nemo

ontology & conceptual modeling research group

Desenvolvimento OO com Java

10 – Arquivos e fluxos

Vítor E. Silva Souza

(vitorsouza@inf.ufes.br)

http://www.inf.ufes.br/~vitorsouza

Departamento de Informática

Centro Tecnológico

Universidade Federal do Espírito Santo

Objetivos da parte 10

- Mostrar como Java lida com arquivos e fluxos de dados;
- Apresentar as classes que permitem leitura e escrita em diversas fontes de dados;
- Explicar o que é serialização e como funciona.

Armazenamento persistente

- Poucas aplicações são funcionais apenas com dados transientes;
- A grande maioria precisa armazenar informações em mídia de longa duração para recuperá-las tempos depois;
- Java trabalha com fluxos de dados, representando acesso a fontes como:
- Memória;
- Arquivos;
- Rede, etc.

Plataformas e sistemas de arquivo

- Diferentes sistemas operacionais representam arquivos e trilhas (paths) de diferentes formas:
- `C:\Documents and Settings\User\Arquivo.txt`;
- `/home/User/Arquivo.txt`.
- Java utiliza a classe `java.io.File`, abstraindo esta representação e provendo portabilidade.

```
// No Windows:
File f = new File("C:\\pasta\\arq.txt");

// No Linux/Unix/Mac:
File f = new File("/pasta/arq.txt");
```

A classe java.io.File

- Pode representar arquivos ou diretórios:

```
File a1 = new File("arq1.txt");
File a2 = new File("/pasta", "arq2.txt");
File d = new File("/pasta");
File a3 = new File(d, "arq3.txt");
```

- Possui métodos úteis para manipulação:
  - canRead(), canWrite(), createNewFile(), delete(), exists(), getName(), getParentFile(), getPath(), isDirectory(), isFile(), isHidden(), lastModified(), length(), list(), listFiles(), mkdir(), mkdirs(), renameTo(), setLastModified(), setReadOnly(), etc.

Fluxos, leitores e escritores

- Até Java 1.4, I/O era feita por:
  - Fluxos (*streams*): subclasses de `InputStream` e `OutputStream` para leitura/escrita byte a byte;
  - Leitores (*readers*) e escritores (*writers*): subclasses de `Reader` e `Writer` para leitura/escrita caractere a caractere (padrão Unicode).
- A partir do Java 5:
  - Foi criada a classe `java.util.Scanner` para facilitar a leitura;
  - Foram adicionados métodos à classe `PrintWriter` para facilitar a escrita (ex.: `printf()`).

Modo de operação

- Cria-se o fluxo, leitor, escritor ou scanner e este estará aberto automaticamente;
- Utiliza-se operações de leitura e escrita:
  - Operações de leitura podem bloquear o processo no caso dos dados não estarem disponíveis;
  - Métodos como `available()` indicam quantos bytes estão disponíveis.
- Fecha-se o fluxo, leitor, escritor ou scanner:
  - A omissão da chamada ao método `close()` pode provocar desperdício de recursos ou leitura/escrita incompleta.

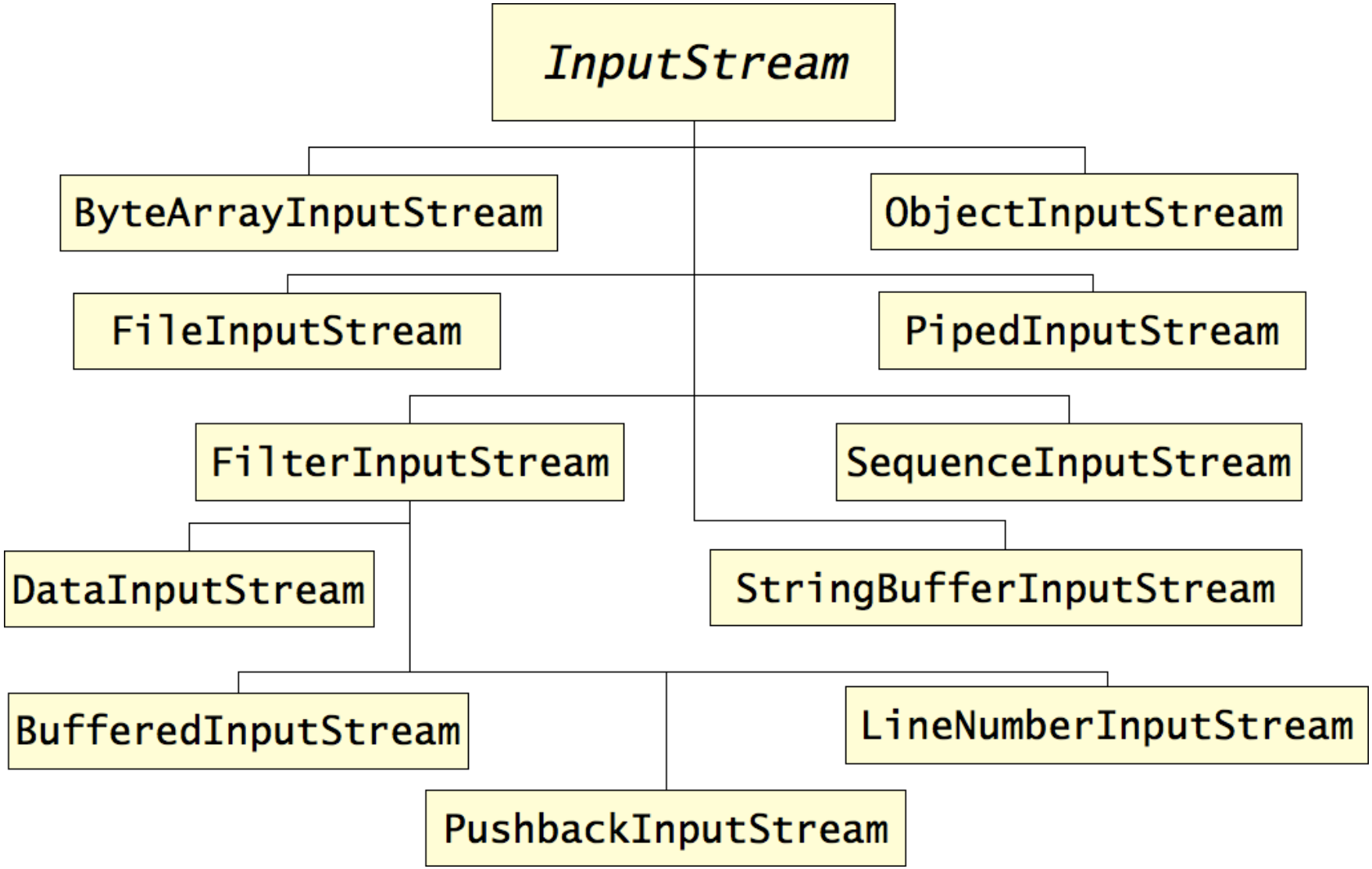
Aplicando o polimorfismo

- Métodos definidos nas classes abstratas e disponíveis em toda a hierarquia:
  - `InputStream`: `available()`, `close()`, `read()`, `read(byte[] b)`, `reset()`, `skip(long l)`, etc.;
  - `OutputStream`: `close()`, `flush()`, `write(int b)`, `write(byte[] b)`, etc.;
  - `Reader`: `close()`, `mark()`, `read()`, `read(char[] c)`, `ready()`, `reset()`, `skip(long l)`, etc.;
  - `Writer`: `append(char c)`, `close()`, `flush()`, `write(char[] c)`, `write(int c)`, `write(String s)`, etc.

A hierarquia de I/O

- São mais de 40 classes, divididas em:
  - Fluxos de entrada (*input streams*);
  - Fluxos de saída (*output streams*);
  - Leitores (*readers*);
  - Escritores (*writers*);
  - Arquivo de acesso aleatório (*random access file*).
- Classes podem indicar a mídia de I/O ou a forma de manipulação dos dados;
- Podem (devem) ser combinadas para atingirmos o resultado desejado.

Fluxos de entrada



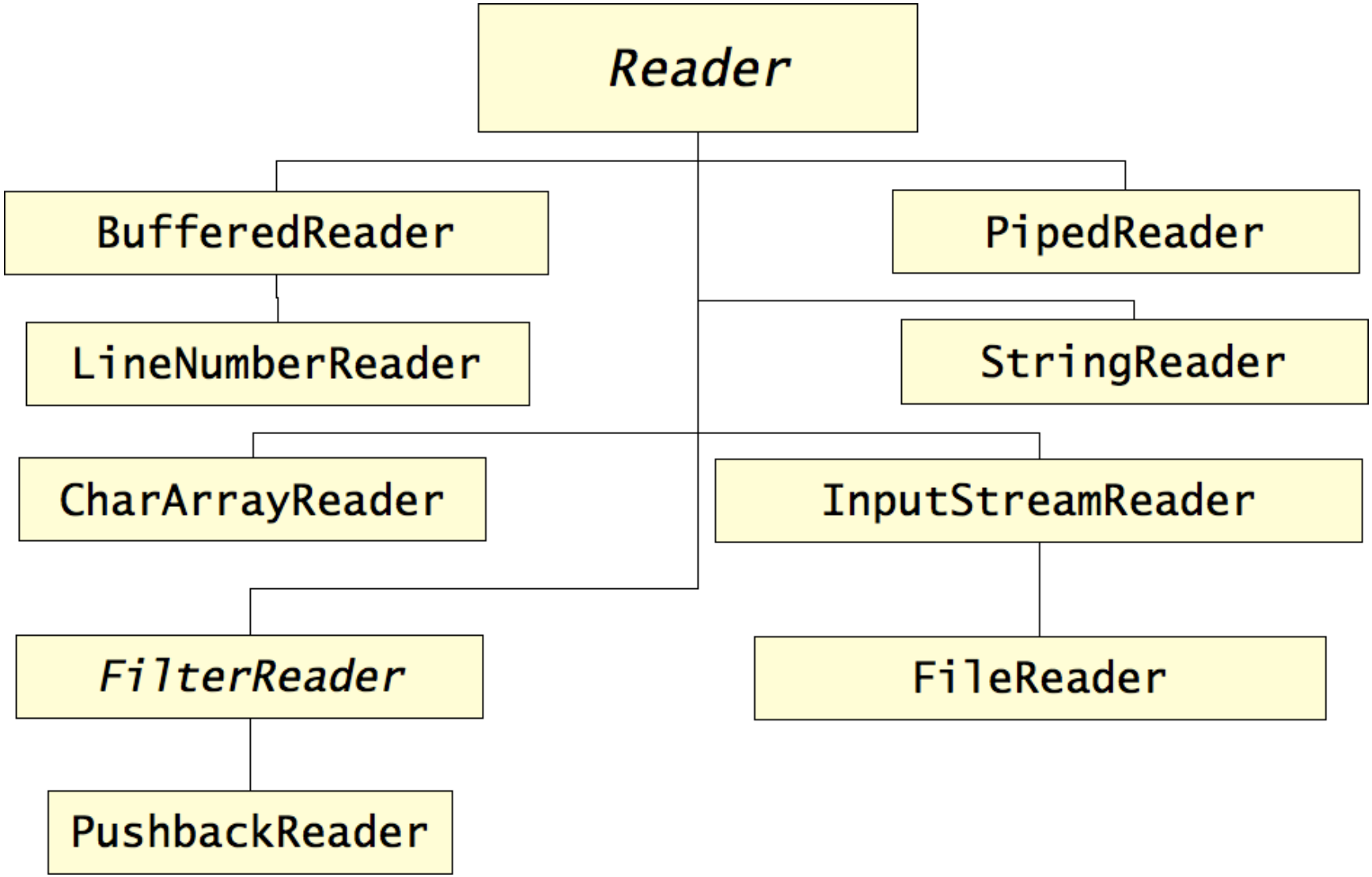
Combinando fluxos

```
// Lê de um arquivo.
FileInputStream fin = new
FileInputStream("arquivo.txt");

// Efetua leitura com buffer.
BufferedInputStream bin = new BufferedInputStream(fin);

// Provê métodos de acesso a tipos de dados.
DataInputStream in = new DataInputStream(bin);

// Resumidamente:
DataInputStream in2 = new DataInputStream(new
BufferedInputStream(new FileInputStream("arq.txt")));
```

Exemplo com fluxo de entrada

```
// Dentro do main. Classe deve importar java.io.*.
try {
  DataInputStream in = new DataInputStream(new
BufferedInputStream(new FileInputStream("arq.txt")));

  String linha;
  String buffer = new String();
  linha = in.readLine();          // Deprecated.
  while (linha != null) {
     buffer += linha + "\n";
     linha = in.readLine();       // Deprecated.
  }
  in.close();
} catch (IOException e) {
  System.out.println("Erro de I/O");
  e.printStackTrace();
}
```

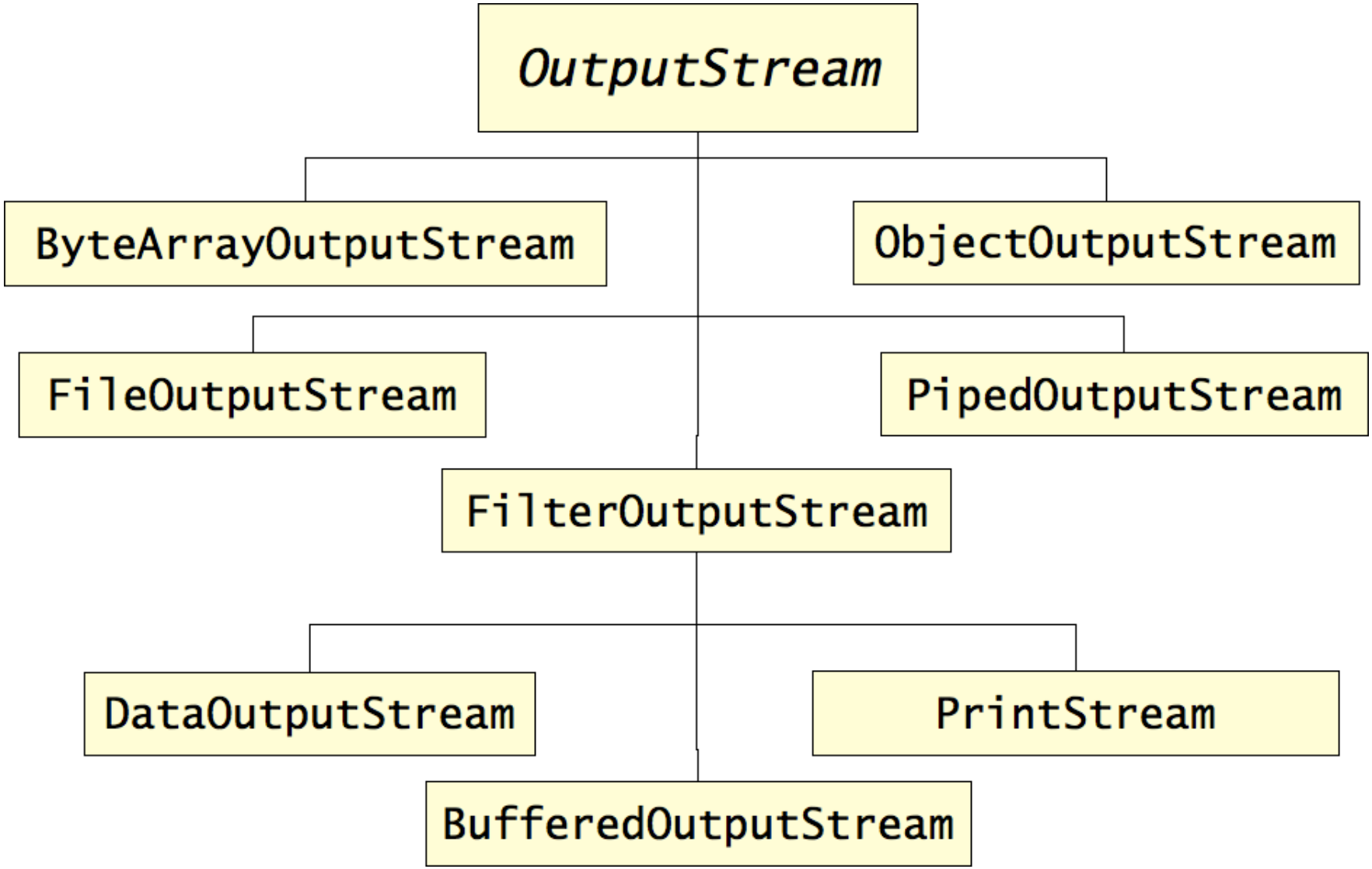
Leitores


Reader
BufferedReader
PipedReader
LineNumberReader
StringReader
CharArrayReader
InputStreamReader
FilterReader
FileReader
PushbackReader

Exemplo com leitor

```
// Dentro do main. Classe deve importar java.io.*.
try {
  BufferedReader reader = new BufferedReader(new
                              FileReader("arq.txt"));

  String linha;
  String buffer = new String();
  linha = reader.readLine();
  while (linha != null) {
     buffer += linha + "\n";
     linha = reader.readLine();
  }
  reader.close();
} catch (IOException e) {
  System.out.println("Erro de I/O");
  e.printStackTrace();
}
```

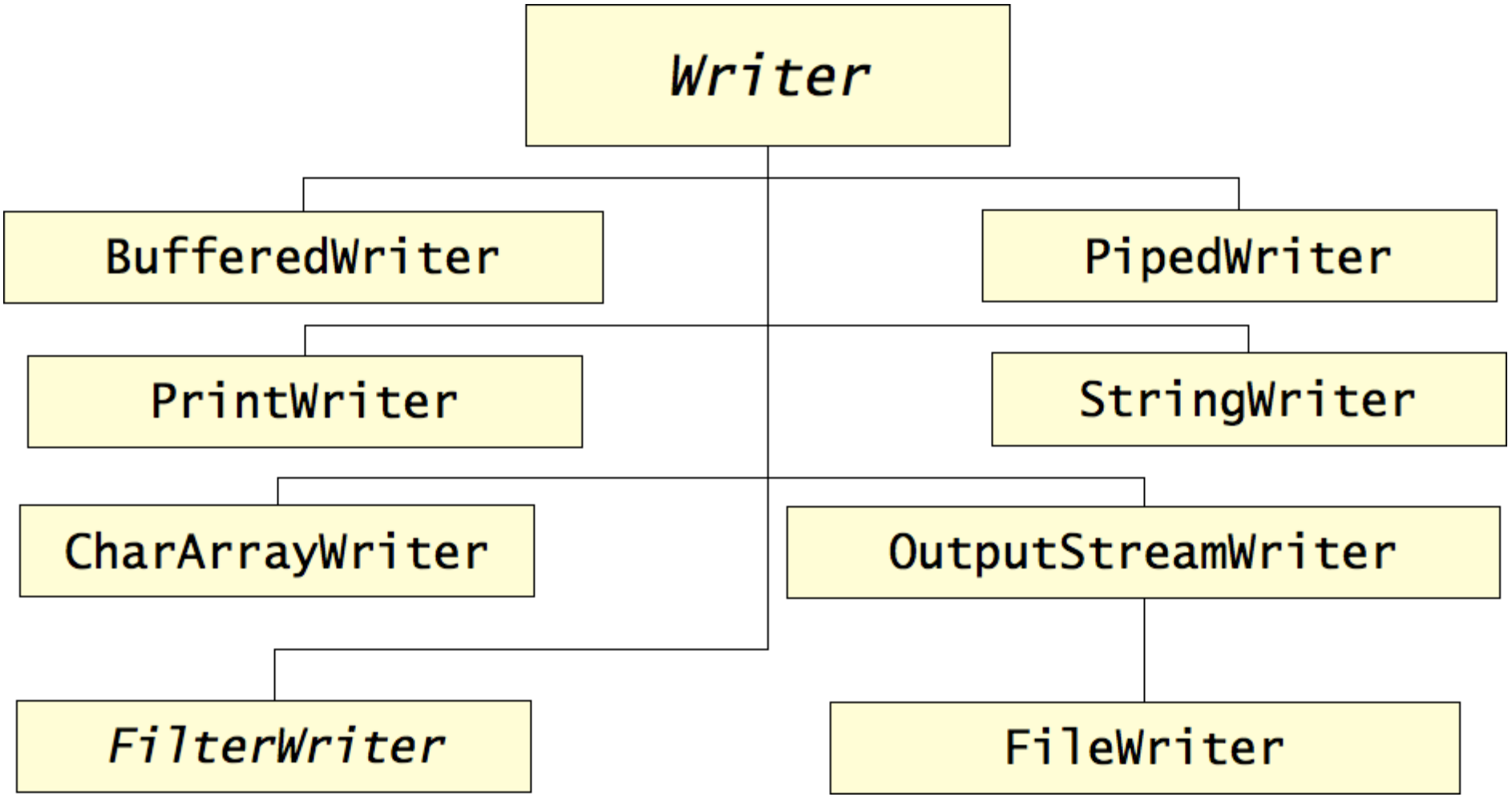
Fluxos de saída



Exemplo com fluxo de saída

```
// Dentro do main. Classe deve importar java.io.*.
try {
  DataOutputStream out = new DataOutputStream(new
          BufferedOutputStream(new
          FileOutputStream("arq.txt")));

  out.writeBytes("Uma frase...");
  out.writeDouble(123.4567);
  out.close();
}
catch (IOException exc) {
  System.out.println("Erro de IO");
  exc.printStackTrace();
}
```

Escritores


Writer
BufferedWriter
PipedWriter
PrintWriter
StringWriter
CharArrayWriter
OutputStreamWriter
FilterWriter
FileWriter

Exemplo com escritor

```
// Dentro do main. Classe deve importar java.io.*.
try {
  BufferedWriter out = new BufferedWriter(new
                       FileWriter("arq.txt"));

  out.write("Uma frase...");
  out.write("" + 123.4567);
  out.close();
}
catch (IOException exc) {
  System.out.println("Erro de I/O");
  exc.printStackTrace();
}
```

E se houver uma exceção?

```
// Dentro do main. Classe deve importar java.io.*.
try {
  BufferedWriter out = new BufferedWriter(new
                       FileWriter("arq.txt"));

  out.write("Uma frase...");
  out.write("" + 123.4567);
  out.close();
}
catch (IOException exc) {
  System.out.println("Erro de I/O");
  exc.printStackTrace();
}
```

Um erro aqui...

o código pula pra cá...

e o escritor não foi fechado.

O problema se repete nos demais exemplos e fica ainda mais complicado quando múltiplos recursos são usados...

Usando múltiplos recursos

```
InputStream in = null; OutputStream out = null;
try {
  in = new FileInputStream(origem);
  out = new FileOutputStream(destino);
  byte[] buf = new byte[8192]; int n;
  while ((n = in.read(buf)) >= 0) out.write(buf, 0, n);
}
catch (FileNotFoundException | IOException ex) {
  System.out.println("Problemas com a cópia: " + ex);
}
finally {
  if (in != null) try { in.close(); }
  catch (IOException ex) {
     System.out.println("Problemas com a cópia: " + ex);
  }
  finally {
     if (out != null) try { out.close(); }
     catch (IOException ex) {
        System.out.println("Problemas com a cópia: " + ex);
     }
} }
```

Java 7: *try with resources*

- Gerenciamento automático de recursos “fecháveis”:

```
try (InputStream in = new FileInputStream(origem);
     OutputStream out = new FileOutputStream(destino);)
{
  byte[] buf = new byte[8192];
  int n;
  while ((n = in.read(buf)) >= 0)
    out.write(buf, 0, n);
}
catch (FileNotFoundException | IOException ex) {
  System.out.println("Problemas com a cópia: " + ex);
}
```

Acesso aleatório

- Fluxos, leitores e escritores fazem apenas acesso sequencial;
- Às vezes precisamos de acessar diferentes posições do arquivo (ex.: banco de dados):
  - Uma determinada posição do arquivo pode ser acessada diretamente;
  - Tratamento mais eficiente de grandes volumes de dados.
- A classe `java.io.RandomAccessFile` oferece suporte de leitura e escrita aleatória.

java.io.RandomAccessFile

- Mistura de `DataInputStream` com `DataOutputStream`;
- Na criação, especifica-se o arquivo e o modo de operação: `"r"` ou `"rw"`;
- Usa-se métodos de manipulação de dados: `close()`, `getFilePointer()`, `length()`, `read(byte[])`, `readBoolean()`, ..., `readLong()`, `readLine()`, `seek(long)`, `skipBytes(long)`, `write(byte[])`, `writeBoolean(boolean)`, ..., `writeLong(long)`.

java.util.Scanner

- Novidade do Java 5.0;
- Facilita a leitura de dados:
  - Construtores podem receber `File`, `InputStream`, `Reader` e `String`;
  - Divide em *tokens* com `useDelimiter(String);`
  - Faz leitura regionalizada com `useLocale(Locale);`
  - Obtém dados diretamente em seus tipos, com `next()`, `nextLine()`, `nextBoolean()`, `nextInt()`, `nextDouble()`, etc.

Java 7: nova API de I/O

- Limitações da API `java.io`:
  - Não há operação de cópia de arquivo;
  - Não há suporte para atributos de arquivos;
  - Não é 100% consistente nas diferentes plataformas;
  - Muitas vezes as exceções não são muito úteis;
  - Não é extensível para suportar novos sistemas de arquivo.
- Desde Java 1.4 existe a `java.nio` (*new I/O*), que adiciona canais de I/O;
- As limitações, porém, foram resolvidas somente no Java 7, com novos pacotes da `java.nio` (NIO.2).

Java 7: nova API de I/O

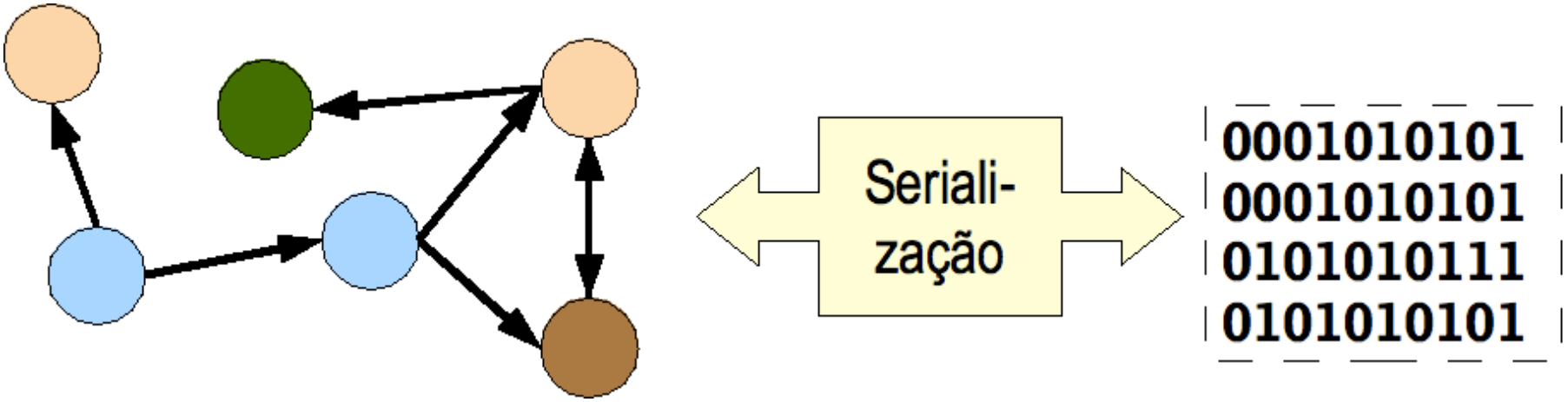
- Com a NIO.2, é possível:
  - Usar filtros *glob*. Ex.:
    ```
    Files.newDirectoryStream(home, "*.txt");
    ```
  - Manipular atributos de arquivos;
  - Navegação recursiva facilitada (*crawling*);
  - Monitoramento de eventos;
  - Etc.
- Muito avançado para inclusão neste curso...

Serialização

- `ObjectInputStream` e `ObjectOutputStream` são fluxos especiais;
- Ao contrário de tudo mais que vimos, eles não leem/escrevem dados de tipos primitivos;
- Serialização é o processo de converter um objeto em um fluxo de bits e vice-versa;
- Serve para gravá-lo em disco e enviá-lo pela rede para outro computador.

Serialização

- Problemas:
  - Um objeto pode possuir referências (ponteiros) para outros. Devemos "relativizá-las" quando formos serializar este objeto;
  - Ao restaurar o objeto a sua forma em memória, devemos recuperar as referências aos objetos certos.



java.io.Serializable

- Felizmente, Java já implementa este mecanismo;
- Basta que a classe que deve ser convertida implemente a interface `Serializable`;
  - Interface sem métodos, “sinalizadora”.
- Mecanismo de serialização:
  - Converte para bytes e vice-versa;
  - Faz e desfaz a relativização das referências;
  - Compensa diferenças entre sistemas operacionais;
  - Usa `ObjectInputStream` e `ObjectOutputStream`.

Exemplo de serialização

```
public class Info implements Serializable {
  private String texto;
  private float numero;
  private Dado dado;

  public Info(String t, float n, Dado d) {
     texto = t; numero = n; dado = d;
  }

  public String toString() {
     return texto + "," + numero + "," + dado;
  }
}
```

Exemplo de serialização

```
import java.util.Date;

public class Dado implements Serializable {
  private Integer numero;
  private Date data;

  public Dado(Integer n, Date d) {
     numero = n; data = d;
  }

  public String toString() {
     return "(" + data + ":" + numero + ")";
  }
}
```

Exemplo de serialização

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

public class Teste {
  public static void main(String[] args)
                                  throws Exception {
    Info[] vetor = new Info[] {
      new Info("Um", 1.1f,
                new Dado(10, new Date())),
      new Info("Dois", 2.2f,
                new Dado(20, new Date()))
    };

    /* Continua... */
```

Exemplo de serialização

```
    ObjectOutputStream out = new ObjectOutputStream(new
FileOutputStream("objs.dat"));
    out.writeObject("Os dados serializados foram:");
    out.writeObject(vetor);
    out.close();

    ObjectInputStream in = new ObjectInputStream(new
FileInputStream("objs.dat"));
    String msg = (String)in.readObject();
    Info[] i = (Info[])in.readObject();
    in.close();

    System.out.println(msg + "\n" + i[0]
                           + "\n" + i[1]);
  }
}
```

Conclusões

- Java possui uma gama enorme de classes para as mais variadas necessidades de I/O;
- Para facilitar, use `Scanner` para leitura e `PrintWriter` para escrita – outros fluxos, leitores ou escritores serão usados “nos bastidores”;
- Para serializar objetos, confie no mecanismo do Java, bastando implementar a interface sinalizadora `Serializable`.

http://nemo.inf.ufes.br/