

## 9. 채팅 프로그램

#1.인강/0.자바/6.자바-고급2편

- /채팅 프로그램 - 설계
- /채팅 프로그램 - 클라이언트
- /채팅 프로그램 - 서버1
- /채팅 프로그램 - 서버2
- /채팅 프로그램 - 서버3
- /채팅 프로그램 - 서버4
- /정리

### 채팅 프로그램 - 설계

지금까지 학습한 네트워크를 활용해서 간단한 채팅 프로그램을 만들어보자.

요구사항은 다음과 같다.

- 서버에 접속한 사용자는 모두 대화할 수 있어야 한다.
- 다음과 같은 채팅 명령어가 있어야 한다.
  - 입장 /join|{name}
    - ◆ 처음 채팅 서버에 접속할 때 사용자의 이름을 입력해야 한다.
  - 메시지 /message|{내용}
    - ◆ 모든 사용자에게 메시지를 전달한다.
  - 이름 변경 /change|{name}
    - ◆ 사용자의 이름을 변경한다.
  - 전체 사용자 /users
    - ◆ 채팅 서버에 접속한 전체 사용자 목록을 출력한다.
  - 종료 /exit
    - ◆ 채팅 서버의 접속을 종료한다.

### 채팅 프로그램 설계 - 클라이언트

기존에 작성한 네트워크 프로그램과 기본 뼈대는 비슷하지만, 어느정도 기본 설계가 필요하다.

#### 클라이언트 설계

채팅은 실시간으로 대화를 주고받아야 한다. 그런데 기존에 작성한 네트워크 클라이언트 프로그램은 사용자의 콘솔 입력이 있을 때까지 무한정 대기하는 문제가 있다.

### 기존 클라이언트 프로그램의 문제

```
System.out.print("전송 문자: ");
String toSend = scanner.nextLine(); // 블로킹

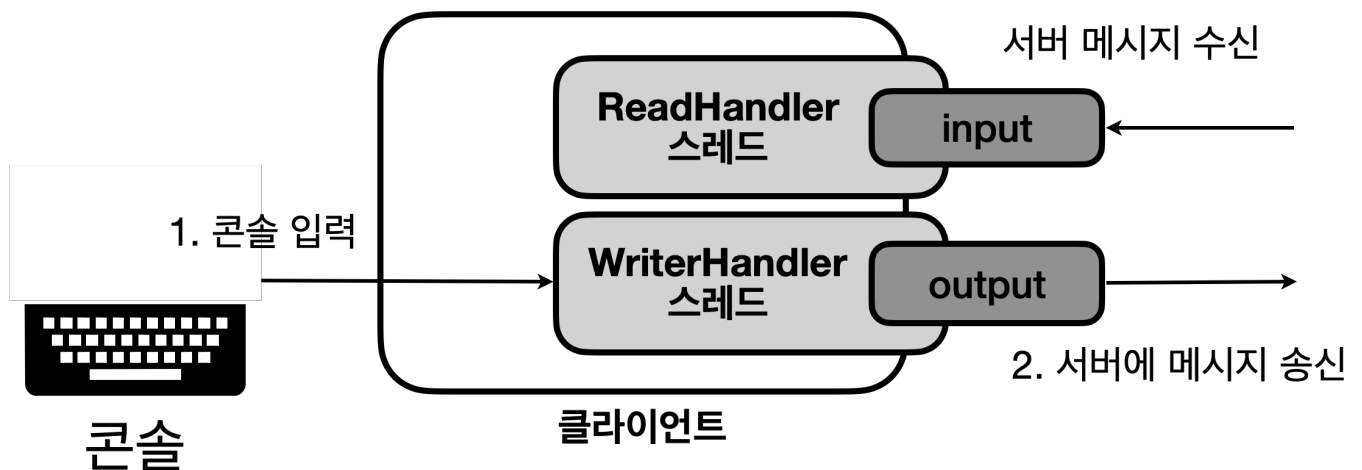
// 서버에게 문자 보내기
output.writeUTF(toSend);
log("client -> server: " + toSend);
```

- 스레드는 사용자의 콘솔 입력을 대기하기 때문에, 실시간으로 다른 사용자가 보낸 메시지를 콘솔에 출력할 수 없다.

콘솔의 입력을 기다리는 부분도 블로킹 되지만, 서버로부터 메시지를 받는 다음 코드도 블로킹 된다.

```
// 서버로부터 문자 받기
String received = input.readUTF(); // 블로킹
```

따라서 사용자의 콘솔 입력과 서버로부터 메시지를 받는 부분을 별도의 스레드로 분리해야 한다.



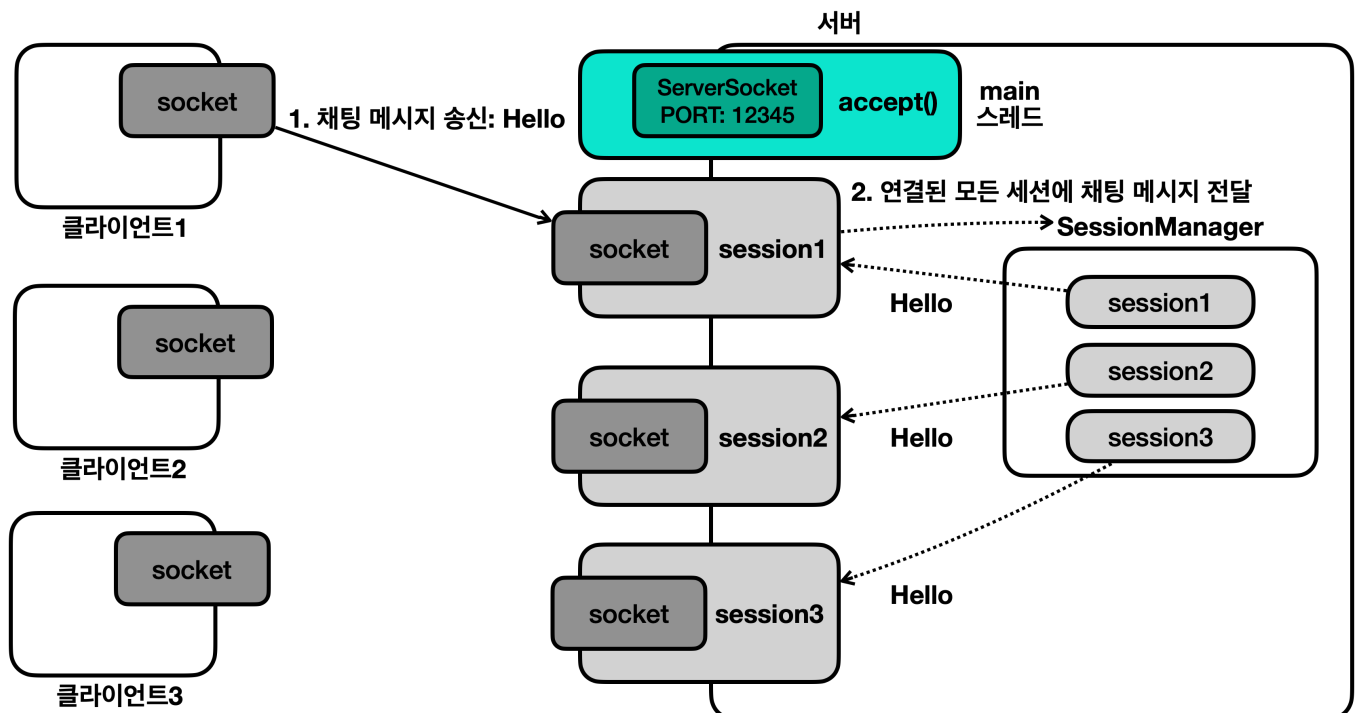
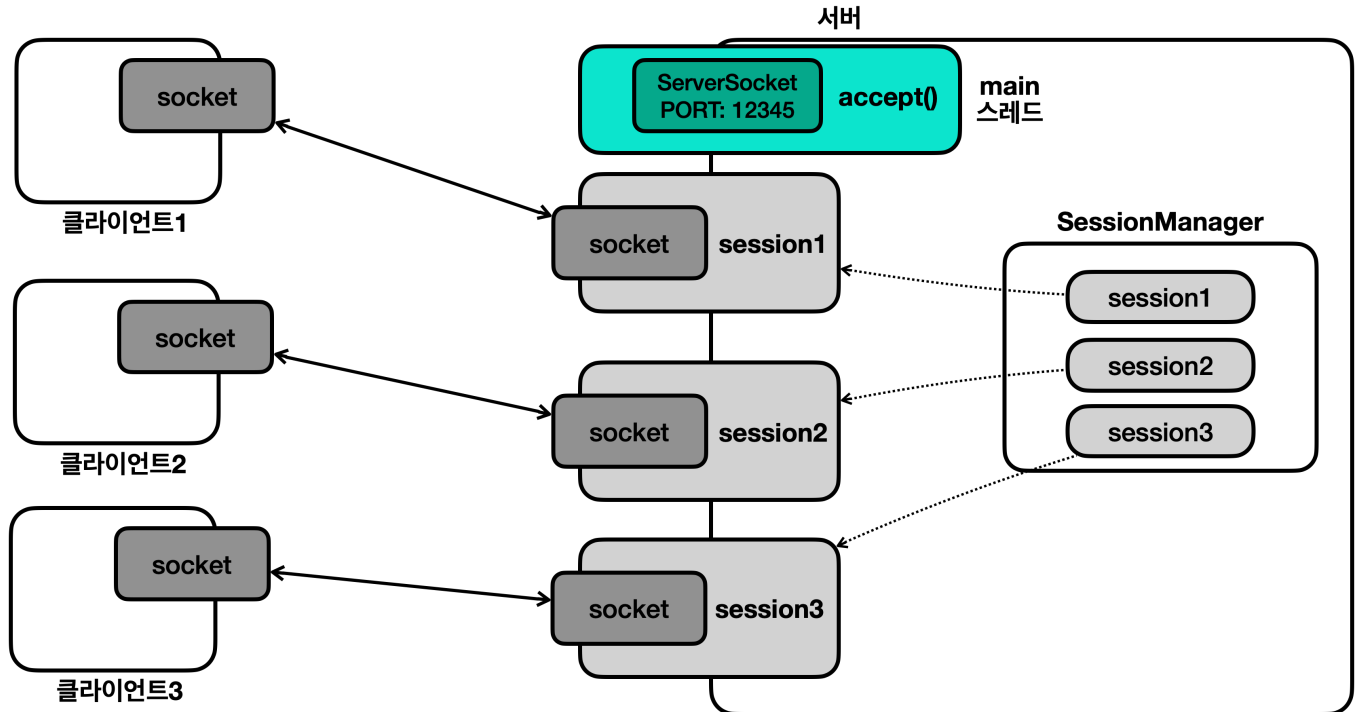
### 채팅 프로그램 설계 - 서버

채팅 프로그램의 핵심은 한 명이 이야기하면 그 이야기를 모두가 들을 수 있어야 한다.

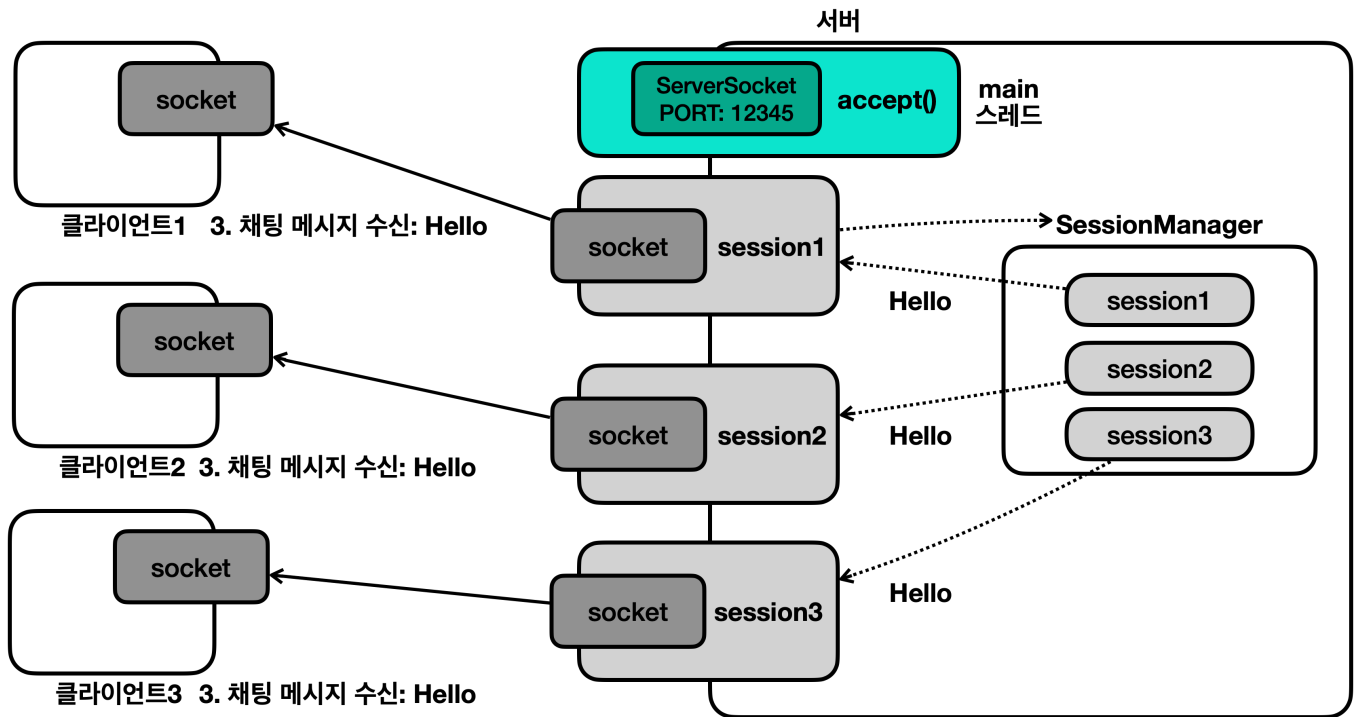
따라서 하나의 클라이언트가 보낸 메시지를 서버가 받은 다음에, 서버에서 모든 클라이언트에게 메시지를 다시 전송해야 한다.

이렇게 하려면 서버에서 모든 세션을 관리해야 한다. 그렇게 해야 모든 세션에 메시지를 전달할 수 있다.  
참고로 우리는 앞서 세션을 관리하는 세션 매니저를 만들어두었다!  
따라서 기존 구조를 잘 활용하면 채팅 서버를 쉽게 구축할 수 있다.

### 채팅 프로그램 서버 구조

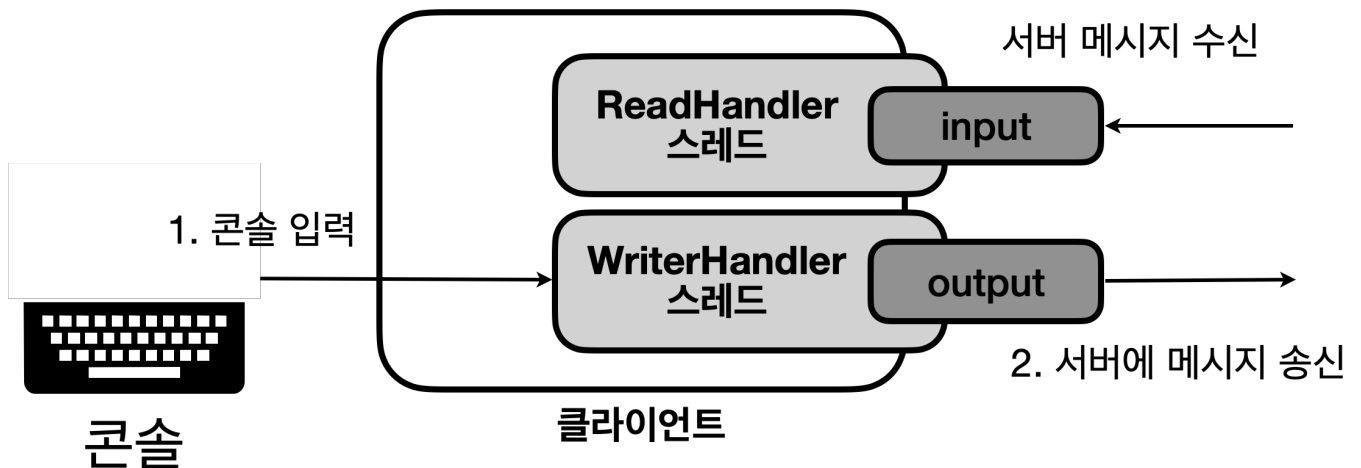


- 클라이언트1이 서버로 "Hello"라는 메시지를 전송한다.
- 서버는 SessionManager를 통해 연결된 모든 세션에 "Hello" 채팅 메시지를 전달한다.



- 각각의 세션은 자신의 클라이언트에게 "Hello" 메시지를 전송한다.
- 모든 클라이언트는 서버로부터 "Hello" 메시지를 전송 받는다.

## 채팅 프로그램 - 클라이언트



클라이언트는 다음 두 기능을 별도의 스레드에서 실행해야 한다.

- 콘솔의 입력을 받아서 서버로 전송한다.
- 서버로부터 오는 메시지를 콘솔에 출력한다.

먼저 서버로부터 오는 메시지를 콘솔에 출력하는 기능을 개발해보자.

```

package chat.client;

import java.io.DataInputStream;
import java.io.IOException;

import static util.MyLogger.log;

public class ReadHandler implements Runnable {

    private final DataInputStream input;
    private final Client client;
    public boolean closed = false;

    public ReadHandler(DataInputStream input, Client client) {
        this.input = input;
        this.client = client;
    }

    @Override
    public void run() {
        try {
            while (true) {
                String received = input.readUTF();
                System.out.println(received);
            }
        } catch (IOException e) {
            log(e);
        } finally {
            client.close();
        }
    }

    public synchronized void close() {
        if (closed) {
            return;
        }
        // 종료 로직 필요시 작성
        closed = true;
        log("readHandler 종료");
    }
}

```

- ReadHandler는 Runnable 인터페이스를 구현하고, 별도의 스레드에서 실행한다.

- 서버의 메시지를 반복해서 받고, 콘솔에 출력하는 단순한 기능을 제공한다.
- 클라이언트 종료시 `ReadHandler` 도 종료된다. 중복 종료를 막기 위해 동기화 코드와 `closed` 플래그를 사용했다.
  - 참고로 예제 코드는 단순하므로 중요한 종료 로직이 없다.
- `IOException` 예외가 발생하면 `client.close()` 를 통해 클라이언트를 종료하고, 전체 자원을 정리한다.

```
package chat.client;

import java.io.DataOutputStream;
import java.io.IOException;
import java.util.NoSuchElementException;
import java.util.Scanner;

import static util.MyLogger.log;

public class WriteHandler implements Runnable {

    private static final String DELIMITER = "|";

    private final DataOutputStream output;
    private final Client client;

    private boolean closed = false;

    public WriteHandler(DataOutputStream output, Client client) {
        this.output = output;
        this.client = client;
    }

    @Override
    public void run() {
        Scanner scanner = new Scanner(System.in);
        try {
            String username = inputUsername(scanner);
            output.writeUTF("/join" + DELIMITER + username);

            while (true) {
                String toSend = scanner.nextLine();
                if (toSend.isEmpty()) {
                    continue;
                }
            }
        } catch (IOException e) {
            client.close();
        }
    }
}
```

```

    }

    if (toSend.equals("/exit")) {
        output.writeUTF(toSend);
        break;
    }

    // "/"로 시작하면 명령어, 나머지는 일반 메시지
    if (toSend.startsWith("/")) {
        output.writeUTF(toSend);
    } else {
        output.writeUTF("/message" + DELIMITER + toSend);
    }
}

} catch (IOException | NoSuchElementException e) {
    log(e);
} finally {
    client.close();
}
}

private static String inputUsername(Scanner scanner) {
    System.out.println("이름을 입력하세요.");
    String username;
    do {
        username = scanner.nextLine();
    } while (username.isEmpty());
    return username;
}

public synchronized void close() {
    if (closed) {
        return;
    }
    try {
        System.in.close(); // Scanner 입력 중지 (사용자의 입력을 닫음)
    } catch (IOException e) {
        log(e);
    }
    closed = true;
    log("writeHandler 종료");
}
}
}

```

- `WriteHandler` 는 사용자 콘솔의 입력을 받아서 서버로 메시지를 전송한다.
- 처음 시작시 `inputUsername()` 을 통해 사용자의 이름을 입력 받는다.
- 처음 채팅 서버에 접속하면 `/join|{name}` 을 전송한다. 이 메시지를 통해 입장했다는 정보와 사용자의 이름을 서버에 전달한다.
- 메시지는 다음과 같이 설계된다.
  - 입장 `/join|{name}`
  - 메시지 `/message|{내용}`
  - 종료 `/exit`
- 만약 콘솔 입력시 `/` 로 시작하면 `/join`, `/exit` 같은 특정 명령어를 수행한다고 가정한다.
- `/` 를 입력하지 않으면 일반 메시지로 보고 `/message` 에 내용을 추가해서 서버에 전달한다.

`close()` 를 호출하면 `System.in.close()` 를 통해 사용자의 콘솔 입력을 닫는다. 이렇게 하면 `Scanner` 를 통한 콘솔 입력인 `scanner.nextLine()` 코드에서 대기하는 스레드에 다음 예외가 발생하면서, 대기 상태에서 빠져나올 수 있다.

```
java.util.NoSuchElementException: No line found
```

서버가 연결을 끊은 경우에 클라이언트의 자원이 정리되는데, 이때 유용하게 사용된다.

`IOException` 예외가 발생하면 `client.close()` 를 통해 클라이언트를 종료하고, 전체 자원을 정리한다.

## 윈도우 OS 추가

윈도우의 경우 `System.in.close()` 를 호출해도 사용자의 콘솔 입력이 닫히지 않는다. 사용자가 어떤 내용이든 입력을 해야 그 다음에 `java.util.NoSuchElementException: No line found` 예외가 발생하면서 대기 상태에서 빠져나올 수 있다.

```
package chat.client;

import java.io.DataInputStream;
import java.io.DataOutputStream;
import java.io.IOException;
import java.net.Socket;

import static network.tcp.SocketCloseUtil.closeAll;
import static util.MyLogger.log;
```



```
public class Client {

    private final String host;
    private final int port;

    private Socket socket;
    private DataInputStream input;
    private DataOutputStream output;

    private ReadHandler readHandler;
    private WriteHandler writeHandler;
    private boolean closed = false;

    public Client(String host, int port) {
        this.host = host;
        this.port = port;
    }

    public void start() throws IOException {
        log("클라이언트 시작");
        socket = new Socket(host, port);
        input = new DataInputStream(socket.getInputStream());
        output = new DataOutputStream(socket.getOutputStream());

        readHandler = new ReadHandler(input, this);
        writeHandler = new WriteHandler(output, this);
        Thread readThread = new Thread(readHandler, "readHandler");
        Thread writeThread = new Thread(writeHandler, "writeHandler");
        readThread.start();
        writeThread.start();
    }

    public synchronized void close() {
        if (closed) {
            return;
        }
        writeHandler.close();
        readHandler.close();
        closeAll(socket, input, output);
        closed = true;
        log("연결 종료: " + socket);
    }
}
```

```
}
```

- 클라이언트 전반을 관리하는 클래스이다.
- `Socket`, `ReadHandler`, `WriteHandler` 를 모두 생성하고 관리한다.
- `close()` 메서드를 통해 전체 자원을 정리하는 기능도 제공한다.

```
package chat.client;

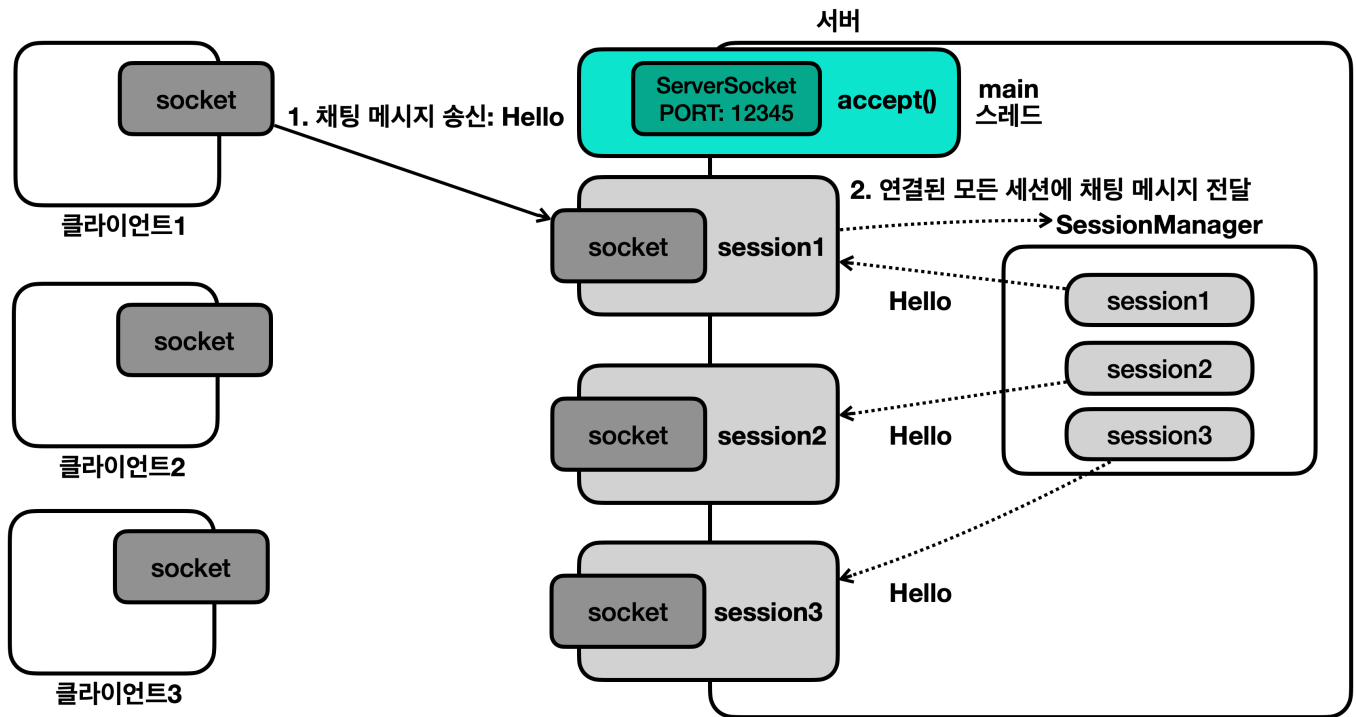
import java.io.IOException;

public class ClientMain {

    private static final int PORT = 12345;

    public static void main(String[] args) throws IOException {
        Client client = new Client("localhost", PORT);
        client.start();
    }
}
```

## 채팅 프로그램 - 서버1



채팅 프로그램 서버의 경우 기존에 작성한 네트워크 프로그램의 서버에서 필요한 기능을 추가하면 된다.

```
package chat.server;

import java.io.DataInputStream;
import java.io.DataOutputStream;
import java.io.IOException;
import java.net.Socket;

import static network.tcp.SocketCloseUtil.closeAll;
import static util.MyLogger.log;

public class Session implements Runnable {

    private final Socket socket;
    private final DataInputStream input;
    private final DataOutputStream output;
    private final CommandManager commandManager;
    private final SessionManager sessionManager;

    private boolean closed = false;
    private String username;

    public Session(Socket socket, CommandManager commandManager,
        SessionManager sessionManager) throws IOException {
```

```

        this.socket = socket;
        this.input = new DataInputStream(socket.getInputStream());
        this.output = new DataOutputStream(socket.getOutputStream());

        this.commandManager = commandManager;
        this.sessionManager = sessionManager;
        this.sessionManager.add(this);
    }

    @Override
    public void run() {
        try {
            while (true) {
                // 클라이언트로부터 문자 받기
                String received = input.readUTF();
                log("client -> server: " + received);
                commandManager.execute(received, this);
            }
        } catch (IOException e) {
            log(e);
        } finally {
            sessionManager.remove(this);
            sessionManager.sendAll(username + "님이 퇴장했습니다.");
            close();
        }
    }

    public void send(String message) throws IOException {
        log("server -> client: " + message);
        output.writeUTF(message);
    }

    public synchronized void close() {
        if (closed) {
            return;
        }
        closeAll(socket, input, output);
        closed = true;
        log("연결 종료: " + socket);
    }

    public String getUsername() {
        return username;
    }

```

```

    }

    public void setUsername(String username) {
        this.username = username;
    }
}

```

- `CommandManager` 는 명령어를 처리하는 기능을 제공한다. 바로 뒤에서 설명한다.
- `Session` 의 생성 시점에 `sessionManager` 에 `Session` 을 등록한다.
- `username` 을 통해 클라이언트의 이름을 등록할 수 있다. 사용자의 이름을 사용하는 기능은 뒤에 추가하겠다. 지금은 값이 없으니 `null` 로 사용된다.

### run()

- 클라이언트로부터 메시지를 전송받는다.
- 전송 받은 메시지를 `commandManager.execute()` 를 사용해서 실행한다.
- 예외가 발생하면 세션 매니저에서 세션을 제거하고, 나머지 클라이언트에게 퇴장 소식을 알린다. 그리고 자원을 정리한다.

### send(String message)

- 이 메서드를 호출하면 해당 세션의 클라이언트에게 메시지를 보낸다.

```

package chat.server;

import java.io.IOException;
import java.util.ArrayList;
import java.util.List;

import static util.MyLogger.log;

public class SessionManager {

    private final List<Session> sessions = new ArrayList<>();

    public synchronized void add(Session session) {
        sessions.add(session);
    }

    public synchronized void remove(Session session) {
        sessions.remove(session);
    }
}

```

```

    }

    public synchronized void closeAll() {
        for (Session session : sessions) {
            session.close();
        }
        sessions.clear();
    }

    public synchronized void sendAll(String message) {
        for (Session session : sessions) {
            try {
                session.send(message);
            } catch (IOException e) {
                log(e);
            }
        }
    }

    public synchronized List<String> getAllUsername() {
        List<String> usernames = new ArrayList<>();
        for (Session session : sessions) {
            if (session.getUsername() != null) {
                usernames.add(session.getUsername());
            }
        }
        return usernames;
    }
}

```

- 세션을 관리한다.
- `closeAll()` : 모든 세션을 종료하고, 세션 관리자에서 제거한다.
- `sendAll()` : 모든 세션에 메시지를 전달한다. 이때 각 세션의 `send()` 메서드가 호출된다.
- `getAllUsername()` : 모든 세션에 등록된 사용자의 이름을 반환한다. 향후 모든 사용자 목록을 출력할 때 사용된다.

```

package chat.server;

import java.io.IOException;

```

```
public interface CommandManager {
    void execute(String totalMessage, Session session) throws IOException;
}
```

- 클라이언트에게 전달받은 메시지를 처리하는 인터페이스이다.
  - 인터페이스를 사용한 이유는 향후 구현체를 점진적으로 변경하기 위해서이다.
- totalMessage: 클라이언트에게 전달 받은 메시지
- Session session: 현재 세션

```
package chat.server;

import java.io.IOException;

public class CommandManagerV1 implements CommandManager {

    private final SessionManager sessionManager;

    public CommandManagerV1(SessionManager sessionManager) {
        this.sessionManager = sessionManager;
    }

    @Override
    public void execute(String totalMessage, Session session) throws
    IOException {
        if (totalMessage.startsWith("/exit")) {
            throw new IOException("exit");
        }

        sessionManager.sendAll(totalMessage);
    }
}
```

- 클라이언트에게 일반적인 메시지를 전달 받으면, 모든 클라이언트에게 같은 메시지를 전달해야 한다.
- sessionManager.sendAll(totalMessage) 를 사용해서 해당 기능을 처리한다.
- /exit 가 호출되면 IOException 을 던진다. 세션은 해당 예외를 잡아서 세션을 종료한다.
- CommandManagerV1 은 최소한의 메시지 전달 기능만 구현했다. 복잡한 나머지 기능들은 다음 버전에 추가할 예정이다.

```
package chat.server;

import java.io.IOException;
import java.net.ServerSocket;
import java.net.Socket;

import static util.MyLogger.log;

public class Server {
    private final int port;
    private final CommandManager commandManager;
    private final SessionManager sessionManager;

    private ServerSocket serverSocket;

    public Server(int port, CommandManager commandManager, SessionManager
sessionManager) {
        this.port = port;
        this.commandManager = commandManager;
        this.sessionManager = sessionManager;
    }

    public void start() throws IOException {
        log("서버 시작: " + commandManager.getClass());
        serverSocket = new ServerSocket(port);
        log("서버 소켓 시작 - 리스닝 포트: " + port);

        addShutdownHook();
        running();
    }

    private void addShutdownHook() {
        ShutdownHook target = new ShutdownHook(serverSocket, sessionManager);
        Runtime.getRuntime().addShutdownHook(new Thread(target, "shutdown"));
    }

    private void running() {
        try {
            while (true) {
                Socket socket = serverSocket.accept(); // 블로킹
                log("소켓 연결: " + socket);
            }
        }
    }
}
```



```

        Session session = new Session(socket, commandManager,
sessionManager);
        Thread thread = new Thread(session);
        thread.start();
    }
} catch (IOException e) {
    log("서버 소켓 종료: " + e);
}
}

static class ShutdownHook implements Runnable {
    private final ServerSocket serverSocket;
    private final SessionManager sessionManager;

    public ShutdownHook(ServerSocket serverSocket, SessionManager
sessionManager) {
        this.serverSocket = serverSocket;
        this.sessionManager = sessionManager;
    }

    @Override
    public void run() {
        log("shutdownHook 실행");
        try {
            sessionManager.closeAll();
            serverSocket.close();

            Thread.sleep(1000); // 자원 정리 대기
        } catch (Exception e) {
            e.printStackTrace();
            System.out.println("e = " + e);
        }
    }
}
}
}

```

- 앞서 설명한 네트워크 프로그램과 거의 같다.
- `addShutdownHook()` 첫다운 훅을 등록한다.
- `running()`: 클라이언트의 연결을 처리하고 세션을 생성한다.

```
package chat.server;
```

```

import java.io.IOException;

public class ServerMain {

    private static final int PORT = 12345;

    public static void main(String[] args) throws IOException {
        SessionManager sessionManager = new SessionManager();

        // CommandManager 점진적으로 변경 예정
        CommandManager commandManager = new CommandManagerV1(sessionManager);

        Server server = new Server(PORT, commandManager, sessionManager);
        server.start();
    }
}

```

- Server는 생성자로 SessionManager와 CommandManager가 필요하다.
- 여기서 CommandManager의 구현체는 점진적으로 변경할 예정이다.

## 프로그램 실행

드디어 채팅 프로그램 CommandManagerV1 버전을 실행해보자.

### 실행 결과 - 서버

```

20:23:28.833 [      main] 서버 시작: class chat.server.CommandManagerV1
20:23:28.837 [      main] 서버 소켓 시작 - 리스닝 포트: 12345
20:23:35.258 [      main] 소켓 연결: Socket[addr=/
127.0.0.1,port=50917,localport=12345]
20:23:55.565 [ Thread-0] client -> server: /join|nate
20:23:55.566 [ Thread-0] server -> client: /join|nate
20:24:09.026 [      main] 소켓 연결: Socket[addr=/
127.0.0.1,port=50990,localport=12345]
20:24:11.706 [ Thread-1] client -> server: /join|seon
20:24:11.706 [ Thread-1] server -> client: /join|seon
20:24:11.707 [ Thread-1] server -> client: /join|seon
20:24:17.237 [ Thread-0] client -> server: /message|hi seon
20:24:17.237 [ Thread-0] server -> client: /message|hi seon
20:24:17.237 [ Thread-0] server -> client: /message|hi seon

```

```

20:24:41.189 [ Thread-1] client -> server: /message|hi nate
20:24:41.190 [ Thread-1] server -> client: /message|hi nate
20:24:41.190 [ Thread-1] server -> client: /message|hi nate
20:24:46.398 [ Thread-0] client -> server: /exit
20:24:46.398 [ Thread-0] java.io.IOException: exit
20:24:46.399 [ Thread-0] server -> client: null님이 퇴장했습니다.
20:24:46.401 [ Thread-0] 연결 종료: Socket[addr=/
127.0.0.1,port=50917,localport=12345]
20:24:50.279 [ Thread-1] client -> server: /exit
20:24:50.280 [ Thread-1] java.io.IOException: exit
20:24:50.280 [ Thread-1] 연결 종료: Socket[addr=/
127.0.0.1,port=50990,localport=12345]

```

- 첫 줄에 `chat.server.CommandManagerV1` 을 꼭! 확인하자.

### 실행 결과 - 클라이언트1

```

20:23:35.253 [      main] 클라이언트 시작
이름을 입력하세요.
nate
/join|nate
/join|seon
hi seon
/message|hi seon
/message|hi nate
/exit
20:24:46.398 [writeHandler] writeHandler 종료
20:24:46.398 [writeHandler] readHandler 종료
20:24:46.401 [writeHandler] 연결 종료: Socket[addr=localhost/
127.0.0.1,port=12345,localport=50917]
20:24:46.402 [readHandler] java.net.SocketException: Socket closed

```

### 실행 결과 - 클라이언트2

```

20:24:09.021 [      main] 클라이언트 시작
이름을 입력하세요.
seon
/join|seon
/message|hi seon
hi nate
/message|hi nate

```

```
null님이 퇴장했습니다.  
/exit  
20:24:50.279 [writeHandler] writeHandler 종료  
20:24:50.280 [writeHandler] readHandler 종료  
20:24:50.281 [readHandler] java.io.EOFException  
20:24:50.281 [writeHandler] 연결 종료: Socket[addr=localhost/  
127.0.0.1,port=12345,localport=50990]
```

아직은 기능이 부족하고, 이름도 없어서 null로 출력되지만 가장 중요한 핵심 기능인, 서로 대화가 잘 되는 것을 확인할 수 있다.

이제 본격적으로 기능을 붙여보자.

## 채팅 프로그램 - 서버2

이제 다음의 모든 기능을 제대로 완성해보자.

- 입장 /join|{name}
  - 처음 채팅 서버에 접속할 때 사용자의 이름을 입력해야 한다.
- 메시지 /message|{내용}
  - 모든 사용자에게 메시지를 전달한다.
- 이름 변경 /change|{name}
  - 사용자의 이름을 변경한다.
- 전체 사용자 /users
  - 채팅 서버에 접속한 전체 사용자 목록을 출력한다.
- 종료 /exit
  - 채팅 서버의 접속을 종료한다.

이 기능들은 모두 클라이언트의 메시지를 처리하는 기능들이다. 따라서 클라이언트의 메시지를 처리하는 `CommandManager` 인터페이스의 구현만 잘하면 된다.

모든 기능이 적용된 `CommandManagerV2` 를 만들어보자.

```
package chat.server;  
  
import java.io.IOException;  
import java.util.List;
```

```

public class CommandManagerV2 implements CommandManager {

    public static final String DELIMITER = "|";
    private final SessionManager sessionManager;

    public CommandManagerV2(SessionManager sessionManager) {
        this.sessionManager = sessionManager;
    }

    @Override
    public void execute(String totalMessage, Session session) throws
IOException {
        if (totalMessage.startsWith("/join")) {
            String[] split = totalMessage.split(DELIMITER);
            String username = split[1];
            session.setUsername(username);
            sessionManager.sendAll(username + "님이 입장했습니다.");

        } else if (totalMessage.startsWith("/message")) {
            // 클라이언트 전체에게 문자 보내기
            String[] split = totalMessage.split(DELIMITER);
            String message = split[1];
            sessionManager.sendAll "[" + session.getUsername() + "]" +
message);

        } else if (totalMessage.startsWith("/change")) {
            String[] split = totalMessage.split(DELIMITER);
            String changeName = split[1];
            sessionManager.sendAll(session.getUsername() + "님이 " + changeName
+ "로 이름을 변경했습니다.");
            session.setUsername(changeName);

        } else if (totalMessage.startsWith("/users")) {
            List<String> usernames = sessionManager.getAllUsername();
            StringBuilder sb = new StringBuilder();
            sb.append("전체 접속자 : ").append(usernames.size()).append("\n");
            for (String username : usernames) {
                sb.append(" - ").append(username).append("\n");
            }
            session.send(sb.toString());

        } else if (totalMessage.startsWith("/exit")) {

```

```

        throw new IOException("exit");

    } else {
        session.send("처리할 수 없는 명령어 입니다: " + totalMessage);
    }
}
}
}

```

#### 입장 /join|{name}

- 메시지는 | (파이프, 수직 막대) 구분자를 기준으로 나눈다.
- 이름을 구하고, session.setUsername(username) 를 사용해서 세션에 이름을 등록한다.
- 그리고 sessionManager.sendAll(username + "님이 입장했습니다.") 를 통해 모든 사람들에게 입장을 알린다.

#### 메시지 /message|{내용}

- 모든 사용자에게 메시지를 전달한다.
- 메시지를 전달할 때 누가 전달했는지 설명하기 위해
- [이름] 메시지 형식으로 전달한다.

#### 이름 변경 /change|{name}

- 사용자의 이름을 변경한다.
- 이때 사용자의 이름이 변경된 사실을 전체 사용자에게 알린다.

#### 전체 사용자 /users

- 채팅 서버에 접속한 전체 사용자 목록을 출력한다.

#### 종료 /exit

- 채팅 서버의 접속을 종료한다.

#### ServerMain - 코드 수정

```

package chat.server;

import java.io.IOException;

public class ServerMain {

    private static final int PORT = 12345;
}

```

```

public static void main(String[] args) throws IOException {
    SessionManager sessionManager = new SessionManager();

    // CommandManager 점진적으로 변경 예정
    // CommandManager commandManager = new
CommandManagerV1(sessionManager);
    CommandManager commandManager = new CommandManagerV2(sessionManager);

    Server server = new Server(PORT, commandManager, sessionManager);
    server.start();
}
}

```

- CommandManagerV1 → CommandManagerV2 를 사용하도록 코드를 변경하자.

## 실행 결과 - 서버

```

20:37:21.375 [    main] 서버 시작: class chat.server.CommandManagerV2
20:37:21.379 [    main] 서버 소켓 시작 - 리스닝 포트: 12345
20:37:25.054 [    main] 소켓 연결: Socket[addr=/
127.0.0.1,port=52784,localport=12345]
20:37:26.203 [ Thread-0] client -> server: /join|seon
20:37:26.204 [ Thread-0] server -> client: seon님이 입장했습니다.
20:37:28.590 [    main] 소켓 연결: Socket[addr=/
127.0.0.1,port=52795,localport=12345]
20:37:29.612 [ Thread-1] client -> server: /join|nate
20:37:29.612 [ Thread-1] server -> client: nate님이 입장했습니다.
20:37:29.613 [ Thread-1] server -> client: nate님이 입장했습니다.
20:37:35.045 [ Thread-0] client -> server: /message|hello nate
20:37:35.056 [ Thread-0] server -> client: [seon] hello nate
20:37:35.056 [ Thread-0] server -> client: [seon] hello nate
20:37:39.320 [ Thread-1] client -> server: /message|hi seon
20:37:39.321 [ Thread-1] server -> client: [nate] hi seon
20:37:39.321 [ Thread-1] server -> client: [nate] hi seon
20:38:01.570 [ Thread-0] client -> server: /change|seon2
20:38:01.572 [ Thread-0] server -> client: seon님이 seon2로 이름을 변경했습니다.
20:38:01.572 [ Thread-0] server -> client: seon님이 seon2로 이름을 변경했습니다.
20:38:09.185 [ Thread-0] client -> server: /users
20:38:09.186 [ Thread-0] server -> client: 전체 접속자 : 2
- seon2
- nate

```

```
20:38:24.588 [ Thread-0] client -> server: /exit
20:38:24.588 [ Thread-0] java.io.IOException: exit
20:38:24.589 [ Thread-0] server -> client: seon2님이 퇴장했습니다.
20:38:24.591 [ Thread-0] 연결 종료: Socket[addr=/
127.0.0.1,port=52784,localport=12345]
20:38:27.218 [ Thread-1] client -> server: /exit
20:38:27.218 [ Thread-1] java.io.IOException: exit
20:38:27.219 [ Thread-1] 연결 종료: Socket[addr=/
127.0.0.1,port=52795,localport=12345]
20:38:30.879 [ shutdown] shutdownHook 실행
20:38:30.880 [      main] 서버 소켓 종료: java.net.SocketException: Socket closed
```

### 실행 결과 - 클라이언트1

```
20:37:25.048 [      main] 클라이언트 시작
이름을 입력하세요.
seon
seon님이 입장했습니다.
nate님이 입장했습니다.
hello nate
[seon] hello nate
[nate] hi seon
/change|seon2
seon님이 seon2로 이름을 변경했습니다.
/users
전체 접속자 : 2
- seon2
- nate

/exit
20:38:24.588 [writeHandler] writeHandler 종료
20:38:24.588 [writeHandler] readHandler 종료
20:38:24.591 [writeHandler] 연결 종료: Socket[addr=localhost/
127.0.0.1,port=12345,localport=52784]
20:38:24.591 [readHandler] java.net.SocketException: Socket closed
```

### 실행 결과 - 클라이언트2

```
20:37:28.585 [      main] 클라이언트 시작
이름을 입력하세요.
nate
```



```
nate님이 입장했습니다.
[seon] hello nate
hi seon
[nate] hi seon
seon님이 seon2로 이름을 변경했습니다.
seon2님이 퇴장했습니다.
/exit
20:38:27.218 [writeHandler] writeHandler 종료
20:38:27.218 [writeHandler] readHandler 종료
20:38:27.219 [readHandler] java.io.EOFException
20:38:27.221 [writeHandler] 연결 종료: Socket[addr=localhost/
127.0.0.1,port=12345,localport=52795]
```

모든 요구사항을 만족하고, 모든 기능이 정상 수행되는 것을 확인할 수 있다.

### 주의! `change{name}`

구분자로 `|` (파이프)를 사용해야 한다. 공백을 사용하면 파싱에서 예외가 발생하면서 연결이 종료된다.

### 문제

앞으로 새로운 기능이 계속 추가될 수 있다.

그런데 각각의 개별 기능을 `CommandManager` 안에 if문으로 덕지덕지 추가하는 것이 영 마음에 들지 않는다.

새로운 기능이 추가되어도 기존 코드에 영향을 최소화하면서 기능을 추가해보자.

## 채팅 프로그램 - 서버3

각각의 명령어를 하나의 `Command` (명령)로 보고 인터페이스와 구현체로 구분해보자.

```
package chat.server.command;

import chat.server.Session;

import java.io.IOException;

public interface Command {
```

```
    void execute(String[] args, Session session) throws IOException;
}
```

- Command 인터페이스이다. 각각의 명령어 하나를 처리하는 목적으로 만들었다.

총 5개의 기능이 있으므로 5개의 기능을 각각의 Command (명령)으로 구현해보자.

- 입장 /join|{name}
- 메시지 /message|{내용}
- 이름 변경 /change|{name}
- 전체 사용자 /users
- 종료 /exit

```
package chat.server.command;

import chat.server.Session;
import chat.server.SessionManager;

public class JoinCommand implements Command {

    private final SessionManager sessionManager;

    public JoinCommand(SessionManager sessionManager) {
        this.sessionManager = sessionManager;
    }

    @Override
    public void execute(String[] args, Session session) {
        String username = args[1];
        session.setUsername(username);
        sessionManager.sendAll(username + "님이 입장했습니다.");
    }
}
```

```
package chat.server.command;

import chat.server.Session;
import chat.server.SessionManager;

public class MessageCommand implements Command {
```

```

private final SessionManager sessionManager;

public MessageCommand(SessionManager sessionManager) {
    this.sessionManager = sessionManager;
}

@Override
public void execute(String[] args, Session session) {
    String message = args[1];
    sessionManager.sendAll "[" + session.getUsername() + "]" + " " + message);
}
}

```

```

package chat.server.command;

import chat.server.Session;
import chat.server.SessionManager;

public class ChangeCommand implements Command {

    private final SessionManager sessionManager;

    public ChangeCommand(SessionManager sessionManager) {
        this.sessionManager = sessionManager;
    }

    @Override
    public void execute(String[] args, Session session) {
        String changeName = args[1];
        sessionManager.sendAll(session.getUsername() + "님이 " + changeName +
"로 이름을 변경했습니다.");
        session.setUsername(changeName);
    }
}

```

```

package chat.server.command;

import chat.server.Session;

```

```

import chat.server.SessionManager;

import java.io.IOException;
import java.util.List;

public class UsersCommand implements Command {

    private final SessionManager sessionManager;

    public UsersCommand(SessionManager sessionManager) {
        this.sessionManager = sessionManager;
    }

    @Override
    public void execute(String[] args, Session session) throws IOException {
        List<String> usernames = sessionManager.getAllUsername();
        StringBuilder sb = new StringBuilder();
        sb.append("전체 접속자 : ").append(usernames.size()).append("\n");
        for (String username : usernames) {
            sb.append(" - ").append(username).append("\n");
        }
        session.send(sb.toString());
    }
}

```

```

package chat.server.command;

import chat.server.Session;

import java.io.IOException;

public class ExitCommand implements Command {
    @Override
    public void execute(String[] args, Session session) throws IOException {
        throw new IOException("exit");
    }
}

```

명령어를 관리하고 찾아서 실행하는 `CommandManagerV3` 를 만들자.

```

package chat.server;

import chat.server.command.*;

import java.io.IOException;
import java.util.HashMap;
import java.util.Map;

public class CommandManagerV3 implements CommandManager {
    public static final String DELIMITER = "\\|";
    private final Map<String, Command> commands = new HashMap<>();

    public CommandManagerV3(SessionManager sessionManager) {
        commands.put("/join", new JoinCommand(sessionManager));
        commands.put("/message", new MessageCommand(sessionManager));
        commands.put("/change", new ChangeCommand(sessionManager));
        commands.put("/users", new UsersCommand(sessionManager));
        commands.put("/exit", new ExitCommand());
    }

    @Override
    public void execute(String totalMessage, Session session) throws
    IOException {
        String[] args = totalMessage.split(DELIMITER);
        String key = args[0];

        Command command = commands.get(key);
        if (command == null) {
            session.send("처리할 수 없는 명령어 입니다: " + totalMessage);
            return;
        }
        command.execute(args, session);
    }
}

```

### Map<String, Command> commands

명령어는 Map에 보관한다. 명령어 자체를 Key로 사용하고, 각 Key 해당하는 Command 구현체를 저장해둔다.

### execute()

```
Command command = commands.get(key)
```

명령어(key)를 처리할 Command 구현체를 commands에서 찾아서 실행한다.

예를 들어 `/join`이라는 메시지가 들어왔다면 `JoinCommand`의 인스턴스가 반환된다.

- `command`를 찾았다면, 다형성을 활용해서 구현체의 `execute()` 메서드를 호출한다.
- 만약 찾을 수 없다면 처리할 수 없는 명령어이다. 이 경우 처리할 수 없다는 메시지를 전달한다.

## ServerMain - 코드 수정

```
package chat.server;

import java.io.IOException;

public class ServerMain {

    private static final int PORT = 12345;

    public static void main(String[] args) throws IOException {
        SessionManager sessionManager = new SessionManager();

        // CommandManager 점진적으로 변경 예정
        // CommandManager commandManager = new
        CommandManagerV1(sessionManager);
        // CommandManager commandManager = new
        CommandManagerV2(sessionManager);
        CommandManager commandManager = new CommandManagerV3(sessionManager);

        Server server = new Server(PORT, commandManager, sessionManager);
        server.start();
    }
}
```

- `CommandManagerV3`를 실행하도록 코드를 변경하자.

## 실행 결과

실행 결과는 이전 예제인 `CommandManagerV2`와 같다.

## 참고 - 동시성과 읽기

여러 스레드가 `commands = new HashMap<>()`을 동시에 접근해서 데이터를 조회한다. 하지만 `commands`는 데이터 초기화 이후에는 데이터를 전혀 변경하지 않는다. 따라서 여러 스레드가 동시에 값을 조회해도 문제가 발생하지 않는다. 만약 `commands`의 데이터를 중간에 변경할 수 있게 하려면 동시성 문제를 고민해야 한다.

## 채팅 프로그램 - 서버4

이전 예제에서 `command` 가 없는 경우에 `null` 을 체크하고 처리해야 하는 부분이 좀 지저분하다.

```
Command command = commands.get(key);
if (command == null) {
    session.send("처리할 수 없는 명령어 입니다: " + totalMessage);
    return;
}
command.execute(args, session);
```

만약 명령어가 항상 존재한다면 다음과 같이 명령어를 찾고 바로 실행하는 깔끔한 코드를 작성할 수 있을 것이다.

```
Command command = commands.get(key);
command.execute(args, session);
```

이 문제의 해결 방안은 이외로 간단하다.

바로 `null` 인 상황을 처리할 객체를 만들어버리는 것이다.

```
package chat.server.command;

import chat.server.Session;

import java.io.IOException;
import java.util.Arrays;

public class DefaultCommand implements Command {
    @Override
    public void execute(String[] args, Session session) throws IOException {
        session.send("처리할 수 없는 명령어 입니다: " + Arrays.toString(args));
    }
}
```

```

package chat.server;

import chat.server.command.*;

import java.io.IOException;
import java.util.HashMap;
import java.util.Map;

public class CommandManagerV4 implements CommandManager {
    public static final String DELIMITER = "\\|";
    private final Map<String, Command> commands;
    private final Command defaultCommand = new DefaultCommand();

    public CommandManagerV4(SessionManager sessionManager) {
        commands = new HashMap<>();
        commands.put("/join", new JoinCommand(sessionManager));
        commands.put("/message", new MessageCommand(sessionManager));
        commands.put("/change", new ChangeCommand(sessionManager));
        commands.put("/users", new UsersCommand(sessionManager));
        commands.put("/exit", new ExitCommand());
    }

    @Override
    public void execute(String totalMessage, Session session) throws
    IOException {
        String[] args = totalMessage.split(DELIMITER);
        String key = args[0];

        // NullObject Pattern
        Command command = commands.getOrDefault(key, defaultCommand);
        command.execute(args, session);
    }
}

```

- Map에는 getOrDefault(key, defaultObject)라는 메서드가 존재한다.
- 만약 key를 사용해서 객체를 찾을 수 있다면 찾고, 찾을 수 없다면 옆에 있는 defaultObject를 반환한다.
- 이 기능을 사용하면 null을 받지 않고 항상 Command 객체를 받아서 처리할 수 있다.
- 여기서는 key를 찾을 수 없다면 DefaultCommand의 인스턴스를 반환한다.

## ServerMain 코드 수정

```

package chat.server;

```



```

import java.io.IOException;

public class ServerMain {

    private static final int PORT = 12345;

    public static void main(String[] args) throws IOException {
        SessionManager sessionManager = new SessionManager();

        // CommandManager 점진적으로 변경 예정
        //CommandManager commandManager = new
CommandManagerV1(sessionManager);
        //CommandManager commandManager = new
CommandManagerV2(sessionManager);
        //CommandManager commandManager = new
CommandManagerV3(sessionManager);
        CommandManager commandManager = new CommandManagerV4(sessionManager);

        Server server = new Server(PORT, commandManager, sessionManager);
        server.start();
    }
}

```

- CommandManagerV4를 사용하자.

## 실행 결과

- 기존과 같다.

## Null Object Pattern

이와 같이 null을 객체(Object)처럼 처리하는 방법을 Null Object Pattern 이라 한다.

이 디자인 패턴은 null 대신 사용할 수 있는 특별한 객체를 만들어, null로 인해 발생할 수 있는 예외 상황을 방지하고 코드의 간결성을 높이는 데 목적이 있다.

Null Object Pattern을 사용하면 null 값 대신 특정 동작을 하는 객체를 반환하게 되어, 클라이언트 코드에서 null 체크를 할 필요가 없어진다. 이 패턴은 코드에서 불필요한 조건문을 줄이고 객체의 기본 동작을 정의하는 데 유용하다.

## Command Pattern

지금까지 우리가 작성한 `Command` 인터페이스와 그 구현체들이 바로 커맨드 패턴을 사용한 것이다. 커맨드 패턴은 디자인 패턴 중 하나로, 요청을 독립적인 객체로 변환해서 처리한다.

커맨드 패턴의 특징은 다음과 같다.

- 분리: 작업을 호출하는 객체와 작업을 수행하는 객체를 분리한다.
- 확장성: 기존 코드를 변경하지 않고 새로운 명령을 추가할 수 있다.

### 커맨드 패턴의 장점

- 이 패턴의 장점은 새로운 커맨드를 쉽게 추가할 수 있다는 점이다. 예를 들어, 새로운 커맨드를 추가하고 싶다면, 새로운 `Command`의 구현체만 만들면 된다. 그리고 기존 코드를 대부분 변경할 필요가 없다.
- 작업을 호출하는 객체와 작업을 수행하는 객체가 분리되어 있다. 이전 코드에서는 작업을 호출하는 `if` 문이 각 작업마다 등장했는데, 커맨드 패턴에서는 이런 부분을 하나로 모아서 처리할 수 있다.
- 각각의 기능이 명확하게 분리된다. 개발자가 어떤 기능을 수정해야 할 때, 수정해야 하는 클래스가 아주 명확해진다.

### 커맨드 패턴의 단점

- **복잡성 증가:** 간단한 작업을 수행하는 경우에도 `Command` 인터페이스와 구현체들, `Command` 객체를 호출하고 관리하는 클래스등 여러 클래스를 생성해야 하기 때문에 코드의 복잡성이 증가할 수 있다.
- 모든 설계에는 트레이드 오프가 있다. 예를 들어 단순한 `if`문 몇개로 해결할 수 있는 문제에 복잡한 커맨드 패턴을 도입하는 것은 좋은 설계가 아닐 수 있다.
- 기능이 어느정도 있고, 각각의 기능이 명확하게 나누어질 수 있고, 향후 기능의 확장까지 고려해야 한다면 커맨드 패턴은 좋은 대안이 될 수 있다.

## 정리