



Entrada de dados do teclado (Scanner) Ficheiros

Classe Scanner

- O Java oferece a classe **Scanner** para ler texto de várias fontes:
 - Teclado (interacção com o utilizador)
 - Objecto String
 - Ficheiro
 - ...
- Para utilizar esta classe é necessário importá-la

Importação de classes

- É feita através da declaração das classes a importar no cabeçalho da classe, antes da definição da mesma

```
import java.util.Scanner;
```

```
public class ... {  
    ...
```

```
    public static void main(String[] args) {  
        Scanner scanner = new Scanner(...);  
        ...  
    }
```

```
}
```

Classe Scanner (teclado)

- Leitura do teclado

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

```
String line = scanner.nextLine();
```

- O programa bloqueia até que o utilizador escreva algo (na consola) e pressione *enter*
- A linha introduzida é guardada num objecto String (Referenciado por `line` no exemplo acima)

Classe Scanner (ficheiro)

- Leitura do teclado

```
Scanner scanner = new Scanner(new File("file.txt"));
```

```
String line = scanner.nextLine();
```

- É obrigatório prever a possibilidade de o ficheiro não existir (e tratar adequadamente a situação).

Classe Scanner (String)

- Processamento de String, palavra a palavra (*tokens*)

```
String sentence = "um dois tres quatro cinco ";
Scanner scanner = new Scanner(sentence);
int n = 0;
String inverted = "";

while(scanner.hasNext()) {
    n++;
    String token = scanner.next();
    inverted = token + " " + inverted;
}
System.out.println(n + " palavras");
System.out.println("Invertida: " + inverted);
```

> 5 palavras

> Invertida: cinco quatro tres dois um

Ficheiros

TEXTUAIS

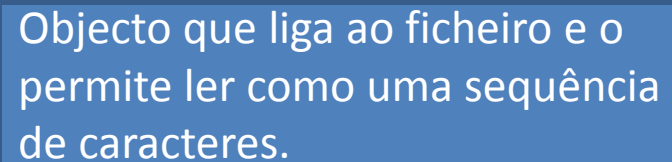
- Legíveis
- Extensos
- Formato (quase) universal
- Formatos padronizados:
 - XML
 - SGML

BINÁRIOS

- Ilegíveis por humanos
- Compactos
- Formato pode depender da arquitetura da máquina
- Formatos padronizados:
 - ASN.1
 - Object Serialization Stream Protocol (Java)

Classes para acesso a ficheiros de texto

- Scanner
 - Para leitura
 - Usada anteriormente para ler do teclado
 - Estabelece *fluxo* (interno) de entrada do ficheiro
- PrintWriter
 - Para escrita
 - Interface semelhante à de System.out
 - Estabelece *fluxo* (interno) de saída para ficheiro
- File
 - Representa ficheiros



Objecto que liga ao ficheiro e o permite ler como uma sequência de caracteres.

Excepções de entrada e saída

- Que acontece quando
 - ficheiro não existe?
 - tipo dos dados pedido não corresponde ao conteúdo a ler?
 - ...
- É lançada uma *excepção* ou fica registado um erro
- Excepções suportadas por *mecanismo de excepções do Java*

Excepções de entrada com Scanner

- `IOException`

Faz parte da lógica do programa

- `FileNotFoundException` – Tentativa de estabelecimento de fluxo de entrada de um ficheiro inexistente

- `RuntimeException`

Erro de programação! A possibilidade de leitura tem de ser verificada *a priori*!

- `InputMismatchException` – Tentativa de leitura de valor de tipo incompatível com conteúdo do ficheiro (e.g., `int` quando ficheiro contém letras)
- `NoSuchElementException` – Tentativa de leitura quando o fluxo de entrada está esgotado
- `IllegalStateException` – Tentativa de leitura quando o fluxo de entrada está fechado

Exceções de saída com PrintWriter

- `IOException`

Faz parte da lógica do programa

- `FileNotFoundException` – Tentativa falhada de estabelecimento de fluxo de saída para ficheiro

- `RuntimeException`

- nada

Não há exceções relacionadas com erros de programação! O sucesso da escrita tem de ser verificado *a posteriori*!

Sequência de acesso a ficheiro

- Abertura – Estabelecimento de fluxo (interno) de entrada ou saída a partir do ficheiro
- Interacção - Leitura ou escrita *sequenciais*
- Fecho – Fecho do fluxo (interno) de entrada (opcional) ou saída (obrigatório nos ficheiros)

São possíveis outras formas de interacção.

Exemplo de leitura: abertura

```
import java.io.File;
import java.io.FileNotFoundException;
import java.util.Scanner;
import static java.lang.System.out;

...
try {
    final Scanner fileScanner =
        new Scanner(new File("My file.txt"));
    ...
} catch (final FileNotFoundException exception) {
    out.println("File was not found. Sorry!");
    ...
}
...
```

O Scanner cria um fluxo de entrada de caracteres entre o ficheiro e si mesmo.

Tem de se lidar com possibilidade de ficheiro não existir!

Exemplo de leitura: leitura e fecho

É um erro de programação se a leitura falhar, sendo por isso lançada uma `RuntimeException`.

```
...
try {
    if (fileScanner.hasNextInt()) {
        final int numberOfCars = fileScanner.nextInt();

        ...
    } else {
        out.println("Ops!");
    }
} finally {
    fileScanner.close();
}
```

Tem de se lidar explicitamente com possibilidade de ficheiro não conter dados no formato esperado!

Uma exceção não apanhada é propagada ao longo da pilha de invocações. Como o fluxo que liga um scanner a um ficheiro *deve* (se possível) *ser fechado*, usa-se o bloco `finally`.

Exemplo de escrita: abertura

```
import java.io.File;
import java.io.FileNotFoundException;
import java.io.PrintWriter;
import static java.lang.System.out;

...
try {
    final PrintWriter filewriter =
        new PrintWriter(new File("My new file.txt"));
    ...
} catch (final FileNotFoundException exception) {
    out.println("Error creating file. Sorry!");
    ...
}
...
```

O `PrintWriter` cria um *fluxo* de saída de caracteres entre si mesmo e o ficheiro.

Tem de se lidar com a possibilidade de não se conseguir ligar ao ficheiro para escrita (e.g., directório não existe, ficheiro já existe e não se pode escrever sobre ele, etc.!).

Exemplo de escrita: escrita e fecho

```
...  
try {  
    filewriter.println(20);  
    ...  
    if (filewriter.checkError())  
        out.println("Error writing");  
} finally {  
    filewriter.close();  
}  
...
```

Tem de se lidar explicitamente com a possibilidade de ocorrerem erros durante a escrita no ficheiro!

Uma excepção não apanhada é propagada ao longo da pilha de invocações. Como o *fluxo* que liga um `Printwriter` a um ficheiro *tem sempre de se fechar*, usa-se o bloco `finally`.

Canais de Objectos

- Mantêm as relações entre Objectos
- Escrevem em formato binário
- São muito simples de usar
- [ObjectOutputStream](#)
- [ObjectInputStream](#)

Canais de Objectos

```
public class Aula implements Serializable {  
    String nome = null;  
    int n_presenças = 0;  
  
    Disciplina disciplina = null;  
  
    public Aula(String nome, int n, Disciplina d) {  
        this.nome = nome;  
        n_presenças = n;  
        disciplina = d;  
    }  
    // ...  
}
```

A interface Serializable

- Para que os objectos criados no decorrer de um programa persistam após o seu termo, devemos garantir que são serializáveis
- Todas as variáveis não static (e não transient) e qualquer objecto são serializáveis
- A interface [Serializable](#) não obriga à implementação de qualquer método; as instâncias das classes que a implementam podem gravar-se em memória secundária como objectos, recorrendo a ObjectOutputStream (que têm métodos como writeObject() e readObject())
- Para uma classe ter instâncias gravadas como objectos em memória secundária (serializadas em disco) que podem ser lidas novamente para a RAM (desserializadas) basta no cabeçalho escrever implements Serializable

Canais de Objectos

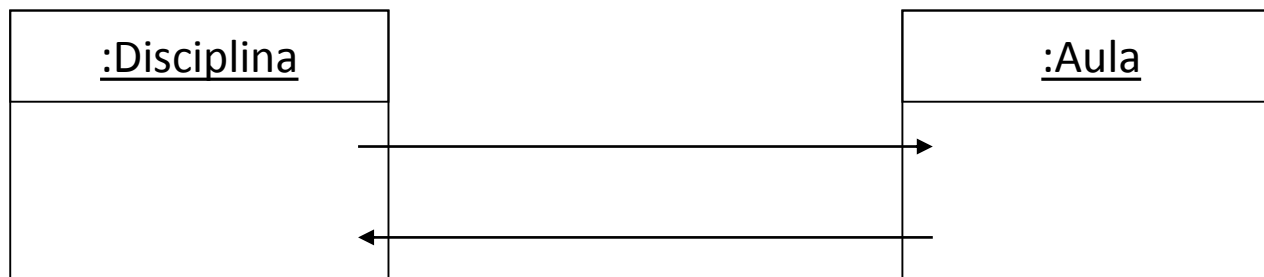
```
public class Disciplina implements Serializable {  
  
    private String nome = null;  
    private int média_presenças = 0;  
    private Aula aula = null;  
  
    public Disciplina(String nome, String nome_aula, int n, int m)  
    {  
        this.nome = nome;  
        média_presenças = m;  
        aula = new Aula(nome_aula, n, this);  
    }  
  
    // ...  
}
```

Canais de Objectos

```
public static void main(String[] args) {  
    Disciplina[] a_escrever = new Disciplina[10];  
    Disciplina[] a_ler = null;  
  
    for (int i = 0; i != a_ler.length; ++i) {  
        a_escrever[i] = new Disciplina("D" + i, "A" +  
i, i*10, i*10 + 1);  
    }  
    // ...  
}
```

Canais de Objectos

a_escrever



Canais de Objectos

```
public static void main(String[] args) {  
    // ...  
    try {  
        ObjectOutputStream o = new ObjectOutputStream(new  
            FileOutputStream("dat.dat"));  
        o.writeObject(a_escrever);  
        o.close();  
    } catch { //... }  
    try {  
        ObjectInputStream in = new ObjectInputStream(new  
            FileInputStream("dat.dat"));  
        a_ler = (Disciplina[]) in.readObject();  
        in.close();  
    } catch { //... }
```

Fica com uma cópia, profunda e integral, da matriz “a_escrever”. Sem qualquer referência repetida, com apenas UMA instrução de escrita e outra de leitura.

Canais Pré-Definidos

- São considerados canais binários apesar de transferirem caracteres – foram criados numa versão do Java onde ainda não existiam fluxos de texto (sub-classes de **System**)
- **System.in**: lê informação proveniente do utilizador, tipicamente do teclado
- **System.out**: escreve informação directamente para o utilizador, tipicamente para a consola
- **System.err**: é um canal especial para transmissão de mensagens de erro que (em geral) tem prioridade sobre o **System.out** (faz sempre o *flush* após cada escrita)

System.in

```
import java.io.*;
// ...
BufferedReader teclado = new
    BufferedReader(new
        InputStreamReader(System.in));

String s = teclado.readLine();

System.out.println("Frase lida:\n" + s + "\n");
// ...
```

System.out

```
import java.io.*;  
// ...  
BufferedWriter monitor = new  
    BufferedWriter(new  
        OutputStreamWriter(System.out));  
monitor.write("olá mundo");  
monitor.flush();
```

Ou utilizar antes o método `print()` da classe `System.out`

```
System.out.println("olá mundo");
```

Acesso Sequencial e Dinâmico

- Existem duas formas padrão de acesso a um dado ficheiro, acesso sequencial e dinâmico
- **Acesso sequencial** limitado por uma ordem sequencial de acesso – informação seguinte sempre colocada à frente da informação anterior
- **Acesso dinâmico** permite avançar e recuar de forma arbitrária

Acesso Sequencial – A Classe **File**

- Escrever no ecrã o conteúdo do ficheiro “info” localizado em [..\info]:

```
import java.io.*;
// ...
int c;
File dados = new File("../", "info");
FileInputStream f = new FileInputStream(dados);
while(c = f.read() != -1)
    System.out.print((char)c + " ");
```

- Modo de acrescento de informação

Acesso para acrescentar informação

```
File dados = new File("info");

try {

    FileWriter f = new FileWriter(dados, true);
    f.write('A');
    f.close();

} catch (IOException e) {

    // ...

}
```

Acesso Dinâmico

- Constructor

`RandomAccessFile(File ficheiro, String modo)`

- Modo leitura "r";
modo de escrita "w";
e modo leitura/escrita "rw"

Acesso aleatório, escrita e leitura simultânea

```
File dados = new File("../", "info");
RandomAccessFile f = new RandomAccessFile(dados, "rw");

for(int i = 65; i < 91; i++)
    f.write(i) // escreve o alfabeto

byte[] ler = new byte[10];

f.seek(4); // salta para o quinto byte
f.read(ler, 0, 5); // lê as próximas 5 posições (E, F, G, H, I)

for(int i = 0; i < 5; i++)
    System.out.println((char)ler[i] + ":");
f.seek(3); // salta para o quarto byte (D)

System.out.println((char)f.read());
f.write('X'); // escreve 'X' por cima de 'E'
```

Mais informação / Referências

- Java2 Platform API, Scanner,
<http://download.oracle.com/javase/6/docs/api/java/util/Scanner.html>
- Java 2 Platform API, FileWriter,
<http://download.oracle.com/javase/1.4.2/docs/api/java/io/FileWriter.html>
- Y. Daniel Liang, "Introduction to Java Programming" 7th Ed. Prentice-Hall, 2010.

Sumário

- Entrada de dados do teclado (Scanner)
- Ficheiros