Appunti dei corsi di

Programmazione di Rete
Sistemi di elaborazione: Reti II

PROF. G. BONGIOVANNI

1) Utilizzo di Java per lo sviluppo di applicazioni di rete

In questa parte del corso si affronta il problema dello sviluppo di applicazioni di rete di tipo client-server.

Tale architettura software è praticamente una scelta obbligata al giorno d'oggi, in quanto si adatta perfettamente alle attuali reti di elaboratori (costituite da molteplici host indipendenti e non più da un singolo mainframe al quale sono connessi vari terminali).

In particolare, si vedrà:
- come sviluppare un client per i protocolli ASCII del livello application dell'architettura TCP/IP;
- come sviluppare un server multithreaded capace di gestire molti client contemporaneamente;
- quali vantaggi si possono ottenere impacchettando il flusso di dati da trasmettere in una serie di messaggi.

Figura 1-1: Formato di un messaggio

Il linguaggio che verrà usato è Java, in quanto dotato di molti vantaggi rispetto a possibili concorrenti:
- è totalmente a oggetti;
- possiede una interfaccia di programmazione molto pulita;
- è multipiattaforma (non ci si deve preoccupare di problemi di interoperabilità fra piattaforme diverse);
- è dotato di strumenti potenti e di alto livello per:
  - l'apertura di canali di comunicazione (classe `Socket` e `ServerSocket`);
  - la gestione degli errori (classe `Exception`);
  - la gestione della concorrenza (classe `Thread`).

Per le sue caratteristiche, questo linguaggio permette di ottenere con facilità le funzioni base richieste a un applicazione di rete, e di concentrarsi sugli aspetti più caratterizzanti dell'applicazione stessa.

Per poter affrontare lo sviluppo di applicazioni di rete, è necessario approfondire la conoscenza del linguaggio nei seguenti settori:

- gestione di Input/Output;
- gestione dei Socket;
- gestione dei Thread.

1.1) Ripasso di AWT

Vediamo ora un'***applicazione*** (cioè un programma che può girare da solo, appoggiandosi a una macchina virtuale, senza bisogno di una pagina HTML che lo richiama).

Le applicazioni non hanno le limitazioni degli applet, che tipicamente:

- non possono accedere al file system locale;
- non possono aprire connessioni di rete con host diversi da quello di provenienza.

Un'applicazione è una classe al cui interno esiste un metodo `main(String args[])` che viene eseguito all'avvio.

Esempio 1

Questa semplicissima applicazione presenta alcuni campi testo e alcuni bottoni, ed è predisposta per la gestione degli eventi.

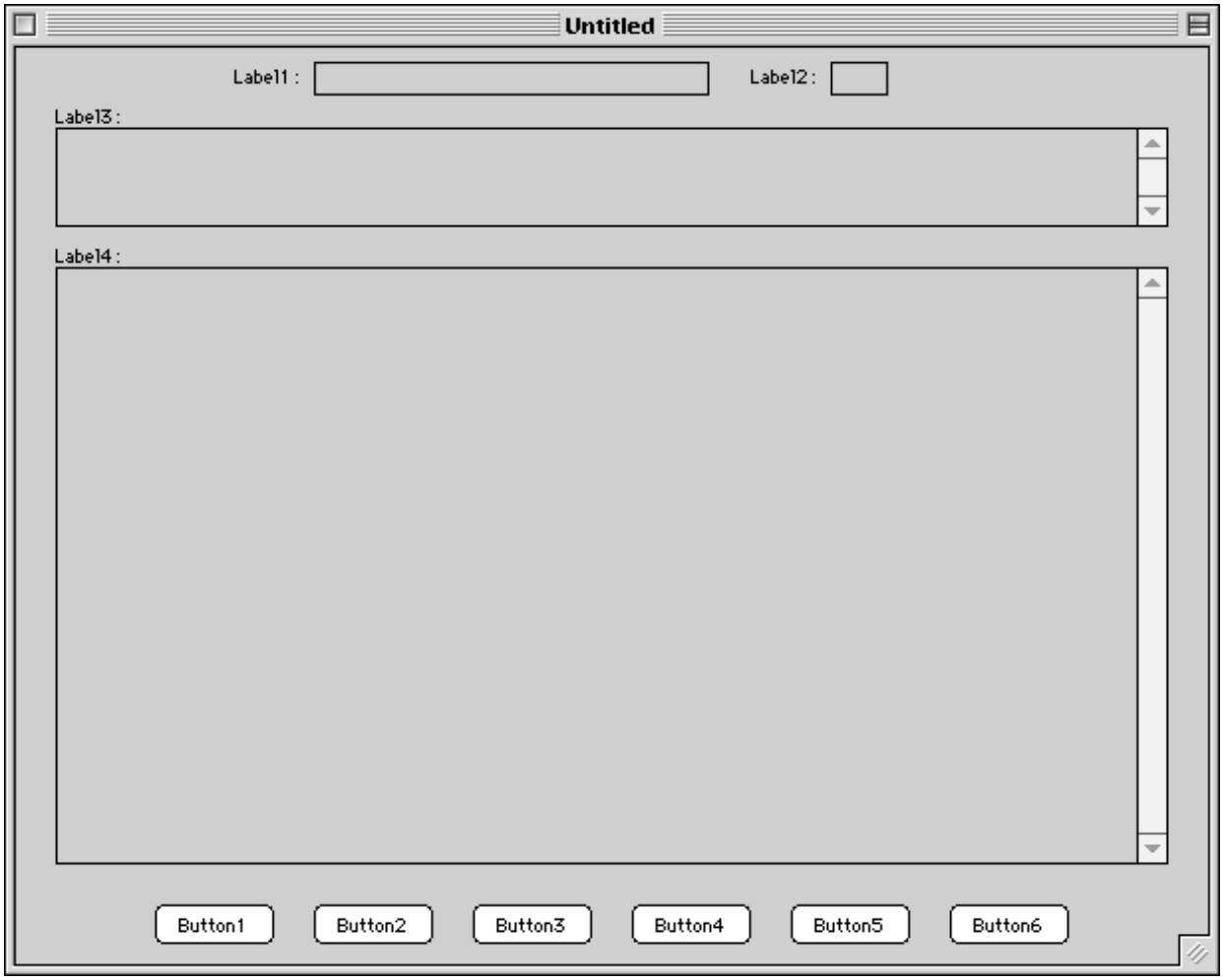

Figura 1-2: Interfaccia utente dell'esempio 1

Nel nostro caso, il `main()` crea una nuova istanza di un oggetto della classe `BaseAppE1`, che estende `Frame` (in pratica una finestra).

```
import java.awt.*;

public class BaseAppE1 extends Frame    {
    Label label1, label2, label3, label4;
    TextField textField1, textField2;
    TextArea textArea1, textArea2;
```

```
    Button button1, button2, button3, button4, button5, button6;
//------------------------------------------------
public BaseAppE1() {
        this.setLayout(null);
        label1 = new Label("Label1:");
        label1.reshape(110, 5, 40, 15);
        this.add(label1);
        textField1 = new TextField();
        textField1.reshape(150, 10, 200, 15);
        this.add(textField1);
        label2 = new Label("Label2:");
        label2.reshape(370, 5, 40, 15);
        this.add(label2);
        textField2 = new TextField();
        textField2.reshape(410, 10, 30, 15);
        this.add(textField2);
        label3 = new Label("Label3:");
        label3.reshape(20, 25, 100, 15);
        this.add(label3);
        textArea1 = new TextArea();
        textArea1.reshape(20, 40, 560, 50);
        this.add(textArea1);
        label4 = new Label("Label4:");
        label4.reshape(20, 95, 100, 15);
        this.add(label4);
        textArea2 = new TextArea();
        textArea2.reshape(20, 110, 560, 300);
        this.add(textArea2);
        button1 = new Button("Button1");
        button1.reshape(70, 430, 60, 20);
        this.add(button1);
        button2 = new Button("Button2");
        button2.reshape(150, 430, 60, 20);
        this.add(button2);
        button3 = new Button("Button3");
        button3.reshape(230, 430, 60, 20);
        this.add(button3);
        button4 = new Button("Button4");
        button4.reshape(310, 430, 60, 20);
        this.add(button4);
        button5 = new Button("Button5");
        button5.reshape(390, 430, 60, 20);
        this.add(button5);
        button6 = new Button("Button6");
        button6.reshape(470, 430, 60, 20);
        this.add(button6);
        resize(600, 460);
        show();
    }
//------------------------------------------------
    public static void main(String args[]) {
            new BaseAppE1();
    }
//------------------------------------------------
    public boolean handleEvent(Event event) {
        if (event.id == Event.WINDOW_DESTROY) {
            hide();          // hide the Frame
            dispose();       // tell windowing system to free resources
            System.exit(0); // exit
            return true;
        }
        if (event.target == button1 && event.id == Event.ACTION_EVENT) {
            button1_Clicked(event);
        }
```

```
        if (event.target == button2 && event.id == Event.ACTION_EVENT) {
            button2_Clicked(event);
        }
        if (event.target == button3 && event.id == Event.ACTION_EVENT) {
            button3_Clicked(event);
        }
        if (event.target == button4 && event.id == Event.ACTION_EVENT) {
            button4_Clicked(event);
        }
        if (event.target == button5 && event.id == Event.ACTION_EVENT) {
            button5_Clicked(event);
        }
        if (event.target == button6 && event.id == Event.ACTION_EVENT) {
            button6_Clicked(event);
        }
        return super.handleEvent(event);
    }
//-------------------------------------------------
    void button1_Clicked(Event event) {
            textArea2.setText(textArea2.getText() + "Hai premuto bottone 1\n");
    }
//-------------------------------------------------
    void button2_Clicked(Event event) {
            textArea2.setText(textArea2.getText() + "Hai premuto bottone 2\n");
    }
//-------------------------------------------------
    void button3_Clicked(Event event) {
        textArea2.setText(textArea2.getText() + "Hai premuto bottone 3\n");
    }
//-------------------------------------------------
    void button4_Clicked(Event event) {
        textArea2.setText(textArea2.getText() + "Hai premuto bottone 4\n");
    }
//-------------------------------------------------
    void button5_Clicked(Event event) {
        textArea2.setText(textArea2.getText() + "Hai premuto bottone 5\n");
    }
//-------------------------------------------------
    void button6_Clicked(Event event) {
        textArea2.setText(textArea2.getText() + "Hai premuto bottone 6\n");
    }
}
```

1.2) Input/Output in Java

L'I/O in Java è definito in termini di ***stream*** (***flussi***). Gli stream sono un'astrazione di alto livello per rappresentare la connessione a un canale di comunicazione.

Il canale di comunicazione può essere costituito fra entità molto diverse, le più importanti delle quali sono:

- un file;
- una connessione di rete (ad esempio TCP/IP);
- un buffer in memoria.

Grazie all'astrazione rappresentata dagli stream, le operazioni di I/O dirette a (o provenienti da) uno qualunque degli oggetti di cui sopra sono realizzate con la stessa interfaccia.

Uno stream rappresenta un ***punto terminale*** di un canale di comunicazione unidirezionale, e può leggere dal canale (`InputStream`) o scrivervi (`OutputStream`):

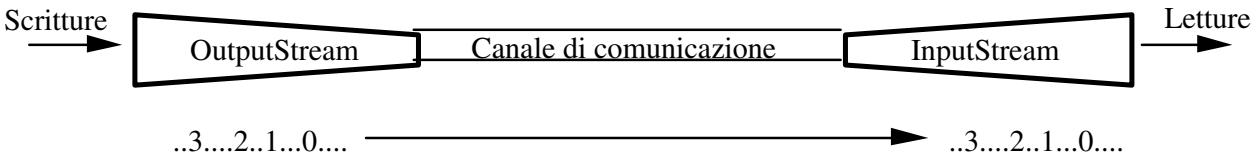

Figura 1-3: Stream di input e output

Tutto ciò che viene scritto sul canale tramite l'`OutputStream` viene letto dall'altra parte dal corrispondente `InputStream`.

Gli stream hanno diverse proprietà:

- sono ***FIFO***: ciò che viene scritto da un `OutputStream` viene letto nello stesso ordine dal corrispondente `InputStream`;
- sono ***ad accesso sequenziale***: non è fornito alcun supporto per l'accesso casuale (solo la classe `RandomAccessFile`, che però non è uno stream, offre tale tipo di accesso);
- sono ***rea4-only oppure write-only***: uno stream consente di leggere (`InputStream`) o scrivere (`OutputStream`) ma non entrambe le cose. Se ambedue le funzioni sono richieste, ci vogliono 2 distinti stream: questo è un caso tipico delle connessioni di rete, tant'è che da una connessione (`Socket`) si ottengono due stream, uno in scrittura e uno in lettura;
- sono ***bloccanti***: la lettura blocca il programma che l'ha richiesta finché i dati non sono disponibili. Analogamente, la scrittura blocca il richiedente finché non è completata;
- quasi tutti i loro metodi possono ***generare eccezioni***.

1.2.1) Classe InputStream

E' la classe astratta dalla quale derivano tutti gli stream finalizzati alla lettura da un canale di comunicazione.

Costruttore

```
public InputStream();
```

Vuoto.

Metodi più importanti

```
public abstract int read() throws IOException;
```

Legge e restituisce un byte (range 0-255) o si blocca se non c'è niente da leggere. Restituisce il valore -1 se incontra la fine dello stream.

```
public int read(byte[] buf) throws IOException;
```

Legge fino a riempire l'array `buf`, o si blocca se non ci sono abbastanza dati. Restituisce il numero di byte letti, o il valore -1 se incontra la fine dello stream. L'implementazione di default chiama tante volte la `read()`, che è astratta.

```
public int read(byte[] buf, int off, int len) throws IOException;
```

Legge fino a riempire l'array `buf`, a partire dalla posizione `off`, con `len` byte (o fino alla fine dell'array). Si blocca se non ci sono abbastanza dati. Restituisce il numero di byte letti, o il valore -1 se incontra la fine dello stream. L'implementazione di default chiama tante volte la `read()`, che è astratta.

```
public int available() throws IOException;
```

Restituisce il numero di byte che possono essere letti senza bloccarsi (cioè che sono disponibili):

- con un file: quelli che rimangono da leggere fino alla fine del file;
- con una connessione di rete: anche 0, se dall'altra parte non è ancora stato mandato nulla.

```
public void close() throws IOException;
```

Chiude lo stream e rilascia le risorse ad esso associate. Eventuali dati non ancora letti vengono perduti.

Questi metodi, e anche gli altri non elencati, sono supportati anche dalle classi derivate, e quindi sono disponibili sia che si legga da file, da un buffer di memoria o da una connessione di rete.

1.2.2) Classe OutputStream

E' la classe astratta dalla quale derivano tutti gli Stream finalizzati alla scrittura su un canale di comunicazione.

Costruttore

```
public OutputStream();
```

Vuoto.

Metodi più importanti

```
public abstract void write(int b) throws IOException;
```

Scrive un byte (8 bit) sul canale di comunicazione. Il parametro è `int` (32 bit) per convenienza (il risultato di espressioni su byte è intero), ma solo gli 8 bit meno significativi sono scritti.

```
public void write(byte[] buf) throws IOException;
```

Scrive un array di byte. Blocca il chiamante finché la scrittura non è completata..

```
public void write(byte[] buf, int off, int len) throws IOException;
```

Scrive la parte di array che inizia a posizione `off`, e consiste di `len` byte. Blocca il chiamante finché la scrittura non è completata..

```
public void flush() throws IOException;
```

Scarica lo stream, scrivendo effettivamente tutti i dati eventualmente mantenuti in un buffer locale per ragioni di efficienza. Ciò deriva dal fatto che in generale si introduce molto overhead scrivendo pochi dati alla volta sul canale di comunicazione:

- su file: molte chiamate di sistema;
- su connessione di rete: un TPDU per pochi byte.

```
public void close() throws IOException;
```

Chiama `flush()` e poi chiude lo stream, liberando le risorse associate. Eventuali dati scritti prima della `close()` vengono comunque trasmessi, grazie alla chiamata di `flush()`.

Esempio 2

Applicazione che copia ciò che viene immesso tramite lo ***standard input*** sullo `standard output`.

A tal fine si usano due stream accessibili sotto forma di ***field*** (cioè variabili) nella ***classe statica*** (e quindi sempre disponibile, anche senza instanziarla) `System`. Essi sono `System.in` e `System.out`.

```
import java.io.*;

public class InToOut    {
//-------------------------------------------------
    public static void main(String args[]) {
        int charRead;
        try {
            while ((charRead = System.in.read()) != -1) {
                System.out.write(charRead);
                System.out.println("");//Ignorare questa istruzione (ma non toglierla)
            }
        } catch (IOException e) {
            //non facciamo nulla
        }
    }
}
```

Il programma si ferma immettendo un particolare carattere (***End Of File***, ***EOF***) che segnala fine del file:

- sotto UNIX si preme Ctrl-D;
- sotto Windows si preme Ctrl-Z;
- sotto MacOS si preme OK nella finestra di dialogo dello standard input.

1.2.3) Estensione della funzionalità degli Stream

Lavorare con singoli byte non è molto comodo. Per fornire funzionalità di I/O di livello più alto, in Java si usano estensioni degli stream dette ***stream di filtro***. Uno stream di filtro (che esiste sia per l'input che per l'output) è una estensione del corrispondente stream di base, si attacca ad uno stream esistente e fornisce ulteriori funzionalità.

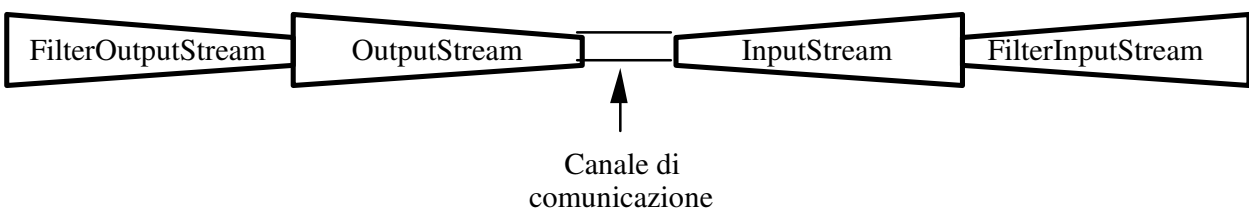

Figura 1-4: Stream di filtro

Aspetti importanti degli stream di filtro:

- sono classi derivate da `InputStream` e `OutputStream`, e quindi supportano tutti i loro metodi;
- si attaccano a un `InputStream` o `OutputStream` al quale passano di norma i metodi della superclasse: chiamare `write()` su uno stream di filtro significa chiamare `write()` sull'`OutputStream` attaccato;
- poiché sono anch'essi degli stream, è possibile attaccare vari stream di filtro in serie per ottenere potenti combinazioni.

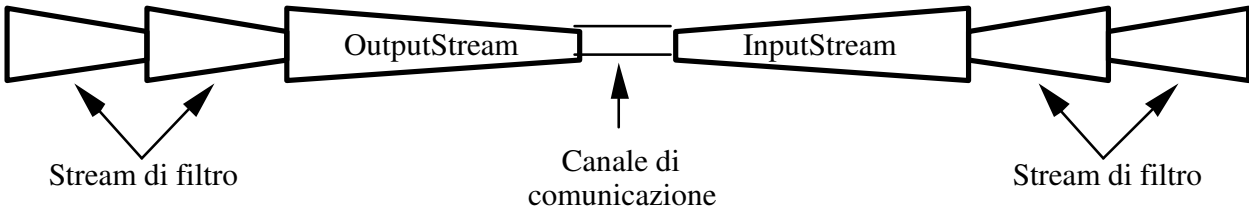

Figura 1-5: Concatenazione di stream di filtro

Il vantaggio principale di questa strategia, rispetto a estendere direttamente le funzionalità di uno stream di base, è che tali estensioni possono essere applicate a tutti i tipi di stream, in quanto sono sviluppate indipendentemente da essi.

1.2.3.1) Classe FilterInputStream

Estende `InputStream` ed è a sua volta la classe da cui derivano tutti gli stream di filtro per l'input. Si attacca a un `InputStream`.

Costruttore

```
protected FilterInputStream(InputStream in);
```

Crea uno stream di filtro attaccato a `in`.

Variabili

```
protected InputStream in;
```

Reference all'`InputStream` a cui è attaccato.

Metodi più importanti

Gli stessi di `InputStream`. Di fatto l'implementazione di default non fa altro che chiamare i corrispondenti metodi dell'`InputStream` attaccato.

1.2.3.2) Classe FilterOutputStream

Estende `OutputStream` ed è a sua volta la classe da cui derivano tutti gli stream di filtro per l'output. Si attacca a un `OutputStream`.

Costruttore

```
protected FilterOutputStream(OutputStream out);
```

Crea uno stream di filtro attaccato a `out`.

Variabili

```
protected OutputStream out;
```

Reference all'`OutputStream` a cui è attaccato.

Metodi più importanti

Gli stessi di `OutputStream`. Di fatto l'implementazione di default non fa altro che chiamare i corrispondenti metodi dell'`OutputStream` attaccato.

Vari stream di filtro predefiniti sono disponibili in Java. I più utili sono descritti nel seguito.

1.2.3.3) BufferedInputStream e BufferedOutputStream

Forniscono, al loro interno, meccanismi di buffering per rendere più efficienti le operazioni di I/O.

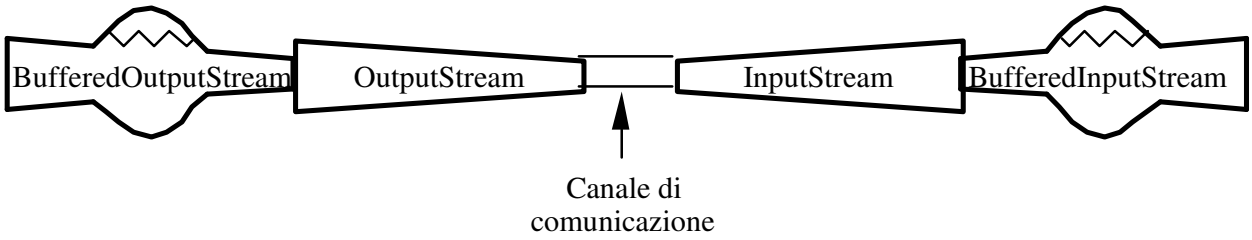

Figura 1-6: Stream di filtro per il buffering

Essi diminuiscono il numero di chiamate al sistema operativo (ad esempio nel caso di accessi a file) o di TPDU spediti (nel caso delle connessioni di rete).

Costruttori

```
public BufferedInputStream(InputStream in);
```

Crea un `BufferedInputStream` attaccato a `in` (con un buffer interno di 512 byte).

```
public BufferedInputStream(InputStream in, int size);
```

Secondo costruttore in cui si specifica la dimensione del buffer interno.

```
public BufferedOutputStream (OutputStream out);
```

Crea un `BufferedOutputStream` attaccato a `out` (con un buffer interno di 512 byte).

```
public BufferedOutputStream (OutputStream out, int size);
```

Secondo costruttore in cui si specifica la dimensione del buffer interno.

Metodi più importanti

Gli stessi degli stream da cui derivano. In più, `BufferedOutputStream` implementa il metodo `flush()`.

1.2.3.4) DataInputStream e DataOutputStream

Forniscono metodi per leggere e scrivere dati a un livello di astrazione più elevato (ad esempio interi, reali, stringhe, booleani, ecc.) su un canale orientato al byte.

I valori numerici vengono scritti in ***network byte order*** (il byte più significativo viene scritto per primo), uno standard universalmente accettato. In C, ciò si ottiene mediante l'uso di apposite funzioni:

- `htons()`: host to network short;
- `htonl()`: host to network long;
- `ntohs()`: network to host short;
- `ntohl()`: network to host long.

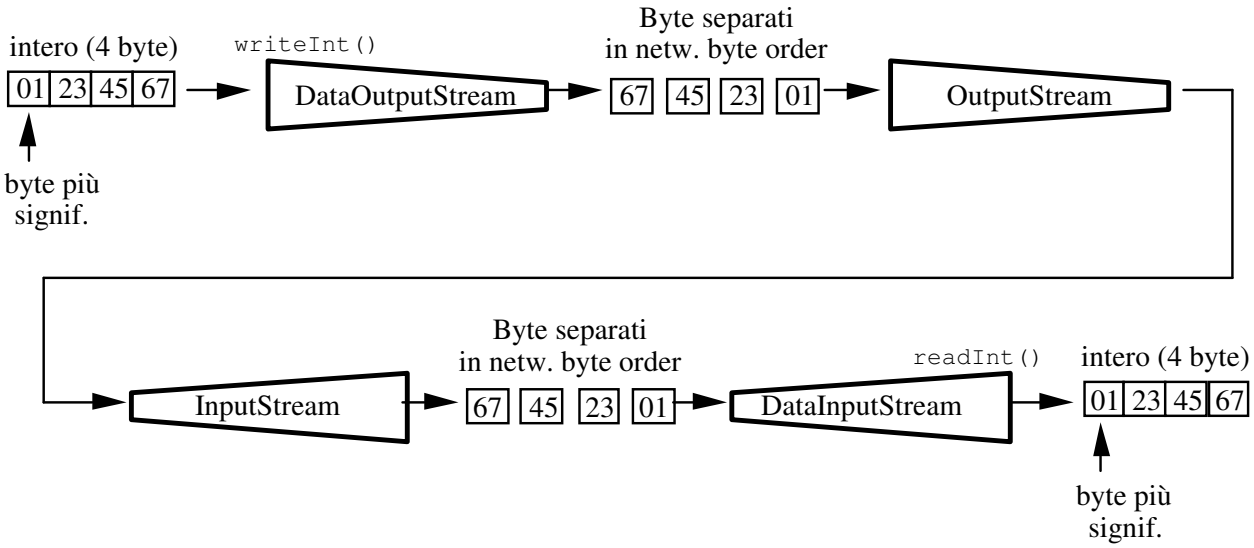

Figura 1-7: Uso di `DataInputStream` e `DataOutputStream`

Costruttori

```
public DataInputStream(InputStream in);
```

Crea un `DataInputStream` attaccato a `in`.

```
public DataOutputStream(OutputStream out);
```

Crea un `DataOutputStream` attaccato a `out`.

Metodi più importanti

```
public writeBoolean(boolean v) throws IOException;
public boolean readBoolean() throws IOException;
public writeByte(int v) throws IOException;
public byte readByte()throws IOException;
public writeShort(int v) throws IOException;
public short readShort()throws IOException;
public writeInt(int v) throws IOException;
```

```
public int readInt() throws IOException;
public writeLong(long v) throws IOException;
public long readLong()throws IOException;
public writeFloat(float v) throws IOException;
public float readFloat() throws IOException;
public writeDouble(double v) throws IOException;
public double readDouble() throws IOException;
public writeChar(int v) throws IOException;
public char readChar() throws IOException;
```

Letture e scritture dei tipi di dati primitivi.

```
public writeChars(String s) throws IOException;
```

Scrittura di una stringa.

```
public String readLine() throws IOException;
```

Lettura di una linea di testo, terminata con ***CR***, ***LF*** oppure ***CRLF***.

1.2.3.5) PrintStream

Fornisce metodi per scrivere, come sequenza di caratteri ASCII, una rappresentazione di tutti i tipi primitivi e degli oggetti che implementano il metodo `toString()`, che viene in tal caso usato.

Costruttori

```
public PrintStream(OutputStream out);
```

Crea un `PrintStream` attaccato a `out`.

```
public PrintStream(OutputStream out, boolean autoflush);
```

Se `autoflush` è impostato a `true`, lo stream effettua un `flush()` ogni volta che incontra la fine di una linea di testo (CR, LF oppure CR+LF).

Metodi più importanti

```
public void print(...) throws IOException;
```

Il parametro può essere ogni tipo elementare, un array di caratteri, una stringa, un Object.

```
public void println(...) throws IOException;
```

Il parametro può essere ogni tipo elementare, un array di caratteri, una stringa, un Object.

`println()` dopo la scrittura va a linea nuova (appende CR, LF oppure CR+LF).

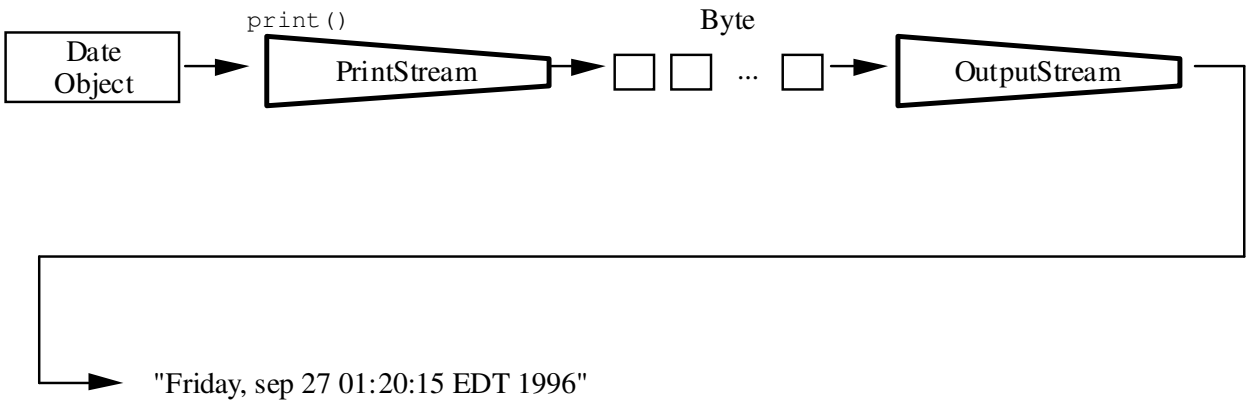

Figura 1-8: Uso di `PrintStream`

Esempi di uso di FilterStream

1.2.3.6) Esempi di uso degli stream di filtro

Vediamo ora alcuni usi tipici degli stream di filtro. In generale ci sono due modi di crearli, a seconda che si vogliano mantenere o no dei riferimenti agli stream a cui vengono attaccati.

Efficienza

Si usano gli stream `BufferedInputStream` e `BufferedOutputStream`, che offrono la funzione di buffering. Supponiamo di lavorare con un `FileOutputStream` (che vedremo nel seguito).

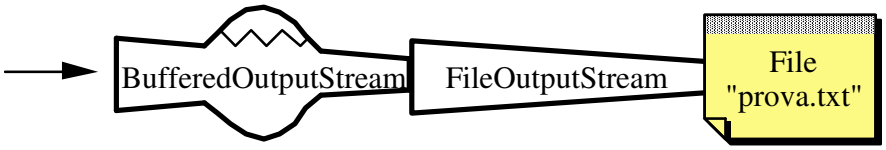

Figura 1-9: Buffering nell'accesso a un file

La prima possibilità è di mantenere in una apposita variabile anche il riferimento allo stream cui si attacca il filtro (cioè al `FileOutputStream`):

```
FileOutputStream fileStream;

BufferedOutputStream bufferedStream;

fileStream = new FileOutputStream("prova.txt");

bufferedStream = new BufferedOutputStream(fileStream);
```

La seconda possibilità è creare il `FileOutputStream` direttamente dentro il costruttore dello stream di filtro :

```
BufferedOutputStream bufferedStream;
bufferedStream = new BufferedOutputStream(new FileOutputStream("prova.txt"));
```

Si noti che in questo caso non vi è modo, nel resto del codice, di riferirsi esplicitamente al `FileOutputStream`.

Lettura e scrittura ASCII
Si usano `DataInputStream` (chiamando ad esempio `readLine()`) e `PrintStream` (chiamando ad esempio `println(...)`).

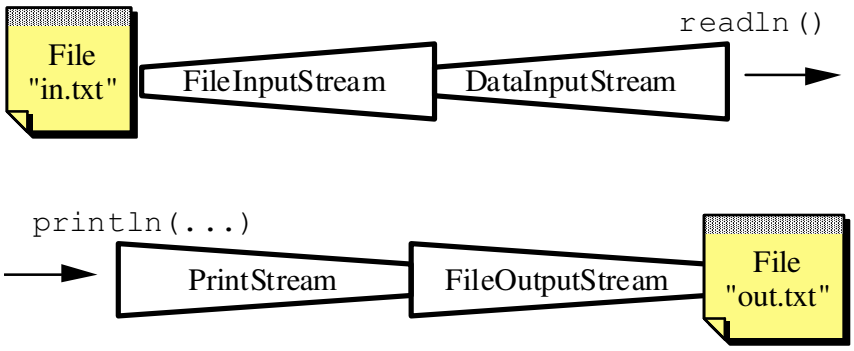

Figura 1-10: Lettura e scrittura ASCII su file

```
DataInputStream asciiIn;

PrintStream asciiOut;

asciiIn = new DataInputStream(new FileInputStream("in.txt"));

asciiOut = new PrintStream(new FileOutputStream("out.txt"));
```

Efficienza e scrittura caratteri ASCII
Usiamo un `PrintStream` attaccato a un `BufferedOutputStream` attaccato a sua volta a un `FileOutputStream`.

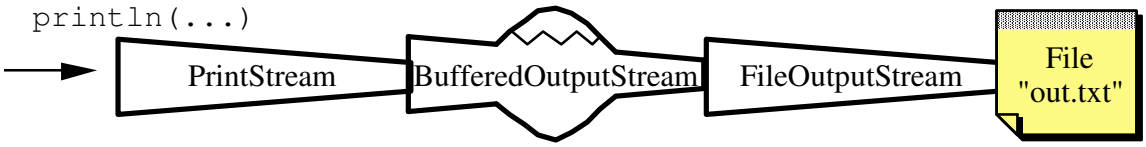

Figura 1-11: Scrittura ASCII, bufferizzata, su file

```
PrintStream asciiOut;
asciiOut = new PrintStream(new BufferedOutputStream(
                           new FileOutputStream("out.txt")));
```

Analogo discorso per la lettura bufferizzata:

```
DataInputStream asciiIn;
asciiIn = new DataInputStream(new BufferedInputStream(
                              new FileInputStream("in.txt")));
```

1.2.4) Tipi base di Stream

Le classi `InputStream` e `OutputStream` sono astratte, e da esse si derivano quelle che rappresentano le connessioni a reali canali di comunicazione di vario tipo. Ne esistono molte; noi parleremo di:

- stream per l'accesso a file;
- stream per l'accesso ad array di byte esistenti in memoria centrale;
- stream per l'accesso a connessioni di rete.

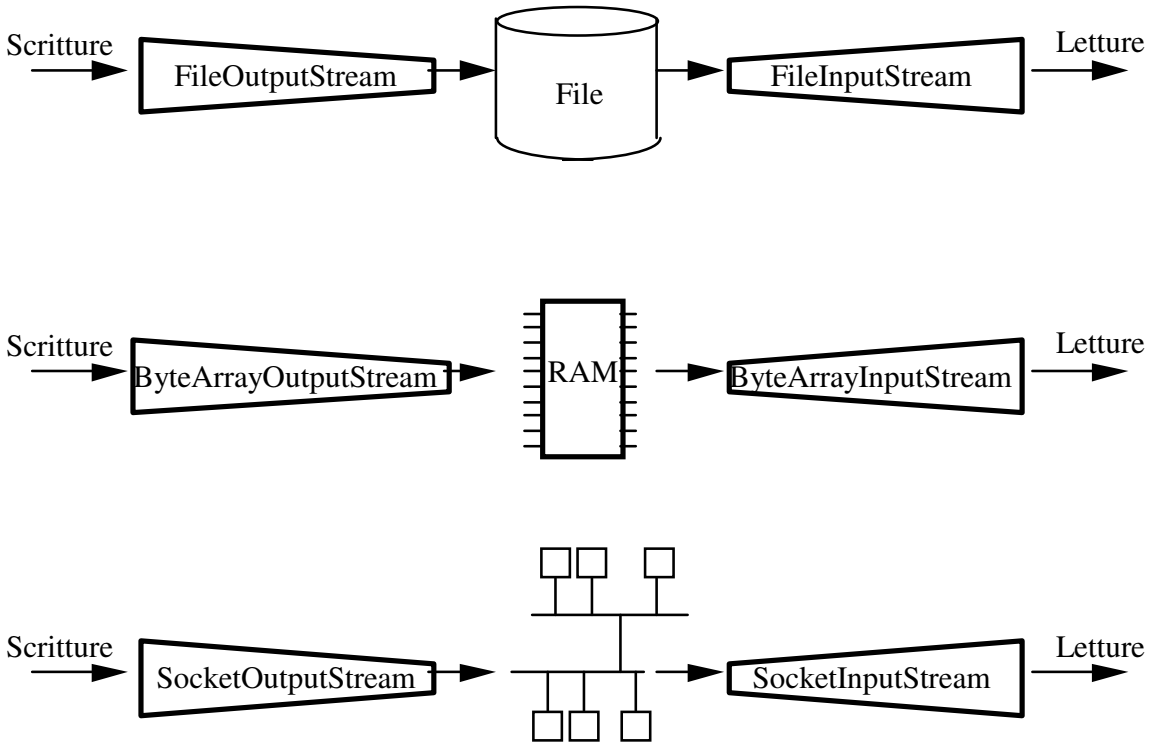

Figura 1-12: Tipi base di stream

Si noti che `SocketInputStream` e `SocketOutputStream` sono classi private, per cui una connessione di rete viene gestita per mezzo di `InputStream` e `OutputStream`.

1.2.5) Stream per accesso a file

L'accesso a un file si ottiene con due tipi di Stream:

- `FileInputStream` per l'accesso in lettura;
- `FileOutputStream` per l'accesso in scrittura.

La classe `FileInputStream` permette di leggere sequenzialmente da un file (che deve già esistere).

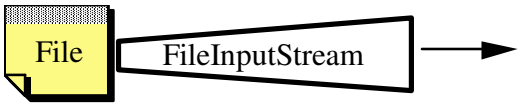

Figura 1-13: `FileInputStream`

Costruttori

```
public FileInputStream(String fileName) throws IOException;
public FileInputStream(File fileName) throws IOException;
```

Sono i più importanti. La classe `File` rappresenta un nome di file in modo indipendente dalla piattaforma, ed è di solito utilizzata in congiunzione con un `FileDialog` che permette all'utente la scelta di un file. Vedremo un esempio più avanti. Se si usa una stringa, essa può essere:

- Nome del file (il direttorio di lavoro è implicito);
- Path completo più nome del file.

Metodi più importanti

Tutti quelli di `InputStream`, più uno (`getFD()`) che non vedremo.

La classe `FileOutputStream` permette di scrivere sequenzialmente su un file, che viene creato quando si istanzia lo stream. Se esiste già un file con lo stesso nome, esso viene distrutto.

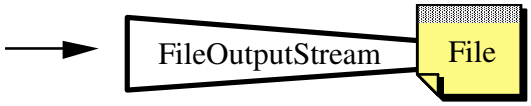

Figura 1-14: `FileOutputStream`

Costruttori

```
public FileOutputStream(String fileName) throws IOException;
public FileOutputStream (File fileName) throws IOException;
```

Sono i più importanti. Per la classe `File` vale quanto detto sopra.

Metodi più importanti

Tutti quelli di `OutputStream`, più uno (`getFD()`) che non vedremo.

Si noti che questi stream, oltre che eccezioni di I/O, possono generare eccezioni di tipo `SecurityException` (ad esempio se un applet cerca di aprire uno stream verso il file system locale).

Esempio 3

Applicazione che mostra in una `TextArea` il contenuto di un file di testo, e che salva il contenuto di tale `TextArea` in un file.

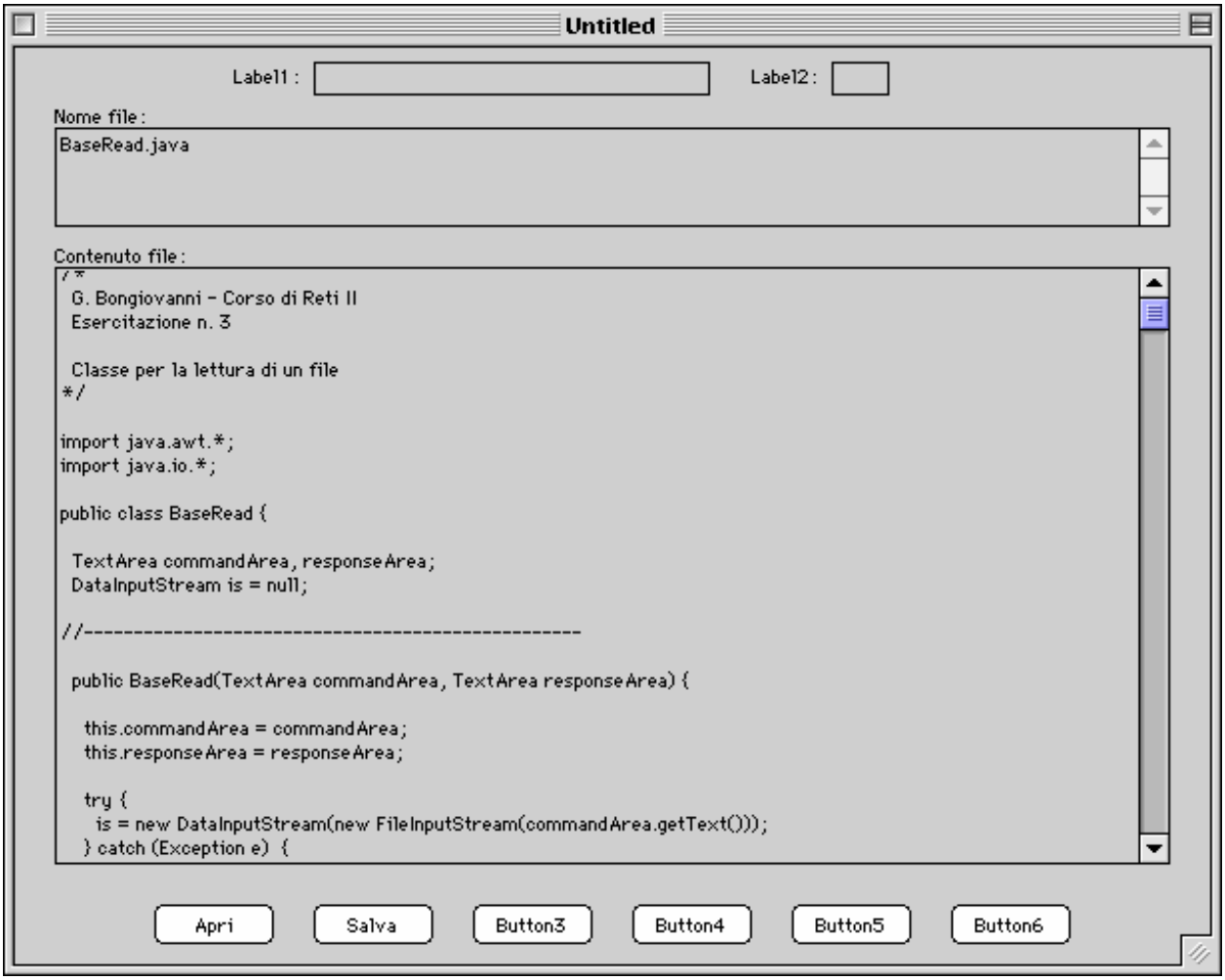

Figura 1-15: Interfaccia utente dell'esempio 3

Il codice è costituito da tre classi. La prima, `BaseAppE3`, è basata su quella dell'esempio 1 ed istanzia a comando una delle altre due quando si preme un bottone. Se ne mostrano qui solo i frammenti di codice rilevanti.

```
import java.awt.*;

public class BaseAppE3 extends Frame    {
...ecc.
    //-----------------------------------------------
    void button1_Clicked(Event event) {
        baseRead = new BaseRead(textArea1, textArea2);
        baseRead.readFile();
    }
//-----------------------------------------------
    void button2_Clicked(Event event) {
        baseWrite = new BaseWrite(textArea1, textArea2);
        baseWrite.writeFile();
    }
//-----------------------------------------------
```

Le altre due si occupano della lettura e scrittura sul file.

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

public class BaseRead {
    TextArea commandArea, responseArea;
    DataInputStream is = null;
//-----------------------------------------------
    public BaseRead(TextArea commandArea, TextArea responseArea) {
        this.commandArea = commandArea;
        this.responseArea = responseArea;
        try {
            is=new DataInputStream(new FileInputStream(commandArea.getText()));
        } catch (Exception e)   {
            responseArea.appendText("Exception" + "\n");
        }
    }
//-----------------------------------------------
    public void readFile() {
        String inputLine;
        try {
            while ((inputLine = is.readLine()) != null) {
                responseArea.appendText(inputLine + "\n");
            }
        } catch (IOException e) {
            responseArea.appendText("IO Exception" + "\n");
        } finally {
            try {
                is.close();
            } catch (IOException e) {
                responseArea.appendText("IO Exception" + "\n");
            }
        }
    }
}
```

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

public class BaseWrite {
    TextArea commandArea, responseArea;
    PrintStream os = null;
//-----------------------------------------------
    public BaseWrite(TextArea commandArea, TextArea responseArea) {

        this.commandArea = commandArea;
        this.responseArea = responseArea;

        try {
            os = new PrintStream(new FileOutputStream(commandArea.getText()));
        } catch (Exception e)   {
            responseArea.appendText("Exception" + "\n");
        }
    }
//-----------------------------------------------
    public void writeFile() {
        os.print(responseArea.getText());
        os.close();
    }
}
```

Se si vuole selezionare il file da leggere o da creare in modo interattivo, si usa la classe `FileDialog`, che serve a entrambi gli scopi.
Il codice per aprire un file in lettura è tipicamente:

```
...
FileDialog openDialog = new FileDialog(null, "Titolo", FileDialog.LOAD);
openDialog.show();
if (openDialog.getFile() != null) {
    is = new DataInputStream(
         new FileInputStream(
         newFile(openDialog.getDirectory(), openDialog.getFile())));
}
...
```

E quello per creare un file in scrittura:

```
...
FileDialog saveDialog = new FileDialog(null, "Titolo", FileDialog.SAVE);
saveDialog.show();
if (saveDialog.getFile() != null) {
    os = new PrintStream(
         new FileOutputStream(
         new File(saveDialog.getDirectory(), saveDialog.getFile()));
}
...
```

1.2.6) Stream per accesso alla memoria

Esistono due tipi di stream che consentono di leggere e scrivere da/su buffer di memoria:

- `ByteArrayInputStream`
- `ByteArrayOutputStream`

Il `ByteArrayInputStream` permette di leggere sequenzialmente da un buffer (ossia da un array di byte) di memoria.

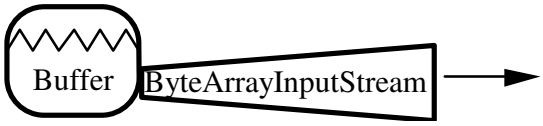

Figura 1-16: `ByteArrayInputStream`

Costruttori

A noi ne serve solo uno:

```
public ByteArrayInputStream (byte[] buf);
```

Crea un `ByteArrayInputStream`, attaccato all'array `buf`, dal quale vengono poi letti i dati.

Metodi più importanti

Quelli di `InputStream`. Da notare che:
- un `EOF` si incontra quando si raggiunge la fine dell'array;
- il metodo `reset()` ripristina lo stream, per cui le successive letture ripartono dall'inizio dell'array.

Il `ByteArrayOutputStream` permette di scrivere in un buffer (array di byte) in memoria. Il buffer viene allocato automaticamente e si espande secondo le necessità, sempre automaticamente.

Figura 1-17: `ByteArrayOutputStream`

Costruttori

I più utilizzati sono:

```
public ByteArrayOutputStream();
```

Crea un `ByteArrayOutputStream` con una dimensione iniziale del buffer di 32 byte. Il buffer comunque cresce automaticamente secondo necessità.

```
public ByteArrayOutputStream (int size);
```

Crea un `ByteArrayOutputStream` con una dimensione iniziale di `size` byte. E' utile quando si voglia massimizzare l'efficienza e si conosca in anticipo la taglia dei dati da scrivere.

Metodi più importanti

Oltre a quelli di `OutputStream` ve ne sono alcuni per accedere al buffer interno.

```
public void reset();
```

Ricomincia la scrittura dall'inizio, di fatto azzerando il buffer.

```
public int size();
```

Restituisce il numero di byte contenuti (cioè scritti dopo l'ultima `reset()`) nel buffer.

```
public byte[] toByteArray();
```

Restituisce una copia dei byte contenuti nel buffer, che non viene resettato.

```
public String toString();
```

Restituisce una stringa che è una copia dei byte contenuti. L'array non viene resettato.

```
public writeTo(OutputStream out) throws IOException;
```

Scrive il contenuto del buffer sullo stream `out`. Il buffer non viene resettato. E' più efficiente che chiamare `toByteArray()` e scrivere quest'ultimo su `out`.

1.2.7) Stream per accesso a connessioni di rete

Il dialogo di un'applicazione Java con una peer entity attraverso la rete può avvenire sia appoggiandosi a TCP che a UDP.

Poiché la modalità connessa è molto più versatile ed interessante per i nostri scopi, ci occuperemo solo di essa.

Come noto, la vita di una connessione TCP si articola in tre fasi:
1. apertura della connessione;
2. dialogo per mezzo della connessione;
3. chiusura della connessione;

Anche un'applicazione Java segue lo stesso percorso:
1. apertura della connessione. Esistono per questo due classi:
   - `Socket`: per aprire una connessione con un server in ascolto;
   - `ServerSocket`: per mettersi in ascolto di richieste di connessione;
2. dialogo per mezzo della connessione. Con opportune istruzioni, dal `Socket` precedentemente creato si ottengono:
   - `InputStream` per ricevere i dati dalla peer entity;
   - `OutputStream` per inviare i dati alla peer entity;
3. chiusura della connessione. Con opportune istruzioni, si chiudono:
   - l'`InputStream`;
   - l'`OutputStream`;
   - il `Socket` (per ultimo, preferibilmente).

1.2.7.1) Classe Socket

Questa classe rappresenta in Java l'estremità locale di una connessione TCP. La creazione di un oggetto `Socket`, con opportuni parametri, stabilisce automaticamente una connessione TCP con l'host remoto voluto.

Se la cosa non riesce, viene generata una eccezione (alcuni dei motivi possibili sono: host irraggiungibile, host inesistente, nessun processo in ascolto sul server).

Una volta che l'oggetto `Socket` è creato, da esso si possono ottenere i due stream (di input e output) necessari per comunicare.

Costruttore

```
public Socket(String host, int port) throws IOException;
```

A noi basta questo (ce ne sono due). `host` è un nome DNS o un indirizzo IP in ***dotted decimal notation***, e `port` è il numero di port sul quale deve esserci un processo server in ascolto.

Ad esempio:

```
...
mySocket = new Socket("cesare.dsi.uniroma1.it", 80);
mySocket = new Socket("151.100.17.25", 80);
...
```

Metodi più importanti

```
public InputStream getInputStream() throws IOException;
```

Restituisce un `InputStream` per leggere i dati inviati dalla peer entity.

```
public OutputStream getOutputStream() throws IOException;
```

Restituisce un `OutputStream` per inviare i dati inviati dalla peer entity.

```
public void close() throws IOException;
```

Chiude il `Socket` (e quindi questa estremità della connessione TCP).

```
public int getPort() throws IOException;
```

Restituisce il numero di port all'altra estremità della connessione.

```
public int getLocalPort() throws IOException;
```

Restituisce il numero di port di questa estremità della connessione.

Esempio 4

Applicazione che apre una connessione via Socket con un host e su un port scelti per mezzo dei due rispettivi campi. Va usata con ***protocolli ASCII***, cioè protocolli che prevedono lo scambio di sequenze di caratteri ASCII.

L'applicazione permette di:

- stabilire la connessione con un server mediante il bottone "Connect";
- inviare un comando col bottone "Send";
- leggere ***una singola linea di testo*** della risposta col bottone "Receive";
- chiudere la connessione col bottone "Close".

Si noti che, a seconda del numero di port specificato, si può dialogare con server differenti. Ad esempio, usando il port 21 ci si connette con un server FTP, mentre con il port 80 (come nell'esempio di figura 2-16) ci si collega ad un server HTTP.

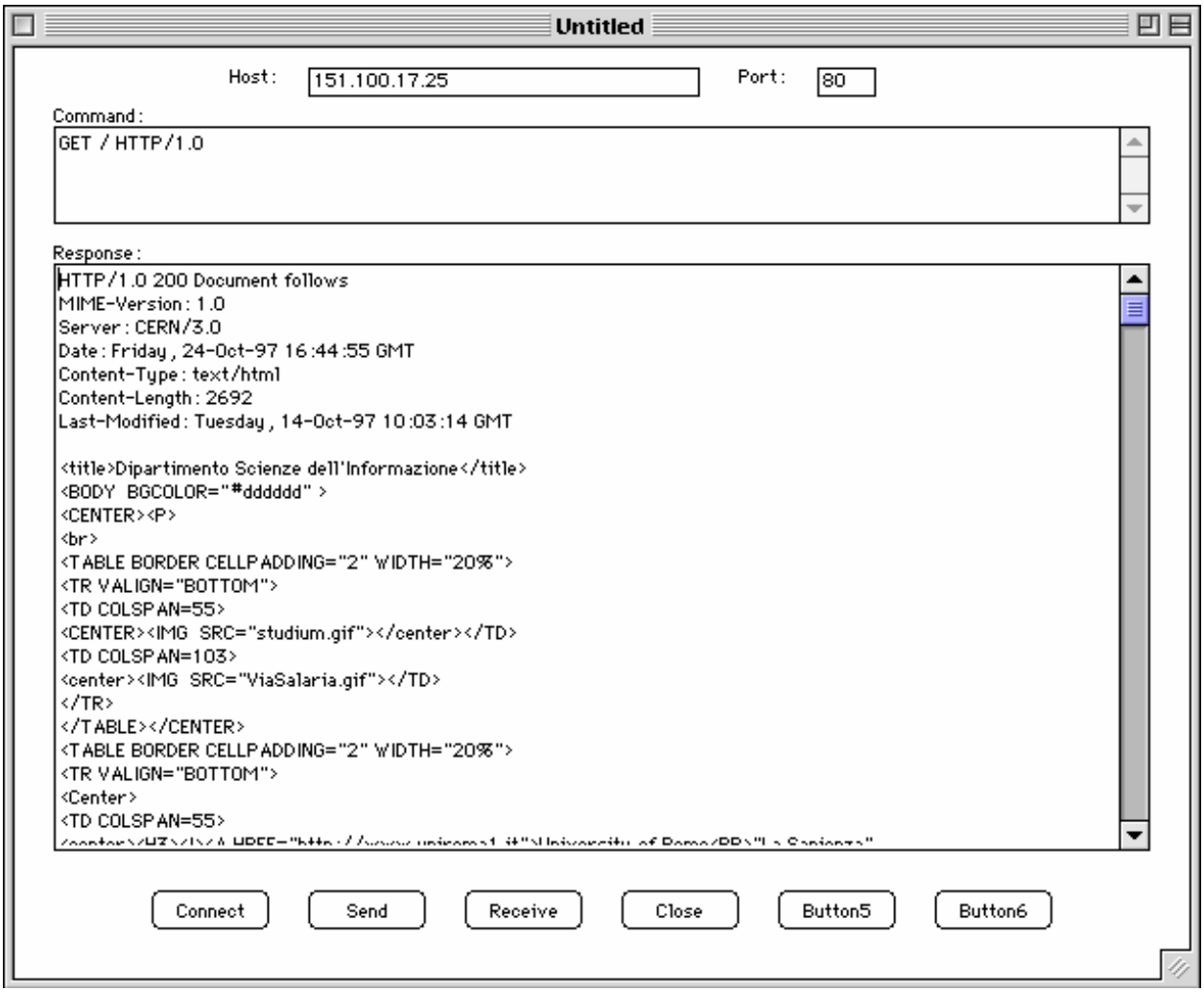

Figura 1-18: Interfaccia utente dell'esempio 4

Il codice è costituito da due classi. La prima, `BaseAppE4`, è basata su quella dell'esempio 1 e provvede ad istanziare la seconda che ha il compito di attivare la connessione e di gestire la comunicazione. Se ne mostrano qui solo i frammenti di codice rilevanti.

```
import java.awt.*;

public class BaseAppE4 extends Frame    {
...ecc.
//------------------------------------------------
    void button1_Clicked(Event event) {
        baseConn = new BaseConn(textField1.getText(),
                                textField2.getText(),
                                textArea1, textArea2);
}
//------------------------------------------------
    void button2_Clicked(Event event) {
        baseConn.send();
}
//------------------------------------------------
void button3_Clicked(Event event) {
        baseConn.receive();
}
//------------------------------------------------
    void button4_Clicked(Event event) {
        baseConn.close();
}
//------------------------------------------------
```

La seconda si occupa della comunicazione.

```
import java.awt.*;
import java.lang.*;
import java.io.*;
import java.net.*;

public class BaseConn {
    TextArea commandArea, responseArea;
    Socket socket = null;
    PrintStream os = null;
    DataInputStream is = null;
//------------------------------------------------
    public BaseConn(String host, String port,
                    TextArea commandArea, TextArea responseArea) {
        this.commandArea = commandArea;
        this.responseArea = responseArea;
        try {
            socket = new Socket(host, Integer.parseInt(port));
            os = new PrintStream(socket.getOutputStream());
            is = new DataInputStream(socket.getInputStream());
            responseArea.appendText("***Connection established" + "\n");
        } catch (Exception e)   {
            responseArea.appendText("Exception" + "\n");
        }

    }
//------------------------------------------------
    public void send() {
        os.println(commandArea.getText());
    }
//------------------------------------------------
    public void receive() {
        String inputLine;
        try {
            inputLine = is.readLine();
            responseArea.appendText(inputLine + "\n");
```

```
        } catch (IOException e) {
            responseArea.appendText("IO Exception" + "\n");
        }
    }
//-------------------------------------------------
    public void close() {
        try {
            is.close();
            os.close();
            socket.close();
            responseArea.appendText("***Connection closed" + "\n");
        } catch (IOException e) {
            responseArea.appendText("IO Exception" + "\n");
        }
    }
}
```

1.2.7.2) Classe ServerSocket

Questa classe costituisce il meccanismo con cui un programma Java agisce da server, e cioè accetta richieste di connessioni provenienti dalla rete.

La creazione di un `ServerSocket`, seguita da una opportuna istruzione di "ascolto", fa si che l'applicazione si metta in attesa di una richiesta di connessione.

Quando essa arriva, il `ServerSocket` crea automaticamente un `Socket` che rappresenta l'estremità locale della connessione appena stabilita. Da tale `Socket` si derivano gli stream che permettono la comunicazione.

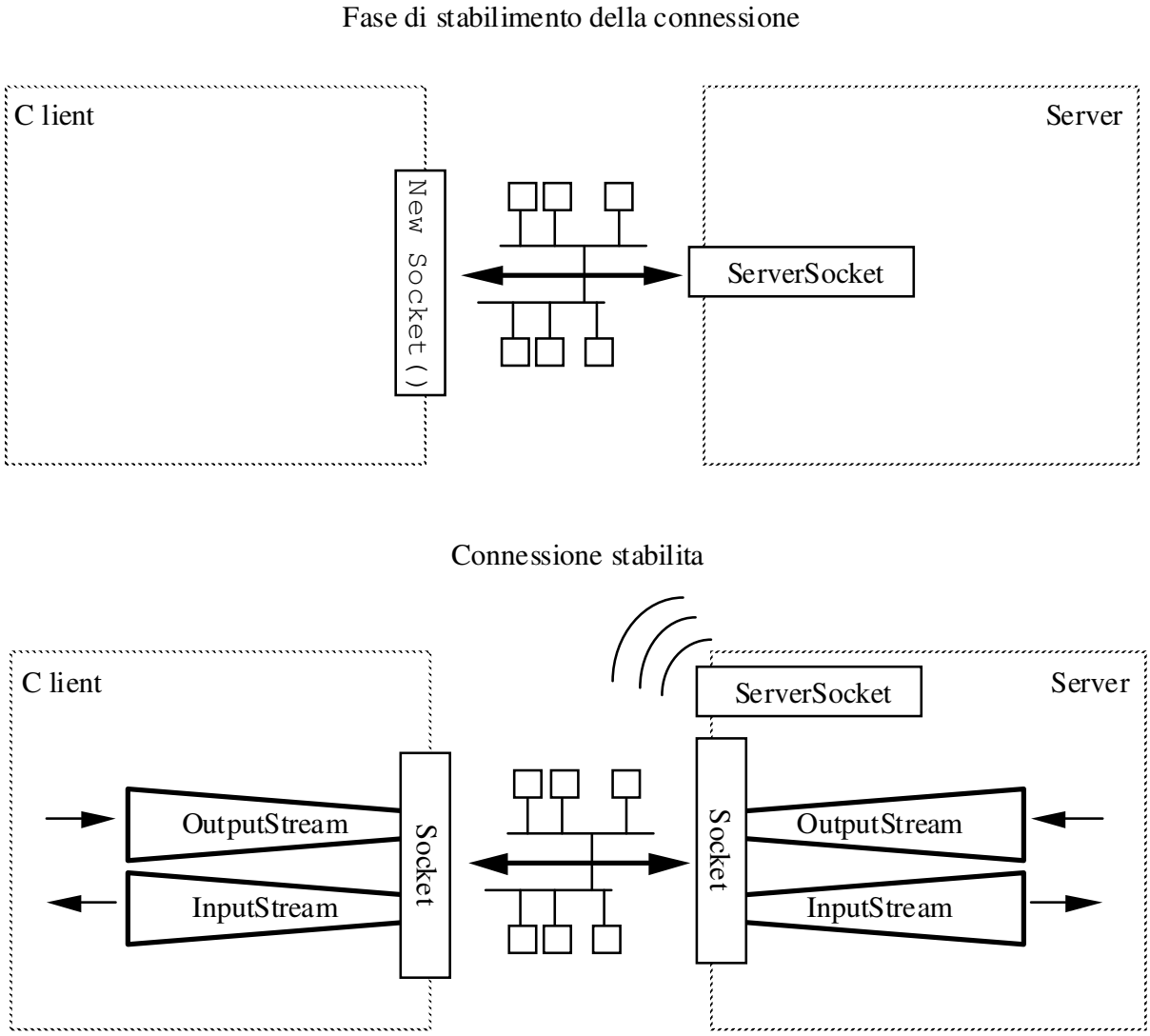

Figura 1-19: Connessione a un `ServerSocket`

Costruttori

```
public ServerSocket(int port) throws IOException;
```

A noi basta questo (ce ne sono due). `port` è il numero del port sul quale il `ServerSocket` si mette in ascolto.

Metodi più importanti

```
public Socket accept() throws IOException;
```

Questo metodo blocca il chiamante finché qualcuno non cerca di connettersi. Quando questo succede, il metodo ritorna restituendo un `Socket` che rappresenta l'estremità locale della connessione.

```
public void close() throws IOException;
```

Chiude il `ServerSocket` (e quindi non ci sarà più nessuno in ascolto su quel port).

```
public int getLocalPort() throws IOException;
```

Restituisce il numero di port su cui il `ServerSocket` è in ascolto.

Esempio 5

Semplice Server che:

- accetta una sola connessione sul port 5000;
- restituisce al Client tutto ciò che riceve.

Il codice è il seguente.

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

public class SimpleServer   {
//-------------------------------------------------
    public static void main(String args[]) {
        ServerSocket server;
        Socket client;
        String inputLine;
        DataInputStream is = null;
        PrintStream os = null;
        try {
            server = new ServerSocket(5000);
            System.out.println("Accepting one connection...");
            client = server.accept();
            server.close();
            System.out.println("Connection from one client accepted.");
            is = new DataInputStream(client.getInputStream());
            os = new PrintStream(client.getOutputStream());
            os.println("From SimpleServer: Welcome!");
            while ((inputLine = is.readLine()) != null) {
                System.out.println("Received: " + inputLine);
                os.println("From SimpleServer: " + inputLine);
            }
            is.close();
            os.close();
            client.close();
            System.out.println("Connection closed.");
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

Sia il client che il server visti precedentemente hanno pesanti limitazioni, derivanti dal fatto che essi consistono di un unico flusso di elaborazione:

- client: può leggere una sola linea di testo ad ogni pressione del bottone "Receive". Infatti, se si inserisse un ciclo infinito di lettura, esso bloccherebbe l'esecuzione (dopo l'ultima linea arrivata dal server) in attesa di ulteriore input che non arriverà mai, dato che è impossibile inviare un altro comando;
- server: può gestire una sola connessione. Infatti, un singolo flusso di elaborazione non può, in maniera semplice, gestire una connessione e rimanere nel contempo anche in ascolto di nuove richieste.

Per superare queste limitazioni la soluzione più potente e versatile è ricorrere al ***multithreading***, cioè alla gestione di multipli flussi di elaborazione all'interno dell'applicazione.

1.3) Multithreading

Java supporta il multithreading in modo nativo, a livello di linguaggio. Ciò rende la programmazione di multipli thread molto più semplice che dovendo usare librerie apposite, come è il caso di altri linguaggi.

L'ambiente Java fornisce la classe `Thread` per gestire i thread di esecuzione. Ogni oggetto istanziato da tale classe (o da una sua derivata) costituisce un flusso separato di esecuzione (ossia un thread di esecuzione).

Si noti in proposito che il metodo `main()` di un qualunque oggetto attiva un thread, che termina con la terminazione del `main()` stesso.

1.3.1) Classe Thread

E' la rappresentazione runtime di un thread di esecuzione.

Costruttori

Ce ne sono vari, i più usati sono i tre elencati sotto. Il terzo serve nel caso si faccia uso dell'***interfaccia*** `Runnable` (che vedremo più avanti).

```
public Thread();
public Thread(String name);
public Thread(Runnable target);
```

Metodi più importanti

Ci sono vari metodi statici (che vengono chiamati sul thread corrente), i più utili dei quali sono:

```
public static void sleep(long millis) throws InterruptedException;
```

Questo metodo mette a dormire il thread corrente per `millis` millisecondi. In questo periodo, altri thread possono avanzare nell'esecuzione. Se in questo tempo qualcun altro chiama il suo metodo `interrupt()`, il thread viene risvegliato da una `InterruptedException`.

```
public static void yield() throws InterruptedException;
```

Questo metodo fa si che il thread corrente ceda la Cpu ad altri thread in attesa. Poiché l'ambiente Java non può garantire la ***preemption*** (essa dipende dal sistema operativo) è consigliabile usarlo, quando un thread deve effettuare lunghe computazioni, a intervalli regolari.

I metodi di istanza più utili (che possono essere chiamati su qualunque thread) sono:

```
public synchronized void start() throws IllegalThreadStateException;
```

E' il metodo che si deve chiamare per far partire un thread, una volta creato; è un errore chiamarlo su un thread già avviato.

```
public void run();
```

E' il metodo che l'ambiente runtime chiama quando un thread viene avviato con il metodo `start()`. costituisce il corpo eseguibile del thread, e determina il suo comportamento. Quando esso finisce, il thread termina. E' dunque il metodo che ogni classe derivata da `Thread` deve ridefinire.

```
public void stop();
```

Termina il thread.

```
public void suspend();
```

Sospende il thread; altri possono eseguire.

```
public void resume();
```

Rende il thread nuovamente eseguibile, cioè ***ready*** (Nota: questo non significa che diventi anche ***running***, in quanto ciò dipende anche da altri fattori).

In più, ci sono altri metodi per gestire la priorità, avere notizie sullo stato del thread, ecc.

Esempio 6

Programma che consiste di due thread, i quali scrivono un messaggio ciascuno sullo standard output con cadenze differenti.

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

public class PingPong extends Thread {
    String word;
    int delay;
//---------------------------------------------------
    public PingPong (String whatToSay, int delayTime) {
        word = whatToSay;
        delay = delayTime;
    }
//---------------------------------------------------
    public void run () {
        try {
            while (true) {
                System.out.println(word);
                sleep(delay);
            }
        } catch (InterruptedException e) {
            return;
        }
    }
//---------------------------------------------------
    public static void main(String args[]) {
        new PingPong("ping", 250).start();  //1/4 sec.
        new PingPong("PONG", 100).start();  //1/10 sec.
    }
}
```

1.3.2) Interfaccia Runnable

Vi sono alcune situazioni nelle quali il ricorso a una estensione della classe `Thread` non è adatto agli scopi, poiché:

- l'oggetto per il quale si vuole avviare un thread è già estensione di una qualche altra classe (e quindi non può estendere anche `Thread`);
- si vogliono avviare molteplici thread dentro la stessa istanza di un oggetto, cosa che è impossibile estendendo la classe `Thread`.

La risposta a questo problema è data dalla interfaccia `Runnable`, che include un solo metodo:

```
public abstract void run();
```

dall'ovvio significato: in esso si specifica il comportamento del thread da creare, come visto prima.

Basta quindi definire una classe che implementa tale interfaccia:

```
public MyClass implements Runnable {
...
}
```

ed includervi un metodo `run()`:

```
public abstract void run(){
...
}
```

per avere la possibilità di lanciare thread multipli all'interno di un oggetto di tale classe.

Tali thread vengono creati col terzo dei costruttori visti per la classe `Thread` e successivamente vengono attivati invocandone il metodo `start()`, che a sua volta causa l'avvio del metodo `run()` dell'oggetto che è passato come parametro nel costruttore:

```
...
new Thread(theRunnableObject).start();
...
```

Esempio 7

Funzionalità simile all'esempio precedente ma ottenuta con l'interfaccia `Runnable`.

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

public class RPingPong /*extends AnotherClass*/ implements Runnable {
    String word;
    int delay;
//-------------------------------------------------
    public RPingPong (String whatToSay, int delayTime) {
        word = whatToSay;
        delay = delayTime;
    }
//-------------------------------------------------
    public void run () {
        try {
            while (true) {
                System.out.println(word);
                Thread.sleep(delay);
            }
        } catch (InterruptedException e) {
            return;
        }
```

```
    }
//---------------------------------------------
    public static void main(String args[]) {
        Runnable ping = new RPingPong("ping", 100); //1/10 sec.
        Runnable PONG = new RPingPong("PONG", 100); //1/10 sec.
        new Thread(ping).start();
        new Thread(ping).start();
        new Thread(PONG).start();
        new Thread(PONG).start();
        new Thread(PONG).start();
        new Thread(PONG).start();
        new Thread(PONG).start();
    }
}
```

Esempio 8

Applicazione che apre una connessione di rete, come nell'esempio 4. La differenza è che un thread separato rimane in ascolto delle risposte del server: questo permette di ricevere risposte costituite da più linee di testo senza alcun problema.

Il codice è costituito da due classi. La prima, `BaseAppE8`, è basata su quella dell'esempio 4 e provvede ad istanziare la seconda che ha il compito di attivare la connessione e di gestire la comunicazione. Se ne mostrano qui solo i frammenti di codice rilevanti, sottolineando il fatto che non c'è più bisogno del bottone "Receive", visto che un thread separato rimane in costante ascolto delle risposte.

```
import java.awt.*;

public class BaseAppE8 extends Frame    {
...ecc.
//---------------------------------------------
    void button1_Clicked(Event event) {
        baseTConn = new BaseTConn(textField1.getText(),
                                  textField2.getText(),
                                  textArea1, textArea2);
    }
//---------------------------------------------
    void button2_Clicked(Event event) {
        baseTConn.send();
    }
//---------------------------------------------
    void button3_Clicked(Event event) {
        baseTConn.close();
    }
//---------------------------------------------
```

La seconda classe si occupa della comunicazione, ed inolte attiva tramite il metodo `run()` un thread separato costituito da un ciclo infinito in cui si ricevono le risposte del server e si provvede a mostrarle sullo schermo.

```
import java.awt.*;
import java.lang.*;
import java.io.*;
import java.net.*;

public class BaseTConn implements Runnable {
    TextArea commandArea, responseArea;
    Socket socket = null;
    PrintStream os = null;
    DataInputStream is = null;
//-------------------------------------------------
    public BaseTConn(String host, String port,
                     TextArea commandArea, TextArea responseArea) {
        this.commandArea = commandArea;
        this.responseArea = responseArea;
        try {
            socket = new Socket(host, Integer.parseInt(port));
            os = new PrintStream(socket.getOutputStream());
            is = new DataInputStream(socket.getInputStream());
            responseArea.appendText("***Connection established" + "\n");
            new Thread(this).start();
        } catch (Exception e)   {
            responseArea.appendText("Exception" + "\n");
        }
    }
//-------------------------------------------------
    public void run() {
        String inputLine;
        try {
            while ((inputLine = is.readLine()) != null) {
                responseArea.appendText(inputLine + "\n");
            }
        } catch (IOException e) {
            responseArea.appendText("IO Exception" + "\n");
        }
    }
//-------------------------------------------------
    public void send() {
            os.println(commandArea.getText());
    }
//-------------------------------------------------
    public void close() {
        try {
            is.close();
            os.close();
            socket.close();
            responseArea.appendText("***Connection closed" + "\n");
        } catch (IOException e) {
            responseArea.appendText("IO Exception" + "\n");
        }
    }
}
```

1.4 Sincronizzazione

Come in tutti i regimi di concorrenza, anche in Java possono sorgere problemi di consistenza dei dati condivisi se i thread sono ***cooperanti***.

I dati condivisi fra thread differenti possono essere:
- variabili statiche di una classe che estende `Thread`, che sono quindi condivise da tutte le sue istanze;
- variabili di istanza di un oggetto `Runnable`, che sono condivise da tutti i thread attivati dentro tale oggetto.

Per risolvere i problemi legati alla concorrenza, in Java è stata incorporata nella classe `Object` (e quindi in tutte le altre, che derivano da essa) e nelle sue istanze la capacità potenziale di funzionare come un ***monitor***.

1.4.1) Metodi sincronizzati

In particolare, tale funzionalità si attiva ricorrendo ai ***metodi sincronizzati***, definiti come:

```
public void synchronized aMethod(...) {
...
}
```

Quando un thread chiama un metodo sincronizzato di un oggetto, acquisisce un ***lucchetto*** su quell'oggetto. Nessun altro thread può chiamare un qualunque metodo sincronizzato dello stesso oggetto finché il lucchetto non viene rilasciato. Se lo fa, verrà messo in attesa che ciò avvenga.

Si noti che eventuali metodi non sincronizzati di quell'oggetto possono comunque essere eseguiti da qualunque thread, in concorrenza col thread che possiede il lucchetto.

Dunque:
- relativamente all'insieme dei suoi metodi sincronizzati, un oggetto si comporta come un monitor, garantendo per essi un accesso in mutua esclusione;
- di conseguenza, è sufficiente gestire i dati condivisi per mezzo di metodi sincronizzati per garantire un corretto funzionamento dei thread cooperanti.

Ad esempio, consideriamo una classe che rappresenta un acconto bancario, sul quale possono potenzialmente essere fatte molte operazioni contemporaneamente:

```
public class Account    {
    private double balance;
//-------------------------------------------------
    public Account(double initialDeposit) {
        balance = initialDeposit;
    }
//-------------------------------------------------
    public synchronized double getBalance() {
        return balance;
    }
//-------------------------------------------------
    public synchronized void deposit(double amount) {
        balance += amount;
    }
}
```

Non può succedere che un thread legga il valore del conto mentre un altro thread lo aggiorna, o che due thread lo aggiornino in concorrenza.

Si noti che il costruttore non ha bisogno di essere sincronizzato, perché è eseguito solo per creare un oggetto, il che avviene una sola volta per ogni nuovo oggetto.

Anche i metodi statici possono essere sincronizzati. In questo caso il lucchetto è relativo alla classe e non alle sue istanze. In altre parole, sia una classe che le sue istanze possono funzionare come monitor. Va notato come tali monitor siano indipendenti, cioè il lucchetto sulla classe non ha alcun effetto sui metodi sincronizzati delle sue istanze e viceversa.

1.4.2) Istruzioni sincronizzate

E' possibile eseguire del codice sincronizzato, che quindi attiva il lucchetto di un oggetto, anche senza invocare un metodo sincronizzato di tale oggetto. Ciò si ottiene con le ***istruzioni sincronizzate***, che hanno questa forma:

```
...
synchronized(anObject) {
... // istruzioni sincronizzate
}
...
```

Questo modo di ottenere la sincrinizzazione fra thread cooperanti richiede maggior attenzione da parte del programmatore, che deve inserire blocchi di istruzioni sincronizzate in tutte le parti di codice interessate.

Esempio 9

Applicazione che realizza un server multithreaded, il quale:

- accetta connessioni da molteplici client;
- invia a tutti quelli connessi, in ***broadcast***, ciò che riceve da uno qualunque di loro.

Costituisce, dunque, un server per una semplice ***chatline***. Può essere usato con una semplice variazione del client visto nell'esempio 8: il client, appena si connette, deve inviare una linea di testo che il server usa come identificativo dell'utente connesso.

In tal modo si realizza di fatto una minimale architettura client-server.

Il codice è costituito da due classi. La prima, `ChatServer`, accetta richieste di connessione sul port 5000 e, ogni volta che ne arriva una, istanzia un oggetto della classe `ChatHandler` che si occupa di gestirla.

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

    public class ChatServer {
//---------------------------------------------
    public ChatServer() throws IOException {
        ServerSocket server = new ServerSocket(5000);
        System.out.println ("Accepting connections...");
        while(true) {
            Socket client = server.accept();
            System.out.println ("Accepted from " + client.getInetAddress());
            new ChatHandler(client).start();
        }
    }
//---------------------------------------------
    public static void main(String args[]) throws IOException {
        new ChatServer();
    }
}
```

La seconda si occupa della gestione di una singola connessione e dell'invio a tutte le altre, in broadcast, dei dati provenienti da tale connessione.

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

public class ChatHandler extends Thread {
  protected static Vector handlers = new Vector();
  protected String userName;
  protected Socket socket;
  protected DataInputStream is;
  protected PrintStream os;
//---------------------------------------------
```

```
  public ChatHandler(Socket socket) throws IOException {
    this.socket = socket;
    is = new DataInputStream(new BufferedInputStream(socket.getInputStream()));
    os = new PrintStream(new BufferedOutputStream(socket.getOutputStream()));
  }
//--------------------------------------------------
  public void run() {
    try {
      userName = is.readLine();
      os.println("Salve " + userName + ", benvenuto nella chatline di Reti II");
      os.flush();
      broadcast(this, "Salve a tutti! Ci sono anch'io.");
      handlers.addElement(this); //e' un metodo sincronizzato
      while (true) {
        String msg = is.readLine();
        if (msg != null) broadcast(this, msg);
        else break;
      }
    } catch(IOException ex) {
      ex.printStackTrace();
    } finally {
      handlers.removeElement(this); //e' un metodo sincronizzato
      broadcast(this, "Arrivederci a presto.");
      try {
        socket.close();
      } catch(Exception ex) {
        ex.printStackTrace();
      }
    }
  }
//--------------------------------------------------
  protected static void broadcast(ChatHandler sender, String message) {
    synchronized (handlers) { //ora nessuno puo' aggiungersi o abbandonare
      Enumeration e = handlers.elements();
      while (e.hasMoreElements()) {
        ChatHandler c = (ChatHandler) e.nextElement();
        try {
          c.os.print("Da " + sender.userName);
          c.os.print("@" + sender.socket.getInetAddress().getHostName() + ": ");
          c.os.println(message);
          c.os.flush();
        } catch(Exception ex) {
          c.stop();
        }
      }
    }
  }
}
```

1.4.3) Wait() e Notify()

In Java esiste anche un meccanismo per sospendere e risvegliare i thread che si trovano all'interno di un monitor, analogo a quello offerto dalle variabili di condizione.

Esistono tre metodi della classe Object che servono a questo:

```
public final void wait() throws InterruptedException;
```

> Un thread che chiama questo metodo di un oggetto viene sospeso finché un altro thread non chiama `notify()` o `notifyAll()` su quello stesso oggetto. Il lucchetto su quell'oggetto viene temporaneamente e atomicamente rilasciato, così altri thread possono entrare.

```
public final void notify() throws IllegalMonitorStateException;
```

> Questo metodo, chiamato su un oggetto, risveglia un thread (quello in attesa da più tempo) fra quelli sospesi dentro quell'oggetto. In particolare, tale thread riprenderà l'esecuzione solo quando il thread che ha chiamato `notify()` rilascia il lucchetto sull'oggetto.

```
public final void notifyAll() throws IllegalMonitorStateException;
```

> Questo metodo, chiamato su un oggetto, risveglia tutti i thread sospesi su quell'oggetto. Naturalmente, quando il chiamante rilascerà il lucchetto, solo uno dei risvegliati riuscirà a conquistarlo e gli altri verranno nuovamente sospesi.

E' importante ricordare che questi metodi possono essere usati solo all'interno di codice sincronizzato sullo stesso oggetto per il quale si invocano, e cioè:

- dentro metodi sincronizzati di quell'oggetto;
- dentro istruzioni sincronizzate su quell'oggetto.

Esempio 10

Esempio di produttore-consumatore che fa uso di `wait()` e `notify()`. La classe `Consumer` rappresenta un consumatore, che cerca di consumare un oggetto (se c'è) da un `Vector`.

Usa un thread separato per questo, e sfrutta `wait()` per rimanere in attesa di un oggetto da consumare.

Il produttore è costituito dall'utente, che attraverso il thread principale (quello del `main()`) aggiunge oggetti e invia `notify()` per risvegliare il consumatore.

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

public class Consumer extends Thread {
    protected Vector objects;
//-------------------------------------------------
    public Consumer () {
        objects = new Vector();
    }
//-------------------------------------------------
    public void run () {
        while (true) {
            Object object = extract ();
            System.out.println (object);
        }
    }
//-------------------------------------------------
    protected Object extract () {
        synchronized (objects) {
            while (objects.isEmpty ()) {
                try {
                    objects.wait ();
                } catch (InterruptedException ex) {
                    ex.printStackTrace ();
                }
            }
            Object o = objects.firstElement ();
            objects.removeElement (o);
            return o;
        }
    }
//-------------------------------------------------
    public void insert (Object o) {
        synchronized (objects) {
            objects.addElement (o);
            objects.notify ();
        }
    }
//-------------------------------------------------
    public static void main (String args[]) throws IOException,
                                                   InterruptedException {
        Consumer c = new Consumer ();
        c.start ();
        DataInputStream i = new DataInputStream (System.in);
        String s;
        while ((s = i.readLine ()) != null) {
            c.insert (s);
            Thread.sleep (1000);
        }
    }
}
```

Torna all'indice | Vai avanti