


Laboratório de Engenharia de Software

# INF1636 – PROGRAMAÇÃO ORIENTADA A OBJETOS

Departamento de Informática – PUC-Rio

Ivan Mathias Filho  
[ivan@inf.puc-rio.br](mailto:ivan@inf.puc-rio.br)

## Programa – Capítulo 16




Laboratório de Engenharia de Software

- Fluxos de Dados
- Fluxos de Caracteres
- Entrada e Saída Formatadas

© LES/PUC-Rio

2

**Programa – Capítulo 16**




Laboratório de Engenharia de Software

- **Fluxos de Dados**
- Fluxos de Caracteres
- Entrada e Saída Formatadas

© LES/PUC-Rio

3

**Fluxos de Dados – Introdução (1)**



Laboratório de Engenharia de Software

- Um fluxo (stream) de E/S representa uma fonte ou um destino de dados;
- Um fluxo pode representar tipos diferentes de fontes, tais como arquivos em disco, dispositivos, outros programas, áreas de memória ou conexões baseadas em sockets;
- Fluxos dão suporte a diferentes tipos de dados, incluindo simples bytes, tipos de dados primitivos, caracteres e objetos;
- Independentemente de como funcionam internamente, todos os fluxos apresentam um modelo único ao programador;
- Isto é, um fluxo é uma sequência de dados.

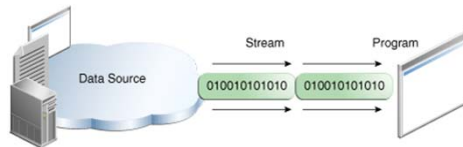
© LES/PUC-Rio

4

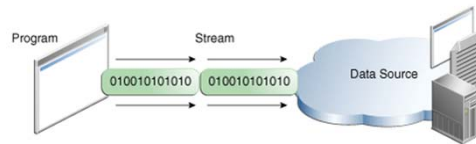
## Fluxos de Dados – Introdução (2)



Um programa usa um stream de entrada para obter dados de uma fonte, um item de cada vez.



Um programa usa um stream de saída para gravar dados em uma fonte, um item de cada vez.



© LES/PUC-Rio

5

## Fluxos de Dados – Algoritmos



- Independentemente da origem e dos tipos de dados, os algoritmos para leitura e escrita de dados são basicamente os mesmos:

### Leitura

abra o stream  
enquanto houver informação faça:  
    leia a informação  
feche o stream

### Escrita

abra o stream  
enquanto houver informação faça:  
    grave a informação  
feche o stream

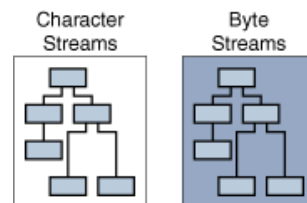
© LES/PUC-Rio

6

## Fluxos de Dados – Pacotes e Hierarquias



- O pacote **java.io** contém uma coleção de classes de fluxos que dão suporte aos algoritmos de leitura e escrita;
- Portanto, para utilizar fluxos de dados, um programa deve importar o pacote **java.io**;
- As classes de fluxos são organizadas em duas hierarquias de classes, baseadas nos tipos de fluxos (caracteres ou bytes).



© LES/PUC-Rio

7

## Programa – Capítulo 16




- Fluxos de Dados
- **Fluxos de Caracteres**
- Entrada e Saída Formatadas

© LES/PUC-Rio

8

Laboratório de Engenharia de Software

## Fluxos de Caracteres – Visão Geral




- A plataforma Java representa caracteres usando as convenções Unicode;
- Os fluxos (streams) de I/O orientados para caracteres traduzem, automaticamente, caracteres Unicode para o conjunto de caracteres local;
- Em países ocidentais, o formato local é, usualmente, um superconjunto de ASCII (8 bits);
- Entrada e saída feitas com classes de streams traduzem automaticamente de/para o conjunto de caracteres local;
- Um programa que usa fluxos de caracteres no lugar de fluxos de bytes adapta-se, automaticamente, ao conjunto de caracteres local, e está pronto para a internacionalização.

© LES/PUC-Rio

9

Laboratório de Engenharia de Software

## Fluxos de Caracteres – Classes



- Todas as classes de fluxo de caracteres descendem de **Reader** e **Writer**.
- Tal como acontece com fluxos de bytes, há classes de fluxo de caracteres que tratam especificamente de arquivos.
- São elas:
  - **FileReader**
  - **FileWriter**

© LES/PUC-Rio

10

## Fluxos de Caracteres – Exemplo



```
import java.io.*;

public class CopyCharacters {
    public static void main(String[] args) throws IOException {
        FileReader inputStream = null;
        FileWriter outputStream = null;

        try {
            inputStream = new FileReader("entrada.txt");
            outputStream = new FileWriter("saida.txt");
            int c;
            while ((c = inputStream.read()) != -1) {
                outputStream.write(c);
            }
        } finally {
            if (inputStream != null) {
                inputStream.close();
            }
            if (outputStream != null) {
                outputStream.close();
            }
        }
    }
}
```

© LES/PUC-Rio

11

## Classes participantes



- A classe **FileReader** estende **InputStreamReader**;
- Lê texto de arquivos de caracteres usando um tamanho de buffer padrão;
- A decodificação de bytes para caracteres usa um conjunto de caracteres especificado (**Locale**) ou o conjunto de caracteres padrão da plataforma.
- A classe **FileWriter** estende **OutputStreamWriter**;
- Grava texto em arquivos de caracteres usando um tamanho de buffer padrão;
- A codificação de caracteres para bytes usa um conjunto de caracteres especificado (**Locale**) ou o conjunto de caracteres padrão.

© LES/PUC-Rio

12

## Entrada e Saída de Linhas



- Entrada e Saída de caracteres ocorrem em unidades maiores do que um único caractere;
- Uma unidade comum é uma linha de caracteres: uma cadeia de caracteres encerrada por um terminador de linha;
- Um terminador de linha pode ser:
  - Uma sequência de carriage-return/line-feed ("\\r\\n")
  - Um simples carriage-return ("\\r")
  - Um simples line-feed ("\\n")
- Dar suporte a todos os possíveis terminadores de linhas permite que um programa leia textos criados em quaisquer sistemas operacionais de uso difundido.

## Entrada e Saída de Linhas – Exemplo




```
import java.io.*;

public class CopyLines {
    public static void main(String[] args) throws IOException {
        BufferedReader inputStream = null;
        PrintWriter outputStream = null;
        try {
            inputStream = new BufferedReader(new FileReader("entrada.txt"));
            outputStream = new PrintWriter(new FileWriter("saida.txt"));

            String l;
            while ((l = inputStream.readLine()) != null) {
                System.out.printf("%s\\n", l);
                outputStream.println(l);
            }
        } finally {
            if (inputStream != null) {
                inputStream.close();
            }
            if (outputStream != null) {
                outputStream.close();
            }
        }
    }
}
```

Laboratório de Engenharia de Software

## Classes participantes – `BufferedReader`




- A classe **`BufferedReader`** estende **`Reader`**;
- Lê texto de um fluxo de entrada de caracteres, armazenando caracteres em um buffer para fornecer uma leitura eficiente de caracteres, matrizes e linhas;
- Pode-se especificar um tamanho para o buffer ou pode-se usar o tamanho padrão;
- O tamanho do buffer é definido por meio do construtor **`BufferedReader(Reader in, int sz)`**;
- O tamanho padrão é grande o suficiente para a maioria dos propósitos.

© LES/PUC-Rio15

Laboratório de Engenharia de Software

## Classes participantes – `BufferedWriter`



- A classe **`BufferedWriter`** estende **`Writer`**;
- Grava texto em um fluxo de saída de caracteres, armazenando caracteres em um buffer para fornecer a gravação eficiente de um único caractere, de arrays e de strings;
- Pode-se especificar um tamanho para o buffer ou pode-se usar o tamanho padrão;
- O tamanho do buffer é definido por meio do construtor **`BufferedWriter(Writer out, int sz)`**;
- O tamanho padrão é grande o suficiente para a maioria dos propósitos.

© LES/PUC-Rio16



Programa – Capítulo 16




Laboratório de Engenharia de Software

- Fluxos de Dados
- Fluxos de Caracteres
- **Entrada e Saída Formatadas**

© LES/PUC-Rio

17

Entrada e Saída Formatadas



Laboratório de Engenharia de Software

- Entrada e saída geralmente envolvem formatar dados para facilitar a comunicação com seres humanos;
- A plataforma Java fornece duas APIs para ajudar o programador na formatação de dados;
- A API **scanner** subdivide os dados de entrada em tokens individuais;
- A API **formatting** formata dados com o intuito de apresentá-los de forma mais adequada ao uso por seres humanos.

© LES/PUC-Rio

18

## Scanning (1)



- Objetos da classe **Scanner** são úteis para ler dados formatados e transformá-los em tokens relativos a um tipo de dado;
- Por default, um scanner usa espaços em branco para separar os tokens;
- Espaços em branco incluem caracteres de espaço, tabulação e terminadores de linhas.

## Scanning – Exemplo (1)



```
import java.io.*;
import java.util.*;

public class Scan {
    public static void main(String[] args) throws IOException {

        Scanner s = null;

        try {
            s = new Scanner(new BufferedReader(new FileReader("entrada.txt")));

            while (s.hasNext())
                System.out.println(s.next());

        } finally {
            if (s != null)
                s.close();
        }
    }
}
```

### Arquivo de Entrada:

INF1636  
PROGRAMAÇÃO ORIENTADA A OBJETOS  
Departamento de Informática  
PUC-Rio

## Scanning – Exemplo (2)



- Note que o exemplo anterior invoca o método **close()** após todos os dados terem sido lidos;
- Mesmo que o scanner não seja um stream de dados, deve-se fechá-lo para indicar que ele não é mais necessário;
- A saída do exemplo anterior é a seguinte:

```
INF1636
PROGRAMAÇÃO
ORIENTADA
A
OBJETOS
Departamento
de
Informática
PUC-Rio
```

© LES/PUC-Rio

21

## Traduzindo Tokens Individualmente (1)



- O exemplo anterior trata todos os tokens de entrada como cadeias de caracteres (Strings);
- A classe **Scanner**, entretanto, suporta tokens de todos os tipos primitivos de Java, exceto caractere (**char**);
- Valores numéricos podem ser informados usando-se separadores de milhares;
- Para tal, deve-se definir a opção de idioma (**Locale**) adequada;
- No exemplo a seguir, o seguinte arquivo será usado:

```
8,5
32.767
3,14159
1.000.000,1
```

© LES/PUC-Rio

22

## Traduzindo Tokens Individualmente (2)



```
import java.io.*;
import java.util.*;

public class ScanSum {
    public static void main(String[] args) throws IOException {
        Scanner s = null;
        double sum = 0;
        try {
            Locale l;
            l = new
Locale.Builder().setLanguage("pt").setScript("Latn").setRegion("BR").build();

            s = new Scanner(new BufferedReader(new FileReader("numeros.txt")));
            s.useLocale(l);

            while (s.hasNext())
                if (s.hasNextDouble())
                    sum += s.nextDouble();
                else
                    s.next();
        } finally {s.close();}

        System.out.println(sum);
    }
}
```

O console exibirá o seguinte:

1032778.74159

© LES/PUC-Rio

23

## Formatação (1)




- Streams de saída que usam formatação são instâncias de **PrintWriter** (character stream) ou **PrintStream** (byte stream).
- Ambas as classes implementam o mesmo conjunto de métodos para formatação de dados de saída:
  - print** e **println** para formatar valores individuais;
  - format** para formatar valores baseado em uma string de formatação, de modo semelhante a **printf** da linguagem C;
- O exemplo a seguir mostra como usar o método **format**.

© LES/PUC-Rio

24

Laboratório de Engenharia de Software

## Formatação (2)



```
import java.io.*;
import java.util.*;

public class Format {
    public static void main(String[] args) throws IOException {
        PrintWriter out = null;
        Scanner s = null;
        double sum = 0;
        Locale l;
        try {
            out = new PrintWriter(new FileWriter("saida.txt"));
            s = new Scanner(new BufferedReader(new FileReader("numeros.txt")));
            l = new
Locale.Builder().setLanguage("pt").setScript("Latn").setRegion("BR").build();
            s.useLocale(l);
            while (s.hasNext())
                if (s.hasNextDouble())
                    sum += s.nextDouble();
                else
                    s.next();
            } finally { s.close(); }

            out.format(l, "Valor: %,.2f", sum);
            out.close();
        }
    }
}
```

Conteúdo do arquivo saida.txt:  
Valor: 1.032.778,74

© LES/PUC-Rio25