8. 네트워크 - 프로그램2

#1.인강/0.자바/6.자바-고급2편

네트워크 프로그램4 - 자원 정리1

지금까지 학습한 자원 정리를 네트워크 프로그램에 도입해서, 네트워크 프로그램이 안전하게 자원을 정리하도록 개발해 보자.
먼저 `finally`를 사용해서 자원을 정리해보고, 이후에 `try-with-resources`를 도입해보자.
참고로 뒤에서 설명하겠지만, `try-with-resources`를 항상 사용할 수 있는 것은 아니고, `finally`에서 직접 자원을 정리해야 하는 경우가 많이 있다.

우선 소켓과 스트림을 종료하기 위해 간단한 유틸리티 클래스를 하나 만들자. 여러곳에서 사용할 예정이므로 `network.tcp` 패키지에 만들자.

```
package network.tcp;

import java.io.IOException;
import java.io.InputStream;
import java.io.OutputStream;
import java.net.Socket;

import static util.MyLogger.log;

public class SocketCloseUtil {

    public static void closeAll(Socket socket, InputStream input, OutputStream
output) {
```

```
        close(input);
        close(output);
        close(socket);
    }

    public static void close(InputStream input) {
        if (input != null) {
            try {
                input.close();
            } catch (IOException e) {
                log(e.getMessage());
            }
        }
    }

    public static void close(OutputStream output) {
        if (output != null) {
            try {
                output.close();
            } catch (IOException e) {
                log(e.getMessage());
            }
        }
    }

    public static void close(Socket socket) {
        if (socket != null) {
            try {
                socket.close();
            } catch (IOException e) {
                log(e.getMessage());
            }
        }
    }
}
```

- 기본적인 `null` 체크와 자원 종료시 예외를 잡아서 처리하는 코드가 들어가 있다. 참고로 자원 정리 과정에서 문제가 발생해도 코드에서 직접 대응할 수 있는 부분은 거의 없다. 이 경우 간단히 로그를 남겨서 이후에 개발자가 인지할 수 있는 정도면 충분하다.
- 각각의 예외를 잡아서 처리했기 때문에 `Socket`, `InputStream`, `OutputStream` 중 하나를 닫는 과정에서 예외가 발생해도 다음 자원을 닫을 수 있다.
- `Socket`을 먼저 생성하고, `Socket`을 기반으로 `InputStream`, `OutputStream`을 생성하기 때문에 닫을

때는 `InputStream`, `OutputStream`을 먼저 닫고 `Socket`을 마지막에 닫아야 한다.

- 참고로 `InputStream`, `OutputStream`의 닫는 순서는 상관이 없다.

클라이언트 코드 먼저 자원을 잘 정리할 수 있도록 수정해보자.

```
package network.tcp.v4;

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

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

public class ClientV4 {

    public static final int PORT = 12345;

    public static void main(String[] args) {
        log("클라이언트 시작");

        // finally 블록에서 변수에 접근해야 한다. 따라서 try 블록 안에서 선언할 수 없다.
        Socket socket = null;
        DataInputStream input = null;
        DataOutputStream output = null;
        try {
            socket = new Socket("localhost", PORT);
            input = new DataInputStream(socket.getInputStream());
            output = new DataOutputStream(socket.getOutputStream());
            log("소캣 연결: " + socket);

            Scanner scanner = new Scanner(System.in);
            while (true) {
                System.out.print("전송 문자: ");
                String toSend = scanner.nextLine();

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

                if (toSend.equals("exit")) {
```

```
                    break;
                }

                // 서버로부터 문자 받기
                String received = input.readUTF();
                log("client <- server: " + received);
            }

        } catch (IOException e) {
            log(e);
        } finally {
            closeAll(socket, input, output);
            log("연결 종료: " + socket);
        }

    }
}
```

- 자원 정리시 `finally` 코드 블럭에서 `SocketCloseUtil.closeAll()`만 호출하면 된다.

이번에는 서버 코드를 수정하자.

```
package network.tcp.v4;

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 SessionV4 implements Runnable {

    private final Socket socket;

    public SessionV4(Socket socket) {
        this.socket = socket;
    }

    @Override
    public void run() {
        // finally 블록에서 변수에 접근해야 한다. 따라서 try 블록 안에서 선언할 수 없다.
```

```
        DataInputStream input = null;
        DataOutputStream output = null;

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

            while(true) {
                // 클라이언트로부터 문자 받기
                String received = input.readUTF();
                log("client -> server: " + received);

                if (received.equals("exit")) {
                    break;
                }

                // 클라이언트에게 문자 보내기
                String toSend = received + " World!";
                output.writeUTF(toSend);
                log("client <- server: " + toSend);
            }

            // 자원 정리
        } catch (IOException e) {
            log(e);
        } finally {
            closeAll(socket, input, output);
            log("연결 종료: " + socket);
        }
    }
}
```

- 자원 정리시 `finally` 코드 블럭에서 `SocketCloseUtil.closeAll()` 만 호출하면 된다.

ServerV4는 기존 `ServerV3` 와 같은 코드이다. `SessionV4` 만 사용하면 된다.

```
package network.tcp.v4;

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

import static util.MyLogger.log;
```

```
public class ServerV4 {

    private static final int PORT = 12345;

    public static void main(String[] args) throws IOException {
        log("서버 시작");
        ServerSocket serverSocket = new ServerSocket(PORT);
        log("서버 소켓 시작 - 리스닝 포트: " + PORT);

        while (true) {
            Socket socket = serverSocket.accept(); // 블로킹
            log("소켓 연결: " + socket);

            SessionV4 session = new SessionV4(socket);
            Thread thread = new Thread(session);
            thread.start();
        }
    }

}
```

- 기존 코드와 같다. `SessionV4`를 사용하는 부분을 꼭! 확인하자.

실행 결과 - 클라이언트 직접 종료시 서버의 로그

```
15:57:51.442 [     main] 서버 시작
15:57:51.444 [     main] 서버 소켓 시작 - 리스닝 포트: 12345
15:57:52.913 [     main] 소켓 연결: Socket[addr=/
127.0.0.1,port=51722,localport=12345]
15:57:54.784 [ Thread-0] java.io.EOFException
15:57:54.785 [ Thread-0] 연결 종료: Socket[addr=/
127.0.0.1,port=51722,localport=12345]
```

- 기존 코드의 문제는 클라이언트를 직접 종료하면 서버의 `Session`에 `EOFException`이 발생하면서 자원을 제대로 정리하지 못했다.
- 변경한 코드에서는 서버에 접속한 클라이언트를 직접 종료해도 서버의 `Session`이 연결 종료라는 메시지를 남기면서 자원을 잘 정리하는 것을 확인할 수 있다.

윈도우 OS 내용 추가

실행 결과 - 클라이언트 직접 종료시 서버의 로그 - 윈도우

```
12:10:51.513 [     main] 서버 시작
12:10:51.515 [     main] 서버 소켓 시작 - 리스닝 포트: 12345
12:10:54.222 [     main] 소켓 연결: Socket[addr=/
127.0.0.1,port=53608,localport=12345]
12:10:55.520 [ Thread-0] java.net.SocketException: Connection reset
12:10:55.521 [ Thread-0] 연결 종료: Socket[addr=/
127.0.0.1,port=53608,localport=12345]
```

참고로 윈도우의 경우 `java.io.EOFException`이 아니라 `java.net.SocketException: Connection reset`이 발생한다.
두 예외 모두 `java.io.IOException`의 자식이므로 예제 코드에서 예외를 잡을 수 있다.

둘의 예외 차이가 발생하는 이유는 다음과 같다.
클라이언트의 연결을 직접 종료하면 클라이언트 프로세스가 종료되면서, 클라이언트와 서버의 TCP 연결도 함께 종료된다. 이때 소켓을 정상적으로 닫지 않고 프로그램을 종료했기 때문에 각각의 OS는 남아있는 TCP 연결을 정리하려고 시도한다. 이때 MAC과 윈도우 OS의 TCP 연결 정리 방식이 다르다.

- MAC: TCP 연결 정상 종료
- 윈도우: TCP 연결 강제 종료

TCP 연결의 정상 종료와 강제 종료의 차이는 뒤의 네트워크 예외에서 설명한다.

네트워크 프로그램5 - 자원 정리2

이번에는 자원 정리에 `try-with-resources`를 적용해보자.

```
package network.tcp.v5;

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

import static util.MyLogger.log;
```

```
public class ClientV5 {

    public static final int PORT = 12345;

    public static void main(String[] args) {
        log("클라이언트 시작");

        try(Socket socket = new Socket("localhost", PORT);
            DataInputStream input = new
DataInputStream(socket.getInputStream());
            DataOutputStream output = new
DataOutputStream(socket.getOutputStream())) {

            log("소캣 연결: " + socket);

            Scanner scanner = new Scanner(System.in);
            while (true) {
                System.out.print("전송 문자: ");
                String toSend = scanner.nextLine();

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

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

                // 서버로부터 문자 받기
                String received = input.readUTF();
                log("client <- server: " + received);
            }

        } catch (IOException e) {
            log(e);
        }
    }

}
```

- 클라이언트에 `try-with-resources`를 적용했다.
- 자원 정리시 `try-with-resources`에 선언되는 순서의 반대로 자원 정리가 적용되기 때문에 여기서는 `output`, `input`, `socket` 순으로 `close()`가 호출된다.

- 참고로 `OutputStream`, `InputStream`, `Socket` 모두 `AutoCloseable`을 구현하고 있다.

```
package network.tcp.v5;

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

import static util.MyLogger.log;

public class SessionV5 implements Runnable {

    private final Socket socket;

    public SessionV5(Socket socket) {
        this.socket = socket;
    }

    @Override
    public void run() {
        try (socket;
             DataInputStream input = new
DataInputStream(socket.getInputStream());
             DataOutputStream output = new
DataOutputStream(socket.getOutputStream())) {

            while (true) {
                // 클라이언트로부터 문자 받기
                String received = input.readUTF();
                log("client -> server: " + received);

                if (received.equals("exit")) {
                    break;
                }

                // 클라이언트에게 문자 보내기
                String toSend = received + " World!";
                output.writeUTF(toSend);
                log("client <- server: " + toSend);
            }
```

```
        } catch (IOException e) {
            log(e);
        }

        log("연결 종료: " + socket + " isClosed: " + socket.isClosed());
    }

}
```

- 서버에 `try-with-resources` 를 적용했다.
- `Socket` 객체의 경우 `Session` 에서 직접 생성하는 것이 아니라 외부에서 받아오는 객체이다. 이 경우 `try` 선언부에 예제와 같이 객체의 참조를 넣어두면 자원 정리 시점에 `AutoCloseable` 이 호출된다.
- `AutoCloseable` 이 호출되어서 정말 소켓의 `close()` 메서드가 호출되었는지 확인하기 위해 마지막에 `socket.isClosed()` 를 호출하는 코드를 넣어두었다.

```
package network.tcp.v5;

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

import static util.MyLogger.log;

public class ServerV5 {

    private static final int PORT = 12345;

    public static void main(String[] args) throws IOException {
        log("서버 시작");
        ServerSocket serverSocket = new ServerSocket(PORT);
        log("서버 소켓 시작 - 리스닝 포트: " + PORT);

        while (true) {
            Socket socket = serverSocket.accept(); // 블로킹
            log("소켓 연결: " + socket);

            SessionV5 session = new SessionV5(socket);
            Thread thread = new Thread(session);
            thread.start();
        }
```

```
    }

}
```

- 기존 코드와 같다. `SessionV5`를 사용하는 부분을 꼭! 확인하자.

실행 결과 - 클라이언트 직접 종료시 서버의 로그

```
16:13:26.171 [     main] 서버 시작
16:13:26.173 [     main] 서버 소켓 시작 - 리스닝 포트: 12345
16:13:27.334 [     main] 소켓 연결: Socket[addr=/
127.0.0.1,port=53788,localport=12345]
16:13:29.256 [ Thread-0] java.io.EOFException
16:13:29.262 [ Thread-0] 연결 종료: Socket[addr=/
127.0.0.1,port=53788,localport=12345] isClosed: true
```

- 마지막에 있는 `isClosed: true` 로그를 통해 소켓의 `close()` 메서드가 `try-with-resources`를 통해 잘 호출된 것을 확인할 수 있다.

참고로 윈도우의 경우 `java.io.EOFException` 대신에 `java.net.SocketException: Connection reset` 예외가 발생한다.

네트워크 프로그램6 - 자원 정리3

이번에는 서버를 종료할 때, 서버 소켓과 연결된 모든 소켓 자원을 다 반납하고 서버를 안정적으로 종료하는 방법을 알아보자.
서버를 종료하려면 서버에 종료라는 신호를 전달해야 한다. 예를 들어서 서버도 콘솔 창을 통해서 입력을 받도록 만들고, "종료"라는 메시지를 입력하면 모든 자원을 정리하면서 서버가 종료되도록 하면 된다.
하지만 보통 서버에서 콘솔 입력은 잘 하지 않으므로, 이번에는 서버를 직접 종료하면서 자원도 함께 정리하는 방법을 알아보겠다.

셧다운 훅(Shutdown Hook)
자바는 프로세스가 종료될 때, 자원 정리나 로그 기록과 같은 종료 작업을 마무리 할 수 있는 셧다운 훅이라는 기능을 지원한다.
프로세스 종료는 크게 2가지로 분류할 수 있다.

- **정상 종료**

    - 모든 non 데몬 스레드의 실행 완료로 자바 프로세스 정상 종료
    - 사용자가 Ctrl+C를 눌러서 프로그램을 중단
    - `kill` 명령 전달 (`kill -9` 제외)
    - IntelliJ의 stop 버튼
- **강제 종료**
    - 운영체제에서 프로세스를 더 이상 유지할 수 없다고 판단할 때 사용
    - 리눅스/유닉스의 `kill -9`나 Windows의 `taskkill /F`

정상 종료의 경우에는 셧다운 훅이 작동해서 프로세스 종료 전에 필요한 후 처리를 할 수 있다.
반면에 강제 종료의 경우에는 셧다운 훅이 작동하지 않는다.

셧다운 훅의 사용 방법을 코드를 통해서 알아보고, 서버 종료시 자원도 함께 정리해보자.

클라이언트 코드는 기존 코드와 같다. 이름만 `ClientV6`로 복사해서 만들자.

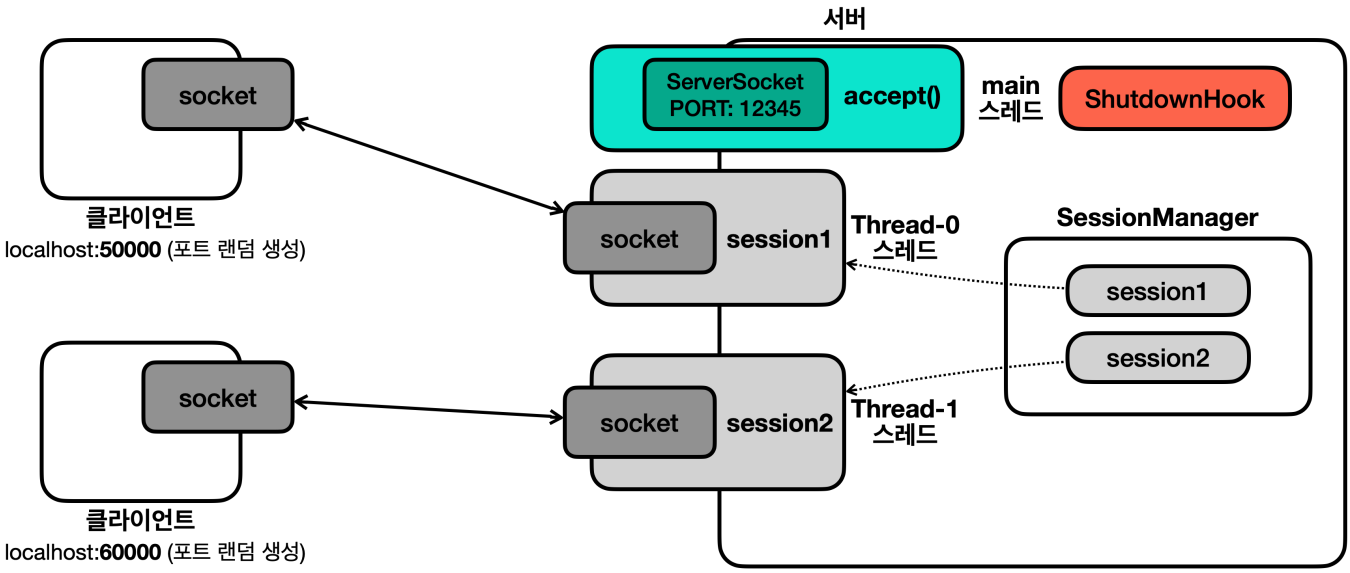
```
package network.tcp.v6;

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

import static util.MyLogger.log;

public class ClientV6 {

    public static final int PORT = 12345;

    public static void main(String[] args) {
        log("클라이언트 시작");

        try(Socket socket = new Socket("localhost", PORT);
            DataInputStream input = new
DataInputStream(socket.getInputStream());
            DataOutputStream output = new
DataOutputStream(socket.getOutputStream())) {

            log("소캣 연결: " + socket);
```

```
            Scanner scanner = new Scanner(System.in);
            while (true) {
                System.out.print("전송 문자: ");
                String toSend = scanner.nextLine();

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

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

                // 서버로부터 문자 받기
                String received = input.readUTF();
                log("client <- server: " + received);
            }

        } catch (IOException e) {
            log(e);
        }
    }

}
```

서버는 세션을 관리하는 세션 매니저가 필요하다.

```
package network.tcp.v6;

import java.util.ArrayList;
import java.util.List;

// 동시성 처리
public class SessionManagerV6 {

    private List<SessionV6> sessions = new ArrayList<>();

    public synchronized void add(SessionV6 session) {
        sessions.add(session);
    }

    public synchronized void remove(SessionV6 session) {
```

```
            sessions.remove(session);
        }

        public synchronized void closeAll() {
            for (SessionV6 session : sessions) {
                session.close();
            }
            sessions.clear();
        }
    }
```



각 세션은 소켓과 연결 스트림을 가지고 있다. 따라서 서버를 종료할 때 사용하는 세션들도 함께 종료해야 한다. 모든 세션들을 찾아서 종료하려면 생성한 세션을 보관하고 관리할 객체가 필요하다.

SessionManager

- `add()` : 클라이언트의 새로운 연결을 통해, 세션이 새로 만들어지는 경우 `add()` 를 호출해서 세션 매니저에 세션을 추가한다.
- `remove()` : 클라이언트의 연결이 끊어지면 세션도 함께 정리된다. 이 경우 `remove()` 를 호출해서 세션 매니저에서 세션을 제거한다.
- `closeAll()` : 서버를 종료할 때 사용하는 세션들도 모두 닫고, 정리한다.

```
package network.tcp.v6;

import java.io.DataInputStream;
```

```java
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 SessionV6 implements Runnable {

    private final Socket socket;
    private final DataInputStream input;
    private final DataOutputStream output;
    private final SessionManagerV6 sessionManager;
    private boolean closed = false;

    public SessionV6(Socket socket, SessionManagerV6 sessionManager) throws
IOException {
        this.socket = socket;
        this.input = new DataInputStream(socket.getInputStream());
        this.output = new DataOutputStream(socket.getOutputStream());
        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);

                if (received.equals("exit")) {
                    break;
                }

                // 클라이언트에게 문자 보내기
                String toSend = received + " World!";
                output.writeUTF(toSend);
                log("client <- server: " + toSend);
            }
        } catch (IOException e) {
            log(e);
```

```
            } finally {
                sessionManager.remove(this);
                close();
            }
        }

        // 세션 종료시, 서버 종료시 동시에 호출될 수 있다.
        public synchronized void close() {
            if (closed) {
                return;
            }
            closeAll(socket, input, output);
            closed = true;
            log("연결 종료: " + socket);
        }

    }
```

아쉽지만 `Session`은 이제 `try-with-resources`를 사용할 수 없다.
왜냐하면 서버를 종료하는 시점에도 `Session`의 자원을 정리해야 하기 때문이다.

try-with-resources는 사용과 해제를 함께 묶어서 처리할 때 사용한다.
`try-with-resources`는 try 선언부에서 사용한 자원을 try가 끝나는 시점에 정리한다. 따라서 try에서 자원의 선언과 자원 정리를 묶어서 처리할 때 사용할 수 있다. 하지만 지금은 서버를 종료하는 시점에도 `Session`이 사용하는 자원을 정리해야 한다. 서버를 종료하는 시점에 자원을 정리하는 것은 `Session` 안에 있는 `try-with-resources`를 통해 처리할 수 없다.

동시성 문제

```
public synchronized void close() {...}
```

- 자원을 정리하는 `close()` 메서드는 2곳에서 호출될 수 있다.
  - 클라이언트와 연결이 종료되었을 때(`exit` 또는 예외 발생)
  - 서버를 종료할 때
- 따라서 `close()`가 다른 스레드에서 동시에 중복 호출될 가능성이 있다.
- 이런 문제를 막기 위해 `synchronized` 키워드를 사용했다. 그리고 자원 정리 코드가 중복 호출 되는 것을 막기 위해 `closed` 변수를 플래그로 사용했다.

네트워크 프로그램6 - 자원 정리4

```
package network.tcp.v6;

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

import static util.MyLogger.log;

public class ServerV6 {

    private static final int PORT = 12345;

    public static void main(String[] args) throws IOException {
        log("서버 시작");
        SessionManagerV6 sessionManager = new SessionManagerV6();
        ServerSocket serverSocket = new ServerSocket(PORT);
        log("서버 소켓 시작 - 리스닝 포트: " + PORT);

        // ShutdownHook 등록
        ShutdownHook shutdownHook = new ShutdownHook(serverSocket,
sessionManager);
        Runtime.getRuntime().addShutdownHook(new Thread(shutdownHook,
"shutdown"));

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

                SessionV6 session = new SessionV6(socket, 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 SessionManagerV6 sessionManager;

        public ShutdownHook(ServerSocket serverSocket, SessionManagerV6
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);
            }
        }
    }
}
```

셧다운 훅 등록

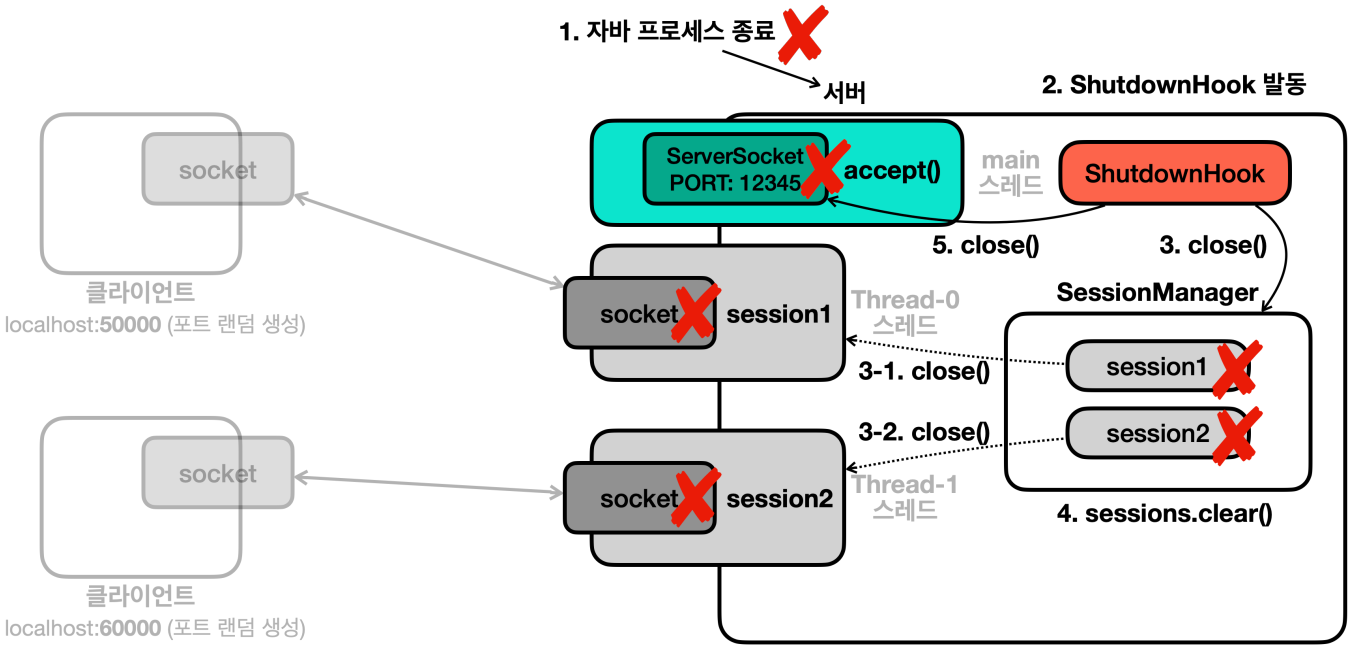
```
// ShutdownHook 등록
ShutdownHook shutdownHook = new ShutdownHook(serverSocket, sessionManager);
Runtime.getRuntime().addShutdownHook(new Thread(shutdownHook, "shutdown"));
```

- `Runtime.getRuntime().addShutdownHook()`을 사용하면 자바 종료시 호출되는 셧다운 훅을 등록할 수 있다.
- 여기에 셧다운이 발생했을 때 처리할 작업과 스레드를 등록하면 된다.

셧다운 훅 실행 코드

```
@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);
        }
    }
```



- 셧다운 훅이 실행될 때 모든 자원을 정리한다.
- `sessionManager.closeAll()` : 모든 세션이 사용하는 자원(`Socket` , `InputStream` , `OutputStream`)을 정리한다.
- `serverSocket.close()` : 서버 소켓을 닫는다.

자원 정리 대기 이유

```
Thread.sleep(1000); // 자원 정리 대기
```

보통 모든 non 데몬 스레드의 실행이 완료되면 자바 프로세스가 정상 종료된다. 하지만 다음과 같은 종료도 있다.

- 사용자가 Ctrl+C를 눌러서 프로그램을 중단
- `kill` 명령 전달 (`kill -9` 제외)
- IntelliJ의 stop 버튼

이런 경우에는 non 데몬 스레드의 종료 여부와 관계없이 자바 프로세스가 종료된다.
단 셧다운 훅의 실행이 끝날 때 까지는 기다려준다.
셧다운 훅의 실행이 끝나면 non 데몬 스레드의 실행 여부와 상관 없이 자바 프로세스는 종료된다.
따라서 다른 스레드가 자원을 정리하거나 필요한 로그를 남길 수 있도록 셧다운 훅의 실행을 잠시 대기한다.

실행 결과 - 서버 종료 결과

```
17:43:57.484 [     main] 서버 시작
17:43:57.486 [     main] 서버 소켓 시작 - 리스닝 포트: 12345
17:43:59.037 [     main] 소켓 연결: Socket[addr=/
127.0.0.1,port=65407,localport=12345]
17:44:00.799 [ shutdown] shutdownHook 실행
17:44:00.800 [ shutdown] 연결 종료: Socket[addr=/
127.0.0.1,port=65407,localport=12345]
17:44:00.800 [ Thread-0] java.net.SocketException: Socket closed
17:44:00.800 [     main] 서버 소캣 종료: java.net.SocketException: Socket closed
```

서버를 종료하면 `shutdown` 스레드가 `shutdownHook`을 실행하고, 세션의 `Socket`의 연결을 `close()`로 닫는다.

- `[ Thread-0] java.net.SocketException: Socket closed`
- `Session`의 `input.readUTF()`에서 입력을 대기하는 `Thread-0` 스레드는 `SocketException: Socket closed` 예외를 받고 종료된다. 참고로 이 예외는 자신의 소켓을 닫았을 때 발생한다.

`shutdown` 스레드는 서버 소켓을 `close()`로 닫는다.

- `[     main] 서버 소캣 종료: java.net.SocketException: Socket closed`
- `serverSocket.accept();`에서 대기하고 있던 `main` 스레드는 `java.net.SocketException: Socket closed` 예외를 받고 종료된다.

정리

드디어 자원 정리까지 깔끔하게 해결한 서버 프로그램을 완성했다.

네트워크 예외1 - 연결 예외

네트워크 연결시 발생할 수 있는 예외들을 정리해보자.

네트워크 연결과 예외

```
package network.exception.connect;

import java.io.IOException;
import java.net.ConnectException;
import java.net.Socket;
import java.net.UnknownHostException;

public class ConnectMain {

    public static void main(String[] args) throws IOException {
        unknownHostEx1();
        unknownHostEx2();
        connectionRefused();
    }

    private static void unknownHostEx1() throws IOException {
        try {
            Socket socket = new Socket("999.999.999.999", 80);
        } catch (UnknownHostException e) {
            e.printStackTrace();
        }
    }

    private static void unknownHostEx2() throws IOException {
        try {
            Socket socket = new Socket("google.gogo", 80);
        } catch (UnknownHostException e) {
            e.printStackTrace();
        }
    }

    private static void connectionRefused() throws IOException {
        try {
            Socket socket = new Socket("localhost", 45678); // 미사용 포트
        } catch (ConnectException e) {
            // ConnectException: Connection refused
            e.printStackTrace();
        }
```

```
    }

}
```

실행 결과

```
java.net.UnknownHostException: 999.999.999.999
...
java.net.UnknownHostException: google.gogo
...
java.net.ConnectException: Connection refused
...
```

java.net.UnknownHostException

- 호스트를 알 수 없음
- `999.999.999.999`: 이런 IP는 존재하지 않는다.
- `google.gogo`: 이런 도메인 이름은 존재하지 않는다.

java.net.ConnectException: Connection refused

- `Connection refused` 메시지는 연결이 거절되었다는 뜻이다.
- 연결이 거절되었다는 것은, 우선은 네트워크를 통해 해당 IP의 서버 컴퓨터에 접속은 했다는 뜻이다.
- 그런데 해당 서버 컴퓨터가 45678 포트를 사용하지 않기 때문에 TCP 연결을 거절한다.
- IP에 해당하는 서버는 켜져있지만, 사용하는 PORT가 없을 때 주로 발생한다.
- 네트워크 방화벽 등에서 무단 연결로 인지하고 연결을 막을 때도 발생한다.
- 서버 컴퓨터의 OS는 이때 TCP RST(Reset)라는 패킷을 보내서 연결을 거절한다.
- 클라이언트가 연결 시도 중에 RST 패킷을 받으면 이 예외가 발생한다.

윈도우 OS

윈도우의 경우 다음과 같이 `Connection refused` 뒤에 `connect` 라는 메시지가 하나 더 붙는 차이가 있다.

- `java.net.ConnectException: Connection refused: connect`

TCP RST(Reset) 패킷

TCP 연결에 문제가 있다는 뜻이다. 이 패킷을 받으면 받은 대상은 바로 연결을 해제해야 한다.

네트워크 예외2 - 타임아웃

네트워크 연결을 시도해서 서버 IP에 연결 패킷을 전달했지만 응답이 없는 경우 어떻게 될까?

TCP 연결 타임아웃 - OS 기본

```
package network.exception.connect;

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

public class ConnectTimeoutMain1 {

    //Windows: 약 21초
    //Linux: 약 75초에서 180초 사이, mac test 75초
    //java.net.ConnectException: Operation timed out
    public static void main(String[] args) throws IOException {
        long start = System.currentTimeMillis();
        try {
            Socket socket = new Socket("192.168.1.250", 45678);
        } catch (ConnectException e) {
            // ConnectException: Operation timed out
            e.printStackTrace();
        }
        long end = System.currentTimeMillis();
        System.out.println("end = " + (end - start));
    }

}
```

- 사설 IP 대역(주로 공유기에서 사용하는 IP 대역)의 `192.168.1.250`을 사용했다. 혹시 해당 IP로 무언가 연결되어 있다면 다른 결과가 나올 수 있다. 이 경우 마지막 3자리를 변경해보자.
- 해당 IP로 연결 패킷을 보내지만 IP를 사용하는 서버가 없으므로 TCP 응답이 오지 않는다.
- 또는 해당 IP로 연결 패킷을 보내지만 해당 서버가 너무 바쁘거나 문제가 있어서 연결 응답 패킷을 보내지 못하는 경우도 있다.
- 그렇다면 이때 무한정 기다려야 할까?

OS 기본 대기 시간

TCP 연결을 시도했는데 연결 응답이 없다면 OS에는 연결 대기 타임아웃이 설정되어 있다.

- Windows: 약 21초
- Linux: 약 75초에서 180초 사이

- mac test 75초

해당 시간이 지나면 `java.net.ConnectException: Operation timed out`이 발생한다.

실행 결과

```
java.net.ConnectException: Operation timed out
    at java.base/sun.nio.ch.Net.connect0(Native Method)
    at java.base/sun.nio.ch.Net.connect(Net.java:589)
    at java.base/sun.nio.ch.Net.connect(Net.java:578)
    at java.base/sun.nio.ch.NioSocketImpl.connect(NioSocketImpl.java:583)
    at java.base/java.net.SocksSocketImpl.connect(SocksSocketImpl.java:327)
    at java.base/java.net.Socket.connect(Socket.java:752)
    at java.base/java.net.Socket.connect(Socket.java:687)
    at java.base/java.net.Socket.<init>(Socket.java:556)
    at java.base/java.net.Socket.<init>(Socket.java:325)
    at
network.exception.connect.ConnectTimeoutMain.connectionWait1(ConnectTimeoutMai
n.java:22)
    at
network.exception.connect.ConnectTimeoutMain.main(ConnectTimeoutMain.java:12)
end = 75008
```

TCP 연결을 클라이언트가 이렇게 오래 대기하는 것은 좋은 방법이 아니다.
연결이 안되면 고객에게 빠르게 현재 연결에 문제가 있다고 알려주는 것이 더 나은 방법이다.

윈도우 OS 내용 추가

실행 결과 - 윈도우

```
java.net.ConnectException: Connection timed out: connect
    at java.base/sun.nio.ch.Net.connect0(Native Method)
    at java.base/sun.nio.ch.Net.connect(Net.java:589)
    at java.base/sun.nio.ch.Net.connect(Net.java:578)
    at java.base/sun.nio.ch.NioSocketImpl.connect(NioSocketImpl.java:583)
    at java.base/java.net.SocksSocketImpl.connect(SocksSocketImpl.java:327)
    at java.base/java.net.Socket.connect(Socket.java:751)
    at java.base/java.net.Socket.connect(Socket.java:686)
    at java.base/java.net.Socket.<init>(Socket.java:555)
    at java.base/java.net.Socket.<init>(Socket.java:324)
    at
```

```
network.exception.connect.ConnectTimeoutMain1.main(ConnectTimeoutMain1.java:13
)
end = 21052
```

- 윈도우의 경우 해당 시간이 지나면 `java.net.ConnectException: Connection timed out: connect`가 발생한다. 예외는 같고, 오류 메시지가 다르다.

TCP 연결 타임아웃 - 직접 설정

TCP 연결 타임아웃 시간을 직접 설정해보자.

```
package network.exception.connect;

import java.io.IOException;
import java.net.InetSocketAddress;
import java.net.Socket;
import java.net.SocketTimeoutException;

public class ConnectTimeoutMain2 {

    public static void main(String[] args) throws IOException {
        try {
            Socket socket = new Socket();
            socket.connect(new InetSocketAddress("192.168.1.250", 45678),
1000);
        } catch (SocketTimeoutException e) {
            // // java.net.SocketTimeoutException: Connect timed out
            e.printStackTrace();
        }
    }

}
```

new Socket()

`Socket` 객체를 생성할 때 인자로 IP, PORT를 모두 전달하면 생성자에서 바로 TCP 연결을 시도한다.
하지만 IP, PORT를 모두 빼고 객체를 생성하면, 객체만 생성되고, 아직 연결은 시도하지 않는다.
추가적으로 필요한 설정을 더 한 다음에 `socket.connect()`를 호출하면 그때 TCP 연결을 시도한다.
이 방식을 사용하면 추가적인 설정을 더 할 수 있는데, 대표적으로 타임아웃을 설정할 수 있다.

```
public void connect(SocketAddress endpoint, int timeout) throws IOException
{...}
```

- `InetSocketAddress` : `SocketAddress`의 자식이다. IP, PORT 기반의 주소를 객체로 제공한다.
- `timeout` : 밀리초 단위로 연결 타임아웃을 지정할 수 있다.

타임아웃 시간이 지나도 연결이 되지 않으면 다음 예외가 발생한다.
`java.net.SocketTimeoutException: Connect timed out`

실행 결과

```
java.net.SocketTimeoutException: Connect timed out
    at java.base/
sun.nio.ch.NioSocketImpl.timedFinishConnect(NioSocketImpl.java:546)
    at java.base/sun.nio.ch.NioSocketImpl.connect(NioSocketImpl.java:592)
    at java.base/java.net.SocksSocketImpl.connect(SocksSocketImpl.java:327)
    at java.base/java.net.Socket.connect(Socket.java:752)
    at
network.exception.connect.ConnectTimeoutMain2.main(ConnectTimeoutMain2.java:13
)
```

실행해보면 설정한 시간인 1초가 지난 후 타임아웃 예외가 발생하는 것을 확인할 수 있다.

TCP 소켓 타임아웃 - read 타임아웃

타임아웃 중에 또 하나 중요한 타임아웃이 있다.
바로 소켓 타임아웃 또는 read 타임 아웃이라고 부르는 타임아웃이다.

앞에서 설명한 연결 타임아웃은 TCP 연결과 관련이 되어있다.
연결이 잘 된 이후에 클라이언트가 서버에 어떤 요청을 했다고 가정하자. 그런데 서버가 계속 응답을 주지 않는다면, 무한정 기다려야 하는 것일까?
서버에 사용자가 폭주하고 매우 느려져서 응답을 계속 주지 못하는 상황이라면 어떻게 해야할까?

이런 경우에 사용하는 것이 바로 소켓 타임아웃(read 타임아웃)이다.

```
package network.exception.connect;
```

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

public class SoTimeoutServer {

    public static void main(String[] args) throws IOException,
InterruptedException {
        ServerSocket serverSocket = new ServerSocket(12345);
        Socket socket = serverSocket.accept();

        Thread.sleep(1000000);
    }
}
```

- 서버는 소켓을 연결은 하지만, 아무런 응답을 주지 않는다.

```
package network.exception.connect;

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

public class SoTimeoutClient {

    public static void main(String[] args) throws IOException,
InterruptedException {
        Socket socket = new Socket("localhost", 12345);
        InputStream input = socket.getInputStream();
        try {
            socket.setSoTimeout(3000); // 타임아웃 시간 설정
            int read = input.read(); // 기본은 무한 대기
            System.out.println("read = " + read);
        } catch (Exception e) {
            e.printStackTrace();
        }

        socket.close();
    }
}
```

- `socket.setSoTimeout()`을 사용하면 밀리초 단위로 타임아웃 시간을 설정할 수 있다. 여기서는 3초를 설정했다.

3초가 지나면 다음 예외가 발생한다.
`java.net.SocketTimeoutException: Read timed out`

타임아웃 시간을 설정하지 않으면 `read()`는 응답이 올 때 까지 무한 대기한다.

클라이언트 실행 결과

```
java.net.SocketTimeoutException: Read timed out
    at java.base/sun.nio.ch.NioSocketImpl.timedRead(NioSocketImpl.java:278)
    at java.base/sun.nio.ch.NioSocketImpl.implRead(NioSocketImpl.java:304)
    at java.base/sun.nio.ch.NioSocketImpl.read(NioSocketImpl.java:346)
    at java.base/sun.nio.ch.NioSocketImpl$1.read(NioSocketImpl.java:796)
    at java.base/java.net.Socket$SocketInputStream.implRead(Socket.java:1108)
    at java.base/java.net.Socket$SocketInputStream.read(Socket.java:1095)
    at java.base/java.net.Socket$SocketInputStream.read(Socket.java:1089)
    at network.exception.connect.SoTimeoutClient.main(SoTimeoutClient.java:15)
```

실무 이야기
실무에서 자주 발생하는 장애 원인 중 하나가 바로 연결 타임아웃, 소켓 타임아웃(read 타임 아웃)을 누락하기 때문에 발생한다.
서버도 외부에 존재하는 데이터를 네트워크를 통해 불러와야 하는 경우가 있다.
예를 들어서 주문을 처리하는 서버가 있는데, 주문 서버는 외부에 있는 서버를 통해 고객의 신용카드 결제를 처리해야 한다고 가정해보자.

신용카드를 처리하는 회사가 3개 있다고 가정하자.
- 고객➡ 주문 서버➡ 신용카드A 회사 서버(정상)
- 고객➡ 주문 서버➡ 신용카드B 회사 서버(정상)
- 고객➡ 주문 서버➡ 신용카드C 회사 서버(문제)

신용카드 A, 신용카드 B 서버는 문제가 없고, 신용카드C 회사 서버에 문제가 발생해서 응답을 주지 못하는 상황이라고 가정해보자. 주문 서버는 계속 신용카드 C 회사 서버의 응답을 기다리게 된다.
여기서 문제는 신용카드C의 결제에 대해서 주문 서버도 고객에게 응답을 주지 못하고 계속 대기하게 된다. 신용카드C로 주문하는 고객이 누적 될 수록 주문 서버의 요청은 계속 쌓이게 되고, 신용카드 C 회사 서버의 응답을 기다리는 스레

드도 점점 늘어난다. 결국 주문 서버에 너무 많은 사용자가 접속하게 되면서 주문 서버에 장애가 발생하게 된다.

결과적으로 신용카드C 때문에 신용카드A, 신용카드B를 사용하는 고객까지 모두 주문을 할 수 없는 사태가 벌어진다.

이런 장애는 신용카드 C 회사의 문제일까? 아니면 주문 서버 개발자의 문제일까?
만약 주문 서버에 연결, 소켓 타임아웃을 적절히 설정했다면, 신용카드 C 회사 서버가 연결이 오래 걸리거나 응답을 주지 않을 때 타임아웃으로 처리할 수 있다. 이렇게 되면 요청이 쌓이지 않기 때문에, 주문 서버에 장애가 발생하지 않는다. 타임아웃이 발생하는 신용카드 C 사용자에게는 현재 문제가 있다는 안내를 하면 된다. 나머지 신용카드A, 신용카드B는 정상적으로 작동한다.

외부 서버와 통신을 하는 경우 반드시 연결 타임아웃과 소켓 타임아웃을 지정하자.

네트워크 예외3 - 정상 종료

TCP에는 2가지 종류의 종료가 있다.

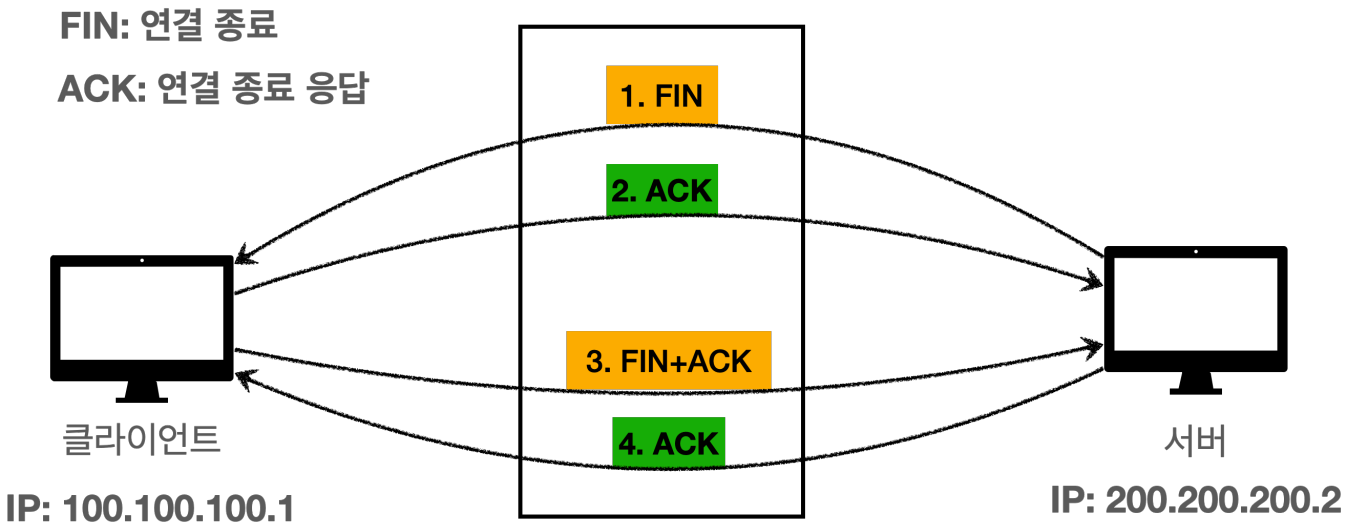
- 정상 종료
- 강제 종료

정상 종료

TCP에서 A, B가 서로 통신한다고 가정해보자.
TCP 연결을 종료하려면 서로 FIN 메시지를 보내야 한다.

- A (FIN)↦ B: A가 B로 FIN 메시지를 보낸다.
- A ↤(FIN) B: FIN 메시지를 받은 B도 A에게 FIN 메시지를 보낸다.

`socket.close()`를 호출하면 TCP에서 종료의 의미인 FIN 패킷을 상대방에게 전달한다.
FIN 패킷을 받으면 상대방도 `socket.close()`를 호출해서 FIN 패킷을 상대방에게 전달해야 한다.



- 클라이언트와 서버가 연결되어 있다.
- 서버가 연결 종료를 위해 `socket.close()`를 호출한다.
  - 서버는 클라이언트에 FIN 패킷을 전달한다.
- 클라이언트는 FIN 패킷을 받는다.
  - 클라이언트의 OS에서 FIN에 대한 ACK 패킷을 전달한다.
- 클라이언트도 종료를 위해 `socket.close()`를 호출한다.
  - 클라이언트는 서버에 FIN 패킷을 전달한다.
  - 서버의 OS는 FIN 패킷에 대한 ACK 패킷을 전달한다.

예제를 통해 소켓의 정상 종료에 대해 알아보자.

```
package network.exception.close.normal;

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

import static util.MyLogger.log;

public class NormalCloseServer {

    public static void main(String[] args) throws IOException,
InterruptedException {
        ServerSocket serverSocket = new ServerSocket(12345);
        Socket socket = serverSocket.accept();
        log("소캣 연결: " + socket);

        Thread.sleep(1000);
        socket.close();
        log("소캣 종료");
```

```
    }
}
```

- 서버는 소켓이 연결되면 1초 뒤에 연결을 종료한다.
- 서버에서 `socket.close()`를 호출하면 클라이언트에 FIN 패킷을 보낸다.

```
package network.exception.close.normal;

import java.io.*;
import java.net.Socket;

import static util.MyLogger.log;

public class NormalCloseClient {

    public static void main(String[] args) throws IOException,
InterruptedException {
        Socket socket = new Socket("localhost", 12345);
        log("소캣 연결: " + socket);
        InputStream input = socket.getInputStream();

        readByInputStream(input, socket);
        readByBufferedReader(input, socket);
        readByDataInputStream(input, socket);

        log("연결 종료: " + socket.isClosed());
    }

    private static void readByInputStream(InputStream input, Socket socket)
throws IOException {
        int read = input.read();
        log("read = " + read);
        if (read == -1) {
            input.close();
            socket.close();
        }
    }

    private static void readByBufferedReader(InputStream input, Socket socket)
throws IOException {
        BufferedReader br = new BufferedReader(new InputStreamReader(input));
        String readString = br.readLine();
```

```
            log("readString = " + readString);
            if (readString == null) {
                br.close();
                socket.close();
            }
        }

        private static void readByDataInputStream(InputStream input, Socket
socket) throws IOException {
            DataInputStream dis = new DataInputStream(input);
            try {
                dis.readUTF();
            } catch (EOFException e) {
                log(e);
            } finally {
                dis.close();
                socket.close();
            }
        }
}
```

- 클라이언트는 서버의 메시지를 3가지 방법으로 읽는다.
  - `read()` : 1byte 단위로 읽음
  - `readLine()` : 라인 단위로 `String` 으로 읽음
  - `readUTF()` : `DataInputStream` 을 통해 `String` 단위로 읽음

실행 결과

```
10:28:42.121 [     main] 소캣 연결: Socket[addr=localhost/
127.0.0.1,port=12345,localport=49188]
10:28:43.128 [     main] read = -1
10:28:43.130 [     main] readString = null
10:28:43.130 [     main] java.io.EOFException
10:28:43.130 [     main] 연결 종료: true
```

전체 과정

- 클라이언트가 서버에 접속한다.
- 클라이언트는 `input.read()` 로 서버의 데이터를 읽기 위해 대기한다.
- 그런데 1초 뒤에 서버에서 연결을 종료한다.
  - 서버에서 `socket.close()` 를 호출하면 클라이언트에 FIN 패킷을 보낸다.

- 클라이언트는 FIN 패킷을 받는다.
- 서버가 소켓을 종료했다는 의미이므로 클라이언트는 더는 읽을 데이터가 없다.
- `FIN` 패킷을 받은 클라이언트의 소켓은 더는 서버를 통해 읽을 데이터가 없다는 의미로 -1(EOF)를 반환한다.

여기서 각각의 상황에 따라 EOF를 해석하는 방법이 다르다.

- `read()` ➜ -1
  - EOF의 의미를 숫자 -1로 반환한다.
- `BufferedReader().readLine()` ➜ `null`
  - `BufferedReader()`는 문자 `String`을 반환한다. 따라서 -1을 표현할 수 없다. 대신에 `null`을 반환한다.
- `DataInputStream.readUTF()` ➜ `EOFException`
  - `DataInputStream`은 이 경우 `EOFException`을 던진다.
  - 예외를 통해서 연결을 종료할 수 있는 방법을 제공한다.

여기서 중요한 점은 EOF가 발생하면 상대방이 FIN 메시지를 보내면서 소켓 연결을 끊었다는 뜻이다.
이 경우 소켓에 다른 작업을 하면 안되고, FIN 메시지를 받은 클라이언트도 `close()`를 호출해서 상대방에 FIN 메시지를 보내고 소켓 연결을 끊어야 한다.
이렇게 하면 서로 FIN 메시지를 주고 받으면서 TCP 연결이 정상 종료된다.

네트워크 예외4 - 강제 종료

강제 종료

TCP 연결 중에 문제가 발생하면 `RST`라는 패킷이 발생한다. 이 경우 연결을 즉시 종료해야 한다.

```
package network.exception.close.reset;

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

import static util.MyLogger.log;

public class ResetCloseServer {

    public static void main(String[] args) throws IOException,
```

```
InterruptedException {
        ServerSocket serverSocket = new ServerSocket(12345);
        Socket socket = serverSocket.accept();
        log("소캣 연결: " + socket);

        socket.close();
        serverSocket.close();
        log("소캣 종료");
    }
}
```

- 서버는 소켓이 연결되면 단순히 연결을 종료한다.

```
package network.exception.close.reset;

import java.io.IOException;
import java.io.InputStream;
import java.io.OutputStream;
import java.net.Socket;
import java.net.SocketException;

import static util.MyLogger.log;

public class ResetCloseClient {

    public static void main(String[] args) throws IOException,
InterruptedException {
        Socket socket = new Socket("localhost", 12345);
        log("소캣 연결: " + socket);
        InputStream input = socket.getInputStream();
        OutputStream output = socket.getOutputStream();

        // client <- server: FIN
        Thread.sleep(1000); // 서버가 close() 호출할 때 까지 잠시 대기

        // client -> server: PUSH[1]
        output.write(1);

        // client <- server: RST
        Thread.sleep(1000); // RST 메시지 전송 대기

        try {
```

```
            // java.net.SocketException: Connection reset 발생!
            int read = input.read();
            System.out.println("read = " + read);
        } catch (SocketException e) {
            e.printStackTrace();
        }

        try {
            output.write(1);
        } catch (SocketException e) {
            //java.net.SocketException: Broken pipe
            e.printStackTrace();
        }

    }

}
```

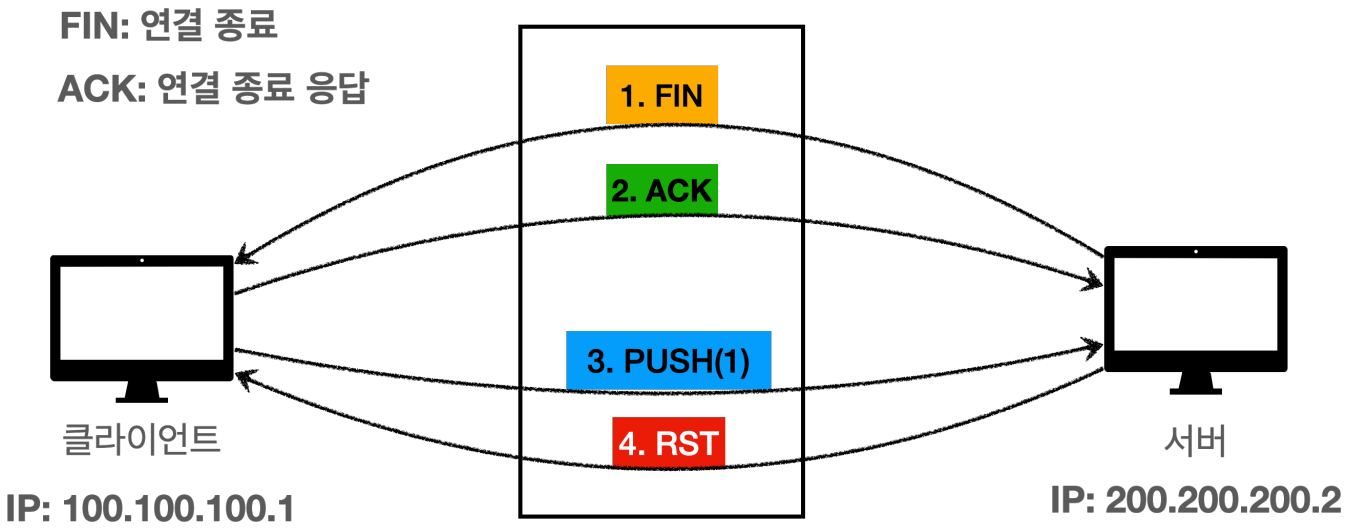
실행 결과

```
11:10:36.119 [     main] 소캣 연결: Socket[addr=localhost/
127.0.0.1,port=12345,localport=54490]
java.net.SocketException: Connection reset
...
java.net.SocketException: Broken pipe
...
```

실행 결과 - 윈도우

```
11:10:36.119 [     main] 소캣 연결: Socket[addr=localhost/
127.0.0.1,port=12345,localport=54490]
java.net.SocketException: 현재 연결은 사용자의 호스트 시스템의 소프트웨어의 의해 중단되었습니다
...
java.net.SocketException: 현재 연결은 사용자의 호스트 시스템의 소프트웨어의 의해 중단되었습니다
...
```

- 윈도우의 경우 같은 `java.net.SocketException`이 발생하지만 오류 메시지가 다르다.



- 클라이언트와 서버가 연결되어 있다.
- 서버는 종료를 위해 `socket.close()`를 호출한다.
  - 서버는 클라이언트에 FIN 패킷을 전달한다.
- 클라이언트는 FIN 패킷을 받는다.
  - 클라이언트의 OS에서 FIN에 대한 ACK 패킷을 전달한다.
- 클라이언트는 `output.write(1)`를 통해 서버에 메시지를 전달한다.
  - 데이터를 전송하는 PUSH 패킷이 서버에 전달된다.
- 서버는 이미 FIN으로 종료를 요청했는데, PUSH 패킷으로 데이터가 전송되었다.
  - 서버가 기대하는 값은 FIN 패킷이다.
- 서버는 TCP 연결에 문제가 있다고 판단하고 즉각 연결을 종료하라는 RST 패킷을 클라이언트에 전송한다.

RST 패킷이 도착했다는 것은 현재 TCP 연결에 심각한 문제가 있으므로 해당 연결을 더는 사용하면 안된다는 의미이다.

RST 패킷이 도착하면 자바는 `read()`로 메시지를 읽을 때 다음 예외를 던진다.

- MAC: `java.net.SocketException: Connection reset`
- 윈도우: `java.net.SocketException: 현재 연결은 사용자의 호스트 시스템의 소프트웨어의 의해 중단되었습니다`

RST 패킷이 도착하면 자바는 `write()`로 메시지를 전송할 때 다음 예외를 던진다.

- MAC: `java.net.SocketException: Broken pipe`
- 윈도우: `java.net.SocketException: 현재 연결은 사용자의 호스트 시스템의 소프트웨어의 의해 중단되었습니다`

참고 - RST(Reset)

- TCP에서 RST 패킷은 연결 상태를 초기화(리셋)해서 더 이상 현재의 연결을 유지하지 않겠다는 의미를 전달한다.

여기서 "Reset"은 현재의 세션을 강제로 종료하고, 연결을 무효화하라는 뜻이다.

- RST 패킷은 TCP 연결에 문제가 있는 다양한 상황에 발생한다. 예를 들어서 다음과 같은 경우들이 있다.
  - TCP 스펙에 맞지 않는 순서로 메시지가 전달될 때
  - TCP 버퍼에 있는 데이터를 아직 다 읽지 않았는데, 연결을 종료할 때
  - 방화벽 같은 곳에서 연결을 강제로 종료할 때도 발생한다.

참고 - java.net.SocketException: Socket is closed

- 자기 자신의 소켓을 닫은 이후에 `read()`, `write()`을 호출할 때 발생한다.

정리

- 상대방이 연결을 종료한 경우 데이터를 읽으면 EOF가 발생한다.
  - -1, `null`, `EOFException` 등이 발생한다.
  - 이 경우 연결을 끊어야 한다.
- `java.net.SocketException: Connection reset`
  - RST 패킷을 받은 이후에 `read()` 호출
  - 윈도우 오류 메시지: `현재 연결은 사용자의 호스트 시스템의 소프트웨어의 의해 중단되었습니다`
- `java.net.SocketException: Broken pipe`
  - RST 패킷을 받은 이후에 `write()` 호출
  - 윈도우 오류 메시지: `현재 연결은 사용자의 호스트 시스템의 소프트웨어의 의해 중단되었습니다`
- `java.net.SocketException: Socket is closed`
  - 자신이 소켓을 닫은 이후에 `read()`, `write()` 호출

네트워크 종료와 예외 정리

네트워크에서 이런 예외를 다 따로따로 이해하고 다루어야 할까? 사실 어떤 문제가 언제 발생할지 자세하게 다 구분해서 처리하기는 어렵다.

따라서 기본적으로 정상 종료, 강제 종료 모두 자원 정리하고 닫도록 설계하면 된다.

예를 들어서 `SocketException`, `EOFException`은 모두 `IOException`의 자식이다. 따라서 `IOException`이 발생하면 자원을 정리하면 된다. 만약 더 자세히 분류해야 하는 경우가 발생하면 그때 예외를 구분해서 처리하면 된다.

정리와 문제

채팅 프로그램 만들기

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.