Tratamento de Arquivos em Java

Prof. Rafael Guterres Jeffman
rafael.jeffman@gmail.com

Basic I/O

- Streams
  - Uma stream é uma seqüência de bytes que podem ser lidos de uma fonte ou gravados em um destino.
- Arquivos
  - Arquivos são repositórios de dados gravados de forma organizada em um sistema de arquivos.

I/O Streams

- Byte Streams
- Character Streams
- Buffered Streams
- Data Streams
- Object Streams

Byte Streams

- Utilizadas por programas para executar I/O de palavras de 8-bits (byte).

- As classes que implementam byte streams derivam de **InputStream** e **OutputStream**.

- Entre as classes existentes incluem-se **AudioInputStream**, **ByteArrayInputStream**, **FileInputStream**, **FilterInputStream**, **InputStream**, **ObjectInputStream**, **PipedInputStream**, **SequenceInputStream**, **StringBufferInputStream**

Exemplo de Byte Stream

```
import java.io.*

class CopyFiles {
        public static void main(String[] args) {
                FileInputStream in = null;
                FileOutputStream out = null;
                try {
                        in = new FileInputStream(“input.dat”);
                        out = new FileOutputStream(“output.dat”);
                        int c = 0;
                        while ((c = in.read()) != -1)
                                out.write(c);
                } finally {
                        if (in != null) in.close();
                        if (out != null) out.close();
                }
        }
}
```

Character Streams

- São muito semelhantes às Byte Streams.
- Quando os dados são lidos de uma stream utilizando streams de caracteres, os dados lidos são convertidos de acordo com a localização especificada.
- As classes que implementam streams de caracteres devem estender **Reader** e **Writer**.

Buffered Streams

- As streams vistas até agora, não utilizam “buffers” de entrada ou saída, o que muitas vezes diminuem a performance do sistema.

- As streams com buffers apenas “enviam” seus dados quando o buffer fica cheio, ou em pontos específicos mediante chamada do métodos **flush**.

- Algumas streams implementam um esquema de **autoflush** quando alguns métodos são executados, como a classe **PrintWriter** que implementa *autoflush* no método **println.**

Buffered Streams

- Para criar uma stream que utiliza um buffer, simplesmente criamos um classe que implementa um buffer “em cima” de uma stream sem buffer.

- Para streams de bytes, utilizamos **BufferedInputStream** e **BufferedOutputStream**.

- Para streams de caracteres, utilizamos **BufferedReader** e **BufferedWriter.**

- BufferedInputStream bis =
  new BufferedInputStream(
  new FileReader(“input.dat”));

Data Streams

- Suportam I/O dos tipos de dados fundamentais boolean, char, byte, short, int, long, float e double, assim como de Strings.

- Data Streams implementam as interfaces DataInput e/ou DataOutput.

Data Stream Output Example

```
import java.io.*;

public class DataStreamExample {

     public static void main(String[] args) {
          DataOutputStream out = null;
          try {
               out = new DataOutputStream(
                         new BufferedOutputStream(
                                   new FileOutputStream("registros.dat")
                         )
                    );
               out.writeInt(123);
               out.writeDouble(3.1415);
               out.writeUTF("Uma String");
          } catch (FileNotFoundException fnfe) {
               fnfe.printStackTrace();
          } catch (IOException ioe) {
               ioe.printStackTrace();
          } finally {
               if (out != null)
                    try { out.close(); }
                    catch (IOException e) { e.printStackTrace(); }
          }
     }

}
```

Data Stream Input Example

```
import java.io.*;

public class DataStreamExample {

     public static void main(String[] args) {
          DataInputStream in = null;
          try {
               in = new DataInputStream(
                         new BufferedInputStream(
                                   new FileInputStream("registros.dat")
                                   )
                         );
               int index = in.readInt();
               double pi = in.readDouble();
               String s = in.readUTF();
          } catch (FileNotFoundException fnfe) {
               fnfe.printStackTrace();
          } catch (IOException ioe) {
               ioe.printStackTrace();
          } catch (EOFException eof) {
               System.out.println(eof.getMessage());
          } finally {
               if (in != null)
                    try { in.close(); }
                    catch (IOException e) { e.printStackTrace(); }
          }
     }

}
```

Object Streams

- Streams que implementem as interfaces **ObjectInput** e **ObjectOutput** podem ser utilizadas para serializar e deserializar objetos.

- Objetos que implementam a interface **Serializable** podem ser serializados.

- Se um atributo não deve ser serializado e enviado para a stream, deve-se declará-lo como **transient**.

- Os métodos utilizados para lidar com as streams são **readObject**() e **writeObject**().

Leitura e Escrita com Formatação de Dados

- Um forma eficiente de ler dados formatados é utilizar objetos da classe **Scanner** associados a streams de dados.

- Uma forma eficiente de formatar os dados para escrita é utilizar o método **String.format** que aceita modificadores para formatar dados com tipos fundamentais, semelhante à forma como é feita na linguagem C.

I/O pela Linha de Comando

- Streams Padrão
    - Existem três streams padrão em Java, declaradas na classe **System**.
        - System.in: Entrada de dados (teclado)
        - System.out: Saída de dados (console)
        - System.err: Saída de erros (console)
- O Console
    - Facilita a entrada de dados do tipo String e o trabalho com Senhas.
    - Permite obter as classes Reader e Writer associadas ao Console.
    - Existe um bug no Eclipse que impede a utilização desta classe.