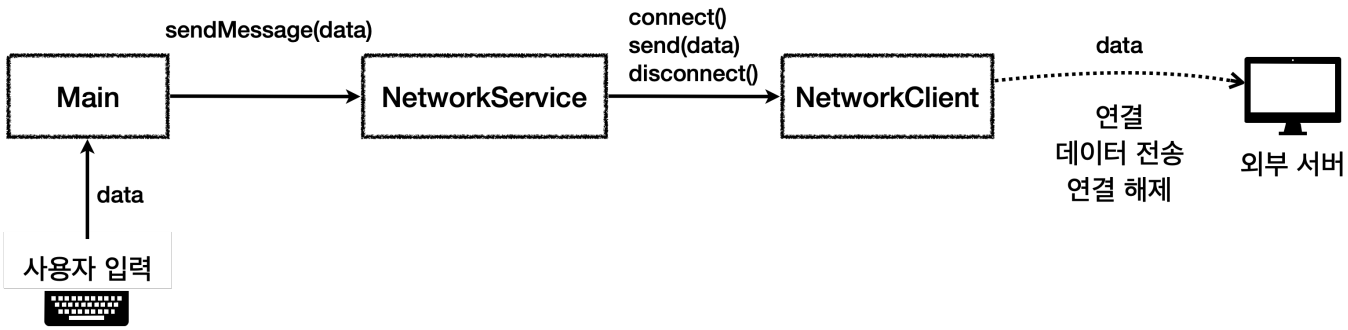
10. 예외 처리2 - 실습

#0.강의/1.자바로드맵/3.자바-중급1편

- /예외 처리 도입1 - 시작
- /예외 처리 도입2 - 예외 복구
- /예외 처리 도입3 - 정상, 예외 흐름 분리
- /예외 처리 도입4 - 리소스 반환 문제
- /예외 처리 도입5 - finally
- /예외 계층1 - 시작
- /예외 계층2 - 활용
- /실무 예외 처리 방안1 - 설명
- /실무 예외 처리 방안2 - 구현
- /try-with-resources
- /정리

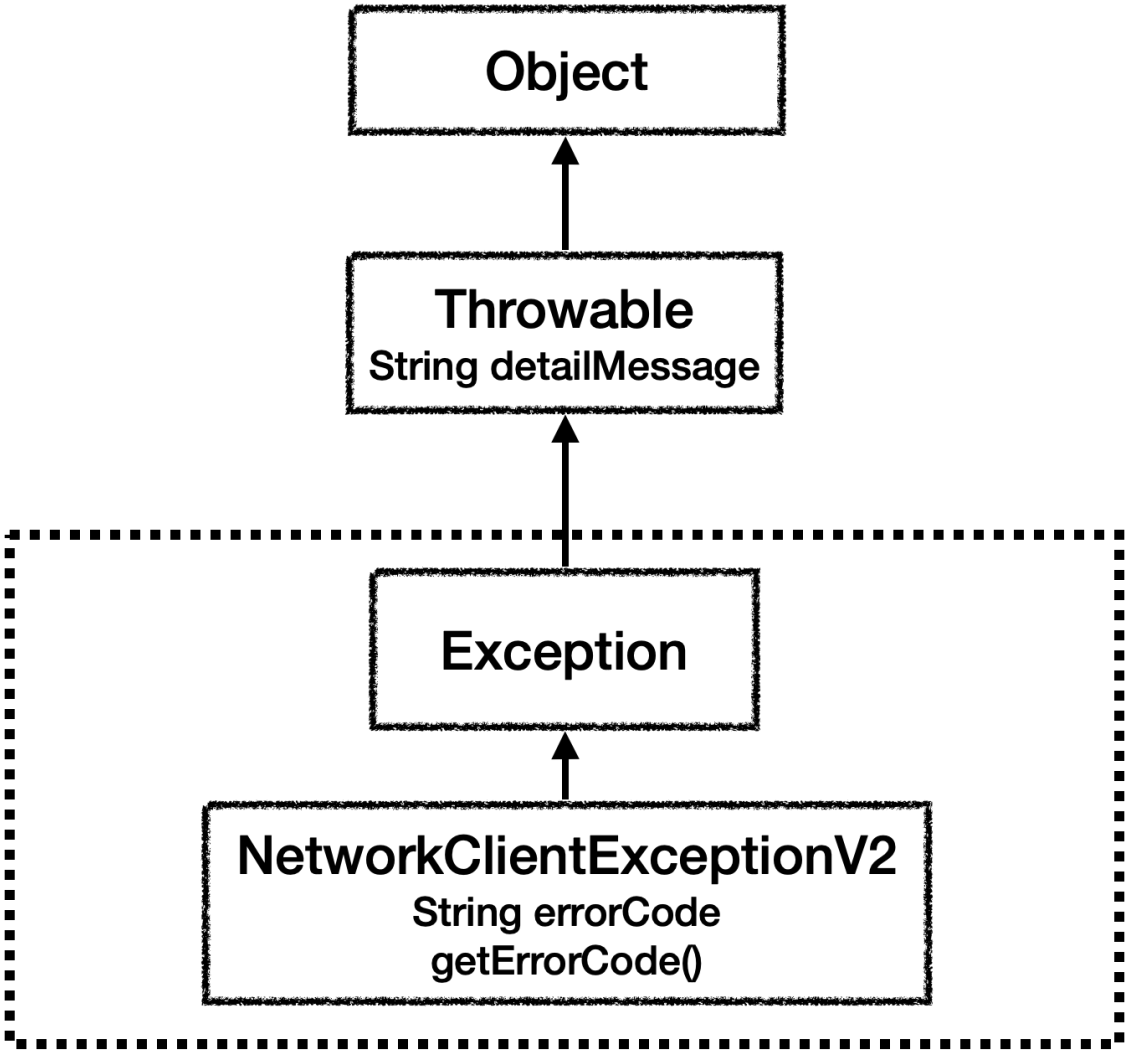
예외 처리 도입1 - 시작



앞서 만든 프로그램은 반환 값을 사용해서 예외를 처리했다. 이런 경우 다음과 같은 문제가 있었다.

- 정상 흐름과 예외 흐름이 섞여 있기 때문에 코드를 한눈에 이해하기 어렵다. 쉽게 이야기해서 가장 중요한 정상 흐름이 한눈에 들어오지 않는다.
- 심지어 예외 흐름이 더 많은 코드 분량을 차지한다. 실무에서는 예외 처리가 훨씬 더 복잡하다.

우리가 처음 만들었던 프로그램에 자바 예외 처리를 도입해서 이 문제를 점진적으로 해결해보자.



체크 예외

```
package exception.ex2;

public class NetworkClientExceptionV2 extends Exception {

    private String errorCode;

    public NetworkClientExceptionV2(String errorCode, String message) {
        super(message);
        this.errorCode = errorCode;
    }

    public String getErrorCode() {
        return errorCode;
    }
}
```

- 예외도 객체이다. 따라서 필요한 필드와 메서드를 가질 수 있다.
- **오류 코드**
  - 이전에는 오류 코드(`errorCode`)를 반환 값으로 리턴해서, 어떤 오류가 발생했는지 구분했다.
  - 여기서는 어떤 종류의 오류가 발생했는지 구분하기 위해 예외 안에 오류 코드를 보관한다.
- **오류 메시지**
  - 오류 메시지(`message`)에는 어떤 오류가 발생했는지 개발자가 보고 이해할 수 있는 설명을 담아둔다.
  - 오류 메시지는 상위 클래스인 `Throwable`에서 기본으로 제공하는 기능을 사용한다.

```
package exception.ex2;

public class NetworkClientV2 {

    private final String address;
    public boolean connectError;
    public boolean sendError;

    public NetworkClientV2(String address) {
        this.address = address;
    }

    public void connect() throws NetworkClientExceptionV2 {
        if (connectError) {
            throw new NetworkClientExceptionV2("connectError", address + " 서버
연결 실패");
        }
        //연결 성공
        System.out.println(address + " 서버 연결 성공");
    }

    public void send(String data) throws NetworkClientExceptionV2 {
        if (sendError) {
            throw new NetworkClientExceptionV2("sendError", address + " 서버에 데
이터 전송 실패: " + data);
        }
        //전송 성공
        System.out.println(address + " 서버에 데이터 전송: " + data);
    }

    public void disconnect() {
        System.out.println(address + " 서버 연결 해제");
```

```
    }

    public void initError(String data) {
        if (data.contains("error1")) {
            connectError = true;
        }
        if (data.contains("error2")) {
            sendError = true;
        }
    }

}
```

- 기존의 코드와 대부분 같지만, 오류가 발생했을 때 오류 코드를 반환하는 것이 아니라 예외를 던진다.
- 따라서 반환 값을 사용하지 않아도 된다. 여기서는 반환 값을 `void`로 처리한다.
- 이전에는 반환 값으로 성공, 실패 여부를 확인해야 했지만, 예외 처리 덕분에 메서드가 정상 종료되면 성공이고, 예외가 던져지면 예외를 통해 실패를 확인할 수 있다.
- 오류가 발생하면, 예외 객체를 만들고 거기에 오류 코드와 오류 메시지를 담아둔다. 그리고 만든 예외 객체를 `throw`를 통해 던진다.

```
package exception.ex2;

public class NetworkServiceV2_1 {

    public void sendMessage(String data) throws NetworkClientExceptionV2 {
        String address = "http://example.com";

        NetworkClientV2 client = new NetworkClientV2(address);
        client.initError(data);

        client.connect();
        client.send(data);
        client.disconnect();
    }
}
```

- 여기서는 예외를 별도로 처리하지 않고, `throws`를 통해 밖으로 던진다.

```
package exception.ex2;
```

```
import java.util.Scanner;

public class MainV2 {

    public static void main(String[] args) throws NetworkClientExceptionV2 {
        NetworkServiceV2_1 networkService = new NetworkServiceV2_1();

        Scanner scanner = new Scanner(System.in);
        while (true) {
            System.out.print("전송할 문자: ");
            String input = scanner.nextLine();
            if (input.equals("exit")) {
                break;
            }
            networkService.sendMessage(input);
            System.out.println();
        }
        System.out.println("프로그램을 정상 종료합니다.");
    }
}
```

- 여기서도 예외를 처리하지 않고, `throws`를 통해 밖으로 던진다.

이렇게 작성한 프로그램을 실행해보자.

실행 결과

```
전송할 문자: hello
http://example.com 서버 연결 성공
http://example.com 서버에 데이터 전송: hello
http://example.com 서버 연결 해제

전송할 문자: exit
프로그램을 정상 종료합니다.
```

```
전송할 문자: error1
Exception in thread "main" exception.ex2.NetworkClientExceptionV2: http://
example.com 서버 연결 실패
    at exception.ex2.NetworkClientV2.connect(NetworkClientV2.java:15)
```

```
    at
exception.ex2.NetworkServiceV2_1.sendMessage(NetworkServiceV2_1.java:11)
    at exception.ex2.MainV2.main(MainV2.java:22)
```

- `error1` 이면 연결 실패가 발생한다.
- 모든 곳에서 발생한 예외를 잡지 않았기 때문에 결과적으로 `main()` 밖으로 예외가 던져진다.
- `main()` 밖으로 예외가 던져지면 예외 메시지와 예외를 추적할 수 있는 스택 트레이스를 출력하고 프로그램을 종료한다.

```
전송할 문자: error2
http://example.com 서버 연결 성공
Exception in thread "main" exception.ex2.NetworkClientExceptionV2: http://
example.com 서버에 데이터 전송 실패: error2
    at exception.ex2.NetworkClientV2.send(NetworkClientV2.java:23)
    at
exception.ex2.NetworkServiceV2_1.sendMessage(NetworkServiceV2_1.java:12)
    at exception.ex2.MainV2.main(MainV2.java:22)
```

- `error2` 이면 데이터 전송 실패이다.
- 모든 곳에서 발생한 예외를 잡지 않았기 때문에 결과적으로 `main()` 밖으로 예외가 던져진다.
- `main()` 밖으로 예외가 던져지면 예외 메시지와 예외를 추적할 수 있는 스택 트레이스를 출력하고 프로그램을 종료한다.

남은 문제

- 예외 처리를 도입했지만, 아직 예외가 복구되지 않는다. 따라서 예외가 발생하면 발생하면 프로그램이 종료된다.
- 사용 후에는 반드시 `disconnect()` 를 호출해서 연결을 해제해야 한다.

예외 처리 도입2 - 예외 복구

이번에는 예외를 잡아서 예외 흐름을 정상 흐름으로 복구해보자.

```
package exception.ex2;
```

```
public class NetworkServiceV2_2 {

    public void sendMessage(String data) {
        String address = "http://example.com";

        NetworkClientV2 client = new NetworkClientV2(address);
        client.initError(data);

        try {
            client.connect();
        } catch (NetworkClientExceptionV2 e) {
            System.out.println("[오류] 코드: " + e.getErrorCode() + ", 메시지: " +
e.getMessage());
            return;
        }

        try {
            client.send(data);
        } catch (NetworkClientExceptionV2 e) {
            System.out.println("[오류] 코드: " + e.getErrorCode() + ", 메시지: " +
e.getMessage());
            return;
        }

        client.disconnect();
    }
}
```

- `connect()`, `send()`와 같이 예외가 발생할 수 있는 곳을 `try ~ catch`를 사용해서 `NetworkClientExceptionV2` 예외를 잡았다.
- 여기서는 예외를 잡으면 오류 코드와 예외 메시지를 출력한다.
- 예외를 잡아서 처리했기 때문에 이후에는 정상 흐름으로 복귀한다. 여기서는 리턴을 사용해서 `sendMessage()` 메서드를 정상적으로 빠져나간다.

MainV2 - 코드 변경

```
public static void main(String[] args) throws NetworkClientExceptionV2 {
    //NetworkServiceV2_1 networkService = new NetworkServiceV2_1();
    NetworkServiceV2_2 networkService = new NetworkServiceV2_2();
    ...
}
```

실행 결과

```
전송할 문자: hello
http://example.com 서버 연결 성공
http://example.com 서버에 데이터 전송: hello
http://example.com 서버 연결 해제

전송할 문자: error1
[오류] 코드: connectError, 메시지: http://example.com 서버 연결 실패

전송할 문자: error2
http://example.com 서버 연결 성공
[오류] 코드: sendError, 메시지: http://example.com 서버에 데이터 전송 실패: error2

전송할 문자: exit
프로그램을 정상 종료합니다.
```

해결된 문제

- 예외를 잡아서 처리했다. 따라서 예외가 복구 되고, 프로그램도 계속 수행할 수 있다.

남은 문제

- 예외 처리를 했지만 정상 흐름과 예외 흐름이 섞여 있어서 코드를 읽기 어렵다.
- 사용 후에는 반드시 `disconnect()`를 호출해서 연결을 해제해야 한다.

예외 처리 도입3 - 정상, 예외 흐름 분리

이번에는 예외 처리의 `try ~ catch` 기능을 제대로 사용해서 정상 흐름과 예외 흐름이 섞여 있는 문제를 해결해보자.

```
package exception.ex2;

public class NetworkServiceV2_3 {
```

```
    public void sendMessage(String data) {
        String address = "http://example.com";

        NetworkClientV2 client = new NetworkClientV2(address);
        client.initError(data);

        try {
            client.connect();
            client.send(data);
            client.disconnect(); //예외 발생시 무시
        } catch (NetworkClientExceptionV2 e) {
            System.out.println("[오류] 코드: " + e.getErrorCode() + ", 메시지: " +
e.getMessage());
        }

    }

}
```

- 하나의 `try` 안에 정상 흐름을 모두 담는다.
- 그리고 예외 부분은 `catch` 블럭에서 해결한다.
- 이렇게 하면 정상 흐름은 `try` 블럭에 들어가고, 예외 흐름은 `catch` 블럭으로 명확하게 분리할 수 있다.

MainV2 - 코드 변경

```
public static void main(String[] args) throws NetworkClientExceptionV2 {
    NetworkServiceV2_3 networkService = new NetworkServiceV2_3();
    ...
}
```

실행 결과

```
전송할 문자: hello
http://example.com 서버 연결 성공
http://example.com 서버에 데이터 전송: hello
http://example.com 서버 연결 해제

전송할 문자: error1
[오류] 코드: connectError, 메시지: http://example.com 서버 연결 실패

전송할 문자: error2
```

```
http://example.com 서버 연결 성공
[오류] 코드: sendError, 메시지: http://example.com 서버에 데이터 전송 실패: error2

전송할 문자: exit
프로그램을 정상 종료합니다.
```

해결된 문제

- 자바의 예외 처리 메커니즘과 `try`, `catch` 구조 덕분에 정상 흐름은 `try` 블럭에 모아서 처리하고, 예외 흐름은 `catch` 블럭에 별도로 모아서 처리할 수 있었다.
- 덕분에 정상 흐름과 예외 흐름을 명확하게 분리해서 코드를 더 쉽게 읽을 수 있게 되었다.

남은 문제

- 사용 후에는 반드시 `disconnect()`를 호출해서 연결을 해제해야 한다.

앞서 이야기했듯이 외부 연결과 같은 자바 외부의 자원은 자동으로 해제가 되지 않는다. 따라서 외부 자원을 사용한 후에는 연결을 해제해서 외부 자원을 반드시 반납해야 한다.
예외가 발생해도 `disconnect()`를 반드시 호출해서 연결을 해제하고 자원을 반납하려면 어떻게 해야할까?

예외 처리 도입4 - 리소스 반환 문제

현재 구조에서 `disconnect()`를 항상 호출하려면 다음과 같이 생각할 수 있다.

```
package exception.ex2;

public class NetworkServiceV2_4 {

    public void sendMessage(String data) {
        String address = "http://example.com";

        NetworkClientV2 client = new NetworkClientV2(address);
        client.initError(data);

        try {
```

```
            client.connect();
            client.send(data);
        } catch (NetworkClientExceptionV2 e) {
            System.out.println("[오류] 코드: " + e.getErrorCode() + ", 메시지: " +
e.getMessage());
        }

        //NetworkClientException이 아닌 다른 예외가 발생해서 예외가 밖으로 던져지면 무시
        client.disconnect();
    }

}
```

- 이 코드를 보면 예외 처리가 끝난 다음에 정상 흐름의 마지막에 `client.disconnect()`를 호출했다.
- 이렇게 하면 예외가 모두 처리되었기 때문에 `client.disconnect()`가 항상 호출될 것 같다.

MainV2 - 코드 변경

```
public static void main(String[] args) throws NetworkClientExceptionV2 {
    NetworkServiceV2_4 networkService = new NetworkServiceV2_4();
    ...
}
```

실행 결과

```
전송할 문자: hello
http://example.com 서버 연결 성공
http://example.com 서버에 데이터 전송: hello
http://example.com 서버 연결 해제

전송할 문자: error1
[오류] 코드: connectError, 메시지: http://example.com 서버 연결 실패
http://example.com 서버 연결 해제

전송할 문자: error2
http://example.com 서버 연결 성공
[오류] 코드: sendError, 메시지: http://example.com 서버에 데이터 전송 실패: error2
http://example.com 서버 연결 해제

전송할 문자: exit
프로그램을 정상 종료합니다.
```

코드를 실행해보면 오류가 발생해도 서버 연결 해제에 성공하는 것을 확인할 수 있다.

하지만 지금과 같은 방식에는 큰 문제가 있다.
바로 `catch`에서 잡을 수 없는 예외가 발생할 때이다.

NetworkClientV2- 코드 변경

```
public void send(String data) throws NetworkClientExceptionV2 {
    if (sendError) {
        //throw new NetworkClientExceptionV2("sendError", address + " 서버에 데이터 전송 실패: " + data);
        //중간에 다른 예외가 발생했다고 가정
        throw new RuntimeException("ex");
    }
    //전송 성공
    System.out.println(address + " 서버에 데이터 전송: " + data);
}
```

- `NetworkClientV2.send()`에서 발생하는 예외를 자바가 기본으로 제공하는 `RuntimeException`으로 잠깐 변경하고 실행해보자.
- `catch`에서 `NetworkClientExceptionV2`은 잡을 수 있지만 새로 등장한 `RuntimeException`은 잡을 수 없다.

실행 결과

```
전송할 문자: error1
[오류] 코드: connectError, 메시지: http://example.com 서버 연결 실패
http://example.com 서버 연결 해제

전송할 문자: error2
http://example.com 서버 연결 성공
Exception in thread "main" java.lang.RuntimeException: ex
    at exception.ex2.NetworkClientV2.send(NetworkClientV2.java:25)
    at exception.ex2.NetworkServiceV2_4.sendMessage(NetworkServiceV2_4.java:13)
    at exception.ex2.MainV2.main(MainV2.java:22)
```

- `error1`의 경우 `client.connect()`에서 `NetworkClientExceptionV2` 예외가 발생하기 때문에 바로 `catch`로 이동해서 정상 흐름을 이어간다.
- `error2`의 경우 `client.send()`에서 `RuntimeException`이 발생한다. 이 예외는 `catch`의 대상이 아니므로 잡지 않고 즉시 밖으로 던져진다.

```
try {
    client.connect();
    client.send(data); //1. RuntimeException은 catch 대상이 아님. 예외가 밖으로 던져짐
} catch (NetworkClientExceptionV2 e) {
    System.out.println("[오류] 코드: " + e.getErrorCode() + ", 메시지: " +
e.getMessage());
}

client.disconnect(); //2. 이 코드는 호출되지 않음
```

따라서 `client.disconnect()`가 호출되지 않는 문제가 발생한다.

주의!

실행을 완료했다면 기존 예제들이 정상 작동하도록 다음과 같이 코드를 다시 복구하자.

NetworkClientV2- 코드 복구

```
public void send(String data) throws NetworkClientExceptionV2 {
    if (sendError) {
        throw new NetworkClientExceptionV2("sendError", address + " 서버에 데이터
전송 실패: " + data);
        //중간에 다른 예외가 발생했다고 가정
        //throw new RuntimeException("ex");
    }
    //전송 성공
    System.out.println(address + " 서버에 데이터 전송: " + data);
}
```

사용 후에 반드시 `disconnect()`를 호출해서 연결 해제를 보장하는 것은 쉽지 않다. 왜냐하면 정상적인 상황, 예외 상황 그리고 어디선가 모르는 예외를 밖으로 던지는 상황까지 모든 것을 고려해야 한다. 하지만 앞서 보았듯이 지금과 같은 구조로는 항상 `disconnect()`와 같은 코드를 호출하는 것이 매우 어렵고 실수로 놓칠 가능성이 높다.
결국 새로운 대안이 필요하다.

예외 처리 도입5 - finally

자바는 어떤 경우라도 반드시 호출되는 `finally` 기능을 제공한다.

```
try {
  //정상 흐름
} catch  {
  //예외 흐름
} finally {
  //반드시 호출해야 하는 마무리 흐름
}
```

- `try ~ catch ~ finally` 구조는 정상 흐름, 예외 흐름, 마무리 흐름을 제공한다.
- 여기서 `try`를 시작하기만 하면, `finally` 코드 블럭은 어떤 경우라도 반드시 호출된다.
- 심지어 `try`, `catch` 안에서 잡을 수 없는 예외가 발생해도 `finally`는 반드시 호출된다.

정리하면 다음과 같다.

- 정상 흐름➜ `finally`
- 예외 catch ➜ `finally`
- 예외 던짐➜ `finally`
  - `finally` 코드 블럭이 끝나고 나서 이후에 예외가 밖으로 던져짐

`finally` 블럭은 반드시 호출된다. 따라서 주로 `try`에서 사용한 자원을 해제할 때 주로 사용한다.

```
package exception.ex2;

public class NetworkServiceV2_5 {

    public void sendMessage(String data) {
        String address = "https://example.com";

        NetworkClientV2 client = new NetworkClientV2(address);
        client.initError(data);

        try {
            client.connect();
```

```
            client.send(data);
        } catch (NetworkClientExceptionV2 e) {
            System.out.println("[오류] 코드: " + e.getErrorCode() + ", 메시지: " +
e.getMessage());
        } finally {
            client.disconnect();
        }
    }

}
```

MainV2 - 코드 변경

```
public static void main(String[] args) throws NetworkClientExceptionV2 {
    NetworkServiceV2_5 networkService = new NetworkServiceV2_5();
    ...
}
```

실행 결과

```
전송할 문자: hello
https://example.com 서버 연결 성공
https://example.com 서버에 데이터 전송: hello
https://example.com 서버 연결 해제

전송할 문자: error1
[오류] 코드: connectError, 메시지: https://example.com 서버 연결 실패
https://example.com 서버 연결 해제

전송할 문자: error2
https://example.com 서버 연결 성공
[오류] 코드: sendError, 메시지: https://example.com 서버에 데이터 전송 실패: error2
https://example.com 서버 연결 해제

전송할 문자: exit
프로그램을 정상 종료합니다.
```

처리할 수 없는 예외와 finally

`try`, `catch` 안에서 처리할 수 없는 예외가 발생해도 `finally`는 반드시 호출된다.
앞서본 예제와 같이 `NetworkClientV2.send(data)`에서 처리할 수 없는 `RuntimeException`이 발생했다고 가정해보자.

NetworkClientV2- 코드 변경

```
public void send(String data) throws NetworkClientExceptionV2 {
    if (sendError) {
        //throw new NetworkClientExceptionV2("sendError", address + " 서버에 데이터 전송 실패: " + data);
        //중간에 다른 예외가 발생했다고 가정
        throw new RuntimeException("ex");
    }
    //전송 성공
    System.out.println(address + " 서버에 데이터 전송: " + data);
}
```

실행 결과

```
전송할 문자: error2
https://example.com 서버 연결 성공
https://example.com 서버 연결 해제
Exception in thread "main" java.lang.RuntimeException: ex
    at exception.ex2.NetworkClientV2.send(NetworkClientV2.java:24)
    at
exception.ex2.NetworkServiceV2_5.sendMessage(NetworkServiceV2_5.java:13)
    at exception.ex2.MainV2.main(MainV2.java:22)
```

- 실행 결과를 보면 예외를 밖으로 던지는 경우에도 서버 연결 해제에 성공하는 것을 확인할 수 있다.
- `catch`에서 잡을 수 없는 예외가 발생해서, 예외를 밖으로 던지는 경우에도 `finally`를 먼저 호출하고 나서 예외를 밖으로 던진다.

주의!

실행을 완료했다면 기존 예제들이 정상 작동하도록 다음과 같이 코드를 복구하자.

NetworkClientV2- 코드 복구

```
public void send(String data) throws NetworkClientExceptionV2 {
    if (sendError) {
```

```
            throw new NetworkClientExceptionV2("sendError", address + " 서버에 데이터 전송 실패: " + data);
            //throw new RuntimeException("ex");
        }
        //전송 성공
        System.out.println(address + " 서버에 데이터 전송: " + data);
    }
```

try ~ finally

다음과 같이 `catch` 없이 `try ~ finally`만 사용할 수도 있다.

```
try {
    client.connect();
    client.send(data);
} finally {
    client.disconnect();
}
```

예외를 직접 잡아서 처리할 일이 없다면 이렇게 사용하면 된다. 이렇게 하면 예외를 밖으로 던지는 경우에도 `finally` 호출이 보장된다.

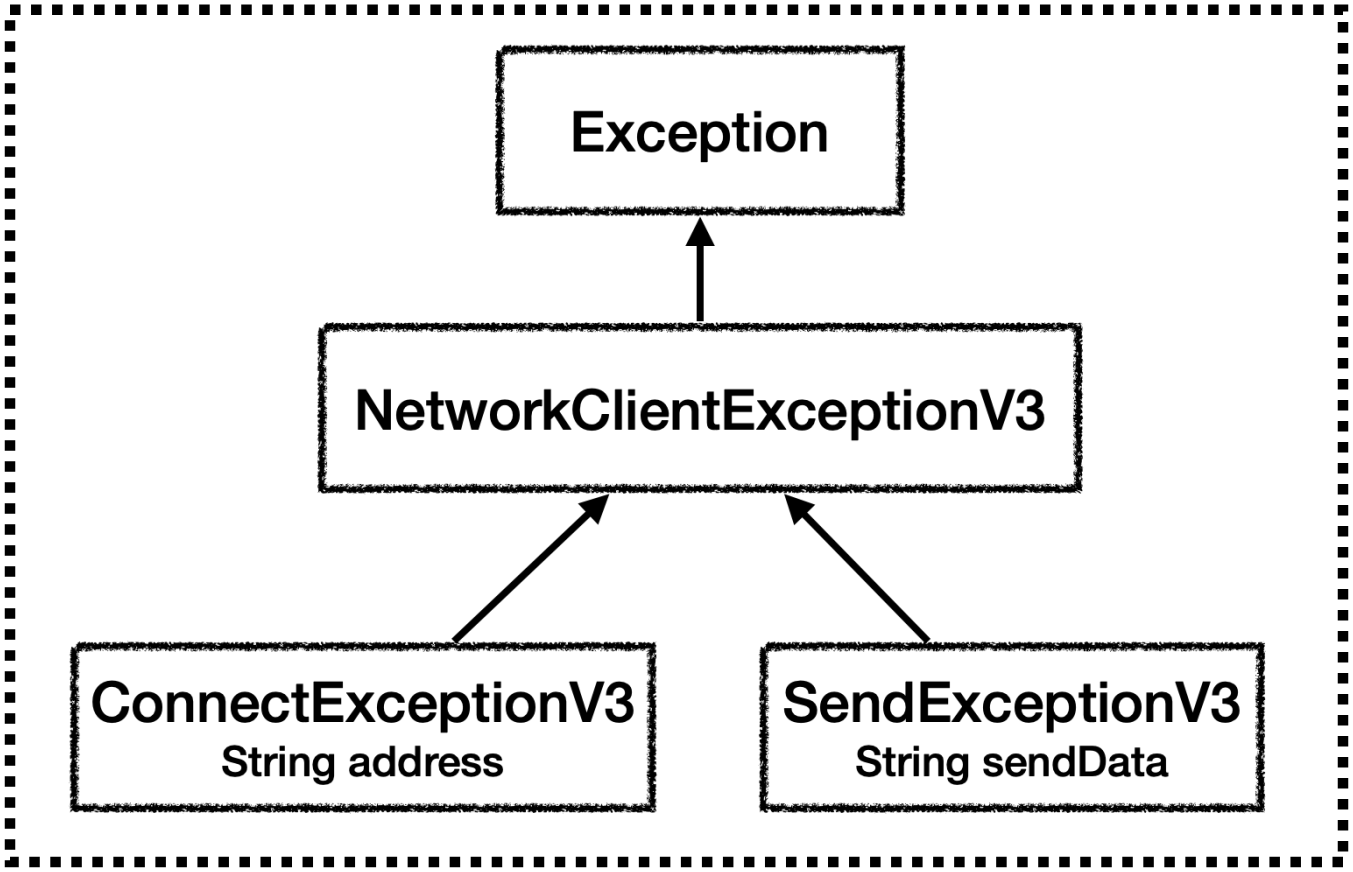
정리

자바 예외 처리는 `try ~ catch ~ finally` 구조를 사용해서 처리할 수 있다. 덕분에 다음과 같은 이점이 있다.

- 정상 흐름과 예외 흐름을 분리해서, 코드를 읽기 쉽게 만든다.
- 사용한 자원을 항상 반환할 수 있도록 보장해준다.

예외 계층1 - 시작

체크 예외



예외를 단순히 오류 코드로 분류하는 것이 아니라, 예외를 계층화해서 다양하게 만들면 더 세밀하게 예외를 처리할 수 있다.

- `NetworkClientExceptionV3` : `NetworkClient` 에서 발생하는 모든 예외는 이 예외의 자식이다.
- `ConnectExceptionV3` : 연결 실패시 발생하는 예외이다. 내부에 연결을 시도한 `address` 를 보관한다.
- `SendExceptionV3` : 전송 실패시 발생하는 예외이다. 내부에 전송을 시도한 데이터인 `sendData` 를 보관한다.

이렇게 예외를 계층화하면 다음과 같은 장점이 있다.

- 자바에서 예외는 객체이다. 따라서 부모 예외를 잡거나 던지면, 자식 예외도 함께 잡거나 던질 수 있다. 예를 들어서 `NetworkClientExceptionV3` 예외를 잡으면 그 하위인 `ConnectExceptionV3` , `SendExceptionV3` 예외도 함께 잡을 수 있다.
- 특정 예외를 잡아서 처리하고 싶으면 `ConnectExceptionV3` , `SendExceptionV3` 와 같은 하위 예외를 잡아서 처리하면 된다.

예외는 구분하기 쉽도록 `exception` 하위 패키지에 별도로 모아두겠다.

```
package exception.ex3.exception;

public class NetworkClientExceptionV3 extends Exception {
```

```
    public NetworkClientExceptionV3(String message) {
        super(message);
    }
}
```

- `NetworkClient`에서 발생하는 모든 예외는 이 예외를 부모로 하도록 설계한다.

```
package exception.ex3.exception;

public class ConnectExceptionV3 extends NetworkClientExceptionV3 {

    private final String address;

    public ConnectExceptionV3(String address, String message) {
        super(message);
        this.address = address;
    }

    public String getAddress() {
        return address;
    }
}
```

- `ConnectExceptionV3` : 연결 실패시 발생하는 예외이다. 내부에 연결을 시도한 `address`를 보관한다.
- `NetworkClientExceptionV3`를 상속했다.

```
package exception.ex3.exception;

public class SendExceptionV3 extends NetworkClientExceptionV3 {

    private final String sendData;

    public SendExceptionV3(String sendData, String message) {
        super(message);
        this.sendData = sendData;
    }

    public String getSendData() {
        return sendData;
```

```
    }
}
```

- `SendExceptionV3` : 전송 실패시 발생하는 예외이다. 내부에 전송을 시도한 데이터인 `sendData` 를 보관한다.
- `NetworkClientExceptionV3` 를 상속했다.

```
package exception.ex3;

import exception.ex3.exception.ConnectExceptionV3;
import exception.ex3.exception.SendExceptionV3;

public class NetworkClientV3 {

    private final String address;
    public boolean connectError;
    public boolean sendError;

    public NetworkClientV3(String address) {
        this.address = address;
    }

    public void connect() throws ConnectExceptionV3 {
        if (connectError) {
            throw new ConnectExceptionV3(address, address + " 서버 연결 실패");
        }

        System.out.println(address + " 서버 연결 성공");
    }

    public void send(String data) throws SendExceptionV3 {
        if (sendError) {
            throw new SendExceptionV3(data, address + " 서버에 데이터 전송 실패: " +
data);
        }

        System.out.println(address + " 서버에 데이터 전송: " + data);
    }

    public void disconnect() {
```

```
        System.out.println(address + " 서버 연결 해제");
    }

    public void initError(String data) {
        if (data.contains("error1")) {
            connectError = true;
        }
        if (data.contains("error2")) {
            sendError = true;
        }
    }
}
```

- 예외는 별도의 패키지에 정의되어 있다. `import`에 주의하자.
- 오류 코드로 어떤 문제가 발생했는지 이해하는 것이 아니라 예외 그 자체로 어떤 오류가 발생했는지 알 수 있다.
- 연결 관련 오류 발생하면 `ConnectExceptionV3`를 던지고, 전송 관련 오류가 발생하면 `SendExceptionV3`를 던진다.

```
package exception.ex3;

import exception.ex3.exception.ConnectExceptionV3;
import exception.ex3.exception.SendExceptionV3;

public class NetworkServiceV3_1 {

    public void sendMessage(String data) {
        String address = "https://example.com";

        NetworkClientV3 client = new NetworkClientV3(address);
        client.initError(data);

        try {
            client.connect();
            client.send(data);
        } catch (ConnectExceptionV3 e) {
            System.out.println("[연결 오류] 주소: " + e.getAddress() + ", 메시지: "
+ e.getMessage());
        } catch (SendExceptionV3 e) {
            System.out.println("[전송 오류] 전송 데이터: " + e.getSendData() + ", 메
시지: " + e.getMessage());
        } finally {
```

```
            client.disconnect();
        }
    }
}
```

- 예외 클래스를 각각의 예외 상황에 맞추어 만들면, 각 필요에 맞는 예외를 잡아서 처리할 수 있다.
- 예를 들면 `e.getAddress()`, `e.getSendData()`와 같이 각각의 예외 클래스가 가지는 고유의 기능을 활용할 수 있다.
- `catch (ConnectExceptionV3 e)`: 연결 예외를 잡고, 해당 예외가 제공하는 기능을 사용해서 정보를 출력한다.
- `catch (SendExceptionV3 e)`: 전송 예외를 잡고, 해당 예외가 제공하는 기능을 사용해서 정보를 출력한다.

```
package exception.ex3;

import exception.ex3.exception.NetworkClientExceptionV3;

import java.util.Scanner;

public class MainV3 {

    public static void main(String[] args) {
        NetworkServiceV3_1 networkService = new NetworkServiceV3_1();

        Scanner scanner = new Scanner(System.in);
        while (true) {
            System.out.print("전송할 문자: ");
            String input = scanner.nextLine();
            if (input.equals("exit")) {
                break;
            }
            networkService.sendMessage(input);
            System.out.println();
        }
        System.out.println("프로그램을 정상 종료합니다.");
    }
}
```

실행 결과

```
전송할 문자: hello
https://example.com 서버 연결 성공
https://example.com 서버에 데이터 전송: hello
https://example.com 서버 연결 해제

전송할 문자: error1
[연결 오류] 주소: https://example.com, 메시지: https://example.com 서버 연결 실패
https://example.com 서버 연결 해제

전송할 문자: error2
https://example.com 서버 연결 성공
[전송 오류] 전송 데이터: error2, 메시지: https://example.com 서버에 데이터 전송 실패: error2
https://example.com 서버 연결 해제

전송할 문자: exit
프로그램을 정상 종료합니다.
```

실행 결과를 보면 `ConnectExceptionV3`, `SendExceptionV3`이 발생한 각각의 경우에 출력된 오류 메시지가 다른 것을 확인할 수 있다.

예외 계층2 - 활용

이번에는 예외를 잡아서 처리할 때 예외 계층을 활용해보자.

`NetworkClientV3`에서 수 많은 예외를 발생한다고 가정해보자. 이런 경우 모든 예외를 하나하나 다 잡아서 처리하는 것은 상당히 번거로울 것이다. 그래서 다음과 같이 예외를 처리하도록 구성해보자.

- 연결 오류는 중요하다. `ConnectExceptionV3`가 발생하면 다음과 같이 메시지를 명확하게 남기도록 하자.
  - 예) `[연결 오류] 주소: ...`
- `NetworkClientV3`을 사용하면서 발생하는 나머지 예외(`NetworkClientExceptionV3`의 자식)는 단순하게 다음과 같이 출력하자
  - 예) `[네트워크 오류] 메시지:`
- 그 외에 예외가 발생하면 다음과 같이 출력하자.
  - 예) `[알 수 없는 오류] 메시지:`

```
package exception.ex3;

import exception.ex3.exception.ConnectExceptionV3;
import exception.ex3.exception.NetworkClientExceptionV3;

public class NetworkServiceV3_2 {

    public void sendMessage(String data) {
        String address = "https://example.com";

        NetworkClientV3 client = new NetworkClientV3(address);
        client.initError(data);

        try {
            client.connect();
            client.send(data);
        } catch (ConnectExceptionV3 e) {
            System.out.println("[연결 오류] 주소: " + e.getAddress() + ", 메시지: "
+ e.getMessage());
        } catch (NetworkClientExceptionV3 e) {
            System.out.println("[네트워크 오류] 메시지: " + e.getMessage());
        } catch (Exception e) {
            System.out.println("[알 수 없는 오류] 메시지: " + e.getMessage());
        } finally {
            client.disconnect();
        }
    }
}
```

`catch`는 순서대로 작동한다. 각 예외 별로 어떻게 작동하는지 알아보자.

ConnectExceptionV3 발생

```
try {
  // 1. ConnectExceptionV3 발생
} catch (ConnectExceptionV3 e) { // 2. 잡아서 처리
} catch (NetworkClientExceptionV3 e) {
} catch (Exception e) {
}
```

SendExceptionV3 발생

```
try {
  // 1. SendExceptionV3 발생
} catch (ConnectExceptionV3 e) { // 2. 대상이 다름
} catch (NetworkClientExceptionV3 e) { // 3.NetworkClientExceptionV3은 부모이므로 여기서 잡음
} catch (Exception e) {
}
```

- `NetworkClientExceptionV3`은 `SendExceptionV3`의 부모이다. 부모 타입은 자식을 담을 수 있다. 따라서 `NetworkClientExceptionV3`을 잡으면 `SendExceptionV3`도 잡을 수 있다.

RuntimeException 발생

```
try {
  // 1. RuntimeException 발생
} catch (ConnectExceptionV3 e) { // 2. 대상이 다름
} catch (NetworkClientExceptionV3 e) { // 3.대상이 다름
} catch (Exception e) { // 4.Exception은 RuntimeException의 부모이므로 여기서 잡음
}
```

- 모든 예외를 잡아서 처리하려면 마지막에 `Exception`을 두면 된다.

주의할 점은 예외가 발생했을 때 `catch`를 순서대로 실행하므로, 더 디테일한 자식을 먼저 잡아야 한다.

MainV3 - 코드 변경

```
public static void main(String[] args) throws NetworkClientExceptionV3 {
    NetworkServiceV3_2 networkService = new NetworkServiceV3_2();
    ...
}
```

실행 결과

```
전송할 문자: hello
https://example.com 서버 연결 성공
https://example.com 서버에 데이터 전송: hello
```

```
https://example.com 서버 연결 해제

전송할 문자: error1
[연결 오류] 주소: https://example.com, 메시지: https://example.com 서버 연결 실패
https://example.com 서버 연결 해제

전송할 문자: error2
https://example.com 서버 연결 성공
[네트워크 오류] 메시지: https://example.com 서버에 데이터 전송 실패: error2
https://example.com 서버 연결 해제

전송할 문자: exit
프로그램을 정상 종료합니다.
```

예외 별로 다른 메시지가 출력되는 것을 확인할 수 있다.

여러 예외를 한번에 잡는 기능

다음과 같이 `|`를 사용해서 여러 예외를 한번에 잡을 수 있다.

```
try {
    client.connect();
    client.send(data);
} catch (ConnectExceptionV3 | SendExceptionV3 e) {
    System.out.println("[연결 또는 전송 오류] 주소: , 메시지: " + e.getMessage());
} finally {
    client.disconnect();
}
```

참고로 이 경우 각 예외들의 공통 부모의 기능만 사용할 수 있다. 여기서는 `NetworkClientExceptionV3`의 기능만 사용할 수 있다.

정리

예외를 계층화하고 다양하게 만들면 더 세밀한 동작들을 깔끔하게 처리할 수 있다. 그리고 특정 분류의 공통 예외들도 한번에 `catch`로 잡아서 처리할 수 있다.

실무 예외 처리 방안1 - 설명

처리할 수 없는 예외

예를 들어서 상대 네트워크 서버에 문제가 발생해서 통신이 불가능하거나, 데이터베이스 서버에 문제가 발생해서 접속이 안되면, 애플리케이션에서 연결 오류, 데이터베이스 접속 실패와 같은 예외가 발생한다.

이렇게 시스템 오류 때문에 발생한 예외들은 대부분 예외를 잡아도 해결할 수 있는 것이 거의 없다. 예외를 잡아서 다시 호출을 시도해도 같은 오류가 반복될 뿐이다.

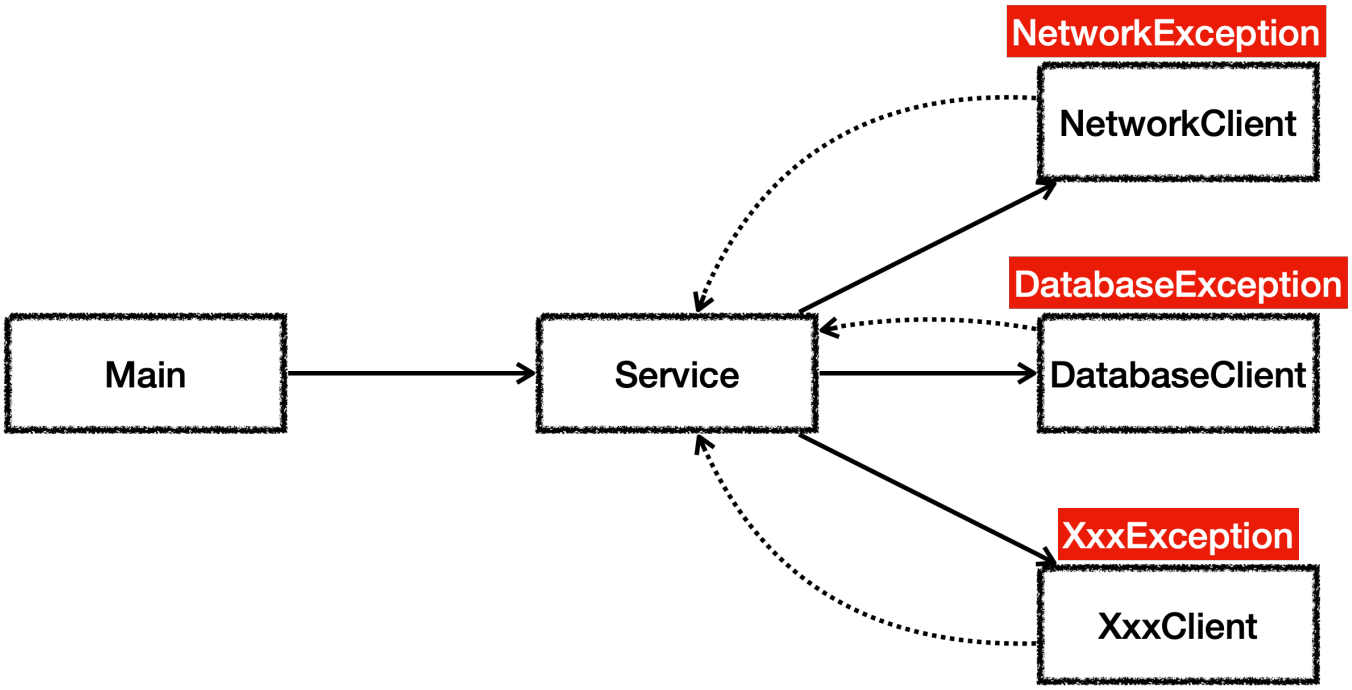
이런 경우 고객에게는 "현재 시스템에 문제가 있습니다."라는 오류 메시지를 보여주고, 만약 웹이라면 오류 페이지를 보여주면 된다. 그리고 내부 개발자가 문제 상황을 빠르게 인지할 수 있도록, 오류에 대한 로그를 남겨두어야 한다.

체크 예외의 부담

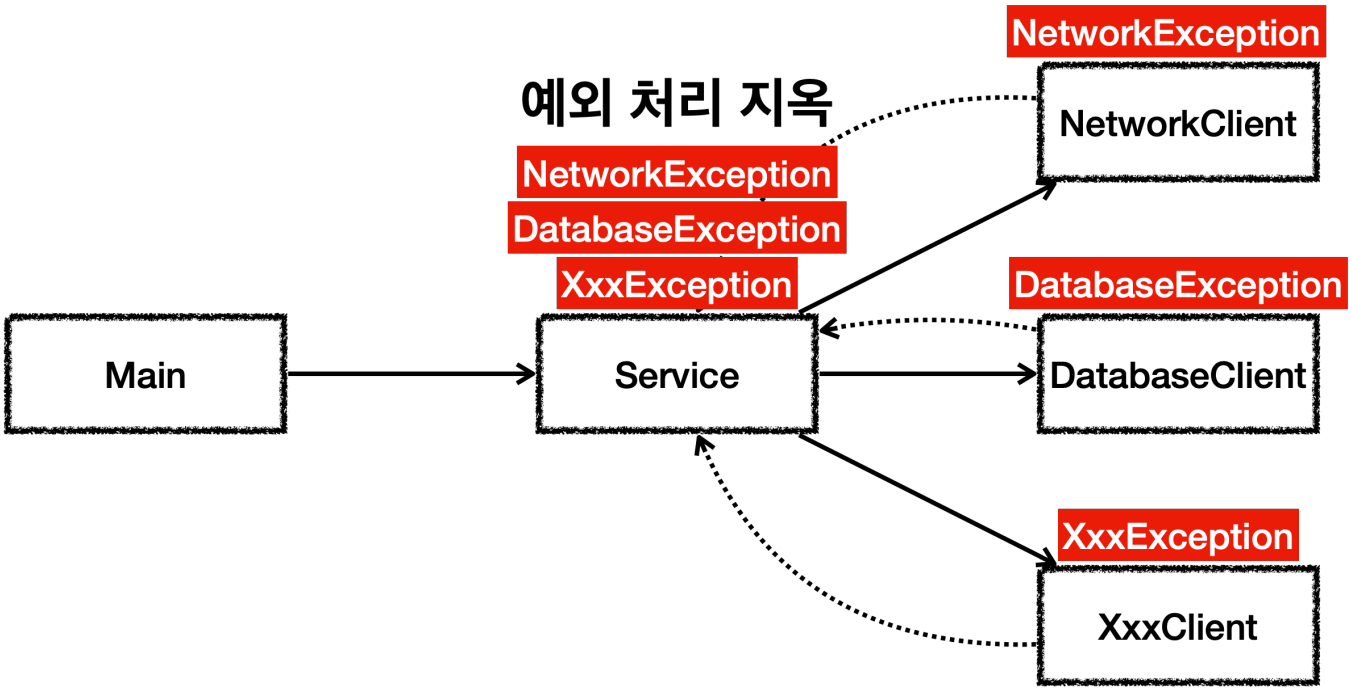
체크 예외는 개발자가 실수로 놓칠 수 있는 예외들을 컴파일러가 체크해주기 때문에 오래전부터 많이 사용되었다. 그런데 앞서 설명한 것 처럼 처리할 수 없는 예외가 많아지고, 또 프로그램이 점점 복잡해지면서 체크 예외를 사용하는 것이 점점 더 부담스러워졌다.

체크 예외 사용 시나리오

체크 예외를 사용하게 되면 어떤 문제가 발생하는지 가상의 시나리오로 이야기하겠다.



- 실무에서는 수 많은 라이브러리를 사용하고, 또 다양한 외부 시스템과 연동한다.
- 사용하는 각각의 클래스들이 자신만의 예외를 모두 체크 예외로 만들어서 전달한다고 가정하자.



- 이 경우 `Service`는 호출하는 곳에서 던지는 체크 예외들을 처리해야 한다. 만약 처리할 수 없다면 밖으로 던져야 한다.

모든 체크 예외를 잡아서 처리하는 예시

```
try {
} catch (NetworkException) {...}
} catch (DatabaseException) {...}
} catch (XxxException) {...}
```

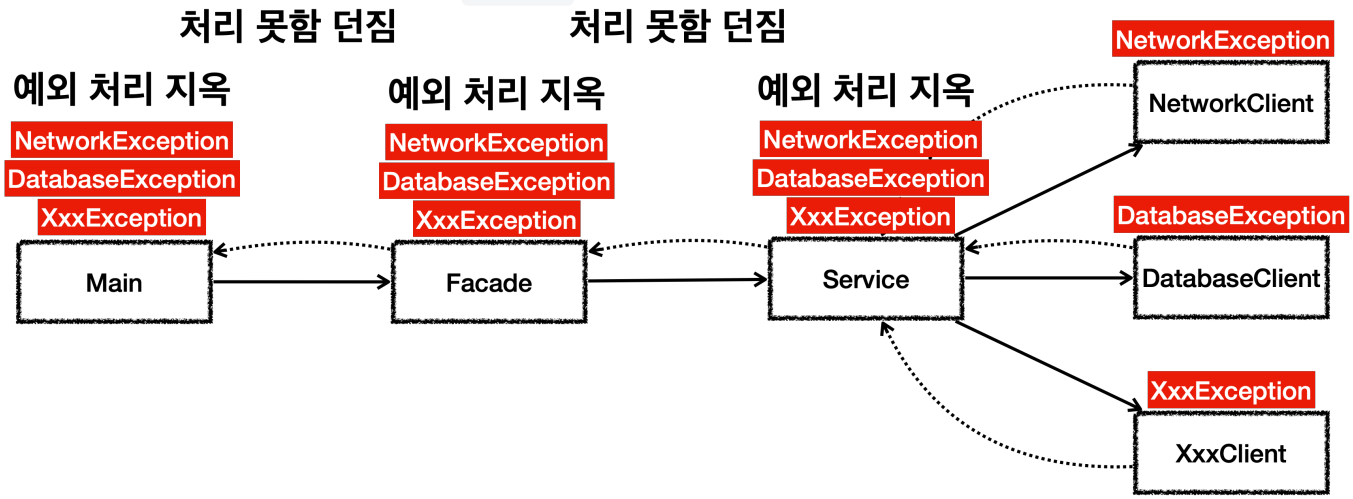
- 그런데 앞서 설명했듯이 상대 네트워크 서버가 내려갔거나, 데이터베이스 서버에 문제가 발생한 경우 `Service`에서 예외를 잡아도 복구할 수 없다.
- `Service`에서는 어차피 본인이 처리할 수 없는 예외들이기 때문에 밖으로 던지는 것이 더 나은 결정이다.

모든 체크 예외를 던지는 예시

```
class Service {
    void sendMessage(String data) throws NetworkException,
DatabaseException, ...{
        ...
    }
}
```

- 이렇게 모든 체크예외를 하나씩 다 밖으로 던져야한다.
- 라이브러리가 늘어날 수 록 다루어야 하는 예외도 더 많아진다. 개발자 입장에서 이것은 상당히 번거로운 일이 된다.

문제는 여기서 끝이 아니다. 만약 중간에 `Facade`라는 클래스가 있다고 가정해보자.



- 이 경우 `Facade` 클래스에서도 이런 예외들을 복구할 수 없다. `Facade` 클래스도 예외를 밖으로 던져야 한다.
- 결국 중간에 모든 클래스에서 예외를 계속 밖으로 던지는 지저분한 코드가 만들어진다.
- `throws`로 발견한 모든 예외를 다 밖으로 던지는 것이다.

```
class Facade {
    void send() throws NetworkException, DatabaseException, ...
}

class Service {
    void sendMessage(String data) throws NetworkException,
DatabaseException, ...
}
```

throws Exception

개발자는 본인이 다룰 수 없는 수 많은 체크 예외 지옥에 빠지게 된다. 결국 다음과 같이 최악의 수를 두게 된다.

```
class Facade {
    void send() throws Exception
}

class Service {
    void sendMessage(String data) throws Exception
}
```

`Exception`은 애플리케이션에서 일반적으로 다루는 모든 예외의 부모이다. 따라서 이렇게 한 줄만 넣으면 모든 예외를 다 던질 수 있다.

이렇게 하면 `Exception`은 물론이고 그 하위 타입인 `NetworkException`, `DatabaseException`도 함께 던지게 된다. 그리고 이후에 예외가 추가되더라도 `throws Exception`은 변경하지 않고 그대로 유지할 수 있다. 코드가 깔끔해지는 것 같지만 이 방법에는 치명적인 문제가 있다.

throws Exception의 문제

`Exception`은 최상위 타입이므로 모든 체크 예외를 다 밖으로 던지는 문제가 발생한다.
결과적으로 체크 예외의 최상위 타입인 `Exception`을 던지게 되면 다른 체크 예외를 체크할 수 있는 기능이 무효화되고, 중요한 체크 예외를 다 놓치게 된다. 중간에 중요한 체크 예외가 발생해도 컴파일러는 `Exception`을 던지기 때문에 문법에 맞다고 판단해서 컴파일 오류가 발생하지 않는다.
이렇게 하면 모든 예외를 다 던지기 때문에 체크 예외를 의도한 대로 사용하는 것이 아니다. 따라서 꼭 필요한 경우가 아니면 이렇게 `Exception` 자체를 밖으로 던지는 것은 좋지 않은 방법이다.

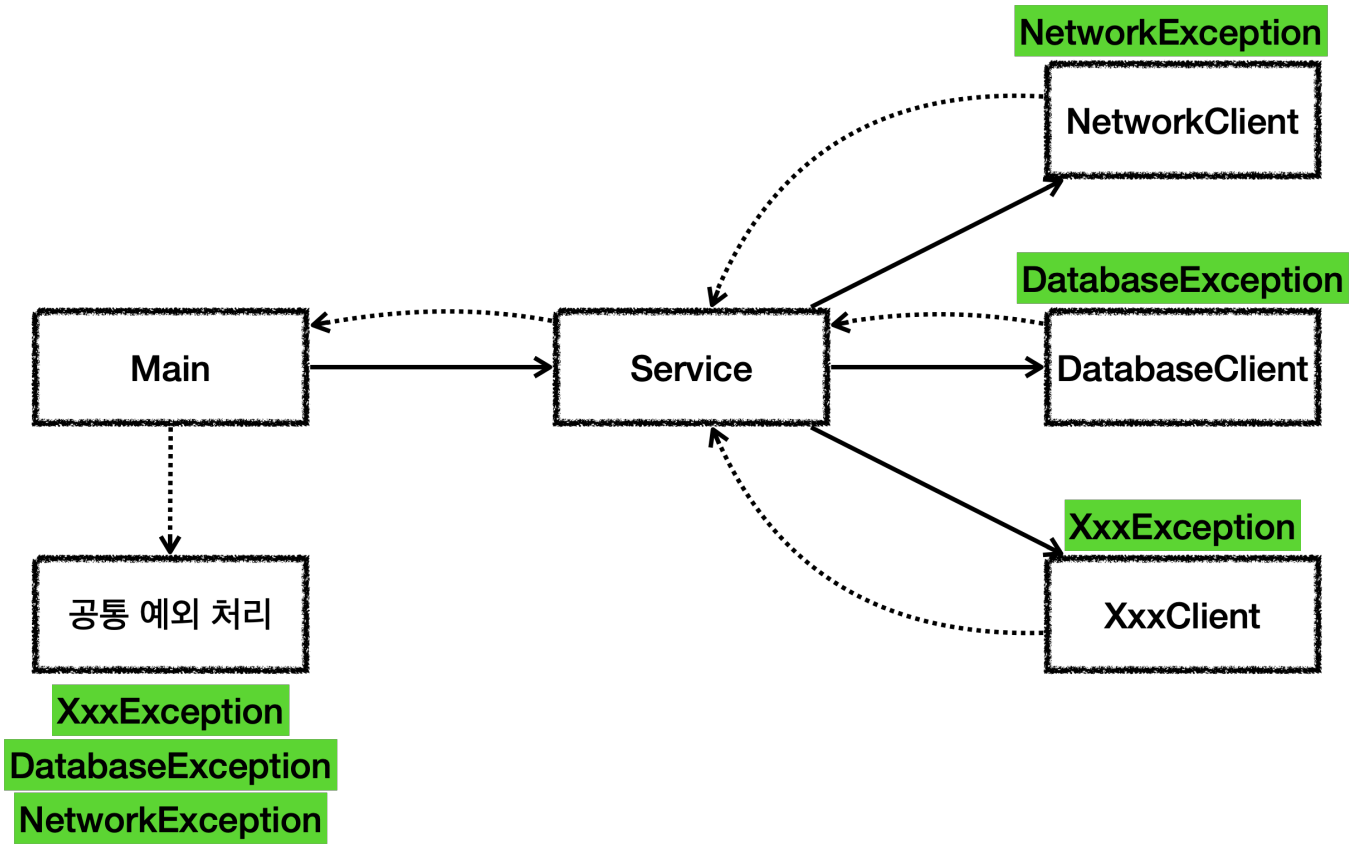
문제 정리

지금까지 알아본 체크 예외를 사용할 때 발생하는 문제들은 다음과 같다.

- **처리할 수 없는 예외**: 예외를 잡아서 복구할 수 있는 예외보다 복구할 수 없는 예외가 더 많다.
- **체크 예외의 부담**: 처리할 수 없는 예외는 밖으로 던져야 한다. 체크 예외이므로 `throws`에 던질 대상을 일일이 명시해야 한다.

사실 `Service`를 개발하는 개발자 입장에서 수 많은 라이브러리에서 쏟아지는 모든 예외를 다 다루고 싶지는 않을 것이다. 특히 본인이 해결할 수 도 없는 모든 예외를 다 다루고 싶지는 않을 것이다. 본인이 해결할 수 있는 예외만 잡아서 처리하고, 본인이 해결할 수 없는 예외는 신경쓰지 않는 것이 더 나은 선택일 수 있다.

언체크(런타임) 예외 사용 사나리오



- 이번에는 `Service`에서 호출하는 클래스들이 언체크(런타임) 예외를 전달한다고 가정해보자.
- `NetworkException`, `DatabaseException`은 잡아도 복구할 수 없다. 언체크 예외이므로 이런 경우 무시하면 된다.

언체크 예외를 던지는 예시

```
class Service {
    void sendMessage(String data) {
        ...
    }
}
```

- 언체크 예외이므로 `throws`를 선언하지 않아도 된다.
- 사용하는 라이브러리가 늘어나서 언체크 예외가 늘어도 본인이 필요한 예외만 잡으면 되고, `throws`를 늘리지 않아도 된다.

일부 언체크 예외를 잡아서 처리하는 예시

```
try {
} catch (XxxException) {...}
```

- 앞서 설명했듯이 상대 네트워크 서버가 내려갔거나, 데이터베이스 서버에 문제가 발생한 경우 `Service`에서 예

외를 잡아도 복구할 수 없다.

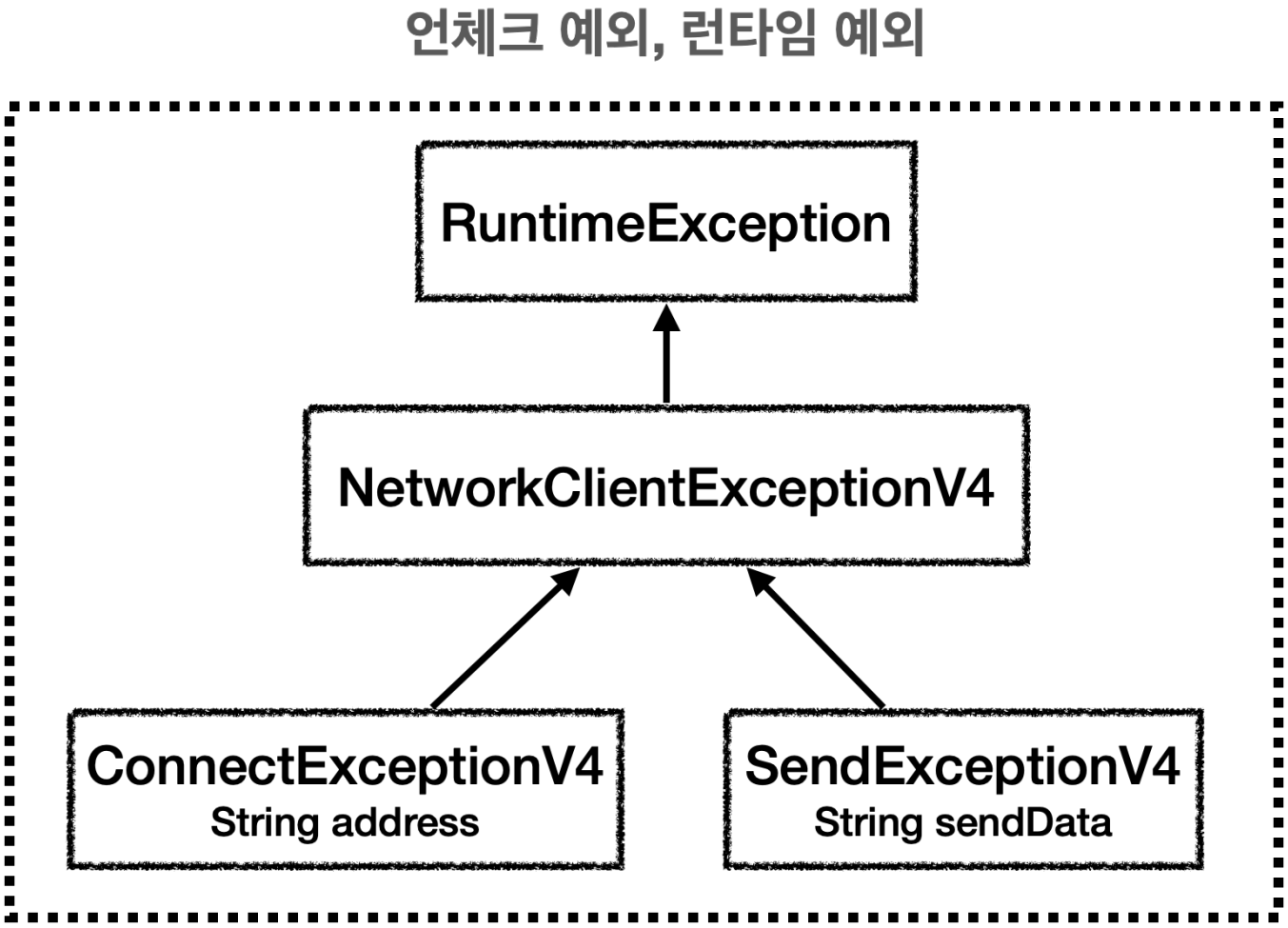
- `Service`에서는 어차피 본인이 처리할 수 없는 예외들이기 때문에 밖으로 던지는 것이 더 나은 결정이다.
- 언체크 예외는 잡지 않으면 `throws` 선언이 없어도 자동으로 밖으로 던진다.
- 만약 일부 언체크 예외를 잡아서 처리할 수 있다면 잡아서 처리하면 된다.

예외 공통 처리

이렇게 처리할 수 없는 예외들은 중간에 여러곳에서 나누어 처리하기 보다는 예외를 공통으로 처리할 수 있는 곳을 만들어서 한 곳에서 해결하면 된다. 어차피 해결할 수 없는 예외들이기 때문에 이런 경우 고객에게는 현재 시스템에 문제가 있습니다. 라고 오류 메시지를 보여주고, 만약 웹이라면 오류 페이지를 보여주면 된다. 그리고 내부 개발자가 지금의 문제 상황을 빠르게 인지할 수 있도록, 오류에 대한 로그를 남겨두면 된다. 이런 부분은 공통 처리가 가능하다.

실무 예외 처리 방안2 - 구현

지금까지 실습한 내용을 언체크 예외로 만들고, 또 해결할 수 없는 예외들을 공통으로 처리해보자.



- `NetworkClientExceptionV4`는 언체크 예외인 `RuntimeException`을 상속 받는다.
- 이제 `NetworkClientExceptionV4`와 자식은 모두 언체크(런타임) 예외가 된다.

```
package exception.ex4.exception;

public class NetworkClientExceptionV4 extends RuntimeException {

    public NetworkClientExceptionV4(String message) {
        super(message);
    }
}
```

```
package exception.ex4.exception;

public class ConnectExceptionV4 extends NetworkClientExceptionV4 {

    private final String address;

    public ConnectExceptionV4(String address, String message) {
        super(message);
        this.address = address;
    }

    public String getAddress() {
        return address;
    }
}
```

```
package exception.ex4.exception;

public class SendExceptionV4 extends NetworkClientExceptionV4 {

    private final String sendData;

    public SendExceptionV4(String sendData, String message) {
        super(message);
        this.sendData = sendData;
```

```
    }

    public String getSendData() {
        return sendData;
    }

}
```

```
package exception.ex4;

import exception.ex4.exception.ConnectExceptionV4;
import exception.ex4.exception.SendExceptionV4;

public class NetworkClientV4 {

    private final String address;
    public boolean connectError;
    public boolean sendError;

    public NetworkClientV4(String address) {
        this.address = address;
    }

    public void connect() {
        if (connectError) {
            throw new ConnectExceptionV4(address, address + " 서버 연결 실패");
        }

        System.out.println(address + " 서버 연결 성공");
    }

    public void send(String data) {
        if (sendError) {
            throw new SendExceptionV4(data, address + " 서버에 데이터 전송 실패: " +
data);
        }

        System.out.println(address + " 서버에 데이터 전송: " + data);
    }

    public void disconnect() {
```

```
        System.out.println(address + " 서버 연결 해제");
    }

    public void initError(String data) {
        if (data.contains("error1")) {
            connectError = true;
        }
        if (data.contains("error2")) {
            sendError = true;
        }
    }
}
```

- 언체크 예외이므로 `throws`를 사용하지 않는다.

```
package exception.ex4;

public class NetworkServiceV4 {

    public void sendMessage(String data) {
        String address = "https://example.com";

        NetworkClientV4 client = new NetworkClientV4(address);
        client.initError(data);

        try {
            client.connect();
            client.send(data);
        } finally {
            client.disconnect();
        }

    }
}
```

- `NetworkServiceV4`는 발생하는 예외인 `ConnectExceptionV4`, `SendExceptionV4`를 잡아도 해당 오류들을 복구할 수 없다. 따라서 예외를 밖으로 던진다.
- 언체크 예외이므로 `throws`를 사용하지 않는다.
- 사실 `NetworkServiceV4` 개발자 입장에서는 해당 예외들을 복구할 수 없다. 따라서 해당 예외들을 생각하지 않는 것이 더 나은 선택일 수 있다. 해결할 수 없는 예외들은 다른 곳에서 공통으로 처리된다.
- 이런 방식 덕분에 `NetworkServiceV4`는 해결할 수 없는 예외 보다는 본인 스스로의 코드에 더 집중할 수 있

다. 따라서 코드가 깔끔해진다.

```
package exception.ex4;

import exception.ex4.exception.SendExceptionV4;

import java.util.Scanner;

public class MainV4 {

    public static void main(String[] args) {

        NetworkServiceV4 networkService = new NetworkServiceV4();

        Scanner scanner = new Scanner(System.in);
        while (true) {
            System.out.print("전송할 문자: ");
            String input = scanner.nextLine();
            if (input.equals("exit")) {
                break;
            }

            try {
                networkService.sendMessage(input);
            } catch (Exception e) { // 모든 예외를 잡아서 처리
                exceptionHandler(e);
            }
            System.out.println();
        }
        System.out.println("프로그램을 정상 종료합니다.");
    }

    //공통 예외 처리
    private static void exceptionHandler(Exception e) {
        //공통 처리
        System.out.println("사용자 메시지: 죄송합니다. 알 수 없는 문제가 발생했습니다.");
        System.out.println("==개발자용 디버깅 메시지==");
        e.printStackTrace(System.out); // 스택 트레이스 출력
        //e.printStackTrace(); // System.err에 스택 트레이스 출력

        //필요하면 예외 별로 별도의 추가 처리 가능
        if (e instanceof SendExceptionV4 sendEx) {
```

```
            System.out.println("[전송 오류] 전송 데이터: " + sendEx.getSendData());
        }
    }
}
```

공통 예외 처리

여기에 예외를 공통으로 처리하는 부분이 존재한다.

```
try {
    networkService.sendMessage(input);
} catch (Exception e) { // 모든 예외를 잡아서 처리
    exceptionHandler(e);
}
```

- `Exception`을 잡아서 지금까지 해결하지 못한 모든 예외를 여기서 공통으로 처리한다. `Exception`을 잡으면 필요한 모든 예외를 잡을 수 있다.
- 예외도 객체이므로 공통 처리 메서드인 `exceptionHandler(e)`에 예외 객체를 전달한다.

exceptionHandler()

- 해결할 수 없는 예외가 발생하면 사용자에게는 시스템 내에 알 수 없는 문제가 발생했다고 알리는 것이 좋다.
  - 사용자가 디테일한 오류 코드나 오류 상황까지 모두 이해할 필요는 없다. 예를 들어서 사용자는 데이터베이스 연결이 안되서 오류가 발생한 것인지, 네트워크에 문제가 있어서 오류가 발생한 것인지 알 필요는 없다.
- 개발자는 빨리 문제를 찾고 디버깅 할 수 있도록 오류 메시지를 남겨두어야 한다.
- 예외도 객체이므로 필요하면 `instanceof`와 같이 예외 객체의 타입을 확인해서 별도의 추가 처리를 할 수 있다.

e.printStackTrace()

- 예외 메시지와 스택 트레이스를 출력할 수 있다.
- 이 기능을 사용하면 예외가 발생한 지점을 역으로 추적할 수 있다.
- 참고로 예제에서는 `e.printStackTrace(System.out)`을 사용해서 표준 출력으로 보냈다.
- `e.printStackTrace()`를 사용하면 `System.err`이라는 표준 오류에 결과를 출력한다.
  - IDE에서는 `System.err`로 출력하면 출력 결과를 빨간색으로 보여준다.
  - 일반적으로 이 방법을 사용한다.

> **참고**: `System.out`, `System.err` 둘다 결국 콘솔에 출력되지만, 서로 다른 흐름을 통해서 출력된다. 따라서 둘을 함께 사용하면 출력 순서를 보장하지 않는다. 출력 순서가 꼬여서 보일 수 있다.

> **참고**: 실무에서는 `System.out`이나 `System.err`을 통해 콘솔에 무언가를 출력하기 보다는, 주로 Slf4J, logback 같은 별도의 로그 라이브러리를 사용해서 콘솔과 특정 파일에 함께 결과를 출력한다. 그런데 `e.printStackTrace()`를 직접 호출하면 결과가 콘솔에만 출력된다. 이렇게 되면 서버에서 로그를 확인하기 어렵다. 서버에서는 파일로 로그를 확인해야 한다. 따라서 콘솔에 바로 결과를 출력하는 `e.printStackTrace()`는 잘 사용하지 않는다. 대신에 로그 라이브러리를 통해서 예외 스택 트레이스를 출력한다. 지금은 로그 라이브러리라는 것이 있다는 정도만 알아두자. 학습 단계에서는 `e.printStackTrace()`를 적극 사용해도 괜찮다.

실행 결과

```
전송할 문자: hello
https://example.com 서버 연결 성공
https://example.com 서버에 데이터 전송: hello
https://example.com 서버 연결 해제

전송할 문자: error1
https://example.com 서버 연결 해제
사용자 메시지: 죄송합니다. 알 수 없는 문제가 발생했습니다.
==개발자용 디버깅 메시지==
exception.ex4.exception.ConnectExceptionV4: https://example.com 서버 연결 실패
    at exception.ex4.NetworkClientV4.connect(NetworkClientV4.java:18)
    at exception.ex4.NetworkServiceV4.sendMessage(NetworkServiceV4.java:12)
    at exception.ex4.MainV4.main(MainV4.java:22)

전송할 문자: error2
https://example.com 서버 연결 성공
https://example.com 서버 연결 해제
사용자 메시지: 죄송합니다. 알 수 없는 문제가 발생했습니다.
==개발자용 디버깅 메시지==
exception.ex4.exception.SendExceptionV4: https://example.com 서버에 데이터 전송 실
패: error2
    at exception.ex4.NetworkClientV4.send(NetworkClientV4.java:26)
    at exception.ex4.NetworkServiceV4.sendMessage(NetworkServiceV4.java:13)
    at exception.ex4.MainV4.main(MainV4.java:22)
[전송 오류] 전송 데이터: error2

전송할 문자: exit
프로그램을 정상 종료합니다.
```

실행 결과를 보면 공통 예외 처리가 잘 적용된 것을 확인할 수 있다. 추가로 "error2"의 경우 `SendExceptionV4`이

발생하면서 `[전송 오류]...`라는 추가 로그가 남은 것도 확인할 수 있다.

try-with-resources

애플리케이션에서 외부 자원을 사용하는 경우 반드시 외부 자원을 해제해야 한다.
따라서 `finally` 구문을 반드시 사용해야 한다.

```
try {
  //정상 흐름
} catch  {
  //예외 흐름
} finally {
  //반드시 호출해야 하는 마무리 흐름
}
```

`try`에서 외부 자원을 사용하고, `try`가 끝나면 외부 자원을 반납하는 패턴이 반복되면서 자바에서는 Try with resources라는 편의 기능을 자바 7에서 도입했다. 이름 그대로 `try`에서 자원을 함께 사용한다는 뜻이다. 여기서 자원은 `try`가 끝나면 반드시 종료해서 반납해야 하는 외부 자원을 뜻한다.

이 기능을 사용하려면 먼저 `AutoCloseable` 인터페이스를 구현해야 한다.

```
package java.lang;

public interface AutoCloseable {
    void close() throws Exception;
}
```

이 인터페이스를 구현하면 Try with resources를 사용할 때 `try`가 끝나는 시점에 `close()`가 자동으로 호출된다.

그리고 다음과 같이 Try with resources 구문을 사용하면 된다.

```
try (Resource resource = new Resource()) {
    // 리소스를 사용하는 코드
}
```

이제 구현 코드를 만들어보자.

기존 코드를 Try with resources 구문을 사용하도록 변경해보자.

```
package exception.ex4;

import exception.ex4.exception.ConnectExceptionV4;
import exception.ex4.exception.SendExceptionV4;

public class NetworkClientV5 implements AutoCloseable {

    private final String address;
    public boolean connectError;
    public boolean sendError;

    public NetworkClientV5(String address) {
        this.address = address;
    }

    public void connect() {
        if (connectError) {
            throw new ConnectExceptionV4(address, address + " 서버 연결 실패");
        }

        System.out.println(address + " 서버 연결 성공");
    }

    public void send(String data) {
        if (sendError) {
            throw new SendExceptionV4(data, address + " 서버에 데이터 전송 실패: " +
data);
        }

        System.out.println(address + " 서버에 데이터 전송: " + data);
    }

    public void disconnect() {
        System.out.println(address + " 서버 연결 해제");
    }

    public void initError(String data) {
        if (data.contains("error1")) {
            connectError = true;
        }
```

```
        if (data.contains("error2")) {
            sendError = true;
        }
    }

    @Override
    public void close() {
        System.out.println("NetworkClientV5.close");
        disconnect();
    }
}
```

- `implements AutoCloseable`을 통해 `AutoCloseable`을 구현한다.
- **close()**: `AutoCloseable` 인터페이스가 제공하는 이 메서드는 `try`가 끝나면 자동으로 호출된다. 종료 시점에 자원을 반납하는 방법을 여기에 정의하면 된다. 참고로 이 메서드에서 예외를 던지지는 않으므로 인터페이스의 메서드에 있는 `throws Exception`은 제거했다.

```
package exception.ex4;

public class NetworkServiceV5 {

    public void sendMessage(String data) {
        String address = "https://example.com";

        try (NetworkClientV5 client = new NetworkClientV5(address)) {
            client.initError(data);
            client.connect();
            client.send(data);
        } catch (Exception e) {
            System.out.println("[예외 확인]: " + e.getMessage());
            throw e;
        }

    }
}
```

- Try with resources 구문은 `try` 괄호 안에 사용할 자원을 명시한다.
- 이 자원은 `try` 블럭이 끝나면 자동으로 `AutoCloseable.close()`를 호출해서 자원을 해제한다.
- 참고로 여기서 `catch` 블럭 없이 `try` 블럭만 있어도 `close()`는 호출된다.
- 여기서 `catch` 블럭은 단순히 발생한 예외를 잡아서 예외 메시지를 출력하고, 잡은 예외를 `throw`를 사용해서

다시 밖으로 던진다.

MainV4 - 코드 변경

```
public static void main(String[] args) throws NetworkClientExceptionV2 {
    //NetworkServiceV4 networkService = new NetworkServiceV4();
    NetworkServiceV5 networkService = new NetworkServiceV5();
}
```

실행 결과

```
전송할 문자: hello
https://example.com 서버 연결 성공
https://example.com 서버에 데이터 전송: hello
NetworkClientV5.close
https://example.com 서버 연결 해제

전송할 문자: error1
NetworkClientV5.close
https://example.com 서버 연결 해제
[예외 확인]: https://example.com 서버 연결 실패
사용자 메시지: 죄송합니다. 알 수 없는 문제가 발생했습니다.
==개발자용 디버깅 메시지==
exception.ex4.exception.ConnectExceptionV4: https://example.com 서버 연결 실패
    at exception.ex4.NetworkClientV5.connect(NetworkClientV5.java:18)
    at exception.ex4.NetworkServiceV5.sendMessage(NetworkServiceV5.java:10)
    at exception.ex4.MainV4.main(MainV4.java:23)

전송할 문자: error2
https://example.com 서버 연결 성공
NetworkClientV5.close
https://example.com 서버 연결 해제
[예외 확인]: https://example.com 서버에 데이터 전송 실패: error2
사용자 메시지: 죄송합니다. 알 수 없는 문제가 발생했습니다.
==개발자용 디버깅 메시지==
exception.ex4.exception.SendExceptionV4: https://example.com 서버에 데이터 전송 실
패: error2
    at exception.ex4.NetworkClientV5.send(NetworkClientV5.java:26)
    at exception.ex4.NetworkServiceV5.sendMessage(NetworkServiceV5.java:11)
    at exception.ex4.MainV4.main(MainV4.java:23)
[전송 오류] 전송 데이터: error2
```

```
전송할 문자: exit
프로그램을 정상 종료합니다.
```

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 `client` 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.

정리

문제와 풀이

예외 처리 강의는 하나의 예제가 큰 흐름으로 이어진다. 예제를 처음부터 복습하면서 큰 흐름을 스스로 정리해보자.

정리

처음 자바를 설계할 당시에는 체크 예외가 더 나은 선택이라 생각했다. 그래서 자바가 기본으로 제공하는 기능들에는 체크 예외가 많다. 그런데 시간이 흐르면서 복구 할 수 없는 예외가 너무 많아졌다. 특히 라이브러리를 점점 더 많이 사용하면서 처리해야 하는 예외도 더 늘어났다. 라이브러리들이 제공하는 체크 예외를 처리할 수 없을 때마다 `throws`에 예외를 덕지덕지 붙어야 했다. 그래서 개발자들은 `throws Exception`이라는 극단적?인 방법도 자주 사용하게 되었다. 물론 이 방법은 사용하면 안된다. 모든 예외를 던진다고 선언하는 것인데, 결과적으로 어떤 예외를 잡고 어떤 예외를 던지는지 알 수 없기 때문이다. 체크 예외를 사용한다면 잡을 건 잡고 던질 예외는 명확하게 던지도록 선언해야 한다.
체크 예외의 이런 문제점 때문에 최근 라이브러리들은 대부분 런타임 예외를 기본으로 제공한다. 가장 유명한 스프링이나 JPA 같은 기술들도 대부분 언체크(런타임) 예외를 사용한다.
런타임 예외도 필요하면 잡을 수 있기 때문에 필요한 경우에는 잡아서 처리하고, 그렇지 않으면 자연스럽게 던지도록 둔다. 그리고 처리할 수 없는 예외는 예외를 공통으로 처리하는 부분을 만들어서 해결하면 된다.