Modulo II – Tópicos em Java - IO

Prof. Ismael H F Santos

Ementa

- Modulo II - Tópicos em JAVA - IO
  - Entrada e Saída - Streams
  - Filtros de Streams
  - Serializacao/Externalizacao de Objetos
  - XML Encoder e XML Decoder
  - Arquivos ZIP e JAR
  - Java New-IO
  - Arquivos de Propriedades

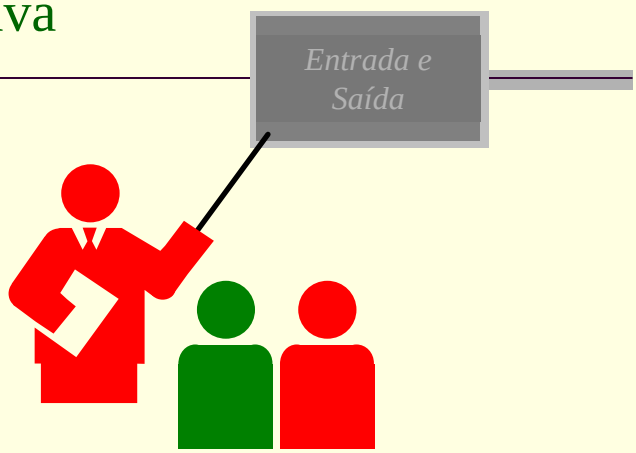
Bibliografia

- *Linguagem de Programação JAVA*
  - *Ismael H. F. Santos, Apostila UniverCidade, 2002*
- *The Java Tutorial: A practical guide for programmers*
  - *Tutorial on-line: http://java.sun.com/docs/books/tutorial*
- *Java in a Nutshell*
  - *David Flanagan, O´Reilly & Associates*
- *Just Java 2*
  - *Mark C. Chan, Steven W. Griffith e Anthony F. Iasi, Makron Books.*
- *Java 1.2*
  - *Laura Lemay & Rogers Cadenhead, Editora Campos*

Livros

- **Core Java 2**, Cay S. Horstmann, Gary Cornell
  - Volume 1 (Fundamentos)
  - Volume 2 (Características Avançadas)
- **Java: Como Programar**, Deitel & Deitel
- **Thinking in Patterns with JAVA**, Bruce Eckel
  - **Gratuito**. http://www.mindview.net/Books/TIJ/

POO-Java

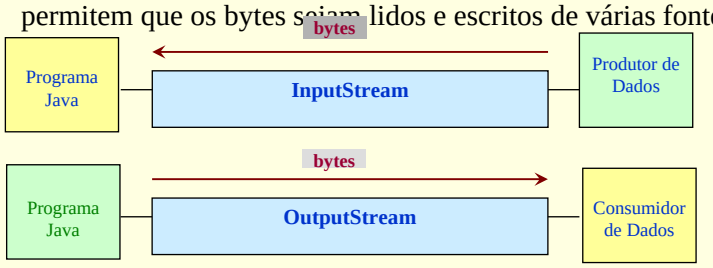


Motivação

- Uma aplicação normalmente precisa “obter” e/ou “enviar” informações a fontes/destinos externos
  - arquivos, conexões de rede, memória
- Essas informações podem ter vários tipos
  - bytes/caracteres, dados, objetos
- Java utiliza um mecanismo genérico que permite tratar E/S de forma uniforme
  - *Streams* de entrada e saída

I/O Streams

- Streams de Entrada e Saída
  - A entrada e saída em Java é elaborada por meio de streams. Uma stream é simplesmente um fluxo de dados.
  - A leitura/escrita de bytes é definida pela classe abstrata **InputStream/OutputStream**. Essa classe modela um canal (*stream*) através do qual bytes podem ser lidos. Suas subclasses permitem que os bytes sejam lidos e escritos de várias fontes.

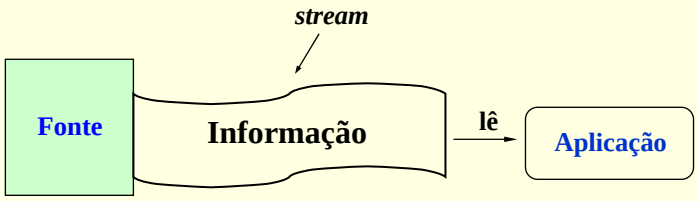


Streams de Entrada e Saída

- *Há várias fontes de onde se deseja ler ou destinos para onde se deseja gravar ou enviar dados*
  - *Arquivos*
  - *Conexões de rede*
  - *Console (teclado / tela)*
  - *Memória*
- *Há várias formas diferentes de ler/escrever dados*
  - *Seqüencialmente / aleatoriamente*
  - *Como bytes / como caracteres*
  - *Linha por linha / palavra por palavra, ...*
- *APIs Java para I/O oferecem objetos que abstraem fontes/destinos (nós) e fluxos de bytes e caracteres*

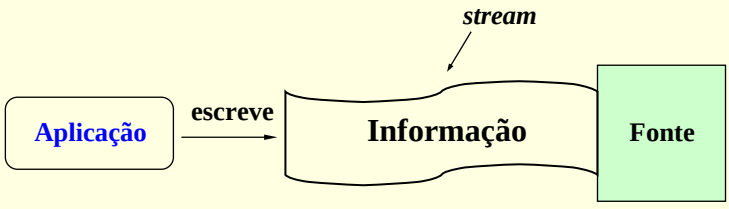
Stream de Entrada

- Para obter informações, uma aplicação abre um *stream* de uma fonte (arquivo, *socket*, memória) e lê sequencialmente



Stream de Saída

- Para enviar informações, uma aplicação abre um *stream* para um destino (arquivo, *socket*, memória) e escreve sequencialmente



Leitura e Escrita de *Streams*

- Independentemente da fonte/destino e do tipo de informações, os algoritmos para leitura e escrita são basicamente os mesmos

Leitura	Escrita
abre um *stream*	abre um *stream*
enquanto há informação	enquanto há informação
lê informação	escreve informação
fecha o *stream*	fecha o *stream*

Pacote **java.io**

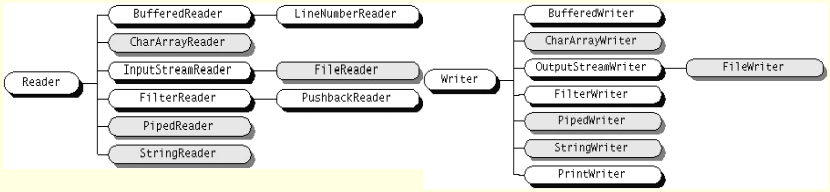
- Pacote java.io
  - O pacote padrão **java.io** define diversas classes e interfaces que permitem a entrada e saída de dados.
  - Esse pacote define dois pares básicos de classes abstratas: entrada e saída de bytes ou de caracteres:

    InputStream / ***OutputStream***
    Reader / ***Writer***

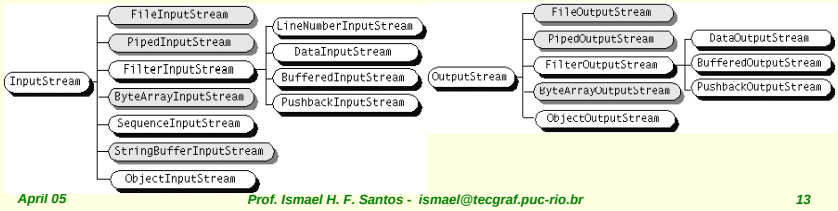
    } **classes abstratas**
  - Dessas classes derivam diversas outras que implementam as operações para algum tipo de mídia.
- Pacote java.nio (New I/O): a parti de j2SDK 1.4
  - Suporta mapeamento de memória e bloqueio de acesso

Streams de Caracteres e Bytes

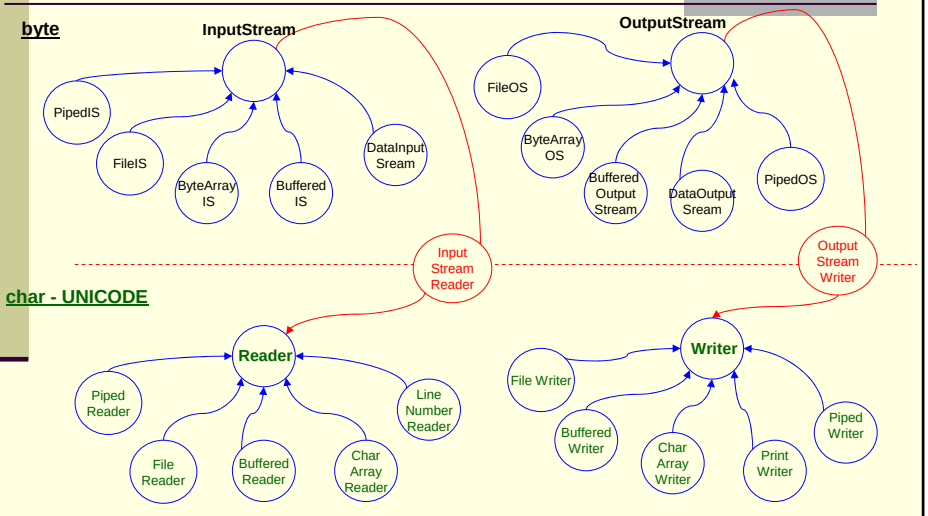
- Streams de Caracteres



- Streams de Bytes



Streams de Bytes e de Caracteres



Streams de *Bytes*

- As classes InputStream e OutputStream são as superclasses abstratas de todos os *streams* de bytes (dados binários)
  - InputStream define um método abstrato **read** para ler um byte de uma *stream*
  - OutputStream define um método abstrato **write** para escrever um byte em uma *stream*
- Subclasses provêem E/S especializada para cada tipo de fonte/destino

Leitura de Bytes

- Um canal específico pode estar conectado a um arquivo, uma conexão de rede, etc. Isso será definido pela classe concreta que nós utilizarmos para efetivamente ler bytes de algum repositório de dados.

InputStream

```
public abstract int read() throws IOException

public int read(byte[] buf) throws IOException
public int available() throws IOException

public boolean markSupported()
public synchronized void mark(int readlimit)
public synchronized void reset() throws
  IOException

public void close() throws IOException
```


Exemplo: **System.in**

- É um objeto do tipo InputStream

```
public static final InputStream in
```

- Esse *stream* já está aberto e pronto para prover dados à aplicação

```
int bytesProntos = System.in.available();
if (bytesProntos > 0){
  byte[] entrada = new byte[bytesProntos];
  System.in.read(entrada);
}
```

Escrita de Bytes

- OutputStream
  - De maneira análoga à leitura, a escrita de bytes é definida através da classe abstrata **OutputStream**. Essa classe modela um canal para o qual bytes podem ser escritos.
  - Novamente, esse canal pode estar enviando os bytes para um arquivo, uma conexão de rede, um array de bytes, etc.

```
public abstract void write(int b) throws
  IOException
public void write(byte b[]) throws IOException
public void flush() throws IOException
public void close() throws IOException
```


Exemplo: **System.out**

- É um objeto do tipo **PrintStream**, subclasse de **OutputStream**

```
public static final PrintStream out
```

- Esse tipo de *stream* fornece a seu "destino" representações de vários tipos de dados

```
public void print(float f)
public void print(String s)
public void println(String s)
```

IOException

- É uma extensão da classe **Exception**
- Sinaliza a ocorrência de uma falha ou interrupção em uma operação de E/S
- Algumas subclasses:
  - **EOFException**, **FileNotFoundException, InterruptedIOException**, **MalformedURLException**, **SocketException.**

Leitura de Arquivo e Arrays

- FileInputStream
  - Uma extensão da classe **InputStream** é a classe **FileInputStream** que lê os bytes de um arquivo.

```
public FileInputStream(String name) throws
  FileNotFoundException
```

- ByteArrayInputStream
  - Um array de bytes também pode ser uma fonte de dados. **ByteArrayInputStream** estende **InputStream** e implementa a leitura a partir de um array.

```
public ByteArrayInputStream(byte buf[])
```

Escrita em Arquivo e Arrays

- FileOutputStream
  - A classe **FileOutputStream** modela um stream de escrita em um arquivo.

```
public FileOutputStream(String name) throws IOException
public FileOutputStream(String name, boolean append)
                                        throws IOException
```

- ByteArrayOutputStream
  - Uma outra extensão de **OutputStream**, **ByteArrayOutputStream**, modela a escrita em um array.

```
public ByteArrayOutputStream(int size)
public byte[] toByteArray()
```


Encadeamento de Streams

- Encadeamento de Streams
  - Um uso bastante comum é o *encadeamento* de streams: podemos, por exemplo, fazer com que um stream de entrada alimente um outro stream de entrada.
  - Um exemplo de aplicação é a "bufferização" das operações de leitura e/ou escrita.
- BufferedInputStream
  - A classe **BufferedInputStream** recebe um stream de entrada e, a partir dele, faz uma leitura "bufferizada" dos dados: lê um bloco inteiro e o armazena, passando os bytes um a um para o usuário.

```
public BufferedInputStream(InputStream in)
```

Buffered Streams

- BufferedInputStream
  - Por *default*, os *streams* não são *bufferizados*
    - essa funcionalidade pode ser obtida adicionando-se uma “camada” sobre o *stream*
  - Subclasse de *BufferedInputStream*, provê a entrada de dados buferizada (a eficiência é aumentada pela leitura de grandes volumes de dados e o armazenamento destes dados em um buffer interno). Quando o dado é requisitado, ele é disponibilizado do buffer, ao invés de ser lido do disco, rede ou outro recurso lento.

Buffered Streams

- BufferedOutputStream
  - Subclasse de *BufferedOutputStream*, provê a buferização dos dados de saída (a eficiência aumenta devido ao armazenamento dos dados de saída e o envio desses dados para a saida somente quando o buffer enche ou quando o método *flush()* é chamado).
- SequenceInputStream
  - Provê um modo de concatenar os dados de 2 ou mais fluxos de dados de entrada.

Outros Streams

- FilterInputStream e FilterOutputStream
  - Implementam o padrão de projeto Decorator. São cncatenados em streams primitivos oferecendo métodos mais úteis com dados filtrados.
  - *FilterInputStream* provê os métodos necessários para filtrar os dados obtidos de um *InputStream*.
    - *DataInputStream: readInt(), readUTF(), readDouble()*
    - *BufferredInputStream: read() mais eficiente*
    - *ObjectOutputStream: readObject() lê objetos serializados*

Outros Streams

- *FilterOutputStream* provê os métodos necessários para filtrar os dados que serão escritos em um *OutputStream*. Os dois são utilizado para permitir operações de sort e filtragem dos dados.
  - *DataOutputStream: writeUTF(), writeInt(), etc.*
  - *BufferedOutputStream: write() mais eficiente*
  - *ObjectOutputStream: writeObject() serializa objetos*
  - *PrintStream: classe que implementa println()*
- PipedInputStream e PipedOutputStream
  - *PipedInputStream* lê bytes de um *PipedOutputStream*, e o *PipedOutputStream* escreve bytes em um *PipedInputStream*. Essas últimas classes trabalham junto para implementar um “pipe” para comunicações entre processos (threads).

Streams de Conversão

- Pontes entre *streams* de *bytes* e de caracteres

```
public InputStreamReader(InputStream i)
public InputStreamReader(InputStream i, String enc)
                  throws UnsupportedEncodingException
public OutputStreamWriter(OutputStream o)
public OutputStreamWriter(OutputStream o, String enc)
                  throws UnsupportedEncodingException
```

- Para maior eficiência, pode-se utilizar *streams* bufferizadas:

```
BufferedReader in = new BufferedReader(new
                    InputStreamReader(System.in));
```


Exemplo de Leitura

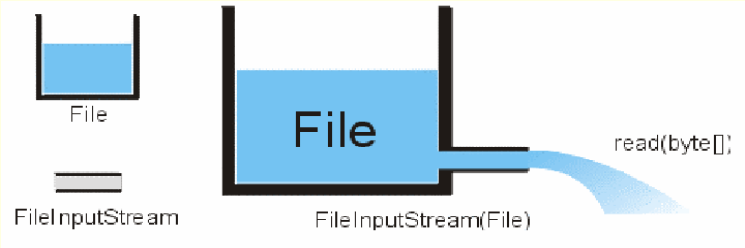
- Leitura de Arquivo

```
import java.io.*;
public class PrintFile {
  public static void main(String[] args) {
    try {
      InputStream fin = new FileInputStream(args[0]);
      InputStream in = new BufferedInputStream(fin);
      int b;
      while ((b = in.read()) != -1) {  System.out.print((char)b);   }
    } catch (IOException e) {
      System.out.println(e);
    }
  }
}
```

Exercícios – Questões 30

Leitura de um stream fonte arquivo

```
// objeto do tipo File
File tanque = new File("agua.txt");
// referência FileInputStream
// cano conectado no tanque
FileInputStream cano = new FileInputStream(tanque);
// lê um byte a partir do cano
byte octeto = cano.read();
```

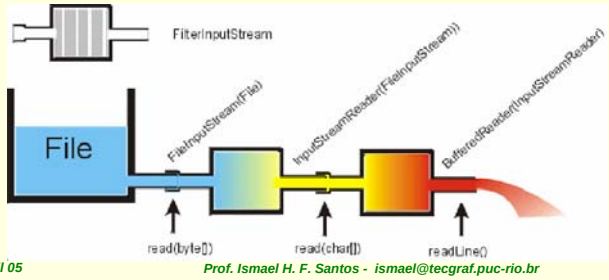


Usando filtro para ler char

```
// 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);

// lê um char a partir do filtro chf
char letra = chf.read();
```

Usando filtro para ler linha

```
// filtro chf conectado no cano
InputStreamReader chf = new nputStreamReader(cano);

// filtro br conectado no chf
BufferedReader br = new BufferedReader (chf);

// lê linha de texto a de br
String linha = br.readLine();
```



Streams de Caracteres

- As classes Reader e Writer são as superclasses abstratas de todos os *streams* de caracteres
- Subclasses provêem E/S especializada diferentes tipos de fonte/destino

Reader

```
public Reader()
public Reader(Object lock)
public int read() throws IOException
public int read(char[] buf) throws IOException
public long skip(long n) throws IOException
public boolean ready() throws IOException
public abstract void close() throws IOException
public void mark(int readlimit)
public void reset() throws IOException
public boolean markSupported()
```


Writer

```
public Writer()
public Writer(Object lock)
public void write(int c) throws IOException
public void write(char[] buf) throws IOException
public void write(String str) throws IOException
public abstract void flush() throws IOException
public abstract void close() throws IOException
```

Streams de Vetores

- **ByteArrayInputStream**
- **ByteArrayOutputStream**
- **CharArrayReader**
- **CharArrayWriter**

```
public ByteArrayInputStream(byte[] buf)
public ByteArrayOutputStream(int buf_size)
```


Streams de *Strings*

- **StringReader**
- **StringWriter**

```
public StringReader(String str)
public StringWriter(int buf_size)
```

Buffered Streams

- **BufferedInputStream**
- **BufferedOutputStream**
- **BufferedReader**
- **BufferedWriter**

```
public BufferedInputStream(InputStream in)
public BufferedInputStream(InputStream in, int size)
```


Streams de Conversão

- **InputStreamReader**
- **OutputStreamWriter**

```
public InputStreamReader(InputStream i)
public InputStreamReader(InputStream i, String enc)
  throws UnsupportedEncodingException

public OutputStreamWriter(OutputStream o)
public OutputStreamWriter(OutputStream o, String e)
  throws UnsupportedEncodingException
```

Arquivos

- Classe **File**
- Acesso via *streams*
  - **FileInputStream**
  - **FileOutputStream**
  - **FileReader**
  - **FileWriter**
- Acesso aleatório
  - **RandomAccessFile**

A classe File

- *Usada para representar o sistema de arquivos*
  - *É apenas uma abstração: 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.*
- *Alguns métodos*
  - *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*

File: exemplo de uso

```
File diretorio = new File("c:\tmp\cesto");
diretorio.mkdir(); // cria, se possível
File arquivo = new File(diretorio, "lixo.txt");
FileOutputStream out =
            new FileOutputStream(arquivo);
// se arquivo não existe, tenta criar
out.write( new byte[]{'l','i','x','o'} );

File subdir = new File(diretorio, "subdir");
subdir.mkdir();
String[] arquivos = diretorio.list();
for (int i = 0; arquivos.length; i++) {
   File filho = new File(diretorio, arquivos[i]);
   System.out.println(filho.getAbsolutePath());
}
if (arquivo.exists()) {
   arquivo.delete();
}
```

O bloco de código acima precisa tratar IOException

Manipulação de diretórios

- diretório atual da aplicação

```
String dirAtual = System.getProperty("user.dir");
```

- Deletando um diretório

```
boolean deletado = (new File("diretorio")).delete();
if ( !deletado ) {         // falhou...     }
```

- E se o diretorio não estiver vazio ?

Manipulação de diretórios

```
public static boolean deleteDir(File dir) {
  if (dir.isDirectory()) {
    String[] subDir = dir.list();
    for (int i=0; i< subDir.length; i++) {
      boolean deletado = deleteDir( new File(dir,
                                        subDir[i]) );
      if( !deletado ) {
          return false;
      }
    }
  }// O diretorio agora está vazio, então removemos !
  return dir.delete();
 }
```


Copiando diretorio -> outro diretorio

```
public void copiaDir(File srcDir, File dstDir) throws
                                                IOException {
 if (srcDir.isDirectory()) {
   if (!dstDir.exists()) { dstDir.mkdir(); }
   String[] subDirs = srcDir.list();
   for (int i=0; i < subDirs.length; i++) {
     copiaDir(new File(srcDir, subDirs[i]),
                            new File(dstDir, subDirs[i]));
   }
 } else {
   // Copiando arquivos usando FileChannel
   FileChannel src= new FileInputStream(src).getChannel();
   FileChannel dst= new FileOutputStream(dst).getChannel();
   // Copia o conteúdo e fecha os FileChannels
   dst.transferFrom(src, 0, src.size());
   src.close(); dst.close();
 }
}
```

Classe **FileInputStream**

- Especialização de **InputStream** para leitura de arquivos

```
public FileInputStream(String name) throws
                                FileNotFoundException
public FileInputStream(File file) throws
                                FileNotFoundException
```

- Usando *stream* bufferizada:

```
BufferedInputStream in = new BufferedInputStream(
                                new FileInputStream(
                                      "arquivo.dat"));
```


Classe **FileOutputStream**

- Especialização de **OutputStream** para escrita em arquivos

```
public FileOutputStream(String name) throws
                        FileNotFoundException

public FileOutputStream(String name,
      boolean append) throws FileNotFoundException

public FileOutputStream(File file) throws
                              FileNotFoundException
```

Leitura e Gravação de Arquivo

- *Trecho de programa que copia um arquivo**

```
String nomeFonte = args[0];
String nomeDestino = args[1];
File fonte = new File(nomeFonte);
File destino = new File(nomeDestino);
if (fonte.exists() && !destino.exists()) {
   FileInputStream in = new FileInputStream(fonte);
   FileOutputStream out = new FileOutputStream(destino);
   byte[] buffer = new byte[8192];
   int length = in.read(buffer);
   while ( length != -1) {
      out.write(buffer, 0, length);
      in.read(buffer);
   }
   in.close();
   out.flush();
   out.close();
}
```

-1 sinaliza EOF

Grava apenas os bytes lidos (e não o buffer inteiro)

* *Método e blocos try-catch (obrigatórios) foram omitidos para maior clareza.*

Classe **FileReader**

- É uma subclasse de **InputStreamReader**

```
public FileReader(String name) throws
                    FileNotFoundException
public FileReader(File file) throws
                FileNotFoundException
```

- Usando *stream* bufferizada:

```
BufferedReader in = new BufferedReader( new
                        FileReader("arquivo.dat"));
```

Exemplo de Leitura de Arquivo

```
try {
  Reader r = new FileReader("test.txt");
  int c;
  while( (c=r.read()) != -1 ) {
    System.out.println("Li caracter "+(char)c);
  }
} catch( FileNotFoundException e ) {
  System.out.println("test.txt não existe");
} catch( IOException e ) {
  System.out.println("Erro de leitura");
} finaly {
  if( r != null )
   r.close();
}
```


Exemplo Leitura de Arquivo Bufferizada

```
try {
  BufferedReader r = new BufferedReader( new
                          FileReader("test.txt") );
  String linha;
  while( (linha=r.readLine()) != null ) {
    System.out.println(“Li linha:” + linha);
  }
} catch( FileNotFoundException e ) {
  System.out.println("test.txt não existe");
} catch( IOException e ) {
  System.out.println("Erro de leitura");
} finaly {
  if( r != null )
   r.close();
}
```

Classe **FileWriter**

- É uma subclasse de **OutputStreamWriter**

```
public FileWriter(String name) throws
                    IOException
public FileWriter(String name, boolean append)
                    throws IOException
public FileWriter(File file) throws
                    IOException
```


Exemplo de leitura e escrita

```
import java.io.*;
public class Copy {
 public static void main(String[] args) throws IOException {
   File fonte = new File(args[0] != null ?
  args[0]:"filein.txt");
   File dest  = new File(args[1] != null ?
  args[1]:"fileout.txt");
   if( fonte.exists() && ! dest.exists() ) {
     Reader in  = new FileReader(fonte);
     Writer out = new FileWriter(dest);
     int c;
     while ((c = in.read()) != -1)
       out.write(c);
     in.close();
     out.flush(); out.close();
   }
 }
}
```

Exemplo de Escrita em Arquivo no final

```
try {
   BufferedWriter w = new BufferedWriter( new
                  FileWriter("test.txt“, true) );
   w.write(“Este é um teste de append !!!”);
} catch( FileNotFoundException e ) {
  System.out.println("test.txt não existe");
} catch( IOException e ) {
  System.out.println("Erro de leitura");
} finaly {
  if( w != null )
   w.close();
}
```


Leitura e gravação de texto com buffer

- A maneira mais eficiente de ler um arquivo de texto é usar FileReader decorado por um BufferedReader. Para gravar, use um PrintWriter decorando o FileWriter

```
File arq = new File("arq.txt");
BufferedReader in = new BufferedReader(new
                                FileReader(“arq.txt”));
StringBuffer sb = new StringBuffer(arq.length());
String linha;
while( (linha=in.readLine()) != null ) {
  sb.append(linha).append('\n');
}
in.close();
String txtLido = sb.toString();
// (...)
PrintWriter out=new PrintWriter(new FileWriter("ARQ.TXT"));
out.print(txtLido.toUpperCase());out.flush();out.close();
```

Leitura da entrada padrão e memória

- A entrada padrão (System.in) é representada por um objeto do tipo InputStream. O exemplo lê uma linha de texto digitado na entrada padrão e grava em uma String. Em seguida lê a String seqüencialmente e imprime uma palavra por linha

```
BufferedReader stdin = new BufferedReader(new
                       InputStreamReader(System.in));
System.out.print("Digite uma linha:");
String linha = stdin.readLine());
StringReader rawIn = new StringReader(linha);
int c;
while((c=rawIn.read()) != -1)
  if ( c== ' ') System.out.println();
  else System.out.print((char)c);
}
```


Streams de Dados

- Definidos por interfaces
  - **DataInput**
  - **DataOutput**
- Permitem escrita e leitura de tipos básicos
- Implementados por
  - **DataInputStream**
  - **DataOutputStream**
  - **RandomAccessFile**

DataInput

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


DataOutput

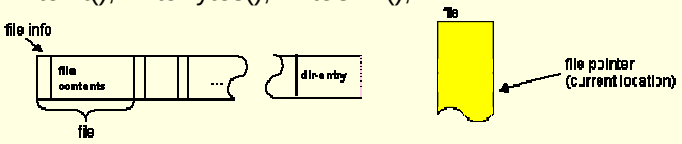
```
public abstract void write(byte b[]) throws IOException
public abstract void writeBoolean(boolean v) throws
   IOException
public abstract void writeByte(int v) throws
   IOException
public abstract void writeChar(int v) throws
   IOException
public abstract void writeInt(int v) throws IOException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

Exemplo de *stream* de dados

```
try {
  FileInputStream fin = new FileInputStream("arquivo.dat");
  DataInputStream din = new DataInputStream(fin);
  int num_valores = din.readInt();
  double[] valores = new double[num_valores];
  for (int i = 0 ; i < num_valores ; i++)
     valores[i] = din.readDouble();
} catch (EOFException e) {
   …
} catch (FileNotFoundException e) {
   …
} catch (IOException e) {
   …
}
```


Classe **RandomAccessFile**

- Permite a leitura e escrita em um arquivo de acesso randômico. Implementa as interfaces **DataInput** e **DataOutput.** Mistura de File com streams: não deve ser usado com outras classes (streams) do java.io. Métodos (DataOutput e DataInput) tratam da leitura e escrita de Strings e tipos primitivos
  - void seek(long)
  - readInt(), readBytes(), readUTF(), ...
  - writeInt(), writeBytes(), writeUTF(), ...



- 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()**

Classe **RandomAccessFile**

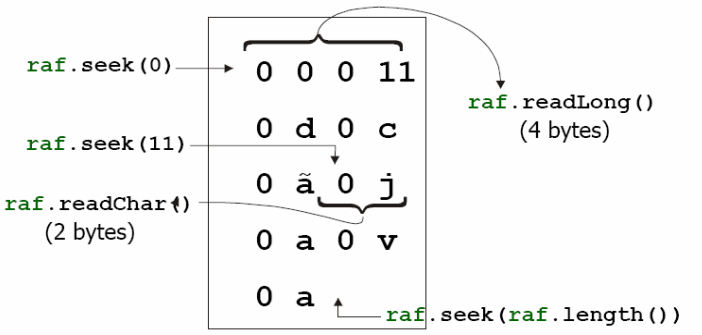
- *Classe "alienígena": não faz parte da hierarquia de fluxos de dados do java.io*
  - *Implementa interfaces DataOutput e DataInput*
  - *Mistura de File com streams: não deve ser usado com outras classes (streams) do java.io*
- *Oferece acesso aleatório a um arquivo através de um ponteiro*
- *Métodos (DataOutput e DataInput) tratam da leitura e escrita de Strings e tipos primitivos*
  - *void seek(long)*
  - *readInt(), readBytes(), readUTF(), ...*
  - *writeInt(), writeBytes(), writeUTF(), ...*

Métodos de **RandomAccessFile**

```
public RandomAccessFile(String name, String mode)
                    throws FileNotFoundException
public RandomAccessFile(File file, String mode)
                    throws FileNotFoundException
public long getFilePointer() throws IOException
public void seek(long pos) throws IOException
public long length() throws IOException
```

Usando RandomAccessFile

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



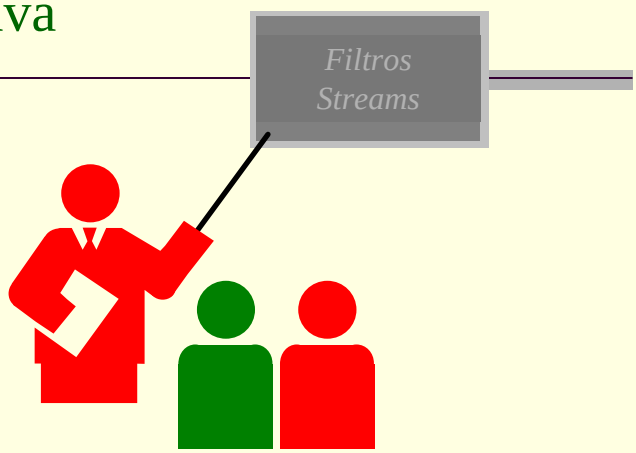
Exceções

- *A maior parte das operações de E/S provoca exceções que correspondem ou são subclasses de IOException*
  - *EOFException*
  - *FileNotFoundException*
  - *StreamCorruptedException*
- *Para executar operações de E/S é preciso, portanto, ou capturar IOException ou repassar a exceção através de declarações throws nos métodos causadores*
- *Nos exemplos mostrados o tratamento de exceções foi omitido. Tipicamente, as instruções close() ocorrem em um bloco try-catch dentro de um bloco finally*

```
try { ... } finally {
   try { stream.close(); } catch (IOException e) {}
}
```

Não adianta saber se o fechamendo to stream falhou

POO-Java



Filtros de *Streams*

- Filtros são acoplados a *streams*
- Permitem manusear os dados “em trânsito”
- Filtros básicos (transparentes)
  - **FilterInputStream**
  - **FilterOutputStream**
  - **FilterReader**
  - **FilterWriter**

Parsing de *Streams*

- Permite a análise léxica de um texto
  - "quebra" o conteúdo de um *stream* em *tokens*, que podem ser lidos um a um
- Padrões configuráveis para:
  - separadores
  - identificadores
  - números
  - textos
  - comentários

StreamTokenizer

```
public StreamTokenizer(Reader r)
public void whitespaceChars(int low, int hi)
public void wordChars(int low, int hi)
public void quoteChar(int ch)
public void commentChar(int ch)
public void ordinaryChar(int ch)
public int nextToken() throws IOException
public void pushBack()
public int lineno()
```

Uniform Resource Locator

- A classe **URL** modela URLs, permitindo a obtenção de informações e conteúdo de páginas na Web
- Essa classe é parte do pacote **java.net**

URL: Construtores

```
public URL(String spec)
   throws MalformedURLException

public URL(String protocol, String host,
           String file)
   throws MalformedURLException

public URL(String protocol, String host,
           int port, String file)
   throws MalformedURLException
```

URL: Métodos de Consulta e Acesso

```
public String getProtocol()
public String getHost()
public int getPort()
public String getFile()

public String getUserInfo()
public String getPath()
public String getQuery()

public final InputStream openStream() throws IOException
public URLConnection openConnection() throws IOException
```


Exemplo de Uso de **URL**

```
import java.io.*;
import java.net.*;

public class PegaPagina {
  public static void main(String[] args) throws Exception {
    if (args.length == 0) {
      System.err.println("Forneça o endereço da página.");
      return;
    }
    URL url = new URL(args[0]);
    InputStream is = url.openStream();
    Reader r = new InputStreamReader(is);
    BufferedReader br = new BufferedReader(r);
    String l;
    while ((l = br.readLine()) != null) {
      System.out.println(l);
    }
  }
}
```

Lendo de Arquivos **.jar**

- A classe **Class** provê métodos para obter um recurso como **URL** ou **InputStream**. Quem efetivamente obtém o recurso é o *class loader* da classe em questão, que sabe de onde ela foi obtida

```
public URL getResource(String name)
public InputStream getResourceAsStream(String name)
```


Exemplos

- Exemplo do Applet

```
getAudioClip(getClass().getResource("spacemusic.au"));
```

- Outro exemplo

```
InputStream is =
   getClass().getResourceAsStream(“arquivo.dat"));
```

POO-Java

Serialização Objetos

Serialização

- Java permite a gravação direta de objetos em disco ou seu envio através da rede
  - Para isto, o objeto deve declarar implementar java.io Serializable
- Um objeto Serializable poderá, então
  - Ser gravado em qualquer stream usando o método writeObject() de ObjectOutputStream
  - Ser recuperado de qualquer stream usando o método readObject() de ObjectInputStream
- Um objeto serializado é um grafo que inclui dados da classe e todas as suas dependências
  - 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)

Streams de Objetos – Serialização

- Java permite a gravação direta de objetos em disco ou seu envio através da rede
  - Para isto, o objeto deve declarar implementar java.io.Serializable
- Um objeto Serializable poderá então
  - Ser gravado em qualquer stream usando o método writeObject() de ObjectOutputStream
  - Ser recuperado de qualquer stream usando o método readObject() de ObjectInputStream
- As interfaces ObjectInput e ObjectOutput estendem DataInput e DataOutput para incluir objetos, arrays e Strings e são implementadas por ObjectInputStream e ObjectOutputStream

Serialização

- Um objeto serializado é um grafo que inclui dados da classe e todas as suas dependências
  - 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)
- Um **ObjectInputStream** "deserializa" dados e objetos anteriormente escritos através de um **ObjectOutputStream**.

Utilização de *streams* de Objetos

- Cenários de utilização:
  - persistência de objetos, quando esses *streams* são usados em conjunto com **FileInputStream** e **FileOutputStream**
  - transferência de objetos entre *hosts via sockets,* utilizando Remote Method Invocation (RMI)

```
public abstract Object readObject()
   throws ClassNotFoundException, IOException
public abstract void writeObject(Object obj) throws
   IOException
```


ObjectInput / ObjectOutput

Exemplo: Salvando data em arquivo

```
FileOutputStream out= new FileOutputStream("theTime");
ObjectOutputStream s = new ObjectOutputStream(out);
s.writeObject("Today"); s.writeObject(new Date()); s.flush();
  out.close();
```

Exemplo: Recuperando data do arquivo

```
FileInputStream in = new FileInputStream("theTime");
ObjectInputStream s = new ObjectInputStream(in);
String today=(String)s.readObject(); Date
  date=(Date)s.readObject();in.close();
```

Interface **Serializable**

- Somente objetos cujas classes implementem a "*marker interface*" **Serializable** podem ser serializados

```
package java.io;
public interface Serializable {
  // there's nothing in here!
};
```

- Essa interface não tem métodos, mas uma classe "Serializable" pode definir métodos **readObject** e **writeObject** para fazer validações no estado do objeto

Implementação

- Customizando a Serialização

```
public class ObjSerializado iplements Serializable {
 ......
 private void writeObject(ObjectOutputStream s) throws IOException {
   s.defaultWriteObject();
   // customized serialization code
 }
 private void readObject(ObjectInputStream s) throws IOException {
   s.defaultReadObject();
   // customized deserialization code ...
   // followed by code to update the object, if necessary
 }
}
```

- Os métodos writeObject e readObject são responsáveis pela serialização somente da classe corrente. Qualquer serialização requerida pelas superclasses é tratada automaticamente pelo Java usando Reflexão.

Usando Serialização

```
ObjectOutputStream out = new ObjectOutputStream(
    new FileOutputStream(armario)
);
Arco a = new Arco();
Flecha f = new Flecha();
// grava objeto Arco em armario
out.writeObject(a);
// grava objeto flecha em armario
out.writeObject(f);
```

Gravação de objetos

Leitura de objetos na mesma ordem

```
ObjectInputStream in = new ObjectInputStream(
    new FileInputStream(armario)
);
// recupera os dois objetos
// método retorna Object (requer cast)
Arco primeiro = (Arco)in.readObject();
Flecha segundo = (Flecha)in.readObject();
```


Exemplo 1 – Serialização

```
import java.io.*;
public class Funcionario implements Serializable {
  ……
  private void readObject(ObjectInputStream is)
       throws ClassNotFoundException, IOException
  {
    is.defaultReadObject();
     if (!isValid())
        throw new IOException("Invalid Object");
  }
  private boolean isValid() {
    ……
  }
}
```

Exemplo 2 – Serialização

- Funcionario.java

```
import java.io.*;
public class Funcionario implements Serializable {
  String nome;
  String cargo;
  int salario;
  Funcionario gerente;
  public Funcionario (String nome, String cargo,
                      int salario, Funcionario gerente)
  {
   this.nome = nome;
   this.cargo = cargo;
   this.salario = salario;
   this.gerente = gerente;
  }
}
```


Exemplo 2 – Serialização

- Serializador.java

```
import java.io.*;
import Funcionario;
public class Serializador {
  public static void main (String args[]) throws
                                          IOException {
   Funcionario f = new Funcionario ("João da Silva",
                "Desenvolvedor Java", 17500, null);
   FileOutputStream s = new FileOutputStream ("tmp");
   ObjectOutputStream oos = new ObjectOutputStream
  (s);
   oos.writeObject (f);
   oos.flush();
  }
}
```

Exemplo 2 – Serialização

- Desserializador.java

```
import java.io.*;
import Funcionario;
public class Desserializador {
  public static void main (String args[]) throws
  Exception {
   FileInputStream s = new FileInputStream ("tmp");
   ObjectInputStream ois = new ObjectInputStream (s);
   Funcionario f = (Funcionario) ois.readObject();
   System.out.println (f.nome+" "+f.cargo+"
  "+f.salario);
  }
}
```


Exemplo 2 – Serialização

- Grafo.java

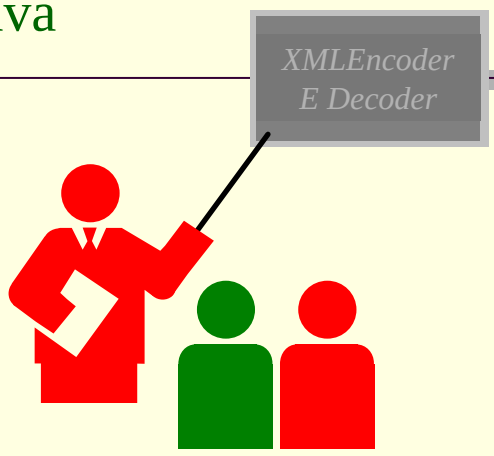
```
// ...
Funcionario g = new Funcionario ("Manoel Joaquim",
                "Gerente de Projeto", 31500, null);
Funcionario f = new Funcionario ("João da Silva",
                "Programador Java", 17500, g);
FileOutputStream s = new FileOutputStream ("tmp");
ObjectOutputStream oos = new ObjectOutputStream (s);
oos.writeObject (f);
// ...
Funcionario x = (Funcionario) ois.readObject();
System.out.println (x.gerente.nome);
```

Interface **Externalizable**

- Para um controle explicito do processo de serialização a classe deve implementar a *interface* **Externalizable**

```
package java.io;
public interface Externalizable extends Serializable {
    public void writeExternal(ObjectOutput out)  throws IOException;
    public void readExternal(ObjectInput in)       throws IOException,
                                                    java.lang.ClassNotFoundException;
}
```


POO-Java



XMLEncoder / XMLDecoder

- XMLEncoder

```
XMLEncoder e = new XMLEncoder( new BufferedOutputStream(
  new FileOutputStream("Test.xml")));
 e.writeObject(new JButton("Hello, world"));e.close();
```

- XMLDecoder

```
try {
 XMLDecoder d = new XMLDecoder( new BufferedInputStream(
  new FileInputStream("Test.xml")));
   d.readObject(); d.close();
} catch (IOException e) {
    ...handle the exception...
}
```

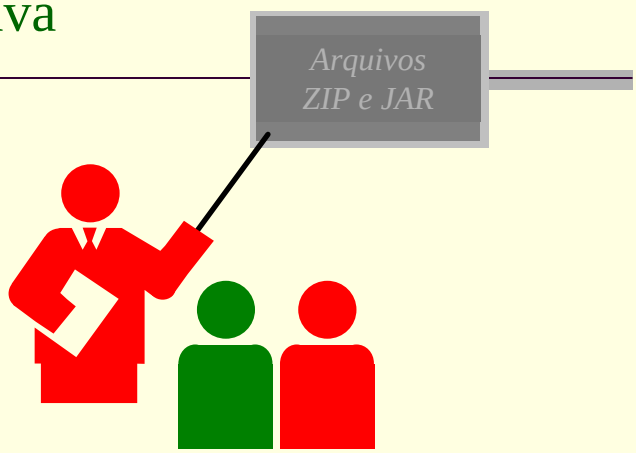

XMLEncoder / XMLDecoder

- Arquivo xml

```
<?xml version="1.0" encoding="UTF-8" ?>
<java version="1.4.0" class="java.beans.XMLDecoder">
  ...objects go here...
</java>
```

- Referencias
  - http://java.sun.com/products/jfc/tsc/articles/persistence3/
  - http://java.sun.com/products/jfc/tsc/articles/persistence4/

POO-Java



ZIP e JAR

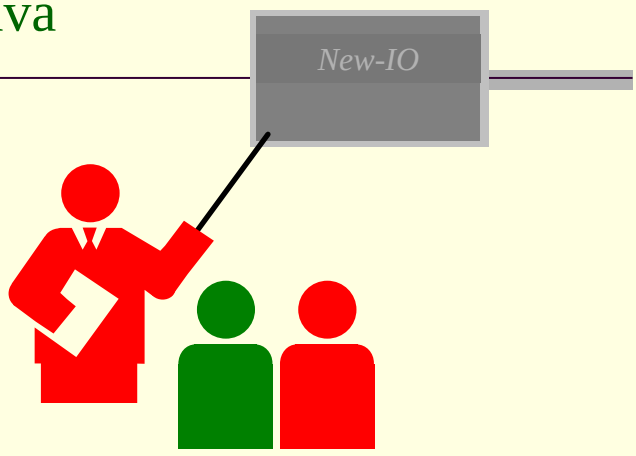
- Os pacotes java.util.zip e java.util.jar permitem comprimir dados e colecionar arquivos mantendo intactas as estruturas e diretórios. Vantagens:
  - Maior eficiência de E/S e menor espaço em disco
  - Menos arquivos para transferir pela rede (também maior eficiência de E/S)
- Use classes de ZIP e JAR para coleções de arquivos
  - ZipEntry, ZipFile, ZipInputStream, etc.
- Use streams GZIP para arquivos individuais e para reduzir tamanho de dados enviados pela rede

Exemplo 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
- 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();
```


POO-Java



Novidades java.nio

- *Novidade no J2SDK 1.4*
- *Permite ler e gravar arquivos, mapeando memória e bloqueando acesso (afeta performance)*
  - *Mapeamento permite abrir o arquivo como se fosse um vetor, usando a classe java.nio.ByteBuffer. Ideal para ler arquivos consistindo de registros de tamanho fixo.*
  - *É preciso importar java.nio.* e java.nio.channels.*;*
- *Exemplo: ler registro de arquivo de registros fixos*

```
FileInputStream stream = new FileInputStream("a.txt");
FileChannel in = stream.getChannel();
int len = (int) in.size();
ByteBuffer map = in.map(FileChannel.MapMode.READ_ONLY, 0, len);
final int TAM = 80; // tamanho de cada registro: 80 bytes
byte[] registro = new byte[TAM]; //array p/ guardar 1 registro
map.position( 5 * TAM ); // posiciona antes do 5o. registro
map.get( registro ); // preenche array com dados encontrados
```