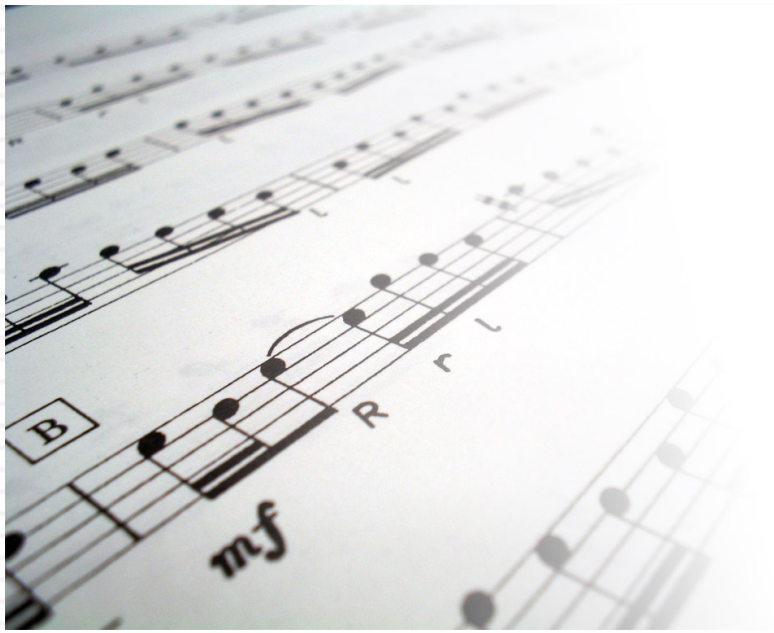
PROGRAMAÇÃO ORIENTADA PELOS OBJECTOS

Aulas 26-27

Streams I/O

Playlist em ficheiro



Relembrando a aplicação PlayList

- A aplicação PlayList permite gerir as músicas favoritas
  - As operações fundamentais são:
    - Adicionar músicas
    - Remover músicas
    - Classificar músicas
    - Pesquisar músicas
    - Listar músicas
    - Obter informações sobre as músicas

Agora queremos guardar num suporte físico

- Para além das operações anteriores, pretendemos também guardar a informação da *playlist* em ficheiro e claro, poder ler mais tarde, alterar a *playlist* e guardar novamente
  - Imaginem por exemplo uma lista com dezenas de músicas !!!
- Aproveitamos a oportunidade para guardar a informação de forma diversa
  - Num simples ficheiro de texto
  - Num ficheiro binário, o que desde logo reduz a tentação de alterar directamente a informação com um editor de texto em detrimento da aplicação
    - A *playlist* pode ser considerada uma sequência de registos de valores
    - A *playlist* pode também ser vista como um objecto Java

5

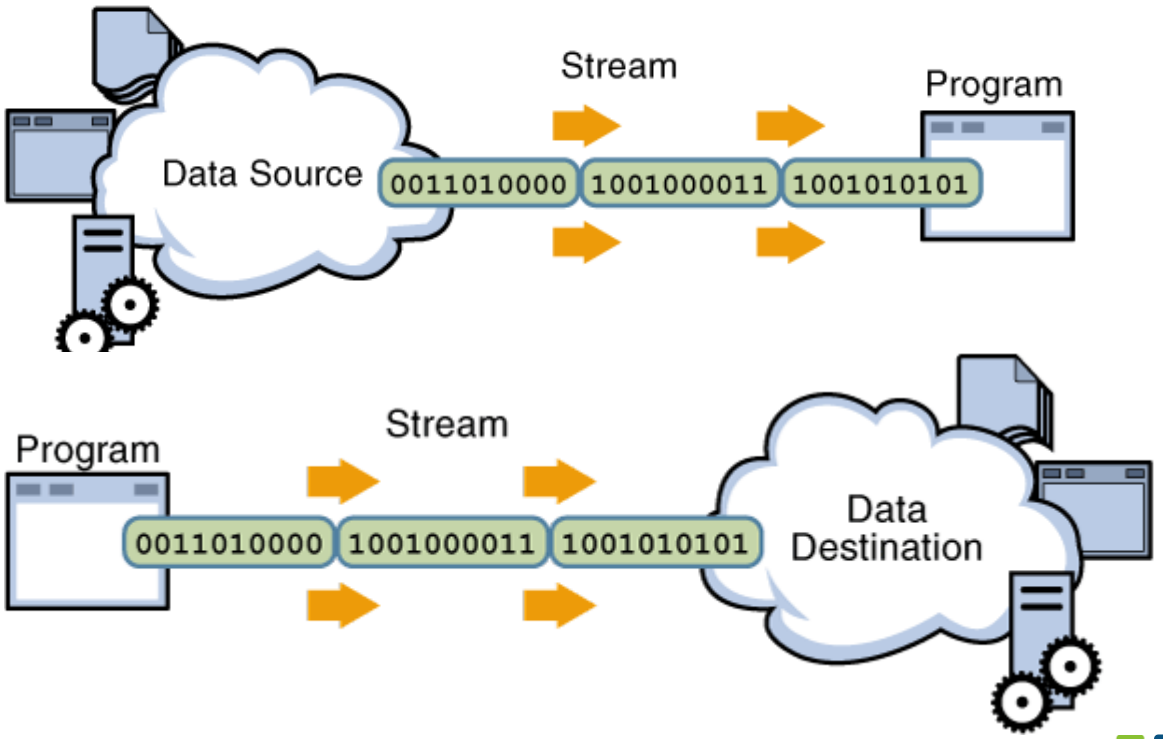
Streams de Input/Output

Streams

- Em geral, as tarefas associadas a leitura e a escrita de dados têm subjacente o conceito de stream
- Uma stream é uma abstracção, que representa uma fonte genérica de entrada de dados ou um destino genérico para escrita de dados, de acesso sequencial e independente dos dispositivos físicos relacionados, formatos ou técnicas de leitura/escrita
- Dispositivos físicos típicos
  - Ficheiros em disco, CDs/DVDs, programas, bancos de memória, sockets de rede, impressoras, etc.
- Tipos de dados a suportar
  - Bytes, caracteres, tipos primitivos de dados, objectos

Modelo implícito

- Stream enquanto sequência de dados, com a qual se pode realizar uma das seguintes operações fundamentais:
  - Leitura: ler elemento a elemento, de uma fonte para o programa
  - Escrita: escrever elemento a elemento, do programa para um destino



Padrão de funcionamento

- As operações de leitura e de escrita sobre uma stream seguem quase sempre o mesmo padrão de funcionamento

```
try {
   abrir a stream
   try {
     ler da stream
     enquanto não chegar ao fim
         processar a informação
         ler da stream
   }
   finally { fechar a stream }
}
catch (IOException exception) {
    ...
}
```

```
try {
   abrir a stream
   try {
     processar a informação
     enquanto não chegar ao fim
         escrever na stream
         processar a informação
   }
   finally { fechar a stream }
}
catch (IOException exception) {
    ...
}
```

Tipos de streams

- Numa primeira abordagem, as streams podem ser categorizadas em dois grandes tipos
  - Streams de bytes, ou seja, streams binárias
  - Streams de caracteres (2 bytes em código UTF), ou seja, streams de texto
- Numa outra perspectiva, temos de considerar a existência de
  - Streams lógicas, adequadas para o nível de programação
  - Streams físicas, adequadas para ser fonte ou destino de dados
- Esta última perspectiva sugere um encapsulamento implícito das streams

Encapsulamento de streams

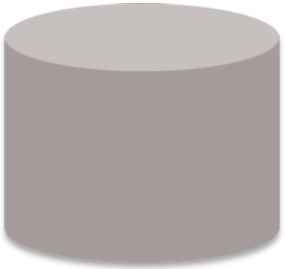
- O encapsulamento (*wrapping*) de streams faz a distinção entre stream lógica e stream física
- Deste modo, usando um protocolo de programação de alto nível, podemos relegar para segundo plano inúmeros detalhes de implementação inerentes à obtenção dos dados na fonte e a sua transmissão efectiva para o destino
- A maioria dos construtores de streams do Java apresenta uma declaração encapsulada de streams
  - Exemplo:

```
public BufferedWriter(Writer out)
```

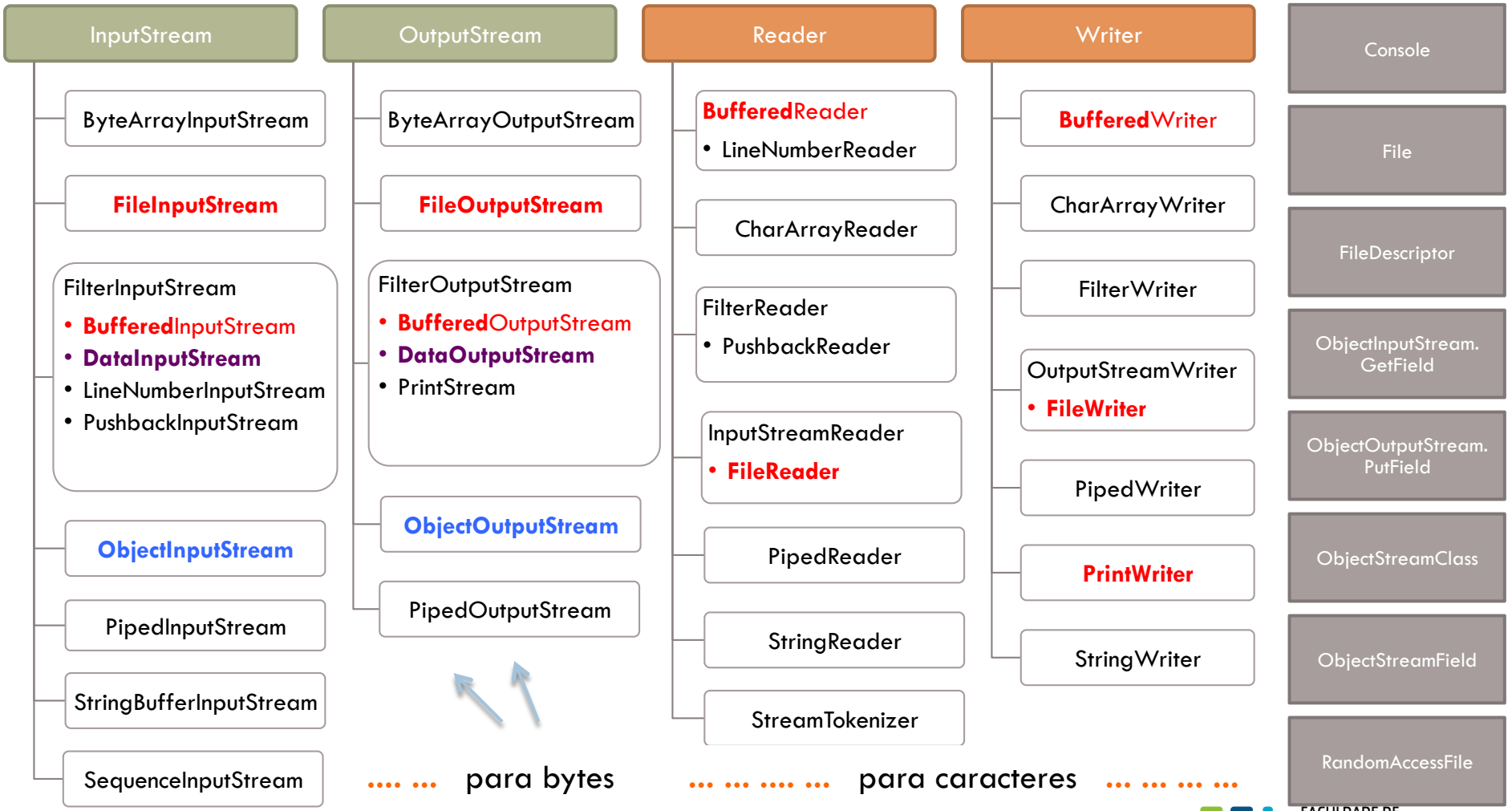
Programa

BufferedWriter → FileWriter

Dispositivo físico



Hierarquia de classes em java.io



Algumas classes I/O de interesse imediato

- Leitura
  - **`BufferedReader` e `FileReader`**
  - **`DataInputStream`**
  - **`ObjectInputStream`**
  - **`Scanner`**
- Escrita
  - **`BufferedWriter` e `FileWriter`**
  - **`PrintWriter`**
  - **`DataOutputStream`**
  - **`ObjectOutputStream`**

13

I/O básico

Streams standard em java.lang.System

- As streams standard são intrínsecas ao sistema operativo
  - Usualmente permitem ler do teclado e escrever no ecrã
  - Suportam através do interpretador da linha de comandos operações I/O de ficheiros e entre programas
- Em Java temos três streams standard, de bytes por razões históricas, as quais estão automaticamente abertas
  - System.in (Standard Input)
  - System.out (Standard Output)
  - System.err (Standard Error)
- Exemplos de utilização

```
System.out.print("texto");
System.out.println("texto");
System.out.format("Quadrado de %2d = %6.2f", x, x*x);
int umByte = System.in.read();
```

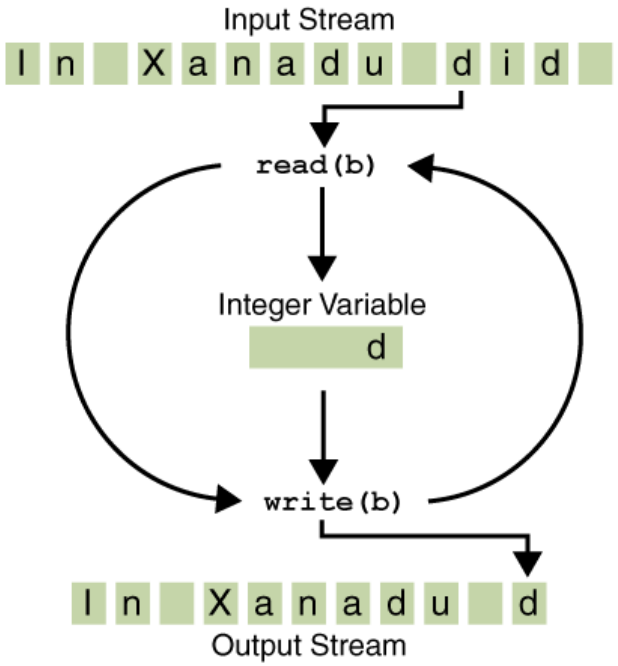
Streams de bytes

- As streams de bytes visam a leitura/escrita de bytes, o que significa que são operações I/O de baixo nível
  - Em Java, as streams derivam das classes **`InputStream`** e **`OutputStream`**, como é o caso particular de streams para ficheiros, **`FileInputStream`** e **`FileOutputStream`**
- Sendo operações I/O de baixo nível
  - Todos os outros tipos de streams recorrem a estas streams de bytes
  - Só devem ser utilizados para as operações I/O mais primitivas, o que quer dizer que sempre que for possível devemos considerar outro tipo de streams
    - Por exemplo, se queremos escrever um ficheiro de texto, devemos usar streams orientados ao carácter e não ao byte

Exemplo com streams de bytes


```
private static void testIOBytes() {

   try {
     FileInputStream inStream = null;
     FileOutputStream outStream = null;
     inStream = new FileInputStream("entrada.txt");
     outStream = new FileOutputStream("bytesaida.txt");
     try {
       int c;
       while ((c = inStream.read()) != -1) {
         outStream.write(c);
       }
     }
     finally {
       if (inStream != null) { inStream.close(); }
       if (outStream != null) {
           outStream.flush();
           outStream.close();
       }
     }
   }
   catch (IOException excep) {
     System.out.println(excep.getMessage());
   }
}
```



`read()` de bytes e devolve um int?
Porque também devolve -1 para indicar quando chega ao fim da stream

Streams de caracteres

- Os caracteres em Java são armazenados segundo as convenções Unicode
  - Nas definições ocidentais, o conjunto de caracteres é normalmente codificado com ASCII 8 bits
  - Quaisquer dados são visualizáveis como texto
- As classes para streams de caracteres ajustam-se ao conjunto de caracteres local pré-definido pelo programa, procedendo a uma tradução automática
- Todas as classes derivam de **`Reader`** e **`Writer`**
  - Existem as classes especializadas para ficheiros, **`FileReader`** e **`FileWriter`**

Exemplo com streams de caracteres


```
private static void testIOChars() {
    try {
      FileReader inStream = null;
      FileWriter outStream = null;
      inStream = new FileReader("entrada.txt");
      outStream = new FileWriter("charsaida.txt");
      try {
        int c;
        while ((c = inStream.read()) != -1) {
          outStream.write(c);
        }
      }
      finally {
        if (inStream != null) { inStream.close(); }
        if (outStream != null) {
           outStream.flush();
           outStream.close();
        }
      }
    }
    catch (IOException excep) {
      System.out.println(excep.getMessage());
    }
}
```

19

Streams encapsuladas com buffers

Streams encapsuladas com buffers

- As streams com buffers visam dar resposta a um problema patente na manipulação de bytes ou caracteres per si, pois neste caso cada pedido de leitura/escrita é processado imediatamente pelo sistema operativo, tornando o processo ainda mais ineficiente
  - Exemplo: maior número de acessos ao disco, que já é uma operação lenta
  - Auxiliam as streams normais, criando buffers de capacidade optimizada em função do sistema operativo e da configuração, que permitem operar de uma só vez um grupo de bytes ou caracteres
- Operação de leitura
  - A stream com buffer faz a leitura a partir de uma área da memória – o buffer de leitura
  - Só quando o buffer está vazio é que a stream de leitura de baixo nível é invocada
- Operação de escrita
  - A stream com buffer escreve numa área da memória – o buffer de escrita
  - Só quando o buffer está cheio é que a stream de escrita de baixo nível é invocada

Streams encapsuladas com buffers

- Por vezes, pode ser conveniente não esperar pelo buffer de escrita cheio para desencadear a chamada da stream de baixo nível
  - A invocação deste procedimento pode ser automático (em alguns casos) ou manual
  - O processo de *flushing* manual do buffer é feito através da invocação do método **`flush()`** no objecto da stream em causa
- Existem classes de stream de buffers para encapsular streams de bytes sem buffers e para encapsular streams de caracteres sem buffers
  - Para bytes, **`BufferedInputStream`** e **`BufferedOutputStream`**
  - Para caracteres, **`BufferedReader`** e **`BufferedWriter`**

Linhas de caracteres

- É bastante usual proceder a leitura/escrita de caracteres por grupo, ou seja ler e escrever um grupo alargado de caracteres
  - Esta situação implicará globalmente um número menor de chamadas ao sistema operativo e à respectiva componente de I/O
- Uma unidade de medida para um grupo de caracteres é a linha, ou seja, uma string de caracteres com um terminador de linha no final
  - Um terminador de linha pode ser um *carriage return* *CR* (13), um *line feed* *LF* (10) ou ainda uma sequência destes, *CR+LF*
    - Exemplo particular de um teste que determina a leitura de um *CR* e consequente limpeza do buffer, para não afectar a leitura posterior de caracteres

```
int val = System.in.read();
if ( (char) val == 13) {
   val = System.in.read();  // limpa o LF do buffer
}
```

  - Se todos os terminadores forem suportados por um programa, então poderemos ler/escrever ficheiros de texto em vários sistemas operativos

Exemplo de linhas de caracteres com buffers


```
private static void testIOLines() {

    try {
      BufferedReader inStream = null;
      PrintWriter outStream = null;
      inStream = new BufferedReader( new FileReader("entrada.txt"));
      outStream = new PrintWriter( new FileWriter("linhasaida.txt"));
      try {
        String line;
        while ((line = inStream.readLine()) != null) {
          // Obs. adiciona o terminador de linha do S.O.
          outStream.println(line);
        }
      }
      finally {
        if (inStream != null) { inStream.close(); }
        if (outStream != null) {
            outStream.flush();
            outStream.close();
        }
      }
    }
    catch (IOException excep) {
      System.out.println(excep.getMessage());
    }
}
```

Aplicação típica das streams com buffers, para leitura/ escrita agrupada de caracteres – uma linha neste caso

Streams de dados

- A utilização de streams de dados apresenta três características fundamentais
  - Suporte de operações I/O para tipos primitivos de dados – *boolean*, *char*, *byte*, *short*, *int*, *long*, *float* e *double* – bem com strings
  - Pressupõe o recurso a buffers, ou seja, é patente o encapsulamento de streams
- As classes associadas implementam as interfaces **`DataInput`** e **`DataOutput`**, sendo as mais importantes as classes **`DataInputStream`** e **`DataOutputStream`**
- Útil por exemplo quando se pretender executar operações I/O para registos de valores de dados
- Não é muito adequado para números fraccionários devido à utilização de aritmética de vírgula flutuante
  - A existência de problemas de precisão é frequente
  - Existem algumas soluções que mitigam este problema

Streams de dados

- As streams de dados detectam a condição de fim- de-ficheiro apenas através do mecanismo de excepção `EOFException`, o que não acontece com as streams de bytes ou de caracteres, que podem recorrer a códigos de retorno
- Qualquer processo de escrita tem implicitamente associado o correspondente processo de leitura
  - Note-se que uma stream de leitura são dados binários simples, sem nenhuma indicação sobre os valores associados a esses mesmos dados nem indicações sobre o seu início

Stream de objectos

- As streams de objectos suportam operações I/O de instâncias de classes que implementem a interface **`Serializable`**
- A interface **`Serializable`** não impõe nenhuma implementação, ou seja, pretende apenas indicar a concordância com uma propriedade
- São muito eficientes e muito simples de usar
  - É possível escrever objectos complexos em ficheiro através de um método que implementa um algoritmo de serialização, o qual garante que todas as referências cruzadas existentes entre objectos são devidamente repostas mais tarde aquando da leitura posterior do ficheiro para a memória
  - Basta uma única chamada para ter uma operação I/O relativa a um objecto complexo que contenha referências a outros objectos, e estes a outros e assim sucessivamente
    - Significa que uma operação I/O de um objecto complexo pode desencadear automaticamente várias operações de I/O, para vários objectos

Stream de objectos

- As classes principais de streams de objectos são **`ObjectInputStream`** e **`ObjectOutputStream`**, que implementam respectivamente as interfaces **`ObjectInput`** e **`ObjectOutput`**
  - Note-se que estas interfaces estendem as interfaces **`DataInput`** e **`DataOutput`**, estas já referidas aquando das streams de dados
  - As stream de objectos são na prática uma extensão das streams de dados
- O ficheiro de escrita não é visualizável com um editor de texto
- Existe uma restrição à escrita de variáveis de classe com o atributo **`static`**

Stream de objectos

- Cada escrita de um objecto deve corresponder a uma leitura do objecto
  - Se a escrita de dois objectos na mesma stream contém referências a um único objecto, a leitura posterior vai manter a mesma referência ao objecto único
  - Uma stream só pode conter uma cópia do objecto, mas pode ter várias referências a ele
  - Se um único objecto é escrito em duas streams distintas, então a leitura posterior das streams originará dois objectos distintos, ou seja, temos uma duplicação de objectos

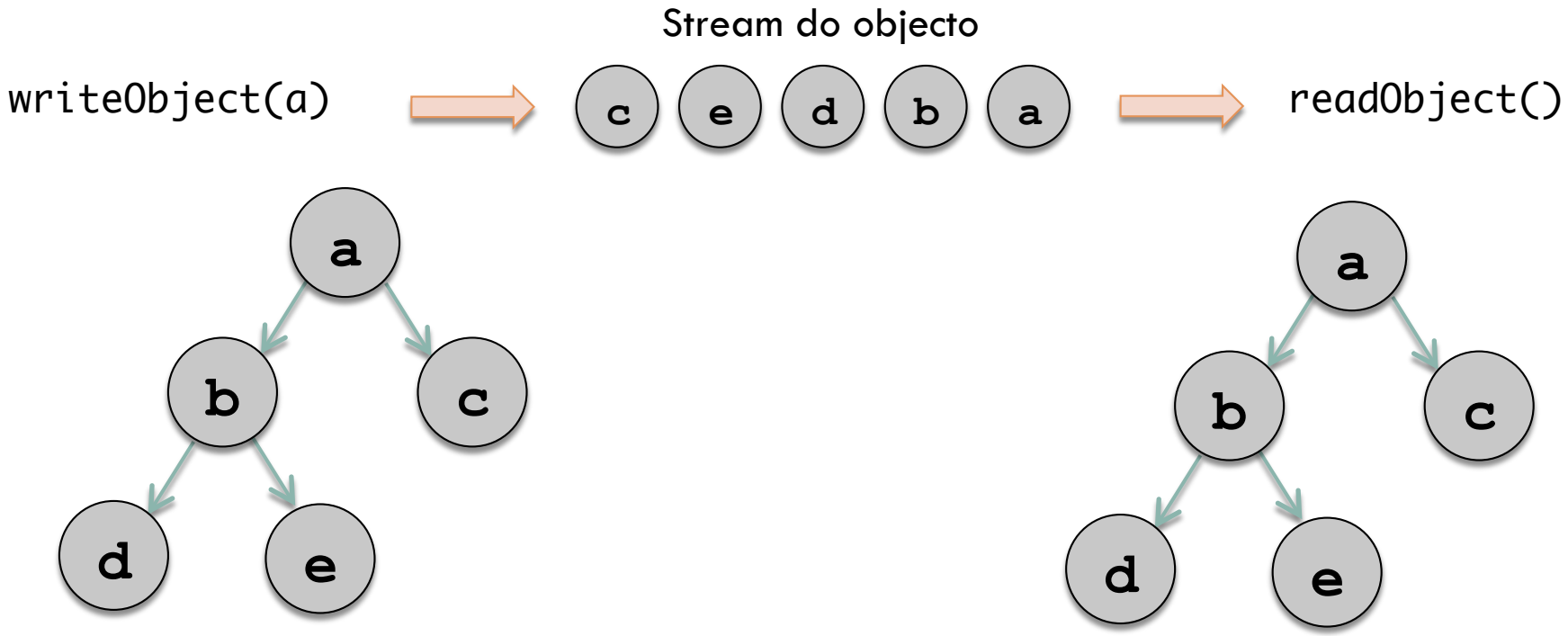
```
Object ob = new Object();
out.writeObject(ob);
out.writeObject(ob);
```

Duas variáveis que referem um único objecto

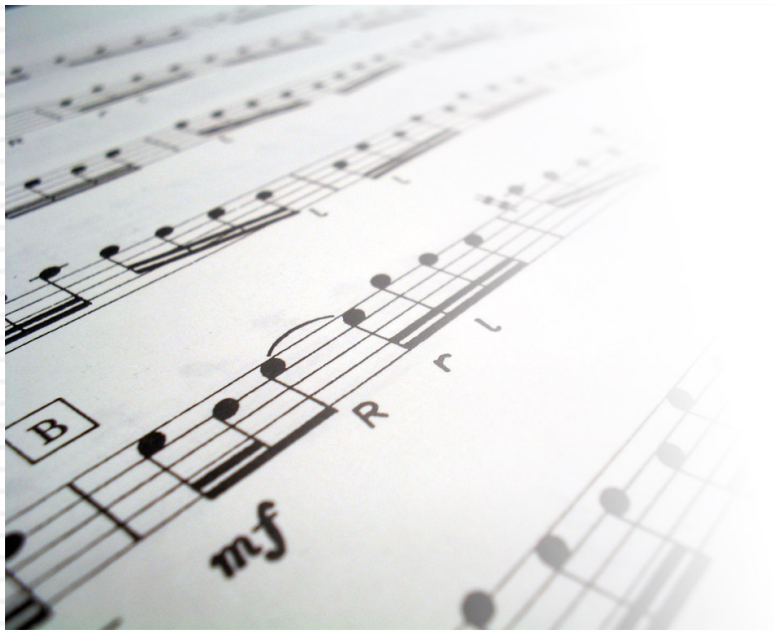
```
Object ob1 = in.readObject();
Object ob2 = in.readObject();
```

Stream de objectos

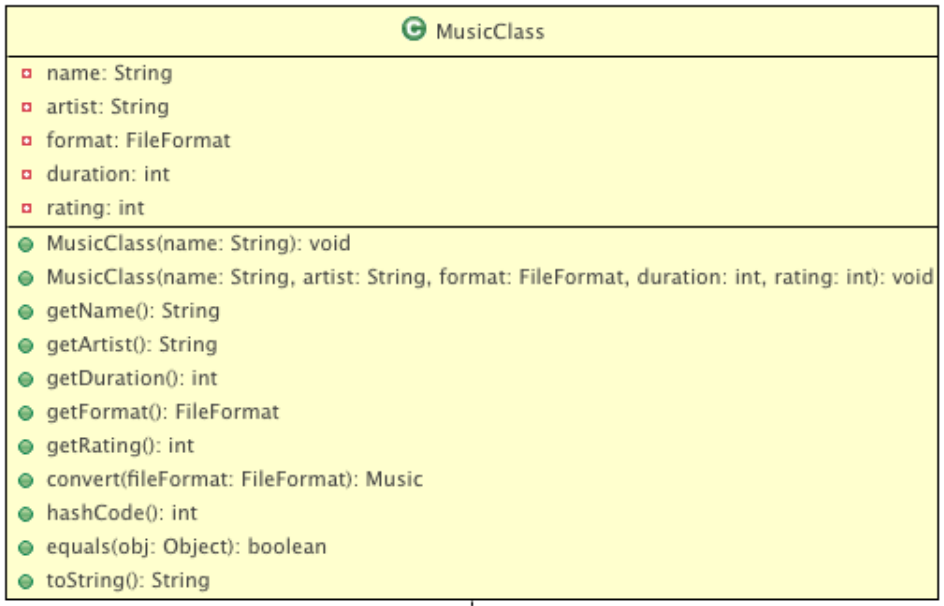
- Exemplo: objecto **a** contém referências ao objecto **b** e ao objecto **c**; objecto **b** contém referências ao objecto **d** e ao objecto **e**

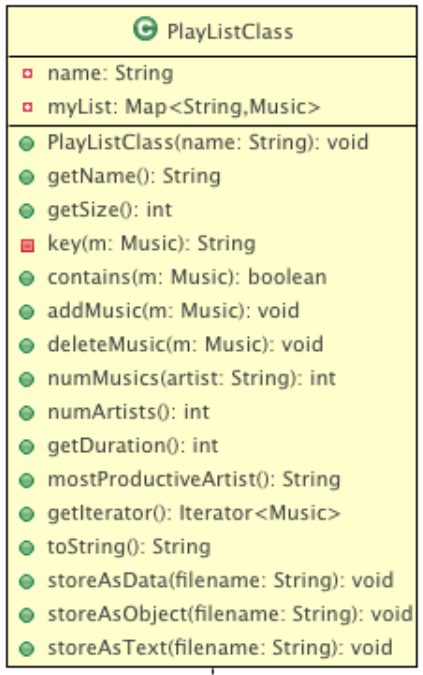


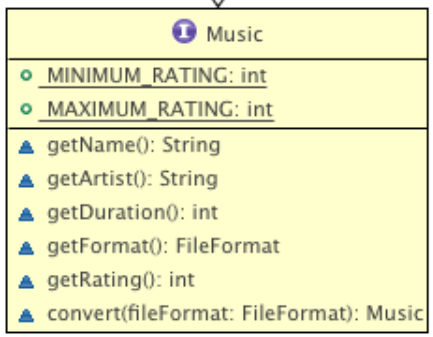
Voltando à aplicação PlayList

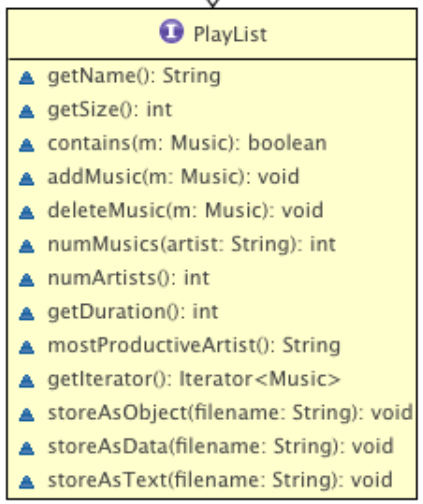


PlayList e I/O









- Em termos de I/O vamos implementar três metodologias, recorrendo a
  - Ficheiros de texto
  - Streams de dados
  - Streams de objectos

Escrita no ficheiro de texto


```
public void storeAsText(String filename) throws IOException {

    PrintWriter outStream = new PrintWriter(filename);
    try {
      outStream.println(this.getName());
      for (Music m : this.myList.values()) {
        outStream.print(m.getName() + "\t");
        outStream.print(m.getArtist() + "\t");
        outStream.print(m.getFormat().toString() + "\t");
        outStream.print(m.getDuration() + "\t");
        outStream.println(m.getRating());
      }
    }
    finally {
      outStream.flush();
      outStream.close();
    }
}
```


```
Canções da minha vida
Ser Benfiquista       Luis Picarra          AIFF       184        2
Eu tenho dois amores  Marco Paulo           MP3        194        3
Master of Puppets     Metallica             WAV        452        4
. . .
```

Leitura do ficheiro de texto


```
private static PlayList readAsText(String filename)
                                        throws IOException {

    PlayList play = null;
    Scanner in = new Scanner( new FileReader (filename) );
    // should use default line separator from JVM
    in.useDelimiter(System.getProperty("line.separator"));
    try {
      String line = in.next();
      play = new PlayListClass(line);
      while (in.hasNext()) {
        line = in.next();
        Music m = parseMusicLine(line);
        play.addMusic(m);
        m = null;
      }
    }
    finally {
      in.close();
    }
    return play;
}
```

Escrita com stream de dados


```
public void storeAsData(String filename) throws IOException {

    // stream as a wrapper for a byte stream
    DataOutputStream outStream = new DataOutputStream(new
        BufferedOutputStream(new FileOutputStream(filename)));
    try {
      outStream.writeUTF(this.getName());
      for (Music m : this.myList.values()) {
        outStream.writeUTF(m.getName());
        outStream.writeUTF(m.getArtist());
        outStream.writeUTF(m.getFormat().toString());
        outStream.writeInt(m.getDuration());
        outStream.writeInt(m.getRating());
      }
    }
    finally {
      outStream.flush();
      outStream.close();
    }
}
```

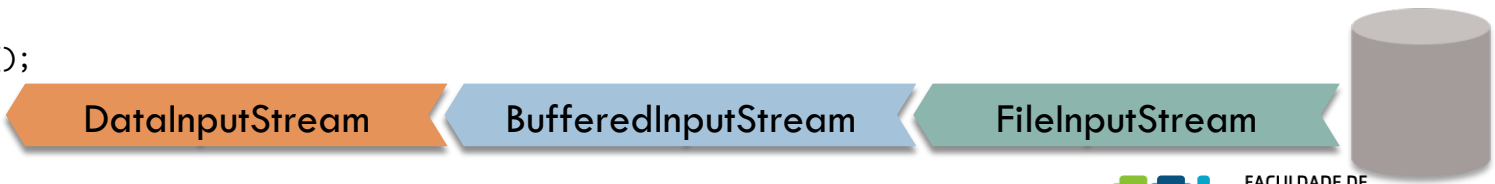
Eventualmente os dados escritos com **`writeUTF()`**, isto é, como caracteres, podem ser perceptíveis no ficheiro

Leitura com stream de dados


```
private static PlayList readAsData(String filename)
            throws IOException, EOFException {

  PlayList play = null;
  DataInputStream inStream = new DataInputStream(new
      BufferedInputStream(new FileInputStream(filename)));
  try {
    String playName = inStream.readUTF();
    play = new PlayListClass(playName);
    try {
      // dataStreams detects EOF by catching EOFException
      while (true) {
        String musicName = inStream.readUTF();
        String artist = inStream.readUTF();
        String format = inStream.readUTF();
        int duration = inStream.readInt();
        int rating = inStream.readInt();
        Music m = new MusicClass(musicName, artist,
            FileFormat.valueOf(format),duration,rating);
        play.addMusic(m);
        m = null;
      }
    }
    catch (EOFException e) { }
  }
  finally {
    inStream.close();
  }
  return play;
}
```

A implementação não é simples !!



Escrita com stream de objectos


```
public void storeAsObject(String filename) throws IOException {

    ObjectOutputStream outStream = new ObjectOutputStream(
            new FileOutputStream(filename));
    try {
      outStream.writeObject(this);
    }
    finally {
      outStream.flush();
      outStream.close();
    }
}
```

Simples !!

Leitura com stream de objectos


```
private static PlayList readAsObject(String filename)
            throws IOException, ClassNotFoundException {

    PlayList play = null;
    ObjectInputStream inStream = new ObjectInputStream(
                              new FileInputStream(filename));
    try {
      play = (PlayList) inStream.readObject();
    }
    finally {
      inStream.close();
    }
    return play;
}
```

Cast obrigatório, razão pela qual o método pode lançar a excepção **`ClassNotFoundException`**

Hierarquia de classes em java.io