Big Picture

PROJECT 1

Problem Set 1: Java Concepts, Thread, Socket Programming

Recitation 1: Java Concepts

Recitation 2: Java Threads and Socket Programming

Recitation 3: Project 1

Due Sunday: Sept. 3

Will be Released Next Tuesday: Sept. 8

Due Sunday: Sept. 3
Outline

• Communication via Sockets in Java

• Multi-threading in Java

• Coding a full Client-Server Example

  On Eclipse, we’ll code an “echo” TCP Server-Client Example
Communication via Sockets

• Sockets provide a communication mechanism between networked computers.

• A **Socket** is an end-point of communication that is identified by an **IP address** and **port number**.

• A **client** sends requests to a **server** using a **client socket**.

• A **server** receives clients’ requests via a **listening socket**.
Communication via Sockets

Person A (A’s home)  |  Person B (Guest)
---|---
Person A knocks the door  |  Person B enters
Person A Is Listening  |  
Person A Opens door  |  

(Note: The diagram illustrates the concept of communication via sockets, where one person knocks on the door and the other opens the door.)
Communication via Sockets

Person A (A’s home) - Person B (Guest)

- Person A knocks the door
- Person A is listening
- A “binds” to his home
- Person B sends a request to communicate
- Person B enters
- B is now “connected” with A

A accepts the request
Communication via Sockets

Server A

A binds to socket address:
(1) IP address
(2) Port number

Client B

Client B sends a request to communicate with the server

Server A is Listening to Requests

Server A accepts request

Client B is now connected with Server A
Communication via Sockets

When writing the code, these steps could be merged.

More on that later!
Socket Communication Recipe

Client socket

Service socket

Listening socket

serverSocket = new ServerSocket(port);
Socket server = serverSocket.accept();

Socket client = new Socket(serverName, port);
## ServerSocket Methods

<table>
<thead>
<tr>
<th>SN</th>
<th>Methods with Description</th>
</tr>
</thead>
</table>
| 1  | `public ServerSocket(int port)`  
Attempts to create a server socket bound to the specified port. An exception occurs if the port is already bound by another application. |
| 2  | `public ServerSocket()`  
Creates an unbound server socket. When using this constructor, use the `bind()` method when you are ready to bind the server socket. |
| 3  | `public void bind(SocketAddress host)`  
Binds the socket to the specified server and port in the `SocketAddress` object. Use this method if you instantiated the `ServerSocket` using the no-argument constructor. |
| 4  | `public Socket accept()`  
Waits for an incoming client. This method blocks until either a client connects to the server on the specified port or the socket times out, assuming that the time-out value has been set using the `setSoTimeout()` method. Otherwise, this method blocks indefinitely. |
| 5  | `public SocketAddress getLocalSocketAddress()`  
Returns the address of the endpoint this socket is bound to, or null if it not bound yet. |
| 6  | `public void close()`  
Closes the socket |

There are two ways to create and bind `ServerSocket`:

1) `ServerSocket(int port)`: which will create the socket and bind it with the given port

2) `InetSocketAddress(port) + ServerSocket() + bind(address)`
## Socket Methods

<table>
<thead>
<tr>
<th>SN</th>
<th>Methods with Description</th>
</tr>
</thead>
</table>
| 1  | `public Socket(String host, int port)`  
This method attempts to connect to the specified server at the specified port. If this constructor does not throw an exception, the connection is successful and the client is connected to the server. |
| 2  | `public Socket()`  
Creates an unconnected socket. Use the `connect()` method to connect this socket to a server. |
| 3  | `public void connect(SocketAddress host)`  
This method connects the socket to the specified host. This method is needed only when you instantiated the `Socket` using the no-argument constructor. |
| 4  | `public InputStream getInputStream()`  
Returns the input stream of the socket. The input stream is connected to the output stream of the remote socket. |
| 5  | `public OutputStream getOutputStream()`  
Returns the output stream of the socket. The output stream is connected to the input stream of the remote socket. |
| 6  | `public SocketAddress getLocalSocketAddress()`  
Returns the address of the endpoint this socket is bound to, or null if it is not bound yet. |
| 7  | `public void close()`  
Closes the socket, which makes this `Socket` object no longer capable of connecting again to any server. |

There are two ways to create and connect a client socket:

1) `Socket(String host, int port)`  
   • You can use “127.0.0.1” for local host
2) `InetSocketAddress(String host, int port) + Socket() + connect(SocketAddress host)`
Transport Protocols

- **Socket**: endpoint to read and write data
- Each Socket has a network protocol
- Two types of protocols used for communicating data/packets over the internet:
  - **TCP**:
    - *Transmission Control Protocol*
    - Connection Oriented (*handshake*)
  - **UDP**:
    - *User Datagram Protocol*
    - “Connectionless”
Transport Protocols

TCP

UDP
Outline

• Communication via Sockets in Java

• Multi-threading in Java

• Coding a full Client-Server Example
  
  On Eclipse, we’ll code an “echo” TCP Server-Client Example
TCP Single-Threading

TCP Multi-Threading
Multi-Threading in General

• STEP 1: A class intended to execute as a thread must implement the `Runnable` interface

  ```java
  public class Service implements Runnable
  ```

  • Implement the method `run()`
  ```java
  public void run() {
    // thread’s logic goes here
  }
  ```

• STEP 2: Instantiate a Thread object passing an instance of the intended class
  ```java
  Thread t = new Thread(new Service())
  ```

• STEP 3: Invoke `start()` on the new thread
  ```java
  t.start() // invokes the run() method implemented in the Service class
  ```
TCP Multi-.Threading Example

public class TCPServer {
    public void runService(Socket serviceSocket) {
        Thread newServiceThread = new Thread(new Service(serviceSocket));
        newServiceThread.start();
    }
}

Public class Service implements Runnable {
    private Service(Socket serviceSocket) {
    }
    public void run() {
        //the code for reading and writing to a client goes here
    }
}
Outline

• Communication via Sockets in Java

• Multi-threading in Java

• Coding a full Client-Server Example

  *On Eclipse, we’ll code an “echo” TCP Server-Client Example*
Let’s start with Psuedocode

Server

serverAddres = new InetSocketAddress(port)
listenSocket= new ServerSocket()
listenSocket.bind(serverAddres)
While(true)
    serviceSocket= listenSocket.accept()
    Thread service= new thread(new Service(serviceSocket))
    service.start()

Service implements Runnable

While(true)
    Read client message from socket
    Write message back to client
    serviceSocket.close()

Client

• serverAddres= new InetSocketAddress(port)
• clientSocket= new Socket()
• clientSocket.connect(serverAddres)
• While(true)
  • Read user’s input message
  • Write the message to the socket
  • Read the echoed message from the socket
  • clientSocket.close()

How to do these?

While(true)
    Read client message from socket
    Write message back to client
    serviceSocket.close()
Useful Java Methods/Classes: To Read User’s input

**Scanner class** allows to read user input.

```java
Scanner( InputStream source )
Constructs a new Scanner that produces values scanned from the specified input stream.
```

Methods to read different input types using the scanner object

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>nextBoolean()</td>
<td>Reads a <strong>boolean</strong> value from the user</td>
</tr>
<tr>
<td>nextByte()</td>
<td>Reads a <strong>byte</strong> value from the user</td>
</tr>
<tr>
<td>nextDouble()</td>
<td>Reads a <strong>double</strong> value from the user</td>
</tr>
<tr>
<td>nextFloat()</td>
<td>Reads a <strong>float</strong> value from the user</td>
</tr>
<tr>
<td>nextInt()</td>
<td>Reads a <strong>int</strong> value from the user</td>
</tr>
<tr>
<td>nextLine()</td>
<td>Reads a <strong>String</strong> value from the user</td>
</tr>
<tr>
<td>nextLong()</td>
<td>Reads a <strong>long</strong> value from the user</td>
</tr>
<tr>
<td>nextShort()</td>
<td>Reads a <strong>short</strong> value from the user</td>
</tr>
</tbody>
</table>

If you pass (System.in), you can read input from the keyboard.
Useful Java Methods/Classes: To Read and Write to Socket

When you create a socket, you can retrieve the socket’s InputStream and OutputStream which allow you to write raw bytes to the socket

```java
public InputStream getInputStream()
public OutputStream getOutputStream()
```

Java has more classes that build on InputStream and OutputStream to allow writing data in different forms and ways

We will create `ObjectInputStream` and `ObjectOutputStream` objects to be able to read and write objects instead of raw bytes.

We will use the following constructors:

```java
ObjectInputStream(InputStream in)
ObjectOutputStream(OutputStream out)
```

Then we can use the `readObject()`, `writeObject()` methods to read from and write to the socket
Demo Time 😊