Universidade Presbiteriana Mackenzie
Faculdade de Computação e Informática

Entrada e Saída de dados em Java

NOTAS DE AULA

Linguagem de Programação II
2º semestre de 2015
Prof. Tomaz Mikio Sasaki
Versão 1.1

Objetivos

- Cobrir as APIs Java para entrada e saída.
- Aprender a acessar arquivos e diretórios.
- Aprender a ler e escrever dados nos formatos texto e binário.
- Apresentar o mecanismo de serialização que permite persistir e recuperar objetos.

Referência

A referência para esta aula é o capítulo 1 de:

> HORSTMANN, C.S.; CORNELL, G. **Core Java, Volume II - Advanced Features** Prentice Hall, 2013.

Observação: As notas de aula são material de apoio para estudo e não têm o objetivo de apresentar o assunto de maneira exaustiva. Não deixe de ler o material de referência da disciplina.

Streams

- *Input stream*: objeto que pode ler uma sequência de bytes.
- *Output stream*: objeto que pode escrever uma sequência de bytes.
- A origem e o destino destas sequências de bytes podem ser **arquivos**, **conexões de rede** ou **blocos de memória**.
- As classes abstratas *InputStream* e *OutputStream* são a base para a hierarquia de classes de entrada e saída.

Leitura de dados

A classe *InputStream* possui os métodos *read* para leitura de bytes:

```
abstract int read()
int read(byte[] b)
int read(byte[] b, int off, int len)
```

Escrita de dados

A classe *OutputStream* define os métodos *write* para escrita de bytes:

```
abstract void write(int b)
void write(byte[] b)
void write(byte[] b, int off, int len)
```

Biblioteca de classes para manipulação de streams

A biblioteca Java possui muitas classes derivadas de *InputStream* e *OutputStream* para acessar e persistir os dados de diversos formatos.

Hierarquia da classe InputStream

```
InputStream (implements Closeable)
|------ ByteArrayInputStream
|------ FileInputStream
|------ FilterInputStream
|         |------ BufferedInputStream
|         |------ DataInputStream (implements DataInput)
|         |------ LineNumberInputStream
|         `------ PushbackInputStream
|
|------ ObjectInputStream (implements ObjectInput, ObjectStreamConstants)
|------ PipedInputStream
|------ SequenceInputStream
`------ StringBufferInputStream
```

Fonte: (HORSTMANN, 2013)

Hierarquia da classe OutputStream

```
OutputStream (implements Closeable, Flushable)
|------ ByteArrayOutputStream
|------ FileOutputStream
|------ FilterOutputStream
|         |------ BufferedOutputStream
|         |------ DataOutputStream (implements DataOutput)
|         `------ PrintStream (implements java.lang.Appendable, Closeable)
|
|------ ObjectOutputStream (implements ObjectOutput, ObjectStreamConstants)
`------ PipedOutputStream
```

Fonte: (HORSTMANN, 2013)

FileInputStream, FileOutputStream

Streams de entrada e saída associados a um arquivo em disco.

Exemplo utilizando FileInputStream

Programa que lê dois bytes do arquivo

```
public class AppFileInputStream {
    public static void main(String[] args)
                    throws FileNotFoundException, IOException {
        FileInputStream fis;
        fis = new FileInputStream("entrada.txt");
        int leitura = fis.read();
        System.out.println("Primeiro valor lido: " + leitura);
        leitura = fis.read();
        System.out.println("Segundo valor lido: " + leitura);
        fis.close();
    }
}
```

Supondo que o arquivo *entrada.txt* tenha o conteúdo

```
AEIOU
```

O resultado da execução do programa será

```
Primeiro valor lido: 65
Segundo valor lido: 69
```

Exemplo utilizando FileOutputStream

Programa que escreve quatro bytes em um arquivo

```
public class AppFileOutputStream {
    public static void main(String[] args)
                        throws FileNotFoundException, IOException {
        FileOutputStream fos;
        fos = new FileOutputStream("saida.txt");
        fos.write(76);
        fos.write(80);
        fos.write(73);
        fos.write(73);
        fos.close();
    }
}
```

Conteúdo do arquivo saida.txt após a execução

```
LPII
```

Verificação do número de bytes disponíveis no buffer

```
public class AppBytesDisponiveis {
    public static void main(String[] args)
                        throws FileNotFoundException, IOException {
        FileInputStream fis = new FileInputStream("entrada.txt");
        int leitura = fis.read();
        System.out.println("Primeiro valor lido: " + leitura);
        leitura = fis.read();
        System.out.println("Segundo valor lido: " + leitura);
        int b = fis.available();
        System.out.println("Bytes disponíveis: " + b);
        fis.close();
    }
}
```

Supondo que o arquivo *entrada.txt* tenha o conteúdo

```
AEIOU
```

O resultado da execução do programa será

```
Primeiro valor lido: 65
Segundo valor lido: 69
Bytes disponíveis: 3
```

As interfaces

Closeable

Implementada por InputStream, OutputStream, Reader, Writer

Flushable

Implementada por OutputStream, Writer

Readable

Implementada por Reader, CharBuffer

Appendable

Implementada por Writer, CharBuffer, StringBuilder

Fonte: (HORSTMANN, 2018)

Close

Ao término da leitura ou escrita em um *stream*, deve-se chamar o método **close**.

Leitura de dados utilizando Reader

É possível ler caracteres Unicode utilizando os métodos:

```
int read()
int read(char[] cbuf)
abstract int read(char[] cbuf, int off, int len)
int read(CharBuffer target)
```

Exemplo de leitura de arquivo

```
public class AppFileReader {
    public static void main(String[] args)
                        throws FileNotFoundException, IOException {
        FileReader fr;
        fr = new FileReader("entrada.txt");
        char[] cbuf = new char[100];
        fr.read(cbuf);
        System.out.println(cbuf);
        fr.close();
    }
}
```

Escrita de dados utilizando Writer

É possível escrever caracteres Unicode utilizando os métodos:

```
void write(char[] cbuf)
abstract void write(char[] cbuf, int off, int len)
void write(int c)
```

Exemplo de utilização de *FileWriter*

```
public class AppFileWriter {
    public static void main(String[] args)
                    throws FileNotFoundException, IOException {
        FileWriter fw;
        fw = new FileWriter("saida.txt");
        fw.write('L');
        fw.write('P');
        fw.write('I');
        fw.write('I');
        fw.close();
    }
}
```

Hierarquia da classe Reader

```
Reader (implements Closeable, java.lang.Readable)
|------ BufferedReader
|       |
|       `------ LineNumberReader
|
|------ CharArrayReader
|------ FilterReader
|       |
|       `------ PushbackReader
|
|------ InputStreamReader
|       |
|       `------ FileReader
|
|------ PipedReader
`------ StringReader
```

Fonte: (HORSTMANN, 2013)

Hierarquia da classe Writer

```
Writer (implements java.lang.Appendable, Closeable, Flushable)
|------ BufferedWriter
|------ CharArrayWriter
|------ FilterWriter
|------ OutputStreamWriter
|       |
|       `------ FileWriter
|
|------ PipedWriter
|------ PrintWriter
`------ StringWriter
```

Fonte: (HORSTMANN, 2013)

Criação de um arquivo texto

```
public class CriaArquivo {
    public static void main(String[] args) {
        PrintWriter pw;
        try {
            pw = new PrintWriter("meu_arquivo.txt");
            pw.println("Teste");
            pw.println(new Date());
            pw.flush();
            pw.close();
        } catch (FileNotFoundException ex) {
            ex.printStackTrace();
        }
    }
}
```

Exemplo de leitura de arquivo

```
public class AppInputStreamReader {
    public static void main(String[] args)
                        throws FileNotFoundException, IOException {
        FileInputStream fis = new FileInputStream("entrada.txt");
        InputStreamReader isr = new InputStreamReader(fis);
        char[] cbuf = new char[100];
        isr.read(cbuf);
        System.out.println(cbuf);
    }
}
```

Exemplo de leitura de dados no formato texto

```
BufferedReader in;
in = new BufferedReader(
            new InputStreamReader(
                        new FileInputStream("empregado.txt"), "UTF-8));
String nome = in.readLine();
```

Obtenção de informações e leitura de um arquivo

```
public class LerArquivo {
    public static void main(String[] args) {
        File arquivo = new File("meu_arquivo.txt");
        if (arquivo.exists()) {
            System.out.println("Arquivo existe");
        }
        else {
            System.out.println("Arquivo não existe");
        }

        if (arquivo.isDirectory()) {
            System.out.println("É um diretório");
        }
        else {
            System.out.println("Não é um diretório");
        }

        BufferedReader reader;
        try {
            reader = new BufferedReader(new FileReader(arquivo));
            String primeiraLinha = reader.readLine();
            System.out.println("Primeira linha: " + primeiraLinha);
        } catch (Exception ex) {
            ex.printStackTrace();
        }
    }
}
```

Exemplo de programa servidor socket que fornece a data do computador

```
public class ServidorData {
    public static void main(String[] args) throws IOException {
        ServerSocket servidor = new ServerSocket(8888);
        System.out.println("Esperando conexão...");
        Socket conexao = servidor.accept();
        System.out.println("Conexão estabelecida!");
        OutputStream os = conexao.getOutputStream();
        PrintWriter writer = new PrintWriter(os);
        writer.println("Bem-vindo ao servidor de data e hora!");
        writer.println("A data e hora do servidor é " + new Date());
        writer.close();
        conexao.close();
        System.out.println("Fim do programa!");
    }
}
```

Para testar este programa, execute-o em um computador e tente se conectar à porta TCP 8888 deste computador utilizando um programa de TELNET.

Lembre-se:

- **InputStream** e **OutputStream** são orientados à leitura e escrita de **bytes**.
- **Reader** e **Writer** são orientados à leitura e escrita de **caracteres**.
- **BufferedReader** e **PrintWriter** conseguem ler e escrever **linhas inteiras** compostas por **strings**.

Bom estudo!