Esercitazione 2020-10-16

Nella prima parte della lezione viene mostrata la soluzione dell'esercizio 3.2, ovvero la realizzazione della chat p2p con l'utilizzo della funzione **select()**.

Socket orientate ai flussi su java

Le classi per la programmazione socket orientate ai flussi sono diverse rispetto a quelle UDP. Utilizzeremo quindi due classi, entrambe diverse da quelle viste ieri.

- **Server socket - ServerSocket()** : è la classe che da vita alla socket che precedentemente abbiamo definito "di benvenuto", quella sulla quale il server si mette in ascolto per richieste di connessione.
- **Socket connessa - Socket()** : è il tipo di un oggetto restituito dalla accept(); immaginiamo che il metodo accept() sia presente nella classe ServerSocket() e che una volta invocato restituisce un oggetto di tipo **Socket()** . Gli oggetti di tipo **Socket()** sono utilizzati come endpoint in una connessione orientata ai flussi. Un oggetto di tipo ServerSocket() viene invece usato solo per accettare richieste di connessione.

Ipotizziamo di scrivere un Web Server in java - Orientato ai flussi

Il servizio prevede che il n di porta utilizzato sia il **numero 80**

Lato Server

ServerSocket()

```
ServerSocket welcomeSocket = new ServerSocket(80);  //usiamo la classe ServerSocket e la porta 80
```

Anche in questo caso il costruttore è caricato di responsabilità; abbiamo infatti una serie di invocazioni non visibili al programmatore. Ad esempio è presente sicuramente la **creazione della socket**, una **bind()** sulla porta, ed infine la **listen()**. In particolare la listen() differenzia il costruttore di questa classe da quella usata nel client, visto che il client non ascolta su nessuna porta.

Una volta creata la welcomeSocket possiamo invocare i metodi previsti nella classe **ServerSocket()**, in particolare **accept()**.

accept()

Il metodo accept sospende il server in attesa di connessioni.

```
// creazione della socket, bind e listen
Socket connectionSocket = welcomeSocket.accept();
```

La accept ritorna una nuova **socket** estendendo il contenuto di **welcomeSocket**; gli errori vengono notificati **lanciando un'eccezione di tipo IOException**.

read() e write()

Una delle differenze fondamentali rispetto alle socket orientate ai datagram in java, che rende questo modello più articolato da gestire, è nella comunicazione dei dati attraverso gli endpoint che andiamo a costruire.

Invocando la **accept()** otteniamo come valore di ritorno il riferimento ad un oggetto di tipo **Socket**, che viene memorizzato all'interno della variabile **connectionSocket**.

```
// creazione della socket, bind e listen
// accettiamo connessioni

SocketInputStream in = connectionSocket.getInputStream();
SocketOutputStream out = connectionSocket.getOutputStream();
```

Invochiamo su **connectionSocket** i metodi **getInputStream()** e **getOutputStream()**

L'I/O sequenziale è relizzato in Java attraverso il concetto di stream e di writers e readers.; se parliamo di comunicazione a byte (byte oriented) parliamo di **stream**, se invece parliamo di comunicazione a caratteri (character oriented - caratteri Unicode e non ASCII) parliamo di **writers e readers**.

In java sono previste due classi da cui discendono tutte le altre che sono **InputStream e OutputStream**, che sono classi **astratte**; sono astratte perchè prendono un **generico stream**, non associato al canale di comunicazione. Queste due classi possono essere specializzate prevedendo l'impiego di uno specifico canale di comunicazione. Ad esempio, **FileInputStream e FileOutputStream** sono due esempi di classi specializzate di InputStream e Outputstream; queste due classi, a differenza delle precedenti, sono delle classi **concrete**, questo perchè si fa riferimento ad uno specifico canale di comunicazione: File.

In maniera analoga possiamo far discendere **SocketInputStream e SocketOutputStream**.

Quando parliamo di "**stream**" facciamo riferimento che il **canale di comunicazione**, non per forza di rete, sia di tipo FIFO, ovvero che rispetta l'ordine di invio; anche il File è un canale di comunicazione di tipo Stream, e quindi FIFO.

Metodi ServerSocket

- `public ServerSocket(int port) throws Exception` crea una socket e la collega ad una porta
- `public ServerSocket(int port, int backlog) throws IOException`
- `public ServerSocket(int port, int backlog, InetAddress address) throws InetAddressException`
- `public Socket accept() throws IOException`
- `public void close() throws IOException`
- `public int getLocalPort()`
- `public InetAddress getInetAddress()`

🏁 0:35 lunga spiegazione sugli stream java

Lato Client

Per programmare un client con il modello orientato ai flussi, usiamo direttamente la classe **Socket()**:

```
String hostName = ... ;
Socket clientSocket = new Socket(hostName, 80);
```

In questo caso usiamo degli argomenti diversi rispetto a quelli usati con il server; questo perchè andiamo ad indicare l'indirizzo IP e la porta della socket server di benvenuto (welcomeSocket). Facendo ciò, avremo all'interno di clientSocket un riferimento di socket già connessa (creando una socket senza parametri la socket non sarebbe connessa).

All'interno del costruttore di **Socket()** troveremo sicuramente la creazione della socket stessa e della connect().

read() e write()

Lato client ci basta creare la socket e possiamo subito iniziare ad inviare dati, usando la funzione **read()**:

```
SocketInputStream in = clientSocket.getInputStream();
SocketInputStream out = clientSocket.getOutputStream();
```

Ognuno dei due stream (sia da client che da server) funzionano tendenzialmente allo stesso modo, solo in maniera duale. Ciò che scriviamo sull'output stream del client, viene letto dall'output stream del server (connection socket), e viceversa. Abbiamo quindi due stream unidirezionali che sono impiegabili grazie ad oggetti come OutputStream ed InputStream.

Grazie a questi stream possiamo addirittura inviare degli oggetti, usando **ObjectInputStream ed ObjectOutputStream**; se ad esempio ho un albero, possiamo trasferire l'intero albero, utilizzando il metodo **writeObject()** della classe **ObjectOutputstream** per inviare l'albero, e leggere dall'altro endpoint con il metodo **readObject()** di **ObjectInputStream**.

Questo perchè le difficoltà di trasferimento sono gestite dai metodi di conversione, e questo se la vede Java.

Quindi? Tante parole, pochi fatti; leggi il codice, è molto più semplice in pratica che in teoria:

Scriviamo il codice

Server

```
Class StreamServer{
   public static void main() throws Exception{
      String clientSentence;
      String capitalizedSentence;

      ServerSocket welcomeSocket = new ServerSocket(6789);

      while(true){
         Socket connectionSocket = welcomeSocket.accept();

         Scanner inFromClient = new Scanner(connectionSocket.getInputStream());  // leggiamo non da input ma dallo stream del client!

         PrintStream outToClient = new PrintStream(connectionSocket.getOutputStream());

         clientsentence = inFromClient.nextLine();

         capitalizedSentence = clientSentence.toUpperCase();

         outToClient.println(capitalizedSentence);

         connectionSocket.close();
      }
   }
}
```

Essenzialmente quello che facciamo è prima stabilire due Stream: **inFromClient** che è di tipo **Scanner**, usata per leggere, e **outToClient**, di tipo **PrintStream**, usata per scrivere.
Molto più semplice da usare, anche se inizialmente può sembrare un concetto difficile.

Client

```
Class  StreamClient{
   public static void main(){
      String sentence;
      String modifiedSentence;

      Scanner inFromUser = new Scanner(System.in);

      Socket clientSocket = new Socket("127.0.0.1", 6789);

      PrintStream outToServer = new PrintStream(clientSocket.getOutputStream());
      Scanner inFromServer = new Scanner(clientSocket.getInputStream());

      sentence = inFromUser.nextLine();

      outToServer.println(sentence);

      modifiedSentence = inFromServer.nextLine();
      sout.println(modifiedSentence);

      clientSocket.close();
   }
}
```

Esercizi 4.x

- **Esercizio 4.1:** Scrivere un programma java client-server per la moltiplicazione remota di due interi. Usare la comunicazione orientata allo stream.
- **Esercizio 4.2:** Scrivere la stessa applicazione del problema 4.2 usando la comunicazione orientata ai datagram.
- **Esercizio 4.3:** Cambia l'esercizio 4.1 in modo che il protocollo dell'applicazione sia separato dal codice richiesto per creare il canale della comunicazione. In questo modo, un'applicazione client/server può essere riusata utilizzando il codice richiesto per l'attivazione della connesione, siccome l'unica porzione di codice che cambia è quella dell'applicazione del protocollo
  Usa la seguente interfaccia per implementare il codice per l'implementazione del protocollo dell'applicazione:

  ```
  interface ProtocolHandler{
      public handle() throws IOException;
  }
  ```

- **Esercizio 4.4:**

Esercizi visti a 1:15 2020-10-16

---

fine lezione 11