Módulo 2

Introdução à Programação II

Lição 12

Stream de Entrada e Saída de Dados (I/O) Avançados

Versão 1.0 - Mar/2007

Autor
Rebecca Ong

Equipe
Joyce Avestro
Florence Balagtas
Rommel Feria
Rebecca Ong
John Paul Petines
Sun Microsystems
Sun Philippines

Necessidades para os Exercícios

Sistemas Operacionais Suportados

NetBeans IDE 5.5 para os seguintes sistemas operacionais:

- Microsoft Windows XP Profissional SP2 ou superior
- Mac OS X 10.4.5 ou superior
- Red Hat Fedora Core 3
- Solaris™ 10 Operating System (SPARC® e x86/x64 Platform Edition)

NetBeans Enterprise Pack, poderá ser executado nas seguintes plataformas:

- Microsoft Windows 2000 Profissional SP4
- Solaris™ 8 OS (SPARC e x86/x64 Platform Edition) e Solaris 9 OS (SPARC e x86/x64 Platform Edition)
- Várias outras distribuições Linux

Configuração Mínima de Hardware

Nota: IDE NetBeans com resolução de tela em 1024x768 pixel

Sistema Operacional	Processador	Memória	HD Livre
Microsoft Windows	500 MHz Intel Pentium III workstation ou equivalente	512 MB	850 MB
Linux	500 MHz Intel Pentium III workstation ou equivalente	512 MB	450 MB
Solaris OS (SPARC)	UltraSPARC II 450 MHz	512 MB	450 MB
Solaris OS (x86/x64 Platform Edition)	AMD Opteron 100 Série 1.8 GHz	512 MB	450 MB
Mac OS X	PowerPC G4	512 MB	450 MB

Configuração Recomendada de Hardware

Sistema Operacional	Processador	Memória	HD Livre
Microsoft Windows	1.4 GHz Intel Pentium III workstation ou equivalente	1 GB	1 GB
Linux	1.4 GHz Intel Pentium III workstation ou equivalente	1 GB	850 MB
Solaris OS (SPARC)	UltraSPARC IIIi 1 GHz	1 GB	850 MB
Solaris OS (x86/x64 Platform Edition)	AMD Opteron 100 Series 1.8 GHz	1 GB	850 MB
Mac OS X	PowerPC G5	1 GB	850 MB

Requerimentos de Software

NetBeans Enterprise Pack 5.5 executando sobre Java 2 Platform Standard Edition Development Kit 5.0 ou superior (JDK 5.0, versão 1.5.0_01 ou superior), contemplando a Java Runtime Environment, ferramentas de desenvolvimento para compilar, depurar, e executar aplicações escritas em linguagem Java. Sun Java System Application Server Platform Edition 9.

- Para **Solaris**, **Windows**, e **Linux**, os arquivos da JDK podem ser obtidos para sua plataforma em http://java.sun.com/j2se/1.5.0/download.html
- Para **Mac OS X**, Java 2 Plataform Standard Edition (J2SE) 5.0 Release 4, pode ser obtida diretamente da Apple's Developer Connection, no endereço: http://developer.apple.com/java (é necessário registrar o download da JDK).

Para mais informações: http://www.netbeans.org/community/releases/55/relnotes.html

Colaboradores que auxiliaram no processo de tradução e revisão

Alexandre Mori
Alexis da Rocha Silva
Aline Sabbatini da Silva Alves
Allan Wojcik da Silva
André Luiz Moreira
Andro Márcio Correa Louredo
Antoniele de Assis Lima
Antonio Jose R. Alves Ramos
Aurélio Soares Neto
Bruno da Silva Bonfim
Bruno dos Santos Miranda
Bruno Ferreira Rodrigues
Carlos Alberto Vitorino de Almeida
Carlos Alexandre de Sene
Carlos André Noronha de Sousa
Carlos Eduardo Veras Neves
Cleber Ferreira de Sousa
Cleyton Artur Soares Urani
Cristiano Borges Ferreira
Cristiano de Siqueira Pires
Derlon Vandri Aliendres
Fabiano Eduardo de Oliveira
Fábio Bombonato
Fernando Antonio Mota Trinta
Flávio Alves Gomes
Francisco das Chagas
Francisco Marcio da Silva
Gilson Moreno Costa
Givailson de Souza Neves
Gustavo Henrique Castellano
Hebert Julio Gonçalves de Paula
Heraldo Conceição Domingues
Hugo Leonardo Malheiros Ferreira
Ivan Nascimento Fonseca
Jacqueline Susann Barbosa
Jader de Carvalho Belarmino
João Aurélio Telles da Rocha
João Paulo Cirino Silva de Novais
João Vianney Barrozo Costa
José Augusto Martins Nieviadonski
José Leonardo Borges de Melo
José Ricardo Carneiro
Kleberth Bezerra G. dos Santos
Lafaiete de Sá Guimarães
Leandro Silva de Morais
Leonardo Leopoldo do Nascimento
Leonardo Pereira dos Santos
Leonardo Rangel de Melo Filardi
Lucas Mauricio Castro e Martins
Luciana Rocha de Oliveira
Luís Carlos André
Luís Octávio Jorge V. Lima
Luiz Fernandes de Oliveira Junior
Luiz Victor de Andrade Lima
Manoel Cotts de Queiroz
Marcello Sandi Pinheiro
Marcelo Ortolan Pazzetto
Marco Aurélio Martins Bessa
Marcos Vinicius de Toledo
Maria Carolina Ferreira da Silva
Massimiliano Giroldi
Mauricio Azevedo Gamarra
Mauricio da Silva Marinho
Mauro Cardoso Mortoni
Mauro Regis de Sousa Lima
Namor de Sá e Silva
Néres Chaves Rebouças
Nolyanne Peixoto Brasil Vieira
Paulo Afonso Corrêa
Paulo José Lemos Costa
Paulo Oliveira Sampaio Reis
Pedro Antonio Pereira Miranda
Pedro Henrique Pereira de Andrade
Renato Alves Félix
Renato Barbosa da Silva
Reyderson Magela dos Reis
Ricardo Ferreira Rodrigues
Ricardo Ulrich Bomfim
Robson de Oliveira Cunha
Rodrigo Pereira Machado
Rodrigo Rosa Miranda Corrêa
Rodrigo Vaez
Ronie Dotzlaw
Rosely Moreira de Jesus
Seire Pareja
Sergio Pomerancblum
Silvio Sznifer
Suzana da Costa Oliveira
Tásio Vasconcelos da Silveira
Thiago Magela Rodrigues Dias
Tiago Gimenez Ribeiro
Vanderlei Carvalho Rodrigues Pinto
Vanessa dos Santos Almeida
Vastí Mendes da Silva Rocha
Wagner Eliezer Roncoletta

Auxiliadores especiais

Revisão Geral do texto para os seguintes Países:

- **Brasil** – Tiago Flach
- **Guiné Bissau** – Alfredo Cá, Bunene Sisse e Buon Olossato Quebi – ONG Asas de Socorro

Coordenação do DFJUG

- **Daniel deOliveira** – JUGLeader responsável pelos acordos de parcerias
- **Luci Campos -** Idealizadora do DFJUG responsável pelo apoio social
- **Fernando Anselmo -** Coordenador responsável pelo processo de tradução e revisão, disponibilização dos materiais e inserção de novos módulos
- **Regina Mariani -** Coordenadora responsável pela parte jurídica
- **Rodrigo Nunes -** Coordenador responsável pela parte multimídia
- **Sérgio Gomes Veloso -** Coordenador responsável pelo ambiente JEDI™ (Moodle)

Agradecimento Especial

John Paul Petines – Criador da Iniciativa JEDI™
Rommel Feria – Criador da Iniciativa JEDI™

1. Objetivos

Em lições anteriores, vimos como obter a entrada do usuário e manipular arquivos utilizando *Stream* (se refere a uma sequência de dados). Agora iremos aprender mais sobre *Stream* e outras classes relacionadas.

Ao final desta lição, o estudante será capaz de:

- Enumerar os tipos de *Stream*
  - *Stream* de caracteres e *bytes*
  - *Stream* de entrada e saída de dados
  - *Node Stream e Filter Stream*
- Usar a classe *File* e seus métodos
- Usar as diferentes classes de Entrada e Saída
  - *Reader*
  - *Writer*
  - *InputStream*
  - *OutputStream*
- Explicar o conceito de encadeamento de *Stream*
- Definir serialização
- Explicar o uso da palavra-chave *transient*
- Escrever e ler a partir de um *Stream*

2. Tipos *Stream* Gerais

2.1. Stream de caracteres e bytes

Geralmente existem dois tipos de *Stream*, de caracteres e de bytes. Vejamos a diferença básica entre estes dois tipos. *Stream de bytes* (ou *Byte Stream*) são abstrações de arquivos ou dispositivos para dados binários, enquanto *Stream* de caracteres (ou *Character Stream*) são para os caracteres Unicode.

A classe *InputStream* é uma classe abstrata raiz para todos os *Stream* de *bytes* utilizados como entrada, enquanto a classe *OutputStream* é a classe abstrata raiz para todos os *Stream* de *bytes* de saída. Para *Stream* de caracteres, a superclasse correspondente as classes de leitura e escrita, respectivamente as classes *Reader* e *Writer*.

2.2. Stream de entrada e saída de dados

Stream são também categorizados segundo o seu uso: se são utilizados para ler ou para escrever. É possível ler a partir de *Stream* de entrada, embora não seja possível escrever nos mesmos. Por outro lado, é possível escrever a um *Stream* de saída, embora não seja possível ler a partir deles.

A classe *InputStream* e a classe *Reader* são superclasses de todos os *Stream* de entrada de dados. A classe *OutputStream* e a classe *Writer* são as superclasses de todos os *Stream* de saída de dados.

Stream de entrada de dados (*input Stream*) são também conhecidos como ***Source Stream***, já que obtemos informações a partir destes *Stream*. Enquanto isso, *Stream* de saída de dados (*output Stream*) são também chamados ***Sink Stream***.

2.3. Node Stream e Filter Stream

O pacote *java.io* difere entre *Node Stream* e *Filter Stream*. Um *Node Stream* é um *Stream* com a funcionalidade básica de ler ou escrever a partir de um local específico como um disco ou a partir da rede. Tipos de *Node Stream* incluem arquivos, memória e pipes. *Filter Stream*, por outro lado, são postos sobre os *Node Stream* entre as tarefas ou os processos para prover funcionalidades adicionais não encontradas nos *Node Stream*. Agregar camadas a um *Node Stream* é chamado encadeamento de *Stream* ou ***Stream Chaining***.

3. Classe *File*

A classe *File* não é uma classe do tipo *Stream*, é uma representação abstrata de arquivos reais e de caminhos de diretórios.

Para construir um objeto da classe *File*, pode-se utilizar o seguinte construtor:

Construtor para a classe *File*
`File(String pathname)`
Instancia um objeto *File* com o *pathname* especificado como seu nome de arquivo. O nome do arquivo pode tanto ser absoluto (ex., contém o caminho completo) ou pode consistir do próprio nome do arquivo e se assume que o mesmo está contido no mesmo diretório.

Tabela 1: Construtor para a classe File

A classe *File* provê diversos métodos para manipulação de arquivos e diretórios. Na tabela abaixo estão alguns destes métodos:

Métodos da classe *File*
`public String getName()`
Retorna o nome do arquivo ou o nome do diretório deste objeto *File*.
`public boolean exists()`
Testa se um arquivo ou diretório existe.
`public long length()`
Retorna o tamanho do arquivo.
`public long lastModified()`
Retorna a data em milissegundos quando o arquivo foi modificado pela última vez.
`public boolean canRead()`
Retorna true se é permitido ler a partir do arquivo. De outro modo, retorna false.
`public boolean canWrite()`
Retorna true se é permitido escrever ao arquivo. De outro modo, retorna false.
`public boolean isFile()`
Testa se este objeto é um arquivo, ou seja, nossa normal percepção do que é um arquivo (não um diretório).
`public boolean isDirectory()`
Testa se este objeto é um diretório.
`public String[] list()`
Retorna a lista de arquivos e sub-diretórios contidos neste objeto. Este objeto deveria ser um diretório.
`public void mkdir()`
Cria um diretório denotado por este caminho abstrato.
`public void delete()`
Remove o arquivo ou diretório representado por este objeto *File*.

Tabela 2: Métodos da classe File

Veremos como estes métodos trabalham, através da classe a seguir:

```
import java.io.*;

public class FileDemo {
    public static void main(String args[]) {
        String fileName = "temp.txt";
        File fn = new File(fileName);
        System.out.println("Name: " + fn.getName());
        if (!fn.exists()) {
            System.out.println(fileName + " does not exists.");
            /* Cria um diretório temporário. */
            System.out.println("Creating temp directory...");
            fileName = "temp";
            fn = new File(fileName);
            fn.mkdir();
            System.out.println(fileName +
                    (fn.exists()? "exists": "does not exist"));
            System.out.println("Deleting temp directory...");
            fn.delete();
            System.out.println(fileName +
                    (fn.exists()? "exists": "does not exist"));
            return;
        }
        System.out.println(fileName + " is a " +
                (fn.isFile()? "file." :"directory."));
        if (fn.isDirectory()) {
            String content[] = fn.list();
            System.out.println("The content of this directory:");
            for (int i = 0; i < content.length; i++) {
                System.out.println(content[i]);
            }
        }
        if (!fn.canRead()) {
            System.out.println(fileName + " is not readable.");
            return;
        }
        System.out.println(fileName + " is " + fn.length() +
                " bytes long.");
        System.out.println(fileName + " was last modified on " +
                fn.lastModified() + ".");
        if (!fn.canWrite()) {
            System.out.println(fileName + " is not writable.");
        }
    }
}
```

Este é o resultado da execução da classe *FileDemo*:

```
Name: temp.txt
temp.txt is a file.
temp.txt is 34 bytes long.
temp.txt was last modified on 1149150489177.
```

temp.txt, deve ser colocado no diretório raiz do seu projeto e contém o seguinte texto:

```
what a wonderful world
1, 2, step
```

4. Classe *Reader*

Esta seção descreve *Stream* de caracteres que são utilizados para leitura.

4.1. Métodos de Reader

A classe *Reader* consiste de diversos métodos para leitura de caracteres. Aqui estão alguns dos métodos desta classe:

Métodos da Classe *Reader*
`public int read(-) throws IOException`
Possui três versões. Lê caractere(s), uma matriz inteira de caracteres, ou uma porção de uma matriz de caracteres.
`public int read()` - Lê um único caracter.
`public int read(char[] cbuf)`- Lê caracteres e os armazena na matriz de caracteres *cbuf*.
`public abstract int read(char[] cbuf, int offset, int length)`- Lê até *length* número de caracteres e os armazena na matriz de caracteres *cbuf* começando no *offset* especificado.
`public abstract void close() throws IOException`
Fecha este Stream. Chamar os outros métodos *Reader* após fechar o Stream iria causar a ocorrência de uma exceção *IOException*.
`public void mark(int readAheadLimit) throws IOException`
Marca a posição atual no Stream. Após marcar, chamadas ao método reset() irão tentar reposicionar o Stream neste ponto. Nem todos os Stream de entrada de caracteres suportam esta operação.
`public boolean markSupported()`
Indica se um Stream suporta a operação de marca ou não. Não é suportado por padrão. Não deveria ser sobrescrito por subclasses.
`public void reset() throws IOException`
Reposiciona o Stream na última posição marcada.

Tabela 3: Métodos da classe Reader

4.2. Subclasses de Reader

A seguir temos algumas das subclasses de *Reader*:

Subclasses de *Reader*
`FileReader`
Para leitura a partir de arquivos de caracteres.
`CharArrayReader`
Implementa um *buffer* de caracteres a partir do qual pode-se ler.
`StringReader`
Para leitura a partir de uma String.
`PipedReader`

Usado em pares (com um correspondente *PipedWriter*) por duas tarefas que queiram comunicar-se. Uma destas tarefas lê caracteres a partir desta fonte.

Tabela 4: Subclasses de Reader

4.3. Subclasses de FilterReader

Para adicionar funcionalidades às classes *Reader* básicas, é possível utilizar as subclasses *FilterReader*. Aqui estão algumas destas subclasses:

Subclasses de *FilterReader*
`BufferedReader`
Permite o armazenamento de caracteres em um buffer, de forma a prover uma eficiente leitura de caracteres, matrizes, e linhas.
`FilterReader`
Para ler Stream de caracteres filtrados.
`InputStreamReader`
Converte bytes lidos em caracteres.
`LineNumberReader`
Uma subclasse da classe *BufferedReader* que é capaz de manter registro do número das linhas.
`PushbackReader`
Uma subclasse da classe *FilterReader* que permite que caracteres sejam devolvidos ou não copiados ao Stream.

Tabela 5: Subclasses de FilterReader

5. Classe *Writer*

Nesta seção descrevemos os *Stream* de caracteres que são utilizados para escrita.

5.1. Métodos de Writer

A classe *Writer* consiste de diversos métodos para escrita de caracteres. Aqui estão alguns dos métodos desta classe:

Métodos da classe *Writer*
`public void write(-) throws IOException`
Possue cinco versões:
`public void write(int c)` – Escreve um único caracter representado pelo valor inteiro dado.
`public void write(char[] cbuf)` – Escreve o conteúdo da matriz de caracteres *cbuf*.
`public abstract void write(char[] cbuf, int offset, int length)` – Escreve *length* número de caracteres a partir da matriz *cbuf* , começando no *offset* especificado.
`public void write(String str)` – Escreve a string *string*.
`public void write(String str, int offset, int length)` – Escreve *length* número de caracteres a partir da string *str*, começando no *offset* especificado.
`public abstract void close() throws IOException`
Fecha este *Stream* após descarregar quaisquer caracteres que não tenham sido escritos. Invocação de outros métodos depois de fechar este *Stream* iriam causar a ocorrência de uma exceção *IOException*.
`public abstract void flush()`
Descarrega o *Stream* (ex., caracteres salvos no buffer são imediatamente escritos à destinação pretendida).

Tabela 6: Métodos da classe Writer

5.2. Subclasses de Writer

A seguir temos algumas das subclasses de *Writer*:

Subclasses de *Writer*
`FileWriter`
Para escrever caracteres a um arquivo.
`CharArrayWriter`
Implementa um buffer de caracteres para o qual pode-se escrever.
`StringWriter`
Para escrever em uma String.
`PipedWriter`
Usado em pares (com um correspondente *PipedReader*) por duas tarefas que queiram comunicar-se. Uma destas tarefas escreve caracteres a este *Stream*.

Tabela 7: Subclasses de Writer

5.3. Subclasses de FilterWriter

Para adicionar funcionalidades às classes *Writer* básicas, é possível utilizar as subclasses *FilterWriter*. Aqui estão algumas destas classes:

Subclasses de *FilterWriter*
`BufferedWriter`
Permite o uso de buffers de caracteres de forma a prover eficiente escrita de caracteres, matrizes, e linhas.
`FilterWriter`
Para escrever Stream de caracteres filtrados.
`OutputStreamWriter`
Codifica caracteres escritos a ele em bytes.
`PrintWriter`
Imprime representações formatadas dos objetos a um Stream de saída de texto.

Tabela 8: Subclasses de Filter Writer

6. Um Exemplo de *Reader/Writer*

O exemplo a seguir utiliza as classes *FileReader* e *FileWriter* para ler a partir de um arquivo especificado pelo usuário e copiar o conteúdo deste para um outro arquivo:

```
import java.io.*;

class CopyDemo {
    private void copy(String input, String output) {
        FileReader reader;
        FileWriter writer;
        int data;
        try {
            reader = new FileReader(input);
            writer = new FileWriter(output);
            while ((data = reader.read()) != -1) {
                writer.write(data);
            }
            reader.close();
            writer.close();
        } catch (IOException ie) {
            ie.printStackTrace();
        }
    }

    public static void main(String args[]) {
        CopyDemo cf = new CopyDemo();
        cf.copy("temp.txt", "temp2.txt");
    }
}
```

Execute a classe e observe o que acontece com os arquivos manipulados.

Usando *temp.txt* a partir de nosso exemplo anterior, aqui está o resultado quando passamos *temp.txt* como o *inputFile* e *temp2.txt* como o *outputFile*:

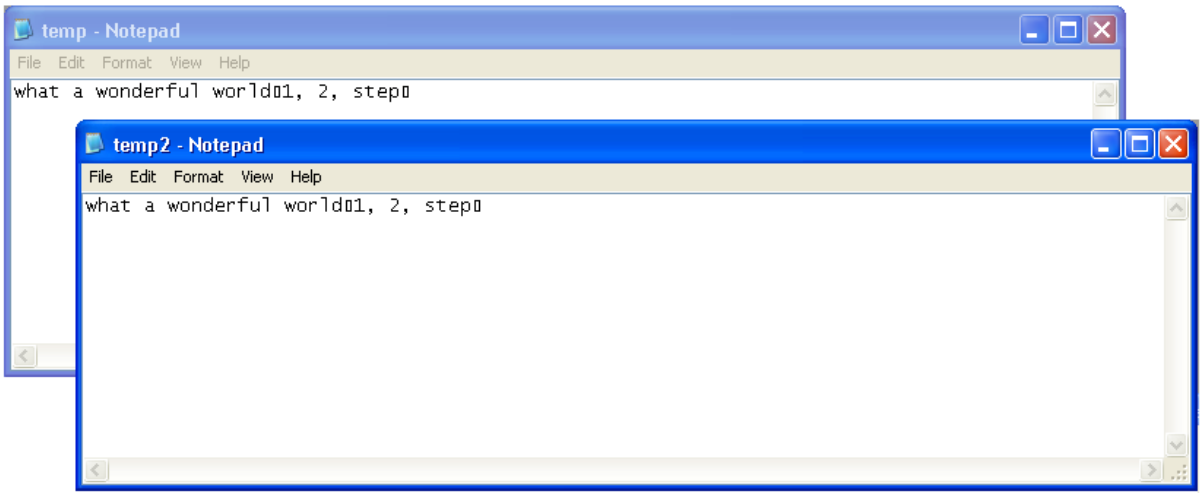

Figura 1: Saída de dados para CopyFile

7. Exemplo *Reader/Writer* modificado

O exemplo a seguir é similar ao anterior, entretanto é mais eficiente. Ao invés de ler e escrever um *Stream* de cada vez, caracteres lidos são primeiramente armazenados em um *buffer* antes de que caracteres sejam escritos linha por linha. O programa usa a técnica de encadeamento de Stream desde que as classes *FileReader* e *FileWriter* sejam decoradas com as classes *BufferedReader* e *BufferedWriter*, respectivamente:

```
import java.io.*;

class CopyDemo {
   void copy(String input, String output) {
      BufferedReader reader;
      BufferedWriter writer;
      String data;
      try {
         reader = new BufferedReader(new FileReader(input));
         writer = new BufferedWriter(new FileWriter(output));
         while ((data = reader.readLine()) != null) {
            writer.write(data, 0, data.length());
         }
         reader.close();
         writer.close();
      } catch (IOException ie) {
         ie.printStackTrace();
      }
   }

   public static void main(String args[]) {
      CopyDemo cf = new CopyDemo();
      cf.copy("temp.txt", "temp2.txt");
   }
}
```

Aqui está o resultado desta versão de *CopyDemo*.

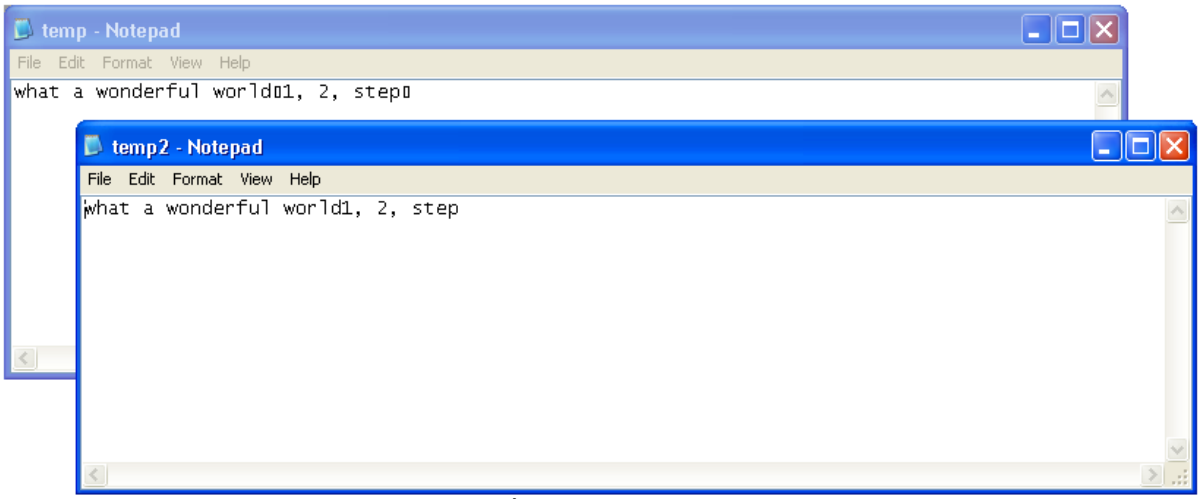

Figura 2: Saída de Dados para CopyDemo

8. Classe *InputStream*

Esta seção dá uma visão geral dos diferentes *Stream* de *bytes* que são utilizados para leitura.

8.1. Métodos de InputStream

A classe *InputStream* consiste em diversos métodos para leitura de *bytes*. Aqui estão alguns dos métodos desta classe:

Métodos da classe InputStream
`public int read(-) throws IOException`
Um método sobrecarregado, o qual também tem três versões como as da classe *Reader*. Lê bytes.
`public abstract int read()` - Lê o próximo byte de dados a partir deste Stream.
`public int read(byte[] bBuf)`- Lê certo número de bytes e os armazena na matriz de bytes *bBuf*.
`public abstract int read(char[] cbuf, int offset, int length)`- Lê até *length* número de bytes e os armazena na matriz de bytes *bBuf* começando no *offset* especificado.
`public abstract void close() throws IOException`
Fecha este *Stream*. Chamar os outros métodos *InputStream* após ter fechado o *Stream* iria causar a ocorrência de uma exceção *IOException*.
`public void mark(int readAheadLimit) throws IOException`
Marca a posição atual no Stream. Após marcar, chamadas ao método reset() irão tentar reposicionar o *Stream* neste ponto. Nem todos os Stream de entrada de bytes suportam esta operação.
`public boolean markSupported()`
Indica se um *Stream* suporta as operações marca e limpeza. Não suportado por padrão. Deve ser feito override pelas subclasses.
`public void reset() throws IOException`
Reposiciona o *Stream* na última posição marcada.

Tabela 9: Métodos da classe InputStream

8.2. Subclasses de InputStream

As seguintes são algumas das subclasses de *InputStream*:

Subclasses de *InputStream*
`FileInputStream`
Para leitura de bytes a partir de um arquivo.
`BufferedInputStream`
Implementa um buffer que contém bytes, os quais podem ser lidos a partir do *Stream*.
`PipedInputStream`
Deveria estar conectado a um *PipedOutputStream*. Estes Stream são tipicamente usados por duas tarefas desde que um destas tarefas leia dados a partir desta fonte enquanto a outra tarefa escreve ao correspondente *PipedOutputStream*.

Tabela 10: Subclasses de InputStream

8.3. Subclasses de FilterInputStream

Para adicionar funcionalidades às classes *InputStream* básicas, utilizamos as subclasses de *FilterInputStream*. Aqui estão algumas destas subclasses:

Subclasses de FilterInputStream
`BufferedInputStream`
Uma subclasse de *FilterInputStream* que permite o uso de um buffer de entrada de forma a prover a eficiente leitura de bytes.
`FilterInputStream`
Para ler Stream de bytes filtrados, os quais podem transformar a fonte básica de dados no decorrer do processo e prover funcionalidades adicionais.
`ObjectInputStream`
Usado para serialização de objetos. De-serializa objetos e dados primitivos previamente escritos usando um *ObjectOutputStream*.
`DataInputStream`
Uma subclasse de *FilterInputStream* que permite que uma aplicação leia dados Java primitivos a partir de um *Stream* de entrada de dados subjacente, independente do tipo de máquina.
`LineNumberInputStream`
Uma subclasse de *FilterInputStream* que permite monitorar o número de linha atual.
`PushbackInputStream`
Uma subclasse da classe *FilterInputStream* que permite que bytes sejam devolvidos ou não transferidos ao *Stream*.

Tabela 11: Subclasses de Filter InputStream

9. Classe *OutputStream*

Esta seção dá uma visão geral dos diferentes *Stream* de *bytes* que são utilizados para escrita.

9.1. Métodos de OutputStream

A classe *OutputStream* consiste em diversos métodos para escrever bytes. Aqui estão alguns dos métodos desta classe:

Métodos da classe OutputStream
`public void write(-) throws IOException`
Um método sobrecarregado para escrever bytes ao *Stream*. Ele tem três versões:
`public abstract void write(int b)` – Escreve o valor em bytes *b* especificado a este Stream de saída de dados.
`public void write(byte[] bBuf)` – Escreve o conteúdo da matriz de bytes *bBuf neste* Stream.
`public void write(byte[] bBuf, int offset, int length)` – Escreve *length* número de bytes a partir da matriz *bBuf* neste Stream, começando pelo *offset* especificado para este Stream.
`public abstract void close() throws IOException`
Fecha este Stream e libera quaisquer recursos do sistema associados com este Stream. Invocação de outros métodos após chamar este método iria causar a ocorrência de uma exceção *IOException*.
`public abstract void flush()`
Descarrega o Stream (ex., bytes salvos no buffer são imediatamente escritos à destinação pretendida).

Tabela 12: Métodos da classe OutputStream

9.2. Subclasses de OutputStream

As seguintes são algumas das subclasses de *OutputStream*:

Subclasses de OutputStream
`FileOutputStream`
Para escrever bytes a um arquivo.
`BufferedOutputStream`
Implementa um buffer que contém bytes, os quais podem ser escritos no Stream.
`PipedOutputStream`
Deveria estar conectado a um *PipedInputStream*. Estes Stream são tipicamente usados por duas tarefas desde que uma destas escreva dados a este Stream enquanto a outra tarefa lê a partir do correspondente *PipedInputStream*.

Tabela 13: Subclasses de OutputStream

9.3. Subclasses de FilterOutputStream

Para adicionar funcionalidades às classes *OutputStream* básicas, utilizamos as subclasses de *Filter Stream*. Aqui estão algumas destas subclasses:

Subclasses de FilterOutputStream
`BufferedOutputStream`
Uma subclasse de *FilterOutputStream* que permite o uso de buffers de saída de forma a prover uma eficiente escrita de bytes. Permite escrever bytes ao Stream de saída de dados subjascente sem necessariamente causar uma chamada ao sistema subjascente para cada byte escrito.
`FilterOutputStream`
Para escrever Stream de bytes filtrados, os quais podem transformar a fonte de dados básica ao longo do processo e prover funcionalidades adicionais.
`ObjectOutputStream`
Usado para serialização de objetos. Serializa objetos e dados primitivos a um *OutputStream*.
`DataOutputStream`
Uma subclasse de *FilterOutputStream* que permite que uma aplicação escreva dados Java primitivos a um Stream de saída de dados subjascent, independentemente do tipo de máquina.
`PrintStream`
Uma subclasse de *FilterOutputStream* que provê capacidade para imprimir representações de diversos valores de dados convenientemente.

Tabela 14: Classes Filter OutputStream

10. Um Exemplo de InputStream/OutputStream

O exemplo a seguir utiliza as classes *FileInputStream* e *FileOutputStream* para ler a partir de um arquivo especificado pelo usuário e copiar o conteúdo deste para um outro arquivo:

```
import java.io.*;

class CopyDemo {
   void copy(String input, String output) {
      FileInputStream inputStr;
      FileOutputStream outputStr;
      int data;
      try {
         inputStr = new FileInputStream(input);
         outputStr = new FileOutputStream(output);
         while ((data = inputStr.read()) != -1) {
            outputStr.write(data);
         }
         inputStr.close();
         outputStr.close();
      } catch (IOException ie) {
         ie.printStackTrace();
      }
   }

   public static void main(String args[]) {
      CopyDemo cf = new CopyDemo();
      cf.copy("temp.txt", "temp2.txt");
   }
}
```

Aqui está o resultado da execução desta classe:

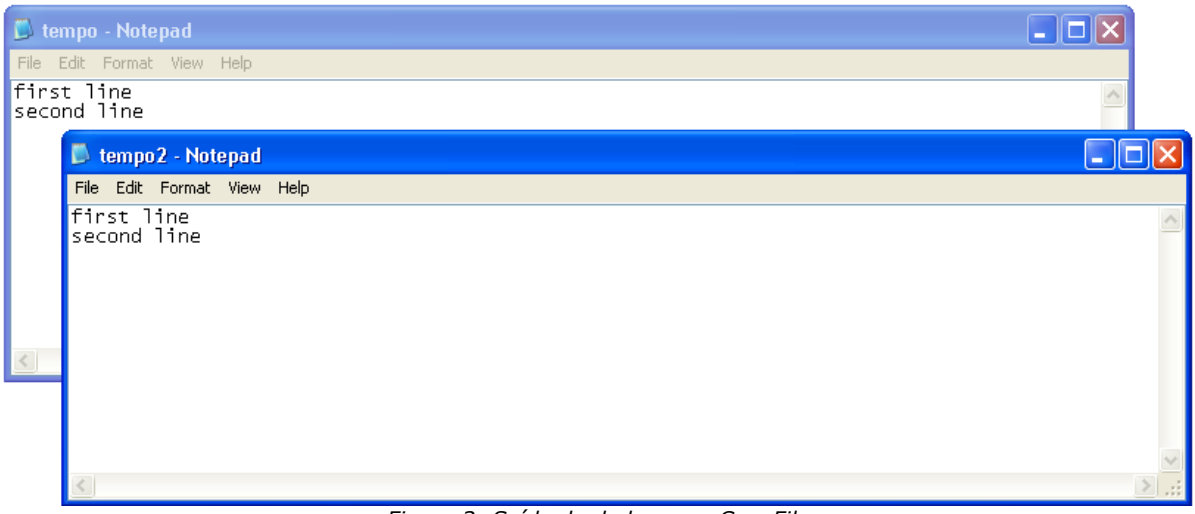

Figura 3: Saída de dados para CopyFile

11. Exemplo InputStream/OutputStream Modificado

Este exemplo utiliza a classe *PushbackInputStream* que decora um objeto *FileInputStream* usando a classe *PrintStream*.

```
import java.io.*;

class CopyDemo {
   void copy(String input) {
      PushbackInputStream inputStr;
      PrintStream outputStr;
      int data;
      try {
         inputStr = new PushbackInputStream(new
                                        FileInputStream(input));
         outputStr = new PrintStream(System.out);
         while ((data = inputStr.read()) != -1) {
            outputStr.println("read data: " + (char) data);
            inputStr.unread(data);
            data = inputStr.read();
            outputStr.println("unread data: " + (char) data);
         }
         inputStr.close();
         outputStr.close();
      } catch (IOException ie) {
         ie.printStackTrace();
      }
   }

   public static void main(String args[]) {
      CopyDemo cf = new CopyDemo();
      cf.copy("temp.txt");
   }
}
```

Teste este código em um arquivo contendo algumas poucas linhas ou caracteres. Supondo um arquivo chamado *temp.txt* contendo o seguinte texto:

```
one 1
two
```

Ao executarmos esta classe obteremos a seguinte saída:

```
read data: o                          unread data:
unread data: o                        read data:
read data: n
unread data: n                        unread data:
read data: e
unread data: e                        read data: t
read data:                            unread data: t
unread data:                          read data: w
read data: 1                          unread data: w
unread data: 1                        read data: o
read data:                            unread data: o
```

12. Serialização

A Máquina Virtual Java ou *JVM* possui a habilidade de ler ou escrever um objeto a um *Stream*. Esta capacidade é chamada **Serialização**, corresponde ao processo de "achatar" um objeto de forma tal que o mesmo possa ser salvo a uma fonte de armazenamento permanente ou passado a outro objeto via a classe *OutputStream*. Ao escrever um objeto, é importante que o seu estado seja escrito em uma forma serializada de tal modo que o objeto possa ser reconstruído conforme o mesmo está sendo lido. Salvar um objeto a alguma forma de armazenamento permanente é conhecido como persistência.

Os *Stream* podem ser utilizados para de-serializar e re-serializar. São representados respectivamente pelas classes *ObjectInputStream* e *ObjectOutputStream*.

Para permitir que um objeto seja serializável (isto é, possa ser salvo e recuperado), a classe deve implementar a interface *Serializable*. A classe também deve prover um construtor padrão (ou um construtor sem argumentos). Uma das coisas interessantes a respeito de serialização é que a mesma é herdada, o que significa que não precisamos implementar *Serializable* em cada classe. Isso significa menos trabalho para os programadores. É possível simplesmente implementar *Serializable* uma única vez ao longo da hierarquia de classes.

12.1. A Palavra-chave transient

Quando um objeto é serializado, apenas os dados do objeto são preservados. Métodos e construtores não são parte do *Stream* serializado. Há, no entanto, alguns objetos que não são serializáveis porque os dados que eles representam mudam constantemente. Alguns exemplos de tais objetos são *FileInputStream* e *Thread*. Uma exceção *NotSerializableException* é lançada se a operação de serialização falhar por qualquer motivo.

Não há necessidade em se desesperar. Uma classe contendo um objeto não serializável ainda pode ser serializada se a referência e este objeto não serializável for marcada com a palavra-chave *transient*. Considere o seguinte exemplo:

```
class MyClass implements Serializable {
   transient Thread thread;   //tente remover transient
   int data;
   /* alguns outros dados */
}
```

A palavra-chave *transient* previne que os dados associados sejam serializados. Objetos instanciados a partir desta classe podem agora ser escritos a um *OutputStream*.

12.2. Serialização: Escrevendo um Stream de Objetos

Para escrever um objeto a um *Stream*, é necessário utilizar a classe *ObjectOutputStream* e seu método *writeObject*. O método *writeObject* tem a seguinte assinatura:

```
public final void writeObject(Object obj) throws IOException
```

onde *obj* é o objeto que será escrito ao *Stream*.

O exemplo abaixo escreve um objeto *Boolean* a um *ObjectOutputStream*. A classe *Boolean* implementa a interface *Serializable*. Deste modo, os objetos instanciados a partir desta classe podem ser escritos e lidos a partir de um *Stream*.

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

```
public class SerializeDemo {
    public SerializeDemo() {
        Boolean booleanData = new Boolean("true");
        try {
            FileOutputStream fos = new
                    FileOutputStream("boolean.txt");
            ObjectOutputStream oos = new ObjectOutputStream(fos);
            oos.writeObject(booleanData);
            oos.close();
        } catch (IOException ie) {
            ie.printStackTrace();
        }
    }
    public static void main(String args[]) {
        new SerializeDemo();
    }
}
```

12.3. De-serialização: Lendo um Stream de Objetos

Para ler um objeto a partir de um *Stream*, é necessário utilizar a classe *ObjectInputStream* e seu método *readObject*. O método *readObject* tem a seguinte assinatura:

```
public final Object readObject()
             throws IOException, ClassNotFoundException
```

onde *obj* é o objeto a ser lido a partir do *Stream*.

O tipo *Object* retornado deveria sofrer typecasting ao nome de classe apropriado antes que métodos naquela classe possam ser executados.

O exemplo abaixo lê um objeto *Boolean* a partir de um *ObjectInputStream*. Esta é uma continuação do exemplo anterior que tratava de serialização.

```
import java.io.*;

public class UnserializeDemo {
    public UnserializeDemo() {
      Boolean booleanData = null;
      try {
         FileInputStream fis = new
                           FileInputStream("boolean.txt");
         ObjectInputStream ois = new ObjectInputStream(fis);
         booleanData = (Boolean) ois.readObject();
         ois.close();
      } catch (Exception e) {
         e.printStackTrace();
      }
      System.out.println("Unserialized Boolean from " +
                         "boolean.txt");
      System.out.println("Boolean data: " + booleanData);
      System.out.println("Compare data with true: " +
                  booleanData.equals(new Boolean("true")));
   }
   public static void main(String args[]) {
      new UnserializeDemo();
   }
}
```

O seguinte é a saída de dados esperada de *UnserializeBoolean*:

```
Unserialized Boolean from boolean.ser
Boolean data: true
Compare data with true: true
```

Parceiros que tornaram JEDI™ possível

Instituto CTS
Patrocinador do DFJUG.

Sun Microsystems
Fornecimento de servidor de dados para o armazenamento dos vídeo-aulas.

Java Research and Development Center da Universidade das Filipinas
Criador da Iniciativa JEDI™.

DFJUG
Detentor dos direitos do JEDI™ nos países de língua portuguesa.

Banco do Brasil
Disponibilização de seus *telecentros* para abrigar e difundir a Iniciativa JEDI™.

Politec
Suporte e apoio financeiro e logístico a todo o processo.

Borland
Apoio internacional para que possamos alcançar os outros países de língua portuguesa.

Instituto Gaudium/CNBB
Fornecimento da sua infra-estrutura de hardware de seus servidores para que os milhares de alunos possam acessar o material do curso simultaneamente.