

# Java

## Sistema de Entrada e Saída (I/O)

Programação III  
José Luis Oliveira; Carlos Costa

# Operações de entrada/saída

## Entrada

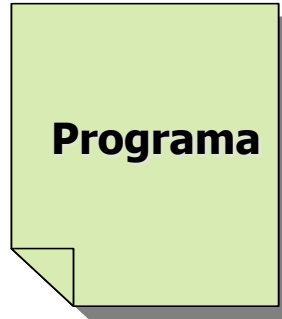
teclado



*leitura*



**Programa**



*escrita*



## Saída

monitor



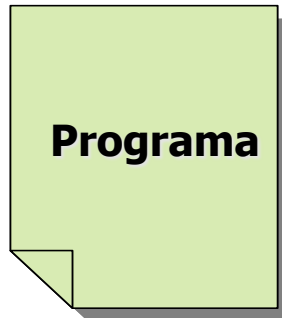
ficheiro



*leitura*



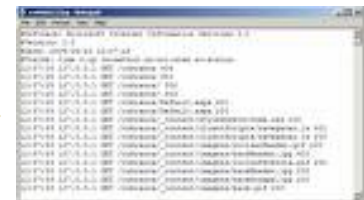
**Programa**



*escrita*



ficheiro

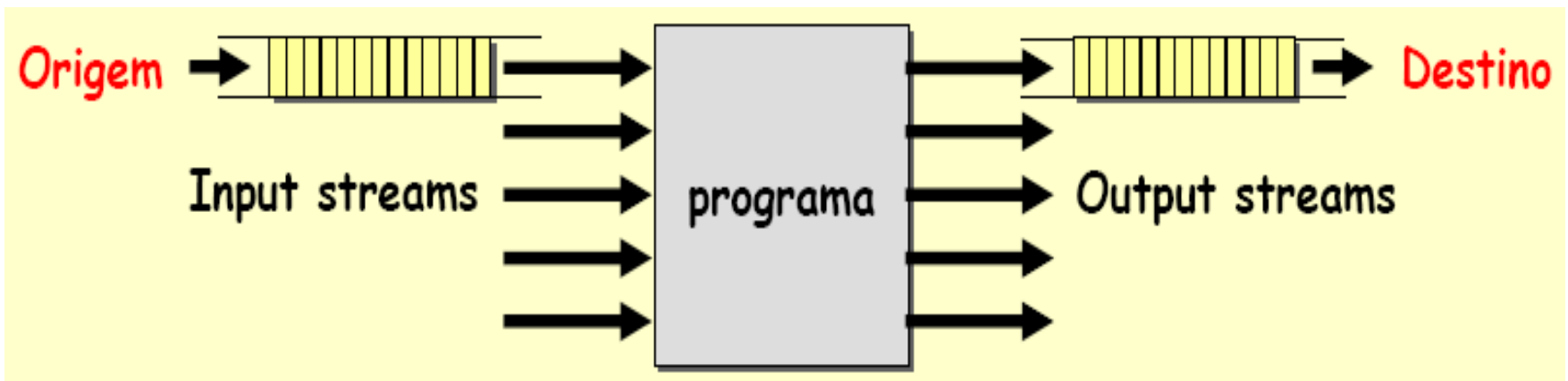


# Introdução

- Sem capacidade de interagir com o "resto do mundo", o nosso programa torna-se inútil
  - Esta interacção designa-se “input/output” (I/O)
- Problema → Complexidade
  - Diferentes e complexos dispositivos de I/O (ficheiros, consolas, canais de comunicação, ...)
  - Diferentes formatos de acesso (sequencial, aleatório, binário, caracteres, linha, palavras, ...)
- Necessidade → Abstracção
  - Libertar o programador da necessidade de lidar com as especificidade e complexidade de cada I/O
- Em Java, a abstracção I/O chama-se “Streams”

# I/O Streams

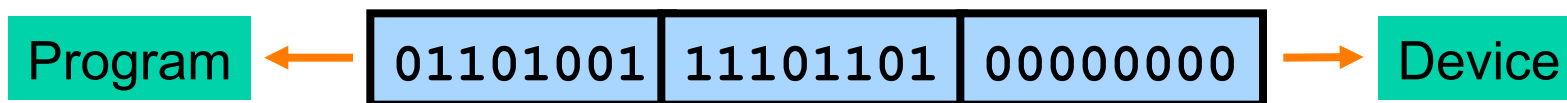
- O que são Streams?
  - um fluxo de dados que pode entrar ou sair de um programa.



# Tipos de Streams

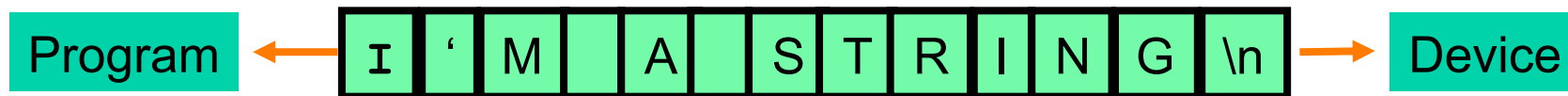
- Byte Streams

- binárias (machine-formatted)
- dados transferidos sem serem alterados de forma alguma
- não são interpretados
- não são feitos juízos sobre o seu valor



- Character Streams

- Os dados estão na forma de caracteres (human-readable data)
- interpretados e transformados de acordo com formatos de representação de texto



# Streams

- Os streams são objectos em Java. Temos 4 classes abstractas para lidar com I/O:

- **InputStream**: byte-input
- **OutputStream**: byte-output
- **Reader**: text-input
- **Writer**: text-output

**Classes  
Abstractas**

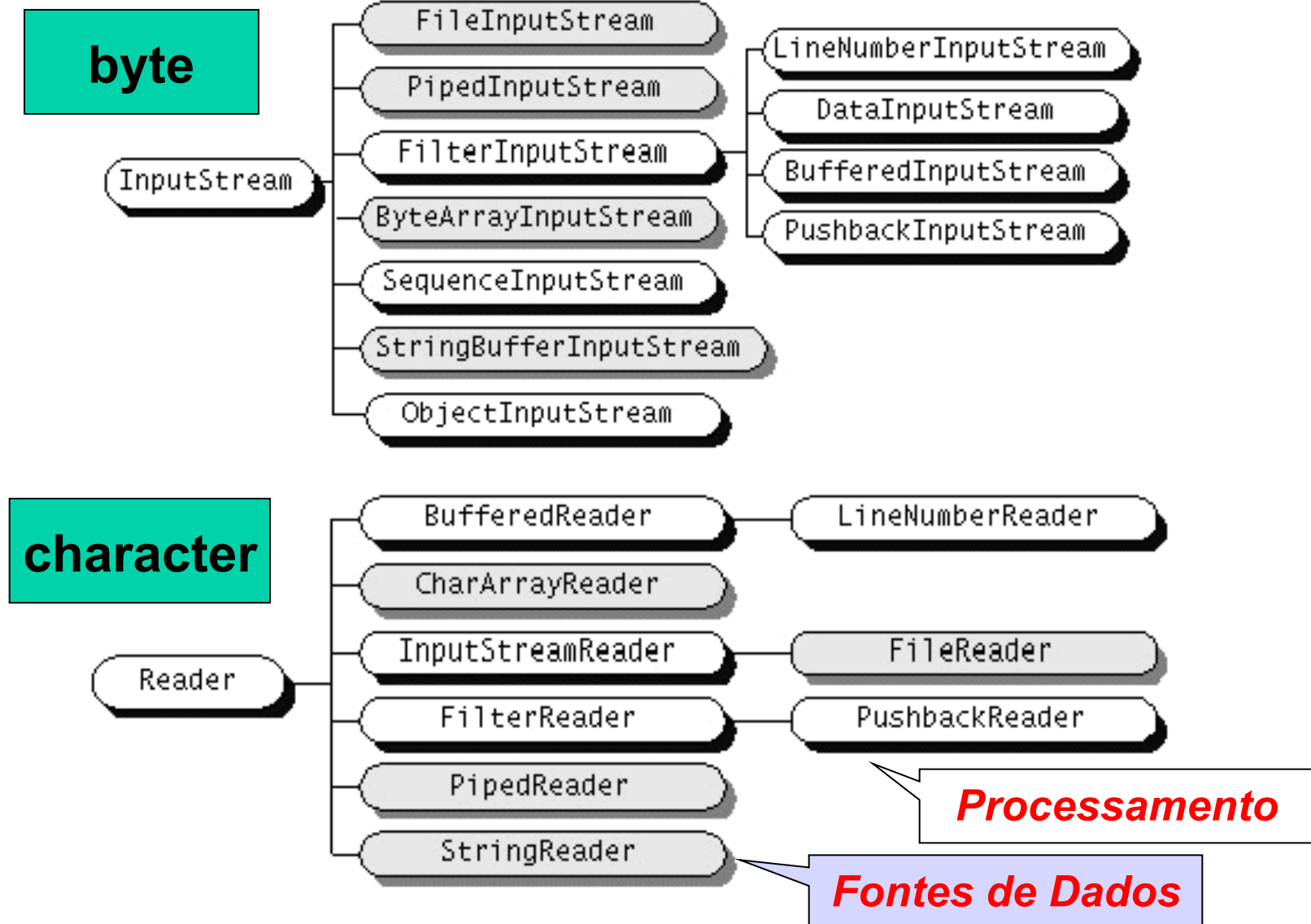
- Todas as classes de I/O são derivadas destas

- Entrada - **InputStream (byte);**      **Reader (char)**
- Saída - **OutputStream (byte);**      **Writer (char)**



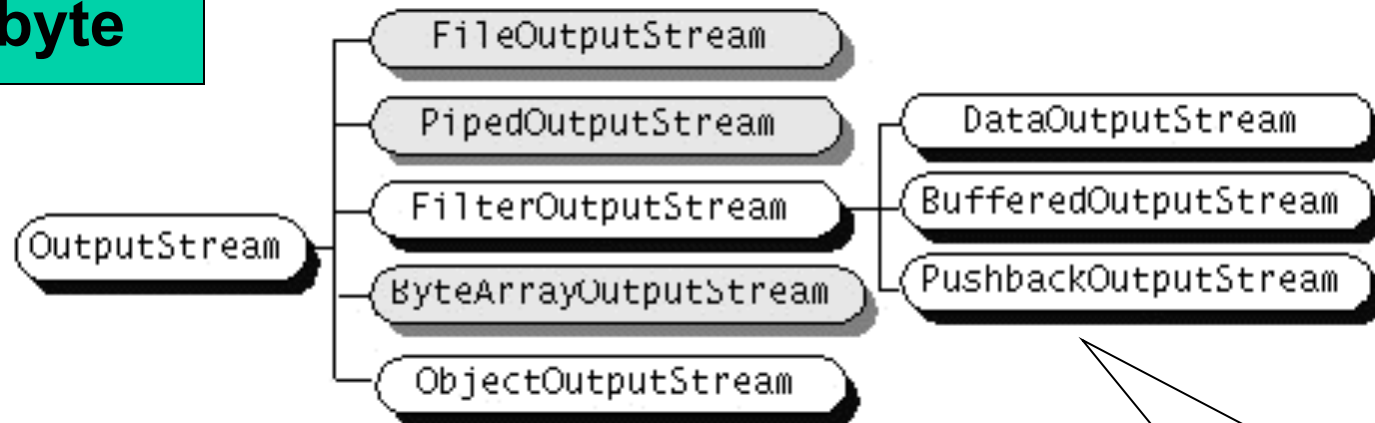
- Estas classes estão incluídas no package java.io

# Streams de Entrada



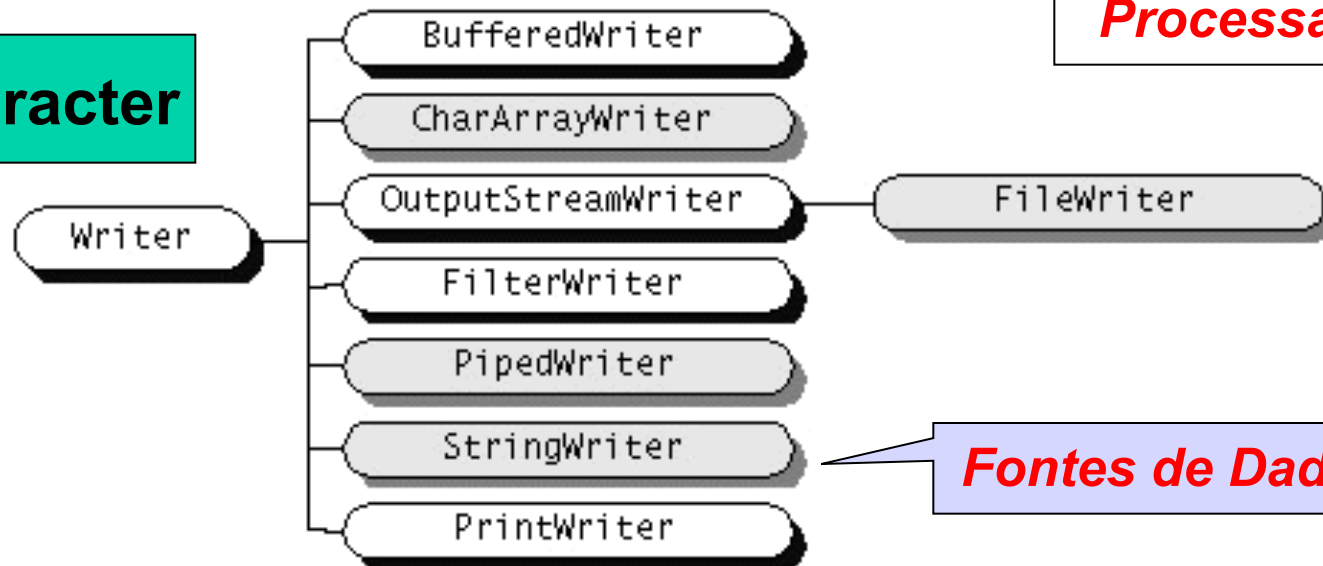
# Streams de Saída

**byte**



**Processamento**

**character**



**Fontes de Dados**



# InputStream/Reader

- **Reader** e **InputStream** têm interfaces semelhantes mas tipos de dados diferentes
- **Reader**
  - `int read()`
  - `int read(char cbuf[])`
  - `int read(char cbuf[], int offset, int length)`
- **InputStream**
  - `int read()`
  - `int read(byte cbuf[])`
  - `int read(byte cbuf[], int offset, int length)`

# OutputStream/Writer

- **Writer** e **OutputStream** têm interfaces semelhantes mas tipos de dados diferentes
- **Writer**
  - `int write()`
  - `int write(char cbuf[])`
  - `int write(char cbuf[], int offset, int length)`
- **OutputStream**
  - `int write()`
  - `int write(byte cbuf[])`
  - `int write(byte cbuf[], int offset, int length)`

# Standard I/O

- **System.in** é do tipo `InputStream`  
`byte[] b = new byte[10];`  
`InputStream stdin = System.in;`  
`stdin.read(b);`
- **System.out** é do tipo `PrintStream` (sub-tipo de `OutputStream`)  
`OutputStream stdout = System.out;`  
`stdout.write(104); // ASCII 'h'`  
`stdout.flush();`

Field Summary		java.lang.System
static <a href="#">PrintStream</a>	<a href="#">err</a>	The "standard" error output stream.
static <a href="#">InputStream</a>	<a href="#">in</a>	The "standard" input stream.
static <a href="#">PrintStream</a>	<a href="#">out</a>	The "standard" output stream.

# Utilização de *Streams*

<b>Sink Type</b> (Fontes de Dados)	<b>Character Streams</b>	<b>Byte Streams</b>
Memory	CharArrayReader, CharArrayWriter	ByteArrayInputStream, ByteArrayOutputStream
	StringReader, StringWriter	StringBufferInputStream
Pipe	PipedReader, PipedWriter	PipedInputStream, PipedOutputStream
File	FileReader, FileWriter	FileInputStream, FileOutputStream

# Classes de processamento

Process	CharacterStreams	Byte Streams
Buffering	BufferedReader, BufferedWriter	BufferedInputStream, BufferedOutputStream
Filtering	FilterReader, FilterWriter	FilterInputStream, FilterOutputStream
Converting between Bytes and Characters	InputStreamReader, OutputStreamWriter	
Concatenation		SequenceInputStream
Object Serialization		ObjectInputStream, ObjectOutputStream
Data Conversion		DataInputStream, DataOutputStream
Counting	LineNumberReader	LineNumberInputStream
Peeking Ahead	PushbackReader	PushbackInputStream
Printing	PrintWriter	PrintStream

# Ficheiros

- Classes principais:

**FileReader**

**FileWriter**

**FileInputStream**

**FileOutputStream**

**File**

**RandomAccessFile**

**Scanner (java 5)**

- Java 7

**Path**

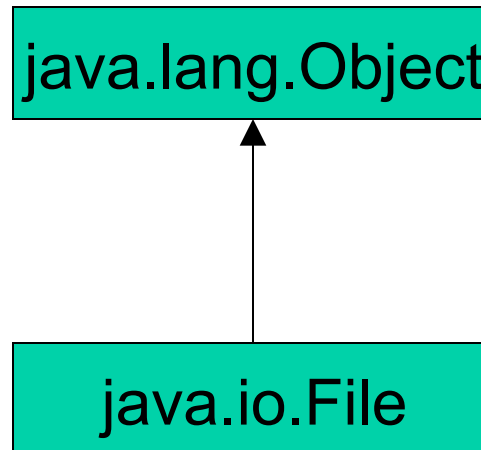
**Paths**

**Files**

**SeekableByteChannel**

# File (Path no java7)

- A classe File representa quer um nome de um **ficheiro** quer o conjunto de ficheiros num **directório**
- Fornece informações e operações úteis sobre ficheiros e directórios, mas não lê ou escreve em arquivos



# Exemplo - Criar Directórios e Ficheiros

```
import java.io.*;
public class FileTest {
    public static void main(String[] args)
        throws IOException {
        File directorio = new File("c:/tmp/newdir");
        directorio.mkdirs(); // cria uma árvore

        File arquivo = new File(directorio, "lixo.txt");
        FileOutputStream out =
            new FileOutputStream(arquivo);
        // criar ficheiro
        out.write(new byte[] { 'l', 'i', 'x', 'o' });
        out.close();

        File subdir = new File(directorio, "subdir");
        subdir.mkdir();
        // cria um subdirectório
    }
```

A partir de Java 7 existem outros métodos  
Files.createFile(..  
Files.createDirectory(..



# Exemplo - Listar um Directório

```
import java.io.*;

public class DirList {
    public static void main(String[] args)
        throws IOException {
        File directorio = new File("c:/tmp/newdir");
        String[] arquivos = directorio.list();
        for (int i = 0; i < arquivos.length; i++) {
            File filho = new File(directorio, arquivos[i]);
            System.out.println(filho.getAbsolutePath());
        }
    }
}
```

A partir de Java 7..

```
Path dir = ...
try (DirectoryStream<Path> stream = Files.newDirectoryStream(dir)) {
    for (Path entry: stream) {        ...    }
}
```

# Exemplo - Copiar um Ficheiro Texto

## Character Streams

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

```
public class Copy {
```

```
    public static void main(String[] args)
```

```
        throws IOException {
```

```
        File inputFile = new File("input.txt");
```

```
        File outputFile = new File("output.txt");
```

} **Create  
File Objects**

```
        FileReader in = new FileReader(inputFile);
```

```
        FileWriter out = new FileWriter(outputFile);
```

} **Create  
File Streams**

```
        int c;
```

```
        while ((c = in.read()) != -1)
```

**// Read from Stream**

```
            out.write(c);
```

**// Write to Stream**

```
        in.close();
```

```
        out.close();
```

} **Close the Streams**

```
    }
```

```
}
```

A partir de Java 7  
existem outros  
métodos

FileStreams podem ser criadas sem utilizar um File Object:  
FileReader(String fileName)

# Exemplo - Copiar um Ficheiro Binário

## Byte Streams

```
import java.io.*;

public class Copy {
    public static void main(String[] args) throws IOException {
        File inputFile = new File("picture1.jpg");
        File outputFile = new File("picture2.jpg");
        FileInputStream in = new FileInputStream(inputFile);
        FileOutputStream out = new FileOutputStream(outputFile);
        int c;

        while ((c = in.read()) != -1) // Read from Stream
            out.write(c);             // Write to Stream

        in.close();
        out.close();
    }
}
```

**Create File Objects**

**Create File Streams**

**Close the Streams**

A partir de Java 7  
existem outros  
métodos

# Classes de processamento - wrappers

- Exemplos

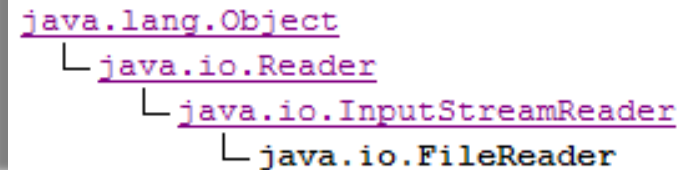
```
DataInputStream src = new DataInputStream(System.in);  
BufferedReader in = new BufferedReader(src);  
...  
PrintWriter out = new PrintWriter(  
    new FileWriter("ARQUIVO.TXT"));
```

- A fonte é um objecto do tipo **DataInputStream** que por sua vez é aberto sobre uma outra fonte
  - **DataInputStream** decorado por **BufferedReader**
  - Desta forma pode usar-se o método **readLine** de **BufferedReader**
- Do mesmo modo **FileWriter** é adaptado (wrapped) num **PrintWriter** para que o programa possa usar o método **println**.
- Desta forma podemos combinar facilidades fornecidas por diferentes manipuladores de streams.

# BufferedReader

- Leitura de caracteres do System.in

```
InputStreamReader isr = new InputStreamReader(System.in);  
char c;  
c = (char) isr.read();  
System.out.write(c);
```

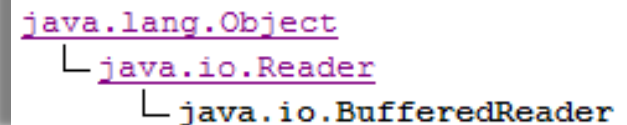


```
java.lang.Object  
└─ java.io.Reader  
    └─ java.io.InputStreamReader  
        └─ java.io.FileReader
```

- Esta leitura caracter-a-caracter não é eficiente!
  - Podemos querer ler uma linha

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

```
System.out.print("Digite uma linha:");  
String linha = stdin.readLine();
```



```
java.lang.Object  
└─ java.io.Reader  
    └─ java.io.BufferedReader
```

# Leitura de dados do teclado

- Exemplo de leitura de um inteiro em Java 1.4

```
try {  
    BufferedReader r =  
        new BufferedReader (  
            new InputStreamReader(System.in)) ;  
    String s = r.readLine() ;  
    int i = (new Integer(s)).intValue() ;  
    System.out.println(i) ;  
} catch(IOException e) { ... }
```

- .. e em Java 5.0

```
Scanner sc = new Scanner(System.in) ;  
int i = sc.nextInt() ;  
System.out.println(i) ;
```

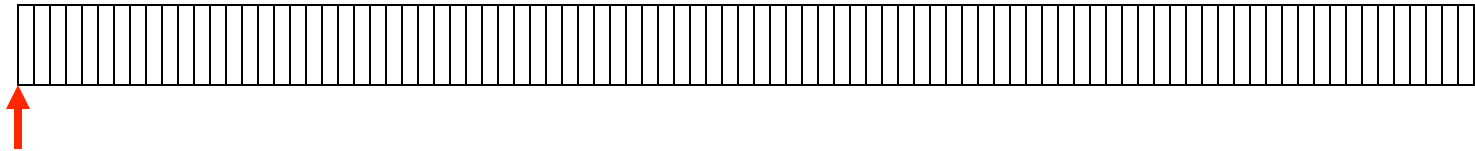
- Compare a manipulação de Exceções!

# java.io.RandomAccessFile

- Vê uma file como uma sequência de bytes
- Possui um ponteiro (seek) para ler ou escrever em qualquer ponto do ficheiro.
- Genericamente inclui operações **seek**, **read**, **write**
- Podemos apenas ler ou escrever tipos primitivos
  - `writeByte()`, `writeInt()`, `writeBoolean()`
  - `writeChars(String s)`, `writeUTF(String str)`, `String readLine()`

A partir de Java 7 existem  
outras classes / métodos  
... `FileChannell`

# java.io.RandomAccessFile

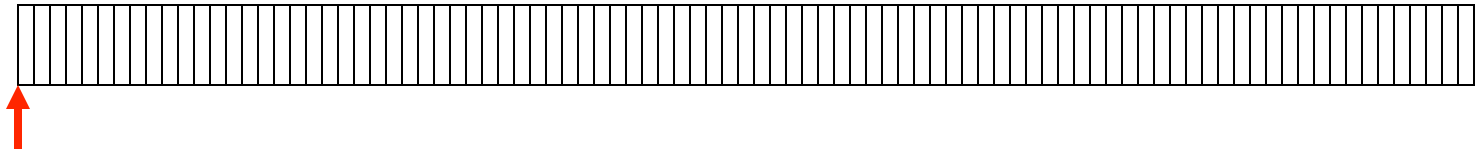


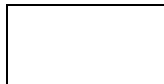
// In the file “./mydata”, copy bytes 10-19 to 0-9.

```
RandomAccessFile file = new RandomAccessFile("mydata", "rw");  
byte[] buf = new byte[10];  
file.seek(10); file.read(buf);  
file.seek(0); file.write(buf);
```



# java.io.RandomAccessFile



 **buf**: 10 bytes in memory

// In the file “./mydata”, copy bytes 10-19 to 0-9.

```
RandomAccessFile file = new RandomAccessFile(“mydata”, “rw”);
```

```
byte[] buf = new byte[10];
```

```
file.seek(10); file.read(buf);
```

```
file.seek(0); file.write(buf);
```

# java.io.RandomAccessFile



// In the file “./mydata”, copy bytes 10-19 to 0-9.

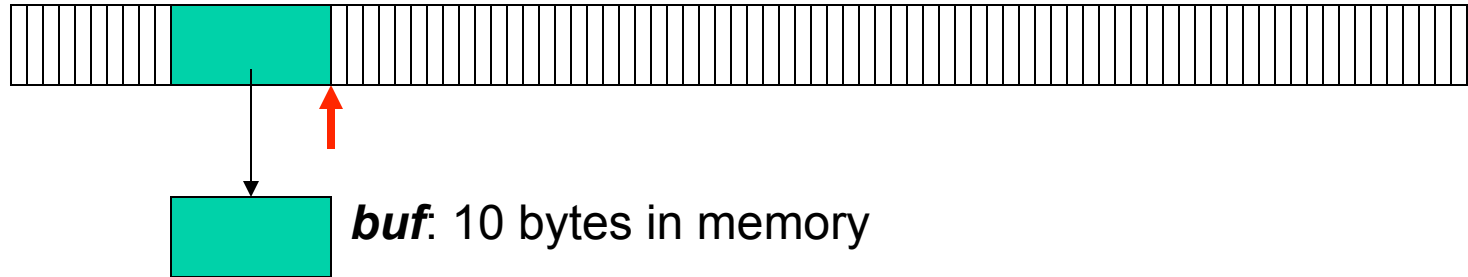
```
RandomAccessFile file = new RandomAccessFile(“mydata”, “rw”);
```

```
byte[] buf = new byte[10];
```

```
file.seek(10); file.read(buf);
```

```
file.seek(0); file.write(buf);
```

# java.io.RandomAccessFile



// In the file “./mydata”, copy bytes 10-19 to 0-9.

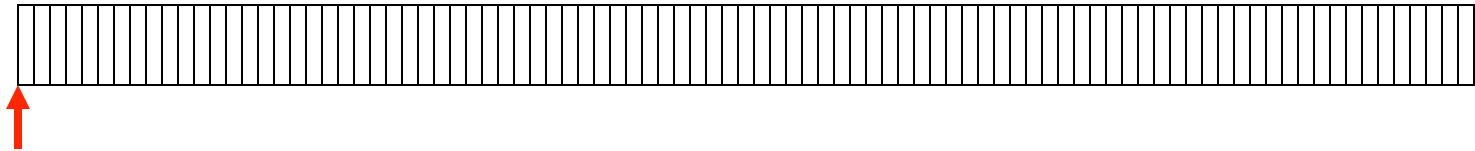
```
RandomAccessFile file = new RandomAccessFile(“mydata”, “rw”);
```

```
byte[] buf = new byte[10];
```

```
file.seek(10); file.read(buf);
```

```
file.seek(0); file.write(buf);
```

# java.io.RandomAccessFile



**buf:** 10 bytes in memory

// In the file “./mydata”, copy bytes 10-19 to 0-9.

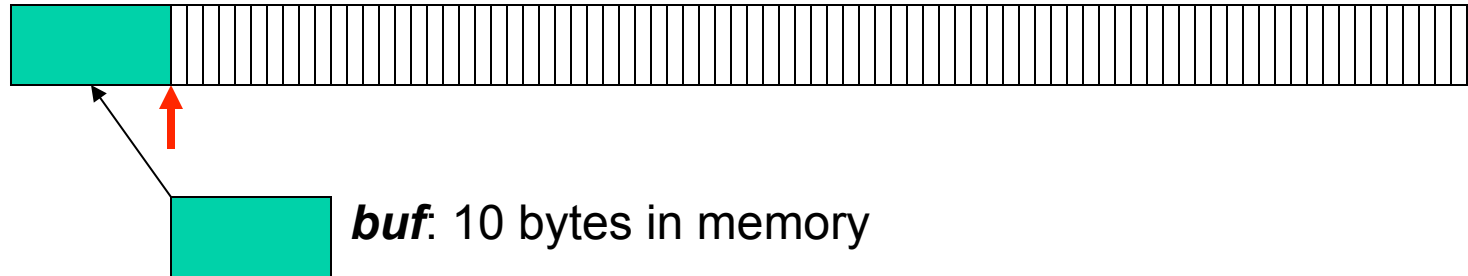
```
RandomAccessFile file = new RandomAccessFile(“mydata”, “rw”);
```

```
byte[] buf = new byte[10];
```

```
file.seek(10); file.read(buf);
```

```
file.seek(0); file.write(buf);
```

# java.io.RandomAccessFile



// In the file “./mydata”, copy bytes 10-19 to 0-9.

```
RandomAccessFile file = new RandomAccessFile(“mydata”, “rw”);
```

```
byte[] buf = new byte[10];
```

```
file.seek(10); file.read(buf);
```

```
file.seek(0); file.write(buf);
```

# java.io.RandomAccessFile

- Fazer *append* a um ficheiro que já existe.

```
File f = new File("um_ficheiro_qualquer");  
RandomAccessFile raf = new RandomAccessFile(f, "rw");  
  
// Seek to end of file  
raf.seek(f.length());  
  
// Append to the end  
raf.writeChars("agora é que é o fim");  
raf.close();
```

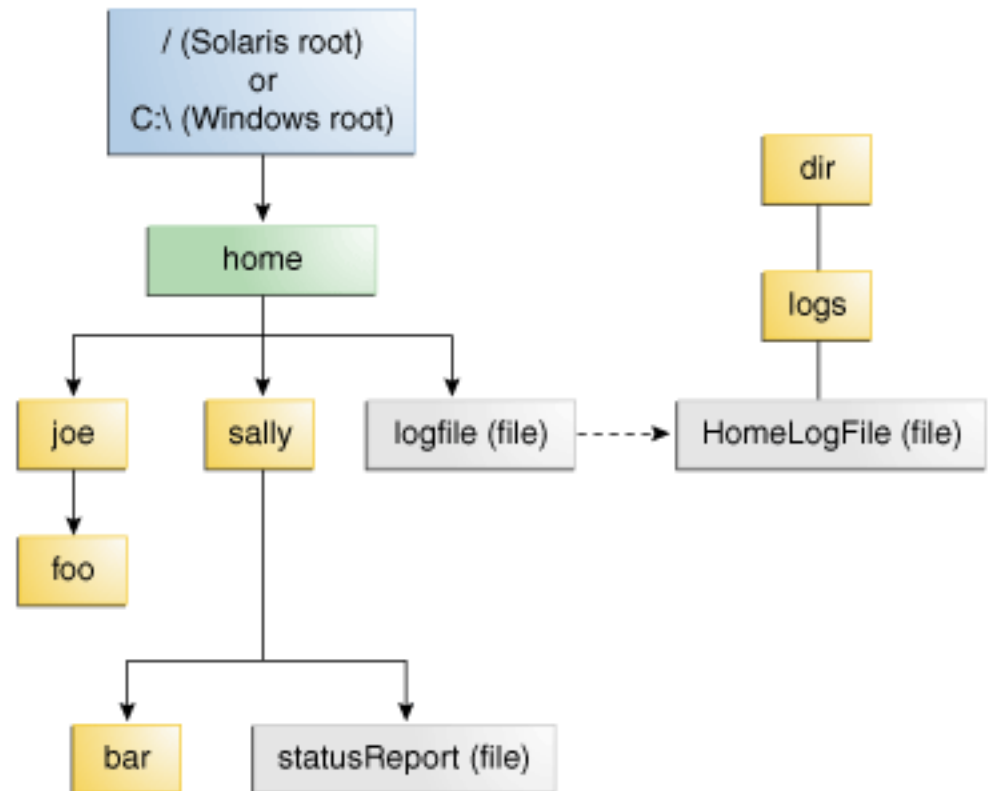
# Java NIO.2 (Java 7)

- Mudanças significativas nas classes principais
- Classe `java.nio.file.Files`
  - Só métodos estáticos para manipular ficheiros, directórios,...
- Classe `java.nio.file.Paths`
  - Só métodos estáticos para retornar um `Path` através da conversão de uma string ou Uniform Resource Identifier (URI)
- Interface `java.nio.file.Path`
  - Utilizada para representar a localização de um ficheiro ou sistema de ficheiros, tipicamente system dependent.
- Utilização comum:
  - Usar `Paths` para obter um `Path`.
  - Usar `Files` para realizar operações.



# java.nio.file.Path

- Notation dependent on the OS
  - `/home/sally/statusReport`
  - `C:\home\sally\statusReport`
- Relative or absolute
- Symbolic links
- `java.nio.file.Path`
  - Interface
  - Path might not exist





# java.nio.file.Paths

- Classe auxiliar com 2 métodos estáticos
- Permite converter strings ou um URI num Path

**static Path get(String first, String... more)**

- Converts a path string, or a sequence of strings that when joined form a path string, to a Path.

**static Path get(URI uri)**

- Converts the given URI to a Path object.

# Path

- Criar

```
Path p1 = Paths.get("/tmp/foo");  
Path p11 = FileSystems.getDefault().getPath("/tmp/foo"); // <=> p1  
Path p2 = Paths.get(args[0]);  
Path p3 = Paths.get(URI.create("file:///Users/joe/FileTest.java"));
```

- Criar no home directory logs/foo.log ou logs\foo.log

```
Path p5 = Paths.get(System.getProperty("user.home"), "logs", "foo.log");
```

# Path

- Alguns métodos:

// None of these methods requires that the file corresponding to the Path exists.

// Microsoft Windows syntax

```
Path path = Paths.get("C:\\home\\joe\\foo");
```

// Solaris syntax

```
Path path = Paths.get("/home/joe/foo");
```

```
System.out.format("toString: %s%n", path.toString());
```

```
System.out.format("getFileName: %s%n", path.getFileName());
```

```
System.out.format("getName(0): %s%n", path.getName(0));
```

```
System.out.format("getNameCount: %d%n", path.getNameCount());
```

```
System.out.format("subpath(0,2): %s%n", path.subpath(0,2));
```

```
System.out.format("getParent: %s%n", path.getParent());
```

```
System.out.format("getRoot: %s%n", path.getRoot());
```

# java.nio.file.Files

- Files helper class is feature rich:
  - Copy
  - Create Directories
  - Create Files
  - Create Links
  - Use of system “temp” directory
  - Delete
  - Attributes - Modified/Owner/Permissions/Size, etc.
  - Read/Write

# java.nio.file.Files

- Só métodos estáticos

- Copy, create, delete, ..
- isDirectory, isReadable, isWritable, ..

- Exemplo de cópia de ficheiros

```
Path src = Paths.get("/home/fred/readme.txt");  
Path dst = Paths.get("/home/fred/copy_readme.txt");  
  
Files.copy(src, dst,  
           StandardCopyOption.COPY_ATTRIBUTES,  
           StandardCopyOption.REPLACE_EXISTING);
```

- Move

- Suporta atomic move

```
Path src = Paths.get("/home/fred/readme.txt");  
Path dst = Paths.get("/home/fred/readme.1st");  
  
Files.move(src, dst, StandardCopyOption.ATOMIC_MOVE);
```

# java.nio.file.Files

- delete(Path)

```
try {  
    Files.delete(path);  
} catch (NoSuchFileException x) {  
    System.err.format("%s: no such" + " file or directory%n", path);  
} catch (DirectoryNotEmptyException x) {  
    System.err.format("%s not empty%n", path);  
} catch (IOException x) {  
    // File permission problems are caught here.  
    System.err.println(x);  
}
```

- deleteIfExists(Path)

- Sem exceções

# java.nio.file.Files

- Verificar se dois Paths indicam a mesma File
  - Num sistema de ficheiros com links simbólicos podemos ter dois caminhos distintos a representar o mesmo ficheiro
  - Usar `isSameFile(Path, Path)` para fazer a comparação

```
Path p1 = ...;
```

```
Path p2 = ...;
```

```
if (Files.isSameFile(p1, p2)) {  
    // Logic when the paths locate the same file  
}
```

# java.nio.file.DirectoryStream<T>

- Interface DirectoryStream actua como um iterador
  - Scales to large directories
  - Uses less resources
  - Smooth out response time for remote file systems
  - Implements Iterable and Closeable for productivity
- Filtering support
  - Build-in support for glob, regex and custom filters

```
Path srcPath = Paths.get("/home/fred/src");
try (DirectoryStream<Path> dir = Files.newDirectoryStream(srcPath, "*.java")) {
    for (Path file : dir)
        System.out.println(file);
}
```



# java.nio.file.DirectoryStream

## Exemplos de glob

- Glob é um padrão para filtragem de ficheiros. Exemplos de sintaxe:

`*.html` - Matches all strings that end in .html

`???` - Matches all strings with exactly three letters or digits

`*[0-9]*` - Matches all strings containing a numeric value

`*.{htm,html,pdf}` - Matches any string ending with .htm, .html or .pdf

`a?*.java` - Matches any string beginning with a, followed by at least one letter or digit, and ending with .java

`{foo*,*[0-9]*}` - Matches any string beginning with foo or any string containing a numeric value

- Exemplo de método que usa glob

`static DirectoryStream<Path> newDirectoryStream(Path dir, String glob)`

# Java NIO.2 Symbolic Links

- Path and Files are “link aware”

```
Path existingFile = Paths.get(. . .);
```

```
Path newLink = Paths.get(. . .);
```

```
try {
```

```
    Files.createLink(newLink, existingFile);
```

```
} catch (IOException x) {
```

```
    System.err.println(x);
```

```
} catch (UnsupportedOperationException x) {
```

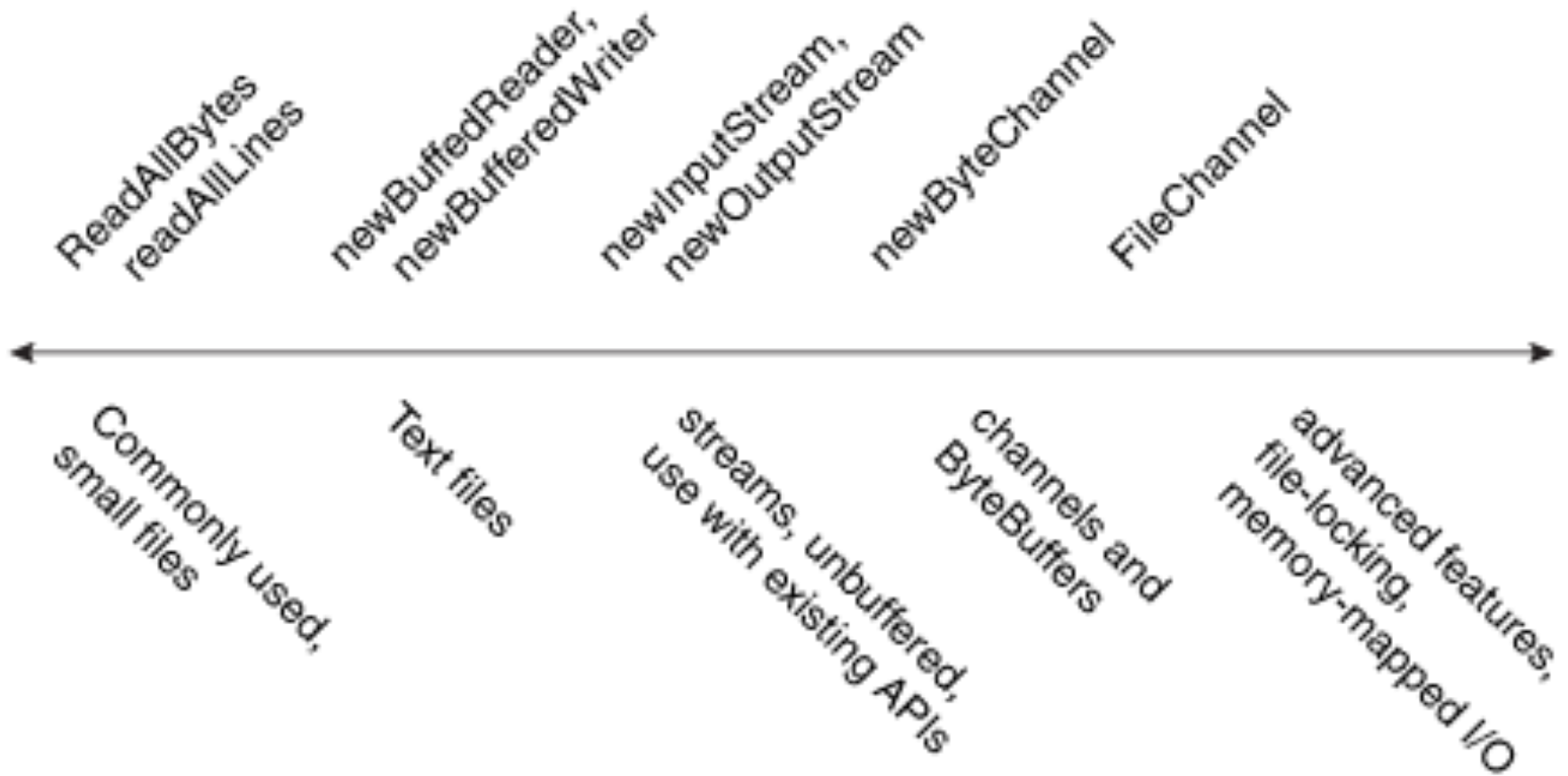
```
    //Some file systems or some configurations
```

```
        //may not support links
```

```
    System.err.println(x);
```

```
}
```

# Reading, Writing, and Creating Files



# Ficheiros pequenos

- Podemos usar alguns métodos novos que lidam já com abrir e fechar o ficheiro
- Reading All Bytes or Lines from a File

```
Path file = ...;  
byte[] fileArray;  
fileArray = Files.readAllBytes(file);
```

- Writing All Bytes or Lines to a File

```
Path file = ...;  
byte[] buf = ...;  
Files.write(file, buf);
```

# newBufferedReader/Writer

- Leitura

```
Charset charset = Charset.forName("US-ASCII");
try (BufferedReader reader = Files.newBufferedReader(file, charset)) {
    String line = null;
    while ((line = reader.readLine()) != null) {
        System.out.println(line);
    }
} catch (IOException x) {
    System.err.format("IOException: %s%n", x);
}
```

- Escrita

```
Charset charset = Charset.forName("US-ASCII");
String s = ...;
try (BufferedWriter writer = Files.newBufferedWriter(file, charset)) {
    writer.write(s, 0, s.length());
} catch (IOException x) {
    System.err.format("IOException: %s%n", x);
}
```

# FileChannel

- Random access files - para aceder temos de abrir, usar seek e ler/escrever
- A interface SeekableByteChannel encapsula agora esta funcionalidade.

**position** – Returns the channel's current position

**position(long)** – Sets the channel's position

**read(ByteBuffer)** – Reads bytes into the buffer from the channel

**write(ByteBuffer)** – Writes bytes from the buffer to the channel

**truncate(long)** – Truncates the file (or other entity) connected to the channel

- A classe FileChannel implementa esta interface

# FileChannel

```
try (RandomAccessFile ra = new RandomAccessFile(filename, "rw");
    FileChannel fc = ra.getChannel());{

    System.out.println("FileChannel - Size: " + fc.size());
    System.out.println("FileChannel - Position: " + fc.position());
    fc.position(fc.position() + 30);

    ByteBuffer buf = ByteBuffer.allocate(8);
    int bytesRead = fc.read(buf);
    if (bytesRead > 0)
        System.out.println("FileChannel - Read 30-38: "
                           + new String(buf.array()) + "\n");

} catch (IOException x) {
    System.err.format("IOException: %s\n", x);
}
```

# Iterar sobre a árvore de pasta e ficheiros

- A interface `FileVisitor` incorpora um conjunto de métodos que torna a navegação na árvore mais fácil.
- A classe `SimpleFileVisitor` implementa:
  - `preVisitDirectory(T dir, BasicFileAttributes attrs);`
  - `visitFile(T dir, BasicFileAttributes attrs);`
  - `visitFileFailed(T dir, IOException exc);`
  - `postVisitDirectory(T dir, IOException exc);`
- Geralmente esta classe deve ser extendida para incluir as funcionalidades desejadas.



# FileVisitor

```
Path inputPath = Paths.get("src");
```

```
SimpleFileVisitor<Path> pf = new SimpleFileVisitor<Path>() {  
    @Override  
    public FileVisitResult visitFile(Path file,  
                                     BasicFileAttributes attrs) throws IOException {  
        System.out.println(file.getFileName()+  
                             ": "+Files.size(file));  
        return FileVisitResult.CONTINUE;  
    }  
};
```

```
Files.walkFileTree(inputPath, pf);
```

BST.java: 13440

BSTNode.java: 490

Col1.java: 1086

Col2.java: 1075

Queue.java: 533

DC.java: 502

# Watching A Directory

- Criar um WatchService
- Registrar num directório
- “Watcher” can be polled or waited on for events
  - Events raised in the form of Keys
  - Retrieve the Key from the Watcher
  - Key has filename and events within it for create/delete/modify

```
import static java.nio.file.StandardWatchEventKinds.*;
Path dir = ...;
try {
    WatchKey key = dir.register(watcher,
                                ENTRY_CREATE, ENTRY_DELETE, ENTRY_MODIFY);
} catch (IOException x) {
    System.err.println(x);
}
```

# Resumo de diferenças NIO.2

- <http://docs.oracle.com/javase/tutorial/essential/io/legacy.html>

# Serialização

# Serialização

- E se quisermos ler ou escrever Objectos em Ficheiros?
  - **Serialização**: permite tornar persistentes os objectos
- O processo de Serialização é complicado em muitas linguagens
  - Podemos ter objectos contendo referências para outros objectos...
- Java permite implementar Serialização de forma simples
- **Definição**: Serialização é o processo de transformar um objecto numa sequência (stream) de bytes

# Serialização

- Para que uma classe seja serializável basta que implemente a interface *Serializable* (que é uma interface vazia!)

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

- **Serializable** - Permite simplesmente indicar quais as classes serializáveis

# Condições de Serialização

- A classe deve ser declarada como public
- A classe deve implementar **Serializable**
- Todos os atributos (dados) devem ser serializáveis:
  - Tipos primitivos (int, double, ...)
  - Objectos serializáveis

# Serialização - Algumas Considerações

- Um atributo definido como `transient` não será “empacotado” no processo de serialização.
  - No processo de desserialização os atributos assumirão valores de defeito.
- Atributos do tipo `static` não são serializados.
- Se uma classe B serializable tem uma super-classe A que não é serializable, então objectos do tipo B podem ser serializados ... desde que a classe A tenha um construtor sem argumentos acessível.

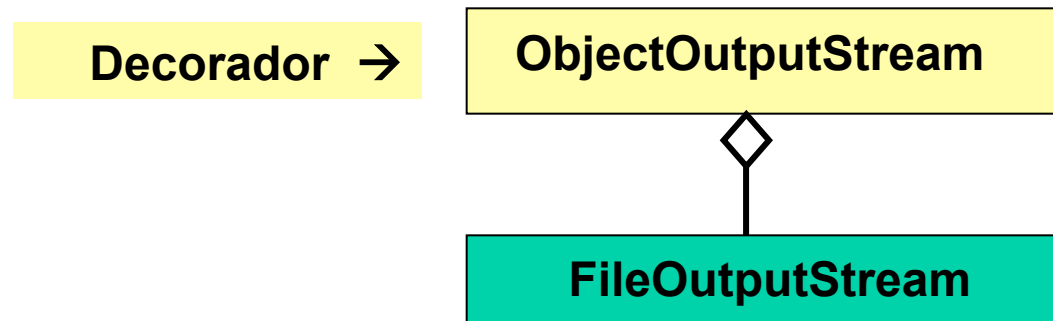


# Serialização - serialVersionUID

- Atributo **Muito Importante**
- Deve ser sempre incluído na Classe:
  - `private static final long serialVersionUID = 75264722956L;`
- Não deve ser alterado em versões futuras das classes, excepto...
- ... ambas as versões gerarem objectos incompatíveis
  - A compatibilidade de novas versões com objectos antigos depende da natureza das alterações.

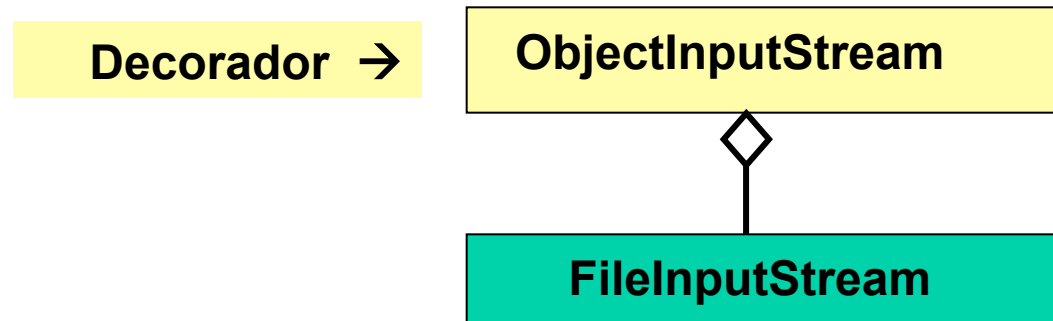
# Escrita de Objectos em Ficheiro

```
ObjectOutputStream objectOut =  
    new ObjectOutputStream(  
        new FileOutputStream(fileName));  
  
objectOut.writeObject(serializableObject);  
  
objectOut.close( );
```



# Leitura de Objectos de Ficheiro

```
ObjectInputStream objectIn =  
    new ObjectInputStream(  
        new FileInputStream(fileName)));  
  
myObject = (ObjectType)objectIn.readObject();  
  
objectIn.close( );
```



# Exemplo - Serialização

- **ObjectOutputStream**

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

- **ObjectInputStream**

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

- A leitura faz-se pela mesma ordem da escrita

# Exemplo - Escrita Objectos

```
public static void main(String[] args) throws FileNotFoundException, IOException {  
    Data d = new Data(11,2,2001);  
    Pessoa p = new Pessoa("Carlos Costa", 234342124, new Data(22,11,1972));  
  
    ObjectOutputStream objectOut = new ObjectOutputStream(  
        new FileOutputStream("c:/out.bin"));  
  
    objectOut.writeObject(d);  
    objectOut.writeObject(p);  
    objectOut.close();  
}
```



C:\out.bin

ACED	0005	7372	0007	494F	2E44	6174	612D	C51E	40EE	1B7D	8E02	0003	4900	0361	...
6E6F	4900	0364	6961	4900	036D	6573	7870	0000	07D1	0000	000B	0000	0002	7372	noI..dial..mesxp.....sr
0009	494F	2E50	6573	736F	6171	314B	6257	84CE	2102	0003	4900	0262	694C	0008	..IO.Pessoaq1KbW...!...I..biL..
6461	7461	4E61	7363	7400	094C	494F	2F44	6174	613B	4C00	046E	6F6D	6574	0012	dataNasct..LIO/Data;L..nomet..
4C6A	6176	612F	6C61	6E67	2F53	7472	696E	673B	7870	0DF7	C6EC	7371	007E	0000	Ljava/lang/String;xp....sq.~..
0000	07B4	0000	0016	0000	000B	7400	0C43	6172	6C6F	7320	436F	7374	61		.....t..Carlos Costa

# Serialização - Utilização

- Persistência
  - Com FileOutputStream
  - Armazena as estruturas de dados em ficheiro para mais tarde recuperar
- Cópia
  - Com ByteArrayOutputStream
  - Armazena as estruturas de dados em memória (array) para poder criar duplicados
- Comunicações
  - Utilizando um stream associado a um Socket
  - Envia as estruturas de dados para outro computador

# Serialização - Deep Copy

```
// serialize object  
ByteArrayOutputStream mOut = new ByteArrayOutputStream();  
ObjectOutputStream serializer = new ObjectOutputStream(mOut);  
serializer.writeObject(serializableObject);  
serializer.flush();  
  
// deserialize object  
ByteArrayInputStream mIn = new  
    ByteArrayInputStream(mOut.toByteArray());  
ObjectInputStream deserializer = new ObjectInputStream(mIn);  
Object deepCopyOfOriginalObject = deserializer.readObject();
```

# Serialização - Controlar o Processo

- Como podemos controlar o processo de serialização e desserialização?

Reescrevendo os métodos `writeObject` e `readObject`:

...

`String a;`

`transient String b;`

```
private void writeObject(ObjectOutputStream stream)
    throws IOException {
    stream.defaultWriteObject();
    stream.writeObject(b);    // forçar a ser. de b
}
```

```
private void readObject(ObjectInputStream stream)
    throws IOException, ClassNotFoundException {
    stream.defaultReadObject();
    b = (String) stream.readObject();
}
```

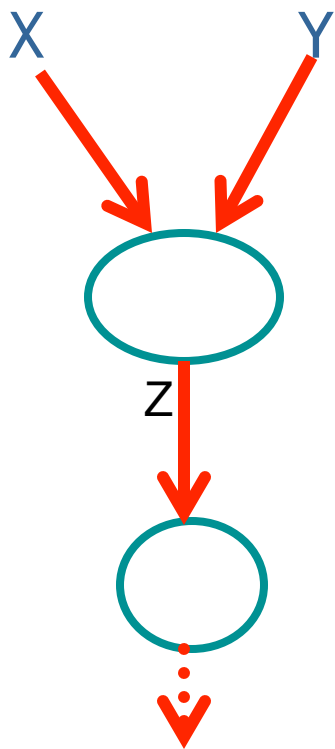
...



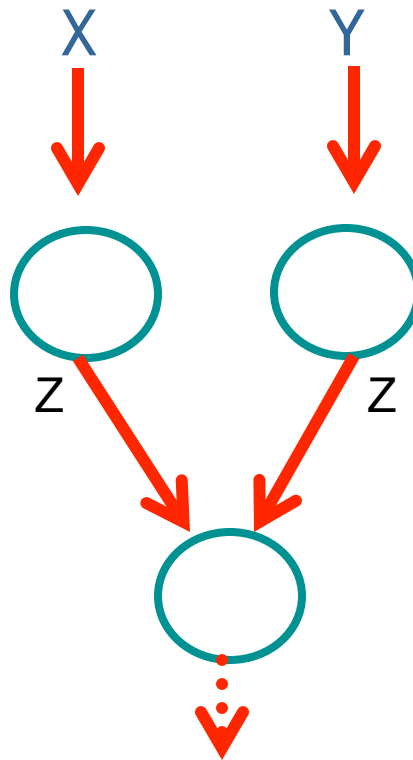
# Java Serializable Comparison

- Example (X.field = Z)

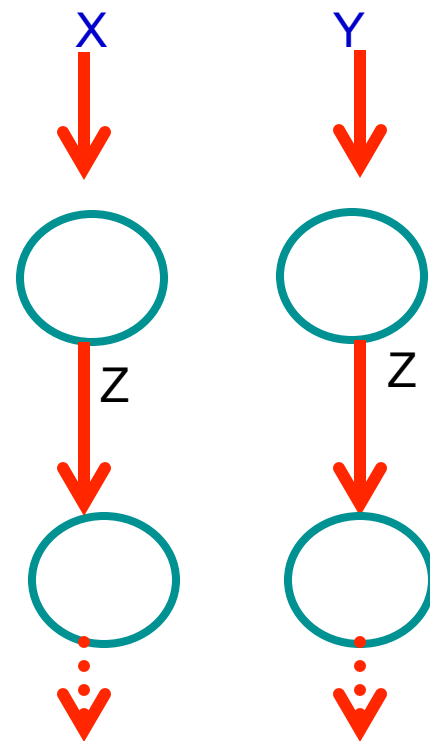
- $Y = X$



- $Y = X.clone()$



- $OS.writeObject(x)$   
 $Y = readObject(IS)$



# Jar Files

# O que são Jar files?

- O Java Archive (JAR) permite a inclusão de múltiplos ficheiros num único ficheiro arquivo.
- Tipicamente, o ficheiro JAR contém “.class files” e recursos auxiliares associados com applets ou aplicações.
- Os ficheiros JAR são compactados em formato ZIP
  - Podemos utilizar o “Winzip” para manipular JARs

# Vantagens

- **Compressão:** O arquivo JAR comprime os seus conteúdos.
  - Aumento da eficiência no transporte (- tempo download) e arquivo (- espaço disco)
- **Segurança:** Os ficheiros JAR podem ser assinados digitalmente.
  - autenticação da proveniência.
  - privilégios do software baseados na certificação da origem.

# Vantagens

- **Packaging for extensions**: é possível adicionar novas funcionalidades ao Java *core platform*, utilizando arquivos Jar.
- **Package Sealing**: forçar a consistência de versões.
  - Todas as classes definidas no package devem ser encontradas no mesmo arquivo Jar.
- **Package Versioning**: suporta informação relativa ao software: vendedor, versão, etc.
- **Portabilidade**: suporte de JARs é uma componente standard do Java *platform's core API*.

# Java Archive Tool - comando jar

## Operações

- create a JAR file
- view the contents of a JAR file
- extract the contents of a JAR file
- extract specific files from a JAR file
- run an application packaged as a JAR file (version 1.2 -- requires Main-Class manifest header)

## Comando

`jar cf jar-file input-file(s)`

`jar tf jar-file`

`jar xf jar-file`

`jar xf jar-file archived-file(s)`

`java -jar app.jar`

# Jar - Manifest File

- Ficheiro especial que contém diversos tipos de ‘Meta’ informação relativas ao arquivo JAR:
  - electronic signing, version control, package sealing, entry-point, ...
- Na criação de um JAR é criada uma “default manifest file”

**META-INF/MANIFEST.MF**



**Manifest-Version: 1.0**

**Created-By: 1.6.0 (Sun Microsystems Inc.)**

# Executable JAR archive

- Como tornar uma aplicação em Java num arquivo JAR executável?

1. Colocar todas as classes num directório (estrutura árvore)
2. Criar um arquivo JAR com esse directório
3. Adicionar na Manifest File um *entry-point*

**Main-Class: classname**

4. A main-class deve ter o método

**public static void main(String[] args)**

```
Manifest-Version: 1.0
```

```
Class-Path: .
```

```
Main-Class: aula1.Palindrome
```

5. Para executar  
**\$ java -jar app.jar**



# Sumário

- Sistema de Entrada e Saída (I/O)
- Streams de Dados (byte e char)
- Classes de Processamento
- Acesso Aleatório a Ficheiros
- Serialização de Objectos
- Ficheiros JAR