Computação Orientada a Objetos

Arquivos

Profa. Thienne Johnson

EACH/USP

Conteúdo

- Java, como programar, 6ª edição
  - Deitel & Deitel
  - Capítulo 14

Introdução

- Programadores utilizam arquivos para armazenar dados a longo prazo
- Dados armazenados em arquivos são chamados de persistentes:
  - eles existem mesmo depois que os programas que os criaram tenham terminado
- O termo fluxo se refere a dados que são lidos ou gravados em um arquivo

Hierarquia de dados

- Bit
  - menor item de dados em um computador
    - assume valor 0 ou 1
  - inadequado para leitor humano
  - usamos caracteres (dígitos, símbolos, letras)
    - representados no computador como padrões de 0's e 1's
    - em Java: caracteres Unicode compostos de dois bytes

Hierarquia de dados

- Byte
  - Grupo de 8 bits;
  - 2 bytes (16 bits) são usados para representar um caractere Unicode;
- Campo
  - Grupo de caracteres com significado;
  - Exemplo: endereço de funcionário;
- Registro
  - Grupo de campos relacionados;
  - Exemplo: registro de um funcionário.

Hierarquia de dados

- Arquivo
  - Grupo de registros relacionados;
  - Exemplo: informações sobre muitos funcionários;

- Banco de Dados
  - Grupo de arquivos relacionados;
  - Exemplo: arquivo de folha de pagamento, arquivo de contas a receber, arquivos de contas a pagar, etc.

Exemplo

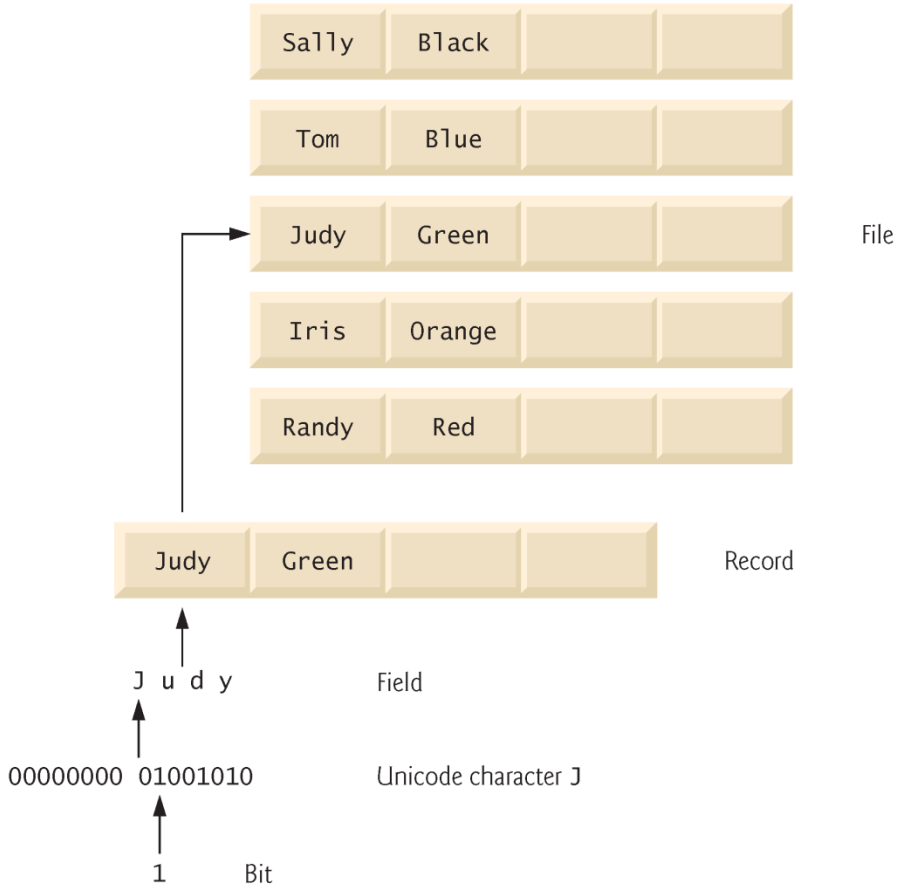


Fig. 17.1 | Data hierarchy.

Arquivos e fluxos

- Java vê cada arquivo como um fluxo sequencial de bytes
  - geralmente terminam com uma marca de final de arquivo ou um código especial

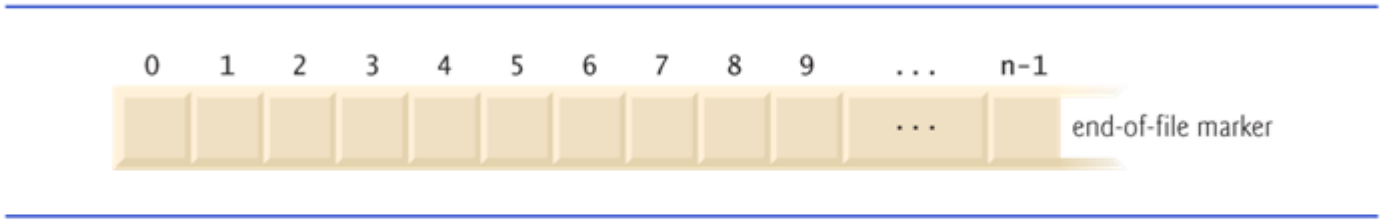


Fig. 17.2 | Java's view of a file of *n* bytes.

  - *Um programa Java simplesmente recebe uma indicação do S.O. quando chega ao fim do arquivo*

- Fluxos de arquivos podem ser utilizados para entrada e saída de dados como caracteres ou bytes

Arquivos e fluxos

Tipos de arquivos

- Arquivos binários: criados com base em fluxos de bytes
  - lidos por um programa que converte os dados em um formato legível por humanos
- Arquivos de texto: criados com base em fluxos de caracteres
  - podem ser lidos por editores de texto

Arquivos e fluxos

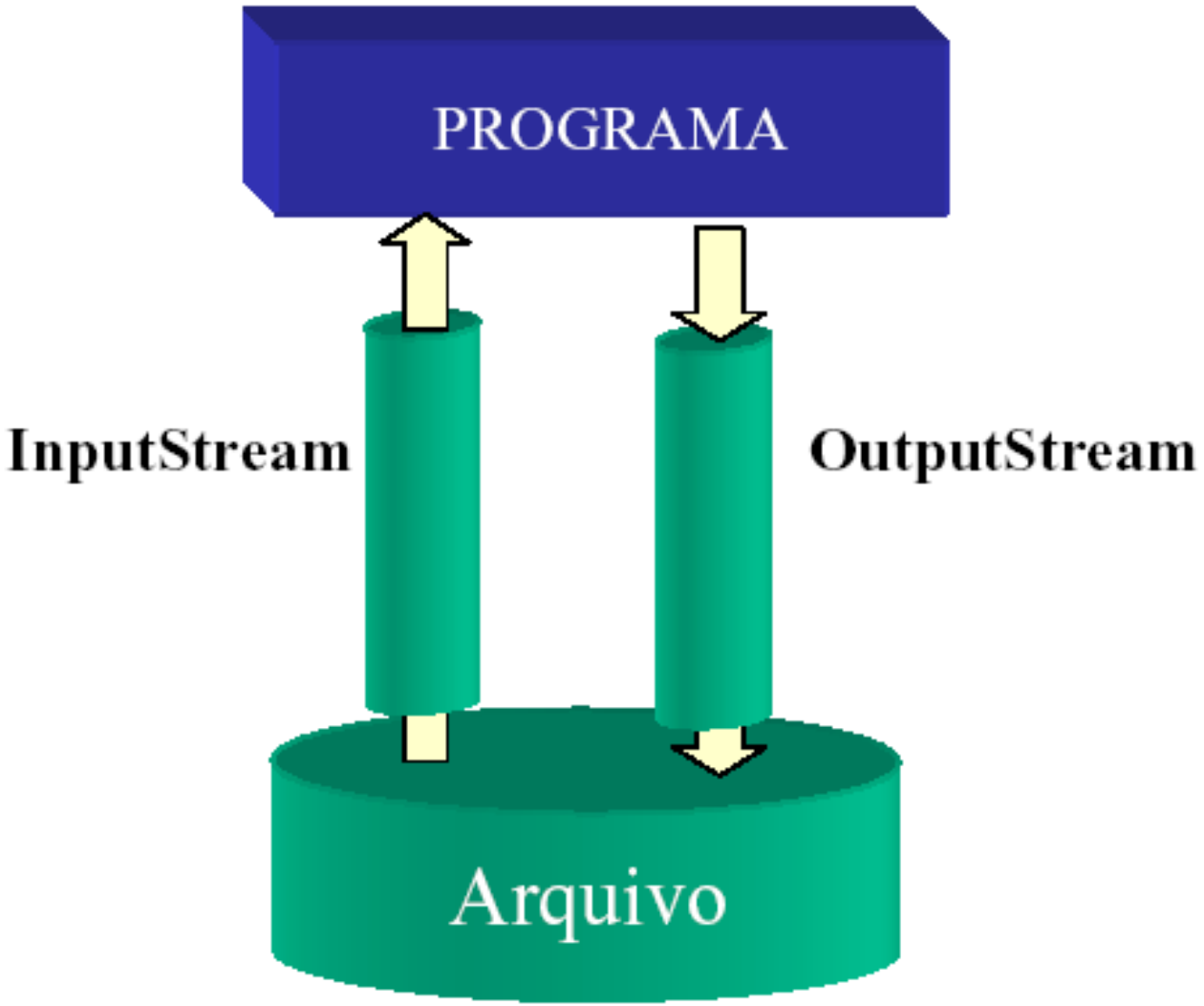
- Um programa Java abre um arquivo criando e associando um objeto a um fluxo de bytes ou caracteres
  - Também pode associar fluxos com dispositivos diferentes
- Java cria 3 objetos de fluxo quando um programa começa execução
  - `System.in` (objeto padrão de fluxo de entrada) normalmente recebe dados do teclado
  - `System.out` (objeto padrão de fluxo de saída) normalmente mostra os dados na tela
  - `System.err` (objeto padrão de fluxo de erros) normalmente mostra as mensagens de erro na tela
- A classe System fornece os métodos **setIn**, **setOut** e **setErr** para **redirecionar** os fluxos de saída, entrada e erros, respectivamente.

Arquivos e fluxos

- O processamento de arquivos é realizado utilizando o pacote **`java.io`**

- APIs Java para I/O oferecem objetos que abstraem fontes e destinos, fluxos de bytes e caracteres

- Dois grupos:
  - Entrada e Saída de bytes:
    - InputStream e OutputStream;
  - Entrada e Saída de caracteres (chars):
    - Reader e Writer.

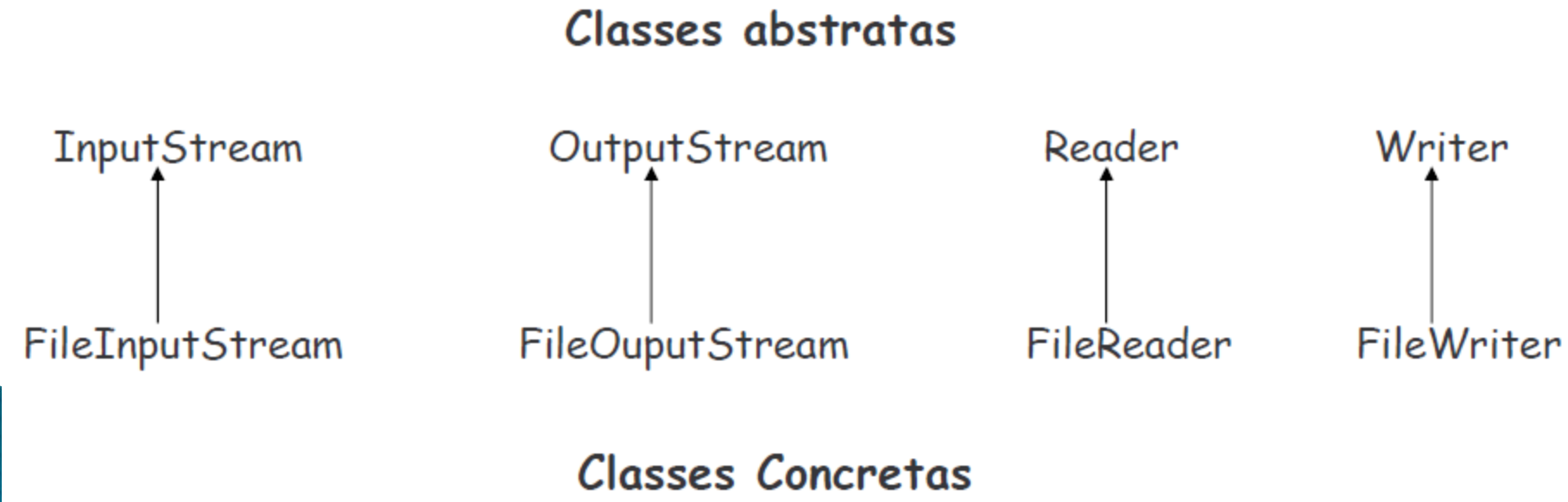

PROGRAMA
InputStream
OutputStream
Arquivo

Hierarquia de Classes do pacote Java.io

Java.java.io
- File
- FileDescriptor
- *InputStream*
  - ByteArrayInputStream
  - FileInputStream
  - *FilterInputStream*
    - BufferedInputStream
    - DataInputStream
    - PushBackInputStream
  - ObjectInputStream
  - PipedInputStream
  - SequenceInputStream
- *OutputStream*
  - ByteArrayOutputStream
  - FileOutputStream
  - *FilterOutputStream*
    - BufferedOutputStream
    - DataOutputStream
    - PrintStream
  - ObjectOutputStream
  - PipedOutputStream
- RandomAccesFile
- *Reader*
  - BufferedReader
    - LineNumberReader
  - CharArrayReader
  - *FilterReader*
    - PushbackReader
  - InputStreamReader
    - FileReader
  - PipedReader
  - StringReader
- *Writer*
  - BufferedWriter
  - CharArrayWriter
  - *FilterWriter*
  - OutpurStreamWriter
    - FileWriter
  - PipedWriter
  - PrintWriter
  - StringReader

Arquivos e fluxos

- Class FileInputStream, FileOutputStream, FileReader, FileWriter
- Os arquivos são abertos criando-se objetos destas classes de fluxo que herdam de InputStream, OutputStream, Reader, Writer como pode ser visto na figura



Arquivos e fluxos

- Pode executar entrada e saída de objetos ou variáveis de tipos de dados primitivos sem se preocupar com os detalhes da conversão para formato em byte.

- **`ObjectInputStream`** e **`ObjectOutputStream`** podem ser usados com as classes de fluxo de bytes `FileInputStream` e `FileOutputStream`.

- Hierarquia das classes de io.

http://java.sun.com/javase/6/docs/api/java/io/package-tree.html

A classe *File*

- Útil para recuperar informações sobre arquivos e diretórios em disco
- Não abre nem processa arquivos
- É utilizada com objetos de outras classes do pacote **`java.io`** para especificar arquivos ou diretórios a manipular
- http://java.sun.com/javase/6/docs/api/java/io/File.html

Criando objetos *File*

- **`File(String nome)`**
  - especifica o nome de um arquivo ou diretório para associar a um objeto File
  - o nome pode conter informações de caminho
    - caminho absoluto: inicia no diretório raiz e inclui todo o caminho levando ao arquivo
    - caminho relativo: inicia no diretório onde a aplicação foi iniciada

Caminho

- Instâncias da classe *java.io.File* representam caminhos (paths) para possíveis locais no sistema operacional. Lembre-se que ele apenas representa um arquivo ou diretório, isto não quer dizer que este caminho exista ou não.

- Exemplos
  - File noDiretorioAtual = **new** File("arquivo.txt");
  - File noDiretorioAnterior = **new** File("../arquivo.txt");
  - File diretorioRaiz = **new** File("/");
  - File arquivo1 = **new** File(diretorioRaiz,"autoexec.bat");
  - File arquivo2 = **new** File(diretorioRaiz,"config.sys");
  - File diretorioWindows = **new** File(diretorioRaiz,"windows");
  - File diretorioWindows2 = **new** File("/windows/");
  - File diretorioWindows3 = **new** File("/windows");
  - File diretorioWindows4 = **new** File("c:\\\\windows");

Caminho – Caracter de separação

- Um **caracter de separação** é usado para separar diretórios e arquivos no caminho
  - No Windows: barra invertida (\).
  - No Linux/UNIX: barra normal (/).

- Java processa ambos de forma idêntica.

- Quando criar `String`s que representam o caminho, use `File.separator` para obter o separador do computador local.
  - Retorna uma `String` consistindo de 1 caracter.

Métodos *File*

- **`boolean canRead()`**
  - retorna **`true`** se um arquivo puder ser lido pelo aplicativo
- **`boolean canWrite()`**
  - retorna **`true`** se um arquivo puder ser gravado pelo aplicativo
- **`boolean exists()`**
  - retorna **`true`** se o argumento para o construtor é um arquivo ou diretório válido

Métodos *File*

- **`boolean isFile()`**
  - retorna **`true`** se o nome especificado como argumento para o construtor é um arquivo

- **`boolean isDirectory()`**
  - retorna **`true`** se o nome especificado como argumento para o construtor é um diretório

- **`boolean isAbsolute()`**
  - retorna **`true`** se o nome especificado como argumento para o construtor é um caminho absoluto

Métodos *File*

- **`String getAbsolutePath()`**
  - retorna uma string com o caminho absoluto do arquivo ou diretório

- **`String getName()`**
  - retorna uma string com o nome do arquivo ou diretório

- **`String getPath()`**
  - retorna uma string com o caminho do arquivo ou diretório

Métodos *File*

- **`String getParent()`**
  - retorna uma string com o diretório-pai do arquivo ou diretório

- **`long length()`**
  - retorna o comprimento do arquivo em bytes (0 se for um diretório)

Error-Prevention Tip 17.1

Use `File` *method* `isFile` *to determine whether a* `File` *object represents a file (not a directory) before attempting to open the file.*

Exemplo 1: Criação de Diretórios e de um Arquivo Vazio- File

```
File diretorio = new File("c:\\novo");
diretorio.mkdir(); // cria, se possível
File subdir1 = new File( diretorio, "subdir1");
subdir1. mkdir();
File subdir2 = new File( diretorio, "subdir2");
subdir2. mkdir();
File arquivo = new File( diretorio, "arquivoVazio.txt");
FileWriter f = new FileWriter(arquivo);
f.close();
String[] arquivos = diretorio. list();
for (int i =0;i<arquivos.length; i++) {
   File filho = new File( diretorio, arquivos[ i]);
   System. out. println(filho.getAbsolutePath());
}
```

Métodos *File* – Exemplo 2

- Exercício: Escrever um programa que teste a classe **`FileDemonstration`**
  - o usuário pode fornecer uma string com o nome do arquivo/diretório para um objeto **`scanner`**
  - essa string é passada como argumento para o método **`analyzepath`** da classe **`FileDemonstration`**

```
1  // Fig. 17.4: FileDemonstration.java
2  // File class used to obtain file and directory information.
3  import java.io.File;
4  import java.util.Scanner;
5
6  public class FileDemonstration
7  {
8     public static void main( String[] args )
9     {
10       Scanner input = new Scanner( System.in );
11
12       System.out.print( "Enter file or directory name: " );
13       analyzePath( input.nextLine() );
14    } // end main
15
16    // display information about file user specifies
17    public static void analyzePath( String path )
18    {
19       // create File object based on user input
20       File name = new File( path );
21
```

Associates a file or directory with a `File` object.

Fig. 17.4 | `File` class used to obtain file and directory information. (Part 1 of 5.)

```
22          if ( name.exists() ) // if name exists, output information about it
23          {
24             // display file (or directory) information
25             System.out.printf(
26                "%s%s\n%s\n%s\n%s\n%s%s\n%s%s\n%s%s\n%s%s\n%s%s",
27                name.getName(), " exists",
28                ( name.isFile() ? "is a file" : "is not a file" ),
29                ( name.isDirectory() ? "is a directory" :
30                   "is not a directory" ),
31                ( name.isAbsolute() ? "is absolute path" :
32                   "is not absolute path" ), "Last modified: ",
33                name.lastModified(), "Length: ", name.length(),
34                "Path: ", name.getPath(), "Absolute path: ",
35                name.getAbsolutePath(), "Parent: ", name.getParent() );
36
37             if ( name.isDirectory() ) // output directory listing
38             {
39                String[] directory = name.list();
40                System.out.println( "\n\nDirectory contents:\n" );
41
42                for ( String directoryName : directory )
43                   System.out.println( directoryName );
44             } // end if
45          } // end outer if
```

Determines if the file or directory exists.

Returns an array of `String`s containing the directory's contents.

Fig. 17.4 | `File` class used to obtain file and directory information. (Part 2 of 5.)

```
46         else // not file or directory, output error message
47         {
48            System.out.printf( "%s %s", path, "does not exist." );
49         } // end else
50      } // end method analyzePath
51   } // end class FileDemonstration
```

Fig. 17.4 | `File` class used to obtain file and directory information. (Part 3 of 5.)

```
Enter file or directory name: E:\Program Files\Java\jdk1.6.0_11\demo\jfc
jfc exists
is not a file
is a directory
is absolute path
Last modified: 1228404395024
Length: 4096
Path: E:\Program Files\Java\jdk1.6.0_11\demo\jfc
Absolute path: E:\Program Files\Java\jdk1.6.0_11\demo\jfc
Parent: E:\Program Files\Java\jdk1.6.0_11\demo

Directory contents:

CodePointIM
FileChooserDemo
Font2DTest
Java2D
Laffy
Metalworks
Notepad
SampleTree
Stylepad
SwingApplet
SwingSet2
SwingSet3
```

Fig. 17.4 | `File` class used to obtain file and directory information. (Part 4 of 5.)

```
Enter file or directory name: C:\Program Files\Java\jdk1.6.0_11\demo\jfc
\Java2D\README.txt
README.txt exists
is a file
is not a directory
is absolute path
Last modified: 1228404384270
Length: 7518
Path: E:\Program Files\Java\jdk1.6.0_11\demo\jfc\Java2D\README.txt
Absolute path: E:\Program Files\Java\jdk1.6.0_11\demo\jfc\Java2D\RE-
ADME.txt
Parent: E:\Program Files\Java\jdk1.6.0_11\demo\jfc\Java2D
```

Fig. 17.4 | `File` class used to obtain file and directory information. (Part 5 of 5.)

Common Programming Error 17.1

Using \ as a directory separator rather than \\ in a string literal is a logic error. A single \ indicates that the \ followed by the next character represents an escape sequence. Use \\ to insert a \ in a string literal.

Arquivos de texto de acesso sequencial

- Vistos como um fluxo de caracteres ou bytes

- Só podem ser percorridos do início para o fim (e não no sentido contrário)

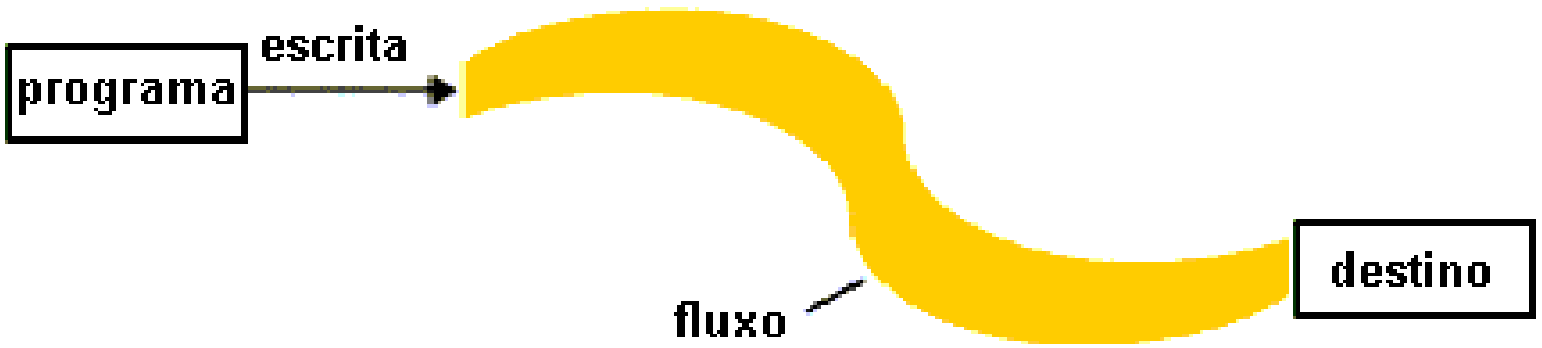
- Java não impõe estruturas no arquivo
  - Registros nao fazem parte da linguagem
  - Deve-se estruturar os arquivos para alcançar os requisitos da aplicação

Leitura sequencial

- Para trazer informação de uma origem (e.g., arquivo, memória etc), um programa Java abre um fluxo (*stream*) para leitura seqüencial.

Escrita sequencial

- Para enviar informação a um destino (e.g., arquivo, memória etc) um programa Java abre um fluxo (*stream*) para escrita seqüencial.



Procedimento geral

Leitura

```
abrir fluxo
enquanto houver dados
  ler
fechar fluxo
```

Gravação

```
abrir fluxo
enquanto houver dados
  escrever
fechar fluxo
```

Classes de fluxos

- O procedimento para utilizar um fluxo de bytes ou um fluxo de caracteres é praticamente o mesmo
  - 1) criar um objeto de fluxo
  - 2) chamar seus métodos para enviar ou receber dados, dependendo se é um fluxo de entrada ou um fluxo de saída
  - 3) fechar o fluxo de dados

Fluxos de bytes

- Todos fluxos de bytes são subclasses de **`InputStream`** ou **`OutputStream`** (abstratas)
- Utilizados para manipulação de arquivos binários (ex: som, imagem ou dados em geral)
- Representam fluxos em arquivos que podem ser referenciados por um caminho na estrutura de diretórios e um nome de arquivo

Fluxos de bytes

- Um fluxo de entrada de bytes pode ser criado com o construtor
  - **`FileInputStream(String nome)`**
- O argumento **`nome`** deverá ser o nome do arquivo a partir do qual os dados serão lidos
- É possível incluir no argumento o caminho onde se encontra o arquivo
  - permite que o arquivo esteja em uma pasta diferente daquela em que o aplicativo é executado

Fluxos de bytes

- Exemplo: a instrução a seguir cria um fluxo de entrada de bytes a partir do arquivo **scores.dat**
  - **FileInputStream fluxo = new FileInputStream("scores.dat")**

Fluxos de bytes

- Depois que um fluxo de entrada de bytes foi criado, é possível ler dados do fluxo, chamando seu método **`read`**
  - **`int read()`**: retorna o próximo byte no fluxo como um inteiro
  - **`int read(byte[], int offset, int lenght)`**: lê bytes para o array de bytes especificado, com o ponto de partida indicado e número de bytes lidos

Fluxos de bytes

- Se o método **`read`** retornar -1 significa que o final do arquivo foi alcançado
- Terminada a leitura dos dados o fluxo deve ser fechado chamando-se o seu método **`close()`**

Fluxos de bytes

- Um fluxo de saída de bytes pode ser criado com o construtor
  - **`FileOutputStream(String nome)`**

- É possível escrever dados do fluxo, chamando seu método **`write(int)`** ou **`write(byte[], int, int)`**

- Terminada a escrita dos dados o fluxo deve ser fechado chamando-se o seu método **`close()`**

Exemplo – fluxo de bytes

```
import java.io.*;
 public class WriteByteArrayToFile {
     public static void main(String[] args) {
          String strFilePath = "C://FileIO//demo.txt";
         try {
             FileOutputStream fos = new FileOutputStream(strFilePath);
             String strContent = "Write File using Java FileOutputStream example !";
             fos.write(strContent.getBytes());
             fos.close();
         }
         catch(FileNotFoundException ex){
             System.out.println("FileNotFoundException : " + ex);
         }
         catch(IOException ioe) {
             System.out.println("IOException : " + ioe);
         }
     }
}
```

Fluxos de caracteres

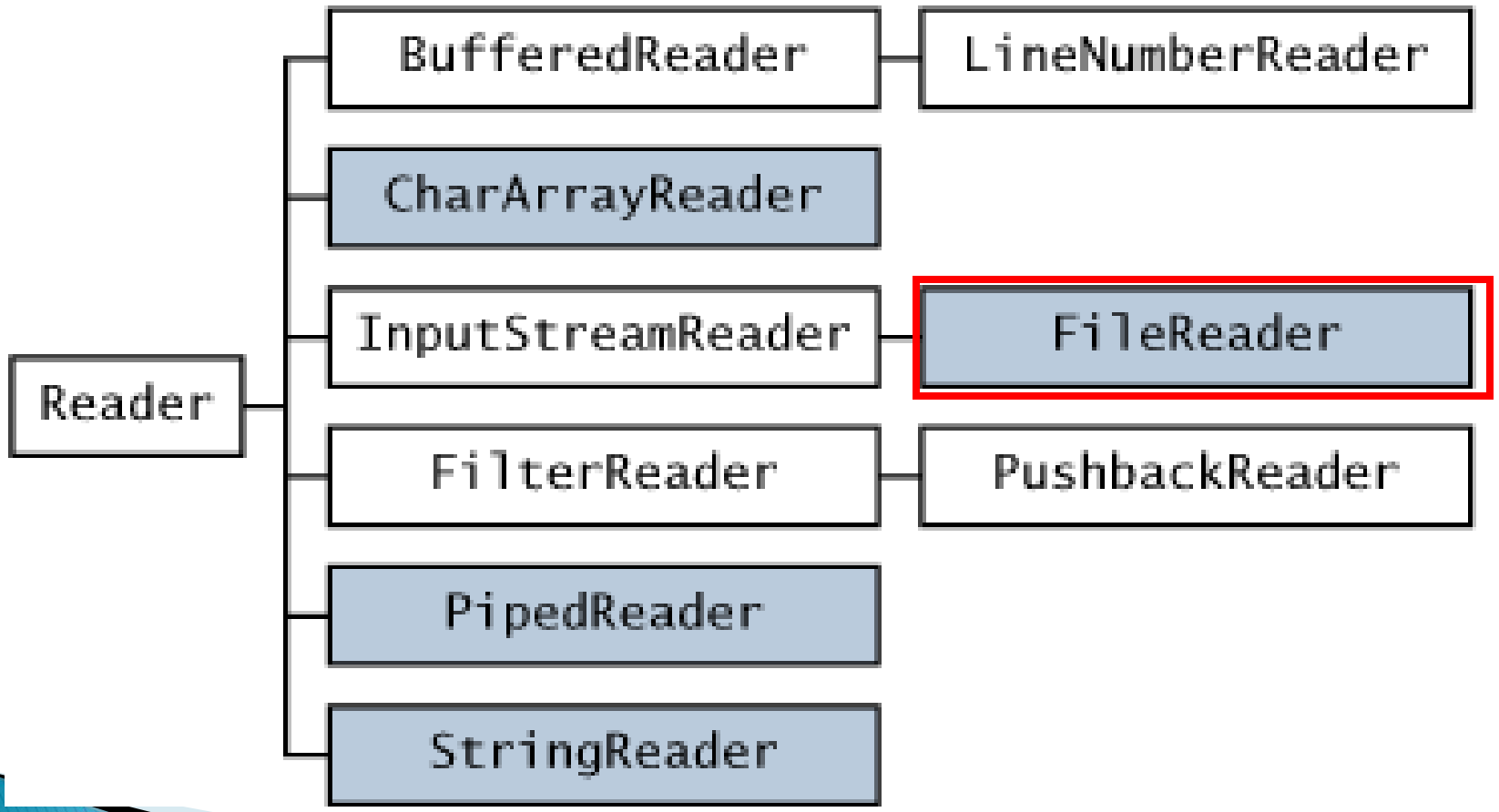
- Usados para lidar com qualquer texto que seja representado pelo conjunto de caracteres Unicode de 16 bits
  - arquivos de texto puro
  - documentos HTML
  - arquivos fontes Java

Fluxos de caracteres

- As classes usadas para ler e escrever fluxos de caracteres são todas derivadas das classes **`Reader`** e **`Writer`**

- **`FileReader`** é a classe principal usada para a leitura de fluxos de caracteres em um arquivo
  - subclasse de **`InputStreamReader`**, que lê um fluxo de bytes e converte os bytes para valores inteiros

Leitura de caracteres (16 bits)



Fluxos de caracteres

- Um fluxo de entrada de caracteres é associado a um nome de arquivo usando o construtor
  - **`FileReader (String nome)`**

- Exemplo: a instrução a seguir cria um fluxo de entrada de caracteres e o associa a um arquivo texto
  - **`FileReader fluxo = new FileReader ("index.txt")`**

Fluxos de caracteres

- Depois que um fluxo de entrada de caracteres foi criado, é possível ler dados do fluxo, chamando seu método **`read`**
  - **`read()`**: retorna o próximo caractere no fluxo como um inteiro
  - **`read(char[], int, int)`**: lê caracteres para o array de caracteres especificado, com o ponto de partida indicado e número de caracteres lidos

Fluxos de caracteres

- Como o método **`read`** retorna um inteiro, é preciso converter esse dado antes de ser
  - exibido
  - armazenado em um array
  - usado para formar uma string, etc.
- O inteiro retornado é um código numérico que representa o caractere no conjunto de caracteres Unicode

Exemplo

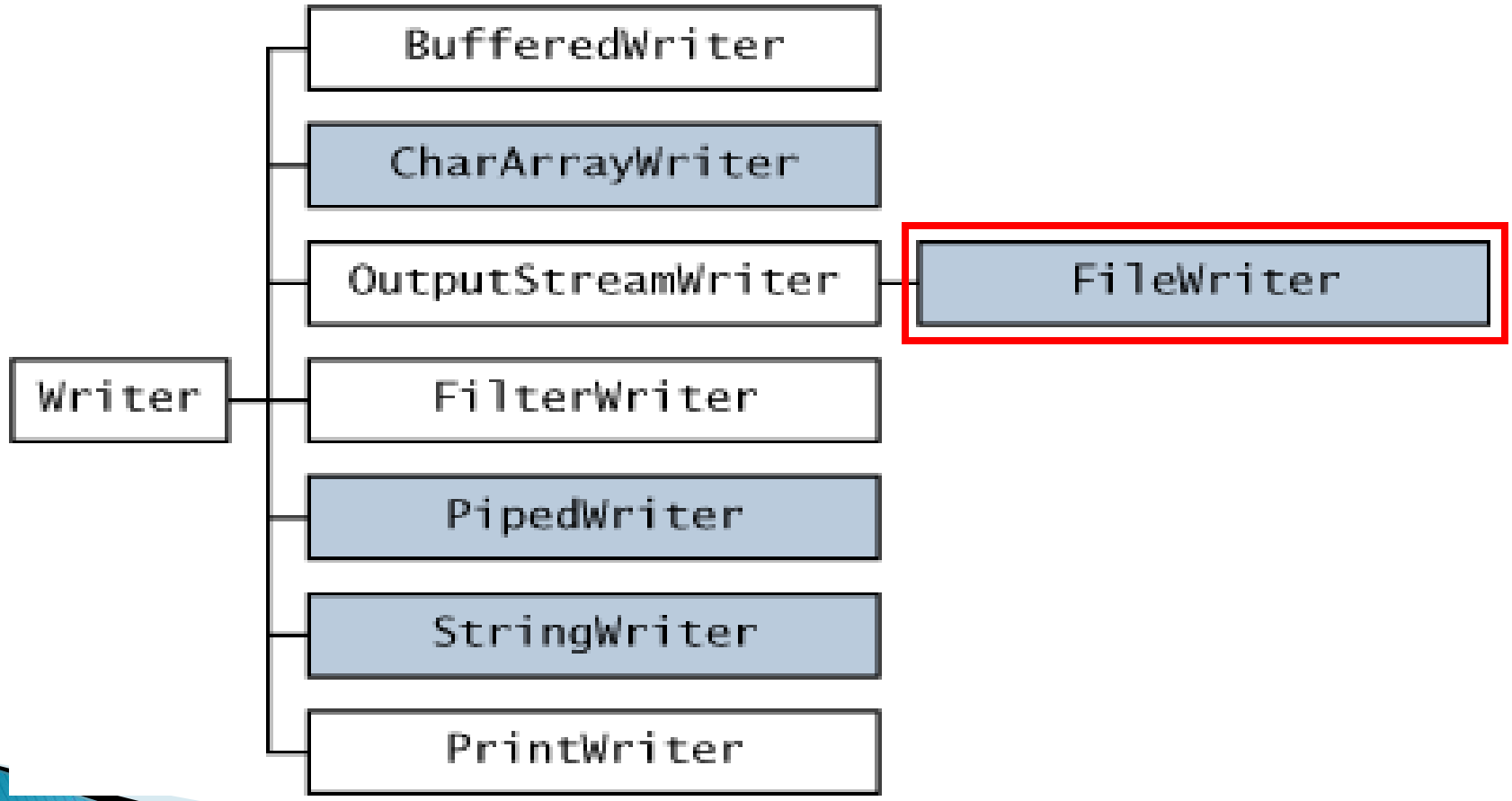
```
import java.io.*;
public class ReadCaracteres{
  public static void main(String[] args){
    int i;
     FileReader entrada = new FileReader("exemplo.txt");

    try{
        while (true){
              i = entrada.read();
              if ( i == -1 ) break;
                 char c = (char) i;
                 System.out.print( c );
              }
      }
    catch (IOException e){
      System.err.println(e);
    }
    System.out.println();
    entrada.close()
 }
}
```

Fluxos de caracteres

- A classe **`FileWriter`** é a classe usada para gravar um fluxo de caracteres em um arquivo
  - subclasse de **`OutputStreamWriter`**, que converte códigos de caractere Unicode em bytes

Escrita de caracteres (16 bits)



Fluxos de caracteres

- Existem dois construtores de **`FileWriter`**
  - **`FileWriter (String nome)`**
  - **`FileWriter (String nome, boolean anexo)`**

- O **`nome`** indica o nome do arquivo ao qual o fluxo de saída será direcionado (pode incluir o caminho)

- O argumento **`anexo`** será **`true`** se o fluxo tiver que ser anexado a um arquivo de texto existente

Fluxos de caracteres

- Três métodos de **`FileWriter`** podem ser usados para gravar dados em um fluxo:
  - **`write(int)`**: grava um caractere
  - **`write(char[], int, int)`**: grava caracteres do array de caracteres especificado, com o ponto de partida indicado e número de caracteres a serem gravados
  - **`write(String, int, int)`**: grava caracteres da string especificada, com o ponto de partida indicado e número de caracteres a serem gravados

Exemplo: cópia de arquivo

```
public static void exemplo() throws IOException {
      File arq_entrada = new File("entrada.txt");
      File arq_saida = new File("saida.txt");

      FileReader entrada = new FileReader(arq_entrada);
      FileWriter saida = new FileWriter(arq_saida);

      int c;
      // -1 indica final de arquivo de caracteres
      while ((c = entrada.read()) != -1)
           saida.write(c);
      entrada.close();
      saida.close();
}
```

Arquivos de dados

- Arquivos de texto não são convenientes para manipulação de dados em geral
- É possível utilizar fluxos de entrada e saída de dados das classes **`DataInputStream`** e **`DataOutputStream`**
- Esses fluxos filtram um fluxo de bytes existente de modo que tipos primitivos (*char*, *int*, *double* etc) possam ser lidos ou escritos

Abertura do fluxo

- Associando um arquivo

```
File arquivo = new File("dados.bin");
```

- Para leitura

```
DataInputStream entrada =
  new DataInputStream(
  new FileInputStream(arquivo));
```

- Para escrita

```
DataOutputStream saida =
  new DataOutputStream(
  new FileOutputStream(arquivo));
```

Leitura, escrita e fechamento

- Leitura (pode gerar *EOFException*)

```
char c = entrada.readChar();
int i = entrada.readInt();
double d = entrada.readDouble();
```

- Escrita

```
saida.writeChar(c);
saida.writeChars(s);
saida.writeInt(i);
saida.writeDouble(d);
```

- Fechamento

```
entrada.close();
saida.close();
```

Exemplo: pedido de compra

- Considere a construção de um arquivo com dados (binários) em forma tabular:

preco	quantidade	descrição
10.00	*12*	*mouse óptico*
82.34	*24*	*teclado*
26.50	*6*	*leitor cd-rom*

- Os dados estão armazenados em arrays

```
double precos[]
int quantidades[]
string descricoes[]
```

Criação da tabela

```
File arquivo = new File("precos.bin");

DataOutputStream saida = new DataOutputStream(
new FileOutputStream(arquivo));

for (int i = 0; i < precos.length; i ++) {

    saida.writeDouble(precos[i]);

    saida.writeChar('\t');

    saida.writeInt(quantidades[i]);

    saida.writeChar('\t');

    saida.writeChars(descricoes[i]);

    saida.writeChar('\n');

}

saida.close();
```

Leitura da tabela

```
DataInputStream entrada = new DataInputStream(
new FileInputStream("precos.bin"));
try {
    while (true) {
            preço = entrada.readDouble();
            entrada.readChar();        // despreza o tab
            quantidade = entrada.readInt();
            entrada.readChar();        // despreza o tab
            // etc…
    }
} catch (EOFException e) { // fim de arquivo }
entrada.close();
```

Leitura da tabela

```
DataInputStream entrada = new DataInputStream(
                          new FileInputStream("precos.bin"));
try {
while (true) {
            preço = entrada.readDouble();
            entrada.readChar();       // despreza o tab
            quantidade = entrada.readInt();
            entrada.readChar();       // despreza o tab
            // etc…
}
} catch (EOFException e) { …de arquivo }
entrada.close();
```

Não exite **readChars**! Deve-se ler um caractere por vez em um loop