

# **Programação em Java - Fundamentos**

## **6 - Ficheiros**

V2.0

**Citeforma**

Jose Aser Lorenzo

[jose.l.aser@sapo.pt](mailto:jose.l.aser@sapo.pt)

Abril de 2013

## Sumário

<b>Ficheiros .....</b>	<b>4</b>
<b>Objetivos.....</b>	<b>5</b>
<b>Como o Java trabalha com ficheiros.....</b>	<b>6</b>
Porquê usar ficheiros?.....	7
Formatos de ficheiros.....	8
Ficheiros em formato texto.....	9
Ficheiros com formato proprietário.....	10
Stream .....	11
<b>Classes Java usadas para trabalhar com ficheiros .....</b>	<b>13</b>
Classes para output de caracteres .....	15
Classes para input de caracteres .....	16
Classes para output de bytes .....	17
Classes para input de bytes .....	18
<b>Escrever dados no standard output .....</b>	<b>19</b>
O que é o Standard Output? .....	20
Escrita formatada com printf() .....	21
Exemplo de escrita formatada de Strings .....	24
Exemplo de escrita formatada de datas .....	24
Exemplo de escrita formatada de números.....	25
<b>Ler dados a partir do standard input .....</b>	<b>27</b>
O que é o Standard Input?.....	28
Criar canal para standard input usando Scanner .....	29
Ler números e Strings usando Scanner .....	30
Exemplo de Scanner com leitura de números .....	31
Exemplo de Scanner com separador.....	31
Receber em String e converter em número .....	33
Exemplo de conversão de String para número .....	34
<b>Escrever dados em ficheiro de texto.....</b>	<b>36</b>
Etapas para escrever dados em ficheiro de texto.....	37
Criar canal de output .....	38
Escrever no canal de output.....	39
Fechar o canal de output.....	40
Exercício: escrever em ficheiro de texto .....	41
<b>Ler dados a partir de um ficheiro de texto .....</b>	<b>44</b>
Etapas para ler dados a partir de um ficheiro de texto .....	45
Criar canal de input .....	46
Ler dados a partir do canal de input .....	47
Fechar o canal de input.....	48
Exercícios para ler um ficheiro de texto .....	49
Leitura direta para tipo primitivo.....	49
Leitura de uma linha e divisão em partes .....	51
Leitura de uma linha, divisão e conversão .....	53
<b>Escrever dados em ficheiro com formato proprietário .....</b>	<b>55</b>
Formato interno para representação de dados.....	56
Etapas para escrever dados em ficheiro com formato proprietário .....	58
Criar canal de output .....	59
Escrever no canal de output.....	60
Fechar o canal de output.....	61
Exercício: escrever dados num ficheiro usando formatos proprietário .....	62
<b>Ler a partir de ficheiro com dados em formato proprietário.....</b>	<b>64</b>

Etapas para ler dados escritos em ficheiro com formato proprietário .....	65
Criar canal de input .....	66
Ler dados a partir do canal de input .....	67
Fechar o canal de input.....	68
Exercício: ler dados escritos em ficheiro com formato proprietário .....	69
<b>Classe File .....</b>	<b>71</b>
A utilidade da classe File .....	72
Usando a classe File (antes do Java 1.7) .....	73
Nova implementação de Java I/O na versão 1.7 .....	74
Usando a nova API de I/O da versão Java 1.7 .....	75
Exercício sobre a classe File .....	76
Solução com File (versão anterior ao Java 1.7).....	76
Solução com a nova API da versão Java 1.7 .....	77

## Ficheiros



# Programação em Java Fundamentos

---

## Capítulo 6 – Ficheiros

V2.0

José Aser Lorenzo



**Java Fundamentos**  
© Citeforma 2007

**Objetivos**

## Objetivos

---

- Compreender como o Java manipula ficheiros;
- Escrever e ler em ficheiros de texto;
- Escrever e ler em ficheiros com formato proprietário
- Formatar a escrita de dados usando máscaras e tendo em conta os Locale;
- Usar a classe File para interagir com o sistema operativo;



O slide descreve os objetivos deste capítulo. As classes desenvolvidas nos exercícios deste capítulo deverão ficar dentro do projeto **JavaFundamentos** e dentro do package **capitulo6**.

## Como o Java trabalha com ficheiros

### Sumário

- Como o Java trabalha com ficheiros;
- Classes Java usadas para trabalhar com ficheiros;
- Escrever dados no standard output;
- Ler dados a partir do standard input;
- Escrever dados em ficheiro de texto;
- Ler dados a partir de ficheiro de texto;
- Escrever dados em formato proprietário;
- Ler dados a partir de formato proprietário;
- Classe File;



**Porquê usar ficheiros?**

## Porquê usar ficheiros?

- As variáveis dos programas estão em **memória RAM**. Os dados guardados nessas variáveis **perdem-se** quando a execução do programa termina;
- Os dados escritos em ficheiro são guardados no **disco** (magnético, *solid state* ou *flash*) que é um suporte não volátil;
- O sistema operativo faz a **gestão do espaço em disco** e portanto dos ficheiros aí guardados. A linguagem de programação tem que interagir com o sistema operativo;
- Para a linguagem de programação **é mais difícil** ler ou escrever dados em ficheiro que trabalhar com as variáveis normais;



A escrita e leitura de dados num ficheiro é essencial para uma aplicação que pretenda armazenar os seus dados de uma forma não volátil. Neste capítulo vamos abordar os ficheiros de acesso sequencial, que nos permitem manipular dados em formato texto ou formato proprietário. Quando a velocidade de acesso, a segurança, a criação de relações entre os dados e a sua integridade são fatores críticos de sucesso é recomendável substituir o armazenamento em ficheiros pelo recurso a um Sistema Gestor de Base de Dados.

Os ficheiros tornam-se muito úteis nas seguintes situações:

- Configuração da aplicação;
- “*Logging*” – a aplicação deixa “rasto” da sua execução;
- Impressão de relatórios em HTML ou XML;
- Armazenamento de dados de pequena dimensão ou que necessitem apenas de acesso sequencial.

**Formatos de ficheiros**

## Formatos de ficheiros

- Os dados podem ser escritos em ficheiros com formato
  - **texto** ou
  - **proprietário**;



Os ficheiros criados por um programa Java dividem-se em dois grandes grupos: de texto ou formato proprietário.



**Ficheiros em formato texto**

## Ficheiros em formato texto

- Nos **ficheiros de texto** os dados são guardados em formato **ASCII** ou **Unicode**;
  - Um **double** tem que ser **convertido** em String antes de ser escrito no ficheiro;
  - Noutro programa esse dado é lido do ficheiro para uma String e **convertido** em double antes que o programa o possa manipular;
- O ficheiro de texto pode ser lido/alterado por um editor de texto, por exemplo o Notepad do Windows;
- Qualquer linguagem de programação consegue ler/escrever este tipo de ficheiros;



Nos ficheiros de texto os dados são armazenados usando o código ASCII, UNICODE ou outro equivalente. Para as cadeias de caracteres isto não é um problema, pois são sequências de letras. Para os números isto é um problema, porque requer a conversão entre o formato numérico usado internamente para representar o número e os caracteres ASCII ou Unicode que representam os algarismos que constituem o número.

Na escrita em ficheiro a conversão é feita num sentido, enquanto durante a leitura é feita no sentido oposto. Esta conversão requer tempo de processamento.

A grande vantagem dos ficheiros de texto é que são suportados por todas as linguagens de programação. Bons exemplos de transferência de dados usando este tipo de ficheiros é o HTML e os seus derivados, como o XML.

**Ficheiros com formato proprietário**

## Ficheiros com formato proprietário

- Nos ficheiros de formato **proprietário** os dados são escritos da mesma forma que se guardam nas variáveis em memória.
- Isto evita conversões e portanto são de leitura/escrita mais rápida;
- Só quem criou o ficheiro consegue ler o seu conteúdo, pois só ele conhece a sequência de armazenamento;
- As representações de números em memória variam com as linguagens de programação e por isso um programa Java não consegue ler um ficheiro deste tipo escrito por outra linguagem de programação (ex: C ou C#);



Os ficheiros de formato proprietário são mais rápidos que os ficheiros de texto, pois não requerem conversão de dados.

**Stream**

## Stream #1/2

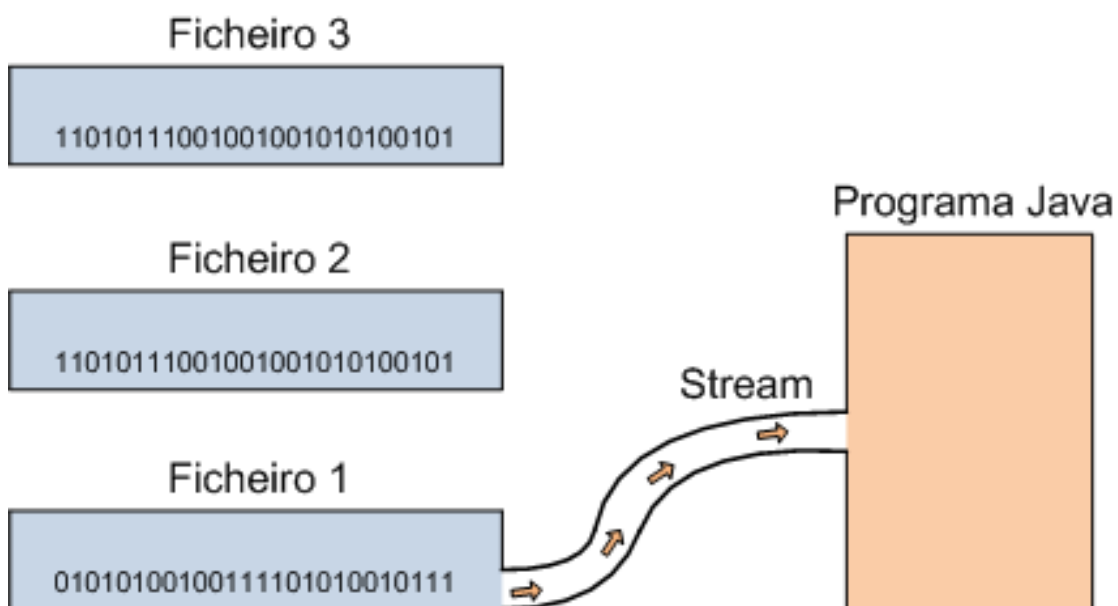
- Os programas Java efectuem operações de **entrada e saída de dados** através de “**Streams**”;
- “stream” é um objeto Java que **comunica com um dispositivo físico** de leitura ou escrita de dados, controlado pelo sistema operativo;
- Através do “stream” os programas Java podem **ler ou escrever bytes sequencialmente**;
  - “Stream” é traduzido para riacho ou ribeiro. Significa que os bytes circulam em fila, todos no mesmo sentido;
- Um objeto de “Stream” disponibiliza os **mesmos métodos** de comunicação para **qualquer dispositivo físico**. Isto permite tratar a comunicação por rede da mesma forma que os ficheiros;



Os programas Java efetuam operações de entrada e saída de dados através de “*streams*”. Um “*stream*” é um objeto que comunica com um dispositivo físico de leitura ou escrita de dados, através do qual os programas Java podem ler ou escrever bytes. O “*stream*” funciona como um **canal** de comunicações com um dispositivo externo. Exemplos de dispositivos externos são: um ficheiro em disco, o teclado, um “*socket*” de comunicações ou a consola standard de entrada ou saída de dados. O “*stream*” disponibiliza sempre os mesmos métodos de comunicação para qualquer dispositivo, o que permite ao programador abstrair-se das diferenças físicas entre esses dispositivos. Os “*streams*” são definidos em Java através de classes pertencentes ao package **java.io.\***.

Quando efetuamos operações de leitura sobre um ficheiro somos notificados do fim de ficheiro quando não há mais dados para ler do “*stream*”. No entanto, se estivermos a ler do teclado ou de um socket de comunicações, os dados a ler poderão ainda não se encontrar no “*stream*”, que neste caso tem um buffer de teclado ou buffer da placa de rede, pelo que a operação de leitura fica bloqueada até chegarem os dados.

## Stream #2/2



O diagrama do slide mostra de forma simplista como funciona um “stream” de entrada de dados. Se o programa Java quisesse escrever dados necessitaria de outro “stream”, agora de output.

**Classes Java usadas para trabalhar com ficheiros**

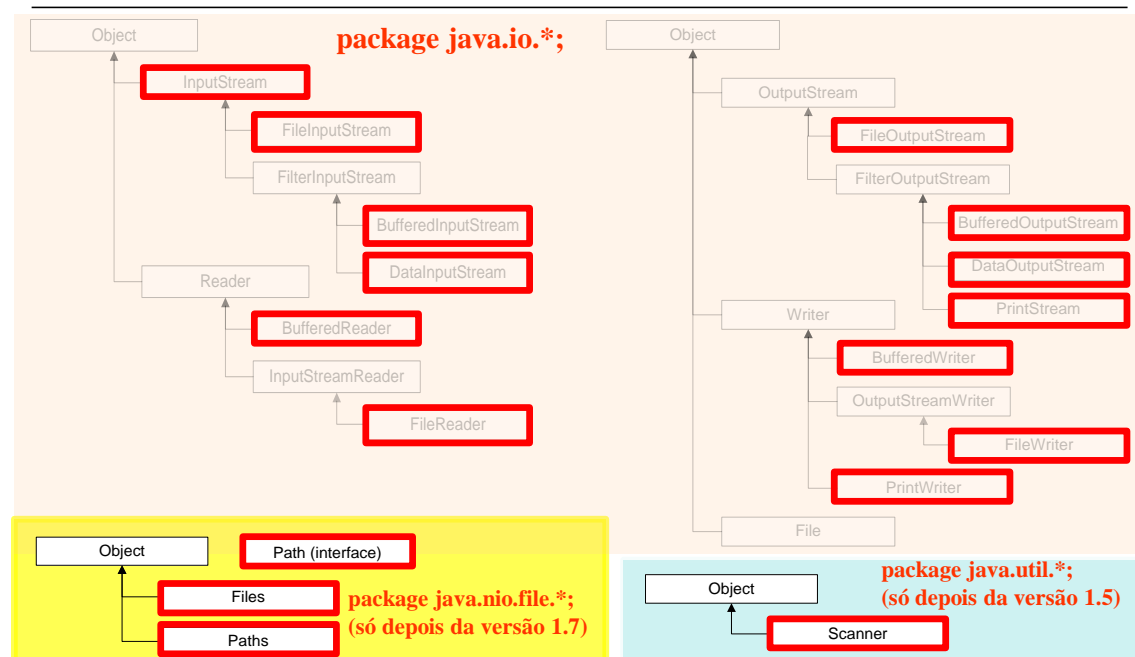
## Sumário

---

- Como o Java trabalha com ficheiros;
- **Classes Java usadas para trabalhar com ficheiros;**
- Escrever dados no standard output;
- Ler dados a partir do standard input;
- Escrever dados em ficheiro de texto;
- Ler dados a partir de ficheiro de texto;
- Escrever dados em formato proprietário;
- Ler dados a partir de formato proprietário;
- Classe File;



## Classes para trabalhar com ficheiros



O slide descreve a hierarquia entre as principais classes do package **java.io**. As que possuem um retângulo vermelho serão utilizadas nos exemplos deste capítulo.

Mostramos também a classe **Scanner**, que pertence ao package **java.util** e que foi criada com o Java 1.5. Esta classe será largamente usada nos nossos exemplos.

Destacamos as classes **Files** e **Paths**, assim como a interface **Path**, todas pertencentes ao package **java.nio.file** e que foram desenvolvidas na versão 1.7 da linguagem Java. A classe **File** do package **java.io** apresentava algumas lacunas, pelo que na versão 1.7 foi desenvolvida uma nova API que ultrapassa as lacunas, adiciona robustez e funcionalidades.

As classes que controlam os “streams” dividem-se em dois grandes grupos:

- **InputStream, OutputStream** e as suas respetivas subclasses permitem ler e escrever **bytes**;
- **Reader, Writer** e as suas respetivas subclasses permitem ler e escrever **caracteres**;

Consoante a natureza dos dados do ficheiro devemos usar **InputStream/OutputStream** ou **Reader/Writer**. Para ler/escrever ficheiros com dados binários (.jpg, .gif, .mp3, etc.) devemos usar **FileInputStream/FileOutputStream** e para ler ficheiros de texto (caracteres) devemos usar **FileReader/FileWriter**.

**Classes para output de caracteres**

## Classes para output de caracteres

- **PrintStream** – Cria um stream de output que aponta para um ficheiro de texto por onde passam bytes agrupados em caracteres. **Usado na classe System apontando para standard output;**
- **FileWriter** – classe adequada para escrever caracteres em ficheiro de texto, criando a stream para o ficheiro;
- **BufferedWriter** – usa buffer de caracteres antes de os escrever no FileWriter, otimizando a operação de escrita. Escreve EOL de acordo com a plataforma;
- **PrintWriter** – o BufferedWriter escreve num **Writer**, que é uma classe abstrata, sendo PrintWriter uma das suas implementações;



O slide descreve as classes que usaremos nos exemplos para escrita de caracteres.

Destacamos o facto da classe PrintWriter combinada com BufferedWriter conseguir escrever linhas de texto que terminam com o caracter adequado a cada plataforma:

- Windows – {CR}{LF} - Carriage Return + Line Feed
- Unix/Linux – {LF} Line Feed
- Mac versões antigas – {CR} Carriage Return
- Mac versões modernas – {LF} Line Feed

O caracter {CR} tem o código ASCII 13 enquanto {LF} é o 10.

**Classes para input de caracteres**

## Classes para input de caracteres

- **InputStream** – Classe abstrata que permite criar um stream de input. Uma implementação é **usada na classe System apontando para standard input**;
- **FileReader** - classe adequada para ler caracteres a partir de um ficheiro de texto. Cria a stream para o ficheiro;
- **BufferedReader** – usa um buffer para otimizar a leitura;
- **Scanner** – tem a capacidade de ler texto e fazer o “parsing” para tipos primitivos;



O slide descreve as classes usadas para input de caracteres.

O *character set* definido nos nossos computadores com sistema operativo Windows é normalmente o “cp1252” que é uma variante feita pela Microsoft do mais antigo Latin-1 (ISO-8859-1). Se o ficheiro de texto for usado apenas na nossa máquina, ou em máquinas com o mesmo *character set*, basta-nos usar `FileReader`. No entanto, se recebermos um ficheiro de uma máquina com um *character set* diferente do nosso, a leitura com um `FileReader` pode fornecer resultados inesperados. Se soubermos qual é o *character set* original, devemos indicá-lo ao Java para que este efetue as conversões corretamente. Nesse caso deveríamos optar pela classe `InputStreamReader`, cujo construtor permite definir o *character set* do ficheiro de origem.



**Classes para output de bytes**

## Classes para output de bytes

- **FileOutputStream** – Cria um stream de output que fica associado a um ficheiro no sistema operativo;
- **BufferedOutputStream** – coloca os bytes num buffer em memória antes de os escrever no output stream;
- **DataOutputStream** – escreve os tipos primitivos (built in types) num OutputStream;



O slide descreve as classes que usaremos para escrita em ficheiros com formato proprietário. Usamos o termo proprietário no contexto em que o ficheiro criado apenas poderá ser lido por outro programa se as duas condições abaixo se verificarem:

- For escrito em Java ou por outra linguagem de programação que use as mesmas normas para representar os dados internamente;
- O programador conhecer a estrutura (ou sequência) com que os dados foram escritos;

No construtor de `FileOutputStream` podemos usar o parâmetro *boolean append*, que indica se os dados a escrever serão adicionados ao fim do ficheiro (*append true*) ou se deve ser criado um novo ficheiro (*append false*) ou substituído por um novo caso já exista.

**Classes para input de bytes**

## Classes para input de bytes

- **FileInputStream** – Cria um stream de input que fica associado a um ficheiro no sistema operativo;
- **BufferedInputStream** – lê os bytes para um buffer em memória para otimizar as leituras;
- **DataInputStream** – consegue ler os tipos primitivos (built in types) a partir de um InputStream;



O slide descreve as classes que usaremos para ler ficheiros com dados em formato proprietário.

**Escrever dados no standard output**

## Sumário

---

- Como o Java trabalha com ficheiros;
- Classes Java usadas para trabalhar com ficheiros;
- Escrever dados no standard output;
- Ler dados a partir do standard input;
- Escrever dados em ficheiro de texto;
- Ler dados a partir de ficheiro de texto;
- Escrever dados em formato proprietário;
- Ler dados a partir de formato proprietário;
- Classe File;



## O que é o Standard Output?

### O que é o Standard Output?

- É a consola do computador onde aparecem as mensagens do sistema operativo.
- É onde a JVM escreve as suas mensagens.
- É um **ficheiro de texto** acedido num programa Java pela variável **System.out**, que é um **PrintStream**;
- Podemos escrever em **out** com dois métodos (funções):
  - `System.out.println()` – escreve no formato pré-definido;
  - `System.out.printf()` – permite formatar output (Java 1.5);



Qualquer programa Java importa automaticamente o package **java.lang**. A classe **System** está definida neste package e implementa algumas funcionalidades importantes para o ambiente de execução, como por exemplo obter a hora do sistema, executar comandos do sistema operativo ou obter o valor de variáveis de ambiente do sistema operativo. Esta classe define três variáveis do tipo *stream*: **in**, **out** e **err** que são variáveis de classe (static) com visualização public:

- **System.out** está ligada ao canal de “*standard output*”, normalmente a consola, sendo um objeto do tipo **PrintStream**.
- **System.err** está ligada ao canal de emissão de mensagens de erro, normalmente também a consola, sendo um objeto do tipo **PrintStream**.
- **System.in** está ligada ao canal de “*standard input*”, normalmente o teclado, sendo um objeto do tipo **InputStream**.

Os IDEs (Netbeans, Eclipse, etc) fazem o redireccionamento dos canais “*standard input e output*” para uma janela interna com o nome Output.

Num programa Java a escrita no *standard output* é feita via classe `PrintStream` que possui os métodos:

- `println()` – escreve usando o formato pré-definido;
- `printf()` – permite uma escrita formatada que “embeleza” o texto. Só está disponível desde o Java 1.5;

**Escrita formatada com printf()**

## Escrita formatada com printf()

- O método `printf()` permite formatar o que se escreve:

```
System.out.printf("String com 30 espaços.....: %30s |%n", s1);
```

Texto que vai ser escrito.

**%30s** representa o sitio onde o valor da variável vai escrito. Representa também a máscara que vai formatar a aparência do valor da variável

Introduzir o valor desta variável no meio do texto anterior .



O método **printf()** permite ao utilizador embelezar o texto. O seu funcionamento baseia-se na definição de uma cadeia de caracteres que contem tudo o que vai ser escrito, tanto o texto fixo como as variáveis cujo conteúdo queremos mostrar. Nesta cadeia de caracteres as variáveis são representadas por máscaras que definem como o seu conteúdo deve ser apresentado. Depois da cadeia de caracteres vem a lista de variáveis, separadas por vírgulas. As máscaras que definem como a variável vai ser escrita começam com o símbolo %.

## Máscara para formatar o output com printf()

%[argument_index\$][flags][width][.precision]conversion	
%	Indica início de máscara de output. Para escrever o carácter '%' introduzir %%; %n é o fim de linha adequado ao hospedeiro;
Conversion	s(string) tY(ano) tm(mês) td(dia) n(new line), d(inteiro), e(número científico) f(número virgula fixa)
argument_index\$	Facultativo → 1\$(1º argumento do output) 2\$(2º argumento). Permite escrever várias vezes o mesmo argumento ou trocar a ordem os argumentos;
flags	Facultativo → -(alinhar esquerda) +(incluir sinal) ((colocar() em número negativo) ,(separador milhares)
width	Facultativo → Número mínimo de caracteres que a variável vai ocupar;
.precision	Facultativo → Dentro de width indica o número de casas decimais que ficarão visíveis. Pode ser usado sem width;



Este slide descreve a forma de construir uma máscara. Mais detalhes podem ser encontrados na definição da classe **java.util.Formatter**.

Uma máscara possui 6 componentes indicados no slide, alguns obrigatórios outros facultativos. Os facultativos estão representadas entre [ ] (parêntesis retos).

Uma máscara começa sempre com o símbolo %. Se o utilizador quiser escrever o carácter % deve indicar %% (dois símbolos seguidos). Se no fim da escrita o utilizador quiser descer de linha ({CR} e/ou {LF}) pode recorrer ao símbolo %n que coloca um fim de linha adequado à plataforma em que o programa está a correr. Relembramos que a maior parte dos editores de texto modernos são flexíveis relativamente ao fim de linha, permitindo conversões de formatos entre as diferentes plataformas (por exemplo de Windows para Unix/Linux).

O parâmetro **Conversion** também é obrigatório e representa o tipo de variável que vai ser escrito. Este parâmetro é importante porque o comando printf() vai proceder a uma conversão do tipo primitivo para texto. O slide mostra alguns dos valores possíveis, dos quais destacamos as letras: **s** para uma String, **tY** para o valor do ano, **g** para um número com parte decimal em vírgula fixa e **e** para um número com parte decimal em vírgula flutuante.

O componente **argument\_index\$** é facultativo e permite alterar a ordem de escrita das variáveis e/ou escrever a mesma variável várias vezes. Por exemplo:



```
System.out.printf("Duas strings ordem inversa.: %2$s, %1$s! %n", s1, s2);
```



A instrução acima vai escrever duas variáveis, s1 e s2, mas no output a variável s2 vai aparecer primeiro que s1, por causa de %2\$s



```
System.out.printf("Duas vezes a mesma variavel: %1$s, %1$s! %n", s1);
```



A instrução acima vai escrever duas vezes a variável s1 graças à máscara **%1\$s**.

O componente **flags** é facultativo, sendo aplicado com números. Entre outros pode assumir os seguintes valores:

- - (sinal de subtrair) para alinhar o número à esquerda, pois por omissão é alinhado à direita;
- + para incluir sinal nos números positivos pois por omissão não é representado;
- ( para envolver os números negativos entre parêntesis curvos;
- , (virgula) para incluir o separador dos milhares. Note que este método tem a capacidade de detetar o LOCALE instalado no computador onde corre a JVM e escrever os números com o formato adequado às definições desse país;

O componente **with** é facultativo e representa o número total de caracteres que a variável vai ocupar. Se a variável requer mais espaço que o definido pelo programador será utilizado o espaço adicional. Este componente pode ser usado com números e com cadeias de caracteres.

O componente **.precision** é facultativo e é usado no contexto de números com parte decimal. Representa o número de algarismos que serão utilizados na parte decimal. A definição do componente inclui um ponto do lado esquerdo.

## Exemplo de máscara de formatação

```
System.out.printf(
    "Real com 30 espaços à esquerda.: %-,30.7f %n",
    1234567890.123456789);
```

Início de máscara de output

Escrever o número em vírgula fixa

%-,30.7f

Alinhar à esquerda

Escrever o separador de milhares (se for o caso)

Reservar 30 espaços, dos quais 7 são para a parte decimal do número



O slide acima mostra um exemplo de utilização da máscara de formatação.

## Exemplo de escrita formatada de Strings

O próximo exemplo mostra como usar as máscaras de formatação com Strings:



```
1 //Escrever no standard output formatando String. Detalhes em java.util.Formatter
2 package capitulo6;
3
4 public class Ex01EscreverStandardString {
5
6     public static void main(String args[]) {
7         //[argument_index$][with][flags]conversion
8         //[argument_index$] 1$ primeiro argumento, 2$ segundo argumento.
9         //Permite escrever várias vezes o mesmo argumento ou trocar argumentos
10        //[with] espaço mínimo ocupado
11        //[flag] ,-(esquerda)
12        //conversion s
13        //[%n escreve new line dependente da plataforma
14        //(\\n escreve sempre à moda do Unix)
15        String s1 = "Ola", s2 = "Bom dia";
16        System.out.printf("Escrever %% %n");
17        System.out.printf("Duas strings.....: %s, %s! %n", s1, s2);
18        System.out.printf("Duas strings ordem inversa.: %2$s, %1$s! %n", s1, s2);
19        System.out.printf("Duas vezes a mesma variavel: %1$s, %1$s! %n", s1);
20        System.out.printf("String com 30 espaços.....: %30s |%n", s1);
21        System.out.printf("String alinhada à esquerda.: %-30s |%n", s1);
22    }
23 }
```

## Exemplo de escrita formatada de datas

O próximo exemplo mostra como usar as máscaras de formatação com datas:





```

1 //Escrever no standard output formatando datas. Detalhes em java.util.Formatter
2 package capitulo6;
3
4 import java.util.Calendar;
5 import java.util.Locale;
6
7 public class Ex02EscreverStandardData {
8
9     public static void main(String args[]) {
10         //[%[argument_index$][with][flags]conversion
11         Calendar cal = Calendar.getInstance();
12         //só usa cal uma vez
13         System.out.printf("Data pré-definida: %tc %n", cal);
14         //usa cal varias vezes
15         System.out.printf("YYYY-mm-dd: %1$tY-%1$tm-%1$td %n", cal);
16         System.out.printf(
17             "HH:MM:SS:SSS TMZ: %1$tH:%1$tM:%1$tS:%1$tL %1$tZ(%1$tz): %n",
18             cal);
19         System.out.printf("Nome do mês: %1$tB %1$tb %n", cal);
20         System.out.printf("Dia da semana: %1$tA %1$ta %n", cal);
21         System.out.printf("Dia do ano: %1$tj %n", cal);
22         System.out.printf(String.format(Locale.ENGLISH,
23             "Nome do mês em inglês: %1$tB %1$tb %n", cal));
24         System.out.printf(String.format(Locale.ENGLISH,
25             "Dia da semana em inglês: %1$tA %1$ta %n", cal));
26         System.out.printf(String.format(new Locale("es", "ES"),
27             "Dia da semana em espanhol: %1$tA %1$ta %n", cal));
28         System.out.printf("Nome do mês alinhado esquerda: **%-10tB** %n", cal);
29     }

```

### Exemplo de escrita formatada de números

O próximo exemplo mostra como usar as máscaras de formatação com números:



```

1 //Escrever standard output numeros formatados. Detalhes em java.util.Formatter
2 package capitulo6;
3
4 import java.util.Locale;
5
6 public class Ex03EscreverStandardNumero {
7
8     public static void main(String args[]) {
9         //[%[argument_index$][flags][width][.precision]conversion
10         System.out.printf("Inteiro em percentagem.....: %d%% %n", 15);
11         System.out.printf(
12             "Inteiro em base 10, 16 e 8.....: %1$d --- %1$x --- %1$o %n",
13             15);
14         System.out.printf(
15             "Inteiro negativo de duas formas.....: %1$d --- %1$(d %n",
16             -33);
17         System.out.printf("Real em virgula fluante.....: %e %n",
18             1234567890.123456789);
19         System.out.printf("Real virgula fluante 30 espaços.....: %30e |%n",
20             1234567890.123456789);
21         System.out.printf("Real virgula fluante 17 decimais.....: %30.25e |%n",
22             1234567890.123456789);
23         System.out.printf("Real em virgula fixa por omissão.....: %f %n",
24             1234567890.123456789);
25         System.out.printf("Real v.f. com 30 espaços.....: %30.4f |%n",
26             1234567890.123456789);
27         System.out.printf("Real v.f. com 30 espaços à esquerda...: %-,30.7f |%n",
28             1234567890.123456789);
29         System.out.printf("Real v.f. 30 espaços e 12 decimais....: %,30.12f |%n",
30             1234567890.123456789);
31         System.out.printf(String.format(Locale.ENGLISH,

```

```
32         "Igual ao anterior mas em inglês.....: %,30.12f |%n",  
33         1234567890.123456789));  
34     }  
35 }
```

## Ler dados a partir do standard input

### Sumário

---

- Como o Java trabalha com ficheiros;
- Classes Java usadas para trabalhar com ficheiros;
- Escrever dados no standard output;
- Ler dados a partir do standard input;
- Escrever dados em ficheiro de texto;
- Ler dados a partir de ficheiro de texto;
- Escrever dados em formato proprietário;
- Ler dados a partir de formato proprietário;
- Classe File;



## O que é o Standard Input?

### O que é o Standard Input?

- É por onde o computador recebe dados do utilizador;
- Normalmente é o **teclado**;
- Para um programa Java é um **ficheiro de texto** acedido pela variável **System.in** que é um `InputStream`;
- Podemos ler de **in** usando dois mecanismos:
  - `System.in.read()` – lê bytes;
  - **Classe Scanner** – lê bytes mas consegue convertê-los em “*built in types*” tendo em conta os “*locales*”, o que é importante no caso dos números e datas (só depois da v.1.5);



O slide descreve o standard input que, como referimos anteriormente, para um programa Java é um objeto da classe `InputStream`. O teclado é o canal de input por omissão, mas pode ser utilizado um ficheiro desde que se faça o redirecionamento usando o comando de sistema operativo `<`.

A leitura de dados provenientes deste stream pode ser feita por dois métodos, como descrito no slide.


**Criar canal para standard input usando Scanner**

## Criar canal para standard input usando Scanner

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

System.in é um InputStream

Scanner recebe bytes do Input Stream e interpreta-os

 **Java Fundamentos**  
© Citeforma 2007

**Capítulo 6 - Ficheiros**

**25**

O slide mostra como podemos usar a classe Scanner para criar um canal de entrada de dados que aponta para o Standard Input.

O método `read()` referido no slide anterior pode ser usado por aplicação direta:



```
char c = System.in.read();
```

Mas este método apresenta vários inconvenientes e por isso na versão 1.5 do Java surgiu a classe Scanner.

**Ler números e Strings usando Scanner**

## Ler números e Strings usando Scanner

- Ler números;
  - `nextInt()`, `nextFloat()`, `nextDouble()`, ...
- Antes de ler podemos verificar o tipo de dado:
  - `hasNextInt()`, `hasNextFloat()`, `hasNextDouble()`, ...
- Ler uma String:
  - `next()`;
  - Pode ser definido um separador de sub-strings;
- Ler uma linha completa para dentro de uma String:
  - `nextLine()`;



A classe `Scanner` possui métodos que recebem bytes provenientes do *stream* de input e os convertem diretamente nos tipos primitivos. A execução destes métodos “consome” os bytes do buffer de input.

Esta classe possui também métodos para verificar se os dados que estão no buffer são de determinado tipo, devolvendo `true` ou `false`. Esta operação de verificação não consome o buffer de input, o que permite depois fazer uma leitura usando o método apropriado.

Um número é uma sequência de algarismos que eventualmente inclui um separador decimal, um ou vários separadores de milhares, um sinal positivo ou negativo e um expoente. Nesta sequência não temos espaço em branco no início, no fim nem no meio. Usar o espaço em branco como separador é um conceito que não faz sentido quando lemos uma String, pois pode conter espaços.

O método usado para ler uma String é **`next()`** que necessita de uma definição prévia de qual o carácter ou sequência de caracteres usados como separador de Strings, como veremos num próximo exemplo.

A classe `Scanner` permite também ler uma linha de texto completa para dentro de uma String. A linha de texto termina com `{CR}` e/ou `{LF}` de acordo com a plataforma.

### Exemplo de Scanner com leitura de números

O próximo exemplo mostra como usar a classe Scanner para ler números a partir do standard input. Forçamos variações ao LOCALE para mostrar as diferentes formas de interpretar os números de acordo com o país e região ativos durante a leitura.



```

1 //Ler int e double diretamente a partir do standard input. Variar Locale.
2 package capitulo6;
3
4 import java.util.Locale;
5 import java.util.Scanner;
6
7 public class Ex04LerStandard {
8
9     public static void main(String args[]) {
10         Scanner teclado = new Scanner(System.in);
11         System.out.println("Ler int:");
12         int n = teclado.nextInt();
13         System.out.printf("n=%d %n", n);
14         //Scanner detecta Locale Portugal (, como separador decimal)
15         System.out.println("Ler double com Locale Portugal:");
16         double x = teclado.nextDouble();
17         System.out.printf("x=%,f %n", x);
18         System.out.println("Ler double com Locale EUA:");
19         teclado.useLocale(Locale.US);
20         x = teclado.nextDouble();
21         System.out.printf("x=%,f %n", x);
22         System.out.println("Ler novamente double com Locale Portugal:");
23         teclado.useLocale(new Locale("pt", "PT"));
24         x = teclado.nextDouble();
25         System.out.printf("x=%,f %n", x);
26     }
27 }

```

### Exemplo de Scanner com separador

O próximo exemplo não usa o standar input porque foi feito com dois objetivos: por um lado mostrar que a classe Scanner pode ler uma String definida pelo utilizador e por outro lado incluir nessa String caracteres separadores, que nos permitem ler números e cadeias de caracteres a partir da mesma origem.



```

1 //Este exercicio usa a classe Scanner para ler vários tipos de dados
2 //a partir da mesma linha. Os diferentes dados estão separados por
3 //uma sequência de caracteres definida como separador
4 package capitulo6;
5
6 import java.util.Scanner;
7
8 public class Ex05LerVariosNaLinha {
9
10     public static void main(String args[]) {
11         //uma linha tem n° de aluno, nome, nota1, nota2 e nota3
12         //definir como limitador a sequência ::
13         //o primeiro campo é um número e não pode ter espaços à esquerda
14         // por causa da classe Scanner
15         String s = "20110001 :: Jose Antonio Almedia :: 10,1 :: 10,2 :: 10,3 ::";
16         //String s = "20110001::Jose Antonio Almedia::10,1::10,2::10,3::";
17         //String s = "20110001::Jose Antonio Almedia::10,1::10,2::10,3";
18         Scanner sIn = new Scanner(s);
19         sIn.useDelimiter("\\s*::\\s*");
20         System.out.println(sIn.nextLong()); //ler numero aluno
21         System.out.println(sIn.next());     //ler nome
22         System.out.println(sIn.nextFloat()); //ler nota1
23         System.out.println(sIn.nextFloat()); //ler nota2

```

```
24     System.out.println(sIn.nextFloat()); //ler nota3
25     sIn.close();
26 }
27 }
```



A String contém uma linha com um número de aluno, nome de aluno e 3 notas em provas de avaliação. Os diferentes componentes estão separados por ::



A classe Scanner vai usar essa String como input porque o seu construtor a recebe como parâmetro.



O método **useDelimiter("\\s\*::\\s\*")** define o separador que deve ser usado pelo Scanner, neste caso os caracteres ::. Este separador pode receber antes ou depois espaços em branco, o que é definido pela sequência **\\s\***. O programa tem várias linhas comentadas que permitem variar a String de input para simular diferentes situações com espaços em branco.



O primeiro campo da String é um número. A classe Scanner não o consegue interpretar se este tiver espaços em branco do lado esquerdo. Esta situação foi evitada nas diferentes situações criadas no exemplo.



**Receber em String e converter em número**

## Receber em String e converter em número #1/2

- Por vezes não é possível ler um número diretamente do canal de input usando `nextInt()` ou `nextDouble()`;
  - Estes métodos são menos flexíveis que a alternativa apresentada a seguir;
- Nestas situações é necessário ler para uma String e depois converter para o número em formato nativo:
  1. Usar o método **parse()** da classe **DecimalFormat** para converter de String em **Number**;
  2. Usar **doubleValue()** para converter de **Number** em double.
  3. **Number** tem conversores para todos os tipos nativos;



Os métodos da classe Scanner que permitem a leitura de números diretamente a partir do canal de input falham em algumas situações, por exemplo quando o número começa por espaços em branco.

Uma forma alternativa e mais flexível é ler o número para dentro de uma String, limpar os espaços em branco à esquerda e à direita (**trim()**) e depois usar o método **parse()** da classe **DecimalFormat** para converter a String em número. À semelhança de Scanner, esta classe é sensível ao LOCALE ativo no programa ou no computador onde corre a JVM. O método `parse()` devolve um objeto do tipo Number, que depois tem que ser convertido num tipo primitivo. Essa conversão é feita pelos métodos disponibilizados por Number, por exemplo `doubleValue()`.

## Receber em String e converter em número #2/2

```
String s="123456.1234567";
DecimalFormat df = new DecimalFormat();
double x = df.parse(s.trim()).doubleValue();
```

DecimalFormat faz o "parse" do número tendo em conta o Locale que estiver activo. A VM deteta o Locale do computador local

A função trim() limpa espaços em branco à esquerda e à direita, que impedem a função parse() de funcionar

A classe DecimalFormat trabalha com Number, que é convertido em double por este método



O slide mostra a sequência de comandos para converter uma String em número usando o método parse() de DecimalFormat.

### Exemplo de conversão de String para número

O próximo exemplo mostra como converter de String em número.



```
1 //Colocar um numero dentro de uma String e converter
2 //para formato numerico nativo (built in type)
3 package capitulo6;
4
5 import java.text.DecimalFormat;
6 import java.text.ParseException;
7
8 public class Ex06ConverterStringNumero {
9
10     public static void main(String args[]) {
11         //em Portugal o separador dos decimais é a virgula
12         String s = "123.456,1234567";
13         //String s="123456.1234567";
14         DecimalFormat df = new DecimalFormat();
15         double x = 0;
16         try {
17             //trim() tira os espacos em branco à esq e direita
18             //df.parse converte a String em numero, com Locale
19             x = df.parse(s.trim()).doubleValue();
20         } catch (ParseException ex) {
21             System.out.println("Erro no formato do número");
22             ex.printStackTrace(System.out);
23             System.exit(-1);
24         }
25         System.out.printf("x= %,20.7f %n", x);
26     }
27 }
```



O número é armazenado numa String. Começamos por limpar os espaços em branco à esquerda e à direita usando o método `trim()` da classe `String`. Depois usamos a String resultante como argumento de entrada no método `parse()` da classe **`DecimalFormat`**, que vai converter a String num objeto da classe `Number`. Este objeto é depois convertido num tipo primitivo usando um dos métodos disponibilizados por `Number`, por exemplo `doubleValue()`.



À semelhança de `Scanner` esta classe é sensível ao `LOCALE` ativo no programa ou no computador onde corre a JVM. O exemplo permite fazer o teste com um número escrito em formato português (virgula como separador decimal e ponto como separador dos milhares) ou formato americano (ponto como separador decimal e virgula como separador dos milhares).



O método `parse()` obriga a tratar a exceção `ParseException` que neste caso será processada por uma instrução `try-catch` que executa três ações:

1. Escreve uma mensagem de erro no standard output (instrução da linha 21);
2. Escreve a pilha de execução (linha 22) no standard output. Isto consiste na escrita da lista dos métodos que foram chamados durante a execução do programa, sua sequência, o número da linha no código fonte onde cada instrução foi invocada e a mensagem de erro gerada pela exceção;
3. Aborta a execução do programa (linha 23) devolvendo ao sistema operativo a indicação de ocorrência de erro;

**Escrever dados em ficheiro de texto**

## Sumário

---

- Como o Java trabalha com ficheiros;
- Classes Java usadas para trabalhar com ficheiros;
- Escrever dados no standard output;
- Ler dados a partir do standard input;
- Escrever dados em ficheiro de texto;
- Ler dados a partir de ficheiro de texto;
- Escrever dados em formato proprietário;
- Ler dados a partir de formato proprietário;
- Classe File;



**Etapas para escrever dados em ficheiro de texto**

## Etapas para escrever dados em ficheiro de texto

1. Criar um canal de output (saída de dados). Esse canal tem que ficar associado a um ficheiro no sistema operativo;
2. Converter os dados em String (texto) e escrever a String no canal de output;
3. Fechar o canal de output para garantir que os “buffers” são “descarregados” no disco;



O slide descreve as etapas que devem ser seguidas para escrever dados num ficheiro de texto.

**Criar canal de output**

## 1-Criar canal de output

O canal de output será a variável `canalOut` que é um objeto do tipo `PrintWriter`. Esta classe escreve no ficheiro em ASCII ou Unicode, de acordo com sistema operativo hospedeiro

Esta String define o nome do ficheiro, que será criado na diretoria do projeto. Podemos usar um PATH longo. Podemos usar a classe `File` (vista mais à frente) para situações mais complexas)

```
PrintWriter canalOut = new PrintWriter(  
    new BufferedWriter(new FileWriter("NotasAlunos1.txt")));
```

Usamos `BufferedWriter` para otimizar as operações de escrita (melhorar I/O)

Criamos um objeto da classe `FileWriter` para criar o stream que aponta para o ficheiro



O slide mostra como criar o canal de output direcionado para um ficheiro que terá o nome **NotasAlunos1.txt**. Este ficheiro será criado na diretoria onde corre a JVM. A sua localização pode ser alterada recorrendo à escrita de um PATH completo, ou utilizando a classe `File`, como descrito no fim deste capítulo. Quando se utiliza um IDE como o Netbeans ou Eclipse o ficheiro será armazenado na diretoria raiz do projeto.

**Escrever no canal de output**

## 2-Escrever no canal de output

O método `println()` permite escrever uma linha de texto sobre o canal de output. Uma linha de texto termina com “fim de linha” ({CR} {LF})

Os números são convertidos em String de forma automática. Entre os dois números é escrito `::` para os separar

```
canalOut.println(numeros[i] + "::" + nomes[i]);
```

```
canalOut.printf("%s::%s::%4.1f::%4.1f::%4.1f::%n",  
    numeros[i],nomes[i],notas[i][0],notas[i][1],notas[i][2]);
```

`canalOutput` também suporta o método `printf()` que permite escrever os números formatados

Usamos `“::”` como separador entre partes



O slide acima indica duas formas alternativas de escrever texto sobre o canal de output, usando métodos que estão disponíveis na classe `PrintWriter`:

- `println()` permite escrever em formato nativo;
- `printf()` permite escrever de forma formatada, sendo utilizadas as máscaras que foram descritas anteriormente neste capítulo.

**Fechar o canal de output**

### 3-Fechar o canal de output

```
canalOut.close();
```

O método `close()` permite fechar o canal de output e libertar os recursos alocados pelo sistema operativo (handlers, locks, ...)



O slide descreve a forma de fechar o canal de output.



**Exercício: escrever em ficheiro de texto**

## Exercício: escrever em ficheiro de texto #1/2

- Escrever num ficheiro de texto os dados relativos a um formando (aluno) que participa num curso:
  - Número de formando (aluno);
  - Nome do formando (aluno);
  - Notas do primeiro segundo e terceiro teste;
- Os dados de um formando devem ocupar uma linha;
- Cada dado deve estar separado do seguinte por ::
- A primeira versão do programa não deve deixar espaços em branco entre o campo e o separador. Na segunda versão pode deixar espaços em branco, exceto no primeiro campo;



## Exercício: escrever em ficheiro de texto #2/2

- Exemplo de ficheiro criado pelo programa:  
Versão 1:

```
20110001::Jose Antonio Almedia::10,1::10,2::10,3::
20110002::Manuel da Fonseca Bragança::11,1::11,2::11,3::
20110003::Antonio Pedro Bruno::12,1::12,2::12,3::
20110004::Catarina Almeida Graça::13,1::13,2::13,3::
```

### Versão 2:

```
20110001 :: Jose Antonio Almedia      :: 10,1 :: 10,2 :: 10,3
20110002 :: Manuel da Fonseca Bragança  :: 11,1 :: 11,2 :: 11,3
20110003 :: Antonio Pedro Bruno         :: 12,1 :: 12,2 :: 12,3
20110004 :: Catarina Almeida Graça     :: 13,1 :: 13,2 :: 13,3
```



Os slides descrevem o enunciado do exercício que visa criar um ficheiro de texto cujas linhas contêm dados relativos à classificação obtida por um formando/aluno em 3 provas de avaliação de um curso. O segundo slide exemplifica o aspeto que as linhas do ficheiro devem ter.

Os dados de um formando estão guardados numa linha. Os campos que compõem uma linha estão descritos no slide e são separados por ::.

Pretende-se criar duas versões do ficheiro de texto: na primeira não deixamos espaços em branco enquanto na segunda deixamos espaços para obter um output mais legível. A segunda versão do output vai criar problemas aos métodos da classe Scanner, como veremos adiante.



```
1 //Escrever num ficheiro as notas dos alunos
2 package capitulo6;
3
4 import java.io.BufferedWriter;
5 import java.io.FileWriter;
6 import java.io.IOException;
7 import java.io.PrintWriter;
8
9 public class Ex07EscreverFicAlunosTexto {
10
11     public static void main(String args[]) {
12         //Definir os arrays numeros, nomes e notas
13         long numeros[] = new long[4];
14         String nomes[] = new String[4];
15         float notas[][] = new float[4][3]; //1º teste, 2º teste e 3º teste
16         //atribuir dados
17         numeros[0] = 20110001;
18         nomes[0] = "Jose Antonio Almedia";
19         notas[0][0] = 10.1f;
20         notas[0][1] = 10.2f;
21         notas[0][2] = 10.3f;
22
23         numeros[1] = 20110002;
24         nomes[1] = "Manuel da Fonseca Bragança";
25         notas[1][0] = 11.1f;
26         notas[1][1] = 11.2f;
27         notas[1][2] = 11.3f;
28
29         numeros[2] = 20110003;
30         nomes[2] = "Antonio Pedro Bruno";
31         notas[2][0] = 12.1f;
32         notas[2][1] = 12.2f;
33         notas[2][2] = 12.3f;
34
35         numeros[3] = 20110004;
36         nomes[3] = "Catarina Almeida Graça";
37         notas[3][0] = 13.1f;
38         notas[3][1] = 13.2f;
39         notas[3][2] = 13.3f;
40         try {
41             //Criar um canal de output
42             PrintWriter canalOut = new PrintWriter(new BufferedWriter(
43                 new FileWriter("NotasAlunos1.txt")));
44
45             //escrever os dados no canal de output
46             for (int i = 0; i < numeros.length; i++) {
47                 //escrever campos separados por :: sem espaços em branco.
48                 //este ficheiro será lido pelos 3 programas de leitura
49                 canalOut.printf("%s::%s::%4.1f::%4.1f::%4.1f::%n",
50                     numeros[i], nomes[i], notas[i][0], notas[i][1], notas[i][2]);
51                 //escrever campos separados por :: mas com espaços em branco.
52                 //só os programas mais flexíveis conseguem ler este formato
53                 //canalOut.printf("%10s :: %-30s :: %4.1f :: %4.1f :: %4.1f %n",
54                     numeros[i], nomes[i], notas[i][0], notas[i][1], notas[i][2]);
55             }
56         } catch (IOException e) {
57             e.printStackTrace();
58         }
59     }
60 }
```

```
56         }
57         //fechar canal de output para garantir que o buffer é descarregado
58         // e que os recursos são libertados
59         canalOut.close();
60         System.out.println("Os dados foram escritos para o ficheiro.");
61     } catch (IOException e) {
62         System.out.println("Problemas ao escrever dados no ficheiro.");
63         e.printStackTrace(System.out);
64         System.exit(-1);
65     }
66 }
67 }
```



Os dados são guardados em 3 arrays. A linha *i* de cada um dos arrays armazena dados relativos ao mesmo formando/aluno. No caso das notas temos um array bidimensional com 3 colunas, sendo cada nota armazenada numa coluna diferente.



Este programa produz a versão 1 do ficheiro de texto (sem espaços em branco). As linhas comentadas 53 e 54 permitem produzir a versão 2 que introduz os espaços em branco. Para isso retiramos os comentários nestas linhas e comentamos as linhas 49 e 50.



O construtor de `FileWriter` obriga a tratar a exceção `IOException` que neste caso será processada por uma instrução `try-catch` que executa três ações:

1. Escreve uma mensagem de erro no standard output;
2. Escreve a pilha de execução no standard output;
3. Aborta a execução do programa devolvendo ao sistema operativo a indicação de ocorrência de erro;

**Ler dados a partir de um ficheiro de texto**

## Sumário

---

- Como o Java trabalha com ficheiros;
- Classes Java usadas para trabalhar com ficheiros;
- Escrever dados no standard output;
- Ler dados a partir do standard input;
- Escrever dados em ficheiro de texto;
- Ler dados a partir de ficheiro de texto;
- Escrever dados em formato proprietário;
- Ler dados a partir de formato proprietário;
- Classe File;



**Etapas para ler dados a partir de um ficheiro de texto**

## Etapas para ler dados a partir de ficheiro de texto

1. Criar um canal de input (entrada de dados). Esse canal tem que ficar associado a um ficheiro já existente no disco do computador;
2. Ler do ficheiro uma linha de cada vez. Converter os dados no formato nativo e guardar nas variáveis apropriadas;
3. Fechar o canal de input para garantir que o sistema operativo liberta os recursos alocados;



O slide descreve as etapas que devem ser seguidas para ler dados a partir de um ficheiro de texto.

**Criar canal de input**

## 1-Criar canal de input

A variável `ficheiroAlunos` é do tipo `Scanner` e tem a capacidade de interpretar os bytes recebidos, convertendo-os em tipos nativos do Java

Nos pontos anteriores usámos a classe `Scanner` sobre o `STANDARD INPUT`, que estava representado pelo objeto `System.in`. Neste caso também usamos um objeto do tipo `InputStream` criado por `FileReader`.

```
Scanner ficheiroAlunos = new Scanner(  
    new BufferedReader(new FileReader("NotasAlunos1.txt")));
```

Os bytes devolvidos pelo stream são agrupados num `BufferedReader` que assim otimiza as operações de leitura.

O objeto da classe `FileReader` cria o stream que aponta para o ficheiro de texto. Esta classe obriga ao tratamento de uma exceção **`FileNotFoundException`**



O slide mostra como criar o canal de input ligado ao ficheiro **NotasAlunos1.txt**. Este ficheiro foi criado na diretoria onde corre a JVM. Se o ficheiro estiver guardado noutra diretoria podemos definir um `PATH` completo, ou utilizar a classe `File` como descrito no fim deste capítulo. Quando se utiliza um IDE como o Netbeans ou Eclipse o ficheiro está armazenado na diretoria raiz do projeto.

**Ler dados a partir do canal de input**

## 2-Ler dados a partir do canal de input

- Vamos usar 3 técnicas alternativas já vistas antes:
  1. **Usar a classe Scanner** para interpretar os diferentes componentes que estão separados por ::
    - a) Esta técnica não funciona se o primeiro número tiver espaços em branco à esquerda;
  2. Ler uma linha para dentro de uma String e separar os diferentes componentes, **sem os converter em tipos nativos**;
  3. Ler uma linha para dentro de uma String, separar os diferentes componentes para subString e converter para os tipos nativos usando **NumberFormat**. Esta técnica suporta espaços entre separadores (::) graças ao método trim();



Cada linha do ficheiro de texto contem os dados de um formando/aluno separados por ::. É preciso separar e converter em tipos nativos os diferentes campos que constituem a linha.

Vamos desenvolver 3 programas que resolvem este problema:

1. Programa 1: usa a classe Scanner que permite interpretar os diferentes campos que compõem a linha, mas não funciona se o primeiro número tiver espaços em branco à esquerda. No entanto suporta espaços em branco à esquerda e direita nos restantes campos;
2. Programa 2: lê uma linha para dentro de uma String e usa a função split() da classe String para obter os diferentes componentes da linha. Os espaços em branco não são retirados;
3. Programa 3: lê uma linha para dentro de uma String e depois executa a seguinte sequência:
  - a. Usa a função split() para separar os componentes (pertence à classe String);
  - b. Usa a função trim() para retirar os espaços em branco à esquerda e direita (pertence à classe String);
  - c. Usa a função parse() para converter as cadeias de caracteres em números. Esta função pertence à classe DecimalFormat e a sua utilização já foi vista num ponto anterior deste manual;
  - d. A função parse() devolve um objeto da classe Number que pode ser convertido num tipo nativo pela utilização de um método conversor, como por exemplo floatValue(). Estes métodos estão disponíveis na classe Number.

**Fechar o canal de input**

### 3-Fechar o canal de input

```
ficheiroAlunos.close();
```

O método `close()` permite fechar o canal de input e libertar os recursos alocados pelo sistema operativo (handlers, locks, ...)



O slide descreve a forma de fechar o canal de input.



**Exercícios para ler um ficheiro de texto****Leitura direta para tipo primitivo**

## Exercício 1: leitura direta para tipo primitivo

- Escrever um programa que leia o ficheiro criado no ponto anterior.
- Deve usar a classe Scanner para ler os caracteres armazenados no ficheiro e converter em formato nativo;
- As diferentes componentes estão separadas por ::
- O ficheiro origem não pode ter espaços em branco do lado esquerdo no primeiro número, pois isso impede a classe Scanner de funcionar. Referimo-nos portanto à versão 1 do ficheiro de texto;



O slide descreve o enunciado do exercício. Uma possível solução é apresentada a seguir:



```
1 package capitulo6;
2
3 import java.io.BufferedReader;
4 import java.io.FileNotFoundException;
5 import java.io.FileReader;
6 import java.util.Scanner;
7
8 public class Ex08LerFicAlunosTexto01 {
9
10     public static void main(String args[]) {
11         long numeros[] = new long[4];
12         String nomes[] = new String[4];
13         float notas[][] = new float[4][3]; //1º teste, 2º teste e 3º teste
14         int i = 0; //contador de linhas
15         try {
16             Scanner ficheiroAlunos = new Scanner(new BufferedReader(
17                 new FileReader("NotasAlunos1.txt")));
18             ficheiroAlunos.useDelimiter("\\s*::\\s*");
19             while (ficheiroAlunos.hasNext()) {
20                 numeros[i] = ficheiroAlunos.nextLong();
21                 nomes[i] = ficheiroAlunos.next();
22                 notas[i][0] = ficheiroAlunos.nextFloat();
23                 notas[i][1] = ficheiroAlunos.nextFloat();
24                 notas[i][2] = ficheiroAlunos.nextFloat();
25                 i++;
26                 System.out.println("Foi lida a linha = " + i);
27             }
28             //terminada a leitura vamos fechar o canal para libertar recursos
```

```
29     ficheiroAlunos.close();
30     System.out.println("Os dados foram lidos do ficheiro.");
31 } catch (FileNotFoundException e) {
32     System.out.println("Problema com o ficheiro.");
33     e.printStackTrace(System.out); //Em caso de erro escrever a causa
34     System.exit(-1);
35 }
36
37 System.out.println("Escrever o conteudo dos arrays para confirmar:");
38 for (i = 0; i < numeros.length; i++) {
39     System.out.printf("%d--%s--%4.1f--%4.1f--%4.1f %n",
40         numeros[i], nomes[i], notas[i][0], notas[i][1], notas[i][2]);
41 }
42 }
43 }
```



A abertura do canal para leitura do ficheiro é feita com as instruções descritas neste manual. A classe **FileReader()** obriga a tratar a exceção **FileNotFoundException** cujo processamento é feito recorrendo à instrução **try-catch**. Se a exceção ocorrer o programa escreve uma mensagem de alerta, imprime o conteúdo do stack (pilha de execução) e termina devolvendo um código de erro diferente de zero, o que informa o sistema operativo de que algo correu mal.



Utilizamos um ciclo **while()** para ler as várias linhas. O método **hasNext()** devolve **true** enquanto houver mais linhas para ler.



Os campos são lidos recorrendo a métodos da classe **Scanner** e respeitando a sequência de campos previamente definida. Se essa sequência não for respeitada teremos um erro de leitura relativo ao formato dos números.



No fim da leitura escrevemos o conteúdo dos arrays para confirmar que os diferentes campos foram lidos corretamente.

**Leitura de uma linha e divisão em partes**

## Exercício2: leitura de linha e divisão em partes

- Escrever um programa que leia o ficheiro criado no ponto anterior.
- Deve usar a classe Scanner para ler uma linha de cada vez;
- Essa linha deve ser dividida nas suas diferentes componentes que estão separadas por ::;
- As componentes não são convertidas nos tipos nativos;



O slide descreve o enunciado do exercício. Uma possível solução é apresentada a seguir:



```
1 //Ler notas de alunos a partir de ficheiro de texto
2 //Lê uma linha para dentro de uma string.
3 //Parte a linha em pedaços usando o separador ::
4 package capitulo6;
5
6 import java.io.BufferedReader;
7 import java.io.FileNotFoundException;
8 import java.io.FileReader;
9 import java.util.Scanner;
10
11 public class Ex09LerFicAlunosTexto02 {
12
13     public static void main(String args[]) {
14         try { //como lidamos com ficheiros é obrigatório tratar a exceção
15             //Criar canal de input
16             Scanner ficheiroAlunos = new Scanner(new BufferedReader(
17                 new FileReader("NotasAlunos1.txt")));
18             String linha; //String que guarda uma linha do ficheiro de texto
19             String[] partes; //Linha dividida em partes guardadas neste array
20             //Começando na primeira linha, enquanto o ficheiro tiver linhas...
21             while (ficheiroAlunos.hasNextLine()) {
22                 linha = ficheiroAlunos.nextLine();
23                 //System.out.println(linha);
24                 //dividir uma linha em partes, usando :: como separador
25                 partes = linha.split("::");
26                 //Partes guardadas num array. Vamos escrever todas as partes
27                 for (int i = 0; i < partes.length; i++) {
28                     System.out.print(partes[i] + "--");
29                 }
30                 System.out.println("");
31             }
32         }
33     }
34 }
```

```
32         //terminada a leitura vamos fechar o canal para libertar recursos
33         ficheiroAlunos.close();
34         System.out.println("Os dados foram lidos do ficheiro.");
35     } catch (FileNotFoundException e) {
36         e.printStackTrace(System.out); //Em caso de erro escrever a causa
37         System.exit(-1);
38     }
39 }
40 }
```



A abertura do canal é feita da mesma forma que no exemplo anterior.



Utilizamos um ciclo `while()` para ler as várias linhas. O método `hasNextLine()` devolve **true** enquanto houver mais linhas para ler.



Uma linha é lida recorrendo ao método `nextLine()` que pertence à classe `Scanner`, sendo armazenada na `String` **linha**. Em seguida usamos o método **`split()`** da classe `String` para separar os diferentes componentes. Este método recebe como argumento a sequência de caracteres que é usada como separador dos campos e devolve um array de `Strings`, que é preenchido com os vários campos. No nosso caso os campos ficam armazenados no array **partes**. Cada “parte” é escrita para confirmação de uma correta leitura.



O construtor de `FileReader` obriga a tratar a exceção `FileNotFoundException` que neste caso será processada por uma instrução `try-catch` que executa duas ações:

1. Escreve a pilha de execução no standard output que consiste numa lista dos métodos que foram chamados durante a execução do programa, sua sequência, o número da linha no código fonte onde essa instrução foi invocada e a mensagem de erro gerada pela exceção;
2. Aborta a execução do programa devolvendo ao sistema operativo a indicação de ocorrência de erro (-1);

**Leitura de uma linha, divisão e conversão**

## Exercício3: leitura de linha, divisão e conversão

- Escrever um programa que leia o ficheiro criado no ponto anterior.
- Usar a classe Scanner para ler uma linha de cada vez;
- Essa linha deve ser dividida nas suas diferentes componentes que estão separadas por ::
- As componentes são convertidas nos tipos nativos;
- O programa dever ser capaz de ler a versão 1 e 2 do ficheiro de texto (sem espaços e com espaços entre os separadores de campo ::);



O slide descreve o enunciado do exercício. Uma possível solução é apresentada a seguir:



```
1 //Ler notas de alunos a partir de ficheiro de texto
2 //Lê uma linha para dentro de uma string.
3 //Parte a linha em pedaços usando o separador ::
4 //Converte cada parte no tipo nativo usando NumberFormat
5 package capitulo6;
6
7 import java.io.BufferedReader;
8 import java.io.FileNotFoundException;
9 import java.io.FileReader;
10 import java.text.NumberFormat;
11 import java.text.ParseException;
12 import java.util.Scanner;
13
14 public class Ex10LerFicAlunosTexto03 {
15
16     public static void main(String args[]) {
17         //Definir arrays que vão receber os dados contidos no ficheiro (4 linhas)
18         long numeros[] = new long[4];
19         String nomes[] = new String[4];
20         float notas[][] = new float[4][3]; //1º teste, 2º teste e 3º teste
21         try { //como lidamos com ficheiros é obrigatório tratar a exceção
22             //Criar canal de input
23             Scanner ficheiroAlunos = new Scanner(new BufferedReader(
24                 new FileReader("NotasAlunos1.txt")));
25             //nf vai formatar numeros de acordo com LOCALE activo neste momento
26             NumberFormat nf = NumberFormat.getInstance();
27             String linha; //guarda uma linha do ficheiro de texto
28             String[] partes; //Linha dividida em partes guardadas neste array
29             int i = 0; //contador de linhas
30             //enquanto o ficheiro tiver linhas...
31             while (ficheiroAlunos.hasNextLine()) {
```

```

32     linha = ficheiroAlunos.nextLine(); //ler uma linha
33     //dividir a linha em partes, usando :: como separador.
34     partes = linha.split("::"); //As partes ficam guardadas num array.
35     //atribuir as partes aos arrays que guardam conteúdos
36     //parse string para long
37     numeros[i] = nf.parse(partes[0].trim()).longValue();
38     //trim() limpa espaços em branco no inicio e fim
39     nomes[i] = partes[1].trim();
40     //parse string para float
41     notas[i][0] = nf.parse(partes[2].trim()).floatValue();
42     notas[i][1] = nf.parse(partes[3].trim()).floatValue();
43     notas[i][2] = nf.parse(partes[4].trim()).floatValue();
44     i++;
45     System.out.println("Foi lida a linha = " + i);
46 }
47 //terminada a leitura vamos fechar o canal para libertar recursos
48 ficheiroAlunos.close();
49 System.out.println("Os dados foram lidos do ficheiro.");
50 } catch (FileNotFoundException e) { //tratar erro ao abrir ficheiro
51     e.printStackTrace(System.out); //Em caso de erro escrever a causa
52     System.exit(-1);
53 } catch (ParseException e) { //tratar erro de formato de numero (parse)
54     e.printStackTrace(System.out);
55     System.exit(-1);
56 }
57
58 System.out.println("Escrever o conteudo dos arrays para confirmar:");
59 for (int i = 0; i < numeros.length; i++) {
60     System.out.printf("%d--s--%4.1f--%4.1f--%4.1f %n",
61         numeros[i], nomes[i], notas[i][0], notas[i][1], notas[i][2]);
62 }
63 }
64 }

```



A abertura do canal é feita da mesma forma que no exemplo anterior.



Utilizamos um ciclo `while()` para ler as várias linhas. O método `hasNextLine()` devolve **true** enquanto houver mais linhas para ler.



A leitura das linhas e a sua separação em componentes é idêntica ao exemplo anterior.



As partes são convertidas para os tipos nativos tendo em conta a sequência com que os campos foram criados no ficheiro. Se houver troca na sequência de campos que origine troca nos tipos de dados teremos um erro na conversão (parsing).



Cada “parte” é primeiro submetida ao método `trim()` para limpar os espaços em branco à esquerda e à direita. Depois usamos o método `parse()` da classe **DecimalFormat** para converter a **String** num objeto da classe **Number**, que depois é convertido num tipo primitivo recorrendo a um dos métodos disponibilizados nesta classe, por exemplo `floatValue()`. À semelhança de `Scanner`, esta classe é sensível ao `LOCALE` ativo no programa ou no computador onde corre a JVM.



O construtor de **FileReader** obriga a tratar a exceção **FileNotFoundException**. O método `parse()` obriga a tratar a exceção **ParseException**. Estas serão processadas cada uma pela sua instrução **try-catch** que executa duas ações:

1. Escreve a pilha de execução no standard output;
2. Aborta a execução do programa devolvendo ao sistema operativo a indicação de ocorrência de erro (-1);

**Escrever dados em ficheiro com formato proprietário**

## Sumário

---

- Como o Java trabalha com ficheiros;
- Classes Java usadas para trabalhar com ficheiros;
- Escrever dados no standard output;
- Ler dados a partir do standard input;
- Escrever dados em ficheiro de texto;
- Ler dados a partir de ficheiro de texto;
- Escrever dados em formato proprietário;
- Ler dados a partir de formato proprietário;
- Classe File;



**Formato interno para representação de dados**

## Formato interno para representação de dados

- O Java usa as seguintes normas para **representar** os dados dos tipos pré-definidos:
  - UTF-8;
  - ISO/IEC 10646;
  - "File System Safe UCS Transformation Format (FSS\_UTF)" escrito por X/Open Company Ltd;
  - Documento número P316 incluído em "X/Open Preliminary Specification";



O Java foi criado pela empresa Sun Microsystems entretanto adquirida pela Oracle. A Sun é muito conhecida pelo seu sistema operativo Solaris baseado um Unix. A Sun participou no consórcio X/Open que pretendia uniformizar os diferentes sistemas Unix. Por esse motivo os tipos numéricos primitivos existentes no Java são armazenados em memória seguindo as recomendações produzidas pelo X/Open.

Em relação aos caracteres e Strings a Sun adotou o UTF (Unicode) para representação interna.



## Ler e escrever dados dos tipos primitivos

- As classes `DataInputStream` e `DataOutputStream` possuem métodos para ler e escrever dados dos tipos primitivos:

Tipo de dado	Escrever	Ler
<code>int</code>	<code>writeInt()</code>	<code>readInt()</code>
<code>double</code>	<code>writeDouble()</code>	<code>readDouble()</code>
<code>char[ ]</code>	<code>writeChars()</code>	<code>readChars()</code>
<code>String</code>	<code>writeUTF()</code>	<code>readUTF()</code>



Como foi visto no início do capítulo as classes **`DataInputStream`** e **`DataOutputStream`** trabalham com bytes, não os interpretando em caracteres.

Estas classes disponibilizam métodos que permitem escrever num ficheiro as sequências de bits, agrupadas em bytes, de forma idêntica à forma como são armazenados em memória, seguindo as normas descritas no slide anterior.

Também disponibilizam métodos que permitem ler em sequência um número de bytes adequado a cada tipo primitivo e em seguida interpretar os respetivos bits de acordo com a norma, obtendo assim um valor do tipo primitivo.

**Etapas para escrever dados em ficheiro com formato proprietário**

## Etapas para escrever em ficheiro proprietário

1. Criar um canal de output (saída de dados). Esse canal tem que ficar associado a um ficheiro no sistema operativo;
2. Escrever os dados usando métodos adequados. Usar sempre a mesma sequência de campos e registar essa sequência;
3. Fechar o canal de output para garantir que os “buffers” são “descarregados” no disco;



O slide descreve as etapas que devem ser seguidas para escrever dados num ficheiro usando o formato dos tipos primitivos.

A combinação entre formatos primitivos e sequência de escrita dos diferentes dados tornam o ficheiro produzido desta forma muito difícil de ler por parte de outros programas, e por isso usamos a designação “formato proprietário”. Para que seja possível ler este ficheiro é necessário:

1. Que o programa leitor conheça a sequência de armazenamento dos diferentes campos;
2. Que a linguagem desse programa consiga ler os bytes e interpretá-los seguindo as mesmas normas com que foram escritos;

**Criar canal de output**

## 1-Criar canal de output

O canal de output será a variável `canalOut` que é um objeto do tipo `DataOutputStream`. Esta classe escreve no ficheiro seguindo os formatos definidos pela norma para cada tipo de dado

Esta String define o nome do ficheiro, que será criado na diretoria do projeto. Podemos usar um PATH longo. Podemos usar a classe `File` (vista mais à frente) para situações mais complexas)

```
DataOutputStream canalOut = new DataOutputStream(  
    new BufferedOutputStream(  
        new FileOutputStream("NotasAlunos1.dat"))));;
```

Usamos `BufferedOutputStream` para otimizar as operações de escrita (melhorar I/O)

Usamos um objeto da classe `FileOutputStream` para criar o stream que aponta para o ficheiro. Esta classe está preparada para lidar com bytes e pode gerar a exceção `FileNotFoundException`



O slide mostra como criar o canal de output direcionado para um ficheiro que terá o nome **NotasAlunos1.dat**. A extensão foi escolhida para mostrar que não se trata de um ficheiro de texto. Este ficheiro será criado na diretoria onde corre a JVM. A sua localização pode ser alterada recorrendo à escrita de um PATH completo, ou utilizando a classe `File`, como descrito no fim deste capítulo. Quando se utiliza um IDE como o Netbeans ou Eclipse o ficheiro será armazenado na diretoria raiz do projeto.

**Escriver no canal de output**

## 2-Escriver no canal de output

```
canalOut.writeLong(numeros[i]);  
canalOut.writeUTF(nomes[i]);  
canalOut.writeFloat(notas[i][0]);  
canalOut.writeFloat(notas[i][1]);  
canalOut.writeFloat(notas[i][2]);
```

canalOutput possui um método para escrever cada um dos tipos pré-definidos. Os respetivos bytes são escritos no ficheiro da mesma forma que são armazenados em memória. Estes métodos obrigam a tratar a exceção **IOException**

Todos estes métodos escrevem bytes. A sequência de escrita dos diferentes campos é importante, pois dita a forma como os bytes devem ser interpretados na leitura. Uma leitura na sequência errada pode não dar erro, mas os dados lidos não serão os que foram escritos



O slide mostra alguns dos métodos disponibilizados pela classe **DataOutputStream** para escrever dados em formato primitivo. Estão disponíveis métodos para todos os tipos primitivos.

**Fechar o canal de output**

### 3-Fechar o canal de output

```
canalOut.close();
```

O método `close()` permite fechar o canal de output e libertar os recursos alocados pelo sistema operativo (handlers, locks, ...)



O slide descreve a forma de fechar o canal de output.

**Exercício: escrever dados num ficheiro usando formatos proprietário**

## Exercício: escrever em ficheiro proprietário

- Escrever num ficheiro os dados relativos a um formando (aluno) que participa num curso:
  - Número de formando (aluno);
  - Nome do formando (aluno);
  - Notas do primeiro segundo e terceiro teste;
- Os dados devem ser escrito respeitando os formatos dos tipos primitivos;
- Não há separador entre os campos razão pela qual a sequência de escrita deve ser respeitada para todos os formandos/alunos;



O slide descreve o enunciado do exercício. É muito importante que a sequência de escrita das diferentes linhas seja respeitada, pois ao contrário dos ficheiros de texto, neste caso não usaremos separadores entre os campos. Se a leitura for feita respeitando esta sequência os separadores não são necessários.



```
1 //Escrever num ficheiro as notas dos alunos
2 package capitulo6;
3
4 import java.io.*;
5
6 public class Ex11EscreverFicAlunosBinario {
7
8     public static void main(String args[]) {
9         //Definir os arrays numeros, nomes e notas
10        long numeros[] = new long[4];
11        String nomes[] = new String[4];
12        float notas[][] = new float[4][3]; //1º teste, 2º teste e 3º teste
13        //atribuir dados
14        numeros[0] = 20110001;
15        nomes[0] = "Jose Antonio Almedia";
16        notas[0][0] = 10.1f;
17        notas[0][1] = 10.2f;
18        notas[0][2] = 10.3f;
19
20        numeros[1] = 20110002;
21        nomes[1] = "Manuel da Fonseca Bragança";
22        notas[1][0] = 11.1f;
23        notas[1][1] = 11.2f;
24        notas[1][2] = 11.3f;
25
26        numeros[2] = 20110003;
27        nomes[2] = "Antonio Pedro Bruno";
```

```
28     notas[2][0] = 12.1f;
29     notas[2][1] = 12.2f;
30     notas[2][2] = 12.3f;
31
32     numeros[3] = 20110004;
33     nomes[3] = "Catarina Almeida Graça";
34     notas[3][0] = 13.1f;
35     notas[3][1] = 13.2f;
36     notas[3][2] = 13.3f;
37     DataOutputStream canalOut;
38     try {
39         //Criar um canal de output
40         canalOut = new DataOutputStream(new BufferedOutputStream(
41             new FileOutputStream("NotasAlunos1.dat")));
42
43         //escrever os dados no canal de output
44         for (int i = 0; i < numeros.length; i++) {
45             canalOut.writeLong(numeros[i]);
46             canalOut.writeUTF(nomes[i]);
47             canalOut.writeFloat(notas[i][0]);
48             canalOut.writeFloat(notas[i][1]);
49             canalOut.writeFloat(notas[i][2]);
50         }
51         //fechar canal de output para garantir que o buffer é descarregado
52         // e libertar recursos
53         canalOut.close();
54         System.out.println("Os dados foram escritos para o ficheiro.");
55     } catch (FileNotFoundException e) {
56         System.out.println("Problema ao abrir canal de output.");
57         e.printStackTrace(System.out);
58         System.exit(-1);
59     } catch (IOException e) {
60         System.out.println("Problema ao escrever dados no ficheiro.");
61         e.printStackTrace(System.out);
62         System.exit(-1);
63     }
64 }
65 }
```



Os dados são guardados em arrays idênticos aos utilizados no exercício que cria um ficheiro de texto.



Os dados guardados nos arrays são escritos no canal de output recorrendo aos métodos da classe **DataOutputStream**.



O construtor de **FileOutputStream** obriga a tratar a exceção **FileNotFoundException** enquanto os métodos de **DataOutputStream** obrigam a tratar a exceção **IOException**. Estas exceções são processadas por instruções **try-catch** que executam duas ações: escrever a pilha de execução no standard output e abortar a execução do programa devolvendo ao sistema operativo a indicação de erro (-1).

**Ler a partir de ficheiro com dados em formato proprietário**

## Sumário

---

- Como o Java trabalha com ficheiros;
- Classes Java usadas para trabalhar com ficheiros;
- Escrever dados no standard output;
- Ler dados a partir do standard input;
- Escrever dados em ficheiro de texto;
- Ler dados a partir de ficheiro de texto;
- Escrever dados em formato proprietário;
- Ler dados a partir de formato proprietário;
- Classe File;





**Etapas para ler dados escritos em ficheiro com formato proprietário**

## Etapas para ler dados de ficheiro proprietário

1. Criar um canal de input (entrada de dados). Esse canal tem que ficar associado a um ficheiro já existente no disco do computador;
2. Recordar a sequência com que os dados foram escritos. Ler os dados respeitando a sequência, usando os métodos adequados;
3. Fechar o canal de input para garantir que o sistema operativo liberta os recursos alocados;



O slide descreve as etapas que devem ser seguidas para ler dados a partir de um ficheiro onde estes foram escritos em formato idêntico à representação interna das variáveis primitivas do Java.

**Criar canal de input**

## 1-Criar canal de input

A variável `canalIn` é do tipo `DataInputStream` e interpreta os bytes recebidos de acordo com o tipo de dados que está a ler. Por isso é importante ler na mesma sequência com que foi escrito, pois uma troca provoca uma interpretação errada

```
DataInputStream canalIn = new DataInputStream(  
    new BufferedInputStream(  
        new FileInputStream("NotasAlunos1.dat")));;
```

Os bytes devolvidos pelo stream são agrupados num `BufferedInputStream`, que assim otimiza as operações de leitura.

O objeto da classe `FileInputStream` cria o stream que aponta para o ficheiro criado no ponto anterior e que está na diretoria do projeto. Pode gerar uma **`FileNotFoundException`**



O slide mostra como criar o canal de input direcionado para um ficheiro criado no ponto anterior. Este ficheiro está na diretoria onde corre a JVM. A sua localização pode ser alterada recorrendo à escrita de um PATH completo, ou utilizando a classe `File`, como descrito no fim deste capítulo. Quando se utiliza um IDE como o Netbeans ou Eclipse o ficheiro será armazenado na diretoria raiz do projeto.

**Ler dados a partir do canal de input**

## 2-Ler dados a partir do canal de input

```
numeros[i] = canalln.readLong();  
nomes[i]   = canalln.readUTF();  
notas[i][0] = canalln.readFloat();  
notas[i][1] = canalln.readFloat();  
notas[i][2] = canalln.readFloat();
```

canalln possui um método para ler cada um dos tipos pré-definidos. Os respetivos bytes são lidos do ficheiro e interpretados de acordo com o tipo pré-definido. Pode gerar uma **IOException**

Todos estes métodos leem bytes. Na leitura deve ser respeitada a sequência com que os dados foram escritos, para que a interpretação dos bytes esteja correta e os dados sejam iguais ao que foi escrito



O slide mostra alguns dos métodos disponibilizados pela classe **DataInputStream** para ler dados a partir do formato primitivo. Estão disponíveis métodos para todos os tipos primitivos.

**Fechar o canal de input**

### 3-Fechar o canal de input

```
canalIn.close();
```

O método `close()` permite fechar o canal de input e libertar os recursos alocados pelo sistema operativo (handlers, locks, ...)



O slide descreve a forma de fechar o canal de output.

**Exercício: ler dados escritos em ficheiro com formato proprietário**

## Exercício: ler de ficheiro proprietário

- Ler os dados escritos no ficheiro criado no ponto anterior. Os campos foram escritos nesta sequência:
  - Número de formando (aluno);
  - Nome do formando (aluno);
  - Notas do primeiro segundo e terceiro teste;
- Os dados foram escritos respeitando os formatos dos tipos primitivos;
- Não há separador entre os campos razão pela qual a sequência de leitura deve seguir a sequência de escrita;



O slide descreve o enunciado do exercício. Na solução proposta a leitura dos dados respeita a sequência com que foram escritos.



```
1 package capitulo6;
2
3 import java.io.*;
4
5 public class Ex12LerFicAlunosBinario {
6
7     public static void main(String args[]) {
8         long numeros[] = new long[4];
9         String nomes[] = new String[4];
10        float notas[][] = new float[4][3]; //1º teste, 2º teste e 3º teste
11        int i = 0; //contador de linhas
12        DataInputStream canalIn;
13        try {
14            //Criar um canal de input
15            canalIn = new DataInputStream(new BufferedInputStream(
16                new FileInputStream("NotasAlunos1.dat")));
17            //ler os dados a partir do canal de input
18            try {
19                //quando chegar ao fim do ficheiro gera exceção e sai do ciclo
20                while (true) {
21                    numeros[i] = canalIn.readLong();
22                    nomes[i] = canalIn.readUTF();
23                    notas[i][0] = canalIn.readFloat();
24                    notas[i][1] = canalIn.readFloat();
25                    notas[i][2] = canalIn.readFloat();
26                    i++;
27                    System.out.println("Foi lida a linha = " + i);
28                }
29            } catch (EOFException e) {
30                System.out.println("Os dados foram lidos do ficheiro.");
31            }
32        }
33    }
34 }
```

```
31     }
32     //fechar o canal de input
33     canalIn.close();
34 } catch (FileNotFoundException e) {
35     System.out.println("Problemas ao abrir o ficheiro.");
36     e.printStackTrace(System.out);
37     System.exit(-1);
38 } catch (IOException e) {
39     System.out.println("Problemas ao ler dados a partir do ficheiro.");
40     e.printStackTrace(System.out);
41     System.exit(-1);
42 }
43
44 System.out.println("Escrever o conteudo dos arrays para confirmar:");
45 for (i = 0; i < numeros.length; i++) {
46     System.out.printf("%d--%s--%4.1f--%4.1f--%4.1f %n",
47         numeros[i], nomes[i], notas[i][0], notas[i][1], notas[i][2]);
48 }
49 }
50 }
```



Os dados são lidos do canal de input usando os métodos disponibilizados pela classe **DataOutputStream**, que os guardam diretamente nos arrays em formato nativo.



Como não existem separadores entre os campos e como não há necessidade de converter cadeias de caracteres em números (fazer parsing) a execução deste programa requer menos processamento que os de leitura a partir de ficheiro de texto. No entanto o programador é obrigado a conhecer os tipos de dados usados na escrita e a sua sequência.



O construtor de **FileInputStream** obriga a tratar a exceção **FileNotFoundException** enquanto os métodos de **DataInputStream** obrigam a tratar a exceção **IOException**. Estas exceções são processadas por instruções **try-catch** que executam duas ações: escrever a pilha de execução no standard output e abortar a execução do programa devolvendo ao sistema operativo a indicação de erro (-1).

**Classe File**

## Sumário

---

- Como o Java trabalha com ficheiros;
- Classes Java usadas para trabalhar com ficheiros;
- Escrever dados no standard output;
- Ler dados a partir do standard input;
- Escrever dados em ficheiro de texto;
- Ler dados a partir de ficheiro de texto;
- Escrever dados em formato proprietário;
- Ler dados a partir de formato proprietário;
- Classe File;



**A utilidade da classe File**

## Utilidade da classe File

- Não manipula o conteúdo do ficheiro;
- Permite:
  - Esconder as diferenças entre sistemas operativos;
  - Determinar se um “path” é absoluto ou relativo;
  - Determinar se há permissões para ler/escrever num ficheiro;
  - Remover um ficheiro e criar uma directoria;
  - Determinar a dimensão de um ficheiro e a data da última alteração;
  - Obter a lista de ficheiros que estão dentro de uma directoria;



Como a linguagem Java é multiplataforma é importante dar ao programador mecanismos que permitam esconder as diferenças existentes entre os diferentes sistemas operativos. O slide mostra que a classe **File**, que pertence ao package **java.io**. Esta classe não possui métodos para trabalhar com o conteúdo do ficheiro mas sim para lidar com o sistema operativo ao nível do gestor de ficheiros (filesystem).



**Usando a classe File (antes do Java 1.7)**

## Usando a classe File (antes do Java 1.7)

```
File fic = new File(System.getProperty("user.dir"));
File fic = new File(System.getProperty("user.dir") + File.separatorChar +
    "NotasAlunos1.txt ");
```

Nome → `fic.getName();`  
Path → `fic.getPath();`  
Canonical name → `fic.getCanonicalPath();`  
Absolute path → `fic.getAbsolutePath();`  
Existe? → `fic.exists();`  
É uma directoria? → `fic.isDirectory();`  
É um ficheiro? → `fic.isFile();`  
Está escondido? → `fic.isHidden();`  
Permissão de read? → `fic.canRead();`  
Permissão de write? → `fic.canWrite();`  
Dimensão → `fic.length();`  
Data da última alteração → `fic.lastModified();`



O slide mostra alguns dos métodos disponibilizados pela classe File.

**Nova implementação de Java I/O na versão 1.7**

## Nova implementação de Java I/O na versão 1.7

- A classe `java.io.File` tinha vários problemas;
- Toda a interface de I/O foi reescrita com o Java 1.7;
- Foram adicionadas novas funcionalidades:
  - Classes `Files`, `Path` e `Paths` pertencentes a `java.nio.file`;
  - Maior fiabilidade nos processos, que agora devolvem erros específicos em vez de erros genéricos;
  - Opção de `rename` agora funciona em todas as plataformas;
  - Suporte para “symbolic links” (usados nas plataformas Unix/Linux);
  - Permissões de ficheiro, dono do ficheiro e atributos de segurança;



A classe `File` apresentava alguns problemas que são descritos no slide. Por esse motivo a versão 1.7 do Java apresenta uma nova interface de I/O, que além de corrigir estes problemas disponibiliza mais funcionalidades.

**Usando a nova API de I/O da versão Java 1.7**

## Usando a nova API de I/O do Java 1.7

```
java.nio.file.Files;    java.nio.file.Path;    java.nio.file.Paths;  
Path fic = Paths.get(System.getProperty("user.dir") + "/NotasAlunos1.txt");  
  
Nome → fic.getFileName();  
Path → fic.getParent();  
Absolute path → fic.toAbsolutePath();  
Número de subdirectorias → fic.getNameCount();  
  
Existe? → Files.exists(fic);  
É uma directoria? → Files.isDirectory(fic);  
É um ficheiro? → Files.isRegularFile(fic);  
Está escondido? → Files.isHidden(fic);  
Permissão de read? → Files.isReadable(fic);  
Permissão de write? → Files.isWritable(fic);  
Dimensão → Files.size(fic);  
Data da última alteração → Files.getLastModifiedTime(fic)
```



Enquanto na interface anterior todas as funcionalidades estavam concentradas na classe `File`, agora temos as funcionalidades distribuídas por várias classes e interfaces. As novas classes pertencem ao novo package **java.nio**:

- **Path** – é uma interface que permite criar um caminho e possui métodos para trabalhar com esse caminho. É uma interface implementada pela classe **Paths** da qual vamos usar um método **factory** (um método que devolve um objeto criado);
- **Files** – permite trabalhar com ficheiros e directorias. Por exemplo valida o tipo, a dimensão, permite criar e mudar atributos no filesystem;

**Exercício sobre a classe File**

## Exercício sobre classe File

- Escrever um programa que permita obter o máximo de propriedades sobre a diretoria de trabalho do utilizador e sobre o ficheiro criado nos exemplos anteriores (NotasAlunos1.txt):
  - Usando a classe File – versão Java anterior a 1.6;
  - Usando a nova API da versão Java 1.7;



O slide descreve o objetivo do exercício.

**Solução com File (versão anterior ao Java 1.7)**

O programa abaixo exemplifica algumas das funcionalidades da classe File:



```
1 package capitulo6;
2
3 import java.io.File;
4 import java.io.IOException;
5 import java.text.SimpleDateFormat;
6
7 public class Ex13FileOld {
8
9     public static void main(String[] args) {
10         //diretoria do projecto
11         File fic = new File(System.getProperty("user.dir"));
12         mostraPropriedades(fic);
13         //ficheiro actual
14         fic = new File(System.getProperty("user.dir") + File.separatorChar
15             + "classes" + File.separator + "capitulo6" + File.separatorChar
16             + "File01.class");
17         mostraPropriedades(fic);
18     }
19
20     private static void mostraPropriedades(File fic) {
21         System.out.println("-----");
22         try {
23             System.out.println("Nome: " + fic.getName());
24             System.out.println("Caminho: " + fic.getParent());
25             System.out.println("Path: " + fic.getPath());
26             System.out.println("Canonical name: " + fic.getCanonicalPath());
```

```

27     System.out.println("Absolute path: " + fic.getAbsolutePath());
28     System.out.println("");
29     System.out.println("Existe? " + fic.exists());
30     System.out.println("É uma directoria? " + fic.isDirectory());
31     System.out.println("É um ficheiro? " + fic.isFile());
32     System.out.println("Está escondido? " + fic.isHidden());
33     System.out.println("Permissão de read? " + fic.canRead());
34     System.out.println("Permissão de write? " + fic.canWrite());
35     System.out.println("");
36     System.out.println("Dimensão: " + fic.length());
37     SimpleDateFormat sdf= new SimpleDateFormat("yyyy MMM dd hh:mm:ss zzz");
38     System.out.println("Data da última alteração: "
39         + sdf.format(fic.lastModified()));
40 } catch (IOException e) {
41     e.printStackTrace(System.out);
42 }
43 System.out.println("-----");
44 }
45 }

```



O construtor da classe `File` não obriga a tratar exceção, mas alguns dos seus métodos obrigam a tratar uma **IOException**.

### Solução com a nova API da versão Java 1.7

O programa abaixo mostra como as mesmas funcionalidades são obtidas na nova API:



```

1 package capitulo6;
2
3 import java.io.IOException;
4 import java.nio.file.Files;
5 import java.nio.file.Path;
6 import java.nio.file.Paths;
7
8 public class Ex14FilesNew {
9
10     public static void main(String[] args) {
11         //directoria do projecto
12         Path fic = Paths.get(System.getProperty("user.dir"));
13         mostraPropriedades(fic);
14
15         //ficheiro
16         fic = Paths.get(System.getProperty("user.dir") + "/NotasAlunos1.txt");
17         mostraPropriedades(fic);
18     }
19
20     private static void mostraPropriedades(Path fic) {
21         System.out.println("-----");
22         try {
23             System.out.println("Nome: " + fic.getFileName());
24             System.out.println("Caminho: " + fic.getParent());
25             System.out.println("Raiz: " + fic.getRoot());
26             System.out.println("Número de subdirectorias: " + fic.getNameCount());
27             System.out.println("Directoria na posição 3: " + fic.getName(3));
28             System.out.println("Canonical path: " + fic.toRealPath());
29             System.out.println("Absolute path: " + fic.toAbsolutePath());
30             System.out.println("");
31             System.out.println("Existe? " + Files.exists(fic));
32             System.out.println("Não existe? " + Files.notExists(fic));
33             System.out.println("É uma directoria? " + Files.isDirectory(fic));
34             System.out.println("É um ficheiro? " + Files.isRegularFile(fic));
35             System.out.println("Dimensão: " + Files.size(fic));
36             System.out.println("Está escondido? " + Files.isHidden(fic));
37             System.out.println("Permissão de read? " + Files.isReadable(fic));
38             System.out.println("Permissão de write? " + Files.isWritable(fic));

```

```
39     System.out.println("Permissão de execute? " + Files.isExecutable(fic));
40     System.out.println("Data da última alteração: "
41         + Files.getLastModifiedTime(fic));
42 } catch (IOException e) {
43     e.printStackTrace(System.out);
44 }
45     System.out.println("-----");
46 }
47 }

45 }
```



O construtor da classe **Path** também não obriga a tratar exceção, mas alguns dos seus métodos obrigam a tratar uma **IOException**.



Além das correções de erros a nova API é mais robusta e rápida, disponibilizando algumas novas funcionalidades. Vale a pena consultar a documentação da classe para ver o que há de novo ou o Java Tutorial.

## Sumário

---

- Como o Java trabalha com ficheiros;
- Classes Java usadas para trabalhar com ficheiros;
- Escrever dados no standard output;
- Ler dados a partir do standard input;
- Escrever dados em ficheiro de texto;
- Ler dados a partir de ficheiro de texto;
- Escrever dados em formato proprietário;
- Ler dados a partir de formato proprietário;
- Classe File;

