileWriter;
3 import java.io.PrintWriter;
4 import java.util.Scanner;
5
6 public class IOArquivoExcel {
7
8     /**
9      * @param args
10     */
11     public static void main(String[] args) throws Exception {
12
13         Scanner scanner = new Scanner (new File("dados_alunos.csv"));
14         PrintWriter writerTxT = new PrintWriter(new File("saidaIOArquivoExcel.txt"));
15         PrintWriter writerCsV = new PrintWriter(new File("contatos_alunos.csv"));
16
17         while (scanner.hasNextLine()){
18             String linha = scanner.nextLine();
19             Scanner lineScanner = new Scanner(linha);
20             lineScanner.useDelimiter(",");
21
22             int id = lineScanner.nextInt();
23             String nome = lineScanner.next();
24             String cpf = lineScanner.next();
25             String telefone = lineScanner.next();
26             String email = lineScanner.next();
27
28             //System.out.println("ID: " + id);
29             writerTxT.println("ID: " + id);
30             //System.out.println("Nome: " + nome);
31             writerTxT.println("Nome: " + nome);
32             //System.out.println("CPF: " + cpf);
33             writerTxT.println("CPF: " + cpf);
34             //System.out.println("Telefone: " + telefone);
35             writerTxT.println("Telefone: " + telefone);
36             //System.out.println("E-mail: " + email);
37             writerTxT.println("E-mail: " + email);
38             //System.out.println("");
39             writerTxT.println("");
40
41             writerCsV.println(nome+","+telefone+","+email);
42
43             System.out.println("OK");
44         }
45         writerTxT.close();
46         writerCsV.close();
47     }
48 }
49 }
```

I/O em Java

O que o programa
está fazendo?

I/O em Java

- Blz..., Vamos agora ter mais uma palavrinha a mais sobre arquivos com Java...

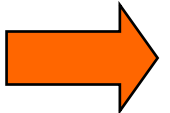
⇒ Até agora, os fluxos, leitores e escritores que vimos fazem apenas acesso sequencial;

⇒ Às vezes precisamos de acessar diferentes posições do arquivo

- Ex.:banco de dados:

- Uma determinada posição do arquivo pode se acessada diretamente;
- Tratamento mais eficiente de grandes volumes de dados.

⇒ É aí que entra a classe `RandomAccessFile`



I/O em Java

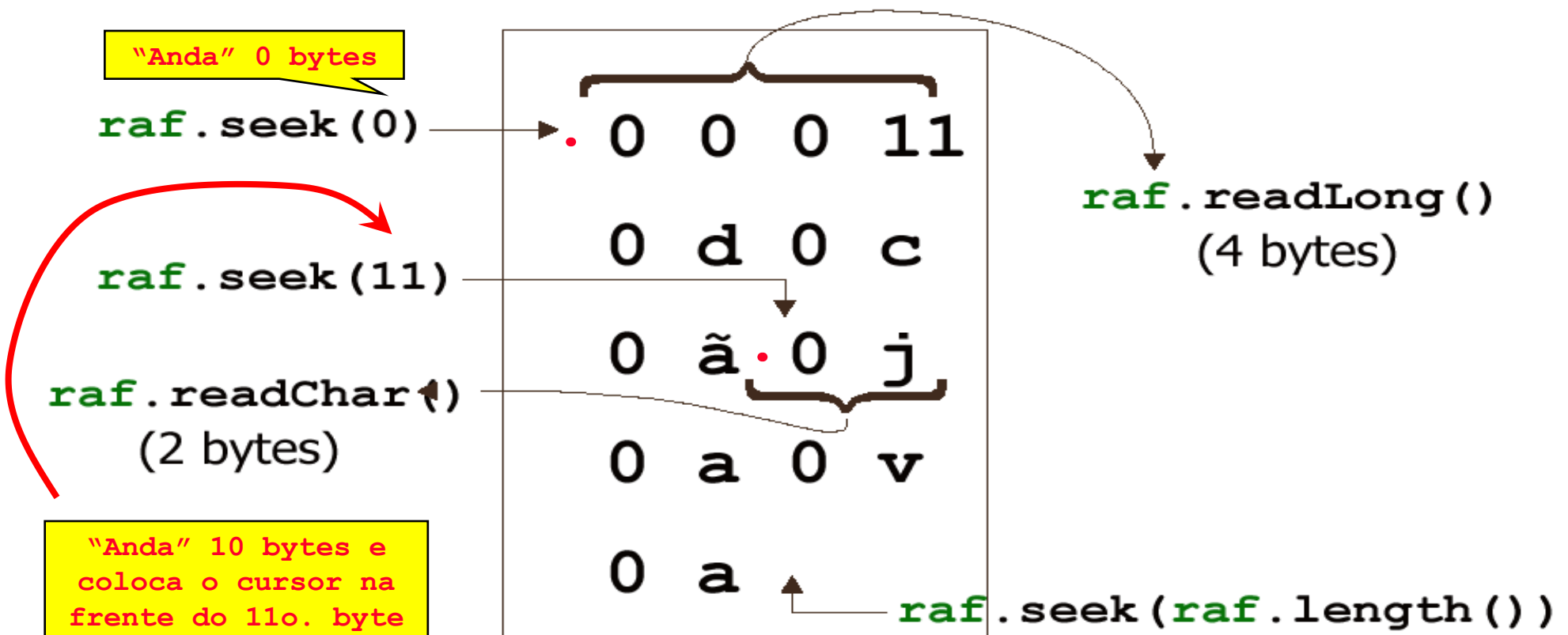
- Características das classe `RandomAccessFile`

- ⇒ A classe `java.io.RandomAccessFile` permite a leitura e escrita em um arquivo de acesso randômico
- ⇒ Implementa as interfaces *`DataInput`* e *`DataOutput`*
 - Mistura de *`DataInputStream`* com *`DataOutputStream`* ;
- ⇒ Possui um *`file pointer`* que indica a posição (índice) corrente
 - O *`file pointer`* pode ser obtido através do método *`getFilePointer`*
 - E alterado através do método *`seek`*
- ⇒ Na criação, especifica-se o arquivo e o modo de operação: "r" ou "rw";

I/O em Java

- Exemplo: RandomAccessFile

```
RandomAccessFile raf =  
    new RandomAccessFile ("arquivo.dat", "rw");
```



```

1 import java.io.*;
2
3
4 public class RandomAccessFileSeek {
5     public static void Le (long pos, RandomAccessFile raf) throws IOException {
6         int c;
7
8         raf.seek(pos);
9         c = raf.readInt();
10        System.out.println("O int na posicao\t" + pos + "\teh: " + c);
11    }
12
13    public static void main(String[] args)    {
14        try {
15            // Cria uma arquivo de acesso randomico.
16            RandomAccessFile raf = new RandomAccessFile("RandomFile.txt", "rw");
17
18            // Todos inteiros tem 4 bytes, logo o seek tem que andar de 4 em 4
19            // Neste exemplo temos 6 inteiros no arquivo => 6 x 4 = 24 bytes
20            raf.writeInt(1);
21            raf.writeInt(2);
22            raf.writeInt(3);
23            raf.writeInt(4);
24            raf.writeInt(5);
25            raf.writeInt(6);
26
27            RandomAccessFileSeek.Le(0, raf);
28            RandomAccessFileSeek.Le(4, raf);
29            RandomAccessFileSeek.Le(8, raf);
30            RandomAccessFileSeek.Le(12, raf);
31            RandomAccessFileSeek.Le(16, raf);
32            RandomAccessFileSeek.Le(20, raf);
33
34            // Close the file.
35            raf.close();
36        }
37        catch (IOException ex) {
38            System.out.println(ex.toString());
39        }
40    }
41 }

```

I/O em Java

O que o programa
está fazendo?

```

1 import java.io.*;
2 public class TesteRandom{
3
4     public void escreve(RandomAccessFile raf) throws IOException {
5         char[] letras = {'a', 'b', 'c', 'd'};
6         for(int i = 0; i < letras.length; i++){
7             // Writes a char to the file as a two-byte value, high byte first.
8             // ou seja, de dois em dois, logo os pares
9             raf.writeChar(letras[i]);
10        }
11    }
12
13    public void leUm(RandomAccessFile raf, int pos) throws IOException {
14        raf.seek(pos);
15        System.out.println(raf.readChar());
16    }
17
18    public void escreveUm(RandomAccessFile raf, int pos, char c) throws IOException {
19        raf.seek(pos);
20        raf.writeChar(c);
21    }
22
23    public static void main(String argv[]){
24        try {
25            File inputFile = new File("TesteRandom.txt");
26            TesteRandom r = new TesteRandom();
27            if (inputFile.exists()) System.out.println(inputFile.delete());
28            //System.out.println(inputFile.exists());
29            RandomAccessFile raf = new RandomAccessFile(inputFile, "rw");
30            //System.out.println(inputFile.exists());
31
32            System.out.println("Tamanho inicial do arquivo: " + raf.length());
33            r.escreve(raf);
34            r.leUm(raf, 2);
35            r.escreveUm(raf, 2, 'x');
36            r.leUm(raf, 2);
37            r.escreveUm(raf, 8, 'f');
38            r.leUm(raf, 8);
39            r.escreveUm(raf, 10, '\0');
40            System.out.println("Tamanho final do arquivo: " + raf.length());
41
42            raf.seek(0);
43            char ch = raf.readChar();
44            while (ch != '\0') {
45                System.out.print(ch + " ");
46                ch = raf.readChar();
47            }
48
49        }
50        catch(IOException ioe) {
51            System.out.println(ioe);
52        }
53
54        System.out.println("");
55    }
56 }

```

I/O em Java

O que o programa
está fazendo?

I/O em Java

O que o programa está fazendo?