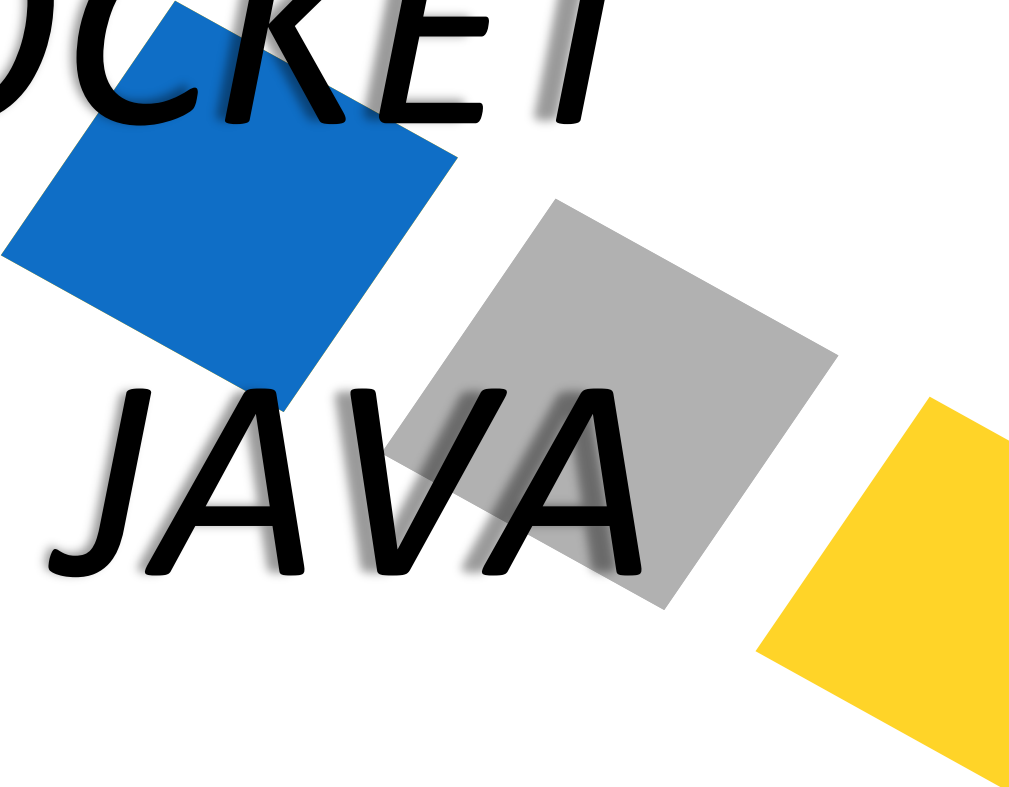

SOCKET *IN JAVA*



Programmazione dei socket in Java

Programmazione dei socket in Java	2
1. Socket TCP e stream di caratteri Unicode	Errore. Il segnalibro non è definito.
Esempio n. 1.1 (Messaggio a singola linea)	Errore. Il segnalibro non è definito.
Esempio n. 1.2 (Messaggio multilinea)	Errore. Il segnalibro non è definito.
2. Socket TCP e stream di byte	Errore. Il segnalibro non è definito.
Esempio n. 2.1 (Messaggio a struttura binaria)	Errore. Il segnalibro non è definito.
Esempio n. 2.2 (Messaggi composti da oggetti serializzabili)	Errore. Il segnalibro non è definito.
3. Multiserver TCP	Errore. Il segnalibro non è definito.
Esempio n. 3.1 (Server multithreaded)	Errore. Il segnalibro non è definito.
4. Socket UDP	3
Esempio n. 4.1 (Datagramma binario)	3
Esempio n. 4.2 (Multicast)	6
Appendice	Errore. Il segnalibro non è definito.
A1 – Generatore di numeri casuali distinti	Errore. Il segnalibro non è definito. A2
– Deployment di un'applicazione Java su server Linux	Errore. Il segnalibro non è definito.
A2.1 – Installazione del software Java Runtime Environment (JRE)	Errore. Il segnalibro non è definito.
A2.2 – Memorizzazione dell'applicazione in formato JAR	Errore. Il segnalibro non è definito.
A2.3 – Trasferimento dell'applicazione sul server	Errore. Il segnalibro non è definito.
Applicazioni proposte	Errore. Il segnalibro non è definito.

Introduzione

La comunicazione tra due o più computer ha sempre rivestito un ruolo di primaria importanza nell'informatica.

Java offre una serie di **classi** semplici e, allo stesso tempo, complete per consentire la scrittura di applicazioni **client-server** dai più svariati utilizzi, attraverso l'utilizzo dei **Socket**.

Cosa sono i Socket?

Possiamo considerare i socket come delle prese (una per ogni computer) interconnesse tra loro attraverso un ipotetico cavo in cui passa il flusso di dati che i computer si scambiano.

Un esempio che rende bene l'idea è quello di **pensare ai socket come alle prese telefoniche** presenti ai due capi durante una conversazione telefonica. Le due persone che colloquiano al telefono comunicano attraverso le rispettive prese. La conversazione, in tal caso, non finirà finché non verrà chiusa la cornetta (fino ad allora la linea risulterà occupata).

I protocolli coinvolti nell'implementazione dei Socket sono, fondamentalmente, 2:

- **TCP** (Transfer Control Protocol)
- **UDP** (User Datagram Protocol)

Per la comunicazione in rete, Java utilizza il **modello a stream**. Un socket può mantenere due tipi di stream: **uno di input ed uno di output**. Dal punto di vista software, ciò che avviene è che un processo invia dei dati ad un altro processo attraverso la rete, **scrivendo sullo stream di output** associato ad un socket. Un altro processo accede ai dati scritti in precedenza **leggendo dallo stream di input del socket stesso**.

Per far sì che una tale comunicazione possa avvenire con successo, è necessario che uno dei due computer (il server) si metta in attesa di una chiamata mentre l'altro (il client) tenti di comunicare con il primo.

Nella realtà, un server che sia in grado di comunicare con un solo client per volta potrebbe essere poco utile. Pertanto, utilizzando i vantaggi offerti dai thread (vedremo come) sarà possibile estendere tale scenario ed ottenere un server in grado di interagire contemporaneamente con più client.

La potenza della classe `URLConnection` risiede nel fatto che essa consente la **gestione del colloquio client-server ad alto livello**, senza che il programmatore debba minimamente preoccuparsi dei dettagli sui socket che ne consentono l'utilizzo.

Vediamo un esempio su come può essere utilizzata tale classe, simulando il comportamento di un comune browser che richieda ad un web server una particolare pagina html:

Simulare un client HTTP

```
import java.io.BufferedReader;
import java.io.PrintWriter;
import java.io.IOException;
import java.io.InputStreamReader;
import java.net.*;

public class URLClient{
```

```
private String strURL;
public URLClient(String strURL) {
    this.strURL = strURL;
}
public String retrievePage() {
    StringBuffer document = new StringBuffer();
    try {
        URL url = new URL(strURL);
        URLConnection conn = url.openConnection();
        BufferedReader reader = new BufferedReader(new
InputStreamReader(conn.getInputStream()));
        String line = null;
        while ((line = reader.readLine()) != null) document.append(line +
"n");
        reader.close();
    } catch (MalformedURLException e) {
        System.out.println("MalformedURLException durante la connessione");
    } catch (IOException e) {
        System.out.println("IOException durante la connessione");
    }
    return document.toString();
}

public static void main(String[] args) {
    URLClient client = new
URLClient("http://www.iisvittorioveneto.edu.it/2018/");
    String webPage = client.retrievePage();
    try {
        FileWriter out = new FileWriter("iisvv.html");
        out.write(webPage);
        out.close();
    } catch (Exception ex) {
        ex.printStackTrace();
    }
}
```

Se si prova ad eseguire questo semplice programma di esempio e si apre successivamente sul proprio browser il file iisvv.html (creato dal programma stesso nella stessa directory in cui lo si è mandato in esecuzione) si potrà osservare che il risultato ottenuto altro non è che la pagina iniziale del nostro sito istituzionale.

Analizzando il codice precedente, si può notare come la classe **URLClient** utilizza un costruttore parametrizzato (che prende in input una URL) e definisce l'implementazione del metodo `retrievePage()`.

Tale metodo, in poche righe di codice, effettua la connessione al web server e ricava il contenuto di una pagina html (che nel nostro caso è, appunto, la pagina iniziale del nostro sito istituzionale).

Il tutto è ottenuto semplicemente aprendo un socket verso la URL di destinazione, grazie all'istruzione `url.openConnection()` e salvando il contenuto della pagina html (ricavata attraverso l'istruzione `conn.getInputStream()`) su un oggetto di classe `BufferedReader`.

La classe `BufferedReader` è una classe cosiddetta wrapper che ci consente di andare a leggere il contenuto dello stream ricavato semplicemente utilizzando una serie di chiamate successive al metodo `readLine()` (ovvero riga per riga).

Le basi della comunicazione Socket

Prima di vedere nel dettaglio come utilizzare le classi Socket e ServerSocket, è importante chiarire alcuni concetti fondamentali.

Utilizziamo ancora l'esempio della chiamata telefonica per rendere più facili i concetti.

Per avviare una conversazione telefonica, si sa che è fondamentale che uno dei due interlocutori conosca il numero dell'altro. In particolare, se prendiamo spunto dai numeri telefonici delle grandi aziende, sappiamo che esiste un numero base che corrisponde, ad esempio, al centralino al quale è possibile accodare altre cifre per effettuare delle chiamate direttamente ad uffici specifici.

Ovvero, vengono utilizzate delle estensioni al numero iniziale.

Se proviamo a riportare gli stessi concetti nella comunicazione in rete (e quindi sui socket), potremo dire che un computer (il client) dovrà conoscere l'indirizzo IP del computer remoto (il server) specificando, inoltre, una particolare estensione definita **Numero di Porta (port number)**, che per semplicità viene spesso chiamata semplicemente porta.

I Port Numbers

Nel protocollo TCP/IP, i port numbers rappresentano numeri a 16 bit il cui valore può variare tra 0 e 65535. Nella pratica, però, non tutti questi valori possono essere utilizzati a proprio piacimento. Infatti, i numeri di porta compresi tra 0 e 1024 sono riservati a particolari servizi come telnet, ftp, SMTP, POP3 e HTTP e possono, pertanto, essere utilizzati soltanto in caso di interfacciamento con tali servizi.

Esiste, in particolare, un'autorità che ha il compito di assegnare queste porte a determinati servizi: la Internet Assigned Numbers Authority (IANA).

Le porte utilizzabili dagli utenti sono quelle i cui valori variano tra 1024 e 49151. Infine, sui port numbers compresi tra 49152 e 65535 non viene applicato alcun controllo. Esse sono definite private ports.

È chiaro che sia il client sia il server devono stabilire una sorta di accordo sulla porta da utilizzare per la comunicazione. Se il client proverà ad utilizzare l'indirizzo IP del server fornendo, però, una porta errata, non si stabilirà alcuna connessione tra le due macchine.

Le classi Socket e ServerSocket

Le classi Socket e ServerSocket sono le due classi che intervengono nell'implementazione di una comunicazione client-server basata sui socket. La prima si occupa della gestione del client che effettua la richiesta mentre la seconda fornisce le funzionalità necessarie a creare un server che stia in ascolto su una particolare porta.

I passi necessari a stabilire una connessione possono essere così riassunti:

1. Il server sceglie una porta su cui mettersi in ascolto. Quando il client tenterà di aprire una connessione, il server si limiterà a richiamare il metodo `accept()` della classe `ServerSocket` ottenendo, a sua volta, un riferimento al canale di comunicazione instaurato con il client.

2. Il client apre una connessione con il server costruendo un oggetto di tipo `Socket` a cui verranno passati due parametri: l'indirizzo del server e la porta su cui il server stesso è in ascolto.
3. Instaurata la connessione, le due macchine (server e client), potranno comunicare semplicemente avvalendosi del modello a stream messo a disposizione dalle classi `InputStream` e `OutputStream`.

Vediamo un esempio pratico su come implementare un server ed un client che comunichino via socket:

Un Server TCP/IP

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

public class SimpleServer
{
    private int port;
    private ServerSocket server;
    public SimpleServer (int port) {
        this.port = port;
        if(!startServer())
            System.err.println("Errore durante la creazione del
Server");
    }

    private boolean startServer() {
        try{
            server = new ServerSocket(port);
        }
        catch (IOException ex){
            ex.printStackTrace();
            return false;
        }
        System.out.println("Server creato con successo!");
        return true;
    }

    public void runServer(){
        while (true) {
            try {
                // Il server resta in attesa di una richiesta
                System.out.println("Server in attesa di
richieste...");

                Socket s1 = server.accept();
                System.out.println("Un client si e' connesso...");

                // Ricava lo stream di output associate al socket
                // e definisce una classe wrapper di tipo
                // BufferedWriter per semplificare le operazioni
                // di scrittura
                OutputStream slout = s1.getOutputStream();
                BufferedWriter bw = new BufferedWriter(new
OutputStreamWriter(slout));
```

```
        // Il server invia la risposta al client
        bw.write("Benvenuto sul server!\n");
        // Chiude lo stream di output e la connessione
        bw.close();
        s1.close();
        System.out.println("Chiusura connessione
effettuata");
    }
    catch (IOException ex) {
        ex.printStackTrace();
    }
}

public static void main (String args[]){
    // Crea un oggetto di tipo SimpleServer in ascolto
    // sulla porta 7777
    SimpleServer ss = new SimpleServer(7777);
    ss.runServer();
}
```

Il codice che segue, invece, implementa un semplice client TCP/IP:

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

public class SimpleClient
{
    public static void main(String args[]){
        try{
            // Apre una connessione verso un server in ascolto
            // sulla porta 7777. In questo caso utilizziamo localhost
            // che corrisponde all'indirizzo IP 127.0.0.1
            System.out.println("Apertura connessione...");
            Socket s1 = new Socket ("127.0.0.1", 7777);

            // Ricava lo stream di input dal socket s1
            // ed utilizza un oggetto wrapper di classe BufferedReader
            // per semplificare le operazioni di lettura
            InputStream is = s1.getInputStream();
            BufferedReader dis = new BufferedReader(
                new InputStreamReader(is));

            // Legge l'input e lo visualizza sullo schermo
            System.out.println("Risposta del server: " + dis.readLine());

            // Al termine, chiude lo stream di comunicazione e il socket.
            dis.close();
            s1.close();
            System.out.println("Chiusura connessione effettuata");
        }
        catch (ConnectException connExc){
            System.err.println("Errore nella connessione ");
        }
        catch (IOException ex){
```

```
        ex.printStackTrace();  
    }  
}
```

Se si prova ad eseguire su due shell separate, il server ed il client (facendo attenzione a far prima partire il server!) si potrà notare come avviene l'interazione tra i due componenti.

Naturalmente, l'esempio precedente è davvero molto semplificato e serve a rendere soltanto l'idea di come funziona lo scambio di informazioni via Socket tra due pc.

La grande limitazione, in questo caso, è rappresentata dal fatto che il server è in grado di accettare le chiamate dei client in modo sequenziale, ovvero un solo client alla volta.

Con l'ausilio dei thread possiamo migliorare le cose e far sì che il server sia sempre in grado di accettare una chiamata, a prescindere dal fatto che in quel momento siano o meno connessi altri client.

Vediamo come:

Server TCP/IP MultiThread

```
import java.net.*;  
import java.io.*;  
  
public class SimpleServer {  
    private int port;  
    private ServerSocket server;  
    private Socket client;  
  
    public SimpleServer(int port) {  
        this.port = port;  
        if (!startServer())  
            System.err.println("Errore durante la creazione del Server");  
    }  
  
    private boolean startServer() {  
        try {  
            server = new ServerSocket(port);  
        } catch (IOException ex) {  
            ex.printStackTrace();  
            return false;  
        }  
        System.out.println("Server creato con successo!");  
        return true;  
    }  
  
    public void runServer() {  
        while (true) {  
            try {
```



```
        // Il server resta in attesa di una richiesta
        System.out.println("Server in attesa di richieste...");
        client = server.accept();
        System.out.println("Un client si e' connesso...");
        ParallelServer pServer = new ParallelServer(client);
        Thread t = new Thread(pServer);
        t.start();
    } catch (IOException ex) {
        ex.printStackTrace();
    }
}
}
```

In questo caso, andiamo ad utilizzare una inner-class ausiliaria **ParallelServer**, che si occupa di parallelizzare le operazioni svolte dal server e che viene passata come input ad un **thread**.

Il risultato ottenuto permetterà al server di gestire contemporaneamente connessioni con più client.

```
import java.io.BufferedWriter;
import java.io.IOException;
import java.io.OutputStream;
import java.io.OutputStreamWriter;
import java.net.Socket;

public class ParallelServer implements Runnable{
    private Socket client;
    public ParallelServer (Socket client){
        this.client = client;
    }

    public void run() {
        try{
            // Ricava lo stream di output associate al socket
            // e definisce una classe wrapper di tipo
            // BufferedWriter per semplificare le operazioni
            // di scrittura
            OutputStream slout = client.getOutputStream();
            BufferedWriter bw = new BufferedWriter(
                new OutputStreamWriter(slout));

            // Il server invia la risposta al client
            bw.write("Benvenuto sul server Multi-Client!");

            // Chiude lo stream di output e la connessione
            bw.close();
            client.close();
            System.out.println("Chiusura connessione effettuata");
        } catch (IOException ex){
            ex.printStackTrace();
        } catch (Exception e){
            e.printStackTrace();
        }
    }
}
```

```
public static void main (String args[]){
    SimpleServer ss = new SimpleServer(7777);
    ss.runServer();
}
```

1. Socket TCP e stream di caratteri Unicode

Due applicazioni si scambiano, in modo affidabile, messaggi costituiti da una o più linee di testo terminanti con il carattere *newline*.

Esempio n. 1.1 (Messaggio a singola linea)

Scrivere un'applicazione client/server in cui il client invia al server una frase e il server la restituisce al client convertita in maiuscolo.

Caratteristiche delle applicazioni

- **Server:** supporta l'accesso da parte di più client ma serve un client alla volta (elaborazione sequenziale); elabora una richiesta composta da una sola linea di testo in formato UTF-8 (Unicode); restituisce una risposta formata da una sola linea di testo UTF-8.
- **Client:** Invia una richiesta composta da una sola linea di testo UTF-8 delimitata dal carattere *newline*; elabora una risposta a singola linea di testo UTF-8.

Server (codifica Java)

```
public class ServerMaiuscolo {    final
static int portaServer = 4000;    public
static void main(String[] args) {

    try (ServerSocket server = new ServerSocket(portaServer)) {
System.out.format("Server in ascolto su: %s\n", server.getLocalSocketAddress());

        // Il server accetta e serve un client alla volta
while (true) {
    try (Socket client = server.accept()) {
        String rem = client.getRemoteSocketAddress().toString();

System.out.format("Client (remoto): %s\n", rem);
        comunica(client);

    } catch (IOException e) {
        System.err.println(String.format("Errore durante la comunicazione
con il client: %s", e.getMessage()));
    }
}
    } catch (IOException e) {
        System.err.println(String.format("Errore server: %s", e.getMessage()));
    }
}
```

```

    }
}

private static void comunica(Socket sck) throws IOException {
    BufferedReader in = new BufferedReader( new
        InputStreamReader(sck.getInputStream(), "UTF-8"));
    PrintWriter out = new PrintWriter(
        new OutputStreamWriter(sck.getOutputStream(), "UTF-8"), true);
    System.out.println("In attesa di ricevere informazioni dal client...");
    String inStr = in.readLine();
    System.out.format("Ricevuto dal client: %s\n", inStr);
    String outStr = elabora(inStr);
    out.println(outStr);
    System.out.format("Inviato al client: %s\n", outStr);
}

private static String elabora(String s) {
return s.toUpperCase();
}
}

```

Client (codifica Java)

```

public class ClientMaiuscolo {
    final static String nomeServer = "localhost";
    final static int portaServer = 4000;

    public static void main(String[] args) {
        System.out.println("Connessione al server in corso...");
        try (Socket sck = new Socket(nomeServer, portaServer)) {

            String rem = sck.getRemoteSocketAddress().toString();
            String loc = sck.getLocalSocketAddress().toString();
            System.out.format("Server (remoto): %s\n", rem);
            System.out.format("Client (client): %s\n", loc);
            comunica(sck);

        } catch (UnknownHostException e) {
            System.err.format("Nome di server non valido: %s\n", e.getMessage());
        } catch (IOException e) {
            System.err.format("Errore durante la comunicazione con il server: %s\n",
                e.getMessage());
        }
    }

    private static void comunica(Socket sck) throws IOException {
        BufferedReader in = new BufferedReader( new
            InputStreamReader(sck.getInputStream(), "UTF-8"));
        PrintWriter out = new PrintWriter( new
            OutputStreamWriter(sck.getOutputStream(), "UTF-8"), true);

        Scanner s = new Scanner(System.in, "UTF-8");
        System.out.print("Frase: ");
    }
}

```

```

        String frase = s.nextLine();
        System.out.format("Invio al server: %s\n", frase);
    out.println(frase);
        System.out.println("In attesa di risposta dal server...");
        String risposta = in.readLine();
        System.out.format("Risposta dal server: %s\n", risposta);
    }
}

```

Esempio n. 1.2 (Messaggio multilinea)

Scrivere un'applicazione client/server in cui il client invia al server una sequenza di linee di testo, ciascuna contenente una parola (la sequenza è conclusa con l'invio di una riga vuota). Il server restituisce un messaggio multilinea contenente le sole parole palindromo (una parola per linea, anche il messaggio di risposta termina con una riga vuota).

Caratteristiche delle applicazioni

- **Server:** supporta l'accesso da parte di più client ma serve un client alla volta (elaborazione sequenziale); elabora una richiesta composta da più linee di testo in formato UTF-8 (l'ultima linea è vuota); restituisce una risposta formata da più linee testo UTF-8 (l'ultima linea del messaggio di risposta è vuota).
- **Client:** Invia una richiesta composta da più linee di testo UTF-8 delimitata dal carattere *newline*; elabora una risposta a singola linea di testo UTF-8.

Tutte le linee di testo inviate o ricevute terminano con un carattere di *newline*.

Server (codifica Java)

```

public class ServerPalindrome {    final
static int portaServer = 4000;    public
static void main(String[] args) {

    try (ServerSocket server = new ServerSocket(portaServer)) {
        System.out.format("Server in ascolto su: %s\n",
            server.getLocalSocketAddress());

        // Il server accetta e serve un client alla volta
    while (true) {
        try (Socket client = server.accept()) {
            String rem = client.getRemoteSocketAddress().toString();
            System.out.format("Client (remoto): %s\n", rem);
            comunica(client);
        } catch (IOException e) {
            System.err.println(String.format("Errore durante la comunicazione
                con il client: %s", e.getMessage()));
        }
    }
    } catch (IOException e) {
        System.err.println(String.format("Errore server: %s", e.getMessage()));
    }
}

private static void comunica(Socket sck) throws IOException {

```

```

        BufferedReader in = new BufferedReader( new
            InputStreamReader(sck.getInputStream(), "UTF-8"));
        PrintWriter out = new PrintWriter( new
            OutputStreamWriter(sck.getOutputStream(), "UTF-8"), true);

        ArrayList<String> richiesta = new ArrayList<String>();
        String inStr;
        System.out.println("In attesa di ricevere informazioni dal client...");
    do {
        inStr = in.readLine().trim();           // Rimuove eventuali spazi superflui
        System.out.format("Ricevuto dal client: %s\n", inStr);
    if (inStr.isEmpty() == false)
        // Si aggiungono solo le stringhe non vuote
        richiesta.add(inStr);
    } while (inStr.isEmpty()
    == false);

        ArrayList<String> risposta = elabora(richiesta);
        for(String outStr : risposta) {
            System.out.format("Inviato al client: %s\n", outStr);
        out.println(outStr);
        }
        out.println();           // Conclude la risposta con una linea vuota
    }

    private static ArrayList<String> elabora(ArrayList<String> richiesta) {
        ArrayList<String> ris = new ArrayList<String>();
        for(String s : richiesta) {
            if
            (stringaPalindroma(s) == true) {
                ris.add(s);
            }
        }
        return ris;
    }

    private static boolean stringaPalindroma(String s) {
        StringBuilder sb = new StringBuilder(s);

        // Inverte la posizione dei caratteri contenuti in sb e restituisce
        // la nuova sequenza sotto forma di stringa

        String s2 = sb.reverse().toString();

        // Restituisce vero se e solo se le stringhe s e s2 sono uguali
        // (confronto case insensitive).
        return (s.compareToIgnoreCase(s2) == 0);
    }
}

```

Client (codifica Java)

```

public class ClientPalindrome {

```

```

    final static String nomeServer = "localhost";
    final static int portaServer = 4000;

    public static void main(String[] args) {
        System.out.println("Connessione al server in corso...");
        try (Socket sck = new Socket(nomeServer, portaServer)) {

            String rem = sck.getRemoteSocketAddress().toString();
            String loc = sck.getLocalSocketAddress().toString();

            System.out.format("Server (remoto): %s\n", rem);
            System.out.format("Client (client): %s\n", loc);          comunica(sck);

            } catch (UnknownHostException e) {
                System.err.format("Nome di server non valido: %s\n", e.getMessage());
            } catch (IOException e) {
                System.err.format("Errore durante la comunicazione con il server: %s\n",
                    e.getMessage());
            }
        }

        private static void comunica(Socket sck) throws IOException {
            BufferedReader in = new BufferedReader( new
                InputStreamReader(sck.getInputStream(), "UTF-8"));
            PrintWriter out = new PrintWriter( new
                OutputStreamWriter(sck.getOutputStream(), "UTF-8"), true);

            Scanner s = new Scanner(System.in, "UTF-8");
            String parola;

            do {
                System.out.print("Parola (premere il solo tasto INVIO per terminare): ");
                parola = s.nextLine().trim();          if (parola.isEmpty() == false) {
                    System.out.format("Invio al server: %s\n", parola);
                }
                out.println(parola);
            } while (parola.isEmpty() == false);

            out.println(parola);    // Invio di una linea vuota

            System.out.println("Le parole palindrome sono:");
            do {
                parola = in.readLine();
                if (parola.isEmpty() == false) {
                    System.out.println(parola);
                }
            } while (parola.isEmpty() == false);
        }
    }

```

2. Socket TCP e stream di byte

Due applicazioni si scambiano, in modo affidabile, messaggi codificati in binario.

Esempio n. 2.1 (Messaggio a struttura binaria)

Scrivere un'applicazione client/server per elaborare una semplice statistica sui lati di un dato poligono: il client invia il numero intero N di lati del poligono e la lunghezza (di tipo reale) di ciascuno dei lati; il server restituisce due numeri reali, corrispondenti alle lunghezze dei lati più corto e più lungo.

Caratteristiche delle applicazioni

- **Server:** supporta l'accesso da parte di più client ma serve un client alla volta (elaborazione sequenziale); elabora una richiesta in formato binario costituita da un numero di tipo *int* e da N numeri di tipo *double*; restituisce una statistica composta da due dati di tipo *double* (max e min).
- **Client:** Invia una richiesta in formato binario secondo le specifiche indicate e riceve la statistica prodotta dal server.

Server (codifica Java)

```
public class ServerPoligono {    final
static int portaServer = 4500;    public
static void main(String[] args) {

    try (ServerSocket server = new ServerSocket(portaServer)) {
System.out.format("Server in ascolto su: %s\n", server.getLocalSocketAddress());

        // Il server accetta e serve un client alla volta
while (true) {
    try (Socket client = server.accept()) {
        String rem = client.getRemoteSocketAddress().toString();
System.out.format("Client (remoto): %s\n", rem);
        comunica(client);
    } catch (IOException e) {
        System.err.println(String.format("Errore durante la comunicazione
        con il client: %s", e.getMessage()));
    }
} catch (IOException e) {
    System.err.println(String.format("Errore server: %s", e.getMessage()));
}
}

private static void comunica(Socket sck) throws IOException {
    DataInputStream in = new DataInputStream( new
        BufferedInputStream(sck.getInputStream()));
    DataOutputStream out = new DataOutputStream( new
        BufferedOutputStream(sck.getOutputStream()));

    int numLati;
    ArrayList<Double> lati = new ArrayList<Double>();
    System.out.println("In attesa di ricevere informazioni dal client...");
    numLati = in.readInt();
    System.out.format("Ricevo dal client un poligono di %d lati:\n", numLati);
    for (int i = 0; i < numLati; i++) {
        Double lato = in.readDouble();
        lati.add(lato);
        System.out.format(" %.2f", lato);
    }
}
```

```

        System.out.println();
        ArrayList<Double> statistica = elabora(lati);
        double min = statistica.get(0);        double max =
        statistica.get(1);        out.writeDouble(min);
        out.writeDouble(max);
        out.flush();    // Svuota il buffer
        System.out.format("Inviata al client la statistica (Min: %.2f    Max: %.2f)%n",
            min, max);
    }

    private static ArrayList<Double> elabora(ArrayList<Double> lati) {
        ArrayList<Double> stat = new ArrayList<Double>();
        stat.add(Collections.min(lati));
        stat.add(Collections.max(lati));        return stat;
    }
}

```

Client (codifica Java)

```

public class ClientPoligono {
    final static String nomeServer = "localhost";
    final static int portaServer = 4500;

    public static void main(String[] args) {
        System.out.println("Connessione al server in corso...");
        try (Socket sck = new Socket(nomeServer, portaServer)) {

            String rem = sck.getRemoteSocketAddress().toString();
            String loc = sck.getLocalSocketAddress().toString();
            System.out.format("Server (remoto): %s%n", rem);
            System.out.format("Client (client): %s%n", loc);
            comunica(sck);

        } catch (UnknownHostException e) {
            System.err.format("Nome di server non valido: %s%n", e.getMessage());
        } catch (IOException e) {
            System.err.format("Errore durante la comunicazione con il server: %s%n",
                e.getMessage());
        }
    }

    private static void comunica(Socket sck) throws IOException {
        DataInputStream in = new DataInputStream( new
            BufferedInputStream(sck.getInputStream()));
        DataOutputStream out = new DataOutputStream(

```

```

        new BufferedOutputStream(sck.getOutputStream()));

    Scanner s = new Scanner(System.in);
    int numLati;
    do {
        System.out.print("Numero di lati del poligono: ");
        numLati = s.nextInt();
    }
    while (numLati < 3);
    out.writeInt(numLati);
    for (int i = 1; i <= numLati; i++) {
        double lato;
        do {
            System.out.format("lunghezza del lato n. %d: ", i);
            lato = s.nextDouble();
        }
        while (lato <= 0);
        out.writeDouble(lato);
    }
    out.flush();

    double min = in.readDouble();
    double max = in.readDouble();
    System.out.format("Statistica ricevuta dal server: min = %.2f  max: %.2f\n",
        min, max);
}
}

```

Esempio n. 2.2 (Messaggi composti da oggetti serializzabili)

Scrivere un'applicazione client/server per eseguire una ricerca nel database degli impiegati di un'azienda: il client invia un oggetto contenente il cognome e il nome dell'impiegato; il server restituisce un altro oggetto in cui sono inseriti i risultati della ricerca (cognome, nome, matricola, stipendio).

Caratteristiche delle applicazioni

- Il client e il server condividono la definizione di due classi *Ricerca* e *Risultato*. Queste classi definiscono la struttura dei messaggi scambiati e devono implementare l'interfaccia *Serializable*
- **Server:** supporta l'accesso da parte di più client ma serve un client alla volta (elaborazione sequenziale); riceve dal client un oggetto di tipo *Ricerca* da cui ricava i criteri di ricerca; genera e restituisce al client un oggetto di tipo *Risultato* contenente i dati dell'impiegato trovato (oppure *null* se l'impiegato non esiste).
- **Client:** legge da tastiera il cognome e il nome di un impiegato, inserisce i dati in un oggetto di tipo *Ricerca* che invia al server; riceve un oggetto di tipo *Risultato* da cui ricava il risultato della ricerca.

Server (codifica Java)

```

class Ricerca implements Serializable {
    public String cognome;    public String
    nome;

```

```
    public Ricerca(String cognome, String nome) {
this.cognome = cognome;        this.nome = nome;
    }
}

class Risultato implements Serializable {
public String cognome;        public String
nome;        public int matricola;        public
double stipendio;

    public Risultato(String cognome, String nome, int matricola, double stipendio) {
this.cognome = cognome;        this.nome = nome;        this.matricola = matricola;
this.stipendio = stipendio;
    }
}

public class ServerImpiegato {        final
static int portaServer = 5000;        public
static void main(String[] args) {

    try (ServerSocket server = new ServerSocket(portaServer)) {
System.out.format("Server in ascolto su: %s%n", server.getLocalSocketAddress());

        // Il server accetta e serve un client alla volta
while (true) {
            try (Socket client = server.accept()) {
                String rem = client.getRemoteSocketAddress().toString();
System.out.format("Client (remoto): %s%n", rem);
                comunica(client);
            } catch (IOException e) {
                System.err.println(String.format("Errore durante la comunicazione
con il client: %s", e.getMessage()));
            }
        }
    } catch (IOException e) {
        System.err.println(String.format("Errore server: %s", e.getMessage()));
    }
}

private static void comunica(Socket sck) throws IOException {
    ObjectInputStream in = new ObjectInputStream(sck.getInputStream());

    System.out.println("In attesa di ricevere informazioni dal client...");
    Ricerca ric = null;
    try {
        // Deserializzazione: ricevo una sequenza di byte e
        // La trasformo in un oggetto
        ric = (Ricerca) in.readObject();
    }
    catch (ClassNotFoundException e) {
        throw new IOException("Tipo di ricerca non supportata.");
    }

    System.out.format("Ricevuta dal client la ricerca: cognome = '%s' nome =
'%s'%n", ric.cognome, ric.nome);
}
```

```
Risultato ris = elabora(ric);

ObjectOutputStream out = new ObjectOutputStream(sck.getOutputStream());
out.writeObject(ris);
System.out.println("Inviato al client il risultato della ricerca");
}

private static Risultato elabora(Ricerca r) {
Risultato ris;          boolean trovato;

    trovato = r.cognome.toUpperCase().equals("ROSSI") &&
              r.nome.toUpperCase().equals("MARIO");

    if (trovato == true) {
        ris = new Risultato("Rossi", "Mario", 10012, 1598.50);
    } else {
ris = null;
    }

    return ris;
}
```

Client (codifica Java)

```
class Ricerca implements Serializable {
public String cognome;      public String
nome;

    public Ricerca(String cognome, String nome) {
this.cognome = cognome;      this.nome = nome;
    }
}

class Risultato implements Serializable {
public String cognome;      public String
nome;      public int matricola;      public
double stipendio;

    public Risultato(String cognome, String nome, int matricola, double stipendio) {
this.cognome = cognome;      this.nome = nome;
        this.matricola = matricola;
this.stipendio = stipendio;
    }
}

public class ClientImpiegato {
    final static String nomeServer = "localhost";
    final static int portaServer = 5000;

    public static void main(String[] args) {
        System.out.println("Connessione al server in corso...");
try (Socket sck = new Socket(nomeServer, portaServer)) {

            String rem = sck.getRemoteSocketAddress().toString();
```

```

        String loc = sck.getLocalSocketAddress().toString();
        System.out.format("Server (remoto): %s\n", rem);
    System.out.format("Client (client): %s\n", loc);                comunica(sck);

    } catch (UnknownHostException e) {
        System.err.format("Nome di server non valido: %s\n", e.getMessage());
    } catch (IOException e) {
        System.err.format("Errore durante la comunicazione con il server: %s\n",
            e.getMessage());
    }
}

private static void comunica(Socket sck) throws IOException {
    ObjectOutputStream out = new ObjectOutputStream(sck.getOutputStream());

    Scanner s = new Scanner(System.in);
    System.out.println("Inserire l'impiegato da ricercare.");
    System.out.print("Cognome: ");
    String co = s.nextLine();
    System.out.print("Nome: ");
    String no = s.nextLine();

    Ricerca ric = new Ricerca(co, no);
    out.writeObject(ric);

    System.out.println("In attesa di risposta dal server...");
    Risultato ris = null;

    ObjectInputStream in = new ObjectInputStream(sck.getInputStream());
    try {
        // Deserializzazione: ricevo una sequenza di byte e
        // la trasformo in un oggetto
        ris = (Risultato) in.readObject();
    } catch (ClassNotFoundException e) {
        throw new IOException("Tipo di risultato non supportato.");
    }
    System.out.println("Risultato della ricerca:");
    if (ris != null) {
        System.out.format("Cognome: %s\n", ris.cognome);
        System.out.format("Nome: %s\n", ris.nome);
        System.out.format("Matricola: %d\n", ris.matricola);
        System.out.format("Stipendio: %.2f Euro\n", ris.stipendio);
    }
    else {
        System.out.println("Impiegato non presente in archivio");
    }
}

```

4. Socket UDP

Due applicazioni si scambiano, in modo rapido ma non necessariamente affidabile, dati codificati in binario.

Esempio n. 4.1 (Datagramma binario)

Scrivere un'applicazione C/S in cui il client invia un nome di persona e il server risponde con una frase di saluto.

Caratteristiche delle applicazioni

- **Server:** riceve dal client un datagramma UDP contenente i dati della codifica UTF-8 di una stringa; trasforma la sequenza in una stringa da cui ricava il nome della persona; costruisce la frase di saluto, la converte in una sequenza di byte e la inserisce in un nuovo datagramma; invia il datagramma prodotto al client.
- **Client:** richiede l'inserimento da tastiera di un nome (stringa UTF-8); genera e invia al server un datagramma contenente il nome; riceve dal server un nuovo datagramma da cui estrae la frase di saluto; stampa la frase.

Server

Questo codice rappresenta l'esempio di un server che utilizza una socket basata su protocollo UDP (in Java).

La classe principale è **ServerSaluti**, che definisce la variabile d'istanza **portaServer** con valore **6789**. Il metodo "main" è il punto di ingresso del programma e crea un oggetto "DatagramSocket" sulla porta specificata da "portaServer".

Il server entra in un ciclo infinito in cui attende pacchetti in entrata chiamando il metodo "receive" sull'oggetto "DatagramSocket".

Quando un pacchetto viene ricevuto, viene creato un oggetto "DatagramPacket" che contiene i dati ricevuti. Il nome inviato dal client viene estratto dal pacchetto ed elaborato **[nome.trim().toUpperCase()]** trasformando il testo in maiuscolo e privandolo di spazi vuoti.

Successivamente, viene creato un messaggio di saluto basato sul nome ricevuto e inviato come risposta al Client (altra classe in Java del progetto) utilizzando il metodo "send" sull'oggetto "DatagramSocket". **Il server continua ad attendere pacchetti in entrata e a inviare saluti ai client.**

In caso di **eccezione**, viene stampato un messaggio di errore utilizzando il metodo "printStackTrace" dell'eccezione.

Codifica Java

```
public class ServerSaluti {    final
static int portaServer = 6789; public
static void main(String[] args) { byte[]
bufferIn = new byte[1024]; byte[]
bufferOut;
    try (DatagramSocket sck = new DatagramSocket(portaServer)) {
        System.out.format("Server in ascolto su %s.", sck.getLocalSocketAddress());
        while(true) {
            DatagramPacket pktIn = new DatagramPacket(bufferIn, bufferIn.length);
            sck.receive(pktIn);
            SocketAddress saClient = pktIn.getSocketAddress();
            String nome = new String(pktIn.getData(), 0, pktIn.getLength(), "UTF-8"); nome
            = nome.trim().toUpperCase();
```

```
        System.out.format("Ricevuto dal client %s il nome %s... ", saClient, nome); String
        saluto = String.format("Salve %s, ti auguro una buona giornata.%n", nome);
        bufferOut = saluto.getBytes("UTF-8");
        DatagramPacket pktOut = new DatagramPacket(bufferOut, bufferOut.length, saClient);
        sck.send(pktOut);
        System.out.println("Inviato il saluto.");
    }
} catch (SocketException e) {
    e.printStackTrace();
} catch (IOException e) {
    e.printStackTrace();
}
}
```

Breve spiegazione dei metodi utilizzati nella classe precedente:

1. **DatagramSocket(int portaServer):** Questo è il costruttore della classe "DatagramSocket". Crea un socket UDP sulla porta specificata da "portaServer".
2. **getLocalSocketAddress():** Questo metodo restituisce l'indirizzo locale a cui il socket è associato.
3. **receive(DatagramPacket pktIn):** Questo metodo blocca l'esecuzione del codice fino a quando non viene ricevuto un pacchetto. Quando un pacchetto viene ricevuto, i suoi dati vengono inseriti nell'oggetto "DatagramPacket" passato come argomento.
4. **getSocketAddress():** Questo metodo restituisce l'indirizzo remoto del client che ha inviato il pacchetto.
5. **String(byte[] data, int offset, int length, String charsetName):** Questo è il costruttore della classe "String". Crea una stringa a partire dall'array di byte "data" usando il charset specificato da "charsetName". L'offset specifica la posizione iniziale dei dati da utilizzare e la lunghezza specifica quanti byte devono essere utilizzati.
6. **trim():** Questo metodo restituisce una nuova stringa priva di spazi vuoti all'inizio e alla fine della stringa originale.
7. **toUpperCase():** Questo metodo restituisce una nuova stringa che rappresenta la stringa originale in maiuscolo.
8. **getBytes(String charsetName):** Questo metodo restituisce un array di byte che rappresenta la stringa usando il charset specificato da "charsetName".
9. **DatagramPacket(byte[] bufferOut, int length, SocketAddress saClient):** Questo è il costruttore della classe "DatagramPacket". Crea un pacchetto che contiene i dati specificati da "bufferOut" e che è destinato all'indirizzo specificato da "saClient". La lunghezza specifica quanti byte devono essere inviati.
10. **send(DatagramPacket pktOut):** Questo metodo invia il pacchetto specificato da "pktOut" attraverso il socket.
11. **printStackTrace():** Questo metodo stampa una rappresentazione testuale della pila di chiamate che ha portato all'eccezione. Questo è utile per identificare la causa di un errore durante lo sviluppo e il debug.

Client

Questo codice rappresenta un semplice client che utilizza una connessione UDP (User Datagram Protocol) per inviare il proprio nome al server e ricevere un saluto in risposta.

Il codice inizia definendo due costanti: **nomeServer**, che contiene l'indirizzo IP o il nome **host** del server a cui il client vuole inviare il nome, e la **portaServer**, che specifica la porta del server sulla quale il client invierà i dati.

Nel metodo **main**, viene utilizzato un oggetto Scanner per leggere l'input dell'utente, che rappresenta il nome, e questo viene poi convertito in una stringa.

Successivamente, viene creato un oggetto **DatagramSocket**, che rappresenta una **socket UDP**, per inviare e ricevere dati. Viene quindi effettuata la conversione del nome in un array di byte utilizzando l'encoding **UTF-8**.

Successivamente, viene creato un oggetto **InetAddress** che rappresenta l'indirizzo IP del server e un oggetto **DatagramPacket** che rappresenta il pacchetto che conterrà i dati da inviare al server. Infine, il pacchetto viene inviato al server utilizzando il metodo **send** della **socket UDP** e viene quindi ricevuta la risposta dal server, che viene convertita in una stringa e stampata a schermo.

In caso di eventuali eccezioni, viene utilizzato il metodo **printStackTrace** per stampare il messaggio di errore.

Codifica Java

```
public class ClientSaluti {
    final static String nomeServer = "localhost";
    final static int portaServer = 6789;
    public static void main(String[] args) {
        byte[] bufferIn = new byte[1024];
        byte[] bufferOut;
        Scanner sc = new Scanner(System.in, "UTF-8");
        System.out.print("Qual è il tuo nome? ");

        String nome = sc.nextLine() + System.LineSeparator();

        try(DatagramSocket sck = new DatagramSocket()) {
            bufferOut = nome.getBytes("UTF-8");
            InetAddress ipServer = InetAddress.getByNome(nomeServer);
            DatagramPacket pktOut = new DatagramPacket(bufferOut, bufferOut.length, ipServer,
                portaServer);
            System.out.format("Invio del nome al server...\n");
            sck.send(pktOut);
            DatagramPacket pktIn = new DatagramPacket(bufferIn, bufferIn.length);
            sck.receive(pktIn);
            String risposta = new String(pktIn.getData(), 0, pktIn.getLength(), "UTF-8");
            risposta = risposta.trim();
            System.out.format("Risposta dal server: %s", risposta);
        } catch (SocketException e) {
            e.printStackTrace();
        } catch (UnsupportedEncodingException e) {
            e.printStackTrace();
        } catch (UnknownHostException e) {
            e.printStackTrace();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

Breve spiegazione dei metodi utilizzati nella classe precedente:

1. **Scanner** - questa classe viene utilizzata per leggere l'input dell'utente.
2. **DatagramSocket** - questa classe rappresenta un socket UDP e viene utilizzata per inviare e ricevere pacchetti.
3. **InetAddress** - questa classe rappresenta un indirizzo IP.
4. **DatagramPacket** - questa classe rappresenta un pacchetto UDP e viene utilizzata per inviare e ricevere dati.
5. **send()** - questo metodo viene utilizzato per inviare un pacchetto a un indirizzo IP specifico.
6. **receive()** - questo metodo viene utilizzato per ricevere un pacchetto da un indirizzo IP specifico.
7. **getByName()** - questo metodo viene utilizzato per ottenere l'oggetto InetAddress associato a un nome host.

In particolare, il client effettua le seguenti operazioni:

1. Legge il nome dell'utente tramite l'input.
2. Crea un oggetto DatagramSocket per inviare e ricevere pacchetti.
3. Crea un oggetto InetAddress per l'indirizzo IP del server.
4. Crea un pacchetto contenente il nome dell'utente, lo indirizza al server e lo invia tramite il metodo send() del socket.
5. Riceve un pacchetto contenente la risposta del server tramite il metodo receive() del socket.
6. Stampa la risposta ricevuta dal server.

Esempio n. 4.2 (Multicast)

Scrivere un'applicazione C/S in cui il server invia periodicamente un aforisma a un gruppo di client.

Caratteristiche delle applicazioni

- **Server:** ripete continuamente la seguente attività: attende per un certo tempo; quindi, ricava una stringa di testo contenente un aforisma e la invia a un indirizzo di multicast.
- **Client:** dopo essersi unito al gruppo multicast previsto, riceve e visualizza sei aforismi. Al termine dell'attività il client abbandona il gruppo.

Questo programma in Java implementa un semplice programma client-server per la trasmissione di aforismi in multicast utilizzando socket UDP.

Il programma si compone di due classi: **'ServerAforismi'** e **'ClientAforismi'**.

Server

La classe utilizza una variabile statica `gruppo` per specificare l'indirizzo di multicast e la porta `portaMulticast` sulla quale il server invia i pacchetti multicast.

Inoltre, il server definisce un ritardo di pochi millisecondi tra l'invio di un aforisma e il successivo.

La classe contiene anche un array di aforismi `"aforismi"`, che il server invia in ordine ciclico.

Il metodo `main` del server è il punto di ingresso del programma. All'interno del metodo, viene creato un oggetto `MulticastSocket` per la comunicazione in multicast. Il server entra in un ciclo infinito, inviando gli aforismi a tutti i client connessi.

La classe **"ServerAforismi"** definisce una serie di variabili costanti, come l'indirizzo del gruppo multicast `"224.0.0.1"`, la porta multicast `"6789"`, il tempo di ritardo tra una trasmissione e l'altra `"8000ms"` e una serie di aforismi in forma di stringa.

- Nel metodo `"main"`, viene creata una socket di tipo `MulticastSocket`.
- Viene poi eseguita una richiesta per ottenere l'indirizzo IP del gruppo multicast.
- In un ciclo infinito, il server invia gli aforismi uno alla volta con una pausa di ritardo specificata. Il buffer di output contiene l'aforisma corrente, codificato in UTF-8.
- La socket invia il pacchetto (l'aforisma) all'indirizzo di multicast e alla porta specificata.
- L'indice degli aforismi viene aumentato di 1 ad ogni iterazione del ciclo. Si utilizza il modulo per calcolare l'indice successivo dell'array `"aforismi"`. In questo modo, si può passare da un aforisma all'altro, ripartendo dall'inizio dell'array una volta che si è raggiunto l'ultimo elemento. Questo garantisce che l'indice resti sempre all'interno dei limiti dell'array e non causi un errore di `"ArrayIndexOutOfBoundsException"`.
- Se si verifica un'eccezione, verrà stampato il backtrace dell'errore.

Codifica Java)

```
public class ServerAforismi { final static String gruppo = "224.0.0.1";
final static int portaMulticast = 6789;
final static int ritardo = 8000; // ms
final static String[] aforismi = {"L'istruzione è l'arma più potente che noi possiamo usare per
    cambiare il mondo (Nelson Mandela)",
    "Tutto quello che puoi fare, o sognare di poter fare, incomincialo. Il coraggio ha in
    sé genio, potere e magia. Incomincia adesso. (Goethe)",
    "Il mondo è un bel posto, per il quale vale la pena di lottare. (Ernest Hemingway)",
    "La ragione e il torto non si dividono mai con un taglio così netto che ogni parte abbia
    soltanto dell'uno e dell'altra... (Alessandro Manzoni)",
    "Due cose sono infinite: l'universo e la stupidità umana, ma riguardo l'universo ho
    ancora dei dubbi. (Albert Einstein)",
    "La natura è il miglior modo per comprendere l'arte; i pittori ci insegnano a vedere...
    (Vincent Van Gogh)"};
};
public static void main(String[] args) {
    byte[] bufferOut;
    int indice = 0;
    try (MulticastSocket sck = new MulticastSocket()) {
        System.out.println("Server avviato.");
    }
```

```
InetAddress ipMulticast = InetAddress.getByName(gruppo);
while(true) {
    Thread.sleep(ritardo);
    String af = aforismi[indice] + System.LineSeparator();
    bufferOut = af.getBytes("UTF-8");
    DatagramPacket pktOut = new DatagramPacket(bufferOut, bufferOut.length, ipMulticast,
        portaMulticast);

    sck.send(pktOut);
    System.out.format("Inviato l'aforisma n. %d all'indirizzo di multicast %s\n", indice,
ipMulticast);
    indice = (indice + 1) % aforismi.length;
}
} catch (IOException e) {e.printStackTrace();}
} catch (InterruptedException e) {
    e.printStackTrace();
}
}
```

Client

La classe **ClientAforismi** rappresenta il **client** che riceve gli aforismi inviati dal **server** attraverso la rete utilizzando il protocollo di **multicast**.

Dopo aver inizializzato le costanti **gruppo** e **portaMulticast** che specificano l'indirizzo IP e la porta del gruppo di multicast, il metodo **main** del client crea un'istanza di **MulticastSocket** che viene utilizzata per la ricezione dei pacchetti.

La prima istruzione all'interno del blocco **try** è una chiamata al metodo **System.out.println** che stampa un messaggio di conferma che il client è stato avviato.

Successivamente, il client si unisce al gruppo di multicast attraverso la chiamata al metodo **joinGroup** dell'oggetto **MulticastSocket** passando come argomento l'indirizzo IP del gruppo.

Dopo essersi unito al gruppo, il client entra in un ciclo **for** che riceve i pacchetti contenenti gli aforismi dal server. Il ciclo **for** riceve un massimo di **maxAforismi** pacchetti, **che in questo caso è impostato a 6**.

All'interno del ciclo **for**, il client crea un oggetto **DatagramPacket** per ricevere i dati inviati dal server. L'oggetto **DatagramPacket** viene inizializzato con un buffer di byte di dimensione 1024. Il metodo **receive** dell'oggetto **MulticastSocket** viene quindi utilizzato per ricevere il pacchetto dal server. Il pacchetto ricevuto viene memorizzato all'interno dell'oggetto **DatagramPacket**.

Dopo aver ricevuto il pacchetto, il client stampa il numero dell'aforisma ricevuto e l'aforisma stesso. Per ottenere l'aforisma sotto forma di stringa, viene creato un oggetto **String** a partire dal buffer di byte contenuto nell'oggetto **DatagramPacket**. **La lunghezza dell'array di byte letto dal pacchetto viene passata come argomento al costruttore della stringa per creare una stringa che contiene solo i dati effettivi letti dal pacchetto.**

Infine, dopo aver ricevuto tutti i pacchetti, il client lascia il gruppo di multicast attraverso la chiamata al metodo **leaveGroup** dell'oggetto **MulticastSocket**, passando come argomento l'indirizzo IP del gruppo.

Client (codifica Java)

```
public class ClientAforismi {
    final static String gruppo = "225.6.7.8";
    final static int portaMulticast = 6789;
    final static int maxAforismi = 6;
    public static void main(String[] args) {
        try (MulticastSocket sck = new MulticastSocket(portaMulticast)) {
            System.out.println("Client avviato.");
            InetAddress ipMulticast = InetAddress.getByName(gruppo);
            sck.joinGroup(ipMulticast);
            try {
                for (int i = 1; i <= maxAforismi; i++) {
                    byte[] bufferIn = new byte[1024];
                    DatagramPacket pktIn = new DatagramPacket(bufferIn, bufferIn.length);
                    sck.receive(pktIn);
                    System.out.format("Aforisma n. %d/%d: ", i, maxAforismi);
                    String af = new String(pktIn.getData(), 0, pktIn.getLength(), "UTF-8");
                    System.out.print(af);
                }
            } finally {
                sck.leaveGroup(ipMulticast);
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```