I. `ClientServer` Class Instructions

1. **Class Definition:**
    - Create a public class named `ClientServer`.
2. **Imports:**
    - Import necessary classes:
        - `BufferedReader`
        - `IOException`
        - `InputStreamReader`
        - `PrintWriter`
        - `ServerSocket`
        - `Socket`
        - `logging.*`
3. **Logger:**
    - Declare a static final `Logger` named `logger` for logging messages. Get the logger instance using `Logger.getLogger(ClientServer.class.getName())`.
4. **Port Constant:**
    - Declare a private static final integer constant named `PORT` and set it to `12345`. This will be the port number used for communication.
5. **`main` Method:**
    - Create the `main` method. This is the entry point of the application.
    - Inside `main`, create and start two threads:
        - Create a new `Thread` object, passing an instance of the `Server` class (defined below) to its constructor. Call `start()` on this thread to begin server execution.
        - Create another new `Thread` object, passing an instance of the `Client` class (defined below) to its constructor. Call `start()` on this thread to begin client execution.

II. `Server` Class Instructions (Nested within `ClientServer`)

1. **Class Definition:**
    - Create a static nested class named `Server` that `implements Runnable`. This allows the server to run in its own thread.
2. **`run` Method:**
    - Override the `run()` method from the `Runnable` interface (`@Override`). This method will contain the server's main logic.
    - **`ServerSocket` Creation (Try-with-Resources):**
        - Use a *try-with-resources* block to create a `ServerSocket` object, binding it to the `PORT` constant. This automatically closes the `ServerSocket` when the block finishes.
        - Inside the `try` block, log an informational message using the `logger` indicating that the server has started and the port it's listening on. Use `Level.INFO`.
        - Catch `IOException` that might occur during `ServerSocket` creation and print the stack trace.
    - **Client Connection Loop:**
        - Create an infinite `while (true)` loop to continuously accept client connections.
        - **Accept Connection:** Inside the loop, call `serverSocket.accept()`. This method *blocks* (waits) until a client connects. When a client connects, it returns a `Socket` object representing the connection to that client. Store this `Socket` in a variable (e.g., `clientSocket`).
        - **Print Connection Message:** Print a message to the console indicating that a client has connected, including the `clientSocket` details (for debugging).
        - **Create `ClientHandler` Thread:** Create a new `Thread`, passing a new instance of the `ClientHandler` class (defined below) to its constructor. Pass the `clientSocket` to the `ClientHandler` constructor. Start the `ClientHandler` thread using `start()`. This allows the server to handle multiple clients concurrently.

III. `ClientHandler` Class Instructions (Nested within `Server`)

1. **Class Definition:**
    - Create a static nested class named `ClientHandler` that `implements Runnable`. This handles communication with a *single* client.
2. **`clientSocket` Field:**
    - Declare a private `Socket` field named `clientSocket` to store the socket associated with the client this handler is responsible for.
3. **Constructor:**
    - Create a constructor that takes a `Socket` object (the client's socket) as a parameter and assigns it to the `clientSocket` field.
4. **`run` Method:**
    - Override the `run()` method.
    - **Input/Output Streams (Try-with-Resources):**
        - Use a *try-with-resources* block to create:
            - A `BufferedReader` named `in` to read data *from* the client. Wrap `clientSocket.getInputStream()` with an `InputStreamReader`.
            - A `PrintWriter` named `out` to write data *to* the client. Wrap `clientSocket.getOutputStream()` and set auto-flush to `true`.
        - Catch `IOException` and print an error message to the console (not the stack trace, just the message).
    - **Communication Loop:**
        - Create a `while` loop that continues as long as `in.readLine()` returns a non-null value. Store the result of `in.readLine()` in a `String` variable (e.g., `inputLine`). `readLine()` blocks until a line of text is received from the client or the connection is closed.
        - **Print Received Message:** Inside the loop, print the received message to the console (for debugging/monitoring).
        - **"exit" Command Check:**
            - Check if `inputLine` is equal to "exit" (case-insensitive).
            - If it is "exit":
                - Send "Bye" to the client using `out.println()`.
                - `break` out of the `while` loop.

        - If it's *not* "exit":
            - Echo the received message back to the client by sending "Echo: " followed by the `inputLine` using `out.println()`.
    - **Close Client Socket:**
        - After the loop finishes (either because the client sent "exit" or the connection was closed), create a new try catch block.
        - Inside the try, close `clientSocket` *outside* of the try-with-resources block. This is important because the try-with-resources block for the streams will have already closed the streams, and closing the socket is a separate operation. Catch `IOException` and print the stack trace.

IV. `Client` Class Instructions (Nested within `ClientServer`)

1. **Class Definition:**
    - Create a static nested class named `Client` that `implements Runnable`.
2. **`run` Method:**
    - Override the `run()` method.
    - **Socket and Streams (Try-with-Resources):**
        - Use a *try-with-resources* block to create:
            - A `Socket` named `socket` to connect to the server. Use `"localhost"` as the hostname and `PORT` as the port number.
            - A `PrintWriter` named `out` to send data to the server. Wrap `socket.getOutputStream()`. Set auto-flush to `true`.
            - A `BufferedReader` named `in` to receive data from the server. Wrap `socket.getInputStream()` with an `InputStreamReader`.
            - A `BufferedReader` named `stdIn` to read input from the *console* (System.in). Wrap `System.in` with an `InputStreamReader`.
        - Catch `IOException` and print error message.
    - **Print Connection Message:** Print a message to the console indicating that the client has connected to the server.
    - **Input Loop:**

- Create a `while` loop that continues as long as `stdIn.readLine()` returns a non-null value. Store the result of `stdIn.readLine()` in a `String` variable (e.g., `userInput`).
- **Send to Server:** Send the `userInput` to the server using `out.println()`.
- **Receive from Server:** Read the server's response using `in.readLine()` and store it in a string variable e.g. receivedMessage.
    - **Null check:** if receivedMessage is null. Print "Server closed the connection." and break the loop
- **Print Server Response:** Print the server's response to the console.
- **"exit" Check:** Check if `userInput` is equal to "exit" (case-insensitive). If it is, `break` out of the loop.