I. `SimpleHttpServer` Class Instructions

1. **Class Definition:**
    - Create a public class named `SimpleHttpServer`.
2. **Imports:**
    - Import the necessary classes:
        - `HttpExchange`
        - `HttpHandler`
        - `HttpServer`
        - `IOException`
        - `InputStream` (although not directly used in the provided code, it's good practice to include it since related classes are used)
        - `OutputStream`
        - `InetSocketAddress`
        - `java.nio.charset.StandardCharsets`
3. **`main` Method:**
    - Create the `public static void main(String[] args)` method. This is the entry point.
    - **`HttpServer` Creation (Try-Catch):**
        - Use a try-catch block to handle any potential `IOException` during server setup.
        - Inside the `try` block:
            - Create an instance of `HttpServer` using and name it as `server`
            - Pass a new `InetSocketAddress` object to `create()`, specifying port 8080.
            - Set the second argument of `create()` to `0`. This represents the *backlog*, or the maximum number of queued incoming connections. `0` means to use the system default.
        - Inside the `catch` block
            - print the message including "Exception happened" and the exception object.
    - **Context Creation:**
        - Call `createContext()`. This associates the path `/myendpoint` with an instance of the `MyHandler` class (defined below). This means that any requests to `/myendpoint` will be handled by the `MyHandler`.

- **Executor:**
  - Call `setExecutor()`. This configures the server to use the default executor, which handles each request in the calling thread. (For more complex scenarios, you might use a thread pool here).
- **Start Server:**
  - Call `start()` to start the server.
  - Print a message to the console indicating that the server has started and the port it's listening on.

II. `MyHandler` Class Instructions (Nested within `SimpleHttpServer`)

1. **Class Definition:**
   - Create a static nested class named `MyHandler` that `implements HttpHandler`.
2. **`handle` Method:**
   - Override the `handle()` method (from the `HttpHandler` interface). This method processes incoming HTTP requests. The method signature should include `throws IOException`.
   - **Get Request Method and Path:**
     - Get the request method (e.g., "GET", "POST") using `getRequestMethod()` and store it in a `String` variable.
     - Get the request URI's path using `getPath()` and store it in a `String` variable.
   - **Path Check**
     - Create an if statement to check the path is not equal to /myendpoint, if true call sendResponse method with relative arguments and "Not Found" message with 404 status code, then return.
   - **Method Dispatch (if-else if-else):**
     - Use an `if-else if-else` structure to handle different HTTP methods:
       - **`if` (GET):** If the request method is "GET" (case-insensitive comparison using `equalsIgnoreCase`), call a separate method named `handleGetRequest(exchange)`.
       - **`else if` (POST):** If the request method is "POST" (case-insensitive), create a String variable that holds a "This is a POST request to /myendpoint. Request body is ignored."

message and then call a separate method named `sendResponse()`. (This method is defined later).

- **`else` (Unsupported Method):** For any other request method, call `sendResponse(exchange, "Method Not Allowed", 405)`.

3. **`handleGetRequest` Method:**
    - Create a void method called `handleGetRequest()` and pass an object of the `HttpExchange` named `exchange` as its argument. This method should `throws IOException`.
    - **Create Response String:** Create a `String` variable holding the response message: "This is a GET request to /myendpoint".
    - **Send Response:** Call the `sendResponse` method (defined below), passing the `exchange` object, the response string, and the status code 200.
4. **`sendResponse` Method:**
    - Create a private void method called sendResponse and pass three arguments to it: 1) an object of the `HttpExchange` named `exchange` 2) a `String` variable named `response` and 3) an `integer` variable named `statusCode`. This method handles the actual sending of the response to the client, and `throws IOException`.
    - **Set Content-Type Header:**
        - Get the response headers using `getResponseHeaders()`.
        - Set the `Content-Type` header to `text/plain`.
    - **Send Response Headers:**
        - Call `sendResponseHeaders()`. This sends the HTTP status code and the length of the response body (in bytes). It is important to call this method *before* writing to the response body. Use `StandardCharsets.UTF_8` to ensure correct character encoding.
    - **Write Response Body (Try-with-Resources):**
        - Use a *try-with-resources* block to get the response body output stream:
            - Inside the `try`, obtain the `OutputStream` using `exchange.getResponseBody()` and store it in a variable (e.g., `os`).

        - Write the `response` string to the `OutputStream` *as bytes*. Use `os.write(response.getBytes(StandardCharsets.UTF_8))`.
    - The `try-with-resources` block automatically closes the `OutputStream` when it's done.