7. 네트워크 - 프로그램1

#1.인강/0.자바/6.자바-고급2편

네트워크 프로그램1 - 예제

이제 본격적으로 자바 네트워크 프로그램을 작성해보자.

여기서는 TCP/IP로 개발할 예정이다. (UDP는 직접 사용할 일이 많지 않으므로 다루지 않겠다.)

프로그램을 작성하기 전에 스레드 정보와 현재 시간을 출력하는 간단한 로깅 유틸리티를 만들자

(멀티스레드 강의에서 사용했던 로깅 유틸리티 클래스다.)

```
package util;

import java.time.LocalTime;
import java.time.format.DateTimeFormatter;

public abstract class MyLogger {

    private static final DateTimeFormatter formatter =
DateTimeFormatter.ofPattern("HH:mm:ss.SSS");

    public static void log(Object obj) {
        String time = LocalTime.now().format(formatter);
        System.out.printf("%s [%9s] %s\n", time,
Thread.currentThread().getName(), obj);
    }

}
```

- 앞으로 대부분의 출력은 이 로깅 클래스를 사용하겠다.

이번에 만들 프로그램은 아주 간단한 네트워크 클라이언트, 서버 프로그램이다.
클라이언트가 "Hello"라는 문자를 서버에 전달하면 서버는 클라이언트의 요청에 " World!"라는 단어를 더해서 "Hello World!"라는 단어를 클라이언트에 응답한다.

- 클라이언트 ➡ 서버: "Hello"
- 클라이언트 ⬅ 서버: "Hello World!"

먼저 코드를 작성해서 실행한 후, 그 다음에 설명하겠다.

```
package network.tcp.v1;

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

import static util.MyLogger.log;

public class ClientV1 {

    private static final int PORT = 12345;

    public static void main(String[] args) throws IOException {
        log("클라이언트 시작");
        Socket socket = new Socket("localhost", PORT);
        DataInputStream input = new DataInputStream(socket.getInputStream());
        DataOutputStream output = new
DataOutputStream(socket.getOutputStream());
        log("소캣 연결: " + socket);

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

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

```
        // 자원 정리
        log("연결 종료: " + socket);
        input.close();
        output.close();
        socket.close();
    }
}
```

```
package network.tcp.v1;

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

import static util.MyLogger.log;

public class ServerV1 {

    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);

        Socket socket = serverSocket.accept();
        log("소켓 연결: " + socket);
        DataInputStream input = new DataInputStream(socket.getInputStream());
        DataOutputStream output = new
DataOutputStream(socket.getOutputStream());

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

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

```
        // 자원 정리
        log("연결 종료: " + socket);
        input.close();
        output.close();
        socket.close();
        serverSocket.close();
    }

}
```

서버를 먼저 실행하고, 그 다음에 클라이언트를 실행해야 한다.

실행 결과 - 클라이언트

```
10:36:42.921 [     main] 클라이언트 시작
10:36:42.927 [     main] 소캣 연결: Socket[addr=localhost/
127.0.0.1,port=12345,localport=57587]
10:36:42.927 [     main] client -> server: Hello
10:36:42.928 [     main] client <- server: Hello World!
10:36:42.929 [     main] 연결 종료: Socket[addr=localhost/
127.0.0.1,port=12345,localport=57587]
```

참고: 클라이언트의 localport는 랜덤으로 생성되므로 다른 숫자가 나올 수 있다. 이 부분은 뒤에서 설명한다.

실행 결과 - 서버

```
10:36:39.161 [     main] 서버 시작
10:36:39.163 [     main] 서버 소켓 시작 - 리스닝 포트: 12345
10:36:42.927 [     main] 소켓 연결: Socket[addr=/
127.0.0.1,port=57587,localport=12345]
10:36:42.927 [     main] client -> server: Hello
10:36:42.928 [     main] client <- server: Hello World!
10:36:42.928 [     main] 연결 종료: Socket[addr=/
127.0.0.1,port=57587,localport=12345]
```

localhost, 127.0.0.1

- `localhost`는 현재 사용 중인 컴퓨터 자체를 가리키는 특별한 호스트 이름이다.
  - `google.com`, `naver.com`과 같은 호스트 이름이지만, 자기 자신의 컴퓨터를 뜻하는 이름이다.
- `localhost`는 127.0.0.1이라는 IP로 매핑된다.
- 127.0.0.1은 IP 주소 체계에서 루프백 주소(loopback address)로 지정된 특별한 IP 주소이다. 이 주소는 컴퓨터가 스스로를 가리킬 때 사용되며, "localhost"와 동일하게 취급된다.
- 127.0.0.1은 컴퓨터가 네트워크 인터페이스를 통해 외부로 나가지 않고, 자신에게 직접 네트워크 패킷을 보낼 수 있도록 한다.

주의 - localhost

만약 `localhost`가 잘 되지 않는다면, 자신의 PC의 운영체제에 `localhost`가 127.0.0.1로 매핑되어 있지 않은 문제이다. 이 경우 `localhost` 대신에 127.0.0.1을 직접 입력하면 된다.

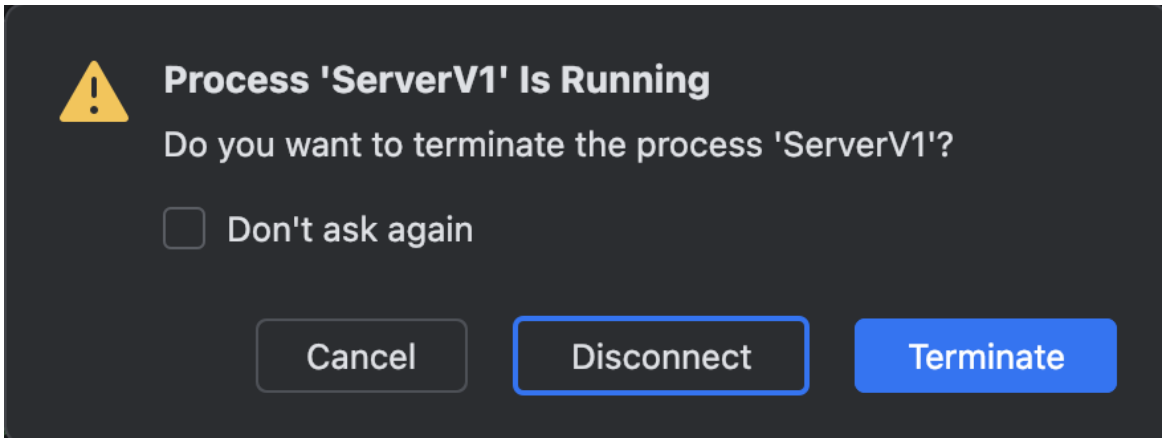
주의! - 서버 연결 불가

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

- 서버를 시작하지 않고, 클라이언트만 실행하면 해당 예외가 발생한다.
- 커넥션 및 소켓과 관련된 다양한 예외들이 있는데, 이런 예외들은 뒤에서 한번에 정리하겠다.

주의 - BindException

`Exception in thread "main" java.net.BindException: Address already in use`

- 만약 이런 예외가 발생한다면, 이미 12345라는 포트를 다른 프로세스가 사용하고 있다는 뜻이다. 포트를 다른 숫자로 변경해서 사용하자.
- 우리가 작성한 서버 프로그램을 아직 종료하지 않은 상태로 다시 실행하는 경우에도 12345 포트를 이미 점유하고 있으므로 같은 예외가 발생할 수 있다.
- 인텔리J를 종료할 때 반드시 `Terminate`로 종료해야 한다. `Disconnect`로 종료하면 12345 포트를 점유하고 있는 서버 프로세스가 그대로 살아있는 상태로 유지된다. 따라서 다시 실행할 경우 12345 포트를 사용하지 못한다.
  - 

  - 이 경우 해당 자바 프로그램을 찾아서 종료하거나 또는 PC를 재부팅 하면 된다.

네트워크 프로그램1 - 분석

TCP/IP 통신에서는 통신할 대상 서버를 찾을 때 호스트 이름이 아니라, IP 주소가 필요하다.

네트워크 프로그램을 분석하기 전에 먼저 호스트 이름으로 IP를 어떻게 찾는지 확인해보자.

DNS 탐색

```
package network.tcp.v1;

import java.net.InetAddress;
import java.net.UnknownHostException;

public class InetAddressMain {

    public static void main(String[] args) throws UnknownHostException {
        InetAddress localhost = InetAddress.getByName("localhost");
        System.out.println(localhost);

        InetAddress google = InetAddress.getByName("google.com");
        System.out.println(google);
    }
}
```

실행 결과

```
localhost/127.0.0.1
google.com/142.250.199.110
```

자바의 `InetAddress` 클래스를 사용하면 호스트 이름으로 대상 IP를 찾을 수 있다.

찾는 과정은 다음과 같다.

1. 자바는 `InetAddress.getByName("호스트명")` 메서드를 사용해서 해당하는 IP 주소를 조회합니다.
2. 이 과정에서 시스템의 호스트 파일을 먼저 확인한다.
    - `/etc/hosts` (리눅스, mac)
    - `C:\Windows\System32\drivers\etc\hosts` (윈도우,Windows)
3. 호스트 파일에 정의되어 있지 않다면, DNS 서버에 요청해서 IP 주소를 얻는다.

호스트 파일 - 예시

```
127.0.0.1    localhost
255.255.255.255  broadcasthost
::1              localhost
```

만약 호스트 파일에 `localhost`가 없다면, `127.0.0.1 localhost`를 추가하거나 또는 127.0.0.1과 같은 IP를 직접 사용하면 된다.

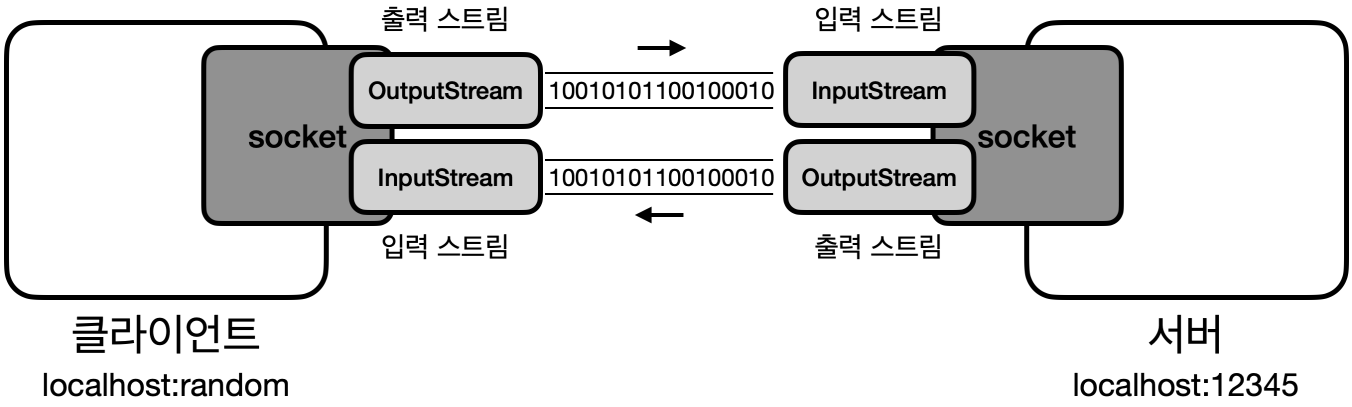
클라이언트 코드 분석

클라이언트와 서버의 연결은 Socket을 사용한다.

```
Socket socket = new Socket("localhost", PORT)
```

- `localhost`를 통해 자신의 컴퓨터에 있는 12345 포트에 TCP 접속을 시도한다.
  - `localhost`는 IP가 아니므로 해당하는 IP를 먼저 찾는다. 내부에서 `InetAddress`를 사용한다.
  - `localhost`는 127.0.0.1이라는 IP에 매핑되어 있다.
  - `127.0.0.1:12345`에 TCP 접속을 시도한다.
- 연결이 성공적으로 완료되면 `Socket` 객체를 반환한다.
- `Socket`은 서버와 연결되어 있는 연결점이라고 생각하면 된다.
- `Socket` 객체를 통해서 서버와 통신할 수 있다.

클라이언트와 서버간의 데이터 통신은 Socket이 제공하는 스트림을 사용한다.




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

- `Socket`은 서버와 데이터를 주고 받기 위한 스트림을 제공한다.
- `InputStream`: 서버에서 전달한 데이터를 클라이언트가 받을 때 사용한다.
- `OutputStream`: 클라이언트에서 서버에 데이터를 전달할 때 사용한다.
- `InputStream`, `OutputStream`을 그대로 사용하면 모든 데이터를 byte로 변환해서 전달해야 하기 때문에 번거롭다. 여기서는 `DataInputStream`, `DataOutputStream`이라는 보조 스트림을 사용해서, 자바 타입의 메시지를 편리하게 주고 받을 수 있도록 했다.

```
// 서버에게 문자 보내기
String toSend = "Hello";
output.writeUTF(toSend);
```

- `OutputStream`을 통해 서버에 "Hello" 메시지를 전송한다.

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

- `InputStream`을 통해 서버가 전달한 메시지를 받을 수 있다.
- 클라이언트가 "Hello"를 전송하면 서버는 " World!" 라는 문자를 붙여서 반환하므로 "Hello World!"라는 문자를 반환 받는다.

사용한 자원은 반드시 정리해야 한다.

```
// 자원 정리
log("연결 종료: " + socket);
input.close();
output.close();
socket.close();
```

사용이 끝나면 사용한 자원은 반드시 반납해야 한다. 지금은 간단하고 허술하게 자원 정리를 했지만, 뒤에서 자원 정리를 매우 자세히 다루겠다.

서버 코드 분석

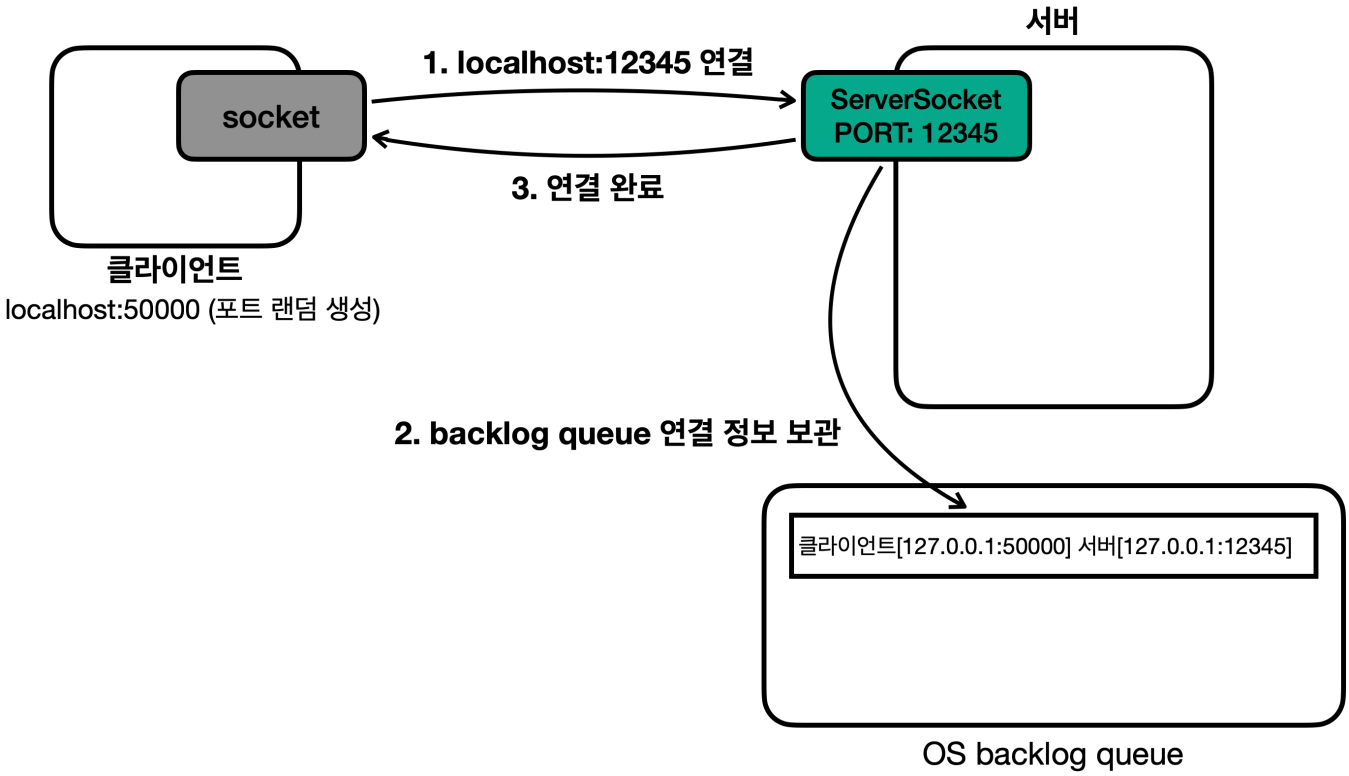
서버 소켓

서버는 특정 포트를 열어두어야 한다. 그래야 클라이언트가 해당 포트를 지정해서 접속할 수 있다.

```
ServerSocket serverSocket = new ServerSocket(PORT);
```

- 서버는 서버 소켓(`ServerSocket`)이라는 특별한 소켓을 사용한다.
- 지정한 포트를 사용해서 서버 소켓을 생성하면, 클라이언트는 해당 포트로 서버에 연결할 수 있다.

클라이언트와 서버의 연결 과정을 그림으로 자세히 알아보자.



- 서버가 12345 포트로 서버 소켓을 열어둔다. 클라이언트는 이제 12345 포트로 서버에 접속할 수 있다.
- 클라이언트가 12345 포트에 연결을 시도한다.
- 이때 OS 계층에서 TCP 3 way handshake가 발생하고, TCP 연결이 완료된다.
- TCP 연결이 완료되면 서버는 OS backlog queue라는 곳에 클라이언트와 서버의 TCP 연결 정보를 보관한다.
  - 이 연결 정보를 보면 클라이언트의 IP, PORT, 서버의 IP, PORT 정보가 모두 들어있다.

클라이언트와 랜덤 포트

TCP 연결시에는 클라이언트 서버 모두 IP, 포트 정보가 필요하다. 예제에서 사용된 IP 포트는 다음과 같다.

- **클라이언트**: localhost(127.0.0.1), 50000(포트 랜덤 생성)
- **서버**: localhost(127.0.0.1), 12345

그런데 생각해보면 클라이언트 자신의 포트를 지정한 적이 없다.

서버의 경우 포트가 명확하게 지정되어 있어야 한다. 그래야 클라이언트에서 서버에 어떤 포트에 접속할지 알 수 있다.

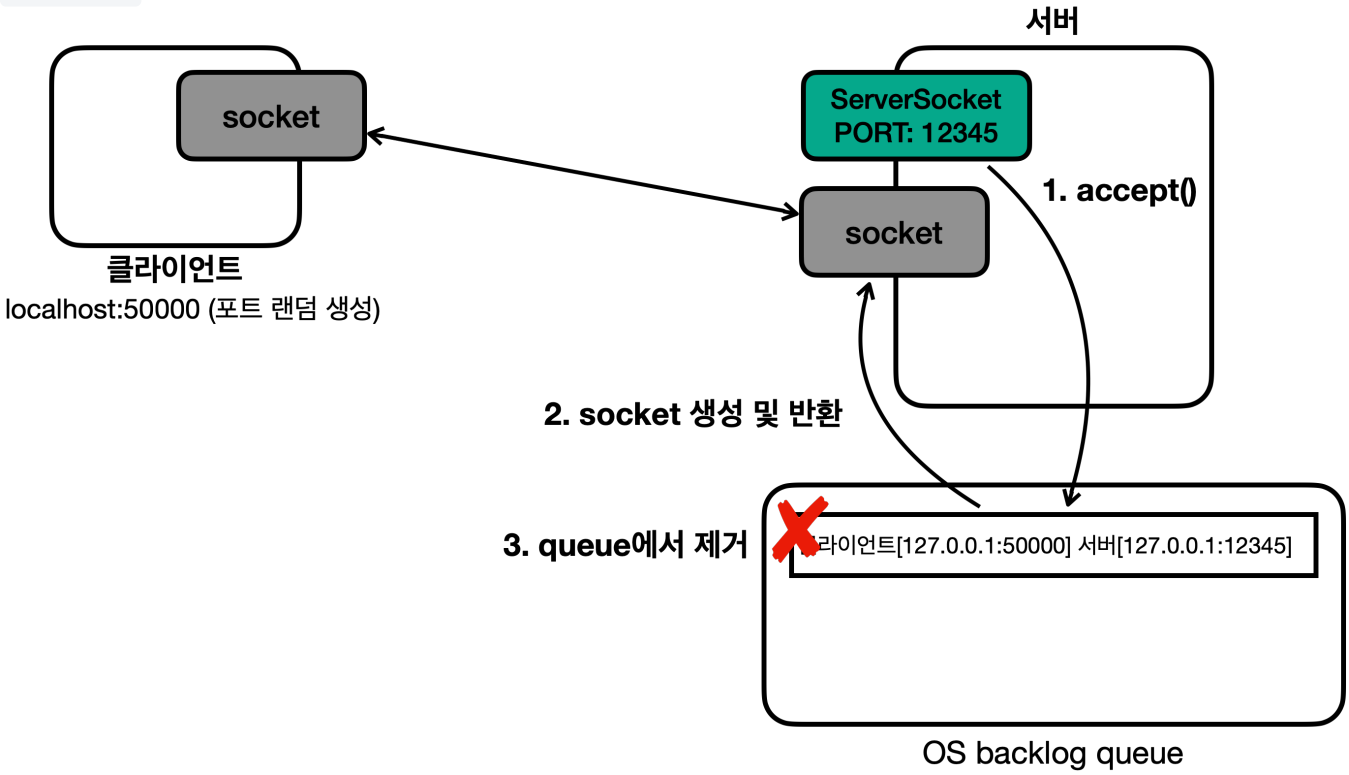
반면에 서버에 접속하는 클라이언트의 경우에는 자신의 포트가 명확하게 지정되어 있지 않아도 된다. 클라이언트는 보통 포트를 생략하는데, 생략할 경우 클라이언트 PC에 남아 있는 포트 중 하나가 랜덤으로 할당된다. 참고로 클라이언트의 포트도 명시적으로 할당할 수는 있지만 잘 사용하지 않는다.

accept()

```
Socket socket = serverSocket.accept();
```

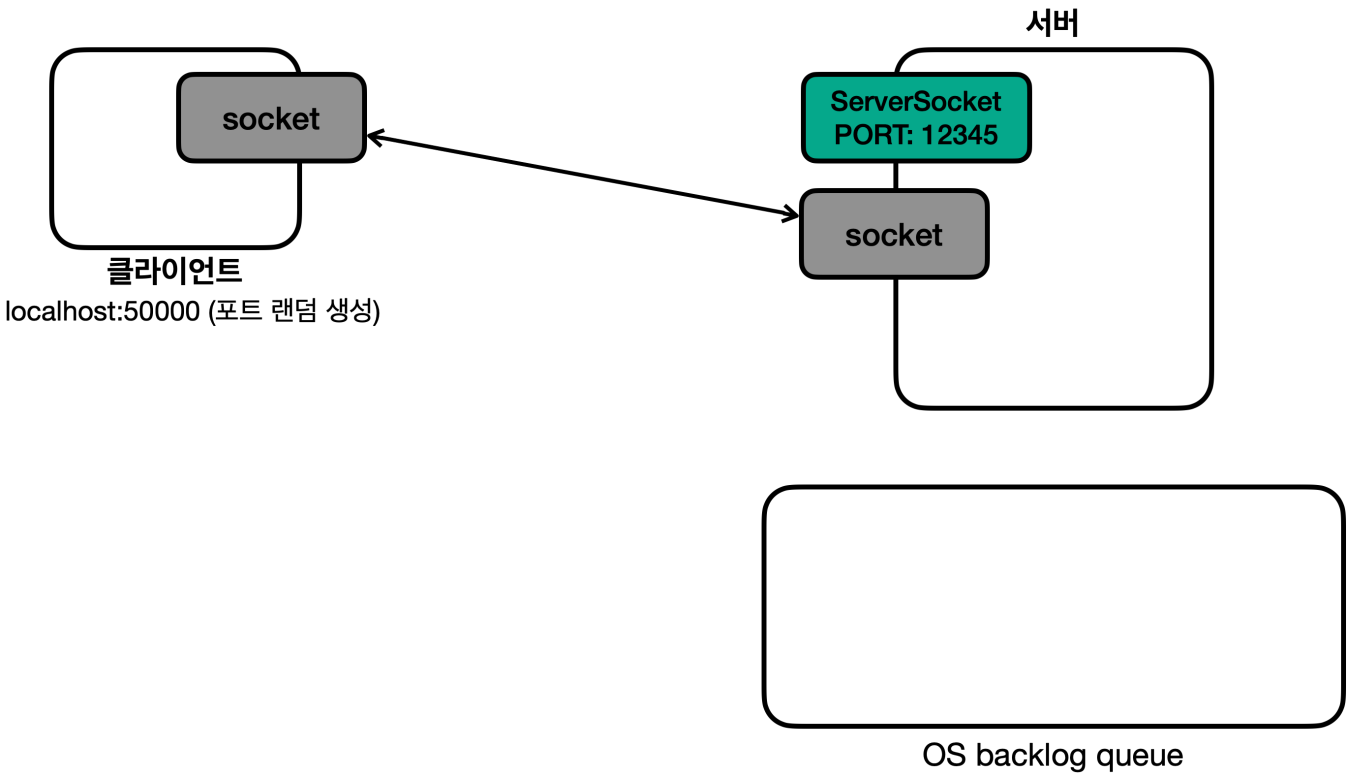
- **서버 소켓은 단지 클라이언트와 서버의 TCP 연결만 지원하는 특별한 소켓이다.**
- 실제 클라이언트와 서버가 정보를 주고 받으려면 `Socket` 객체가 필요하다. (서버 소켓이 아니다! 소켓이다!)
- `serverSocket.accept()` 메서드를 호출하면 TCP 연결 정보를 기반으로, `Socket` 객체를 만들어서 반환한다.

`accpet()` 호출 과정을 그림으로 자세히 알아보자.



- `accept()`를 호출하면 backlog queue에서 TCP 연결 정보를 조회한다.
  - 만약 TCP 연결 정보가 없다면, 연결 정보가 생성될 때 까지 대기한다. (블로킹)
- 해당 정보를 기반으로 `Socket` 객체를 생성한다.
- 사용한 TCP 연결 정보는 backlog queue에서 제거된다.

Socket 생성 후 그림



- 클라이언트와 서버의 `Socket`은 TCP로 연결되어 있고, 스트림을 통해 메시지를 주고 받을 수 있다.

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

- `Socket`은 클라이언트와 서버가 데이터를 주고 받기 위한 스트림을 제공한다.
- `InputStream`: 서버 입장에서 보면 클라이언트가 전달한 데이터를 서버가 받을 때 사용한다.
- `OutputStream`: 서버에서 클라이언트에 데이터를 전달할 때 사용한다.

클라이언트의 Output은 서버의 Input이고 반대로 서버의 Output은 클라이언트의 Input이다.
자신을 기준으로 생각하면 된다.

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

- 클라이언트가 전달한 "Hello" 메시지를 전달 받는다.

```
// 클라이언트에게 문자 보내기
String toSend = received + " World!";
output.writeUTF(toSend);
```

- 클라이언트의 메시지에 " World!" 메시지를 붙여서 반환한다.
- `OutputStream`을 통해 서버에서 클라이언트로 메시지를 전송한다.

```
// 자원 정리
log("연결 종료: " + socket);
input.close();
output.close();
socket.close();
serverSocket.close();
```

- 필요한 자원을 사용하고 나면 꼭! 정리해야 한다.

문제

이 프로그램은 메시지를 하나만 주고 받으면 클라이언트와 서버가 모두 종료된다. 메시지를 계속 주고 받고, 원할 때 종료할 수 있도록 변경해보자.

네트워크 프로그램2 - 예제

이번에는 클라이언트와 서버가 메시지를 계속 주고 받다가, "exit"라고 입력하면 클라이언트와 서버를 종료해보자.

```
package network.tcp.v2;

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 ClientV2 {

    public static final int PORT = 12345;
```

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

        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);
        }

        // 자원 정리
        log("연결 종료: " + socket);
        input.close();
        output.close();
        socket.close();
    }
}
```

- 클라이언트와 서버가 메시지를 주고 받는 부분만 `while`로 반복하면 된다.
- `exit`를 입력하면 클라이언트는 `exit` 메시지를 서버에 전송하고, 클라이언트는 `while`문을 빠져나가면서 연결을 종료한다.

```
package network.tcp.v2;

import java.io.DataInputStream;
```

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

import static util.MyLogger.log;

public class ServerV2 {

    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);

        Socket socket = serverSocket.accept(); // 블로킹
        log("소켓 연결: " + socket);
        DataInputStream input = new DataInputStream(socket.getInputStream());
        DataOutputStream output = new
DataOutputStream(socket.getOutputStream());

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

            // 클라이언트 종료시 서버도 함께 종료
            if (received.equals("exit")) {
                break;
            }

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

        // 자원 정리
        log("연결 종료: " + socket);
        input.close();
        output.close();
        socket.close();
```

```
            serverSocket.close();
        }

    }
```

- 클라이언트와 서버가 메시지를 주고 받는 부분만 `while`로 반복하면 된다.
- 클라이언트로 부터 `exit` 메시지가 전송되면, 서버는 `while`문을 빠져나가면서 연결을 종료한다.

실행 결과 - 클라이언트

```
13:59:50.460 [     main] 클라이언트 시작
13:59:50.466 [     main] 소캣 연결: Socket[addr=localhost/
127.0.0.1,port=12345,localport=56524]
전송 문자: hello
13:59:53.882 [     main] client -> server: hello
13:59:53.882 [     main] client <- server: hello World!
전송 문자: hi
13:59:55.017 [     main] client -> server: hi
13:59:55.018 [     main] client <- server: hi World!
전송 문자: exit
13:59:56.552 [     main] client -> server: exit
13:59:56.552 [     main] 연결 종료: Socket[addr=localhost/
127.0.0.1,port=12345,localport=56524]
```

실행 결과 - 서버

```
13:59:47.388 [     main] 서버 시작
13:59:47.389 [     main] 서버 소켓 시작 - 리스닝 포트: 12345
13:59:50.466 [     main] 소켓 연결: Socket[addr=/
127.0.0.1,port=56524,localport=12345]
13:59:53.882 [     main] client -> server: hello
13:59:53.882 [     main] client <- server: hello World!
13:59:55.017 [     main] client -> server: hi
13:59:55.018 [     main] client <- server: hi World!
13:59:56.552 [     main] client -> server: exit
13:59:56.553 [     main] 연결 종료: Socket[addr=/
127.0.0.1,port=56524,localport=12345]
```

덕분에 클라이언트와 서버가 필요할 때 까지 계속 메시지를 주고 받을 수 있다.

문제

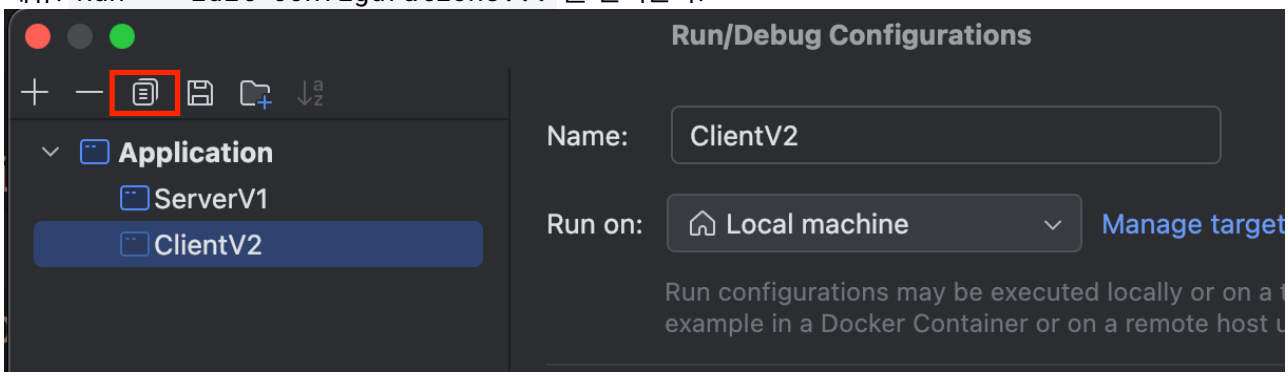
서버는 하나의 클라이언트가 아니라, 여러 클라이언트의 연결을 처리할 수 있어야 한다.

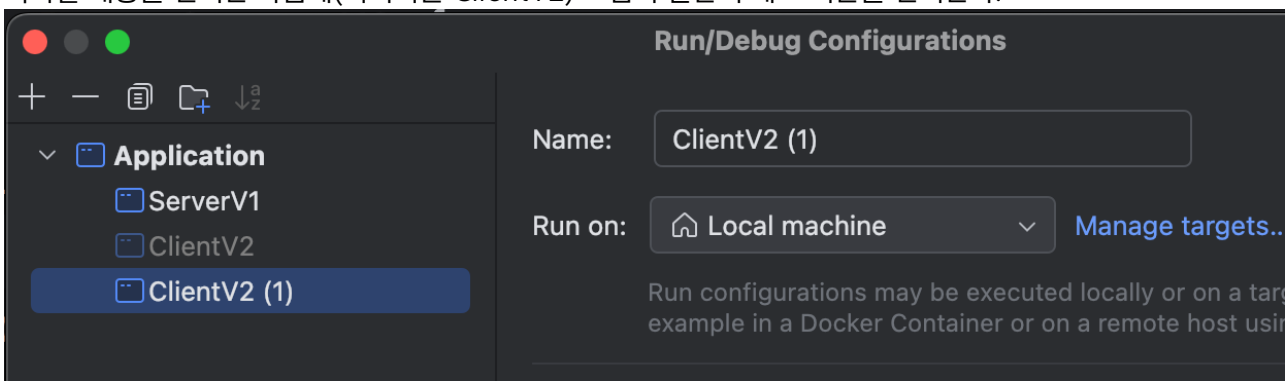
여러 클라이언트가 하나의 서버에 접속하도록 해보자.

참고 - IntelliJ에서 같은 클라이언트 동시에 실행하기

IntelliJ에서 같은 클래스의 `main()` 메서드를 다시 실행하면 기존에 실행하던 자바 프로세스를 종료해버리고 다시 실행한다.

여기서는 같은 `ClientV2`를 여러 개 실행해야 한다.

- 메뉴: `Run -> Edit Configurations...` 를 선택한다.
- 

- 복사할 내용을 선택한 다음에(여기서는 ClientV2) 그림의 빨간색 네모 버튼을 선택한다.
- 

- 그림과 같이 새로운 설정이 복사된 것을 확인할 수 있다. 해당 설정으로 실행하면 같은 자바 코드를 중복 실행할 수 있다.
- 여기서 `Name` 부분을 `ClientV2-2`라고 변경해두자

그런데 새로운 클라이언트가 접속하면 다음과 같이 정상 수행이 되지 않는다.

ClientV2-2 실행 결과

```
14:12:42.420 [     main] 클라이언트 시작
14:12:42.426 [     main] 소캣 연결: Socket[addr=localhost/
127.0.0.1,port=12345,localport=58298]
```

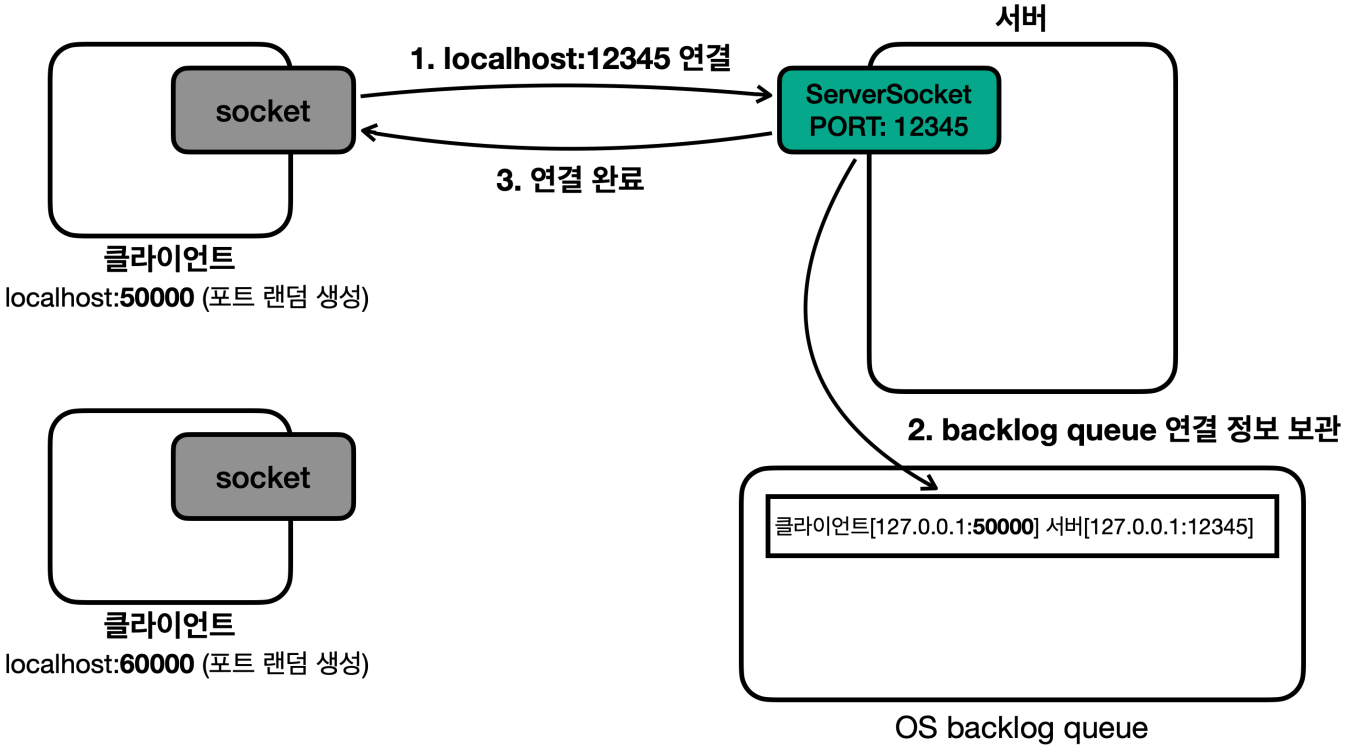
```
전송 문자: hello
14:12:46.158 [     main] client -> server: hello
```

처음 접속한 `ClientV2`는 문제없이 작동한다. 하지만 중간에 새로 연결한 `ClientV2-2`는 소켓 연결은 되지만, 메시지를 전송해도 서버로 부터 아무런 응답이 오지 않는다.
왜 이런 문제가 발생하는 것일까?

네트워크 프로그램2 - 분석

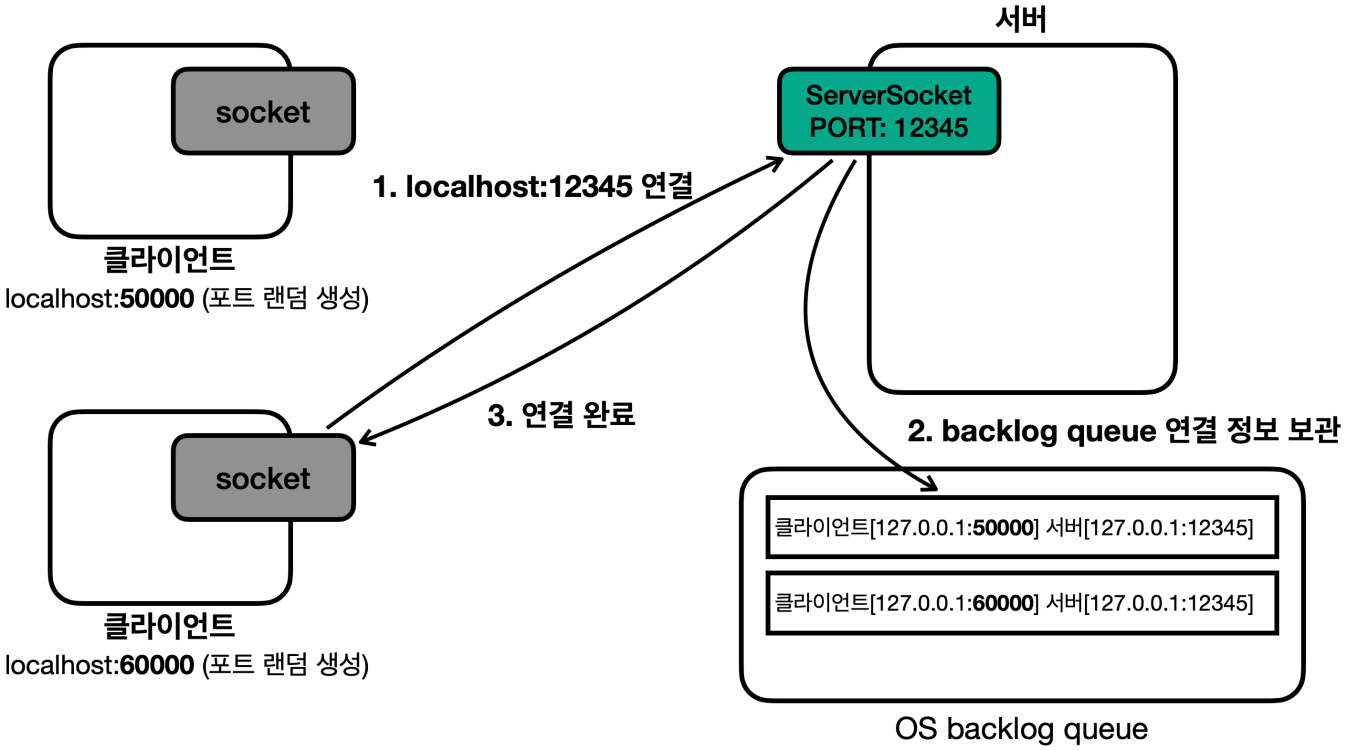
서버 소켓과 연결 더 자세히

서버 소켓과 연결에 대해서 좀 더 자세히 알아보자. 이번에는 여러 클라이언트가 서버가 접속한다고 가정해보자.

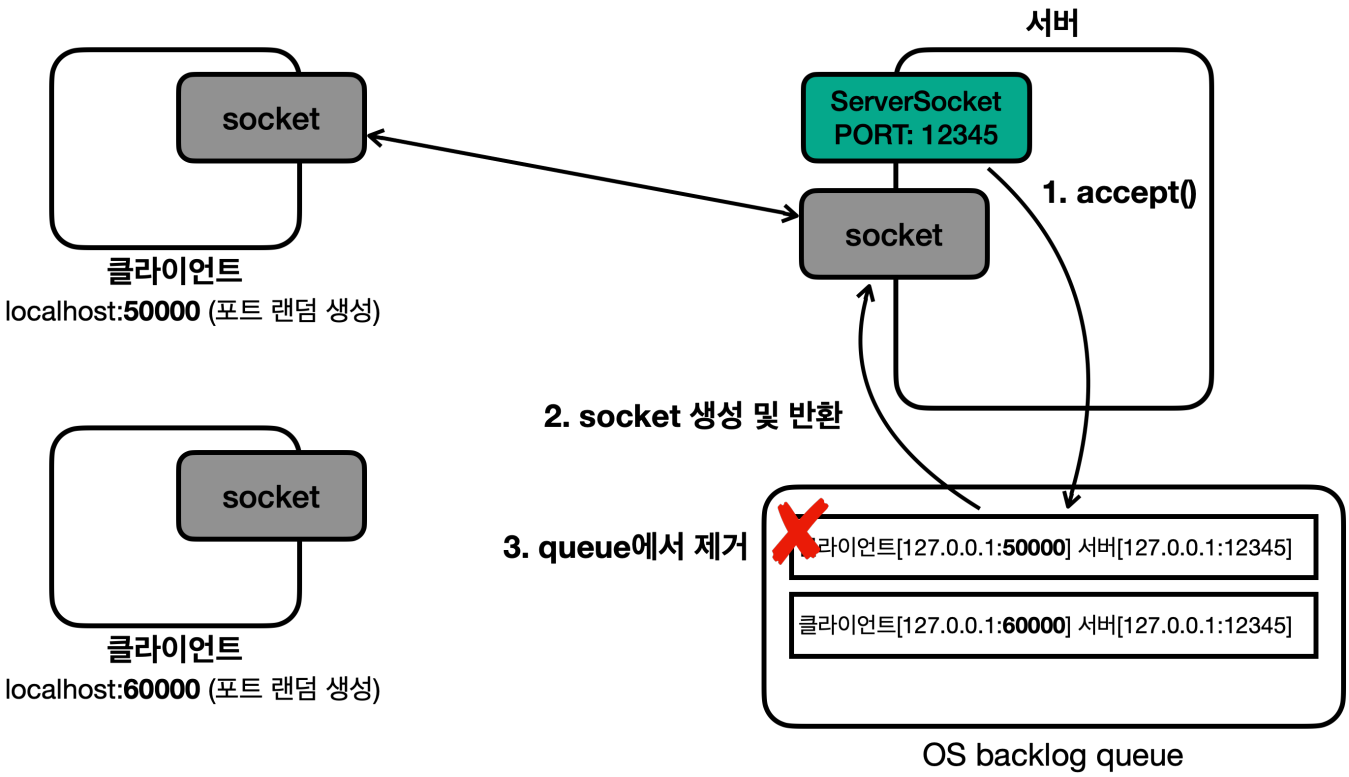


- 서버는 12345 서버 소켓을 열어둔다.
- 50000번 랜덤 포트를 사용하는 클라이언트가 먼저 12345 포트의 서버에 접속을 시도한다.
- 이때 OS 계층에서 TCP 3 way handshake가 발생하고, TCP 연결이 완료된다.
- TCP 연결이 완료되면 서버는 OS backlog queue라는 곳에 클라이언트와 서버의 TCP 연결 정보를 보관한다.

여기서 중요한 점이 있는데, 이 시점에 TCP 3 way handshake가 완료되었기 때문에, 클라이언트와 서버의 TCP 연결은 이미 완료되고, 클라이언트의 소켓 객체도 정상 생성된다. 참고로 이 시점에 아직 서버의 소켓 객체(서버 소켓 아님)는 생성되지 않았다.



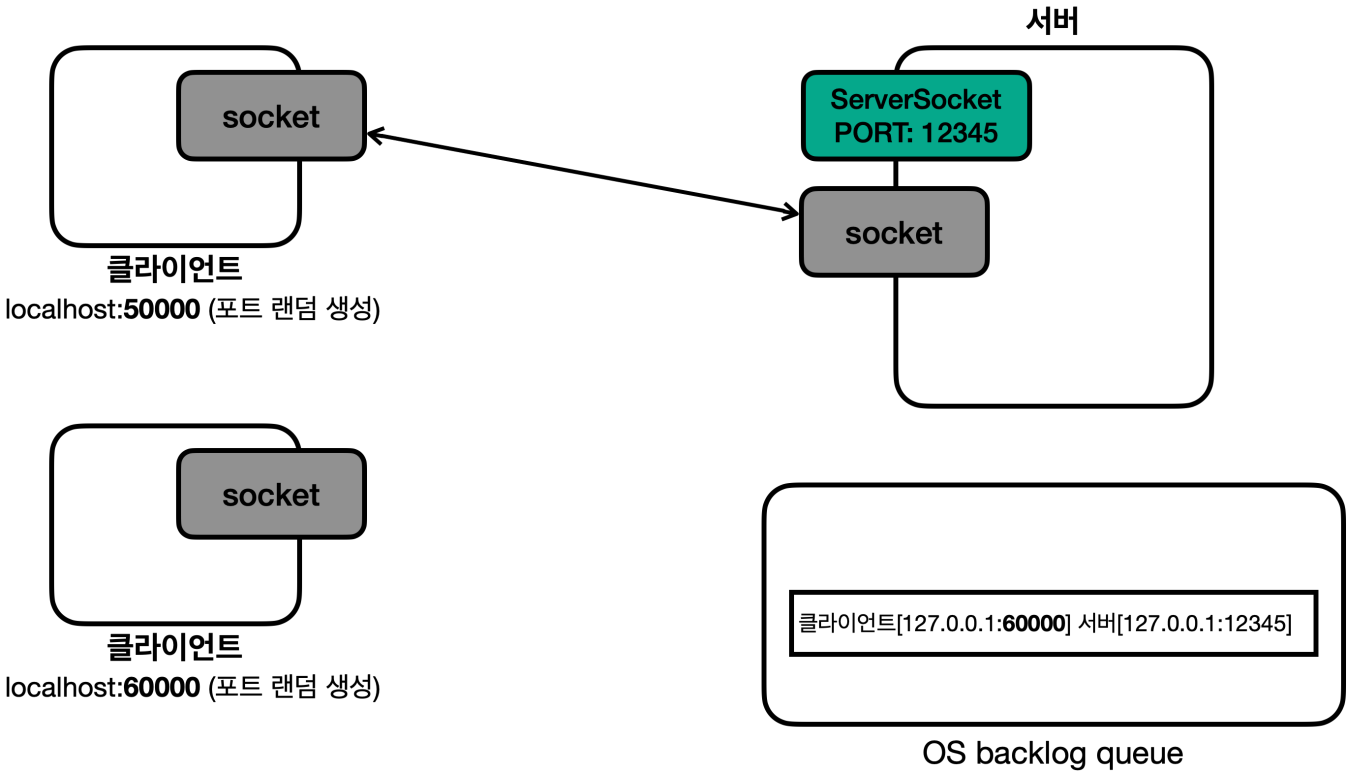
- 이번에는 60000번 랜덤 포트를 사용하는 클라이언트가 12345 포트의 서버에 접속을 시도하고 연결을 완료한다.
- 50000번 클라이언트와 60000번 클라이언트 모두 서버와 연결이 완료되었고, 클라이언트의 소켓도 정상 생성된다.



- 서버가 클라이언트와 데이터를 주고 받으려면 소켓을 획득해야 한다.
- `ServerSocket.accpet()` 메서드를 호출하면 backlog 큐의 정보를 기반으로 소켓 객체를 하나 생성한다.
- 큐이므로 순서대로 데이터를 꺼낸다. 처음 50000번 클라이언트의 접속 정보를 기반으로 서버에 소켓이 하나 생

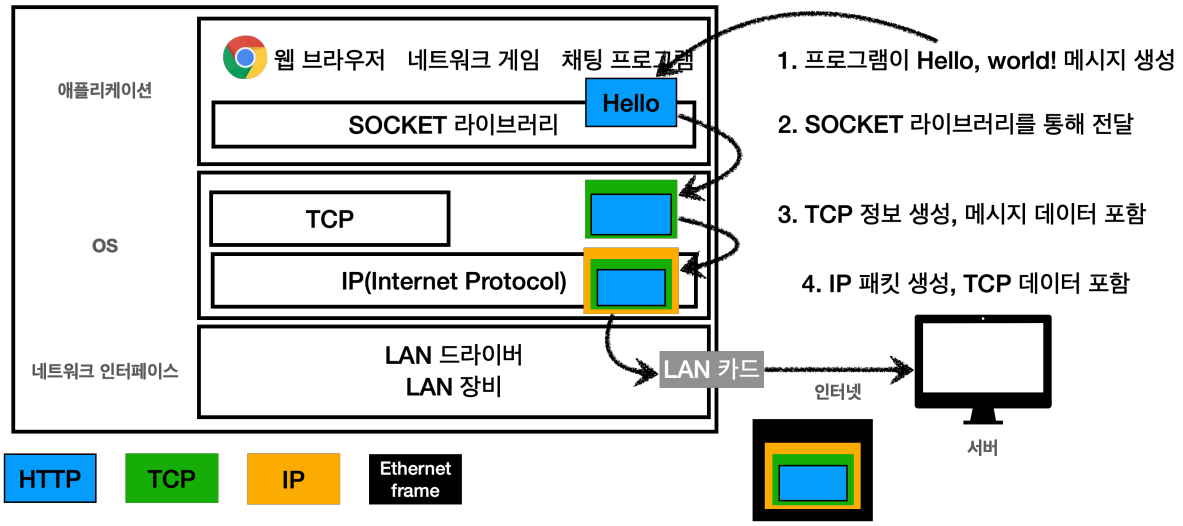
성된다.

- 50000번 클라이언트와 서버는 소켓의 스트림을 통해 서로 데이터를 주고 받을 수 있다.



- 그림에서 60000번 클라이언트도 이미 서버와 TCP 연결은 되어 있다.
  - OS 계층에서 TCP 3 way handshake가 발생하고, TCP 연결이 완료되었다.
- 60000번 클라이언트도 서버와 TCP 연결이 되었기 때문에 서버로 메시지를 보낼 수 있다.
  - 아직 서버에 Socket 객체가 없더라도 메시지는 보낼 수 있다. TCP 연결은 이미 완료되었다.

프로토콜 계층



그림을 보자. 소켓을 통해 스트림으로 메시지를 주고 받는다는 것은 사실은 이러한 과정을 거치는 것이다.
자바 애플리케이션은 소켓 객체의 스트림을 통해 서버와 데이터를 주고 받는다. 데이터를 주고 받는 과정은 다음과 같다.

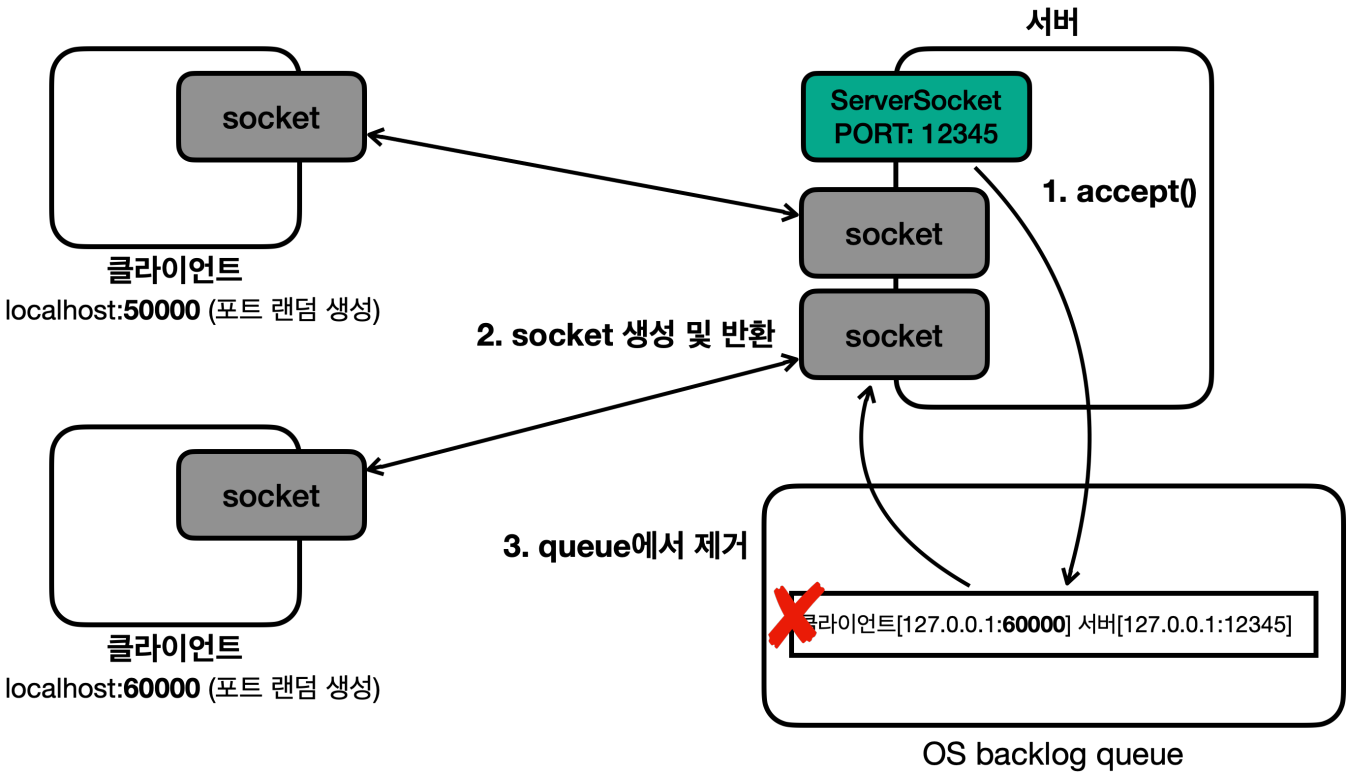
클라이언트가 "Hello, world!"라는 메시지를 서버에 전송하는 경우

- 클라이언트
  - 애플리케이션 ➔ **OS TCP 송신 버퍼** ➔ 클라이언트 네트워크 카드
- 클라이언트가 보낸 메시지가 서버에 도착했을 때, 서버
  - 서버 네트워크 카드 ➔ **OS TCP 수신 버퍼** ➔ 애플리케이션

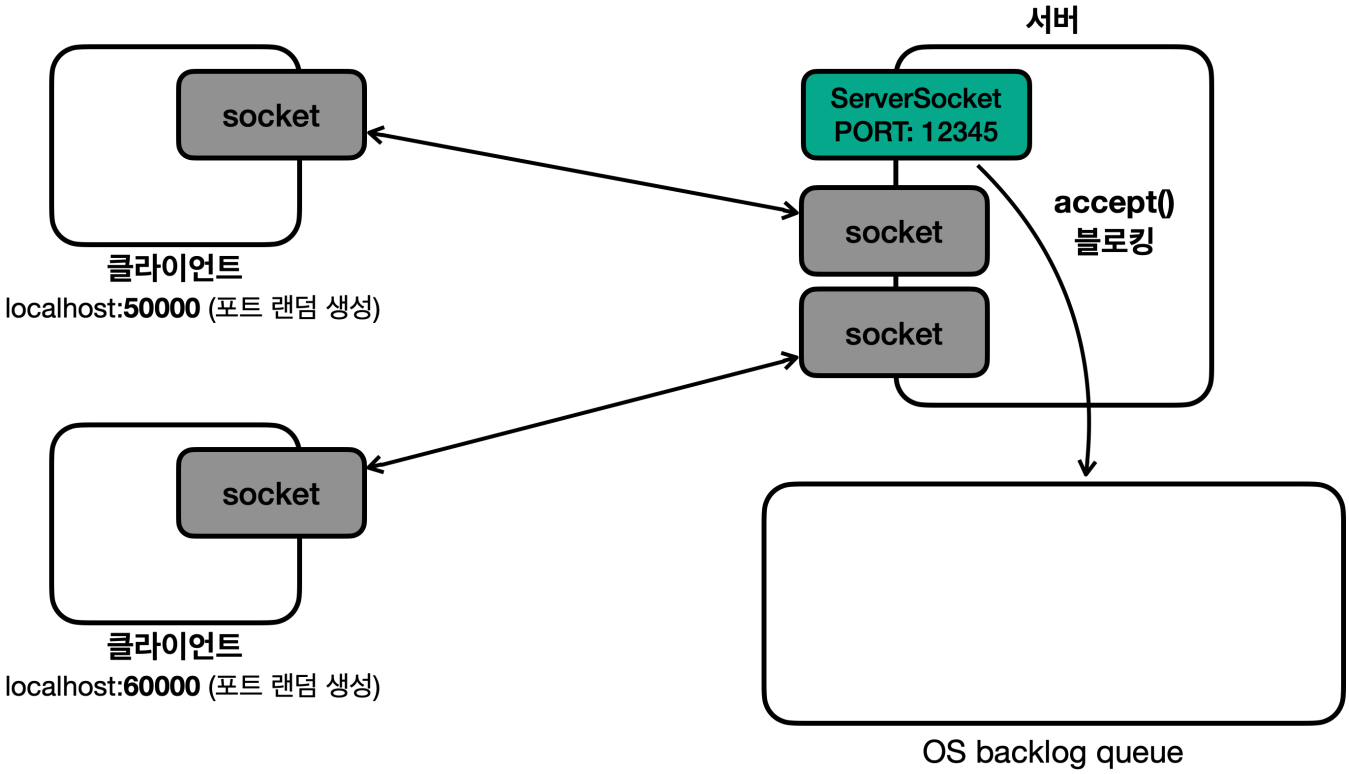
여기서 60000번 클라이언트가 보낸 메시지는 서버 애플리케이션에서 아직 읽지 않았기 때문에, **서버 OS의 TCP 수신 버퍼에서 대기**하게 된다.

여기서 핵심적인 내용이 있는데, 소켓 객체 없이 서버 소켓만으로도 TCP 연결은 완료된다는 점이다.
(서버 소켓은 연결만 담당한다)
하지만 연결 이후에 서로 메시지를 주고 받으려면 소켓 객체가 필요하다.

`accept()`는 이미 연결된 TCP 연결 정보를 기반으로 서버 측에 소켓 객체를 생성한다. 그리고 이 소켓 객체가 있어야 스트림을 사용해서 메시지를 주고 받을 수 있다.



- 이렇게 소켓을 연결하면 소켓의 스트림을 통해 OS TCP 수신 버퍼에 있는 메시지를 읽을 수 있고, 또 전송할 수도 있다.



- `accept()` 메서드는 backlog 큐에 새로운 연결 정보가 도착할 때 까지 블로킹 상태로 대기한다. 새로운 연결 정보가 오지 않으면 계속 대기하는 블로킹 메서드이다.

ServerV2의 문제

`ServerV2`에 둘 이상의 클라이언트가 작동하지 않는 이유는 다음과 같다.

- 새로운 클라이언트가 접속하면?
  - 새로운 클라이언트가 접속했을 때 서버의 `main` 스레드는 `accept()` 메서드를 절대로 호출할 수 없다! 왜냐하면 `while` 문으로 기존 클라이언트와 메시지를 주고 받는 부분만 반복하기 때문이다.
  - `accept()`를 호출해야 소켓 객체를 생성하고 클라이언트와 메시지를 주고 받을 수 있다.
- 2개의 블로킹 작업 - 핵심은 별도의 스레드가 필요하다!
  - `accept()` : 클라이언트와 서버의 연결을 처리하기 위해 대기
  - `readXxx()` : 클라이언트의 메시지를 받아서 처리하기 위해 대기
  - 각각의 블로킹 작업은 별도의 스레드에서 처리해야 한다. 그렇지 않으면 다른 블로킹 메서드 때문에 계속 대기할 수 있다.

ServerV2 코드 - 간략

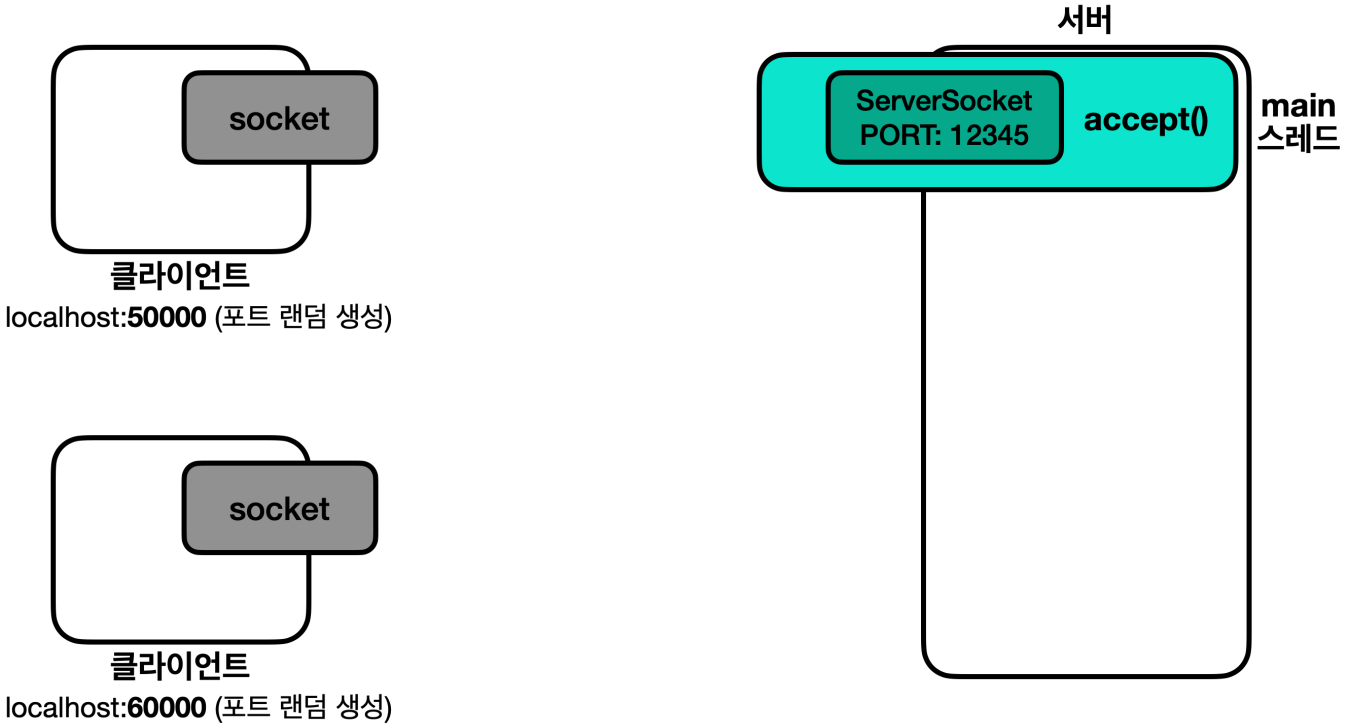
```
ServerSocket serverSocket = new ServerSocket(PORT);
Socket socket = serverSocket.accept(); // 블로킹

while(true) {
```

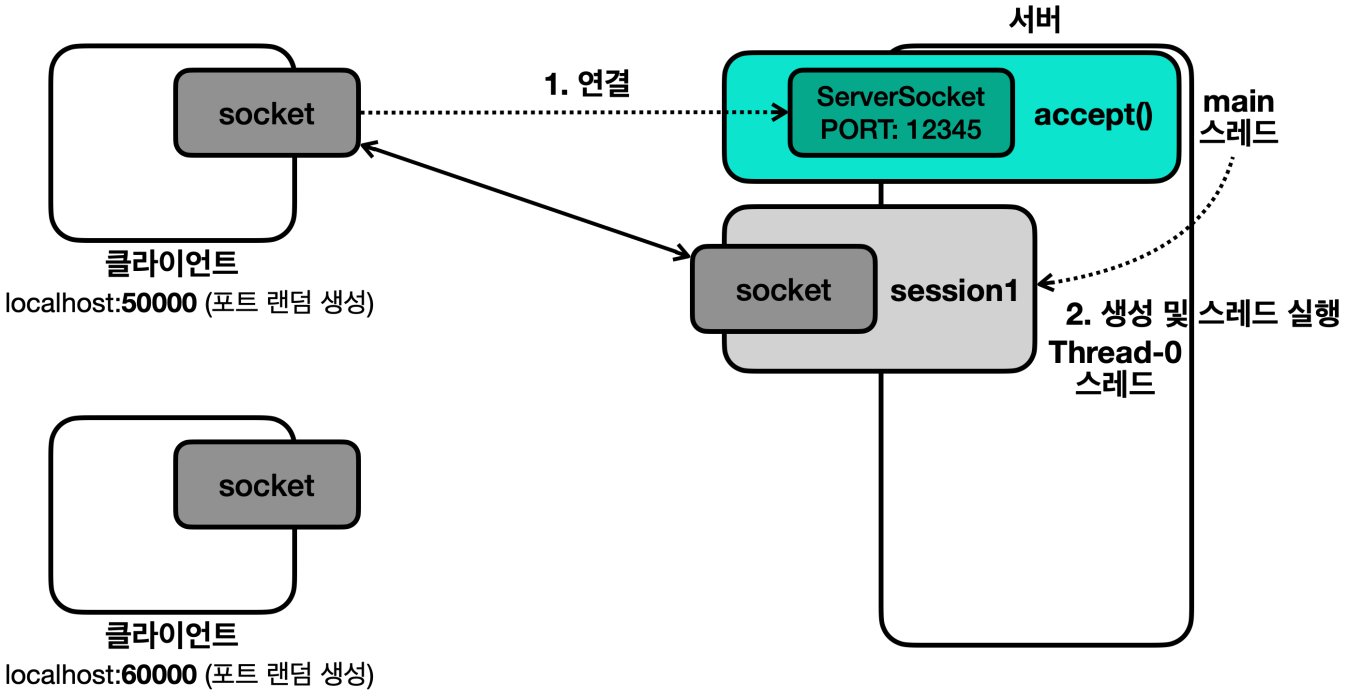
```
        String received = input.readUTF(); // 블로킹
        output.writeUTF(toSend);
    }
```

네트워크 프로그램3

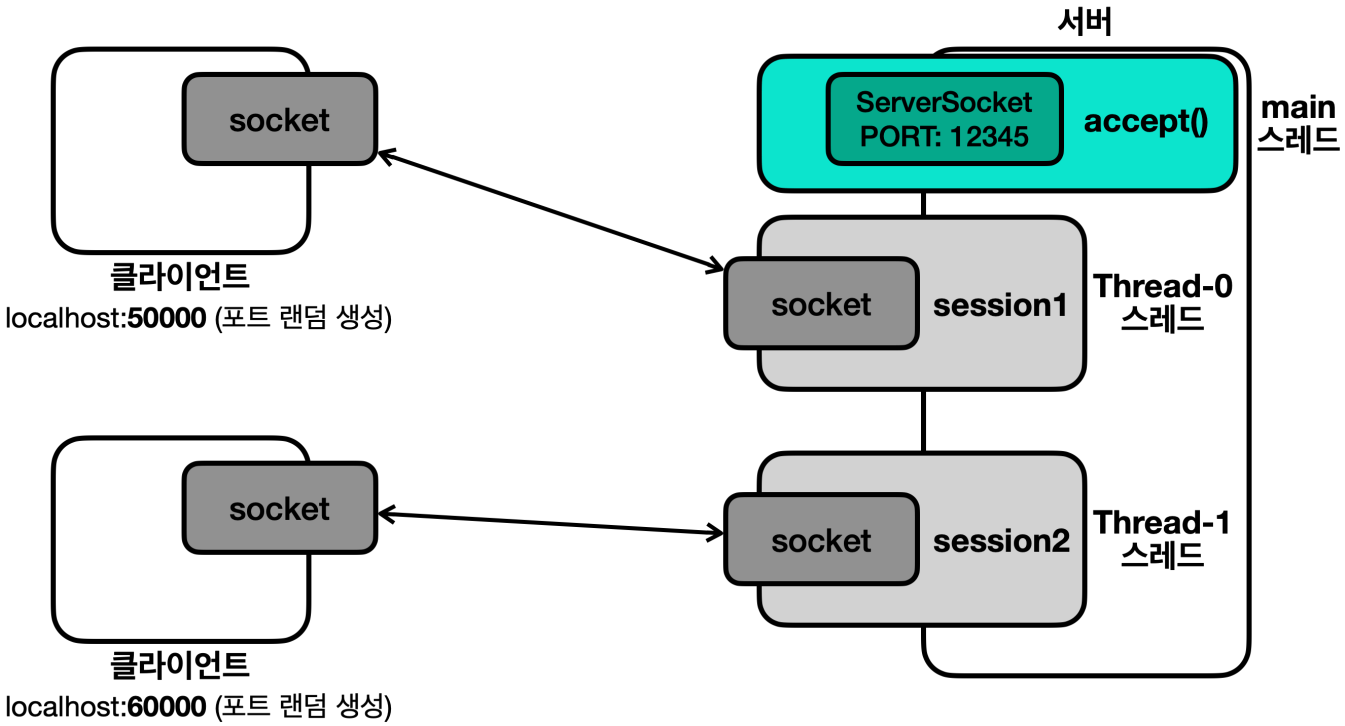
이번에는 여러 클라이언트가 동시에 접속할 수 있는 서버 프로그램을 작성해보자.



- 서버의 `main` 스레드는 서버 소켓을 생성하고, 서버 소켓의 `accept()`를 반복해서 호출해야 한다.



- 클라이언트가 서버에 접속하면 서버 소켓의 `accept()` 메서드가 `Socket`을 반환한다.
- `main` 스레드는 이 정보를 기반으로 `Runnable`을 구현한 `Session`이라는 별도의 객체를 만들고, 새로운 스레드에서 이 객체를 실행한다. 여기서는 `Thread-0`이 `Session`을 실행한다.
- `Session` 객체와 `Thread-0`은 연결된 클라이언트와 메시지를 주고 받는다.



- 새로운 TCP 연결이 발생하면 `main` 스레드는 새로운 `Session` 객체를 별도의 스레드에서 실행한다. 그리고 이 과정을 반복한다.
- `Session` 객체와 `Thread-1`은 연결된 클라이언트와 메시지를 주고 받는다.

역할의 분리

- **`main` 스레드**
  - `main` 스레드는 새로운 연결이 있을 때 마다 `Session` 객체와 별도의 스레드를 생성하고, 별도의 스레드가 `Session` 객체를 실행하도록 한다.
- **Session 담당 스레드**
  - `Session`을 담당하는 스레드는 자신의 소켓이 연결된 클라이언트와 메시지를 반복해서 주고 받는 역할을 담당한다.

```
package network.tcp.v3;

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 ClientV3 {

    public static final int PORT = 12345;

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

        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);
        }

        // 자원 정리
        log("연결 종료: " + socket);
        input.close();
        output.close();
        socket.close();
    }
}
```

- 클라이언트 코드는 기존 코드와 완전히 같다. 클래스 이름만 `ClientV2` ➜ `CilentV3` 로 변경했다.

```
package network.tcp.v3;

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

import static util.MyLogger.log;

public class SessionV3 implements Runnable {

    private final Socket socket;

    public SessionV3(Socket socket) {
        this.socket = socket;
    }

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

```
DataOutputStream(socket.getOutputStream());

            while(true) {
                // 클라이언트로부터 문자 받기
                String received = input.readUTF(); // 블로킹
                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
            log("연결 종료: " + socket);
            input.close();
            output.close();
            socket.close();
        } catch (IOException e) {
            throw new RuntimeException(e);
        }
    }
}
```

- `Session`의 목적은 소켓이 연결된 클라이언트와 메시지를 반복해서 주고 받는 것이다.
- 생성자를 통해 `Socket` 객체를 입력 받는다.
- `Runnable`을 구현해서 별도의 스레드에서 실행한다.

```
package network.tcp.v3;

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

import static util.MyLogger.log;
```

```
public class ServerV3 {

    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);

            SessionV3 session = new SessionV3(socket);
            Thread thread = new Thread(session);
            thread.start();
        }
    }

}
```

- main 코드는 `main` 스레드가 작동하는 부분이다.
- `main` 스레드는 서버 소켓을 생성하고, `serverSocket.accept()`을 호출해서 연결을 대기한다.
- 새로운 연결이 추가될 때 마다 `Session` 객체를 생성하고 별도의 스레드에서 `Session` 객체를 실행한다.
- 이 과정을 반복한다.

실행 결과 - ClientV3

```
15:46:08.605 [     main] 클라이언트 시작
15:46:08.611 [     main] 소캣 연결: Socket[addr=localhost/
127.0.0.1,port=12345,localport=53427]
전송 문자: hello
15:46:24.744 [     main] client -> server: hello
15:46:24.744 [     main] client <- server: hello World!
전송 문자: exit
15:46:36.380 [     main] client -> server: exit
15:46:36.381 [     main] 연결 종료: Socket[addr=localhost/
127.0.0.1,port=12345,localport=53427]
```

실행 결과 - ClientV3-2

```
15:46:23.286 [     main] 클라이언트 시작
15:46:23.292 [     main] 소캣 연결: Socket[addr=localhost/
127.0.0.1,port=12345,localport=53460]
전송 문자: hi
15:46:27.418 [     main] client -> server: hi
15:46:27.419 [     main] client <- server: hi World!
전송 문자: exit
15:46:34.100 [     main] client -> server: exit
15:46:34.101 [     main] 연결 종료: Socket[addr=localhost/
127.0.0.1,port=12345,localport=53460]
```

실행 결과 - ServerV3

```
15:46:06.414 [     main] 서버 시작
15:46:06.415 [     main] 서버 소켓 시작 - 리스닝 포트: 12345
15:46:08.611 [     main] 소켓 연결: Socket[addr=/
127.0.0.1,port=53427,localport=12345]
15:46:23.292 [     main] 소켓 연결: Socket[addr=/
127.0.0.1,port=53460,localport=12345]
15:46:24.744 [ Thread-0] client -> server: hello
15:46:24.744 [ Thread-0] client <- server: hello World!
15:46:27.418 [ Thread-1] client -> server: hi
15:46:27.418 [ Thread-1] client <- server: hi World!
15:46:34.101 [ Thread-1] client -> server: exit
15:46:34.101 [ Thread-1] 연결 종료: Socket[addr=/
127.0.0.1,port=53460,localport=12345]
15:46:36.380 [ Thread-0] client -> server: exit
15:46:36.381 [ Thread-0] 연결 종료: Socket[addr=/
127.0.0.1,port=53427,localport=12345]
```

여러 서버가 접속해도 문제없이 작동하는 것을 확인할 수 있다. 그리고 각각의 연결이 별도의 스레드에서 처리되는 것도 확인할 수 있다.

서버 소켓을 통해 소켓을 연결하는 부분과 각 클라이언트와 메시지를 주고 받는 부분이 별도의 스레드로 나뉘어 있다. 블로킹 되는 부분은 이렇게 별도의 스레드로 나누어 실행해야 한다.

문제

여기서 실행 중인 클라이언트를 인텔리J의 빨간색 Stop 버튼을 눌러서 직접 종료해보자.

클라이언트를 직접 종료한 경우 서버 로그

```
16:13:06.075 [     main] 서버 시작
16:13:06.076 [     main] 서버 소켓 시작 - 리스닝 포트: 12345
16:13:07.471 [     main] 소켓 연결: Socket[addr=/
127.0.0.1,port=57110,localport=12345]
Exception in thread "Thread-0" java.lang.RuntimeException:
java.io.EOFException
    at network.tcp.v3.SessionV3.run(SessionV3.java:45)
    at java.base/java.lang.Thread.run(Thread.java:1570)
Caused by: java.io.EOFException
    at java.base/java.io.DataInputStream.readFully(DataInputStream.java:210)
    at java.base/
java.io.DataInputStream.readUnsignedShort(DataInputStream.java:341)
    at java.base/java.io.DataInputStream.readUTF(DataInputStream.java:575)
    at java.base/java.io.DataInputStream.readUTF(DataInputStream.java:550)
    at network.tcp.v3.SessionV3.run(SessionV3.java:26)
    ... 1 more
```

클라이언트의 연결을 직접 종료하면 클라이언트 프로세스가 종료되면서, 클라이언트와 서버의 TCP 연결도 함께 종료된다. 이때 서버에서 `readUTF()`로 클라이언트가 메시지를 읽으려고 하면 `EOFException`이 발생한다. 소켓의 TCP 연결이 종료되었기 때문에 더는 읽을 수 있는 메시지가 없다는 뜻이다. EOF(파일의 끝)가 여기서는 전송의 끝이라는 뜻이다.

그런데 여기서 심각한 문제가 하나 있다. 이렇게 예외가 발생해버리면 서버에서 자원 정리 코드를 호출하지 못한다는 점이다.

서버 로그를 보면 연결 종료 로그가 없는 것을 확인할 수 있다.

윈도우 OS 내용 추가

클라이언트를 직접 종료한 경우 서버 로그 - 윈도우

```
Exception in thread "Thread-0" java.lang.RuntimeException:
java.net.SocketException: Connection reset
    at network.tcp.v3.SessionV3.run(SessionV3.java:45)
    at java.base/java.lang.Thread.run(Thread.java:1583)
Caused by: java.net.SocketException: Connection reset
    at java.base/sun.nio.ch.NioSocketImpl.implRead(NioSocketImpl.java:318)
    ... 1 more
```

참고로 윈도우의 경우 `java.io.EOFException`이 아니라 `java.net.SocketException: Connection reset`이 발생한다. 둘의 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.

```
// 자원 정리
log("연결 종료: " + socket);
input.close();
output.close();
socket.close();
```

자바 객체는 GC가 되지만 자바 외부의 자원은 자동으로 GC가 되는게 아니다. 따라서 꼭! 정리를 해주어야 한다.
(TCP 연결의 경우 운영체제가 어느정도 연결을 정리해주지만, 직접 연결을 종료할 때 보다 더 많은 시간이 걸릴 수 있다.)

이야기
첫 회사에 출근했는데, 1주일에 한 번씩 서버가 내려가는 이야기

클라이언트는 종료하고 다시 실행해도 되고, 컴퓨터를 자주 재부팅하기도 한다. 하지만 서버의 경우 프로세스가 계속 살아서 실행되어야 하기 때문에, 사용한 자원을 사용 후에는 즉각 정리해야 한다.
자원 정리는 서버 개발자에게 매우 중요한 내용이기 때문에, 이번에 한번 제대로 정리해보자.

자원 정리1

이번 내용은 자바 중급 1편에서 다룬 예외 처리의 실전 심화 내용이다.

자원 정리를 이해하기 위해 간단한 예제 코드를 만들어보자.

```
package network.tcp.autocloseable;
```

```
public class CallException extends Exception {

    public CallException(String message) {
        super(message);
    }
}
```

```
package network.tcp.autocloseable;

public class CloseException extends Exception {

    public CloseException(String message) {
        super(message);
    }
}
```

```
package network.tcp.autocloseable;

public class ResourceV1 {

    private String name;

    public ResourceV1(String name) {
        this.name = name;
    }

    public void call() {
        System.out.println(name + " call");
    }

    public void callEx() throws CallException {
        System.out.println(name + " callEx");
        throw new CallException(name + " ex");
    }

    public void close() {
        System.out.println(name + " close");
    }
```

```
    public void closeEx() throws CloseException {
        System.out.println(name + " closeEx");
        throw new CloseException(name + " ex");
    }
}
```

- `call()` : 정상 로직 호출
- `callEx()` : 비정상 로직 호출 `CallException`을 던진다.
- `close()` : 정상 종료
- `closeEx()` : 비정상 종료, `CloseException`을 던진다.

```
package network.tcp.autocloseable;

public class ResourceCloseMainV1 {

    public static void main(String[] args) {
        try {
            logic();
        } catch (CallException e) {
            System.out.println("CallException 예외 처리");
            e.printStackTrace();
        } catch (CloseException e) {
            System.out.println("CloseException 예외 처리");
            e.printStackTrace();
        }
    }

    private static void logic() throws CallException, CloseException {
        ResourceV1 resource1 = new ResourceV1("resource1");
        ResourceV1 resource2 = new ResourceV1("resource2");

        resource1.call();
        resource2.callEx(); // CallException;

        System.out.println("자원 정리"); // 호출 안됨
        resource2.closeEx();
        resource1.closeEx();
    }

}
```

- 서로 관련된 자원은 나중에 생성한 자원을 먼저 정리해야 한다.
- 예를 들어서 `resource1`을 생성하고, `resource1`의 정보를 활용해서 `resource2`를 생성한다면, 닫을 때는 그 반대인 `resource2`를 먼저 닫고, 그 다음에 `resource1`을 닫아야 한다. 왜냐하면 `resource2`의 입장에서 `resource1`의 정보를 아직 참고하고 있기 때문이다.
- 이 예제에서는 두 자원이 서로 관련이 없기 때문에 생성과 종료 순서가 크게 상관이 없지만, `resource1`의 정보를 기반으로 `resource2`를 생성한다고 가정하겠다.

실행 결과

```
resource1 call
resource2 callEx
CallException 예외 처리
network.tcp.autocloseable.CallException: resource2 ex
    at network.tcp.autocloseable.ResourceV1.callEx(ResourceV1.java:17)
    at
network.tcp.autocloseable.ResourceCloseMainV1.logic(ResourceCloseMainV1.java:2
6)
    at
network.tcp.autocloseable.ResourceCloseMainV1.main(ResourceCloseMainV1.java:11
)
```

`callEx()`를 호출하면서 예외가 발생했다. 예외 때문에 자원 정리 코드가 정상 호출되지 않았다.
이 코드는 예외가 발생하면 자원이 정리되지 않는다는 문제가 있다.

자원 정리2

이번에는 예외가 발생해도 자원을 정리하도록 해보자.

```
package network.tcp.autocloseable;

public class ResourceCloseMainV2 {

    public static void main(String[] args) {
        try {
            logic();
```

```
        } catch (CallException e) {
            System.out.println("CallException 예외 처리");
            e.printStackTrace();
        } catch (CloseException e) {
            System.out.println("CloseException 예외 처리");
            e.printStackTrace();
        }
    }

    private static void logic() throws CallException, CloseException {
        ResourceV1 resource1 = null;
        ResourceV1 resource2 = null;

        try {
            resource1 = new ResourceV1("resource1");
            resource2 = new ResourceV1("resource2");

            resource1.call();
            resource2.callEx(); // CallException

        } catch (CallException e) {
            System.out.println("ex: " + e);
            throw e; // CallException 다시 던짐
        } finally {
            if (resource2 != null) {
                resource2.closeEx(); // CloseException 발생!
            }
            if (resource1 != null) {
                resource1.closeEx(); // 이 코드 호출 안됨!
            }
        }
    }

}
```

실행 결과

```
resource1 call
resource2 callEx
ex: network.tcp.autocloseable.CallException: resource2 ex
resource2 closeEx
CloseException 예외 처리
```

```
network.tcp.autocloseable.CloseException: resource2 ex
    at network.tcp.autocloseable.ResourceV1.closeEx(ResourceV1.java:26)
    at
network.tcp.autocloseable.ResourceCloseMainV2.logic(ResourceCloseMainV2.java:4
3)
    at
network.tcp.autocloseable.ResourceCloseMainV2.main(ResourceCloseMainV2.java:17
)
```

null 체크

이번에는 `finally` 코드 블록을 사용해서 자원을 닫는 코드가 항상 호출되도록 했다.
만약 `resource2` 객체를 생성하기 전에 예외가 발생하면 `resource2`는 `null`이 된다. 따라서 `null` 체크를 해야 한다.
`resource1`의 경우에도 `resource1`을 생성하는 중에 예외가 발생한다면 `null` 체크가 필요하다.

자원 정리중에 예외가 발생하는 문제

`finally` 코드 블록은 항상 호출되기 때문에 자원이 잘 정리될 것 같지만, 이번에는 자원을 정리하는 중에 `finally` 코드 블록 안에서 `resource2.closeEx()`를 호출하면서 예외가 발생한다. 결과적으로 `resource1.closeEx()`는 호출되지 않는다.

핵심 예외가 바뀌는 문제

이 코드에서 발생한 핵심적인 예외는 `CallException`이다. 이 예외 때문에 문제가 된 것이다. 그런데 `finally` 코드 블록에서 자원을 정리하면서 `CloseException` 예외가 추가로 발생했다. 예외 때문에 자원을 정리하고 있는데, 자원 정리중에 또 예외가 발생한 것이다. 이 경우 `logic()`을 호출한 쪽에서는 핵심 예외인 `CallException`이 아니라 `finally` 블록에서 새로 생성된 `CloseException`을 받게 된다. 핵심 예외가 사라진 것이다!
개발자가 원하는 예외는 당연히 핵심 예외다. 이 핵심 예외를 확인해야 제대로 된 문제를 찾을 수 있다. 자원을 닫는 중에 발생한 예외는 부가 예외일 뿐이다.

정리하면 이 코드는 다음과 같은 문제가 있다.

- `close()` 시점에 실수로 예외를 던지면, 이후 다른 자원을 닫을 수 없는 문제 발생
- `finally` 블럭 안에서 자원을 닫을 때 예외가 발생하면, 핵심 예외가 `finally`에서 발생한 부가 예외로 바뀌어 버린다. 그리고 핵심 예외가 사라진다.

자원 정리3

이번에는 자원 정리의 코드에서 try-catch를 사용해서 자원 정리 중에 발생하는 예외를 잡아서 처리해보자.

```
package network.tcp.autocloseable;

public class ResourceCloseMainV3 {

    public static void main(String[] args) {
        try {
            logic();
        } catch (CallException e) {
            System.out.println("CallException 예외 처리");
            e.printStackTrace();
        } catch (CloseException e) {
            System.out.println("CloseException 예외 처리");
            e.printStackTrace();
        }
    }

    private static void logic() throws CallException, CloseException {
        ResourceV1 resource1 = null;
        ResourceV1 resource2 = null;

        try {
            resource1 = new ResourceV1("resource1");
            resource2 = new ResourceV1("resource2");

            resource1.call();
            resource2.callEx(); // CallException
        } catch (CallException e) {
            System.out.println("ex: " + e);
            throw e;
        } finally {
            if (resource2 != null) {
                try {
                    resource2.closeEx();
                } catch (CloseException e) {
                    // close()에서 발생한 예외는 버린다. 필요하면 로깅 정도
                    System.out.println("close ex: " + e);
                }
            }
            if (resource1 != null) {
```

```
                try {
                    resource1.closeEx();
                } catch (CloseException e) {
                    System.out.println("close ex: " + e);
                }
            }
        }
    }

}
```

- `finally` 블럭에서 각각의 자원을 닫을 때도, 예외가 발생하면 예외를 잡아서 처리하도록 했다.
- 이렇게 하면 자원 정리 시점에 예외가 발생해도, 다음 자원을 닫을 수 있다.
- 자원 정리 시점에 발생한 예외를 잡아서 처리했기 때문에, 자원 정리 시점에 발생한 부가 예외가 핵심 예외를 가리지 않는다.
- 자원 정리 시점에 발생한 예외는 당장 더 처리할 수 있는 부분이 없다. 이 경우 로그를 남겨서 개발자가 인지할 수 있게 하는 정도면 충분하다.

실행 결과

```
resource1 call
resource2 callEx
ex: network.tcp.autocloseable.CallException: resource2 ex
resource2 closeEx
close ex: network.tcp.autocloseable.CloseException: resource2 ex
resource1 closeEx
close ex: network.tcp.autocloseable.CloseException: resource1 ex
CallException 예외 처리
network.tcp.autocloseable.CallException: resource2 ex
    at network.tcp.autocloseable.ResourceV1.callEx(ResourceV1.java:17)
    at
network.tcp.autocloseable.ResourceCloseMainV3.logic(ResourceCloseMainV3.java:3
6)
    at
network.tcp.autocloseable.ResourceCloseMainV3.main(ResourceCloseMainV3.java:17
)
```

이전에 발생했던 다음 2가지 문제를 해결했다.

- `close()` 시점에 실수로 예외를 던지면, 이후 다른 자원을 닫을 수 없는 문제 발생
- `finally` 블럭 안에서 자원을 닫을 때 예외가 발생하면, 핵심 예외가 `finally`에서 발생한 부가 예외로 바뀌

어 버린다. 그리고 핵심 예외가 사라진다.

핵심적인 문제들은 해결되었지만 코드 부분에서 보면 아쉬운 부분이 많다.

- `resource` 변수를 선언하면서 동시에 할당할 수 없음(`try`, `finally` 코드 블록과 변수 스코프가 다른 문제)
- `catch` 이후에 `finally` 호출, 자원 정리가 조금 늦어진다.
- 개발자가 실수로 `close()`를 호출하지 않을 가능성
- 개발자가 `close()` 호출 순서를 실수, 보통 자원을 생성한 순서와 반대로 닫아야 함

지금까지 수 많은 자바 개발자들이 자원 정리 때문에 고통 받아왔다.
이런 문제를 한번에 해결하는 것이 바로 자바 중급1편에서 학습한 `try-with-resources` 구문이다.

자원 정리4

자바 중급 1편에서도 `try-with-resources`를 학습했지만, 이번에는 `try-with-resources`에 대해서 조금 더 깊이있게 들어가보자.

```
package network.tcp.autocloseable;

public class ResourceV2 implements AutoCloseable {

    private String name;

    public ResourceV2(String name) {
        this.name = name;
    }

    public void call() {
        System.out.println(name + " call");
    }

    public void callEx() throws CallException {
        System.out.println(name + " callEx");
        throw new CallException(name + " ex");
    }

    @Override
```

```
    public void close() throws CloseException {
        System.out.println(name + " close");
        throw new CloseException(name + " ex");
    }
}
```

- `AutoCloseable`을 구현했다.
- `close()`는 항상 `CloseException`을 던지도록 했다.

```
package network.tcp.autocloseable;

public class ResourceCloseMainV4 {

    public static void main(String[] args) {
        try {
            logic();
        } catch (CallException e) {
            System.out.println("CallException 예외 처리");
            Throwable[] suppressed = e.getSuppressed();
            for (Throwable throwable : suppressed) {
                System.out.println("suppressedEx = " + throwable);
            }
            e.printStackTrace();
        } catch (CloseException e) {
            System.out.println("CloseException 예외 처리");
            e.printStackTrace();
        }
    }

    private static void logic() throws CallException, CloseException {
        try (ResourceV2 resource1 = new ResourceV2("resource1");
             ResourceV2 resource2 = new ResourceV2("resource2")) {

            resource1.call();
            resource2.callEx(); // CallException;
        } catch (CallException e) {
            System.out.println("ex: " + e);
            throw e; // CallException;
        }
    }

}
```

실행 결과

```
resource1 call
resource2 callEx
resource2 close
resource1 close
ex: network.tcp.autocloseable.CallException: resource2 ex
CallException 예외 처리
suppressedEx = network.tcp.autocloseable.CloseException: resource2 ex
suppressedEx = network.tcp.autocloseable.CloseException: resource1 ex
network.tcp.autocloseable.CallException: resource2 ex
    at network.tcp.autocloseable.ResourceV2.callEx(ResourceV2.java:17)
    at
network.tcp.autocloseable.ResourceCloseMainV4.logic(ResourceCloseMainV4.java:2
6)
    at
network.tcp.autocloseable.ResourceCloseMainV4.main(ResourceCloseMainV4.java:7)
    Suppressed: network.tcp.autocloseable.CloseException: resource2 ex
        at network.tcp.autocloseable.ResourceV2.close(ResourceV2.java:23)
        at
network.tcp.autocloseable.ResourceCloseMainV4.logic(ResourceCloseMainV4.java:2
2)
        ... 1 more
    Suppressed: network.tcp.autocloseable.CloseException: resource1 ex
        at network.tcp.autocloseable.ResourceV2.close(ResourceV2.java:23)
        at
network.tcp.autocloseable.ResourceCloseMainV4.logic(ResourceCloseMainV4.java:2
2)
        ... 1 more
```

`try-with-resources`는 단순하게 `close()`를 자동 호출해준다는 정도의 기능한 제공하는 것이 아니다.
고민한 6가지 문제를 모두 해결하는 장치이다.

2가지 핵심 문제

- `close()` 시점에 실수로 예외를 던지면, 이후 다른 자원을 닫을 수 없는 문제 발생
- `finally` 블럭 안에서 자원을 닫을 때 예외가 발생하면, 핵심 예외가 `finally`에서 발생한 부가 예외로 바뀌어 버린다. 그리고 핵심 예외가 사라진다.

4가지 부가 문제

- `resource` 변수를 선언하면서 동시에 할당할 수 없음(`try`, `finally` 코드 블록과 변수 스코프가 다른 문제)
- `catch` 이후에 `finally` 호출, 자원 정리가 조금 늦어진다.
- 개발자가 실수로 `close()`를 호출하지 않을 가능성
- 개발자가 `close()` 호출 순서를 실수, 보통 자원을 생성한 순서와 반대로 닫아야 함

Try with resources 장점

- **리소스 누수 방지**: 모든 리소스가 제대로 닫히도록 보장한다. 실수로 `finally` 블록을 적지 않거나, `finally` 블럭 안에서 자원 해제 코드를 누락하는 문제들을 예방할 수 있다.
- **코드 간결성 및 가독성 향상**: 명시적인 `close()` 호출이 필요 없어 코드가 더 간결하고 읽기 쉬워진다.
- **스코프 범위 한정**: 예를 들어 리소스로 사용되는 `resource1,2` 변수의 스코프가 `try` 블럭 안으로 한정된다. 따라서 코드 유지보수가 더 쉬워진다.
- **조금 더 빠른 자원 해제**: 기존에는 try ➜ catch ➜ finally로 catch 이후에 자원을 반납했다. Try with resources 구분은 `try` 블럭이 끝나면 즉시 `close()`를 호출한다.
- **자원 정리 순서**: 먼저 선언한 자원을 나중에 정리한다.
- **부가 예외 포함**: 다음 내용에서 설명

Try with resources 예외 처리와 부가 예외 포함

`try-with-resources`를 사용하는 중에 핵심 로직 예외와 자원을 정리하는 중에 발생하는 부가 예외가 모두 발생하면 어떻게 될까?

- `try-with-resources`는 핵심 예외를 반환한다.
- 부가 예외는 핵심 예외안에 `Suppressed`로 담아서 반환한다.
- 개발자는 자원 정리 중에 발생한 부가 예외를 `e.getSuppressed()`를 통해 활용할 수 있다.

`try-with-resources`를 사용하면 핵심 예외를 반환하면서, 동시에 부가 예외도 필요하면 확인할 수 있다.

참고로 자바 예외에는 `e.addSuppressed(ex)`라는 메서드가 있어서 예외 안에 참고할 예외를 담아둘 수 있다. 참고로 이 기능도 `try-with-resources`와 함께 등장했다.