SOCKET PROGRAMMING IN JAVA

Prerequisito: saper realizzare applicazioni multithread. Rivedere gli appunti del modulo 1 Programmazione concorrente (da p.1 a p.13).

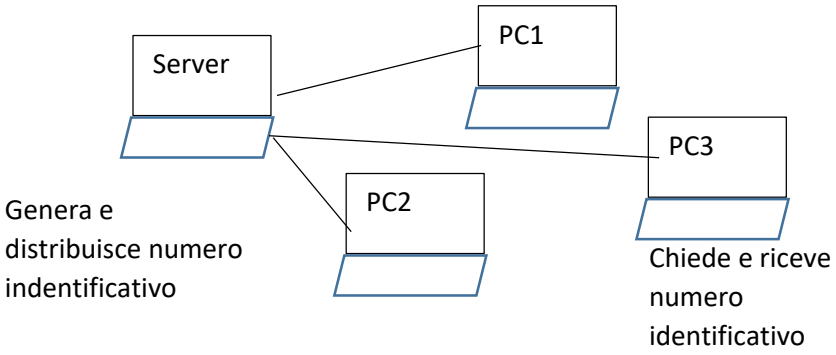
Quello che andremo a fare è scrivere una applicazione di tipo client server. Quando può essere necessario scrivere un'applicazione di questo tipo?

Facciamo un esempio: in un'azienda si vuole che a tutti i documenti realizzati da vari PC connessi in rete venga assegnato un numero identificativo univoco progressivo.

Per fare questo la soluzione sarebbe quella di realizzare un' applicazione in esecuzione su un server che "distribuisce" il numero identificativo da assegnare a ciascun documento prodotto da un qualunque PC appartenente alla rete.

Un' applicazione di questo tipo è facilmente realizzabile, basterebbe una classe con un attributo intero (statico) chimato *numero_identificativo*, con un metodo che incrementa il numero_identificativo ogni volta che viene invocato. Per tenere memoria dell'ultimo numero_identificativo generato basterebbe salvare su un file di testo tale valore dopo che è stato generato e leggerlo prima di generare il successivo. Non è particolarmente complesso.

Qual è la parte difficile, quella che non sappiamo fare? E' la comunicazione del numero identificativo fra l'applicazione server (quella che genera il numero_identificativo) e l'applicazione client (quella che richiede e riceve il numero_identificativo).



L'applicazione che vogliamo realizzare produce dei dati (nel nostro caso solamente UN dato, il numero_identificativo). Per trasmettere questi dati utilizzeremo il "protocollo" TCP/IP quindi dovremo "passare" i dati da trasmettere al livello di trasporto e scegliere uno dei due protocolli disponibili: UDP o TCP. Questo sarà uno dei prossimi esercizi. Prima però spieghiamo come far comunicare fra loro due host.

Ricordiamo molto brevemente la differenza fra i due protocolli: UDP è quello non affidabile, connectionless, best effort, veloce, TCP è quello affidabile, connection oriented, lento.

Come si fa ad utilizzare il protocollo di trasporto?

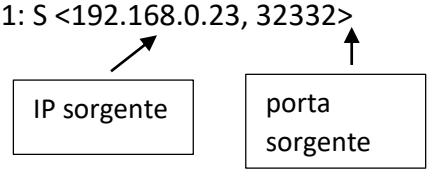
Il protocollo di trasporto è implementato da apposite **API** (Application Program Interface) del Sistema Operativo. Ogni SO espone **delle interfacce per utilizzare queste API** (tali interfacce sono **oggetti** Java) per i due principali protocolli a livello di trasporto, UDP e TCP.

Queste interfacce vengono generalmente chiamata **socket**, in analogia con la presa elettrica della rete (che è, infatti, un'interfaccia che consente di utilizzare l'energia trasportata dalla rete elettrica).

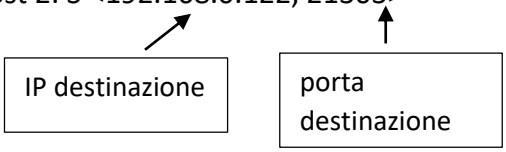
I processi che devono comunicare, che sono in esecuzione generalmente su due host diversi comunicheranno fra loro scrivendo/leggendo i dati su/da un socket.

Poiché, come abbiamo visto, per identificare un processo in esecuzione su un host della rete sono necessari due parametri: **l'indirizzo IP dell'host** e il **numero di porta** associato al processo, un **socket è completamente identificato dalla coppia IP/numero_di _porta.** Ad esempio, supponiamo che host 1 voglia trasmettere dei dati ad host 2:

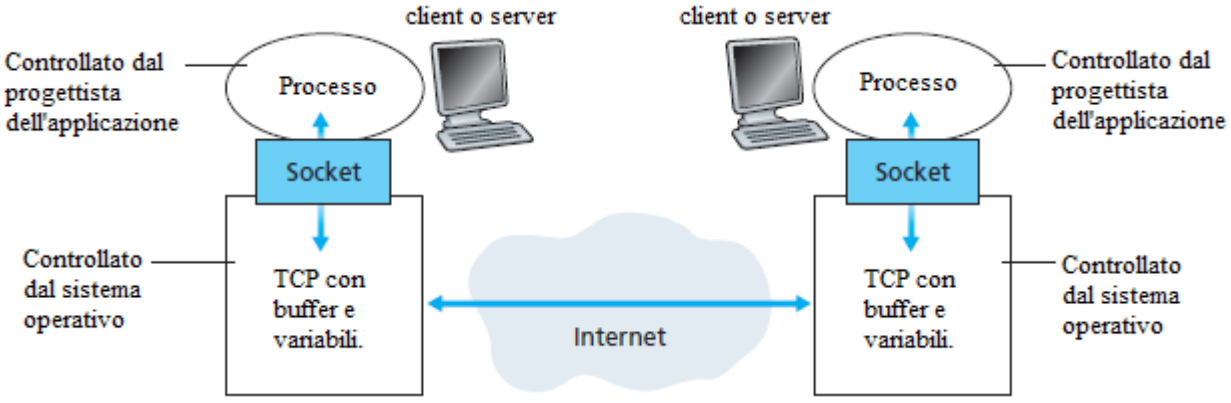
Socket host 1: S <192.168.0.23, 32332>



Socket host 2: S <192.168.0.122, 21303>



Nella seguente immagine si mostrano due processi che comunicano utilizzando i socket



Java dispone del package **java.net** che contiene classi appositamente realizzate per la comunicazione di rete.

Le classi principali di questo package sono le seguenti 7:

Classe	Descrizione
`InetAddress`	Rappresenta un indirizzo IP (indifferentemente versione 4 a 32 bit, o versione 6 a 128 bit) e/o un nome di dominio che risolve in modo trasparente utilizzando un servizio DNS; le classi derivate *Inet4Address* e *Inet6Address* rappresentano in modo specifico indirizzi IP versione 4 o versione 6.
`Socket`	Rappresenta il socket client di una connessione TCP; la classe derivata *SSLSocket* consente di usare le funzionalità del protocollo crittografico TLS.
`ServerSocket`	Rappresenta il socket di ascolto di un server TCP; la classe derivata *SSLServerSocket* consente di usare le funzionalità del protocollo crittografico TLS.
`DatagramSocket`	Rappresenta il socket che impiega il protocollo UDP.
`DatagramPacket`	Rappresenta un *datagram* UDP ricevuto o da trasmettere: oltre ai dati comprende l'indirizzo IP mittente o di destinazione e il numero di porta UDP mittente o di destinazione.
`URL`	Rappresenta una qualsiasi risorsa presente nel web mediante il suo *Uniform Resource Locator*.
`URLConnection`	Permette la creazione di client che utilizzano risorse rese disponibili utilizzando il protocollo HTTP. Oltre a scaricare e caricare risorse da/su un server HTTP, i metodi della classe *URLConnection* consentono di acquisire, impostare e gestire gli *header* del protocollo.

TCP
socket client
socket server

UDP
socket
datagramma

Crea un client per inviare richieste http ad un server

Le eccezioni che possono sollevare queste classi sono le seguenti (figlie di **IOException**, quindi eccezioni **cheked**). Particolarmente interessante è la SocketTimeoutException.

TABELLA **2**

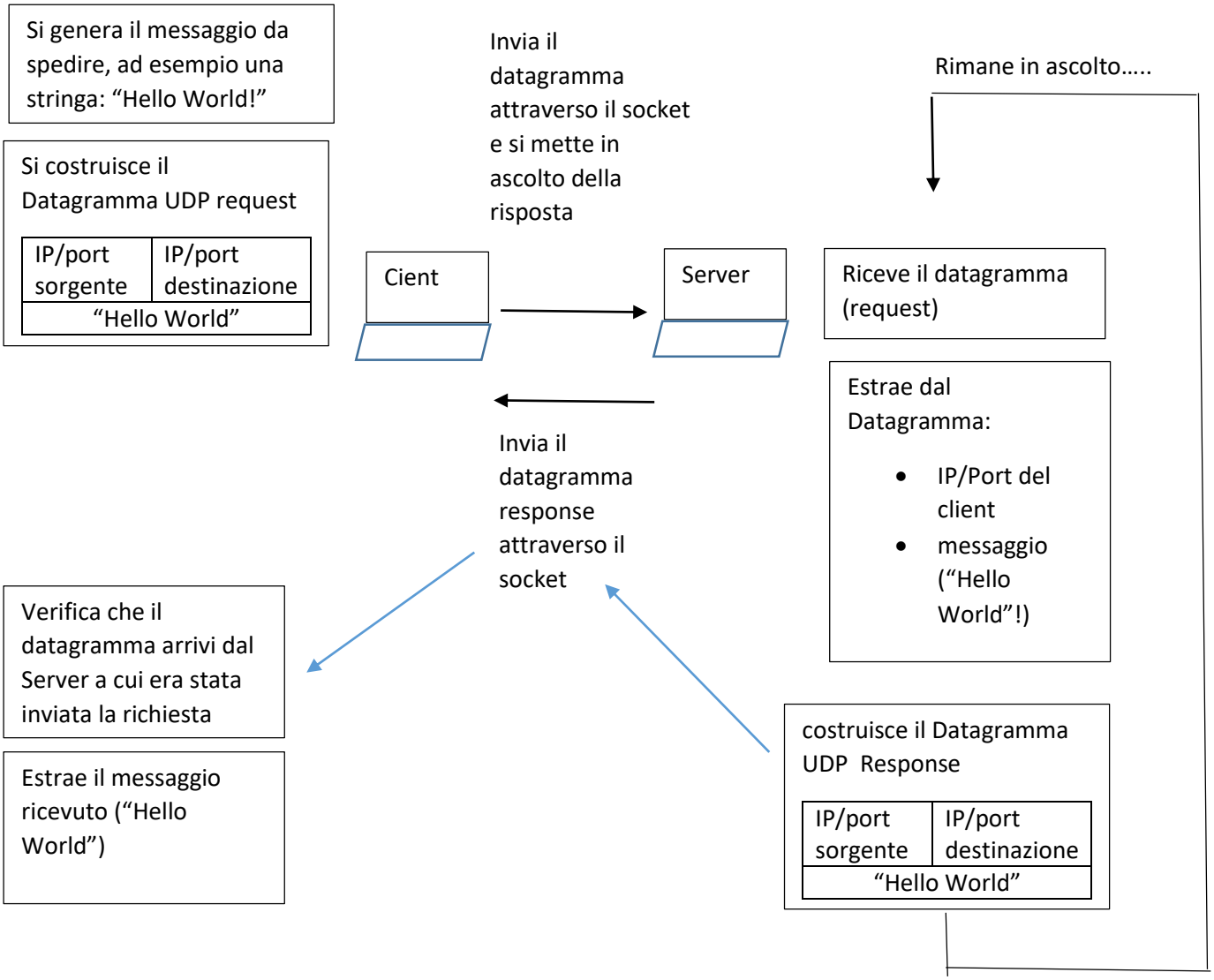
Eccezione	Descrizione
`SocketException`	Errore di comunicazione (sono derivate da *SocketException* le eccezioni *ConnectException*, *NoRouteToHostException* e *PortUnreachableException*).
`MalformedURLException`	Errore di formato di un URL.
`ProtocolException`	Errore del protocollo di trasporto utilizzato (UDP o TCP).
`SocketTimeoutException`	Attesa superiore al *timeout* impostato per un'operazione di lettura o di connessione.
`UnknownHostException`	Errore di risoluzione del nome di un server in un indirizzo IP mediante il servizio DNS.

Per un socket si può impostare un valore di timeout. Il socket rimarrà in ascolto fino allo scadere del timeout dopodichè solleverà questa eccezione.

1. SOCKET UDP IN LINGUAGGIO JAVA

Come prima applicazione che utilizza un socket, realizzaremo un semplice protocollo applicativo che viene spesso utilizzato per testare la configurazione di una rete LAN o WAN. Il protocollo prevede un server UDP (ma si potrebbe realizzare anche con TCP, che vedremo successivamente) attivo in ascolto su una porta (generalmente la porta 7, *well know port* per ECHO SERVER) che effettua l'eco di tutti i dati che riceve dal client, ossia ritrasmette al client tutti i dati ricevuti.

Visto che utilizziamo UDP, la comunicazione avviene, a livello di trasporto, senza connessione, nel seguente modo:



Spiegazione: come implementiamo l'applicazione Echo.

- Nel processo server istanziamo un socket associato alla porta 7 e lo mettiamo in "attesa" di un datagramma con il metodo *void receive(datagramPacket request).* (Il pacchetto request è un datagramma "vuoto" precedentemente istanziato in cui verrà "messo" il datagramma ricevuto quando arriverà). Questo significa che ogni datagramma ricevuto con numero di porta destinazione=7 sarà elaborato da questa applicazione Server.
- Nel processo client istanziamo un socket senza specificare il numero di porta associato al processo (la assegnerà il SO, è la porta effimera).
- Sempre nel processo client istanziamo un datagramma specificando: l'IP del server, la porta del server(7), i dati sotto forma di array di byte (in questo caso la stringa che verrà trasmessa e restituita) e la lunghezza dei dati in termini di numero di byte:

  datagramPacket datagram=new datagramPacket(arrayDiByteDati,numero_byte_Dati, IP_server, portaServer).

  Il datagramma da inviare conterrà anche l'IP del client e il numero di porta (casuale, ad esempio 5067) associata al processo client, ma questi valori vengono aggiunti automaticamente dal sistema operativo.
- Il processo client invia la richiesta con il metodo della classe DatagramSocket "*send (datagramPacket datagram)"* , e poi si mette in "ascolto" della risposta del server con il metodo della classe **DatagramSocket** "*receive (datagramPacket datagram)".*
- Il processo server riceve il datagramma inviato dal client, estrae le seguenti informazioni:
  - array di dati inviati dal client
  - IP del client *IP_client*
  - porta (effimera) associata al client (5067) *portaClient*

  Con questi dati il server elabora l'array di dati della risposta (in questo caso, essendo un ECHO, uguale all'array di dati ricevuta) e istanzia il datagramma di risposta:

  datagramPacket response=new datagramPacket(arrayDiByteDati,numero_byte_Dati, IP_client, portaClient).

- Il server invia il datagramma *response* di risposta (con il metodo *void send (response)*)
- Il server si mette in ascolto della ricezione di un nuovo datagramma di richiesta
- Il client riceve il datagramma di risposta ed estrae i dati (e li utilizza come vuole)
- Il client chiude il socket e termina l'esecuzione.

OSSERVAZIONE: se il client invia una nuova richiesta eseguendo nuovamente il processo, la porta associata al processo client può cambiare rispetto a prima, è effimera (ad esempio 5209). Naturalmente se la nuova richiesta avviene sempre dal processo precedente (ad esempio realizzando un ciclo), la porta effimera non cambia.

Per realizzare i client e server dell'applicazione utilizzeremo dunque le classi Java **DatagramSocket** e **DatagramPacket**. Vediamo i loro metodi principali:

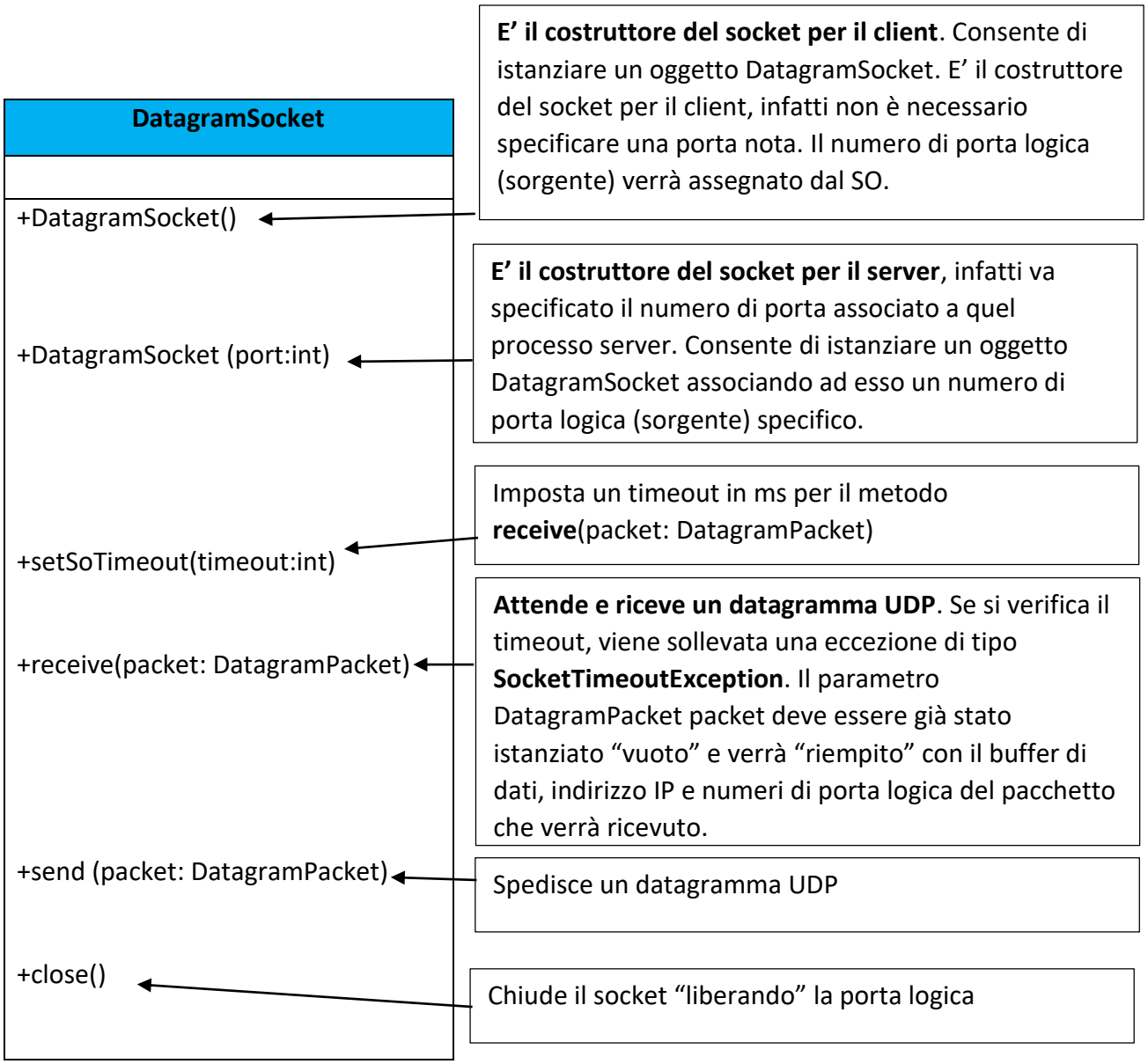
CLASSE DatagramSocket

La classe Java *DatagramSocket* permette di creare e gestire la ricezione e la trasmissione di *datagram* UDP.

I metodi maggiormente utilizzati della classe sono descritti nella TABELLA **3**.

TABELLA **3**

Metodo	Descrizione
`DatagramSocket()`	Costruttore: crea un socket per un numero di porta UDP arbitrario.
`DatagramSocket (int port)`	Costruttore: crea un socket per lo specifico numero di porta UDP.
`void setSoTimeout (int timeout)`	Imposta il tempo di attesa massimo espresso in millisecondi di un *datagram* da parte del metodo *receive*.
`void setBroadcast (boolean broadcast)`	Abilita/disabilita la possibilità di ricevere/trasmettere *datagram* di tipo *broadcast*.
`void receive (DatagramPacket packet)`	Attende e riceve un *datagram* UDP; in caso di *timeout* solleva la relativa eccezione.
`void send (DatagramPacket packet)`	Trasmette un *datagram* UDP.
`close()`	Chiude il socket.

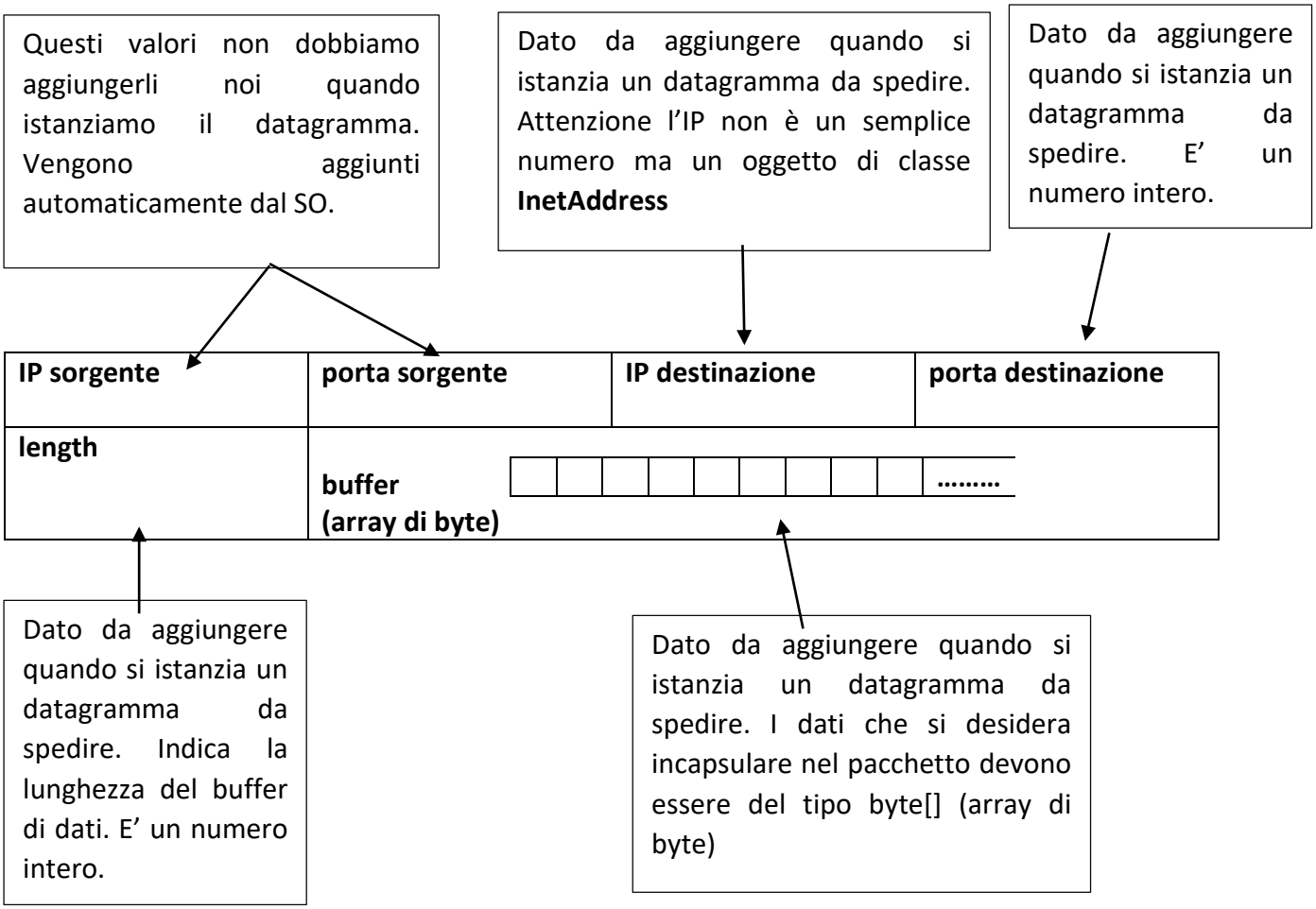


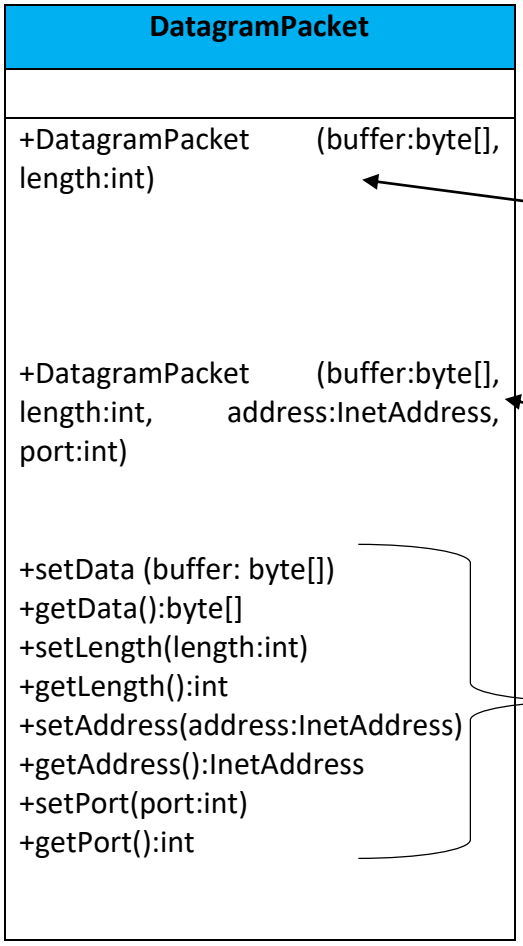
L'associazione fra un socket ed una porta logica è detta **binding**.

Dato un processo in esecuzione su un host viene istanziato un socket al quale viene associata una porta logica (operazione di binding), da quel momento in poi quel processo è **univocamente identificato** sulla rete dalla coppia **IP host/porta logica.**

Per istanziare un datagramma si utilizza la classe **DatagramPacket**.

La struttura di un datagramma è la seguente:





Costruttore per istanziare un datagramma in ricezione (datagramma "vuoto"), i vari elementi di questo oggetto saranno valorizzati quando il datagramma verrà ricevuto dal socket.

lenght indica il numero di byte che verranno letti (e posti nel buffer) quando verrà ricevuto il datagramma, deve avere un valore minore o uguale alla lunghezza del buffer

Costruttore per istanziare un datagramma da trasmettere. I parametri sono: i dati da trasmettere, la loro lunghezza, la coppia indirizzo IP/porta di destinazione.

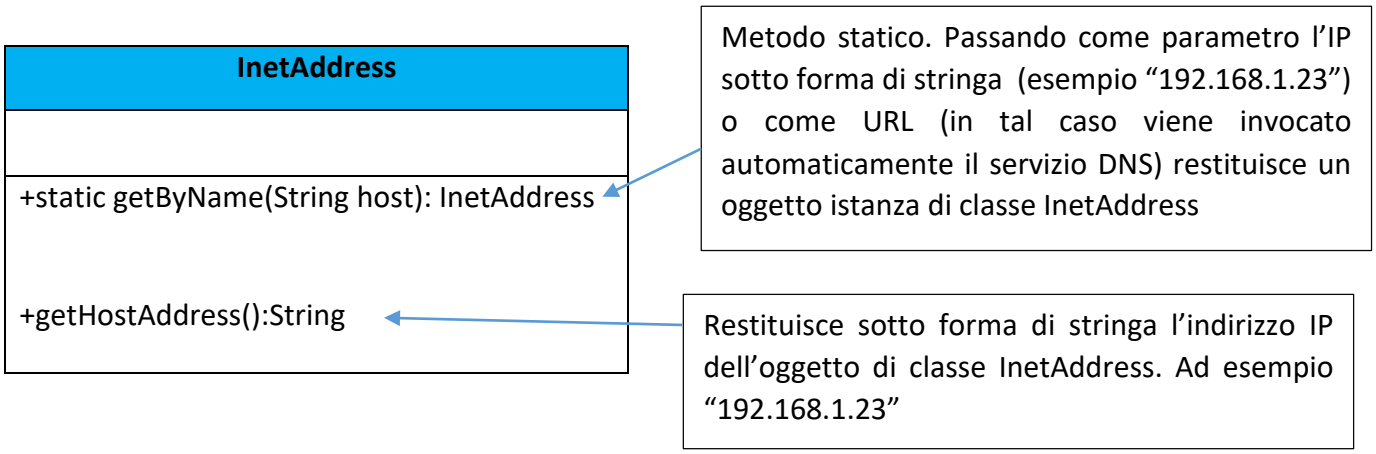
setter e getter per inserire nel/estrarre dal datagramma i vari elementi

Address e port sono quelli di destinazione se il pacchetto è da spedire, sono quelli di origine se il pacchetto è stato ricevuto. Intelligente!

Per spedire un pacchetto è necessario passare come parametro al datagramma l'IP dell' host di destinazione. L'host viene specificato come istanza di una classe ***InetAddress*.** Tale classe **non espone un costruttore,** quindi per ottenerne un'istanza (un oggetto InetAddress) è necessario invocare alcuni suoi metodi statici.

L'indirizzo IP incapsulato nella classe InetAddress può essere impostato (quindi inserito ed estratto) in 3 modi

- serie di byte (4 o 16) che rappresentano l'indirizzo IP (IPV4 o IPV6)
- Stringa che rappresenta la versione testuale dell'indirizzo IP. Esempio "127.0.0.1"
- Stringa che rappresenta l'URL dell' host. Esempio "www.google.it". In tal caso l'indirizzo IP viene risolto in maniera trasparente mediante il servizio DNS



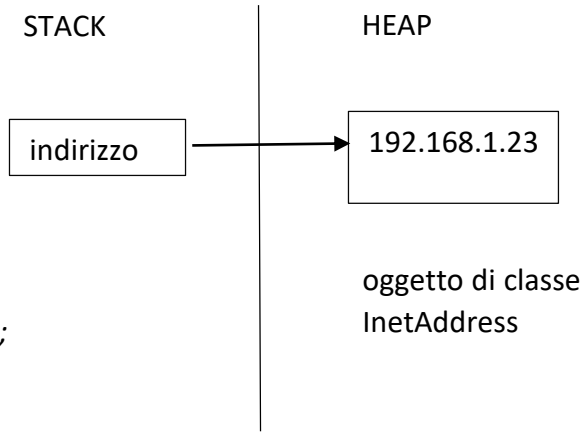
Esempio per istanziare un oggetto di classe InetAddress:

InetAddress indirizzo;

indirizzo=InetAddress.getByName ("192.168.1.23");

//per mostrare l'indirizzo

system.out.print("l'indirizzo è: " + indirizzo.getHostAddress);



La classe consente di rappresentare sia indirizzi IPv4 sia IpPv6, (vi sono poi due apposite classi figlie per le due specifiche tipologie di indirizzi).

I metodi principali della classe InetAddress sono i seguenti:

Metodo	Descrizione
`static InetAddress getByAddress(byte[] address)`	Restituisce un oggetto istanza della classe *InetAddress* a partire dalla rappresentazione binaria dell'indirizzo IP.
`static InetAddress getByName(String host)`	Restituisce un oggetto istanza della classe *InetAddress* a partire da un URL, o dalla rappresentazione testuale di un indirizzo IP.
`static InetAddress getLocalHost()`	Restituisce un oggetto istanza della classe *InetAddress* che rappresenta l'indirizzo IP del computer su cui viene eseguito il metodo.
`byte[] getAddress()`	Restituisce la rappresentazione binaria dell'indirizzo IP rappresentato dall'oggetto.
`String getHostName()`	Restituisce l'URL corrispondente all'indirizzo IP rappresentato dall'oggetto.
`String getHostAddress()`	Restituisce la rappresentazione testuale dell'indirizzo IP rappresentato dall'oggetto.
`boolean isReachable(int timeout)`	Verifica la raggiungibilità di rete nel tempo specificato espresso in millisecondi dell'indirizzo IP rappresentato dall'oggetto[3].

Restituisce oggetto InetAddress a partire dall' Indirizzo IP espresso come array di byte (4 per IPv4, 16 per IPv6)

Restituisce oggetto InetAddress a partire dall' Indirizzo IP espresso come String o dall' URL

Realizziamo il server con ciclo infinito

La classe UDPechoServer viene inizialmente realizzata senza thread nel seguente modo:

UDPServer
-socket: datagramSocket
+UDPServer(int port) +run():void

costruttore: istanzia il datagramSocket e gli assegna una porta e un timeout di 1s

Metodo run(): esegue un ciclo infinito in cui

il socket rimane in "ascolto" in attesa di un datagramma

quando arriva il datagramma ne estrae i dati, costruisce il datagramma di risposta e lo invia

Nella classe aggiungiamo poi il metodo main nel quale istanziamo un UDPechoServer e lo mandiamo in run.

```
public class UDPServer
{
   private DatagramSocket socket;

   public UDPServer(int port) throws SocketException
   {
      socket=new DatagramSocket(port);
      socket.setSoTimeout(1000);                    //Aggiungiamo un timeout che serve per
                                                    // interrompe  l'ascolto del socket ogni secondo
   }

   public void run()
   {
      byte[] buffer=new byte[8192];
      DatagramPacket request=new DatagramPacket(buffer, buffer.length);
       //è il datgram packet "vuoto" che conterrà la request ricevuta

      while (true)
      {
         try
         {
            socket.receive(request);
            DatagramPacket response=new
            DatagramPacket(request.getData(),request.getLength(),request.getAddress(),request.getPo
rt());
            socket.send(response);
         }
         catch (SocketTimeoutException ex)
         {
            System.out.println("Timeout!");
         }
         catch (IOException ex)
```

```
            {
               System.out.println("Errore di comunicazione!");
            }
         }
      }

      public static void main(String[] args)
      {
         try
         {
            UDPServer EchoServer=new UDPServer(7);
            EchoServer.run();
         }
         catch (SocketException ex)
         {
            System.out.println("Impossibile aprire il socket!");
         }
      }
   }
```

Abbiamo aggiunto all'attributo socket dell'UDPechoServer un timeout da un secondo affinchè il socket rimanga in ascolto solamente per tale intervallo di tempo, poi si interrompa e poi si rimetta in ascolto grazie al ciclo infinito.

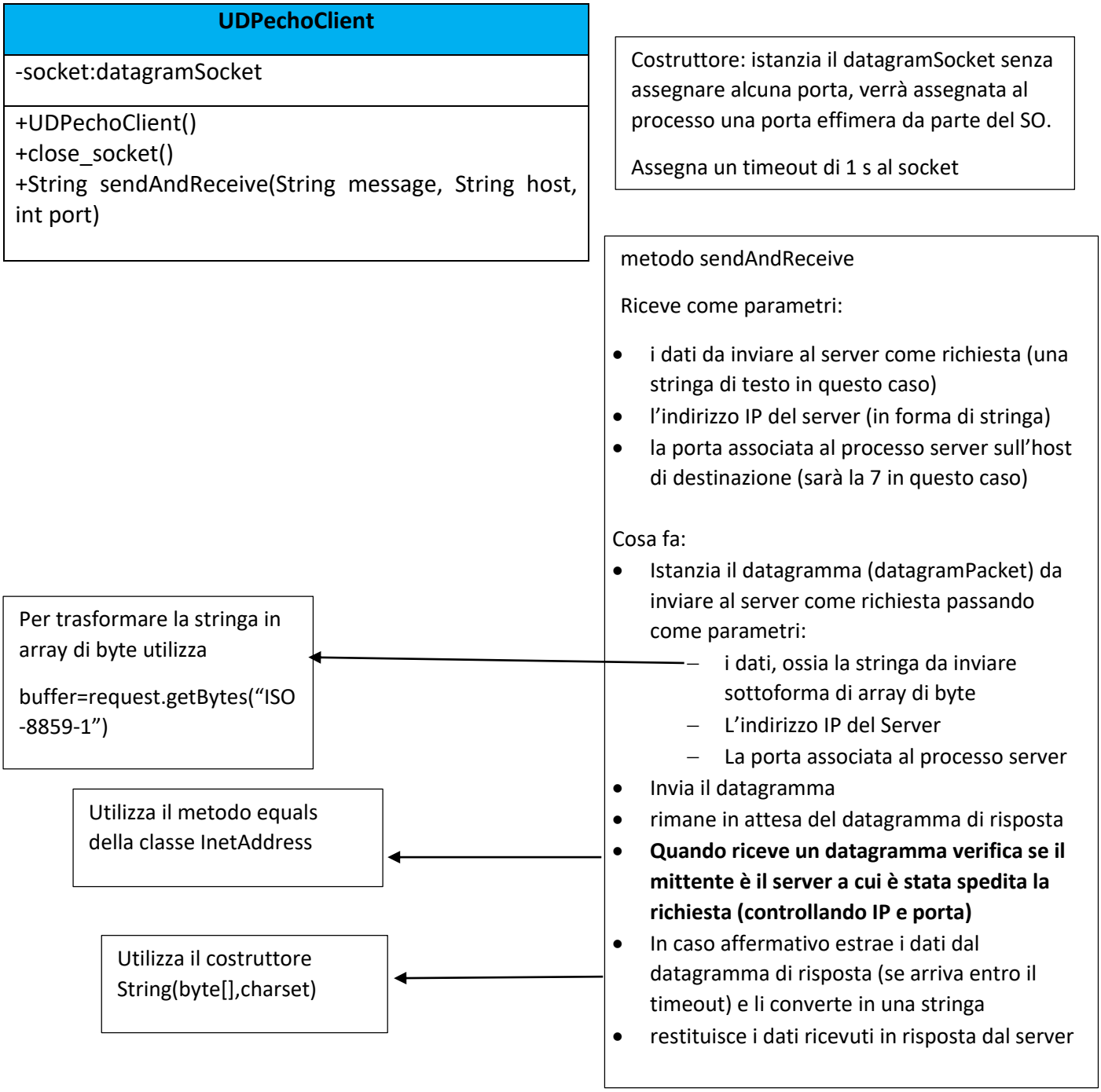
Quando nel metodo main si istanzia e si manda in esecuzione un UDPechoServer, il ciclo infinito fa sì che l'esecuzione non si possa mai interrompere.

Per evitare questo ciclo infinito modifichiamo la classe UDPEchoServer facendola diventare un Thread e chiamiamolo threadServer. In questo modo, quando il suo metodo run verrà avviato (con il metodo start, attenzione!. Parallelamente il codice nel metodo main può continuare ad essere eseguito. Nel metodo main quindi inseriremo le istruzioni che permetteranno di acquisire un input qualsiasi da tastiera che andrà ad interrompere il metodo run() del threadServer, che quindi chiuderà il socket e terminerà la propria esecuzione.

```
public class UDPServer implements Runnable       //per poter essere parametro di Thread
{
   private DatagramSocket socket;

   public UDPServer(int port) throws SocketException
   {
      socket=new DatagramSocket(port);
      socket.setSoTimeout(1000);
   }

   public void run()
   {
      byte[] buffer=new byte[8192];
      DatagramPacket request=new DatagramPacket(buffer, buffer.length);
```

```
      while (!Thread.interrupted())
      {
         try
         {
            socket.receive(request);
            DatagramPacket response=new
DatagramPacket(request.getData(),request.getLength(),request.getAddress(),request.getPort());
            socket.send(response);
         }
         catch (SocketTimeoutException ex)
         {
            System.out.println("Timeout!");
         }
         catch (IOException ex)
         {
            System.out.println("Errore di comunicazione!");
         }
      }
      socket.close();
      System.out.println("Il server è stato disconnesso!");
   }

   public static void main(String[] args)
   {

      Scanner tastiera=new Scanner(System.in);
      try
      {
         UDPServer echoServer=new UDPServer(7);
         Thread threasServer=new Thread();
         threasServer.start();
         tastiera.nextLine();
         threasServer.interrupt();
         System.out.println("Server interrotto");
      }
      catch (SocketException ex)
      {
         System.out.println("Impossibile aprire il socket!");
      }
      catch (InterruptedException ex)
      {
         System.out.println("Errore di comunicazione!");
      }
   }
}
```

Realizziamo ora un'applicazione client del protocollo echo per testare il server.



Nel main viene istanziato un oggetto di classe UDPechoClient e viene invocato il metodo sendAndReceive passando come parametri la stringa di richiesta (una qualsiasi stringa), l'indirizzo IP del server, la porta associata al server.

Tali valori, se specificati, vengono acquisiti come argomenti dal metodo main (String args[]) altrimenti assumono valori di default ("Ciao Mondo!", "127.0.0.1", 7)

Le possibili eccezioni vengono gestite nel main.

```
public class UDPClient
{
   private DatagramSocket socket;

   public UDPClient() throws SocketException
   {
      socket=new DatagramSocket();
      socket.setSoTimeout(1000);
   }

   public String sendAndReceive(String message, String serverAddress, int port) throws
UnknownHostException, UnsupportedEncodingException, IOException
   {
      byte[] buffer;
      InetAddress ipAddress=InetAddress.getByName(serverAddress);
      String answer;

      buffer=message.getBytes("UTF-8");
      DatagramPacket datagram=new DatagramPacket(buffer,buffer.length,ipAddress,port);
      socket.send(datagram);
      socket.receive(datagram);
      answer=new String(datagram.getData(),"UTF-8");
      return answer;
   }
   public void close()
   {
      socket.close();
   }

   public static void main(String[] args)
   {
      //valori di default
      String message="Hello World!";
      String IPAddress="127.0.0.1";
      int port=7;

      //args[0]-->messaggio   args[1]-->indirizzo IP del server   args[2]--> porta del server
      if (args.length==3)
      {
         message=args[0];
         IPAddress=args[1];
         port=Integer.parseInt(args[2]);
      }

      try
      {
         String answer;
         UDPClient echoClient=new UDPClient();
         answer=echoClient.sendAndReceive(message, IPAddress, port);
         echoClient.close();
         System.out.println("Risposta del server: "+answer);
      }
      catch (SocketException ex)
```

```
        {
            System.out.println("Impossibile aprire il socket!");
        }
        catch (UnsupportedEncodingException ex)
        {
            System.out.println("Formato della risposta non supportato");
        }
        catch (SocketTimeoutException ex)
        {
            System.out.println("Il server non risponde!");
        }
        catch (IOException ex)
        {
            System.out.println("Errore di comunicazione");
        }

    }
}
```

ISTRUZIONE PER L'ESECUZIONE

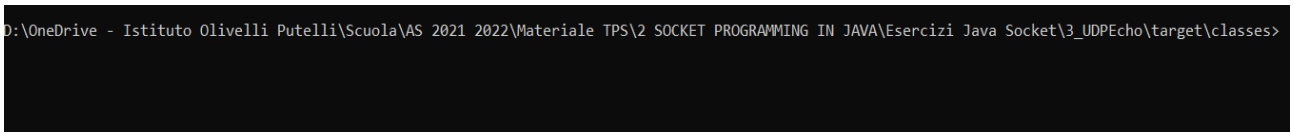
Per eseguire i due processi client e server sullo stesso pc: avviare due sessioni del prompt dei comandi e avviare la classe client da un prompt e la classe server dall'altro.

Per eseguire una classe con la JVM dal prompt: dalla cartella che contiene la classe scrivere

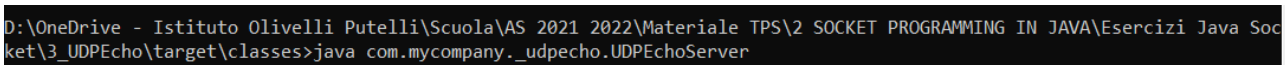
java nomeclasse "parametro1" "parametro2" …..

(con virgolette perché i parametri sono stringhe)

Per eseguire una classe da riga di comando devo "portarmi" con il path del prompt nella cartella che contiene i package (altrimenti mostra l'errore: could not find or load class…)



Da qui scrivere java nomePackage.nomeClasse



Ricapitolando, per eseguire da riga di comando devo trovarmi nella cartella che contiene il package e digitare "java" seguito dal nome della classe compreso tutto il package (package.sottopackage.nomeclasse).

Un'altra soluzione, più comoda, per eseguire una classe da linea di comando è la seguente:

1. Creare un file batch. Il file batch è un file di testo con estensione .bat che contiene istruzioni eseguibili da linea di comando. Il file batch da creare è un file di testo che contiene la riga "java package.sottopackage.nomeclasse". Nel nostro caso:

   java com.mycompany._udpecho.UDPEchoServer

2. Salvare il file nella cartella al seguente path “nomeProgetto\target\classes” nominando il file “launcherUDPEchoServer.bat”
3. Creare un collegamento a questo file e spostarlo dove si vuole. Il collegamento consentirà di eseguire il file batch e quindi avviare la classe java.

ESERCIZIO 1: “DateTimeUDP”

Partendo dall’applicazione UDPecho, creare una nuova applicazione client/server in cui il server restituisce al client, in seguito all’invio di un qualunque messaggio nella request, una response contenente una stringa in cui sono indicati data e ora attuali.

Per vedere come si generano data e ora attuali con la classe Java LocalDateTime, visionare gli appunti Java dell’anno scorso.

ESERCIZIO 2: “NumeroProtocolloUDP”

Partendo dall’applicazione UDPecho, creare una nuova applicazione client/server che realizzi il seguente servizio: il server restituisce un numero di protocollo progressivo ogni volta che viene inviata una richiesta dal client. Utilizzare sul server la porta 1100 per questo servizio.

Per conoscere l’ultimo numero di protocollo inviato, il server, ogni volta che invia ad un client un nuovo numero di protocollo, memorizza tale numero di protocollo su un file di testo.

Il protocollo a livello applicativo è il seguente

REQUEST: stringa contenente “**?**”

RESPONSE: Stringa contenente il numero di protocollo (in formato String) oppure una stringa contenente “Error” se il formato delle richiesta non è corretta.

Suggerimenti:

- nel client utilizzare un byteBufferRequest da 1 byte per la richiesta (contenente: “?”) e un apposito datagramRequest , inoltre utilizzare un byteBufferResponse da 10 byte per la risposta e un apposito datagramResponse

- nel server, per verificare se la richiesta arrivata dal client è corretta, utilizzare:
  if (…….request.getData()[0]==63) (il byte in posizione 0 del buffer)

  poiché 63 è il codice ascii (e anche “ISO 8859-1” o “UTF-8”) del carattere ‘?’.

- nel server creare un metodo “leggiDaFile()” per leggere dal file il numero di protocollo (Nprotocollo) da inviare e un metodo “scriviSuFile(Nprotocollo)” per memorizzare su file il numero di protocollo dopo che è stato spedito. Per leggere e scrivere da/su un file di testo utilizzare le classi TextFile e FileException degli esercizi precedenti

- per convertire una stringa in un numero si usa:

  numero=Integer.parseInt(stringa);

- predisporre inizialmente nel server il file di testo con valore di protocollo iniziale=0.

ESERCIZIO 3: server per timestamp (fare da soli)

- Il timestamp è un tipo long e si ottiene con:

  long timestamp= Instant.now().getEpochSecond();

- Ipotizziamo che il server invii il messaggio Timestamp sottoforma di stringa e di utilizzare 15 byte per contenere il messaggio (questo contiene il timestamp per un tempo molto lungo, milioni di anni)
- Si utilizzi per il server la porta 2000.

PROBLEMI

15 CODING Scrivere in linguaggio Java, utilizzando il protocollo di trasporto UDP, un servizio di tipo *time* che restituisca un *datagram* contenente il valore numerico in formato String del numero di secondi trascorsi dal 1 gennaio 1970 in risposta a ogni *datagram* ricevuto, indipendentemente dal suo contenuto. Realizzare in linguaggio Java un programma client che visualizzi la risposta ricevuta dal server UDP *time*, o il messaggio «TIMEOUT» nel caso in cui non riceva una risposta dal server entro un secondo.

ESERCIZIO 4: calcolatrice server

16 DESIGN CODING **Progettare un protocollo applicativo binario per un servizio basato sul protocollo di trasporto UDP che richieda a un server di effettuare le comuni operazioni aritmetiche (somma, sottrazione, moltiplicazione, divisione e potenza) tra coppie di valori numerici. Scrivere in linguaggio Java un server UDP che implementi il protocollo progettato: verificarne il corretto funzionamento – compresa la gestione di errori – scrivendo uno specifico client.**

ESEMPIO

Per realizzare una calcolatrice è possibile implementare un protocollo testuale basato sui seguenti comandi terminati dai caratteri CR (codice ASCII 13) ed LF (codice ASCII 10):

Comando	Operazione
ADD,*X*,*Y*	Addizione (*X* + *Y*)
SUB,*X*,*Y*	Sottrazione (*X* – *Y*)
MUL,*X*,*Y*	Moltiplicazione (*X* × *Y*)
DIV,*X*,*Y*	Divisione (*X* : *Y*)

Esempi di comandi del protocollo sono le stringhe «`ADD,1.5,-0.5`», «`SUB,0,1`», «`MUL,1.125,8`» e «`DIV,1024,2.0`». Il server che implementa la calcolatrice risponderà con una stringa contenente il risultato numerico o con la stringa «`ERROR`»; in entrambi i casi la risposta sarà terminata dai caratteri CR ed LF.

ESERCIZIO 5: ORIENTEEERING P.88. Fare insieme “definendo” un possibile protocollo (vedi file)

CORRISPONDE ALL’ ESERCIZIO 19 P. 88-89 CON UDP

19 DESIGN CODING Nello sport dell’orienteering i concorrenti dimostrano il passaggio dai punti di tappa acquisendo con l’APP del proprio smartphone il QR-code esposto: l’APP invia l’identificativo della tappa costituito da una stringa alfanumerica e quello numerico del concorrente registrato nell’APP a un server **UDP**
Dopo aver definito e documentato un protocollo testuale applicativo che permetta l’invio al server dell’identificativo numerico del concorrente e di quello alfanumerico del QR-code di una tappa,
implementare in linguaggio Java una classe che consenta lato server di registrare gli identificativi concorrente/tappa ricevuti associandoli a un *timestamp* rappresentante il numero di secondi trascorsi dalle 00:00:00 del 1/1/1970.
Realizzare infine un server **UDP** in linguaggio Java che utilizzi la classe implementata per la registrazione dei dati ricevuti.

Realizzare una classe client UDP che consenta di testare il server

SERVER E CLIENT TCP IN LINGUAGGIO JAVA

Il protocollo TCP, come visto, è un protocollo **affidabile** e **connection oriented**, ciò significa che prima di iniziare lo scambio di dati fra due host, richiede che sia **esplicitamente** realizzata fra di essi una connessione.

Una volta stabilita la connessione, il protocollo garantisce che tutti i byte trasmessi sul socket sorgente raggiungano il socket di destinazione **nell'ordine in cui sono stati trasmessi**.

La connessione avviene nel seguente modo (**three way hndshake**):

- Il processo server si mette in ascolto (su una porta nota al processo client) della richiesta di connessione da parte di uno o più processi client.

1. Un processo client inoltra la richiesta di connessione al processo server indicando il proprio indirizzo IP e la porta (effimera) sulla quale vuole stabilire la connessione.
2. Il processo server, una volta ricevuta la richiesta, trasmette un messaggio al processo client di "accettazione" della connessione (in maniera trasparente per il livello applicazione)
3. Il client risponde inviando al server la conferma della connessione

Vediamo un esempio pratico:

Realizzazione di un server DateTimeServer e relativo client. Il client manda una richiesta di connessione al server ed esso risponde inviando data e ora correnti (p.60 libro).

Utilizzeremo questo esempio pratico per spiegare le seguenti 3 fasi della realizzazione di un'applicazione client server con TCP.

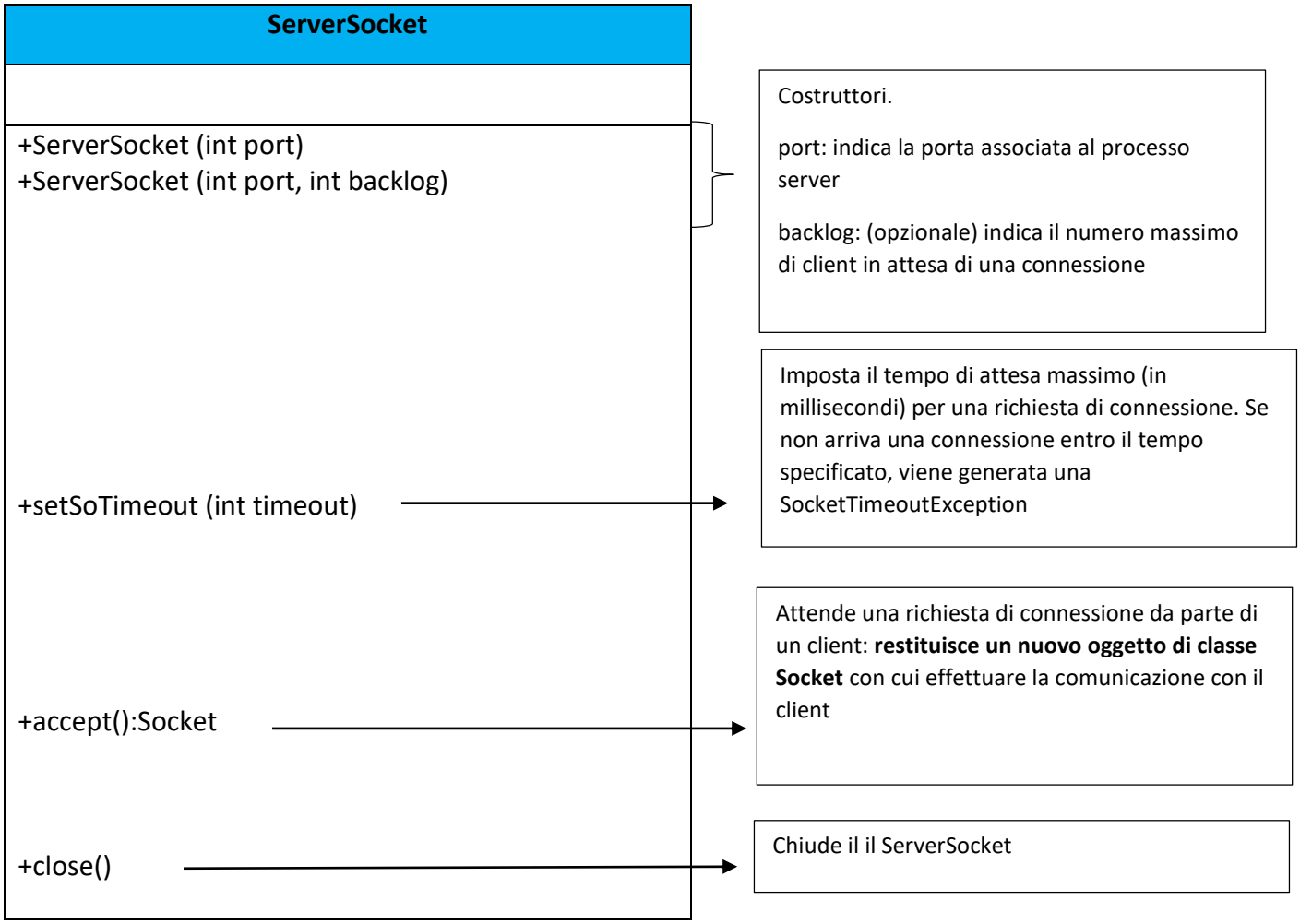
Fase 1: come stabilire una connessione TCP

Fase 2: come trasmettere i dati con i socket

Fase 3: realizzazione del server e del client

Fase 1: come stabilire una connessione TCP

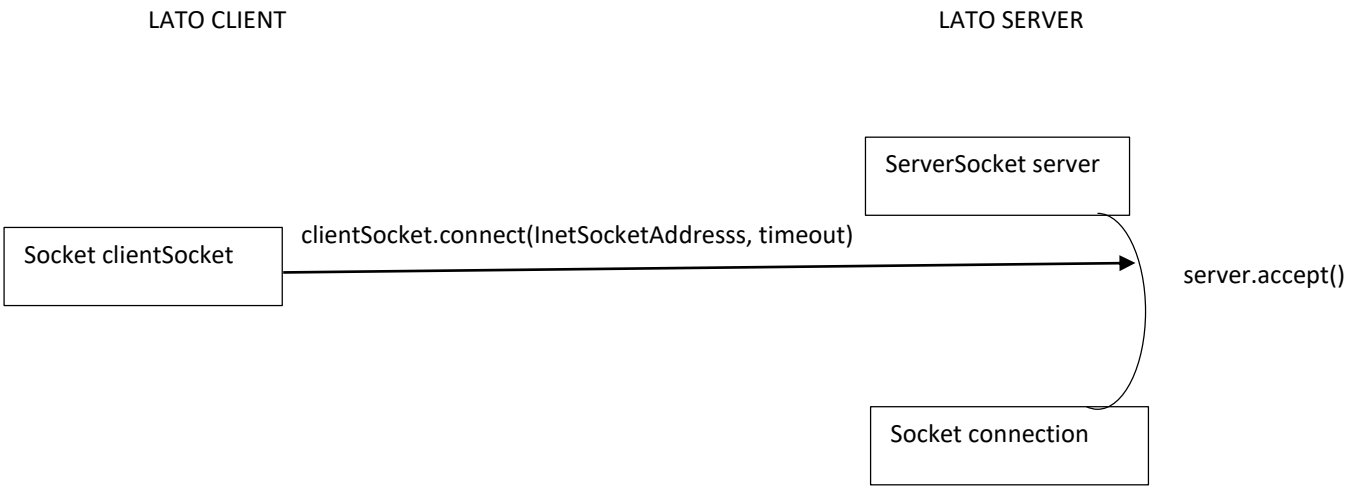
La classe ServerSocket: permette di **creare** un socket che realizza un server TCP



Quindi **l'oggetto di classe Socket** del server viene istanziato solamente nel momento in cui arriva la richiesta da parte del client.

La comunicazione fra server e client avviene mediante l'oggetto di classe **Socket** restituito dal metodo **accept()** della classe ServerSocket nel momento in cui un client effettua la richiesta di connessione al server.

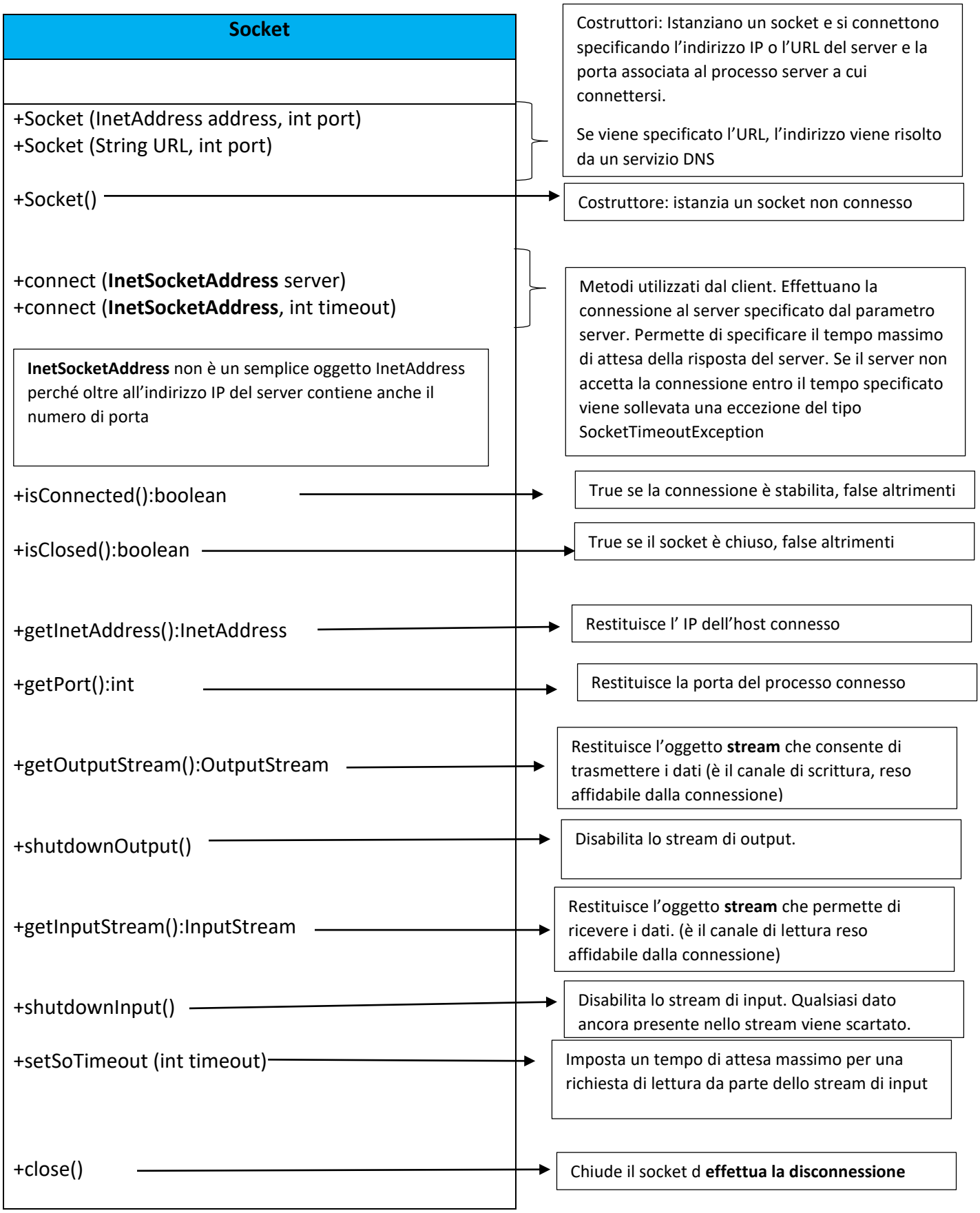
Nel seguente schema viene riportata la sequenza di invocazioni dei metodi dai vari oggetti coinvolti che porta a stabilire la connessione (i nomi degli oggetti sono quelli usati nell'esempio):



A questo punto la connessione è stabilita

Fase 2: come trasmettere i dati con i socket

Ora vediamo i metodi della classe Socket che consente la comunicazione fra client e server



Dopo aver stabilito la connessione, i due socket si scambiano dati mediante i due oggetti di classe InputStream e OutputStream.

InputStream espone dei metodi che consentono la ricezione di dati.

TABELLA **9**

Metodo	Descrizione
`int available()`	Restituisce il numero di byte ricevuti dal client e ancora presenti nello *stream*.
`int read(byte[] data)`	Legge dallo *stream* un *array* di byte; restituisce il numero di byte letti, −1 se lo *stream* è terminato perché il client ha chiuso la connessione.
`void close()`	Chiude lo *stream*. e rilascia le risorse

OutputStream espone dei metodi per la trasmissione dei dati

TABELLA **10**

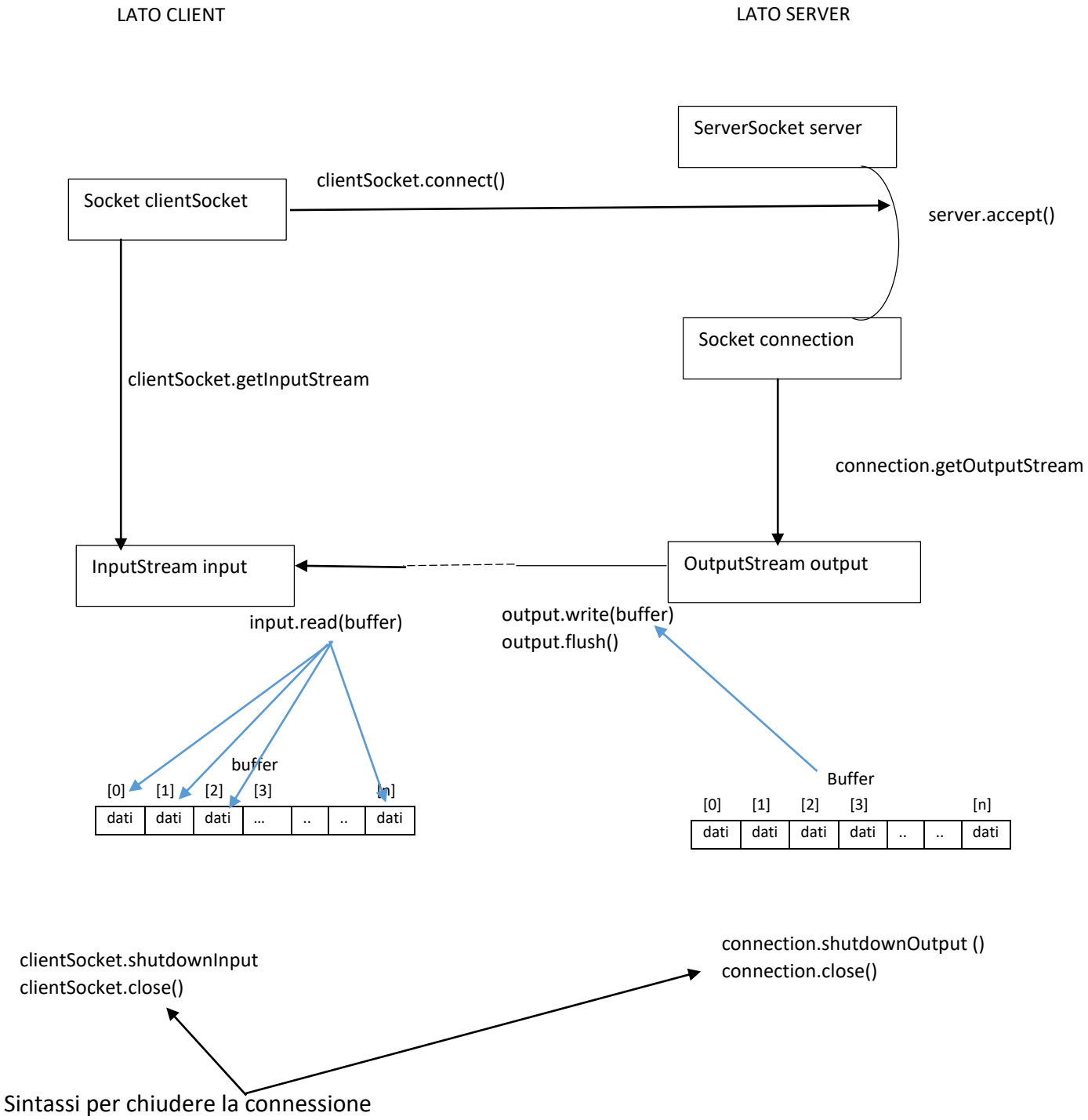
Metodo	Descrizione
`void flush()`	Invia al client i byte presenti nello *stream*.
`void write(byte[] data)`	Scrive nello *stream* un *array* di byte.
`void close()`	Chiude lo *stream*.

invia eventuali byte presenti nel buffer non ancora scritti

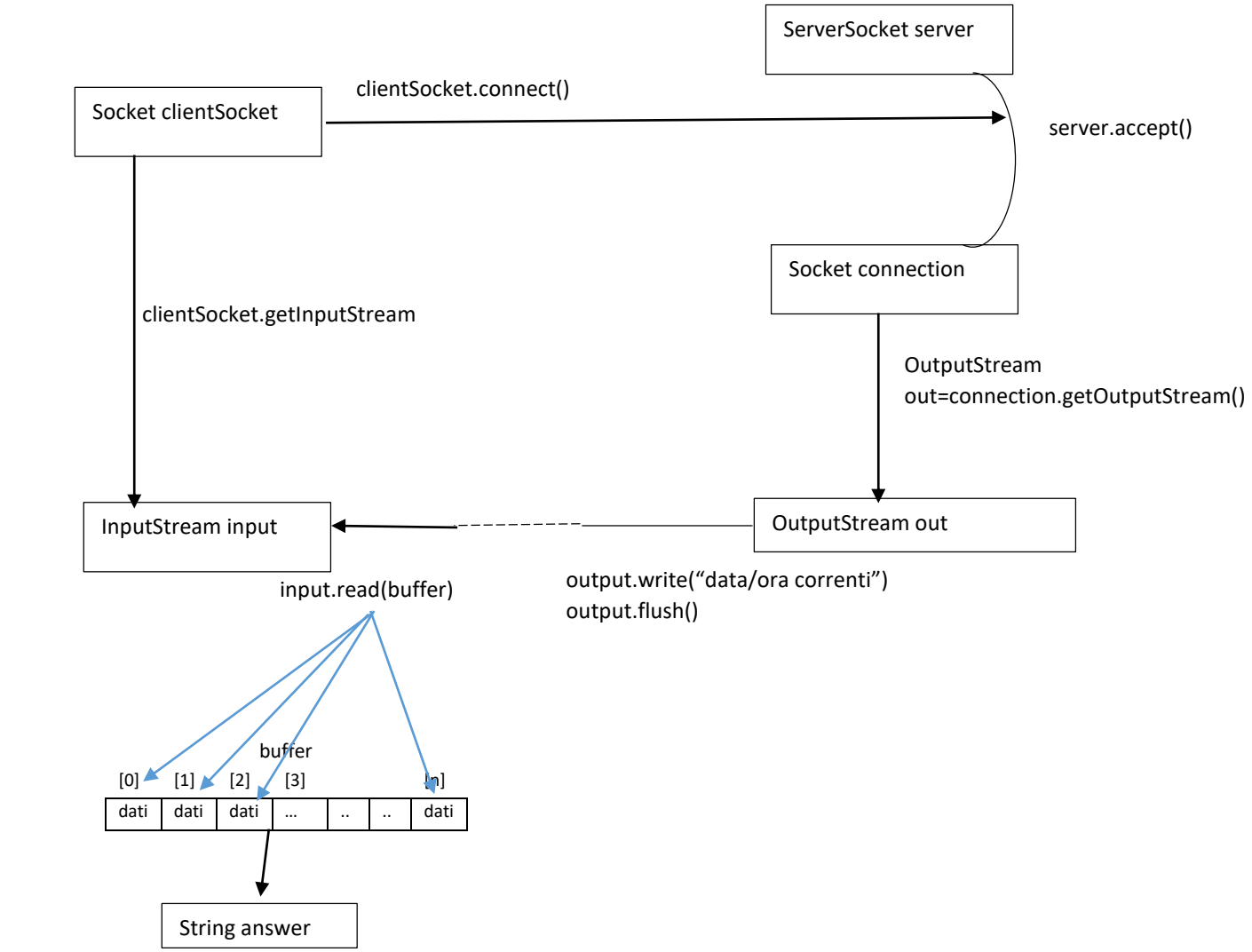
Grazie alle istanze di queste due classi è quindi possibile lo scambio di dati fra i due socket connessi.

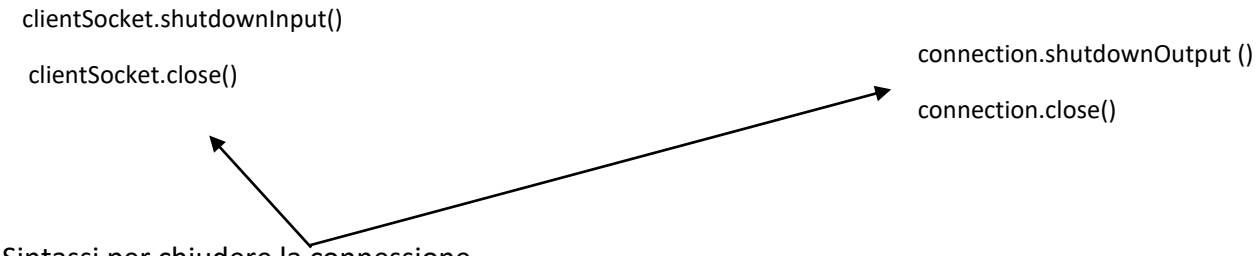
I dati trasmessi dal server (e ricevuti dal client) sono array di byte, per questo motivo vengono istanziati sia nel client sia nel server appositi array di byte chiamati buffer che contengono i dati trasmessi (e ricevuti).

Schema generico di connessione e trasmissione dati TCP:



Schema specifico per il protocollo DateTime





Sintassi per chiudere la connessione

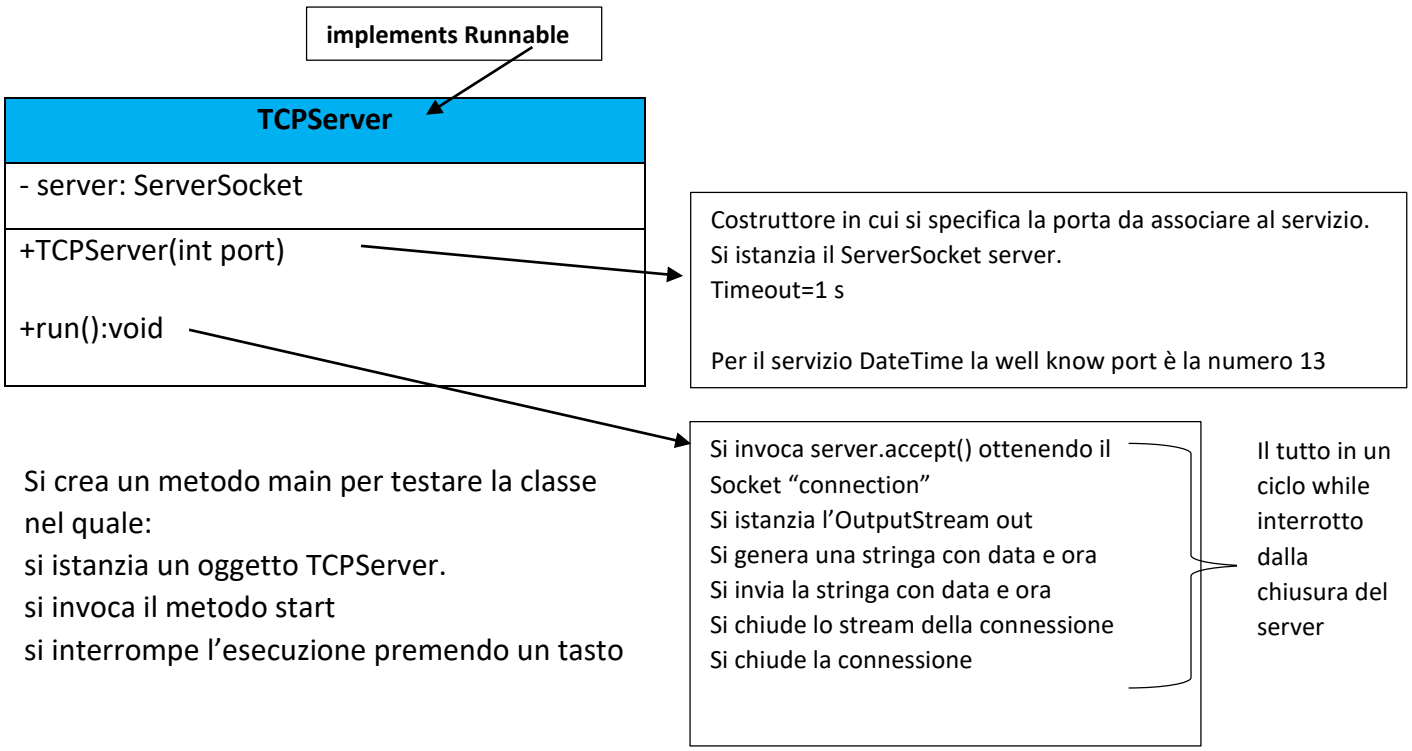
Come già sottolineato, il metodo connect() della classe Socket accetta come parametro non un semplice InetAddress, ma un oggetto di classe InetSocketAddress, che rappresenta in modo congiunto l'indirizzo IP e la porta logica del server.

I metodi principali della classe InetSocketAddress sono:

TABELLA **11**

Metodo	Descrizione
`InetSocketAddress(InetAddress address, int port)` `InetSocketAddress(String URL, int port)`	Costruttore: permette di specificare l'indirizzo IP, o in alternativa l'URL che viene risolto utilizzando un servizio DNS, e il numero di porta TCP.
`InetAddress getAddress()`	Restituisce l'indirizzo IP.
`int getPort()`	Restituisce il numero di porta TCP.
`boolean isUnresolved()`	Verifica se l'URL del server è stato risolto o meno mediante un servizio DNS.

Fase 3: realizzazione del server e del client



Si crea un metodo main per testare la classe nel quale:
si istanzia un oggetto TCPServer.
si invoca il metodo start
si interrompe l'esecuzione premendo un tasto

Testiamo il client con un telnet:

Comando per il telnet da CMD:

```
C:\Users\gian>telnet 127.0.0.1 13
```

Risultato:

```
2021-11-20T11:10:20.235855800


Connessione all'host perduta.
```

CODICE:

```
22  public class TCPserver implements Runnable
23  {
24      private ServerSocket server;
25
26      public TCPserver(int port) throws IOException
27      {
28          server=new ServerSocket(port);
29          server.setSoTimeout(1000);
30      }
31
32      public void run()
33      {
34          //Socket del server generato dalla richiesta di connessione
35          Socket connection = null;
36          byte[] outputBuffer=new byte[8192];
37
38          while(!Thread.interrupted())
39          {
40              try
41              {
42                  connection=server.accept();
43                  //se arriva la richiesta, istanzio un output stream (di byte)
44                  //porei istanziare un outputStreamWriter
45                  OutputStream out=connection.getOutputStream();
46                  String adesso=LocalDateTime.now().toString();
47                  outputBuffer=adesso.getBytes(charsetName: "UTF-8");
48                  out.write(b: outputBuffer);
49                  out.flush();
50                  System.out.println(x: "Risposta inviata");
51                  connection.shutdownOutput();
52                  connection.close();
53              }
54              catch (SocketTimeoutException ex)
55              {
56                  System.out.println(x: "Timeout");
57              }
58              catch (IOException ex)
59              {
60                  System.out.println(x: "Errore di connessione");
61              }
62              finally
63              {
64                  if (connection!=null && !connection.isClosed())
65                  {
66                      try
67                      {
68                          connection.close();
69                      }
70                      catch (IOException ex)
71                      {
72                          System.out.println(x: "Errore nella chiusura d
73                      }
74                  }
75              }
76          }
77          //Chiudo il serverSocket quando il ciclo viene interrotto
78          try
79          {
80              server.close();
81          }
82          catch (IOException ex)
83          {
84              System.out.println(x: "Errore nella chiusura del socket");
85          }
86      }
87
```

Rimane in attesa di una richiesta di connessione. Dopo 1 s solleva una SocketTimeoutException. Se arriva la richiesta istanzia un nuovo socket “connection” sul quale avverrà la comunicazione con il client.

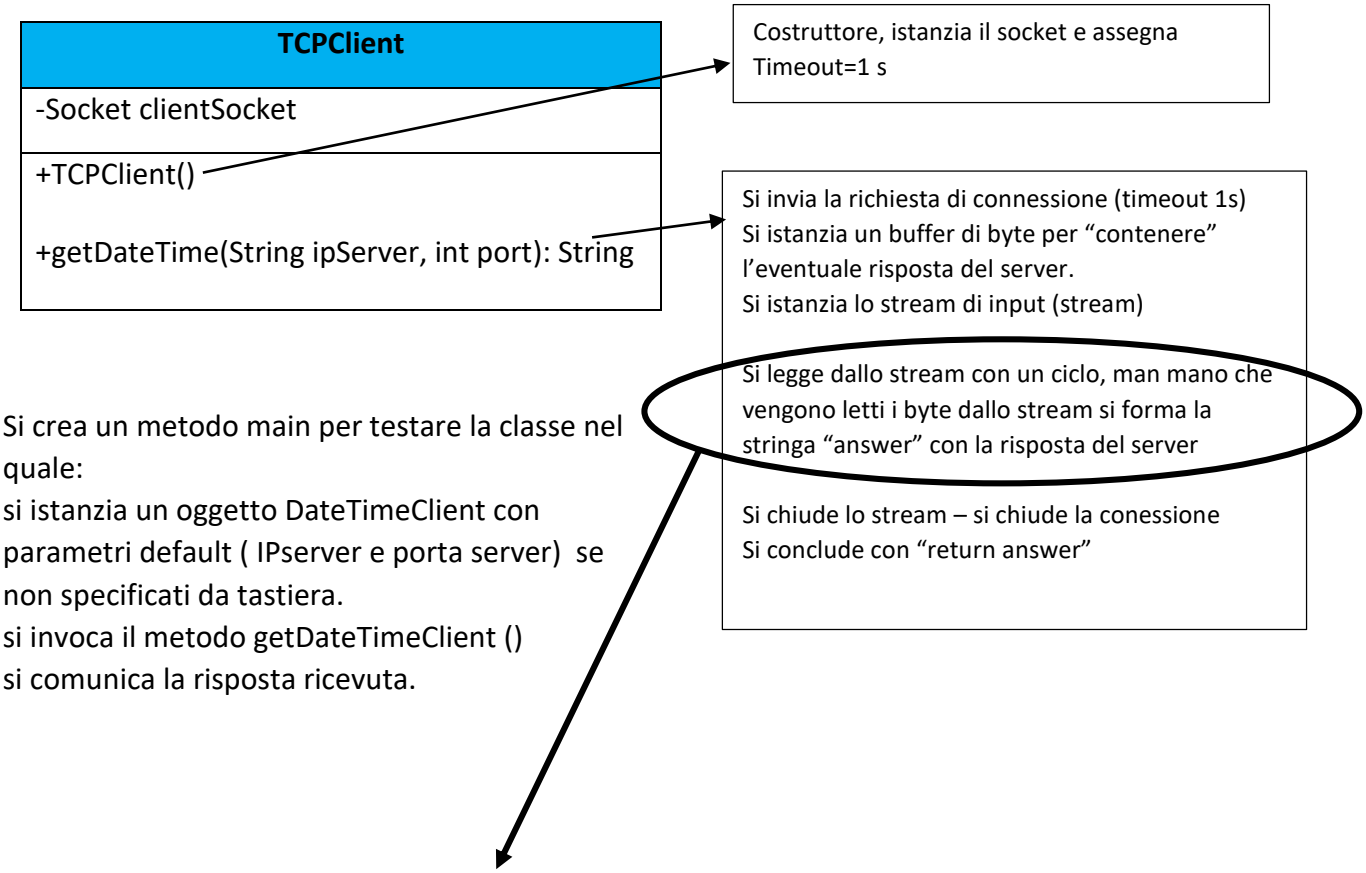
Interrompe il flusso di scrittura (OutputStream). Questa operazione è detta semi chiusura del socket. Il socket, una volta trasmessi gli ultimi dati scritti nel flusso, non consentirà di scriverne altri (generando eventualmente IOException) ma consentirà che avvenga lo normale procedura di chiusura della connessione a 4 vie

Chiude il socket. Questo socket non sarà più disponibile per ulteriori operazioni di scrittura/lettura sulla rete. Per comunicare di nuovo sarà necessario istanziare un nuovo socket. Con questa istruzione anche gli InputStream e OutputStream associati vengono chiusi

Questo finally va a chiudere il socket nel caso in cui il socket non fosse stato chiuso in precedenza a causa di un’eccezione

```
90      public static void main(String[] args)
91      {
92          Scanner tastiera=new Scanner(source:System.in);
93
94          try
95          {
96
97              TCPServer server=new TCPServer(port: 13);
98              Thread serverThread=new Thread(task: server);
99              serverThread.start();
100             tastiera.nextLine();
101             serverThread.interrupt();
102             serverThread.join();
103             System.out.println(x: "Server interrotto");
104         }
105         catch (IOException ex)
106         {
107             System.out.println(x: "Impossibile avviare il server");
108         }
109         catch (InterruptedException ex)
110         {
111             System.out.println(x: "Impossibile interrompere il server");
112         }
113     }
114
115 }
116
```

Realizziamo il client TCP a p. 62



Si crea un metodo main per testare la classe nel quale:
si istanzia un oggetto DateTimeClient con parametri default (IPserver e porta server) se non specificati da tastiera.
si invoca il metodo getDateTimeClient ()
si comunica la risposta ricevuta.

Come avviene la lettura dei byte dallo stream di input (stream):

```
while ((n = stream.read(buffer)) != -1) {
   fragment = new String(buffer, 0, n, "ISO-8859-1");
   answer = answer + fragment;
}
```

oppure "UTF-8"

Si invoca il metodo "read" dello stream che pone nel buffer i byte ricevuti dallo stream. Il metodo restituisce il numero di byte letti, non ci interessano, ci interessa solo quando il risultato è -1 perché indica che lo stream è terminato perché la connessione è stata chiusa.

Ogni volta che vengono letti dei byte si costruisce un "pezzo" di risposta con la stringa fragment (il costruttore String utilizzato converte una array di byte in una stringa secondo lo specifico charset).

Ogni nuovo fragment si aggiunge ad answer per formare la risposta

Il tutto viene ripetuto finchè dallo stream non arrivano più byte (stream.read(buffer)==-1)

Due osservazioni:

1. **Il socket del client deve essere istanziato senza parametri (senza IP e senza porta) del server.**
   Tali parametri vanno specificati quando si effettua la richiesta della connessione (ovvio, non puoi creare un socket verso un server se prima non gli hai chiesto la connessione!)

2. **Anche lo stream lo puoi istanziare dal socket solo dopo che è stata accettata la connessione (diciamo che prima che venga accettata la connessione il socket del client non esiste!)**

Codice:

```
21  public class TCPClient
22  {
23      private Socket clientSocket;
24      private final int MAX_BYTE=1024;
25
26      public TCPClient() throws SocketException
27      {
28          clientSocket=new Socket();
29          clientSocket.setSoTimeout(timeout: 1000);
30      }
31
32      public String getDateTime(String serverAddressString, int port) throws IOException
33      {
34          InetSocketAddress serverAddress=new InetSocketAddress(hostname: serverAddressString, port);
35          InputStream inputStream;
36          byte[] inputBuffer=new byte[MAX_BYTE];
37          int n;
38          String fragment;
39          String responseMessage="";
40
41          clientSocket.connect(endpoint: serverAddress,timeout: 1000);
42          inputStream=clientSocket.getInputStream();
43          while((n=inputStream.read(b: inputBuffer))!=-1)
44          {
45              fragment=new String(bytes: inputBuffer, offset:0, length:n, charsetName: "UTF-8");
46              responseMessage+=fragment;
47          }
48          clientSocket.shutdownInput();
49          clientSocket.close();
50          return responseMessage;
51      }
```

Timeout per quando il socket richiede la connessione

Il metodo connect(), se il server è spento, genera una eccezione di tipo IOException immediatamente (non dopo il timeout).

Timeout dopo che la connessione è stata stabilita

```
53      public static void main(String[] args)
54      {
55          String IPServer="127.0.0.1";
56          int serverPort=13;
57
58          String responseMessage;
59          try
60          {
61              TCPClient client=new TCPClient();
62              responseMessage=client.getDateTime(serverAddressString:IPServer, port: serverPort);
63              System.out.println(x: responseMessage);
64          }
65          catch (SocketException ex)
66          {
67              System.out.println(x: "Impossibile istanziare il socket");
68
69          }
70           catch (SocketTimeoutException ex)
71          {
72              System.out.println(x: "Il server non risponde");
73          }
74          catch (IOException ex)
75          {
76              System.out.println(x: "Errore di comunicazione");
77          }
78      }
79  }
```

2. SERVER TCP CONCORRENTI IN LINGUAGGIO JAVA

Il server che abbiamo realizzato nell'esempio precedente (classe TCPServer) funziona nel seguente modo:

1. Attende una richiesta dal client
2. Quando riceve una richiesta (metodo accept() della classe ServerSocket) istanzia un socket con il quale si connette al client, risponde inviando un messaggio (il DateTime nell' esempio), chiude la connessione.
3. Ritorna ad attendere un'altra richiesta (ciclo while)

Questo funzionamento consente di realizzare solo semplici protocolli (come il DateTime utilizzato nell'esempio), ma nel caso più comune, in cui si vuole che la comunicazione fra client e server si protragga nel tempo, per consentire la comunicazione fra i due host, il server dovrebbe consentire la gestione di una connessione che, una volta instaurata, rimanga attiva nel tempo (connessione **persistente**) e contemporaneamente consenta al server di accettare ulteriori richieste di connessione da altri client. Il server, in questo modo, quindi consentire la connessione di più client contemporaneamente.

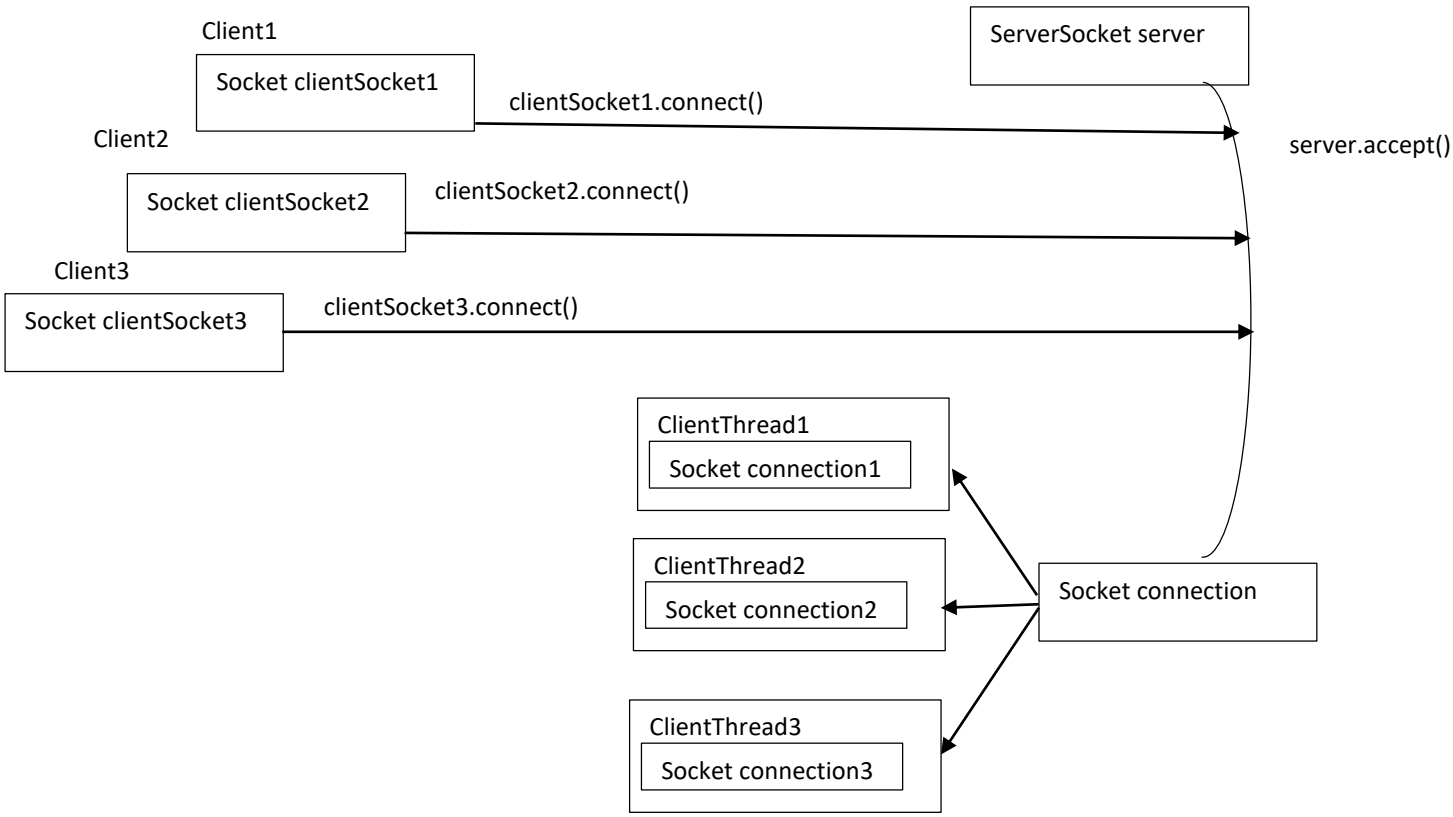
Per fare questo è necessario realizzare un server **concorrente**, ossia un server che, quando arriva la richiesta da parte del client, gestisca la comunicazione con "quello" specifico client per mezzo di un apposito thread, mentre contemporaneamente ritorni ad ascoltare (eseguendo il ciclo while) l'arrivo di eventuali richieste da parte di altri client.

Quindi La parte di gestione della comunicazione fra client e server deve avvenire in un apposito thread, che viene creato ogni volta che un client si connette al server.

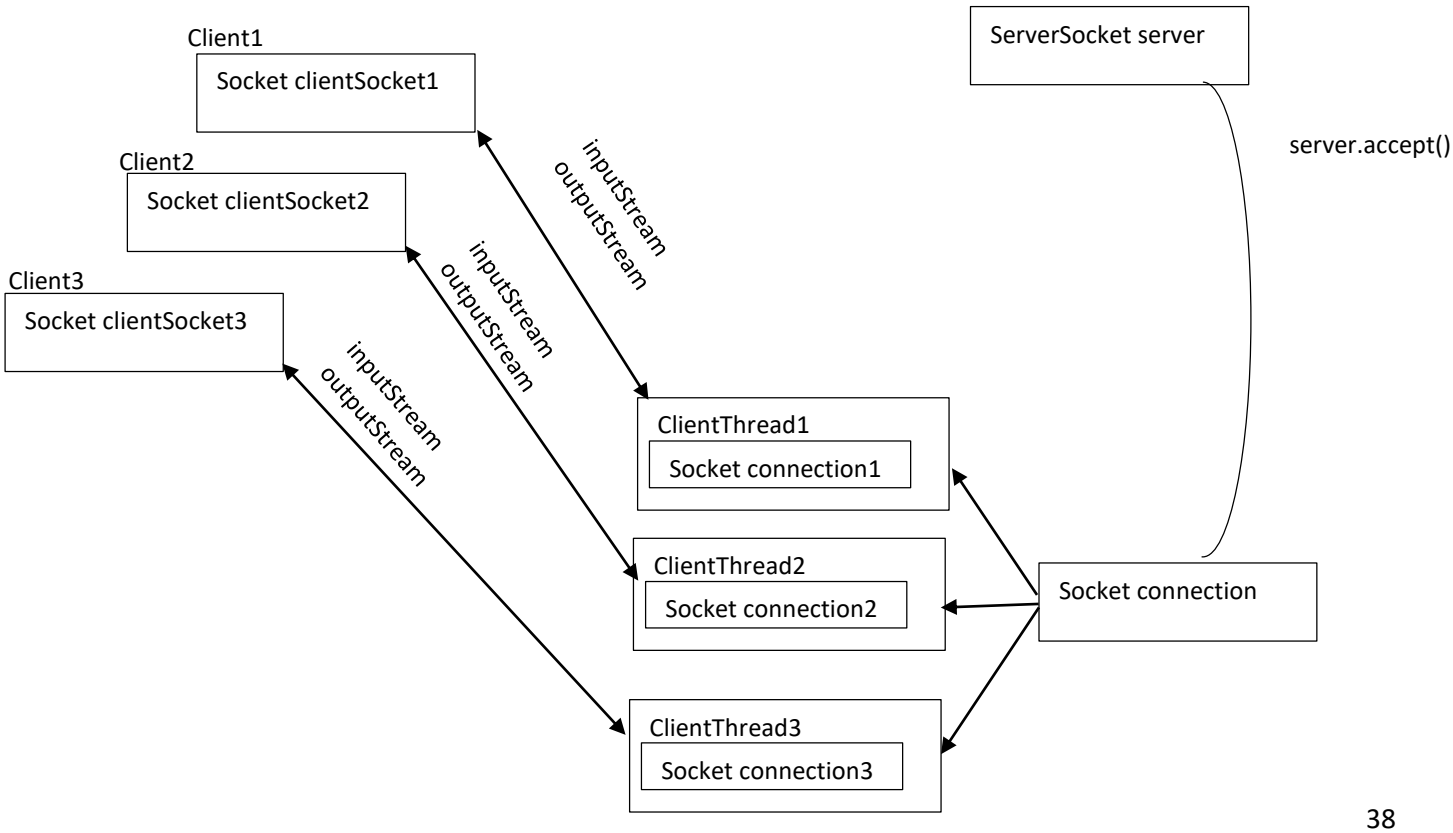
Chiamiamo la classe che implementa questi thread "ClientThread". Un thread istanza della classe ClientThread, quindi, verrà istanziato dal TCPserver ogni volta che esso riceverà una richiesta dal client. In altre parole: per ogni richiesta effettuata da un client al server, il server istanzierà un nuovo oggetto (istanza della classe ClientThread) per consentire la comunicazione con quello specifico client. (Sarà poi il client ad avviare la chiusura della connessione, situazione rilevata dallo stream di input che legge -1). Attenzione: gli oggetti della classe ClientThread vengono istanziati dal processo Server, non facciamoci ingannare dai nomi.

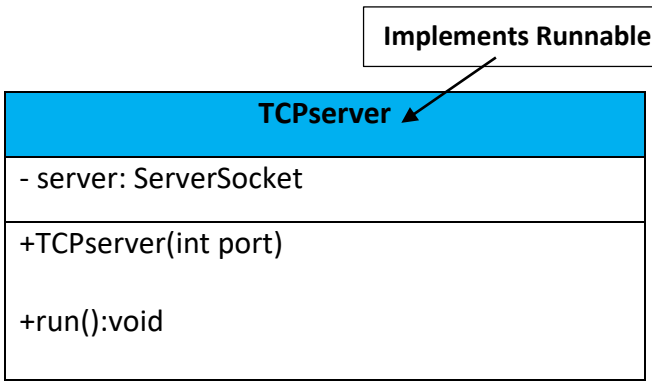
Andiamo quindi a realizzare il TCPserver concorrente (senza client, testeremo il suo funzionamento con telnet), il cui funzionamento possiamo riassumere con il seguente schema:

Creazione dei ClientThread da parte del ServerSocket



La connessione ora è fra ciascun ClientThread del server e ciascun client





Costruttore:
Si istanzia il ServerSocket
Si assegna Timeout=1 s

-Si invoca server.accept() ottenendo il Socket connection

-Si istanzia un oggetto ClientThread clientThread(connection)

-Si avvia il clientThread

Il tutto in un ciclo while che permette di istanziare un nuovo clientThread per ogni richiesta di connessione

Si crea un metodo main per eseguire il server nel quale:
si istanzia un oggetto TCPserver(port) si avvia il thread (TCPServer.start()) che è il main thread del server

il main thread del server viene interrotto dalla pressione di un tasto sulla tastiera

```
21  public class TCPServer implements Runnable
22  {
        private ServerSocket server;
24
25      public TCPServer(int port) throws IOException
26      {
27          server=new ServerSocket(port);
28          server.setSoTimeout(1000);
29      }
30
```

```
    public void run()
    {
        Socket connection;

        while(!Thread.interrupted())
        {
            try
            {
                connection=server.accept();
                System.out.println("Richiesta da "+connection.getInetAddress().toString()+":"+connection.getPort());
                ClientThread clientThread=new ClientThread(connection);
                Thread clientConnection=new Thread(clientThread);
                clientConnection.start();
            }
            catch (SocketException ex)
            {
                System.out.println("Impossibile istanziare il socket");
            }
            catch (SocketTimeoutException ex)
            {
                System.out.println("Timeout!");
            }
            catch (IOException ex)
            {
                System.out.println("Errore di comunicazione ");
            }
        }
        try
        {
            server.close();
        }
        catch (IOException ex)
        {
            System.out.println("Impossibile chiudere il socket");
        }
    }

    public static void main(String[] args)
    {
        Scanner tastiera=new Scanner(System.in);
        try
        {
            TCPServer server=new TCPServer(2000);
            Thread serverThread=new Thread(server);
            serverThread.start();
            tastiera.nextLine();
            serverThread.interrupt();
            serverThread.join();
            System.out.println("Server interrotto");
        }
        catch (IOException ex)
        {
            System.out.println("Errore di comunicazione");
        }
        catch (InterruptedException ex)
        {
            System.out.println("Impossibile interrompere il server");
        }

    }
}
```

Creiamo la classe ClientThread che realizza un EchoServer. Tutto ciò che viene inviato dal client viene ritornato dal server.

Implements Runnable

ClientThread
- connection: Socket - input: InputStream -output: OutputStream
+ClientThread(connection: Socket) +run():void

Costruttore:
Si assegna il parametro connessione all'attributo connessione
Si istanzia input (da connessione.getInputStream)
Si istanzia output (da connessione.getOutputStream)

Si leggono (con l'inputStream) i dati inviati dal client.
Man mano che arrivano, gli stessi dati vengono inviati al client (con l'outputStream)

Il ciclo di lettura/invio dei dati è questo

```
public void run() {
  int n;
  byte [] buffer = new byte[1024];
  try {
    // ciclo di ricezione dei dati dal client
    while ((n = input.read(buffer)) != -1) {
      if (n > 0) {
        // invio dei dati ricevuti al client
        output.write(buffer, 0, n);
        output.flush();
      }
    }
  }
  catch (IOException exception) {
  }
  try {
    System.out.println("Connessione chiusa!");
    input.close();
    output.close();
    connection.shutdownInput();
    connection.shutdownOutput();
    connection.close();
  }
  catch (IOException _exception) {
  }
}
}
```

Osservazione: il metodo accept del ServerSocket restituisce sempre un nuovo Socket, quindi si ha un nuovo Socket "connection" (da passare al nuovo thread ClientThread) ogni volta che un client invia una richiesta di connessione

```
19      */
20      public class ClientThread implements Runnable
21      {
22          private Socket connection;
23          private InputStream input;
24          private OutputStream output;
25
26          public ClientThread(Socket c) throws SocketException, IOException
27          {
28              connection=c;
29              input=connection.getInputStream();
30              output=connection.getOutputStream();
31          }
32
```

```
33          public void run()
34          {
35              int n;
36              byte[] buffer=new byte[1024];  //max 1 kbyte
37              try
38              {
39                  while((n=input.read(buffer))!=-1)
40                  {
41                      if (n>0)
42                      {
43                          output.write(buffer,0,n);
44                          output.flush();
45                      }
46
47                  }
48              }
49              catch (IOException ex)
50              {
51                  System.out.println("Errore di comunicazione");
52              }
53
54              try
55              {
56                  connection.shutdownOutput();
57                  connection.shutdownInput();
58                  connection.close();
59              }
60              catch (IOException ex)
61              {
62                  Logger.getLogger(ClientThread.class.getName()).log(Level.SEVERE, null, ex);
63              }
64
65          }
66      }
```

Come testare il clientThread con Telnet

1. Apri una connessione telnet.

2. Ogni volta che digiti una lettera, l'Echo ti rimanda in dietro la stessa lettera. Dalla seconda lettera il telnet mostra anche ciò che viene digitato sulla tastiera

3. Test interessante: apri due finestre cmd ognuno con un client telnet (telnet 127.0.0.0 2000 entrambi). Verifica il funzionamento dell'echo server su entrambi.

   Quando esci da uno dei due telnet chiudendo la finestra del prompt dei comandi, la connessione di quel telnet viene chiusa ma la connessione con l'altro telnet continua a funzionare (thread concorrenti di cui uno viene chiuso).

4. Altro test interessante (che evidenzia il fatto che l'istanza della classe TCPServer ha come unico scopo quello di istanziare la connessione, poi la connessione è autonoma in quanto eseguita su un thread separato.) Se attivo due connessioni (due finestre cmd con due telnet) e poi chiudo il server (**inserendo un input vuoto da console java per arrestare il server, occhio: NON con il pulsante rosso terminate perché questo elimina tutti gli elementi in esecuzione, quindi anche la connessione!),** i due EchoServer sui telnet continuano a funzionare, la differenza è che ora, avendo chiuso il server, non è possibile realizzare nuove connessioni ("Le connessioni esistenti vengono chiuse dal client chiudendo la finestra Telnet").

Il pattern di progettazione thread-pool

Ricordo che, come studiato l'anno scorso in TPS, con il termine i "pattern" si indicano delle soluzioni software progettuali già pronte.

Il pattern thread-pool è un pattern per la gestione di più thread. Tale pattern prevede di realizzare un "gestore di thread" in grado di creare un numero adeguato ma limitato di worker-thread (thread in esecuzione) a ciascuno dei quali viene assegnato un task (compito).

Nel nostro caso specifico il task di ogni thread è quello di gestire la comunicazione con uno specifico client.

Il pattern di progettazione thread-pool ci consente dunque, nel caso di un server TCP, di gestire a livello software la situazione in cui vi sono molteplici richieste di connessione da parte dei client verso un server, e per la gestione di ogni connessione viene istanziato un thread.

Il pattern thread-pool gestisce le molteplici richieste di connessione nel seguente modo:

- Si istanzia nel server un oggetto di classe **ThreadPoolExecutor** (insieme di thread) chiamato *threads* indicando **il massimo numero** di thread gestibili contemporaneamente. Questo oggetto è il "gestore di thread".
- Ogni volta che un client effettua una richiesta di connessione, se la connessione viene accettata si istanzia nel server un nuovo thread (che possiamo chiamare *ClientThread*) per gestire la comunicazione client-server.
- Il ClientThread viene inviato all'oggetto *threads.*
- Se nel ThreadPoolExecutor "ci sono posti disponibili", allora il ClientThread può essere gestito, esso viene avviato dall' oggetto threads, altrimenti, se non c'è "posto" per la gestione del ClientThread, esso viene messo in coda e verrà gestito quando si "libererà un posto". Un posto si "libera" quando un ClientThread in esecuzione viene terminato.
- Per arrestare il thread-pool threads, si invoca su di esso il metodo shutdown. Questo metodo non interrompe i thread in esecuzione, semplicemente non consente di aggiungere nuovi ThreadClient al thread-pool.

Dal punto di vista del codice, per realizzare il pattern Thread Pool, ad un Server TCP vengono apportate le seguenti modifiche:

Si aggiunge un attributo di classe ExecutorService:

```
private ExecutorService threads;
```

Nel costruttore del server si istanzia l'oggetto di classe ExecutorService (è il thread pool) specificando il massimo numero di thread da gestire. Il codice per istanziarlo è il seguente:

```
threads=Executors.newFixedThreadPool(MAX_NUMERO_THREADS);
```

Quando viene accettata una richiesta di connessione da un client , si istanzia un nuovo thread che gestisce tale connessione (ClientThread), e tale thread viene "spedito " al thread pool. Il codice è il seguente:

```
connection=server.accept();
Thread clientThread=new Thread(new ClientThread(connection));
```

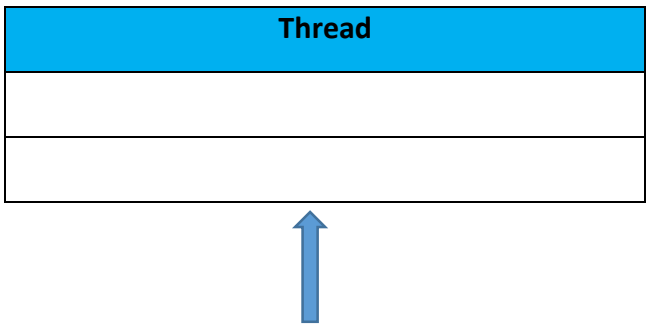
```
try
{
    threads.submit(clientThread);//sostituisce clientThread.start()
}
     catch (RejectedExecutionException e)
{
     connection.close();
}
```

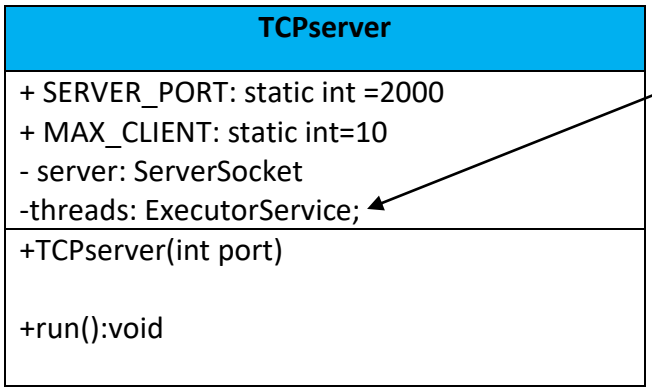
Nel caso in cui il ClientThread venga rifiutato (ad esempio perché il thread pool è stato arrestato con shutdown()), viene sollevata l'eccezione *RejectedExecutionException* (not checked). Nel codice di gestione dell'eccezione viene chiusa la connessione che ha dato origine al ClientThread.

Quando si chiude il serverSocket all'arresto del server è necessario dunque arrestare il thread pool invocando su di esso il metodo shutdown().

```
try
{
     threads.shutdown();
     server.close();
}
     catch (IOException e)
{
     System.out.println("Errore nella chiusura del server");
}
```

Realizziamo dunque di nuovo la classe TCPserver implementando il pattern thread-pool (codice a p. 68)





E' un reference all'interfaccia ExecutorService. Non è un'istanza (non si potrebbe). il reference punterà poi a un oggetto di classe ThreadPoolExecutor (Upcasting)

Costruttore:
Si istanzia il ServerSocket server (porta port, timeout 1000)

Si istanzia threads invovando il metodo statico *Executors.newFixedThreadPool(MAX_CLIENT)*

Si crea un metodo main nel quale:
si istanzia un oggetto TCPserver(SERVER_PORT)
si avvia il thread (TCPServer.start())

il thread viene interrotto dalla pressione di un tasto sulla tastiera

-Si invoca server.accept() ottenendo il Socket connection

-Si istanzia un ClientThread clientThread(connection)

-SI invia il clientThread al thread-pool invocando il metodo *threads.submit (clientThread)*

(questo sostituisce clientThread.start())

Il tutto in un ciclo while che permette di istanziare e spedire un nuovo clientThread per ogni richiesta di connessione

ESERCIZIO: Applicando il pattern thread-pool andiamo a realizzare un server che implementa le quattro operazioni base di una calcolatrice e un client che richieda il servizio.

Il protocollo da implementare è testuale ed è il seguente:

Per realizzare una calcolatrice TCP è possibile implementare un protocollo testuale basato sui seguenti comandi terminati dai caratteri CR (codice ASCII 13) ed LF (codice ASCII 10):

Comando	Operazione
ADD,*X*,*Y*	Addizione (*X* + *Y*)
SUB,*X*,*Y*	Sottrazione (*X* – *Y*)
MUL,*X*,*Y*	Moltiplicazione (*X* × *Y*)
DIV,*X*,*Y*	Divisione (*X* : *Y*)

Esempi di comandi del protocollo sono le stringhe «`ADD,1.5,-0.5`», «`SUB,0,1`», «`MUL,1.125,8`» e «`DIV,1024,2.0`». Il server che implementa la calcolatrice risponderà con una stringa contenente il risultato numerico o con la stringa «`ERROR`»; in entrambi i casi la risposta sarà terminata dai caratteri CR ed LF.
La seguente classe Java implementa il protocollo della calcolatrice mediante un *thread* di gestione della comunicazione con il client:

Anche il comando di richiesta deve terminare con CR (ASCII 13) e LF(carattere ASCII 10), è necessario perché il server comprenda la fine del messaggio inviato.

I comandi vengono inviati dal client al ClientThread dopo la richiesta di connessione al Server. Il ClientThread riceve il comando, svolge il calcolo, risponde con il risultato del calcolo in forma testuale.

Ogni volta che il client vuole usufruire dei servizi del server (un'operazione), apre una connessione (ogni request crea una connessione). Il client chiude la connessione in seguito alla risposta del client. In questo modo le risorse del server sono impiegate solamente per il tempo strettamente necessario all'esecuzione della richiesta inviata.

ESERCIZIO 18 P. 88 MAGAZZINO: (occhio al problema della sincronizzazione)

18 DESIGN CODING Il sistema informatico del magazzino di un grande sito web di *e-commerce* mantiene per ogni prodotto in vendita identificato da un codice numerico la quantità disponibile. Dopo aver definito il protocollo testuale di comunicazione tra client e server, realizzare in linguaggio Java un server TCP concorrente che legga inizialmente da file l'elenco dei prodotti con le relative quantità e gestisca, prevedendo una corretta gestione della concorrenza tra thread, i seguenti comandi ricevuti dai client:

A. restituisca la quantità presente di un prodotto specificato nell'interrogazione;
B. modifichi la quantità di un prodotto specificato nel comando.

1. Prima ipotizza di avere i prodotti in un arrayList di oggetti prodotto (attributi: codice, quantità)

2. Realizza il protocollo di comunicazione, ad esempio:

COMANDO REQUEST	RISPOSTE	
CODICE_PRODOTTO;\n\r	codice del prodotto; quantità del prodotto	ERROR se prodotto non presente
CODICE_PRODOTTO; QUANTITA;\n\r	OK se la quantità è stata aggiornata	ERROR se prodotto non presente o non è stato possibile aggiornare il file

3. Aggiungi la parte di lettura/scrittura da file

ESERCIZIO: rifare esercizio "oggetti mostra" simulazione verifica con un server TCP concorrente con Thread Pool