Appunti di Progettazione del Software

Canale 2 · Prof. Massimo Mecella

Capitolo 9: Socket e applicazioni client-server

Autore: Tommaso C.

Appunti tratti dalle lezioni in aula e dal materiale del corso

---

Argomenti: Reti, client e server · Indirizzi e porte · Socket in Java
Server di eco · Server multithreading · Protocollo applicativo
Client con interfaccia grafica · Server con thread per client

Indice

1 Reti, client e server **1**

2 Socket in Java **2**

3 Un server di eco **3**

4 Server multithreading **5**

5 Una prova pratica: la bandiera **6**

5.1 Il testo 6

5.2 Il protocollo 7

6 Il client **8**

6.1 La finestra 8

6.2 L'ascoltatore 11

6.3 Il lettore 13

7 Il server **15**

7.1 La finestra del server 15

7.2 Server 17

7.3 ClientThread e SenderThread 19

8 Esercizi **22**

9 Soluzioni **25**

1 Reti, client e server

Una **rete** collega più calcolatori, che si scambiano dati. Perché due programmi su macchine diverse si capiscano, devono seguire le stesse regole.

Definizione – Protocollo

Un **protocollo** è l'insieme delle regole con cui due programmi si scambiano messaggi: quali messaggi esistono, in che formato e in che ordine.

Su Internet i dati viaggiano con i protocolli **TCP/IP**. Il TCP offre una connessione affidabile: i dati arrivano tutti, senza errori e nell'ordine in cui sono partiti. Sopra il TCP ogni applicazione definisce il proprio protocollo, detto **protocollo applicativo**: per esempio, quali stringhe manda il client e che cosa risponde il server.

Definizione – Client e server

- Il **server** è un programma che resta in attesa di richieste e offre un servizio.
- Il **client** è un programma che si connette al server e gli chiede il servizio.

Nel modello **peer-to-peer**, invece, ogni programma fa sia da client sia da server.

Definizione – Indirizzo e porta

- L'**indirizzo IP** identifica una macchina in rete. Nella versione IPv4 è un numero di 32 bit, scritto come quattro numeri da 0 a 255 separati da punti, per esempio `127.0.0.1`. Il DNS traduce i nomi, come `www.uniroma1.it`, in indirizzi.
- La **porta** è un numero da 0 a 65535 che identifica, dentro una macchina, il programma a cui sono destinati i dati.

L'indirizzo `127.0.0.1` indica sempre la macchina stessa: con esso client e server si provano sullo stesso calcolatore. Le porte sotto 1024 sono riservate ai servizi noti, per esempio 80 per HTTP, 21 per FTP e 22 per SSH; un'applicazione nuova usa una porta più alta, come 4400.

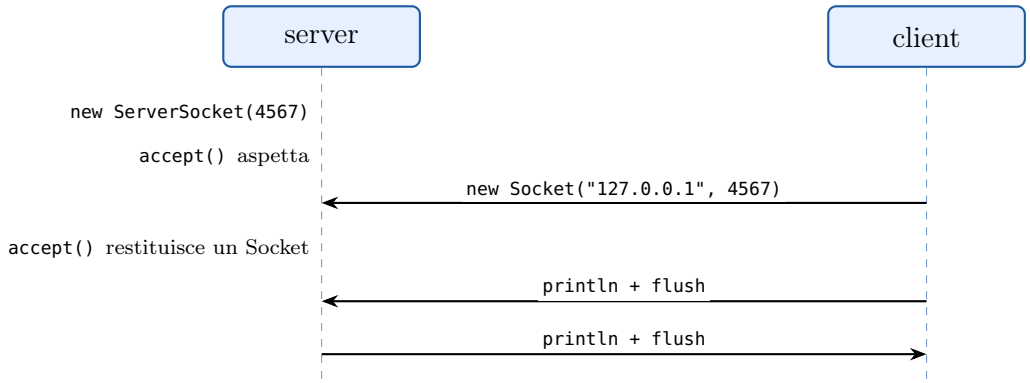
2 Socket in Java

Definizione – Socket

Una **socket** è un estremo di una connessione fra due programmi. Ciascun programma scrive nella propria socket e legge dalla propria socket, come se fossero un file.

Le classi stanno nel package `java.net`. Il server e il client usano due classi diverse:

- il server crea un `ServerSocket` su una porta, e con `accept()` aspetta un client;
- il client crea un `Socket` con l'indirizzo e la porta del server;
- quando il client si connette, `accept()` restituisce un `Socket` che, nel server, rappresenta quella connessione.



La connessione: il server aspetta in `accept()`, il client si connette, poi i due si scambiano righe di testo nei due versi.

Nel server:

```
ServerSocket serverSocket = new ServerSocket(4567);
Socket socket = serverSocket.accept(); // aspetta un client
```

Nel client:

```
Socket socket = new Socket("127.0.0.1", 4567);
```

Da un `Socket` si ottengono due flussi: `getInputStream()` per leggere ciò che arriva e `getOutputStream()` per scrivere ciò che parte. Per scambiare righe di testo si avvolgono in uno `Scanner`, già usato per la tastiera, e in un `PrintWriter`, nuovo:

```
Scanner in = new Scanner(socket.getInputStream());
PrintWriter out = new PrintWriter(socket.getOutputStream());
out.println("ciao");
out.flush();
String risposta = in.nextLine();
```

Proposizione – flush

`PrintWriter` raccoglie i caratteri in un buffer e li manda tutti insieme. Dopo ogni messaggio che l'altro programma deve ricevere subito si chiama `flush()`: senza, il messaggio può restare nel buffer, e l'altro programma resta fermo in `nextLine()` ad aspettarlo.

`nextLine()` blocca il thread finché non arriva una riga intera. Quando l'altro programma chiude la connessione, non arriveranno più righe, e `nextLine()` lancia una `NoSuchElementException`. Molte operazioni sulle socket, a partire dai costruttori, possono lanciare una `IOException`, da gestire o da dichiarare con `throws` (Capitolo 2).

Proposizione – Chiudere

Alla fine si chiudono i flussi e la socket con `close()`. Chiudere il `ServerSocket` sblocca un thread fermo in `accept()`, che lancia una `IOException`; `isClosed()` dice se una socket è già chiusa.

Client e server sono due programmi distinti, ciascuno con il suo `main`: si avviano in due processi, prima il server e poi il client. La classe `InetAddress` rappresenta un indirizzo e, con `InetAddress.getByName("www.uniroma1.it")`, lo ricava da un nome.

3 Un server di eco

Il primo esempio è un server di **eco**. Il protocollo è semplice: il client manda righe; il server risponde a ciascuna con `Received:` seguito dalla riga, e chiude la connessione quando riceve `quit`.

```
import java.io.IOException;
import java.io.PrintWriter;
import java.net.ServerSocket;
import java.net.Socket;
import java.util.Scanner;

public class EchoServer {
    public static void main(String[] args) throws IOException {
        ServerSocket serverSocket = new ServerSocket(1337);
        System.out.println("In ascolto sulla porta 1337");
        Socket socket = serverSocket.accept();
        System.out.println("Client connesso");
        Scanner in = new Scanner(socket.getInputStream());
        PrintWriter out = new PrintWriter(socket.getOutputStream());
        while (true) {
            String line = in.nextLine();
```

```
            if (line.equals("quit")) {
                break;
            }
            out.println("Received: " + line);
            out.flush();
        }
        System.out.println("Chiudo la connessione");
        in.close();
        out.close();
        socket.close();
        serverSocket.close();
    }
}
```

Il client legge una riga dalla tastiera, la manda al server e stampa la risposta. Ha quindi due `Scanner`: uno sulla tastiera e uno sulla socket.

```
import java.io.IOException;
import java.io.PrintWriter;
import java.net.Socket;
import java.util.NoSuchElementException;
import java.util.Scanner;

public class LineClient {
    public static void main(String[] args) throws IOException {
        Socket socket = new Socket("127.0.0.1", 1337);
        System.out.println("Connessione stabilita");
        Scanner socketIn = new Scanner(socket.getInputStream());
        PrintWriter socketOut = new PrintWriter(socket.getOutputStream());
        Scanner stdin = new Scanner(System.in);
        try {
            while (true) {
                String inputLine = stdin.nextLine();
                socketOut.println(inputLine);
                socketOut.flush();
                String socketLine = socketIn.nextLine();
                System.out.println(socketLine);
            }
        } catch (NoSuchElementException e) {
            System.out.println("Connessione chiusa");
        } finally {
            stdin.close();
            socketIn.close();
            socketOut.close();
            socket.close();
        }
    }
```

```
}
```

Esempio – Una sessione

Si avvia `EchoServer`, poi `LineClient`, e nel client si scrivono `ciao`, `mondo` e `quit`. Il client stampa:

```
Connessione stabilita
Received: ciao
Received: mondo
Connessione chiusa
```

Dopo `quit` il server chiude la socket senza rispondere: la `nextLine()` del client lancia una `NoSuchElementException`, e il client stampa `Connessione chiusa`.

4 Server multithreading

`EchoServer` serve un solo client e poi termina. Un server vero serve molti client insieme. Il thread principale del server fa solo una cosa: aspetta in `accept()` e, per ogni client che arriva, avvia un thread nuovo che lo serve. Poi torna subito ad aspettare il client successivo.

```
import java.io.IOException;
import java.net.ServerSocket;
import java.net.Socket;

public class MultiEchoServer {
    public static void main(String[] args) throws IOException {
        ServerSocket serverSocket = new ServerSocket(1337);
        while (true) {
            Socket socket = serverSocket.accept();
            new Thread(new ClientHandler(socket)).start();
        }
    }
}
```

Il lavoro per un client sta nel `run` di un `Runnable`, che riceve la sua socket nel costruttore:

```
import java.io.IOException;
import java.io.PrintWriter;
import java.net.Socket;
import java.util.Scanner;

public class ClientHandler implements Runnable {
    private final Socket socket;
```

```
    public ClientHandler(Socket socket) {
        this.socket = socket;
    }

    public void run() {
        try {
            Scanner in = new Scanner(socket.getInputStream());
            PrintWriter out = new PrintWriter(socket.getOutputStream());
            while (true) {
                String line = in.nextLine();
                if (line.equals("quit")) {
                    break;
                }
                out.println("Received: " + line);
                out.flush();
            }
            in.close();
            out.close();
            socket.close();
        } catch (IOException e) {
            System.err.println(e.getMessage());
        }
    }
}
```

Osservazione – Perché un thread per client

`nextLine()` blocca il thread che la chiama. Con un solo thread, mentre il server aspetta una riga da un client, non può accettarne altri né rispondere agli altri. Con un thread per client, ciascun thread si blocca solo sulla propria socket.

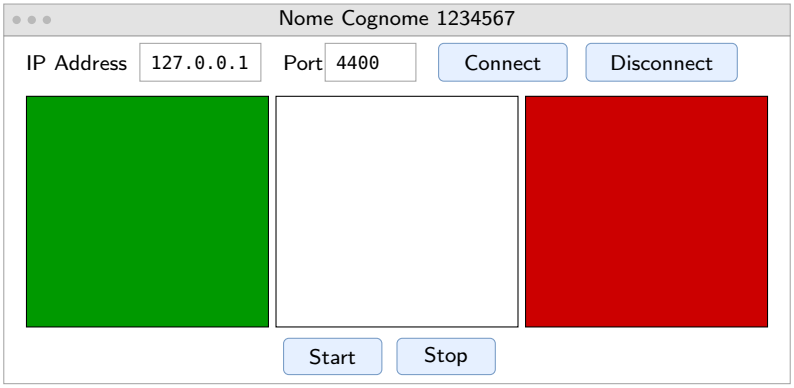
5 Una prova pratica: la bandiera

La prova pratica chiede di scrivere in Java, in un tempo limitato, il client di un'applicazione client-server: un'interfaccia grafica che comunica con un server già pronto. L'esempio completo di questo capitolo è la prova del 25/01/2019.

5.1 Il testo

Il server, multithreading, manda ai client i colori di bandiere a tre bande verticali. Il client mostra la bandiera in tre pannelli affiancati.

- La finestra ha come titolo nome, cognome e matricola dello studente; ha due campi per l'indirizzo e la porta del server, quattro pulsanti (`Connect`, `Disconnect`, `Start`, `Stop`) e tre pannelli verticali, grigio chiaro all'avvio.
- All'avvio è abilitato solo `Connect`.
- `Start` e `Disconnect` funzionano solo da connessi. Dopo `Start` è abilitato solo `Stop`; dopo `Stop`, `Start` e `Disconnect`.



Il client mentre riceve una bandiera: indirizzo, porta e connessione in alto, i tre pannelli al centro, `Start` e `Stop` in basso.

5.2 Il protocollo

La comunicazione è solo uno scambio di stringhe, una per riga.

Messaggio	Significato
`start`	dal client: il server inizia a mandare bandiere
`codice;posizione`	dal server: colora un pannello; `posizione` è `SX`, `CX` o `DX` (sinistra, centro, destra)
`stop`	dal client: il server smette di mandare bandiere
`-1;-1`	dal server: risposta a `stop`, la trasmissione è finita
`disconnect`	dal client: chiude la connessione

I codici dei colori sono: 0 bianco, 1 nero, 2 verde, 3 rosso, 4 giallo, 5 blu e 6 arancione. Una bandiera arriva come tre righe, una per pannello: l'Italia, per esempio, come `2;SX`, `0;CX` e `3;DX`.

Dopo `Stop` i pannelli restano colorati con l'ultima bandiera; un nuovo `Start` e `Disconnect` li riportano grigio chiaro. `Disconnect` manda `disconnect`, chiude i canali e riabilita `Connect`: ci si può riconnettere senza riavviare il client.

Situazione	`Connect`	`Disconnect`	`Start`	`Stop`
non connesso	sì	no	no	no
connesso, fermo	no	sì	sì	no
connesso, in trasmissione	no	no	no	sì

Al server della prova si possono collegare più client insieme, e ciascuno riceve bandiere diverse: ogni client ha il suo thread nel server.

6 Il client

Il client ha quattro classi:

- `Main` crea la finestra e la mostra;
- `StatusDownloaderFrame` è la finestra;
- `ClientListener` è l'ascoltatore dei pulsanti e gestisce la connessione;
- `Downloader` legge le righe del server in un thread separato e colora i pannelli.

```
public class Main {
    public static void main(String[] args) {
        StatusDownloaderFrame frame = new StatusDownloaderFrame();
        frame.setSize(800, 600);
        frame.setVisible(true);
    }
}
```

6.1 La finestra

La finestra usa un `BorderLayout`: a nord un pannello con indirizzo, porta, `Connect` e `Disconnect`; al centro la bandiera; a sud `Start` e `Stop`. La bandiera è un pannello con `GridLayout(1, 3)`, che contiene i tre pannelli con bordo nero e sfondo grigio chiaro. Il testo non impone una disposizione: ne vanno bene molte, purché i componenti richiesti ci siano.

```
import java.awt.BorderLayout;
import java.awt.Color;
import java.awt.Container;
import java.awt.FlowLayout;
import java.awt.GridLayout;
import javax.swing.BorderFactory;
import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JPanel;
import javax.swing.JTextField;

public class StatusDownloaderFrame extends JFrame {
    private JButton connectButton, disconnectButton, startButton, stopButton;
```

```
    public StatusDownloaderFrame() {
        super("Nome Cognome 1234567");
        Container mainContainer = this.getContentPane();

        JPanel addressPanel = new JPanel(new FlowLayout());
        addressPanel.add(new JLabel("IP Address"));
        JTextField addressText = new JTextField(10);
        addressText.setText("127.0.0.1");
        addressPanel.add(addressText);

        JPanel portPanel = new JPanel(new FlowLayout());
        portPanel.add(new JLabel("Port"));
        JTextField portText = new JTextField(10);
        portText.setText("4400");
        portPanel.add(portText);

        JPanel sinistra = new JPanel();
        JPanel centro = new JPanel();
        JPanel destra = new JPanel();
        JPanel bandiera = new JPanel(new GridLayout(1, 3));
        for (JPanel p : new JPanel[] {sinistra, centro, destra}) {
            p.setBorder(BorderFactory.createLineBorder(Color.BLACK));
            p.setBackground(Color.LIGHT_GRAY);
            bandiera.add(p);
        }

        // prima l'ascoltatore, poi i pulsanti
        ClientListener list = new ClientListener(addressText, portText,
                sinistra, centro, destra);
        connectButton = creaPulsante("Connect", ClientListener.CONNECT, list);
        disconnectButton = creaPulsante("Disconnect", ClientListener.DISCONNECT
    , list);
        startButton = creaPulsante("Start", ClientListener.START, list);
        stopButton = creaPulsante("Stop", ClientListener.STOP, list);

        JPanel northPanel = new JPanel(new FlowLayout());
        northPanel.add(addressPanel);
        northPanel.add(portPanel);
        northPanel.add(connectButton);
        northPanel.add(disconnectButton);
        JPanel southPanel = new JPanel(new FlowLayout());
        southPanel.add(startButton);
        southPanel.add(stopButton);

        mainContainer.add(northPanel, BorderLayout.NORTH);
        mainContainer.add(bandiera, BorderLayout.CENTER);
```

```
        mainContainer.add(southPanel, BorderLayout.SOUTH);
        this.setLocation(200, 100);
        this.setButtons(false, false);
    }

    private JButton creaPulsante(String testo, String comando, ClientListener
    list) {
        JButton b = new JButton(testo);
        b.setActionCommand(comando);
        b.addActionListener(list);
        return b;
    }
```

Nota – Prima l'ascoltatore, poi i pulsanti

I pulsanti ricevono l'ascoltatore appena sono creati. Un pulsante visibile senza ascoltatore ignorerebbe i clic dell'utente, e l'applicazione sembrerebbe non rispondere.

I pulsanti usano come comando le costanti di `ClientListener`, che coincidono con le stringhe del protocollo. Lo stato dei pulsanti dipende da due condizioni: se il client è connesso e se sta ricevendo. Il metodo `setButtons` le traduce nella tabella della Sezione 5:

```
    public void setButtons(boolean connected, boolean transmitting) {
        if (connected) {
            this.setDefaultCloseOperation(JFrame.DO_NOTHING_ON_CLOSE);
            connectButton.setEnabled(false);
            disconnectButton.setEnabled(!transmitting);
            startButton.setEnabled(!transmitting);
            stopButton.setEnabled(transmitting);
        } else {
            this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
            connectButton.setEnabled(true);
            disconnectButton.setEnabled(false);
            startButton.setEnabled(false);
            stopButton.setEnabled(false);
        }
    }
}
```

Da connesso, la finestra non si chiude con la crocetta (`DO_NOTHING_ON_CLOSE`): prima bisogna disconnettersi, così la connessione si chiude in modo ordinato.

6.2 L'ascoltatore

`ClientListener` tiene lo stato della connessione: i due flag, la socket, lo `Scanner` e il `PrintWriter`, e il thread che legge.

```
import java.awt.Color;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.io.IOException;
import java.io.PrintWriter;
import java.net.Socket;
import java.util.Scanner;
import javax.swing.JComponent;
import javax.swing.JOptionPane;
import javax.swing.JPanel;
import javax.swing.JTextField;
import javax.swing.SwingUtilities;

public class ClientListener implements ActionListener {
    public static final String CONNECT = "connect";
    public static final String DISCONNECT = "disconnect";
    public static final String START = "start";
    public static final String STOP = "stop";

    private final JTextField addressField, portField;
    private final JPanel sinistra, centro, destra;
    private boolean connected = false, transmitting = false;
    private Socket socket;
    private PrintWriter netPw;
    private Scanner scan;
    private StatusDownloaderFrame frame;
    private Thread downloaderThread;

    public ClientListener(JTextField addressField, JTextField portField,
            JPanel sinistra, JPanel centro, JPanel destra) {
        this.addressField = addressField;
        this.portField = portField;
        this.sinistra = sinistra;
        this.centro = centro;
        this.destra = destra;
    }

    private void setupConnection() throws IOException {
        socket = new Socket(addressField.getText(),
                Integer.parseInt(portField.getText()));
        netPw = new PrintWriter(socket.getOutputStream());
        scan = new Scanner(socket.getInputStream());
```

```
    }

    private void grigio() {
        sinistra.setBackground(Color.LIGHT_GRAY);
        centro.setBackground(Color.LIGHT_GRAY);
        destra.setBackground(Color.LIGHT_GRAY);
    }
```

`ClientListener` non riceve la finestra nel costruttore, perché la finestra lo crea prima di avere i pulsanti. La recupera al primo evento: `SwingUtilities.getRoot` risale l'albero dei componenti dalla sorgente dell'evento fino alla finestra. In alternativa si può passare `this` al costruttore dell'ascoltatore.

```
    public void actionPerformed(ActionEvent e) {
        if (frame == null) {
            frame = (StatusDownloaderFrame)
                    SwingUtilities.getRoot((JComponent) e.getSource());
        }
        String command = e.getActionCommand();
        if (command.equals(CONNECT)) {
            try {
                setupConnection();
                connected = true;
                System.out.println("Connesso");
            } catch (IOException | NumberFormatException ex) {
                JOptionPane.showMessageDialog(null,
                        "Impossibile connettersi al server");
                return;
            }
        } else if (command.equals(START)) {
            grigio();
            Downloader downloader = new Downloader(scan, sinistra, centro,
    destra);
            downloaderThread = new Thread(downloader);
            downloaderThread.start();
            netPw.println(command);
            netPw.flush();
            transmitting = true;
        } else if (command.equals(STOP)) {
            netPw.println(command);
            netPw.flush();
            try {
                downloaderThread.join(); // fino a -1;-1
            } catch (InterruptedException ex) {
            }
            transmitting = false;
```

```
        } else if (command.equals(DISCONNECT)) {
            grigio();
            netPw.println(command);
            netPw.flush();
            scan.close();
            netPw.close();
            try {
                socket.close();
            } catch (IOException ex) {
                System.err.println(ex.getMessage());
            }
            connected = false;
        }
        frame.setButtons(connected, transmitting);
    }
}
```

Alcuni punti meritano attenzione.

- **Prima il lettore, poi `start`.** Su `Start` il client avvia il `Downloader` e solo dopo manda `start`: chi riceve deve essere pronto prima che l'altro cominci a mandare. Le righe arrivate prima non andrebbero perse, perché la socket le conserva, ma è l'ordine naturale.
- **Una `flush()` dopo ogni `println`**, altrimenti il comando può restare nel buffer.
- **`setButtons` per ultimo**, con lo stato già aggiornato.
- **Errori visibili**: se la connessione fallisce, una finestra di dialogo lo dice all'utente.

Proposizione – Non leggere nel thread di Swing

`actionPerformed` gira nel thread che gestisce l'interfaccia (Capitolo 8). Se leggesse le righe del server, resterebbe bloccato in `nextLine()` per tutta la trasmissione: la finestra non si ridisegnerebbe e non risponderebbe più ai pulsanti, compreso `Stop`. Per questo la lettura va in un thread separato.

6.3 Il lettore

`Downloader` riceve solo ciò che gli serve: lo `Scanner` e i tre pannelli. Legge una riga alla volta, la divide con `split(";")` e colora il pannello indicato. Si ferma quando riceve `-1;-1` o quando la connessione si chiude.

```
import java.awt.Color;
import java.util.NoSuchElementException;
import java.util.Scanner;
import javax.swing.JPanel;
```

```java
public class Downloader implements Runnable {
    private final Scanner scan;
    private final JPanel sinistra, centro, destra;
    private boolean running = false;

    public Downloader(Scanner scan, JPanel sinistra, JPanel centro, JPanel
    destra) {
        this.scan = scan;
        this.sinistra = sinistra;
        this.centro = centro;
        this.destra = destra;
    }

    public void run() {
        running = true;
        while (running) {
            String line;
            try {
                line = scan.nextLine();
            } catch (NoSuchElementException | IllegalStateException e) {
                running = false; // connessione chiusa
                break;
            }
            System.out.println(line);
            String[] command = line.split(";");
            if (command[0].equals("-1") && command[1].equals("-1")) {
                running = false;
            } else {
                Color colore = colore(command[0]);
                if (command[1].equals("SX")) {
                    sinistra.setBackground(colore);
                } else if (command[1].equals("CX")) {
                    centro.setBackground(colore);
                } else if (command[1].equals("DX")) {
                    destra.setBackground(colore);
                }
            }
        }
    }

    private Color colore(String codice) {
        switch (codice) {
            case "0": return Color.WHITE;
            case "1": return Color.BLACK;
            case "2": return Color.GREEN;
```

```
            case "3": return Color.RED;
            case "4": return Color.YELLOW;
            case "5": return Color.BLUE;
            case "6": return Color.ORANGE;
            default: return Color.LIGHT_GRAY;
        }
    }

    public boolean isRunning() {
        return running;
    }
}
```

`setBackground` ridisegna il pannello da solo: non serve chiamare altri metodi. Il server manda le tre righe di una bandiera una dopo l'altra, e il client le tratta una per volta, perché ogni riga è un messaggio completo.

7 Il server

All'esame il server è già pronto; leggerne il codice mostra però lo schema della Sezione 4 in un caso completo. Ha quattro classi:

- `GUI_Main`: una piccola finestra con la porta, `Start`, `Stop` e il numero dei client connessi;
- `Server`: un `Runnable` che accetta le connessioni e tiene la lista dei client;
- `ClientThread`: serve un client e ne legge i comandi;
- `SenderThread`: manda le bandiere a quel client.

7.1 La finestra del server

`GUI_Main` fa da ascoltatore di sé stessa, perché ha due soli pulsanti. Su `Start` non fa il lavoro del server: crea un `Server` e lo avvia in un thread nuovo, così la finestra resta reattiva.

```
import java.awt.BorderLayout;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.io.IOException;
import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JPanel;
import javax.swing.JTextField;
```

```
public class GUI_Main extends JFrame implements ActionListener {
    private final JButton startBtn = new JButton("Start");
    private final JButton stopBtn = new JButton("Stop");
    private final JTextField portTxt = new JTextField(10);
    private final JLabel clientsLabel = new JLabel("0");
    private Server serv;

    public GUI_Main() {
        super("Server");
        portTxt.setText("4400");
        JPanel topPanel = new JPanel();
        topPanel.add(new JLabel("Port"));
        topPanel.add(portTxt);
        JPanel centralPanel = new JPanel();
        centralPanel.add(startBtn);
        centralPanel.add(stopBtn);
        JPanel southPanel = new JPanel();
        southPanel.add(clientsLabel);
        this.getContentPane().add(topPanel, BorderLayout.NORTH);
        this.getContentPane().add(centralPanel, BorderLayout.CENTER);
        this.getContentPane().add(southPanel, BorderLayout.SOUTH);
        this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        startBtn.setActionCommand("start");
        startBtn.addActionListener(this);
        stopBtn.setActionCommand("stop");
        stopBtn.addActionListener(this);
        stopBtn.setEnabled(false);
        this.pack();
        this.setVisible(true);
    }

    public void actionPerformed(ActionEvent e) {
        if (e.getActionCommand().equals("start")) {
            serv = new Server(portTxt.getText(), clientsLabel);
            new Thread(serv).start();
            startBtn.setEnabled(false);
            stopBtn.setEnabled(true);
            this.setDefaultCloseOperation(JFrame.DO_NOTHING_ON_CLOSE);
        } else {
            try {
                serv.stop();
            } catch (IOException ex) {
                System.err.println(ex.getMessage());
            }
            stopBtn.setEnabled(false);
```

```
            this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        }
    }

    public static void main(String[] args) {
        new GUI_Main();
    }
}
```

7.2 Server

`Server` legge all'avvio le quattro bandiere da file CSV, una riga per pannello, con il `BufferedReader` del Capitolo 2. Nel `run` apre il `ServerSocket` e ripete: `accept()`, un `ClientThread` nuovo avviato in un thread, l'aggiunta alla lista e l'aggiornamento dell'etichetta.

```
import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;
import java.net.ServerSocket;
import java.net.Socket;
import java.util.ArrayList;
import java.util.LinkedList;
import java.util.List;
import javax.swing.JLabel;

public class Server implements Runnable {
    private final String port;
    private final JLabel clientsLabel;
    private final List<ClientThread> ctList = new LinkedList<ClientThread>();
    private final List<List<String>> bandiere = new ArrayList<List<String>>();
    private ServerSocket lis;

    public Server(String port, JLabel clientsLabel) {
        this.port = port;
        this.clientsLabel = clientsLabel;
        try {
            bandiere.add(leggiFile("BELGIO.csv"));
            bandiere.add(leggiFile("FRANCIA.csv"));
            bandiere.add(leggiFile("IRLANDA.csv"));
            bandiere.add(leggiFile("ITALIA.csv"));
        } catch (IOException e) {
            System.err.println(e.getMessage());
        }
    }
```

```
public void run() {
    try {
        lis = new ServerSocket(Integer.parseInt(port));
    } catch (IOException e) {
        System.err.println("Errore nella creazione del ServerSocket");
        return;
    }
    System.out.println("Server avviato");
    while (true) {
        Socket sock;
        try {
            sock = lis.accept();
        } catch (IOException e) {
            break; // stop() ha chiuso il ServerSocket
        }
        System.out.println("Connessione accettata");
        ClientThread ct = new ClientThread(sock, this);
        new Thread(ct).start();
        synchronized (ctList) {
            ctList.add(ct);
            clientsLabel.setText("" + ctList.size());
        }
    }
}
```

La lista dei client è condivisa: la modifica il thread di `Server` quando un client arriva, e il thread di un `ClientThread` quando il suo client se ne va. Per questo ogni accesso sta in un blocco `synchronized` sulla lista (Capitolo 8).

```
public void stop() throws IOException {
    lis.close();
    synchronized (ctList) {
        for (ClientThread ct : ctList) {
            ct.stop();
        }
        ctList.clear();
    }
    System.out.println("Server fermato");
}

public void removeClient(ClientThread ct) {
    synchronized (ctList) {
        ctList.remove(ct);
        clientsLabel.setText("" + ctList.size());
    }
```

```
    }

    public List<String> getBandiera(int i) {
        return bandiere.get(i);
    }

    private List<String> leggiFile(String nome) throws IOException {
        List<String> righe = new ArrayList<String>();
        BufferedReader br = new BufferedReader(new FileReader(nome));
        String riga = br.readLine();
        while (riga != null) {
            righe.add(riga);
            riga = br.readLine();
        }
        br.close();
        return righe;
    }
}
```

7.3 ClientThread e SenderThread

`ClientThread` legge i comandi del suo client. Su `start` avvia un `SenderThread`, un secondo thread dedicato all'invio: così `ClientThread` resta libero di leggere `stop` mentre le bandiere partono. Su `stop` ferma il `SenderThread`. Ogni altra stringa, compreso `disconnect`, chiude la connessione.

```
import java.io.IOException;
import java.io.PrintWriter;
import java.net.Socket;
import java.util.NoSuchElementException;
import java.util.Scanner;

public class ClientThread implements Runnable {
    private final Socket sock;
    private final Server parent;
    private SenderThread st;
    private Thread stThread;

    public ClientThread(Socket sock, Server parent) {
        this.sock = sock;
        this.parent = parent;
    }

    public void run() {
        try {
```

```
            Scanner scan = new Scanner(sock.getInputStream());
            PrintWriter pw = new PrintWriter(sock.getOutputStream());
            boolean running = true;
            while (running) {
                String cmd = scan.nextLine();
                System.out.println("Ricevuto: " + cmd);
                if (cmd.equals("start")) {
                    st = new SenderThread(pw, parent);
                    stThread = new Thread(st);
                    stThread.start();
                } else if (cmd.equals("stop")) {
                    fermaSender();
                } else {
                    running = false; // disconnect o comando sconosciuto
                }
            }
            scan.close();
            pw.close();
            sock.close();
        } catch (IOException | NoSuchElementException e) {
            System.out.println("Connessione chiusa");
        }
        parent.removeClient(this);
    }

    private void fermaSender() {
        st.stop();
        stThread.interrupt(); // sveglia la sleep
        try {
            stThread.join(); // aspetta che abbia mandato -1;-1
        } catch (InterruptedException e) {
        }
    }

    public void stop() throws IOException {
        if (st != null) {
            fermaSender();
        }
        sock.close();
    }
}
```

`SenderThread` sceglie a caso una bandiera, ne manda le tre righe, chiama `flush()` e dorme un secondo; poi ricomincia, finché non viene fermato. Alla fine manda `-1;-1`. Il flag `running` è letto dal `SenderThread` e scritto da `ClientThread`, quindi i due metodi che lo usano sono `synchronized`, come nel Capitolo 8.

```java
import java.io.PrintWriter;
import java.util.List;

public class SenderThread implements Runnable {
    private final PrintWriter pw;
    private final Server serv;
    private boolean running = true;

    public SenderThread(PrintWriter pw, Server serv) {
        this.pw = pw;
        this.serv = serv;
    }

    public void run() {
        while (isRunning()) {
            int i = (int) (Math.random() * 4);
            List<String> bandiera = serv.getBandiera(i);
            for (String riga : bandiera) {
                pw.println(riga);
            }
            pw.flush();
            try {
                Thread.sleep(1000);
            } catch (InterruptedException e) {
            }
        }
        pw.println("-1;-1");
        pw.flush();
    }

    public synchronized void stop() {
        running = false;
    }

    public synchronized boolean isRunning() {
        return running;
    }
}
```

Osservazione – Tre righe, una sola flush

Le tre righe di una bandiera partono insieme con una sola `flush()`, ma il client non deve aspettarle insieme: legge una riga alla volta con `nextLine()` e la tratta subito. Il protocollo è fatto di righe, e il TCP le consegna intere e nell'ordine giusto, comunque siano raggruppate durante il viaggio.

Riepilogo

- **Indirizzo e porta**: l'IP individua la macchina, la porta il programma; `127.0.0.1` è la macchina stessa.
- **Server**: `new ServerSocket(porta)` e `accept()`, che aspetta un client e restituisce un `Socket`.
- **Client**: `new Socket(indirizzo, porta)`.
- **Righe**: `Scanner` sull'input, `PrintWriter` sull'output, `flush()` dopo ogni messaggio; a connessione chiusa `nextLine()` lancia una `NoSuchElementException`.
- **Server multithreading**: il thread principale accetta, un thread per client lo serve; le strutture condivise vanno protette con `synchronized`.
- **Client grafico**: la lettura dalla socket va in un thread separato, mai nel thread di Swing.
- **Protocollo**: costanti uguali alle stringhe scambiate, prima il lettore e poi il comando che fa partire i dati.
- **Pulsanti**: uno stato (connesso, in trasmissione) e un metodo che abilita i pulsanti di conseguenza.

8 Esercizi

Esercizio 1 – Leggere in actionPerformed

Nel client della Sezione 6, uno studente toglie `Downloader` e scrive il ciclo di lettura direttamente nel ramo `START` di `actionPerformed`, fino a `-1;-1`. Che cosa vede l'utente dopo aver premuto `Start`? Riesce a premere `Stop`?

Soluzione a pagina 25 →

Esercizio 2 – La flush dimenticata

In `LineClient` (Sezione 3) si toglie `socketOut.flush()`. Si avviano `EchoServer` e il client, e nel client si scrive `ciao`. Che cosa succede? Come lo si spiega?

Soluzione a pagina 25 →

Esercizio 3 – Stop e subito Start

Nel ramo `STOP` di `ClientListener` si toglie la `join` e si mette subito `transmitting = false`, come se la trasmissione finisse con il clic. L'utente preme `Stop` e, prima che arrivi `-1;-1`, `Start`. Che cosa può succedere? Perché la `join` lo evita?

Soluzione a pagina 25 →

Esercizio 4 – Fermare il server

Il thread di `Server` è fermo in `lis.accept()`. L'utente preme `Stop` nella finestra del server. Come termina il ciclo di `run`? Che cosa ricevono i client connessi?

Soluzione a pagina 26 →

Esercizio 5 – Appello del 06/09/2023, Canale 1, Prof. Paolo Liberatore, esercizio 1, adattato

Si vuole realizzare il client di un'applicazione client-server che mostra nomi e dimensioni di file presenti in un server. Il server è multithreading; la comunicazione è solo uno scambio di stringhe, separate dal fine riga.

La finestra ha come titolo nome, cognome e matricola; ha due campi editabili per indirizzo e porta; tre aree di testo non editabili, *Log*, *.pdf* e *.mp3*; un campo non editabile con la dimensione totale dei file ricevuti; i pulsanti `Connect`, `Disconnect`, `Start`, `Stop` e `Clear`.

- All'avvio è abilitato solo `Connect`. Dopo la connessione, `Connect` si disabilita e si abilitano `Start`, `Clear` e `Disconnect`.
- `Start` manda `start`, cancella le aree dei risultati e disabilita `Start`, `Clear` e `Disconnect`. Il server manda, a intervalli regolari, righe nella forma

  `<tipo>:<nome>.<formato>:<dimensione> KB`

  con `<tipo>` uguale a `PDF` o `MP3` e `<dimensione>` un numero decimale, per esempio `PDF:doc_wmeg0r40.pdf:73.33 KB`. Per ogni riga il client aggiunge il nome all'area del suo tipo, aggiunge al Log

  `<nome>.<formato> <dimensione> KB`

e aggiorna il totale.

- `Stop` manda `stop`. Al termine dell'invio il server manda `END` (completato) o `INTERRUPTED` (fermato dal client); allora il client abilita `Start`, `Clear` e `Disconnect` e disabilita `Stop`.
- `Disconnect` manda `disconnect`, chiude i canali e abilita solo `Connect`.
- Per ogni altra stringa il server risponde `ERROR` e chiude la connessione: il client chiude i canali e abilita solo `Connect`.
- `Clear` cancella le aree di testo.

Suggerimenti. `new JScrollPane(area)` mette una `JTextArea` in un contenitore con le barre di scorrimento, e si aggiunge al pannello al posto dell'area. `area.append(s)` aggiunge una stringa in fondo all'area; `setEditable(false)` rende un campo o un'area non editabile; `Double.parseDouble(s)` converte una stringa in un `double`.

Soluzione a pagina 26 →

9 Soluzioni

Soluzione dell'Esercizio 1 – Leggere in actionPerformed

Il ciclo gira nel thread che gestisce l'interfaccia e resta bloccato in `nextLine()` in attesa delle righe. Finché il ciclo non finisce, quel thread non ridisegna la finestra e non gestisce altri eventi. L'utente vede la finestra ferma: i pannelli non cambiano colore anche se `setBackground` viene chiamato, perché il ridisegno aspetta lo stesso thread.

`Stop` non funziona: il clic resta in coda finché `actionPerformed` non termina, e `actionPerformed` termina solo dopo `-1;-1`, che il server manda solo dopo aver ricevuto `stop`. Il client resta bloccato per sempre. Per questo la lettura va in un thread separato.

← *Torna all'Esercizio 1, pagina 22*

Soluzione dell'Esercizio 2 – La flush dimenticata

Non succede niente: il client non stampa la risposta e resta fermo. La riga `ciao` è rimasta nel buffer del `PrintWriter` e non è partita. Il server aspetta in `nextLine()` una riga che non arriva, e il client aspetta in `nextLine()` la risposta del server: ciascuno aspetta l'altro. Rimettendo `flush()` dopo `println`, la riga parte subito.

← *Torna all'Esercizio 2, pagina 23*

Soluzione dell'Esercizio 3 – Stop e subito Start

Dopo `Stop` il primo `Downloader` è ancora vivo: aspetta le righe rimaste, fino a `-1;-1`. Il nuovo `Start` crea un secondo `Downloader` sullo stesso `Scanner`. Ora due thread leggono la stessa socket, e ogni riga arriva a uno solo dei due, a caso. Il secondo `Downloader` può ricevere il `-1;-1` destinato al primo e fermarsi subito, mentre il primo resta vivo e continua a colorare i pannelli con le bandiere nuove. Al `Stop` successivo, poi, il `-1;-1` ferma uno solo dei lettori rimasti, e lo stato del client non corrisponde più a quello dei thread.

Con `downloaderThread.join()` il ramo `STOP` aspetta che il primo `Downloader` abbia letto `-1;-1` e sia terminato. Solo allora `setButtons` riabilita `Start`, quindi c'è sempre al più un lettore.

← *Torna all'Esercizio 3, pagina 23*

Soluzione dell'Esercizio 4 – Fermare il server

`stop()` chiude il `ServerSocket`: l'`accept()` bloccata lancia una `IOException`, il `catch` esegue `break`, e `run` termina.

Poi `stop()` chiama `stop()` su ogni `ClientThread`. Se il client sta ricevendo, il suo `SenderThread` viene fermato e manda `-1;-1`; poi la socket viene chiusa. Nel client, il `Downloader` legge `-1;-1` e termina, oppure, se la socket si chiude prima, la sua `nextLine()` lancia una `NoSuchElementException` e il thread termina lo stesso.

← *Torna all'Esercizio 4, pagina 23*

Soluzione dell'Esercizio 5 – Appello del 06/09/2023, Canale 1, Prof. Paolo Liberatore, esercizio 1, adattato

Lo schema è quello della bandiera: quattro classi, con la lettura in un thread separato. `Main` crea la finestra:

```
public class Main {
    public static void main(String[] args) {
        FileFrame frame = new FileFrame();
        frame.setSize(800, 500);
        frame.setVisible(true);
    }
}
```

La finestra mette indirizzo, porta, `Connect` e `Disconnect` a nord, le tre aree in una griglia 1×3 al centro, il totale e gli altri pulsanti a sud:

```
import java.awt.BorderLayout;
import java.awt.GridLayout;
import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JPanel;
import javax.swing.JScrollPane;
import javax.swing.JTextArea;
import javax.swing.JTextField;

public class FileFrame extends JFrame {
    private final JButton connect = new JButton("Connect");
    private final JButton disconnect = new JButton("Disconnect");
    private final JButton start = new JButton("Start");
    private final JButton stop = new JButton("Stop");
    private final JButton clear = new JButton("Clear");

    public FileFrame() {
```

```
        super("Nome Cognome 1234567");
        JTextField indirizzo = new JTextField("127.0.0.1", 10);
        JTextField porta = new JTextField("4400", 5);
        JTextArea log = area(), pdf = area(), mp3 = area();
        JTextField totale = new JTextField(10);
        totale.setEditable(false);

        FileListener list = new FileListener(this, indirizzo, porta,
                log, pdf, mp3, totale);
        JPanel nord = new JPanel();
        nord.add(new JLabel("Indirizzo"));
        nord.add(indirizzo);
        nord.add(new JLabel("Porta"));
        nord.add(porta);
        aggiungi(nord, connect, FileListener.CONNECT, list);
        aggiungi(nord, disconnect, FileListener.DISCONNECT, list);

        JPanel centro = new JPanel(new GridLayout(1, 3));
        centro.add(colonna("Log", log));
        centro.add(colonna(".pdf", pdf));
        centro.add(colonna(".mp3", mp3));

        JPanel sud = new JPanel();
        sud.add(new JLabel("Totale"));
        sud.add(totale);
        aggiungi(sud, start, FileListener.START, list);
        aggiungi(sud, stop, FileListener.STOP, list);
        aggiungi(sud, clear, FileListener.CLEAR, list);

        this.getContentPane().add(nord, BorderLayout.NORTH);
        this.getContentPane().add(centro, BorderLayout.CENTER);
        this.getContentPane().add(sud, BorderLayout.SOUTH);
        this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        this.setButtons(false, false);
    }

    private JTextArea area() {
        JTextArea a = new JTextArea();
        a.setEditable(false);
        return a;
    }

    private JPanel colonna(String titolo, JTextArea a) {
        JPanel p = new JPanel(new BorderLayout());
```

```
        p.add(new JLabel(titolo), BorderLayout.NORTH);
        p.add(new JScrollPane(a), BorderLayout.CENTER);
        return p;
    }

    private void aggiungi(JPanel p, JButton b, String comando,
    FileListener list) {
        b.setActionCommand(comando);
        b.addActionListener(list);
        p.add(b);
    }

    public void setButtons(boolean connected, boolean transmitting) {
        connect.setEnabled(!connected);
        disconnect.setEnabled(connected && !transmitting);
        start.setEnabled(connected && !transmitting);
        clear.setEnabled(connected && !transmitting);
        stop.setEnabled(connected && transmitting);
    }
}
```

Qui l'ascoltatore riceve la finestra nel costruttore. Su `Stop` non cambia i pulsanti: la fine della trasmissione la segnala il server con `END` o `INTERRUPTED`, e allora il lettore chiama `fineDownload`.

```
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.io.IOException;
import java.io.PrintWriter;
import java.net.Socket;
import java.util.Scanner;
import javax.swing.JOptionPane;
import javax.swing.JTextArea;
import javax.swing.JTextField;

public class FileListener implements ActionListener {
    public static final String CONNECT = "connect";
    public static final String DISCONNECT = "disconnect";
    public static final String START = "start";
    public static final String STOP = "stop";
    public static final String CLEAR = "clear";

    private final FileFrame frame;
    private final JTextField indirizzo, porta, totale;
    private final JTextArea log, pdf, mp3;
```

```
    private Socket socket;
    private PrintWriter netPw;
    private Scanner scan;

    public FileListener(FileFrame frame, JTextField indirizzo, JTextField
porta,
            JTextArea log, JTextArea pdf, JTextArea mp3, JTextField totale
) {
        this.frame = frame;
        this.indirizzo = indirizzo;
        this.porta = porta;
        this.log = log;
        this.pdf = pdf;
        this.mp3 = mp3;
        this.totale = totale;
    }

    public void actionPerformed(ActionEvent e) {
        String command = e.getActionCommand();
        if (command.equals(CONNECT)) {
            try {
                socket = new Socket(indirizzo.getText(),
                        Integer.parseInt(porta.getText()));
                netPw = new PrintWriter(socket.getOutputStream());
                scan = new Scanner(socket.getInputStream());
            } catch (IOException | NumberFormatException ex) {
                JOptionPane.showMessageDialog(null,
                        "Impossibile connettersi al server");
                return;
            }
            frame.setButtons(true, false);
        } else if (command.equals(START)) {
            pulisci();
            Downloader d = new Downloader(scan, this, log, pdf, mp3,
totale);
            new Thread(d).start();
            invia(START);
            frame.setButtons(true, true);
        } else if (command.equals(STOP)) {
            invia(STOP); // i pulsanti tornano attivi alla risposta
        } else if (command.equals(DISCONNECT)) {
            invia(DISCONNECT);
            chiudi();
        } else if (command.equals(CLEAR)) {
```

```
            pulisci();
        }
    }

    private void invia(String s) {
        netPw.println(s);
        netPw.flush();
    }

    private void pulisci() {
        log.setText("");
        pdf.setText("");
        mp3.setText("");
        totale.setText("");
    }

    public void fineDownload() {
        frame.setButtons(true, false);
    }

    public void chiudi() {
        scan.close();
        netPw.close();
        try {
            socket.close();
        } catch (IOException ex) {
            System.err.println(ex.getMessage());
        }
        frame.setButtons(false, false);
    }
}
```

Il lettore divide ogni riga sui due punti: `campi[0]` è il tipo, `campi[1]` il nome con il formato, `campi[2]` la dimensione con `KB`. Il numero è la parte di `campi[2]` prima dello spazio.

```
import java.util.Scanner;
import javax.swing.JTextArea;
import javax.swing.JTextField;

public class Downloader implements Runnable {
    private final Scanner scan;
    private final FileListener list;
    private final JTextArea log, pdf, mp3;
    private final JTextField totale;
```

```
    public Downloader(Scanner scan, FileListener list, JTextArea log,
            JTextArea pdf, JTextArea mp3, JTextField totale) {
        this.scan = scan;
        this.list = list;
        this.log = log;
        this.pdf = pdf;
        this.mp3 = mp3;
        this.totale = totale;
    }

    public void run() {
        double somma = 0;
        while (true) {
            String line = scan.nextLine();
            if (line.equals("END") || line.equals("INTERRUPTED")) {
                list.fineDownload();
                return;
            }
            if (line.equals("ERROR")) {
                list.chiudi();
                return;
            }
            String[] campi = line.split(":"); // tipo, nome, dimensione
            log.append(campi[1] + " " + campi[2] + "\n");
            if (campi[0].equals("PDF")) {
                pdf.append(campi[1] + "\n");
            } else if (campi[0].equals("MP3")) {
                mp3.append(campi[1] + "\n");
            }
            somma += Double.parseDouble(campi[2].split(" ")[0]);
            totale.setText(String.format("%.2f KB", somma));
        }
    }
}
```

Durante il download `Disconnect` è disabilitato, quindi nessuno chiude lo `Scanner` mentre il lettore lo usa. Dopo `END` o `INTERRUPTED` il lettore termina, e il `Start` successivo ne crea uno nuovo: c'è sempre al più un lettore. `String.format("%.2f KB", somma)` scrive il totale con due cifre decimali.

← *Torna all'Esercizio 5, pagina 23*