estgoh
Politécnico de Coimbra

Programação
Programação III

Ficheiros

Marco Veloso
marco.veloso@estgoh.ipc.pt

Agenda

Ficheiros

Introdução à Linguagem Java
- Paradigmas de Programação
- Linguagem Java

Programação Orientada a Objectos
- Objectos
- Classes
- Heranças
- Polimorfismo

Tratamento de Excepções

Estruturas de dados
- Tabelas unidimensionais
- Tabelas multidimensionais
- Vectores
- Dicionários (Hashtables)
- Collections

Ficheiros
- Manipulação do sistema de ficheiros
- Ficheiros de Texto
- Ficheiros Binários
- Ficheiros de Objectos
- Leitura de dados do dispositivo de entrada

Os **objectos** e tipos básicos até agora utilizados têm uma característica comum: são **armazenadas na memória central** do computador

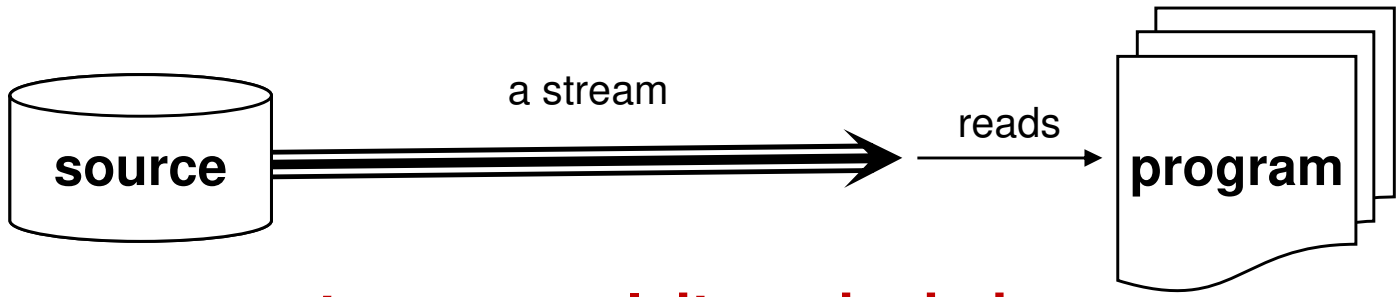
Isto significa que **só existem durante a execução do programa**, pelo que os dados que armazenam apenas estão acessíveis durante esse período

Esta situação é inaceitável em muitas situações

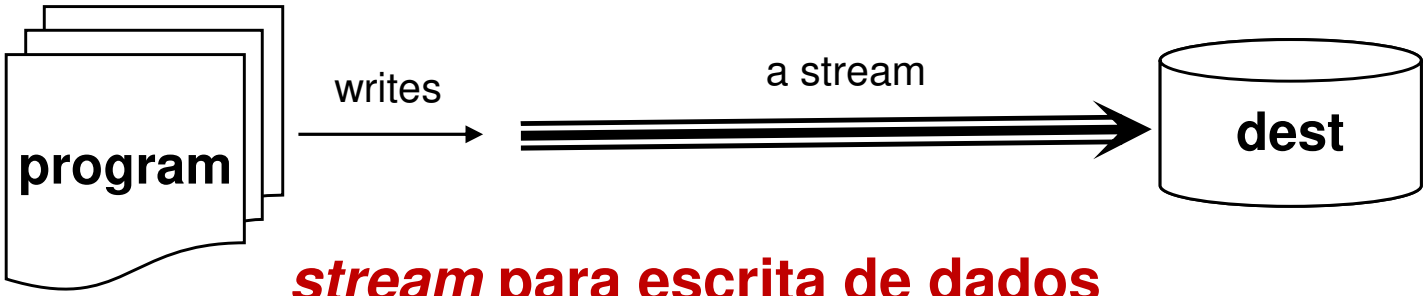
É necessário dispor e utilizar **dispositivos de armazenamento fixo e persistente** (por exemplo discos rígidos) que **mantenham a informação (em ficheiros) para além do final da execução do programa** que a manipula.

Fluxo de dados

Ficheiros



***stream* para leitura de dados**



***stream* para escrita de dados**

Ficheiros binários e ficheiros de texto

Existem dois tipos de ficheiros que normalmente são utilizados: ***ficheiros de texto*** e ***ficheiros binários***.

Os **ficheiros de texto** **contêm caracteres ASCII**, sendo utilizados sempre que é necessário dar ao utilizador a possibilidade de ler o ficheiro usando um editor de texto normal.

Os **ficheiros binários** **contêm informação que está organizada de uma forma normalmente não compreensível para o utilizador (*bytes*)**, mas que um programa entende.

Suponha que é necessário armazenar um ponto (coordenada *x* e *y*) num ficheiro, sendo esse ponto (13, 5). Caso se utilize um ficheiro de texto, são armazenados os caracteres ASCII correspondentes a cada um dos dígitos no ponto (caracter '1', caracter '3', caracter "espaço", caracter '5'). Caso se utilize um ficheiro binário, é armazenado o valor de cada coordenada (um byte com 13 e um byte com 5).

Para ler e escrever **ficheiros de texto** utilizam-se um conjunto de classes denominadas ***Readers*/*Writers***. Para se ler e escrever **ficheiros binários** normalmente utilizam-se ***Streams*** (na verdade os *Readers*/*Writers* são *streams* pensadas para trabalhar com ficheiros de texto).

Manipulação de Ficheiros

Classe de modelação de ficheiros

O Java inclui diversas classes destinadas a **manipular ficheiros** (incluídas na *package* **`java.io`**)

A classe **`File`** serve para **modelar ficheiros em disco**

Para criar um objecto File:

```
File f = new File (nomeDoFicheiro);
```

O `nomeDoFicheiro` deve incluir toda a estrutura de directórios necessária para o localizar (e.g. '`c:/tmp/file.txt`'), caso contrário o ficheiro será acedido a partir da **directoria de trabalho**

A criação de um objecto **`File`** não garante a criação de um ficheiro correspondente em disco, **serve apenas para o representar logicamente**

Se o ficheiro correspondente existir, o objecto **`File`** fornece alguns métodos para o manipular

Manipulação de ficheiros

Ficheiros

Método	Funcionalidade
`getName()`	Obtém o nome do ficheiro
`delete()`	Apaga o ficheiro.
`exists()`	Verifica se o ficheiro existe em disco.
`getPath()`	Obtém o caminho (*path*) completo para aceder ao ficheiro.
`isDirectory()`	Verifica se a entidade representada pelo objecto é uma directoria.
`isFile()`	Verifica se a entidade representada pelo objecto é um ficheiro regular.
`length()`	Obtém o tamanho do ficheiro.
`list()`	Caso o objecto seja uma directoria, lista os ficheiros presentes nesta.
`mkdir()`	Cria uma directoria.
`renameTo()`	Altera o nome do ficheiro.

Um dos métodos permite verificar se um dado **ficheiro** existe:

```
File f = new File("c:/Exemplos/nomeFicheiro.txt");
if (f.exists()) {
      System.out.print("Ficheiro existe");
} else {
      System.out.print("Ficheiro não existe");
}
```

Outro permite apagar um ficheiro:

```
File f = new File ("nomeFicheiro.txt");
f.delete();
```

Esta classe **não possui métodos para criar um ficheiro ou para ler/escrever de/para um ficheiro**

Um objecto do tipo **`File`** pode estar associado a uma **directoria**:

```
File pasta = new File("D://ESTGOH/aulas");

// listar ficheiros de uma directoria
String[] ficheiros = pasta.list();
if(ficheiros == null || ficheiros.isFile())
   // "pasta" não existe ou não é directoria
   System.out.println("há problema com a directoria indicada...");
else
   for(int i=0; i<ficheiros.length; i++){
     // determinar o nome dos ficheiros ou das directorias
     System.out.println(ficheiros[i].getName());
   }
```

Para distinguir se um objecto **`File`** é um **ficheiro ou uma directoria** podem usar-se os seguintes métodos: **`isFile()`** e **`isDirectory()`**

Acesso universal a ficheiros

Para manipular ficheiros independentemente do sistema, e da sua localização (por exemplo para incluir ficheiros em ficheiros JAR), é necessário recorrer à propriedade **`class.getResource()`**

Esta pode ser formada de duas formas:

```
getClass().getResource("/images/logo.png");

NomeDaClasse.class.getResource("/images/logo.png");
```

Neste caso o compilador vai procurar os ficheiros (imagens) à directoria `/images` que deverá ser criada na directoria `/bin` ou `/src` do projecto (em eclipse) e não na raiz do projecto

É assim possível partilhar/incluir projectos e recursos (ficheiros, como imagens) independentemente do sistema (ou através de JARs)

Atenção que o nome da directoria começa com o símbolo "/", ou seja "**`/images/logo.png`**" e não "`images/logo.png`"

Acesso universal a ficheiros

A propriedade `class.getResource()` devolve um objecto `java.net.URL` (*uniform resource locator*) que representa o endereço do recurso (ficheiro)

```
java.net.URL url = MainApplication.class.getResource(aFileName);
java.net.URL url = this.getClass().getResource(aFileName);
```

Em alternativa podemos recorrer à propriedade `class.getClassLoader().getResourceAsStream()` que devolve um objecto do tipo `java.io.InputStream`, que representa um *streamer* para o recurso (ficheiro)

```
java.io.InputStream is =
                MainApplication.class.getClassLoader().getResourceAsStream(aFileName);
java.io.InputStream is =
                this.getClass().getClassLoader().getResourceAsStream(aFileName);
```

Ambos os recursos podem ser usados para aceder a um ficheiro independentemente do sistema

```
File file = new File(url.toString());
```

Ficheiros de Texto

Leitura e escrita de caracteres em ficheiros

Para que isso seja possível (ou para reinicializar um ficheiro já existente) é necessário **estabelecer um caminho de saída dirigido para esse ficheiro**

No caso dos **ficheiros de texto**, podem ser utilizadas as classes `FileReader` ou `FileWriter`, consoante a operação desejada seja de **leitura** (*reader*) ou de **escrita** (*writer*) de **caracteres**

Para criar objectos destas classes, é necessário **fornecer como parâmetro um objecto da classe `File`** correspondente ao ficheiro pretendido:

```
FileReader fileReader = new FileReader(new File(nomeDoFicheiro));
FileWriter fileWriter = new FileWriter(new File(nomeDoFicheiro));
```

Ou, de forma abreviada:

```
FileReader fileReader = new FileReader(nomeDoFicheiro);
FileWriter fileWriter = new FileWriter(nomeDoFicheiro);
```

Assim que a *stream* é construída **o ficheiro de texto é criado se não existia antes ou os seus conteúdos removidos se o ficheiro já existia** (no caso de acesso de escrita)

Leitura e escrita de caracteres em ficheiros

Estas classes são específicas de **ficheiros de caracteres**, **permitindo a sua leitura ou escrita caracter a caracter**

A leitura ou escrita de um caracter de cada vez não é muito conveniente na maior parte das vezes

Por esse motivo, geralmente faz-se apelo a um terceiro objecto, de forma a conseguir a **leitura e a escrita linha a linha**

As classes em causa são a `BufferedReader` e a `BufferedWriter`, que recebem um objecto da classe `FileReader` e `FileWriter`, respectivamente, como parâmetro:

```
BufferedReader bufferReader = new BufferedReader(fileReader);
BufferedWriter bufferWriter = new BufferedWriter(fileWriter);
```

Estas classes possuem métodos convenientes para a **leitura e escrita de linhas de caracteres**, tornando-se, por isso, mais convenientes para a manipulação de ficheiros

Alternativamente à classe `BufferedWriter` pode usar a classe `PrintWriter`

Leitura de ficheiros de texto

Sobre os objectos **`BufferedReader`** e **`BufferedWriter`** será possível aplicar os **métodos `readLine()` e `write()` para leitura e escrita em ficheiros de texto**, respectivamente.

Assim, a **leitura de uma linha a partir de um ficheiro de texto** pode ser feita utilizando o método **`readLine()`** da classe **`BufferedReader`**:

```
// Criação do descritor de leitura
FileReader reader = new FileReader("ponto.txt");
// Criação do buffer de leitura
BufferedReader input = new BufferedReader(reader);
// leitura de uma linha do ficheiro ponto.txt
String line = input.readLine();
```

Métodos relevantes da classe **`BufferedReader`**: **`readLine()`**, **`close()`**

Exemplo de leitura de ficheiros de texto


```
import java.io.*;
//(...)
  try
  {
        FileReader reader = new FileReader("ponto.txt");
        BufferedReader input = new BufferedReader( reader );
        // Lê todas as linhas e mostra-as no ecrã.
        String line = input.readLine();
        while (line != null)
        {
              System.out.println(line);
              line = input.readLine();
        }
        input.close();
        reader.close();
  }
  catch (IOException erro)
  {
        System.out.println("Ocorreu um problema ao usar o ficheiro.");
        System.out.println(erro);
  }
```

Escrita em ficheiros de texto

A **escrita de uma linha** **num ficheiro de texto** pode ser feita utilizando o método **`write()`** da classe **`BufferedWriter`**:

```
// Criação do descritor de escrita
FileWriter writer = new FileWriter("ponto.txt");
// Criação do buffer de escrita
BufferedWriter output = new BufferedWriter(writer);
// escrita de uma linha no ficheiro ponto.txt
output.write("novo texto");
```

Métodos relevantes da classe **`BufferedWriter`**: **`write()`**, **`newLine()`**, **`close()`**

Exemplo de escrita em ficheiros de texto


```
import java.io.*;
//(...)
  try
  {
        FileWriter writer = new FileWriter("ponto.txt");
        BufferedWriter out = new BufferedWriter( writer );

        out.write("Eis um belo ponto");
        out.write("(" + x + "," + y + ")");

        out.close();
        writer.close();
  }
  catch (IOException erro)
  {
        System.out.println("Ocorreu um problema ao usar o ficheiro.");
        System.out.println(erro);
  }
```

Escrita em ficheiros de texto

A **escrita de uma cadeia de caracteres num ficheiro de texto** pode ser obtida com o método **`write()`** da classe **`BufferedWriter`**:

```
FileWriter writer = new FileWriter ("ponto.txt");
BufferedWriter output = new BufferedWriter(writer);
String line = "Texto a inserir no ficheiro";
output.write(line,0,line.length());
output.newLine();
```

Caso use a **classe `PrintWriter` pode recorrer ao método `println()`**, mais simples de aplicar que o método **`write()`** da classe **`BufferedReader`**:

```
FileWriter writer = new FileWriter ("ponto.txt");
PrintWriter output = new PrintWriter(writer));
String line = "Texto a inserir no ficheiro";
output.println(line);
```

Exemplo de escrita em ficheiros de texto


```
import java.io.*;
//(...)
  try
  {
        FileWriter writer = new FileWriter("ponto.txt");
        PrintWriter out = new PrintWriter( writer );

        out.println("Eis um belo ponto");
        out.println("(" + x + "," + y + ")");

        out.close();
        writer.close();
  }
  catch (IOException erro)
  {
        System.out.println("Ocorreu um problema ao usar o ficheiro.");
        System.out.println(erro);
  }
```

Quando utiliza o construtor simples

```
FileWriter fWr = new FileWriter("nome_ficheiro");
```

se o ficheiro já existir, ele é truncado passando a ter tamanho 0. Todo o seu conteúdo é perdido.

Caso se queira **acrescentar dados ao ficheiro**, deverá abri-lo para ***append***:

```
FileWriter fWr = new FileWriter("nome_ficheiro", true);
```

Libertação de recursos

Ficheiros

Após a utilização dos ficheiros é sempre necessário **libertar os recursos reservados**, invocando para tal o método **`close()`**, disponível nas classes **`BufferedReader`** e **`BufferedWriter`**:

```
FileReader reader = new FileReader("ponto.txt");
BufferedReader input = new BufferedReader(reader);
//…
input.close();
reader.close();
```

Se este procedimento não for efectuado, o ficheiro não poderá ser aberto e manipulado por outro processo

Excepções na manipulação de ficheiros

A manipulação de ficheiros implica o **tratamento de excepção do tipo `IOException`**

O tratamento pode ser realizado localmente (através da **bloco `try … catch`**) ou proceder ao lançamento (***throw***) da excepção

As palavras reservadas **`throws IOException`** são obrigatórias no cabeçalho de todos os métodos que incluam operações de entrada/saída ou que chamem métodos que as incluam e não realizem o tratamento local da excepção

A sua função é indicar ao compilador que **o método pode gerar ou propagar uma excepção do tipo `IOException`**

Exemplo de leitura de ficheiros de texto


```
import java.io.*;
//(...)
  try
  {
        FileReader reader = new FileReader("ponto.txt");
        BufferedReader input = new BufferedReader( reader );

        do {
              line = input.readLine();
              System.out.println(line);
        } while (line != null)

        input.close();
        reader.close();
  }
  catch (IOException erro)
  {
        System.out.println("Ocorreu um problema ao usar o ficheiro.");
        System.out.println(erro);
  }
```

Exemplo de leitura de ficheiros de texto

Leitura de ficheiros de texto:

```
// Criação da Stream de leitura
FileReader fRd = new FileReader(new File(nomeDoFicheiro));
BufferedReader bRd = new BufferedReader(fRd);
// Leitura de uma String
String str = bRd.readLine();
// Encerramento da stream de leitura
bRd.close(); fRd.close();
```

Escrita em ficheiros de texto:

```
// Criação da Stream de escrita
FileWriter fWr = new FileWriter(new File(nomeDoFicheiro));
//Nota: para adicionar dados: new FileWriter(new File(nomeDoFicheiro, true));
BufferedWriter bWr = new BufferedReader(fWr);
// Escrita de uma String
bWr.write(String,0,String.length());
bWr.newLine();
// Encerramento da stream de escrita
bWr.close(); fWr.close();
```

Manipulação de ficheiros de texto - Resumo

Dois procedimentos para escrita em ficheiros de texto:

Recorrendo à classe **BufferedWriter**:

```
// Criação da Stream de escrita
FileWriter fWr = new FileWriter(new File(nomeDoFicheiro));
BufferedWriter bWr = new BufferedReader(fWr);
// Escrita de uma String
bWr.write(String,0,String.length());
bWr.newLine();
// Encerramento da stream de escrita
bWr.close(); fWr.close();
```

Recorrendo à classe **PrintWriter**:

```
// Criação da Stream de escrita
FileWriter fWr = new FileWriter(new File(nomeDoFicheiro));
PrintWriter pWr = new PrintWriter(fWr);
// Escrita de uma String
pWr.println(String);
// Encerramento da stream de escrita
pWr.close(); fWr.close();
```

Determinar o número de linhas de um ficheiro de texto

Após o acesso a um ficheiro de texto (através da classe `FileReader`) é possível determinar o **número de linhas existentes** através das classes **`LineNumberReader`** e **`RandomAccessFile`**:

```
public int getNumberLines(){
      int numberLines = 0;
      try {
          RandomAccessFile randFile = new RandomAccessFile(inputFileName, "r");
          long lastRec = randFile.length();
          randFile.close();

          FileReader fileRead = new FileReader(inputFileName);
          LineNumberReader lineRead = new LineNumberReader(fileRead);
          lineRead.skip(lastRec);
          numberLines = lineRead.getLineNumber() - 1;
          fileRead.close();
          lineRead.close();
      } catch (IOException e) { }
      return numberLines;
}
```

Exemplo de manipulação de ficheiros de texto

Operações sobre ficheiros

Utilizando as classes anteriores, é possível criar uma nova classe que permite manipular este tipo de ficheiros através de uma interface simplificada

A utilização de **ficheiros de texto envolve essencialmente quatro operações**: **abertura**, **leitura**, **escrita** e **fecho**

Esta nova classe deve então implementar estes quatro comportamentos

Exemplo de manipulação de ficheiros

Para poder funcionar devidamente, esta classe deve ter como atributos uma referência para um objecto da classe **BufferedReader** e outra para um objecto da classe **BufferedWriter**:

```
import java.io.*;

public class FicheiroDeTexto {
        private BufferedReader fR;
        private BufferedWriter fW;
}
```

Podem agora ser adicionados **dois métodos de abertura do ficheiro**

```
public void abreLeitura(String nomeDoFicheiro) throws IOException{

        fR = new BufferedReader(new FileReader(nomeDoFicheiro));

}

public void abreEscrita(String nomeDoFicheiro) throws IOException{

        fW = new BufferedWriter(new FileWriter(nomeDoFicheiro));

}
```

A **leitura de uma linha a partir de um ficheiro de texto** pode ser feita utilizando o método **`readLine`**`()` da classe `BufferedReader`:

```
//Método para ler uma linha do ficheiro
//Devolve a linha lida
public String lerLinha() throws IOException {
      return fR.readLine();
}
```

Exemplo de leitura de ficheiros de texto

É comum utilizar ficheiros de texto para armazenar representações de números, um em cada linha.

Claro que internamente os números são armazenados como cadeias de caracteres, pelo que após a leitura de uma linha é necessário converter a cadeia de caracteres obtida para um número.

Para o caso de números inteiros:

```
//Método para ler um número do ficheiro
//Devolve o número lido
public int[] lerNumeroInt() throws IOException {
      int[] result = new int[2];
      String st = fR.readLine();
      if (st != null) {
            result[0] = 0;
            result[1] = Integer.parseInt(st);
      } else {
            result[0] = -1;
      }
      return result;
}
```

Escrita em ficheiros de texto

A **escrita de uma cadeia de caracteres num ficheiro de texto** pode ser obtida com o método **`write()`**:

```
//Método para escrever uma linha no ficheiro
//Recebe a linha a escrever
public void escreverLinha(String linha) throws IOException{
        fW.write(linha,0,linha.length());
        fW.newLine();
}
```

Exemplo de escrita em ficheiros de texto

Tal como para a leitura, é útil criar um método que permita escrever um número inteiro num ficheiro:

```
//Método para escrever um número inteiro no ficheiro
//Recebe o número a escrever
public void escreverNúmero(int num) throws IOException {
      String st = "";
      st = String.valueOf(num);
      escreverLinha(st);
}
```

Libertação de recursos

Ficheiros

Os **métodos para fechar ficheiros** são muito simples, pois limitam-se a utilizar o método **`close()`** das classes **`BufferedReader`** e **`BufferedWriter`**.

Incluem-se na classe `FicheiroDeTexto` apenas para manter a consistência da sua interface:

```
//Método para fechar um ficheiro aberto em modo leitura
public void fechaLeitura() throws IOException {
      fR.close();
}

//Método para fechar um ficheiro aberto em modo escrita
public void fechaEscrita() throws IOException {
      fW.close();
}
```

Exemplo de manipulação de ficheiros de texto

Como exemplo, vamos criar um programa que crie um ficheiro de texto contendo os quadrados de todos os números menores que 10. Posteriormente, os dados armazenados no ficheiro devem ser lidos.

A execução do programa resulta na criação, no directório corrente, de um ficheiro de nome "`teste.txt`".

```
public class ExemploFicheiros {
  public static void main(String args[]) throws IOException {
    //Cria um ficheiro e escreve nele os quadrados dos números de 1 a 9
    FicheiroDeTexto f = new FicheiroDeTexto();
    f.abreEscrita("teste.txt");
    for (int i=1;i<10;i++) {
            f.escreverNumero(i*i);
    }
    f.fechaEscrita();
    System.out.println("Quadrados dos números entre 1 e 10");

    //Abre o ficheiro, lê e imprime os dados
    f.abreLeitura("teste.txt");
    int i = 1;
    int[] resultado;
    do {
        resultado = f.lerNumeroInt();
        if (resultado[0]==0) {
            System.out.println(i+" "+resultado[1]);
            i++;
        }
    } while (resultado[0]==0);
    f.fechaLeitura();
  }
}
```

Ficheiros Binários

Leitura de ficheiros binários

Para **ler ficheiros binários** o processo é semelhante aos ficheiros de texto. Basta criar o **encadeamento correcto de *streams*** para se obterem os dados na forma desejada

Para ler os ficheiros binários é típico usar a combinação `FileInputStream` / `DataInputStream`

```
FileInputStream fileInputStr = new FileInputStream("dados.dat");
DataInputStream dataInputStr = new DataInputStream(fileInputStr);
```

E sobre o objecto `DataInputStream` podem-se aplicar os métodos `readInt()` para leitura de valores inteiros, `readChar()` para leitura de caracteres, `readFloat()` para leitura de valores de vírgula flutuante, etc.

Pode-se recorrer ao método `available()` para **determinar o número de bytes disponíveis para serem lidos da *stream***, herdado da classe `FileInputStream`.

Exemplo de leitura de ficheiros binários


```
import java.io.*;
//(...)
  try
  {
        FileInputStream fileInputStr = new FileInputStream("dados.dat");
        DataInputStream dataInputStr = new DataInputStream( fileInputStr );

        while (dataInputStr.available() > 0)
        {
              int  k = dataInputStr.readInt();
              System.out.println(k);
        }

        dataInputStr.close();
        fileInputStr.close()
  }
  catch (IOException erro)
  {
        System.out.println("Ocorreu um problema ao usar o ficheiro.");
        System.out.println(erro);
  }
```

Escrita em ficheiros binários

Para **escrever para um ficheiro binário**, tipicamente utiliza-se a combinação `FileOutputStream` / `DataOutputStream`.

A **primeira classe permite abrir e/ou criar o ficheiro binário**, existindo **na segunda métodos que permitem escrever dados** que vão desde inteiros a *strings*.

```
FileOutputStream fileOutputStr = new FileOutputStream("dados.dat");
DataOutputStream dataOutputStr = new DataOutputStream(fileOutputStr);
```

Sobre o objecto `DataOutputStream` pode-se aplicar o método `writeInt()` para escrita de valores inteiros, `writeChar()` para escrita de caracteres, `writeFloat()` para escrita de valores de virgula flutuante

O método `size()` permite saber o tamanho em bytes escrito até ao momento, enquanto o método `flush()` para esvaziar a *stream* escrevendo os dados no ficheiro

Exemplo de escrita em ficheiros binários


```
import java.io.*;
//(...)
  try
  {
        FileOutputStream fileOutputStr = new FileOutputStream("dados.dat");
        DataOutputStream dataOutputStr = new DataOutputStream(fileOutputStr);

        for (int i=0; i<10; i++)
              dataOutputStr.writeInt(i);

        dataOutputStr.close();
        fileOutputStr.close();
  }
  catch (IOException erro)
  {
        System.out.println("Ocorreu um problema ao usar o ficheiro.");
        System.out.println(erro);
  }
```

Manipulação de ficheiros binários - Resumo

Leitura de ficheiros binários:

```
// Criação da Stream de leitura
FileInputStream fIS = new FileInputStream(new File(ficheiro));
DataInputStream dIS = new DataInputStream(fIS);

// Leitura de um valor inteiro (usar o método available() para determinar o
// número de bytes disponíveis para leitura)
int valI = dIS.readInt(); char valC = dIS.readChar(); …

// Encerramento da stream de leitura
dIS.close(); fIS.close();
```

Escrita em ficheiros binários:

```
// Criação da Stream de escrita
FileOutputStream fOS = new FileOutputStream(new File(ficheiro));
DataOutputStream dOS = new DataOutputStream(fOS);

// Escrita de valor inteiro
dOS.writeInt(int); dOS.writeChar(char); dOS.writeFloat(float); …

// Encerramento da stream de escrita
dOS.close(); fOS.close();
```

Ficheiros de Objectos

Manipulação de ficheiros de objectos

Ficheiros

A **leitura e a escrita de objectos de e para ficheiros** são asseguradas pelas classes `ObjectInputStream` e `ObjectOutputStream`.

A classe `ObjectInputStream`, através do método `readObject()`, **recolhe os dados do fluxo de entrada e reorganiza-os, de forma a reconstruir um objecto** igual ao inicialmente escrito.

A classe `ObjectOutputStream`, através do seu método `writeObject()`, **organiza os dados do objecto, de modo a que possam ser enviados sequencialmente para o ficheiro** através do fluxo de saída de dados.

Estes métodos podem trabalhar com qualquer classe de objectos predefinidos na linguagem, como sejam as cadeias de caracteres, os vectores ou os dicionários .

A leitura de um ficheiro de objectos pode lançar uma excepção `ClassNotFoundException` para além da tradicional `IOException`, pois não há garantia que o tipo de classe em leitura exista (necessário *cast*).

Classes *serializable*

Ficheiros

Para armazenar em ficheiro objectos de classes não pré-definidas, é **necessário indicar que se autoriza a sua reorganização para armazenamento em ficheiro**.

Para isso é preciso acrescentar ao cabeçalho dessas classes as palavras **`implements Serializable`**, por exemplo:

```
public class Turma implements Serializable
```

Este cabeçalho permite que todos os objectos da classe `Turma` possam ser fornecidos ao método **`writeObject()`** para serem enviados para um ficheiro (é recomendável invocar o método **`flush()`** após a escrita de um objecto).

O método **`readObject()`** pode ser usado para ler dados do ficheiro e construir um objecto correspondente em memória central.

É importante notar que **apenas as variáveis de instância (atributos) de um objecto são incluídas no ficheiro**. As **variáveis globais ou de classe (*static*) que um objecto utilize não são incluídas no ficheiro**.

Leitura e escrita de objectos em ficheiros

No caso dos ficheiros de texto, foram utilizadas as classes **`FileReader`** e **`FileWriter`** para estabelecer os **fluxos de dados**, uma vez que se tratava apenas de caracteres.

Neste caso, a utilização destas classes não é adequada. Em sua substituição devem ser utilizadas as classes **`FileInputStream`** e **`FileOutputStream`**:

```
FileInputStream  fIS = new FileInputStream(new File(ficheiro));
FileOutputStream fOS = new FileOutputStream(new File(ficheiro));
```

Ou de forma simplificada:

```
FileInputStream  fIS = new FileInputStream(nomeDoFicheiro);
FileOutputStream fOS = new FileOutputStream(nomeDoFicheiro);
```

Agora é possível criar objectos para manipular os ficheiros de objectos :

```
ObjectInputStream  oIS = new ObjectInputStream( fIS );
ObjectOutputStream oOS = new ObjectOutputStream( fOS );
```

e recorrer aos métodos **`readObject()`** para leitura e **`writeObject()`** para escrita de objectos

Exemplo de leitura de ficheiros de objectos


```
import java.io.*;
//(...)
  try {
        File f = new File("teste.dat");

        FileInputStream   fIS = new FileInputStream(f);
        ObjectInputStream oIS = new ObjectInputStream( fIS );

        System.out.println( (String)oIS.readObject() );

        oIS.close();
        fIS.close();
  } catch (IOException ioe) {
        ioe.printStackTrace()
  } catch (ClassNotFoundException cnfe) {
        cnfe.printStackTrace()
  }
```

Exemplo de escrita em ficheiros de objectos


```
import java.io.*;
//(...)
  try {
        File f = new File("teste.dat");

        FileOutputStream   fOS = new FileOutputStream(f);
        ObjectOutputStream oOS = new ObjectOutputStream( fOS );

        oOS.writeObject(new String("Texto"));
        oOS.flush();

        oOS.close();
        fOS.close();
  } catch (IOException ioe) {
        ioe.printStackTrace()
  }
```

Manipulação de ficheiros de objectos - Resumo

Ficheiros

Leitura de ficheiros de objectos:

```
// Criação da Stream de leitura
FileInputStream   fIS = new FileInputStream(new File(fich));
ObjectInputStream oIS = new ObjectInputStream(fIS);
// Leitura de um Objecto
oIS.readObject();
// Encerramento da stream de leitura
oIS.close(); fIS.close();
```

Escrita em ficheiros de objectos:

```
// Criação da Stream de escrita
FileOutputStream   fOS = new FileOutputStream(new File(fich));
ObjectOutputStream oOS = new ObjectOutputStream(fOS);
// Escrita de um Objecto
oOS.writeObject(Object); oOS.flush();
// Encerramento da stream de escrita
oOS.close(); fOS.close();
```

Leitura de ficheiros de objectos

Na leitura de ficheiros de objectos não existe forma **de verificarmos quando chegamos ao fim do processo** (o teste '`!= null`' não funciona, pois quando se proceder a uma leitura após o último objecto será lançada uma excepção)

A forma usual de o fazer é aplicar um bloco **`try / catch`** e tratar a respectiva **excepção `EOFException`** que surge quando é invocado o método **`readObject()`** após a leitura do último objecto

Ou seja, estamos em **ciclo infinito a ler o ficheiro de objectos**, até que a excepção **`EOFException`** seja lançada. Quando isso suceder significa que atingimos o fim ficheiro, não existindo mais objectos a obter

No próximo exemplo, apenas escrevemos no dispositivo de saída (por defeito o ecrã) a mensagem "*`Fim de Ficheiro`*". Podíamos neste caso não realizar qualquer acção no tratamento da excepção **`EOFException`**, sendo, no entanto, um procedimento incorrecto

Exemplo de leitura de ficheiros de objectos


```
FileInputStream   fIS = null;
ObjectInputStream oIS = null;
File f = new File("teste.dat");

try {
      fIS = new FileInputStream(f);
      oIS = new ObjectInputStream(fIS);

      while(true) {
            System.out.println(oIS.readObject());
      }
      // oIS.close();
      // fIS.close(); // Código não será executado devido à excepção EOF
}
catch (EOFException eof) {System.out.println("Fim do Ficheiro");}
catch (IOException eio)  {System.out.println("Erro na leitura");}
catch (ClassNotFoundException cnf) {System.out.println("Obj inexistente");}
catch (Exception e)      {e.printStackTrace();}
finally {
      oIS.close();
      fIS.close();
}
```

Actualização de ficheiros de objectos

Se pretendermos realizar a **abertura de um ficheiro de objectos para acrescentar dados** (sem apagar o seu conteúdo), é necessário integrar uma alteração ao procedimento adoptado com os ficheiros de texto ou binários.

Se o ficheiro de objectos for acedido para escrita, acrescentando novos dados:

```
try {
      File f = new File("teste.dat");
      FileOutputStream   fOS = new FileOutputStream(f, true);
      ObjectOutputStream oOS = new ObjectOutputStream(fOS);

      oos.writeObject("Texto");

      oOS.flush();
      oOS.close();
      fOS.close();
} catch (IOException ioe) {
      ioe.printStackTrace()
}
```

Actualização de ficheiros de objectos – problema

e posteriormente acedido para realizar a respectiva leitura:

```
try {
        FileInputStream   fIS = new FileInputStream(f);
        ObjectInputStream oIS = new ObjectInputStream(fIS);
        while(true){System.out.println(oIS.readObject());}
        oIS.close();
        fIS.close();
} catch (ClassNotFoundException cnfe) {cnfe.printStackTrace()}
  catch (IOException ioe) {ioe.printStackTrace()}
```

Uma **excepção é lançada**:

```
java.io.StreamCorruptedException: invalid type code: AC
at java.io.ObjectInputStream.readObject0(Unknown Source)
at java.io.ObjectInputStream.readObject(Unknown Source)
```

Esta excepção deve-se ao facto de se realizar um acrescento (*append*) a um ficheiro de objectos existente. O problema quando se pretende acrescentar informação a um ficheiro de objectos encontra-se **na forma como o ficheiro foi encerrado pela última vez (criação de um cabeçalho adicional)**.

Um procedimento para lidar com o problema é apresentado de seguida.

Actualização de ficheiros de objectos – possível solução

É necessário primeiro criar uma classe, que **herda da classe `ObjectOutputStream`** (para leitura de ficheiros de objectos), permitindo lidar com problema na abertura de ficheiros de dados. Por exemplo:

```
import java.io.IOException;
import java.io.ObjectOutputStream;
import java.io.OutputStream;

public class ObjectOutputStreamExtended extends ObjectOutputStream{
        ObjectOutputStreamExtended(OutputStream os) throws IOException {
                super(os);
        }
        protected void writeStreamHeader() throws IOException {
                reset();
        }
}
```

No nosso código, quando abrimos um ficheiro de objectos para acrescentar dados devemos **verificar se o ficheiro existe**. Se não existir, abrimos o ficheiro sem *append*, pelo processo normal, se existir abrimos o ficheiro com um *append* (mas não usamos a classe normal **`ObjectOutputStream`** mas sim a que acabamos de criar **`ObjectOutputStreamExtended`**)

Actualização de ficheiros de objectos – possível solução


```
try {
      FileOutputStream   fOS = null;
      ObjectOutputStream oOS = null;
      File f = new File("teste.dat");

      if (f.exists()) {
            fOS = new FileOutputStream(f, true);
            oOS = new ObjectOutputStreamExtended(fOS);
      } else {
            fOS = new FileOutputStream(f);
            oOS = new ObjectOutputStream(fOS);
      }
      oOS.writeObject("Texto");
      oOS.flush();

      oOS.close();
      fOS.close();
} catch (IOException ioe) {
      ioe.printStackTrace();
}
```

Exemplo de manipulação de ficheiros de objectos

Abertura de ficheiros de objectos para leitura e escrita

Tal como para os ficheiros de texto, pode ser desenvolvida uma classe que simplifique a utilização de ficheiros de objectos.

Esta classe, que vai ser denominada `FicheiroDeObjectos`, incluirá **métodos para abrir um ficheiro, escrever um objecto num ficheiro, ler um objecto a partir de um ficheiro e fechar um ficheiro**.

A classe terá como atributos referências para um objecto da classe `ObjectInputStream` e para um objecto da classe `ObjectOutputStream`:

```
import java.io.*;
public class FicheiroDeObjecto {
        private ObjectInputStream iS;
        private ObjectOutputStream oS;
        //…
}
```

Abertura de ficheiros de objectos para leitura e escrita

Os métodos de **abertura do ficheiro** para **leitura** e para **escrita** podem ser:

```
//Método para abrir um ficheiro para leitura
//Recebe o nome do ficheiro
public void abreLeitura(String nomeFicheiro) throws IOException {
        iS = new ObjectInputStream(new FileInputStream(nomeFicheiro));
}


//Método para abrir um ficheiro para escrita
//Recebe o nome do ficheiro
public void abreEscrita(String nomeFicheiro) throws IOException {
        os = new ObjectOutputStream(new FileOutputStream(nomeFicheiro));
}
```

Leitura de ficheiros de objectos

leitura de um objecto a partir de um ficheiro pode ser efectuada através do método **`readObject`**`()` da classe `ObjectInputStream`:

```
//Método para ler um objecto do ficheiro
//Devolve o objecto lido
public Object leObjecto() throws IOException, ClassNotFoundException {
        return iS.readObject();
}
```

Este método devolve um resultado **`Object`**, de modo a poder ler qualquer tipo de objecto. **Caberá ao método que o chamar concretizar, através de um *cast*, qual a classe a que esse objecto deve pertencer.**

Notar a indicação de que o método pode propagar um erro do tipo **`ClassNotFoundException`**. Este erro será gerado pelo método `readObject()` se o ficheiro não contiver objectos.

Escrita em ficheiros de objectos

A **escrita** **de um objecto** num ficheiro pode ser obtida com o método **`writeObject()`** da classe `ObjectOutputStream()`:

```
//Método para escrever um objecto no ficheiro
//Recebe o objecto a escrever
public void escreveObjecto(Object o) throws IOException {
        os.writeObject(o);
}
```

Libertação de recursos

Ficheiros

Os métodos para **fechar os ficheiros** limitam-se a utilizar o método `close()` das classes `ObjectInputStream` e `ObjectOutputStream`:

```
//Método para fechar um ficheiro aberto em modo leitura
public void fechaLeitura() throws IOException {
        is.close();
}

//Método para fechar um ficheiro aberto em modo escrita
public void fechaEscrita() throws IOException {
        os.close();
}
```

Exemplo de manipulação de ficheiros de objectos

Ficheiros

```
import java.io.*;
public class GereTurma {
   public static void main(String arg[])
                                        throws IOException, ClassNotFoundException) {
      String nome;
      int escolha;
      Turma t = new Turma();

      //Lê a turma a partir do ficheiro
      FicheiroDeObjectos fo = new FicheiroDeObjectos();
      fo.abreLeitura("turma.dat");
      t = (Turma) fo.leObjecto();
      fo.fechaLeitura();

      do {
          System.out.println("1 - Adicionar estudante");
          System.out.println("0 - Sair");
          escolha = Le.umInt();
          switch (escolha) {
             case 1 t.juntaEstudante(); break;
             //Na opção de saída guarda os dados da turma no ficheiro
             case 0: fo.abreEscrita("turma.dat");
                     fo.escreveObjecto(t);
                     fo.fechaEscrita();
                     System.exit(0);
          }
      } while (escolha != 0);
   }
}
```

Java Properties

Java Properties

Ficheiros

Uma forma eficiente para aceder a informação em ficheiros com uma estrutura pré-definida é através do uso de **Java Properties** (**`java.util.Properties`**).

O procedimento procura ler e escrever em ficheiros de texto, com uma **estrutura pré-definida**, **acedendo directamente às variáveis pretendidas** (evitando assim percorrer a totalidade do ficheiro para encontrar a informação desejada).

É muito comum em ficheiros de configuração (e.g. para leitura dos parâmetros de acesso a uma base de dados) com a estrutura:

```
nomeVariável1 = valor1
nomeVariável2 = valor2
```

```
sql.safe_mode = Off
odbc.allow_persistent = On
odbc.defaultlrl = 4096
mysql.connect_timeout = 60
mysql.trace_mode = Off
mysqli.max_links = -1
mysqli.default_port = 3306
mysqli.default_host = localhost
mysqli.default_user = root
mysqli.reconnect = Off
msql.allow_persistent = On
msql.max_persistent = -1
pgsql.allow_persistent = On
```

A título de exemplo, observe-se o seguinte **ficheiro (de texto)** '`config.properties`' contendo os parâmetros de acesso a uma base de dados

```
dbuser      = root
dbpassword = qwl2$sjs&sk0
dburl       = localhost
```

Neste ficheiro estão armazernadas **três variáveis ou *properties*** (`dbuser`, `dbpassword` e `dburl`) e respectivos valores (à direita do sinal de igual '=').

Para aceder aos seus valores bastará indicar o nome da variável, não sendo necessário percorrer todo o ficheiro

Java Properties - Leitura

Ficheiros

Para aceder ao ficheiro basta criar uma instância da classe **`Properties`**

```
Properties prop = new Properties();
```

aceder ao ficheiro estruturado através de um **`FileInputStream`**

```
prop.load(new FileInputStream("config.properties"));
```

e ler os valores pretendidos:

```
String db   = prop.getProperty("dburl");
String user = prop.getProperty("dbuser");
String pwd  = prop.getProperty("dbpassword");
```

Java Properties - Escrita

Ficheiros

A escrita segue o mesmo princípio da leitura: criar uma instância da classe **`Properties`**

```
Properties prop = new Properties();
```

definir os valores das variáveis (*properties*):

```
prop.setProperty("dburl",      "localhost");
prop.setProperty("dbuser",     "root");
prop.setProperty("dbpassword", "qwl2$sjs&sk0");
```

guardar os dados no ficheiro estruturado através de um **`FileOutputStream`**

```
prop.store(new FileOutputStream("config.properties"), null);
```

Java Properties – Exemplo Escrita

Ficheiros

```
import java.io.FileOutputStream;
import java.io.IOException;
import java.util.Properties;

public class App
{
    public static void main( String[] args )
    {
       Properties prop = new Properties();

       try {
              //set the properties value
              prop.setProperty("dburl", "localhost");
              prop.setProperty("dbuser", "mkyong");
              prop.setProperty("dbpassword", "fdj34k1k4#2d");

              //save properties to project root folder
              prop.store(new FileOutputStream("config.properties"), null);

       } catch (IOException ex) {
              ex.printStackTrace();
         }
    }
}
```

Java Properties – Exemplo Leitura

Ficheiros

```
import java.io.FileInputStream;
import java.io.IOException;
import java.util.Properties;

public class App
{
    public static void main( String[] args )
    {
       Properties prop = new Properties();

       try {
              //load a properties file
              prop.load(new FileInputStream("config.properties"));

              //get the property value and print it out
              System.out.println( prop.getProperty("dburl") );
              System.out.println( prop.getProperty("dbuser") );
              System.out.println( prop.getProperty("dbpassword") );

       } catch (IOException ex) {
              ex.printStackTrace();
         }
    }
}
```

Leitura de dados do dispositivo de entrada

Em Java existem 3 dispositivos distintos:

- `System.in` – **dispositivo de entrada** de dados (por defeito o teclado);
- `System.out` – **dispositivo de saída** de dados (por defeito o monitor);
- `System.err` – **dispositivo para disponibilização de erros** (por defeito um ficheiro).

A **leitura de dados do dispositivo de entrada** (quando o utilizador pretende introduzir informação através do teclado) **faz-se recorrendo ao objecto** `System.in`.

Existem várias técnicas para atingir esse fim. Iremos abordar duas.

A primeira técnica consiste em **criar uma ponte para a leitura de cadeias de caracteres**.

Isso é conseguido com a classe **`InputStreamReader`**. Sobre esta ponte vamos **aplicar um *buffer*** para ser possível realizar uma **leitura eficiente de texto (linha completa)**, recorrendo à classe **`BufferedReader`**. Este é um procedimento **semelhante ao utilizado para aceder a ficheiros de texto**.

Com o objecto resultante podemos então proceder à leitura de texto do teclado, invocando método **`readLine()`** que devolve uma *String* (a introdução de texto termina com a tecla '**`Enter`**'). Exemplo:

```
BufferedReader in =
   new BufferedReader(new InputStreamReader(System.in), 1);

String s = in.readLine();
```

O argumento 1 representa o tamanho do *buffer* de leitura

Como o método **`readLine()`** **pode lançar uma excepção do tipo** **`IOException`**, é necessário proceder ao seu tratamento com um bloco **`try ... catch`**.

```
try {
     BufferedReader in =
          new BufferedReader(new InputStreamReader(System.in));
     String s = in.readLine();
} catch (IOException ioe) {
     System.out.println("Erro na leitura de dados de entrada.");
}
```

Se pretendemos realizar a **leitura de outros tipos de dados que não *String*** (por exemplo inteiros ou *floats*) **teremos sempre que obter uma *String*** através do **`BufferedReader`** e **realizar a conversão** para o respectivo tipo de dados. Por exemplo, para obter um inteiro o procedimento será o seguinte:

```
try {
      BufferedReader in =
                        new BufferedReader(new InputStreamReader(System.in));
      int i = Integer.parseInt(in.readLine().trim());
} catch (IOException ioe) {
      System.out.println("Erro na leitura de dados de entrada.");
} catch (NumberFormatException nfe) {
      System.out.println("O valor não é um número inteiro");
} catch (Exception e) {
      e.printStackTrace();
}
```

De realçar que a aplicação do método **`parseInt()`** da classe **`Integer`** pode lançar uma excepção do tipo **`NumberFormatException`** já que não existem garantias que a *String* obtida corresponda a um número inteiro, pelo que terá que ser tratada.

Outra técnica consiste em recorrer à classe **`Scanner`**, que **permite realizar a leitura de dados do dispositivo de entrada no formato dos tipos primitivos** (***`int`***, ***`float`***, ***`long`***, ***`double`***, ***`char`***, ***`byte`*** e ***`boolean`***), formatados com expressões regulares:

```
Scanner input = new Scanner(System.in);
```

Sobre o objecto criado podemos realizar duas acções**: verificar se existem mais elementos para leitura** (recorrendo ao método **`hasNext()`**) ou **realizar a leitura** (invocando o método **`next()`**). Assim, para proceder à leitura de uma *String*:

```
Scanner input = new Scanner(System.in);
String s = input.next();
```

É possível especificar o tipo de dados a recolher recorrendo a métodos próprios. Por exemplo para **obter um inteiro recorre-se ao método** `nextInt()`, para **obter uma *String* invoca-se o método** `nextLine()`, enquanto que para se **aceder a um *long* usa-se o método** `nextLong()`.

No caso de **leitura de *Strings*** é possível usar o método `next()`, que **lê uma palavra (sem espaços)** ou `nextLine()`, que **lê uma frase com espaços**.

Devido à ultima actualização do JDK, após o uso do método `nextInt()`, o **método não interpreta a mudança de linha final**, provocada pelo pressionar da tecla **`ENTER`** (*carriage return*). Nesta situação, **é recomendável após o uso do** `nextInt()` **invocar o método** `nextLine()` para limpar o *buffer*.

Obviamente a invocação destes métodos (e.g. **`nextInt()`**, **`nextLong()`**) pode **lançar algumas excepções devido à impossibilidade de converter os dados** recolhidos do dispositivo de entrada no formato pretendido, sendo necessário realizar o tratamento das respectivas excepções, nomeadamente:

- **`InputMismatchException`**,
- **`NoSuchElementException`**,
- e **`IllegalStateException`**.

Exemplo:

```
try {
      long val = (new Scanner(System.in)).nextLong();
} catch (InputMismatchException ime) {
      System.out.println("O valor não é um número");
} catch (NoSuchElementException noee) {
      nsee.printStackTrace();
} catch (IllegalStateException ise) {
      ise.printStackTrace();
} catch (Exception e) {
      e.printStackTrace();
}
```

É possível **aumentar a eficiência da classe `Scanner`** se sobre o objectivo do dispositivo de entrada (**`System.in`**) se **aplicar um `BufferedInputStream`**:

```
Scanner input = new Scanner (new BufferedInputStream (System.in));
```

Leitura de dados a partir do dispositivo de entrada - Resumo

Ficheiros

Através da classe `BufferedReader`:

```
// Criação da Stream de leitura
BufferedReader input = new
            BufferedReader(new InputStreamReader(System.in), 1);
// Leitura de um Objecto
String s = input.readLine();
```

Através da classe `Scanner`:

```
// Criação da Stream de escrita
Scanner input = new Scanner(System.in);
// Alternativa: Scanner input = new Scanner (new BufferedInputStream (System.in));
// Leitura de um Objecto
String s = input.next();
// Também possível nextLine(), nextInt(), nextLong()
```

Referências

Ficheiros

"***Programação Orientada a Objectos***"
António José Mendes
Departamento de Engenharia Informática, Universidade de Coimbra

"***Java in a Nutshell***", 4ª Edição, Capítulo 4 "*The Java Platform*"
David Flanagan
O'Reilly, ISBN: 0596002831

"***Thinking in Java***, ", 4ª Edição, Capítulo 11 "*The Java I/O System*"
Bruce Eckel
Prentice Hall, ISBN: 0131872486

"***The Java Tutorial – Essential Classes: Basic I/O***"
Java Sun Microsystems
http://java.sun.com/docs/books/tutorial/essential/io/index.html

Bibliografia complementar

Ficheiros

"***Fundamentos de Programação em Java 2***", Capítulo 10 "*Ficheiros*"
António José Mendes, Maria José Marcelino
FCA, ISBN: 9727224237

"***Java 5 e Programação por Objectos***", Capítulo 10 "*Streams*"
F. Mário Martins
FCA, ISBN: 9727225489