Orientación a Objetos 2

EchoServer (versión Java)

This program provides a basic Echo Server that can handle only one client connection.

You can connect to the Echo Server from clients (written in languages that can create sockets) or even using applications like Netcast on Linux and Putty on Windows.

Explanation:

1. **`SingleThreadEchoServer` Class:** is the main class containing the server behavior and the response logic.
2. **`main` Method:**
    - It gets the *`portNumber`* on which the server will listen for incoming connections as an argument.
    - It creates a `ServerSocket` that listens on the specified port. The `try-with-resources` statement ensures the `ServerSocket` is automatically closed when the block finishes.
    - Prints a message indicating that the server has started.
    - Enters an infinite `while` loop to continuously accept client connections.
    - **`serverSocket.accept()`:** This is a blocking call. The server will pause here until a client attempts to connect. When a connection is established, it returns a `Socket` object representing the connection to that specific client.
    - Prints a message indicating a client has connected.
    - Calls the `handleClient()` method to manage the communication with the connected client.
    - After `handleClient()` finishes (when the client disconnects), the loop continues to wait for the next client.
3. **`handleClient` Method:**
    - Takes a `Socket` object as input, representing the connection to a specific client.

- Uses `try-with-resources` to create `PrintWriter` for sending data to the client and `BufferedReader` for receiving data from the client. These resources are automatically closed when the block finishes.
- Enters a `while` loop that reads lines of text from the client using `in.readLine()`. This call is also blocking, waiting for the client to send data.
- If `in.readLine()` returns a non-null value (meaning the client sent data):
  - Prints the received message to the server's console.
  - Uses `out.println()` to send an "Echo:" message back to the client, followed by the data the client sent.
- The `finally` block ensures that the `clientSocket` is closed, regardless of whether an exception occurred during communication.

How it Works (Single-Threaded):

The key characteristic of this server is that it's **single-threaded**. This means:

1. When the server starts, it creates a single thread (the main thread) that listens for incoming connections.
2. When a client connects, the `serverSocket.accept()` method returns, and the `handleClient()` method is executed **in the same main thread**.
3. The `handleClient()` method then handles all communication with that specific client (reading input and sending the echo back) **sequentially**.
4. **Crucially, while the server is handling one client's requests in the `handleClient()` method, it cannot accept or process connections from any other clients.** Other clients attempting to connect will have to wait until the current client's connection is closed and the `handleClient()` method finishes. Only then will the `main` thread go back to the `serverSocket.accept()` call to listen for new connections.

To Run This Code:

1. Compile `SingleThreadEchoServer.java from a terminal`:
   javac SingleThreadEchoServer.java
2. Run the compiled class:
   java SingleThreadEchoServer <porNumber>
3.

To Test with a Client (e.g., `netcat`):

1. Open a terminal and run the server.

   nc localhost 12345

2. Type some text in the `netcat` terminal and press Enter. You should see the server echo it back.
3. To disconnect the client enter an empty message