9. 예외 처리1 - 이론

#0.강의/1.자바로드맵/3.자바-중급1편

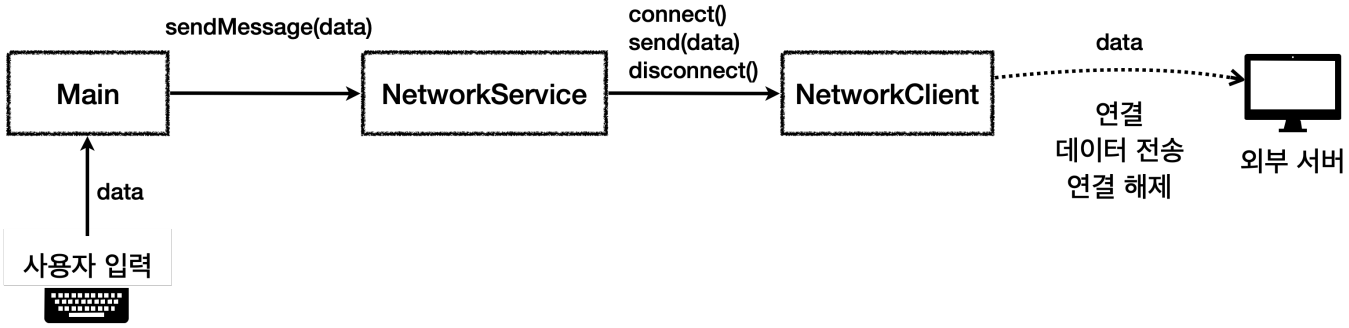
예외 처리가 필요한 이유1 - 시작

예외 처리가 필요한 이유를 알아보기 위해 간단한 예제 프로그램을 먼저 만들어보자.

이 프로그램은 사용자의 입력을 받고, 입력 받은 문자를 외부 서버에 전송하는 프로그램이다.

참고로 아직 네트워크를 학습하지 않았기 때문에 실제 통신하는 코드가 들어가지는 않는다. 그래도 예외 처리가 필요한 상황을 이해하기에는 충분할 것이다.

프로그램 구성도



실행 예시

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

클래스 설명

- `NetworkClient`: 외부 서버와 연결하고, 데이터를 전송하고, 연결을 종료하는 기능을 제공한다.
- `NetworkService`: `NetworkClient`를 사용해서 데이터를 전송한다. `NetworkClient`를 사용하려면 연결, 전송, 연결 종료와 같은 복잡한 흐름을 제어해야 하는데, 이런 부분을 `NetworkService`가 담당한다.
- `Main`: 사용자의 입력을 받는다.
- 전체 흐름: `Main`을 통해 사용자의 입력을 받으면 사용자의 입력을 `NetworkService`에 전달한다. `NetworkService`는 `NetworkClient`를 사용해서 외부 서버에 연결하고, 데이터를 전송하고, 전송이 완료되면 연결을 종료한다.

NetworkService 예시 코드

```
String address = "http://example.com";
NetworkClientV1 client = new NetworkClientV1(address);
client.connect();
client.send(data);
client.disconnect();
```

NetworkClient 사용법

- `connect()`를 먼저 호출해서 서버와 연결한다.
- `send(data)`로 연결된 서버에 메시지를 전송한다.
- `disconnect()`로 연결을 해제한다.

NetworkClient 사용시 주의 사항

- `connect()`가 실패한 경우 `send()`를 호출하면 안된다.
- 사용 후에는 반드시 `disconnect()`를 호출해서 연결을 해제해야 한다.
  - `connect()`, `send()` 호출에 오류가 있어도 `disconnect()`는 반드시 호출해야 한다.

NetworkServiceV0

다음 예제 코드를 만들어보자.

- `NetworkClientV0`
- `NetworkServiceV0`
- `MainV0`

```
package exception.ex0;

public class NetworkClientV0 {
```

```
    private final String address;

    public NetworkClientV0(String address) {
        this.address = address;
    }

    public String connect() {
        //연결 성공
        System.out.println(address + " 서버 연결 성공");
        return "success";
    }

    public String send(String data) {
        //전송 성공
        System.out.println(address + " 서버에 데이터 전송: " + data);
        return "success";
    }

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

}
```

- `String address` : 접속할 외부 서버 주소
- `connect()` : 외부 서버에 연결한다.
- `send(String data)` : 연결한 외부 서버에 데이터를 전송한다.
- `disconnect()` : 외부 서버와 연결을 해제한다.

실제로 외부 네트워크에 접속하는 코드는 없지만, 외부 네트워크와 접속한다고 가정하자.

```
package exception.ex0;

public class NetworkServiceV0 {

    public void sendMessage(String data) {
        String address = "http://example.com";
        NetworkClientV0 client = new NetworkClientV0(address);

        client.connect();
```

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

- `NetworkService`는 복잡한 `NetworkClient` 사용 로직을 처리한다.
- `NetworkClient`를 생성하고, 이때 접속할 외부 서버 주소도 함께 전달한다.
- `NetworkClient`를 사용하는데 필요한 `connect()`, `send(data)`, `disconnect()`를 순서대로 호출한다.

```
package exception.ex0;

import java.util.Scanner;

public class MainV0 {

    public static void main(String[] args) {
        NetworkServiceV0 networkService = new NetworkServiceV0();

        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.");
    }
}
```

- 전송할 문자를 `Scanner`를 통해서 입력 받는다.
- `NetworkService.sendMessage()`를 통해 메시지를 전달한다.
- `exit`를 입력하면 프로그램을 정상 종료한다.

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.
```

예외 처리가 필요한 이유2 - 오류 상황 만들기

외부 서버와 통신할 때는 다음과 같은 다양한 문제들이 발생한다.

- 외부 서버와 연결에 실패한다. (네트워크 오류 등등)
- 데이터 전송에 문제가 발생한다.

물론 실제 외부 서버와 통신하는 것이 아니기 때문에 이런 오류가 발생하지는 않는다.
대신에 오류 상황을 가상으로 시뮬레이션 해보자.

NetworkServiceV1_1

오류 상황을 시뮬레이션 할 수 있는 다음 코드를 만들어보자.

- `NetworkClientV1`
- `NetworkServiceV1_1`
- `MainV1`

오류 상황을 시뮬레이션 하기 위해 사용자의 입력 값을 활용한다.

- **연결 실패**: 사용자가 입력하는 문자에 "error1" 단어가 있으면 연결에 실패한다. 오류 코드는 "connectError" 이다.
- **전송 실패**: 사용자가 입력하는 문자에 "error2" 단어가 있으면 데이터 전송에 실패한다. 오류 코드는 "sendError" 이다.

```
package exception.ex1;

public class NetworkClientV1 {

    private final String address;
```

```
    public boolean connectError;
    public boolean sendError;

    public NetworkClientV1(String address) {
        this.address = address;
    }

    public String connect() {
        if (connectError) {
            System.out.println(address + " 서버 연결 실패");
            return "connectError";
        }

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

    public String send(String data) {
        if (sendError) {
            System.out.println(address + " 서버에 데이터 전송 실패: " + data);
            return "sendError";
        }

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

    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;
        }
    }

}
```

- `NetworkClientV1`에는 `connectError`, `sendError` 필드가 있다.
  - `connectError`: 이 필드의 값이 `true`가 되면 연결에 실패하고, `connectError` 오류 코드를 반환한다.
  - `sendError`: 이 필드의 값이 `true`가 되면 데이터 전송에 실패한다. `sendError` 오류 코드를 반환한다.
  - 문제가 없으면 `success` 코드를 반환한다.
- `initError(String data)`
  - 이 메서드를 통해서 `connectError`, `sendError` 필드의 값을 `true`로 설정할 수 있다.
  - 사용자 입력 값에 "error1"이 있으면 `connectError` 오류가 발생하고, "error2"가 있으면 `sendError` 오류가 발생한다.

```
package exception.ex1;

public class NetworkServiceV1_1 {

    public void sendMessage(String data) {
        String address = "http://example.com";
        NetworkClientV1 client = new NetworkClientV1(address);
        client.initError(data); //추가

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

- `client.initError(data)`: 사용자의 입력 값을 기반으로 오류를 활성화 한다.

```
package exception.ex1;

import java.util.Scanner;

public class MainV1 {

    public static void main(String[] args) {
        NetworkServiceV1_1 networkService = new NetworkServiceV1_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
http://example.com 서버 연결 성공
http://example.com 서버에 데이터 전송: hello
http://example.com 서버 연결 해제

전송할 문자: error1
http://example.com 서버 연결 실패
http://example.com 서버에 데이터 전송: error1
http://example.com 서버 연결 해제

전송할 문자: error2
http://example.com 서버 연결 성공
http://example.com 서버에 데이터 전송 실패: error2
http://example.com 서버 연결 해제

전송할 문자: exit
프로그램을 정상 종료합니다.
```

- **error1 입력**: 서버 연결에 실패한다.
- **error2 입력**: 데이터 전송에 실패한다.

남은 문제

- 연결이 실패하면 데이터를 전송하지 않아야 하는데, 여기서는 데이터를 전송한다.

추가 요구 사항

- 오류가 발생했을 때 어떤 오류가 발생했는지 자세한 내역을 남기면 이후 디버깅에 도움이 될 것이다. 오류 로그를 남겨야 한다.

예외 처리가 필요한 이유3 - 반환 값으로 예외 처리

앞서 발생한 다음 문제들을 해결해보자.

- 연결에 실패하면 데이터를 전송하지 않아야 하는데, 여기서는 데이터를 전송한다.
- 오류 로그를 남겨야 한다.

`NetworkClientV1`은 `connectError`, `sendError`와 같은 오류 코드를 문자열로 반환해주고 있다. 이 반환 값을 사용해서 예외 상황을 처리해보자.

NetworkServiceV1_2

```
package exception.ex1;

public class NetworkServiceV1_2 {

    public void sendMessage(String data) {
        NetworkClientV1 client = new NetworkClientV1("http://example.com");
        client.initError(data);

        String connectResult = client.connect();
        if (isError(connectResult)) {
            System.out.println("[네트워크 오류 발생] 오류 코드: " + connectResult);
            return;
        }

        String sendResult = client.send(data);
        if (isError(sendResult)) {
            System.out.println("[네트워크 오류 발생] 오류 코드: " + sendResult);
            return;
        }

        client.disconnect();
```

```
    }

    private static boolean isError(String resultCode) {
        return !resultCode.equals("success");
    }

}
```

- `NetworkService`는 `NetworkClient`를 사용하는 전체 흐름을 관리한다.
- 오류가 발생한 경우 오류 코드를 출력으로 남긴다.
- 오류가 발생한 경우 프로그램이 더 이상 진행되지 않도록 `return`을 사용해서 중지한다. 따라서 연결에 실패하면 데이터를 전송하는 문제가 해결된다.

MainV1 - 코드 변경

```
public static void main(String[] args) {
    //NetworkServiceV1_1 networkService = new NetworkServiceV1_1();
    NetworkServiceV1_2 networkService = new NetworkServiceV1_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
http://example.com 서버 연결 실패
[네트워크 오류 발생] 오류 코드: connectError

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

전송할 문자: exit
프로그램을 정상 종료합니다.
```

잠시 돌아가서 앞서 이야기한 주의 사항을 다시 살펴보자.

NetworkClient 사용시 주의 사항

- `connect()`가 실패한 경우 `send()`를 호출하면 안된다. ➜ **해결**
- 사용 후에는 반드시 `disconnect()`를 호출해서 연결을 해제해야 한다. ➜ **해결 안됨**
  - `connect()`, `send()` 호출에 오류가 있어도 `disconnect()`는 반드시 호출해야 한다.

`connect()`가 실패한 경우 `send()`를 호출하면 안되는 부분은 해결되었다. 하지만 사용 후에는 `disconnect()`를 반드시 호출해야 하는 문제는 해결되지 않았다. `error2`를 보면 데이터 전송에 실패하는 경우, 연결이 해제 되지 않는다. 계속 이렇게 두면 네트워크 연결 자원이 고갈될 수 있다.

> **참고**: 자바의 경우 GC가 있기 때문에 JVM 메모리에 있는 인스턴스는 자동으로 해제할 수 있다. 하지만 외부 연결과 같은 자바 외부의 자원은 자동으로 해제가 되지 않는다. 따라서 외부 자원을 사용한 후에는 연결을 해제해서 외부 자원을 반드시 반납해야 한다.

NetworkServiceV1_3

이번에는 `disconnect()`를 반드시 호출하도록 코드를 작성해보자.

```
package exception.ex1;

public class NetworkServiceV1_3 {

    public void sendMessage(String data) {
        NetworkClientV1 client = new NetworkClientV1("http://example.com");
        client.initError(data);

        String connectResult = client.connect();
        if (isError(connectResult)) {
            System.out.println("[네트워크 오류 발생] 오류 코드: " + connectResult);
        } else {
            String sendResult = client.send(data);
            if (isError(sendResult)) {
                System.out.println("[네트워크 오류 발생] 오류 코드: " + sendResult);
            }
        }
```

```
            client.disconnect();
        }

        private static boolean isError(String resultCode) {
            return !resultCode.equals("success");
        }

    }
```

- 프로그램에서 `return` 문을 제거하고 `if` 문으로 적절한 분기를 사용했다.
- `connect()` 에 성공해서 오류가 없는 경우에만 `send()` 를 호출한다.
- 중간에 `return` 하지 않으므로 마지막에 있는 `disconnect()` 를 호출할 수 있다.
  - 연결에 실패해도 `disconnect()` 를 호출한다.
  - 데이터 전송에 실패해도 `disconnect()` 를 호출한다.

MainV1 - 코드 변경

```
public static void main(String[] args) {
    NetworkServiceV1_3 networkService = new NetworkServiceV1_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
http://example.com 서버 연결 실패
[네트워크 오류 발생] 오류 코드: connectError
http://example.com 서버 연결 해제

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

```
전송할 문자: exit
프로그램을 정상 종료합니다.
```

문제들이 해결되었는지 확인해보자.

NetworkClient 사용시 주의 사항

- `connect()`가 실패한 경우 `send()`를 호출하면 안된다. ➜ **해결**
- 사용 후에는 반드시 `disconnect()`를 호출해서 연결을 해제해야 한다. ➜ **해결**
  - `connect()`, `send()` 호출에 오류가 있어도 `disconnect()`는 반드시 호출해야 한다.

정상 흐름과 예외 흐름

그런데 반환 값으로 예외를 처리하는 `NetworkServiceV1_2`, `NetworkServiceV1_3`와 같은 코드들을 보면 정상 흐름과 예외 흐름이 전혀 분리되어 있지 않다. 어떤 부분이 정상 흐름이고 어떤 부분이 예외 흐름인지 이해하기가 너무 어렵다. 심지어 예외 흐름을 처리하는 부분이 더 많다.

정상 흐름 코드

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

- 정상 흐름은 연결하고, 데이터를 전송하고, 연결을 종료하면 끝이다. 매우 단순하고 직관적이다.

정상 흐름 + 예외 흐름이 섞여 있는 코드

NetworkServiceV1_2 코드

```
String connectResult = client.connect();
if (isError(connectResult)) {
    System.out.println("[네트워크 오류 발생] 오류 코드: " + connectResult);
    return;
}

String sendResult = client.send(data);
if (isError(sendResult)) {
    System.out.println("[네트워크 오류 발생] 오류 코드: " + sendResult);
    return;
}
```

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

NetworkServiceV1_3 코드

```
String connectResult = client.connect();
if (isError(connectResult)) {
    System.out.println("[네트워크 오류 발생] 오류 코드: " + connectResult);
} else {
    String sendResult = client.send(data);
    if (isError(sendResult)) {
        System.out.println("[네트워크 오류 발생] 오류 코드: " + sendResult);
    }
}

client.disconnect();
```

- 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.

어떻게 하면 정상 흐름과 예외 흐름을 분리할 수 있을까? 지금과 같이 반환 값을 사용해서 예외 상황을 처리하는 방식으로는 해결할 수 없는 것은 확실하다.
이런 문제를 해결하기 위해 바로 예외 처리 메커니즘이 존재한다. 지금부터 본격적으로 자바 예외 처리에 대해 알아보자. 예외 처리를 사용하면 정상 흐름과 예외 흐름을 명확하게 분리할 수 있다.

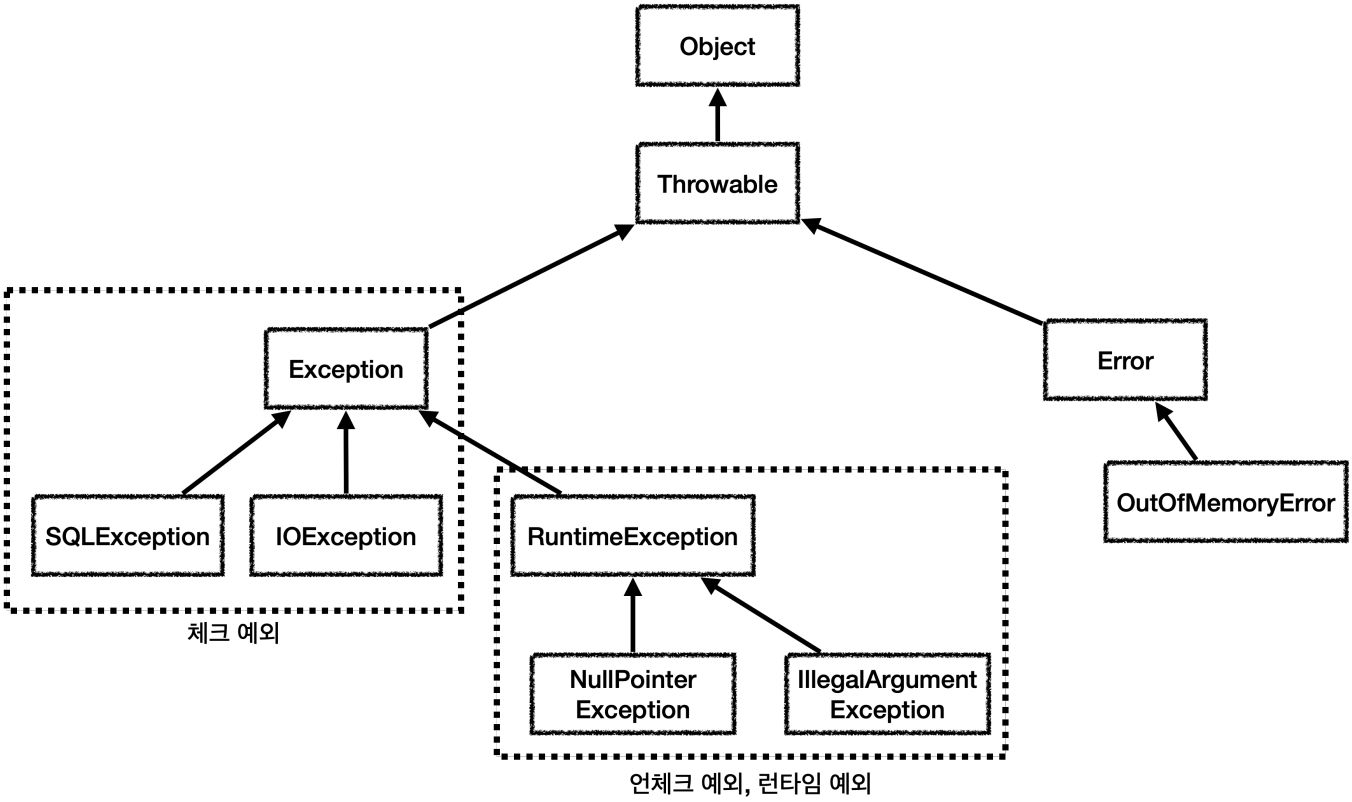
자바 예외 처리1 - 예외 계층

자바는 프로그램 실행 중에 발생할 수 있는 예상치 못한 상황, 즉 예외(Exception)를 처리하기 위한 메커니즘을 제공한다. 이는 프로그램의 안정성과 신뢰성을 높이는 데 중요한 역할을 한다.

자바의 예외 처리는 다음 키워드를 사용한다.
`try`, `catch`, `finally`, `throw`, `throws`

그리고 예외를 다루기 위한 예외 처리용 객체들을 제공한다.

예외 계층 그림



- `Object`: 자바에서 기본형을 제외한 모든 것은 객체다. 예외도 객체이다. 모든 객체의 최상위 부모는 `Object` 이므로 예외의 최상위 부모도 `Object` 이다.
- `Throwable`: 최상위 예외이다. 하위에 `Exception` 과 `Error` 가 있다.
- `Error`: 메모리 부족이나 심각한 시스템 오류와 같이 애플리케이션에서 복구가 불가능한 시스템 예외이다. 애플리케이션 개발자는 이 예외를 잡으려고 해서는 안된다.
- `Exception`: 체크 예외
  - 애플리케이션 로직에서 사용할 수 있는 실질적인 최상위 예외이다.
  - `Exception` 과 그 하위 예외는 모두 컴파일러가 체크하는 체크 예외이다. 단 `RuntimeException` 은 예외로 한다.
- `RuntimeException`: 언체크 예외, 런타임 예외
  - 컴파일러가 체크 하지 않는 언체크 예외이다.
  - `RuntimeException` 과 그 자식 예외는 모두 언체크 예외이다.
  - `RuntimeException` 의 이름을 따라서 `RuntimeException` 과 그 하위 언체크 예외를 **런타임 예외**라고 많이 부른다. 여기서도 앞으로는 런타임 예외로 종종 부르겠다.

체크 예외 vs 언체크 예외(런타임 예외)

체크 예외는 발생한 예외를 개발자가 명시적으로 처리해야 한다. 그렇지 않으면 컴파일 오류가 발생한다. 언체크 예외는 개발자가 발생한 예외를 명시적으로 처리하지 않아도 된다. 자세한 내용은 조금 뒤에서 코드로 알아보자.

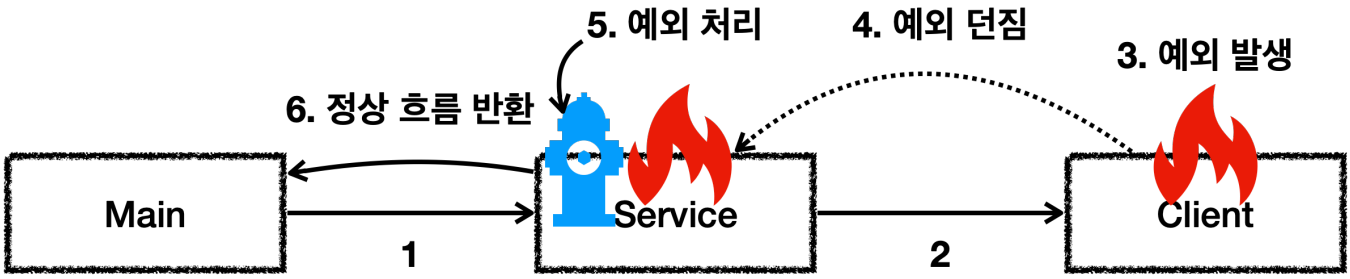
주의

상속 관계에서 부모 타입은 자식을 담을 수 있다. 이 개념이 예외 처리에도 적용되는데, 상위 예외를 `catch`로 잡으면 그 하위 예외까지 함께 잡는다. 따라서 애플리케이션 로직에서는 `Throwable` 예외를 잡으면 안되는데, 앞서 이야기한 잡으면 안되는 `Error` 예외도 함께 잡을 수 있기 때문이다. 애플리케이션 로직은 이런 이유로 `Exception`부터 필요한 예외로 생각하고 잡으면 된다.

자바 예외 처리2 - 예외 기본 규칙

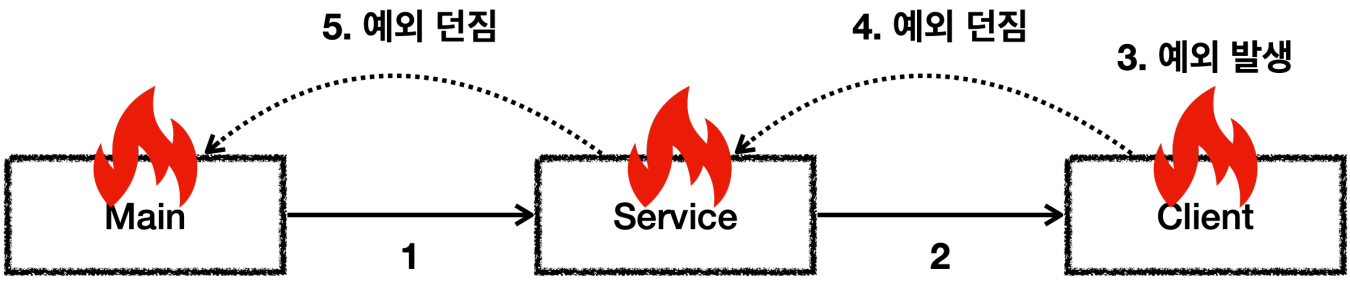
예외는 폭탄 돌리기와 같다. 예외가 발생하면 잡아서 처리하거나, 처리할 수 없으면 밖으로 던져야한다.

예외 처리



- 1: `Main`은 `Service`를 호출한다.
- 2: `Service`는 `Client`를 호출한다.
- 3: `Client`에서 예외가 발생했다.
- 4: `Client`에서 예외를 처리하지 못하고 밖으로 던진다. 여기서 `Client`의 밖은 `Client`를 호출한 `Service`를 뜻한다.
- 5: `Service`에 예외가 전달된다. `Service`에서 예외를 처리했다. 이후에는 애플리케이션 로직이 정상 흐름으로 동작한다.
- 6: 정상 흐름을 반환한다.

예외 던짐



예외를 처리하지 못하면 자신을 호출한 곳으로 예외를 던져야 한다.

예외에 대해서는 2가지 기본 규칙을 기억하자.

1. 예외는 잡아서 처리하거나 밖으로 던져야 한다.
2. 예외를 잡거나 던질 때 지정한 예외뿐만 아니라 그 예외의 자식들도 함께 처리할 수 있다.
    - 예를 들어서 `Exception`을 `catch`로 잡으면 그 하위 예외들도 모두 잡을 수 있다.
    - 예를 들어서 `Exception`을 `throws`로 던지면 그 하위 예외들도 모두 던질 수 있다.

참고: 예외를 처리하지 못하고 계속 던지면 어떻게 될까?

- 자바 `main()` 밖으로 예외를 던지면 예외 로그를 출력하면서 시스템이 종료된다.

이제 본격적으로 코드를 통해 예외 처리를 알아보자.

자바 예외 처리3 - 체크 예외

- `Exception`과 그 하위 예외는 모두 컴파일러가 체크하는 체크 예외이다. 단 `RuntimeException`은 예외로 한다.
- 체크 예외는 잡아서 처리하거나, 또는 밖으로 던지도록 선언해야한다. 그렇지 않으면 컴파일 오류가 발생한다.

체크 예외 전체 코드

먼저 전체 코드를 작성하고 그 다음에 하나씩 설명하겠다.

```
package exception.basic.checked;

/**
 * Exception을 상속받은 예외는 체크 예외가 된다.
 */
public class MyCheckedException extends Exception {
    public MyCheckedException(String message) {
        super(message);
    }
}
```

- 예외 클래스를 만들려면 예외를 상속 받으면 된다.
- `Exception`을 상속받은 예외는 체크 예외가 된다.

```
package exception.basic.checked;
```

```
public class Client {
    public void call() throws MyCheckedException {
        throw new MyCheckedException("ex");
    }
}
```

- `throw 예외`라고 하면 새로운 예외를 발생시킬 수 있다. 예외도 객체이기 때문에 객체를 먼저 `new`로 생성하고 예외를 발생시켜야 한다.
- `throws 예외`는 발생시킨 예외를 메서드 밖으로 던질 때 사용하는 키워드이다.
- `throw`, `throws`의 차이에 주의하자

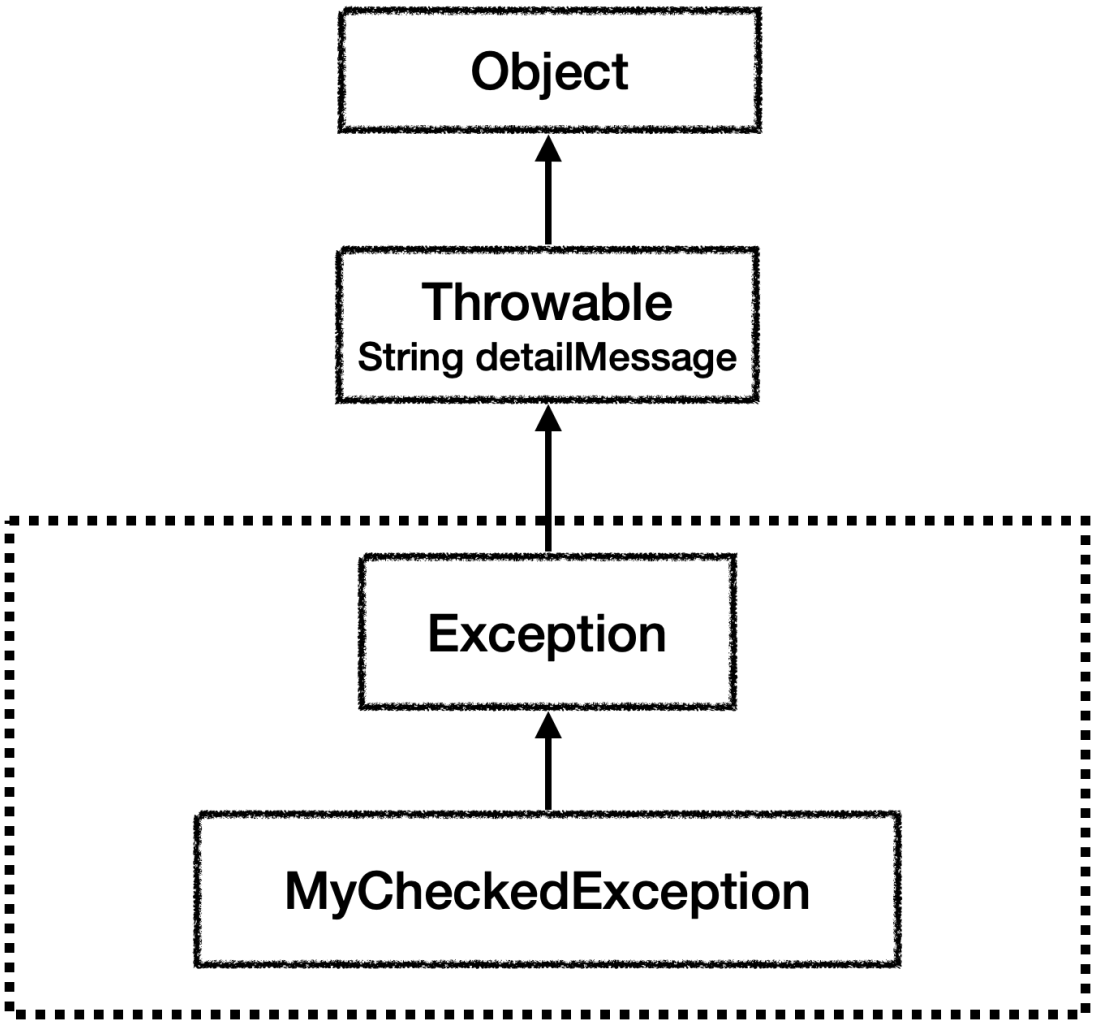
```
package exception.basic.checked;

/**
 * Checked 예외는
 * 예외를 잡아서 처리하거나, 던지거나 둘중 하나를 필수로 선택해야 한다.
 */
public class Service {
    Client client = new Client();

    /**
     * 예외를 잡아서 처리하는 코드
     */
    public void callCatch() {
        try {
            client.call();
        } catch (MyCheckedException e) {
            //예외 처리 로직
            System.out.println("예외 처리, message=" + e.getMessage());
        }
        System.out.println("정상 흐름");
    }

    /**
     * 체크 예외를 밖으로 던지는 코드
     * 체크 예외는 예외를 잡지 않고 밖으로 던지려면 throws 예외를 메서드에 필수로 선언해야한다.
     */
    public void callThrow() throws MyCheckedException {
        client.call();
    }
```

```
}
```

Exception을 상속받은 예외는 체크 예외가 된다.

```
public class MyCheckedException extends Exception {
    public MyCheckedException(String message) {
        super(message);
    }
}
```

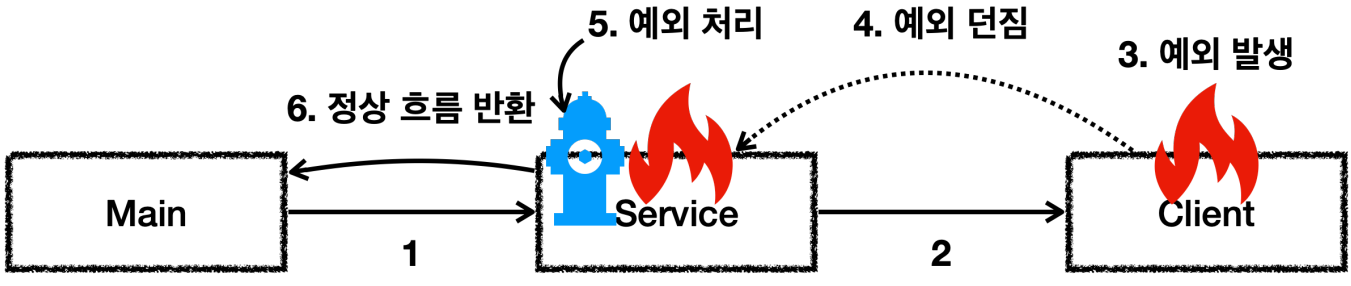
- `MyCheckedException`는 `Exception`을 상속받았다. `Exception`을 상속받으면 체크 예외가 된다.
- 참고로 `RuntimeException`을 상속받으면 언체크 예외가 된다. 이런 규칙은 자바 언어에서 문법으로 정한 것이다.
- 예외가 제공하는 기본 기능이 있는데, 그 중에 오류 메시지를 보관하는 기능도 있다. 예제에서 보는 것 처럼 생성자를 통해서 해당 기능을 그대로 사용하면 편리하다.



체크 예외

- `super(message)`로 전달한 메시지는 `Throwable`에 있는 `detailMessage`에 보관된다.
- `getMessage()`를 통해 조회할 수 있다.

예외를 잡아서 처리



먼저 그림과 같이 예외를 잡아서 처리하는 코드를 실행해보자.

```
package exception.basic.checked;
```

```
public class CheckedCatchMain {

    public static void main(String[] args) {
        Service service = new Service();
        service.callCatch();
        System.out.println("정상 종료");
    }
}
```

실행 결과

```
예외 처리, message=ex
정상 흐름
정상 종료
```

- `service.callCatch()`에서 예외를 처리했기 때문에 `main()` 메서드까지 예외가 올라오지 않는다.
- `main()`에서 출력하는 "정상 종료" 문구가 출력된 것을 확인할 수 있다.

실행 순서를 분석해보자.

1. `main()` ➜ `service.callCatch()` ➜ `client.call()` **[예외 발생, 던짐]**
2. `main()` ← `service.callCatch()` **[예외 처리]** ← `client.call()`
3. `main()` **[정상 흐름]** ← `service.callCatch()` ← `client.call()`

`Client.call()`에서 `MyCheckedException`예외가 발생하고, 그 예외를 `Service.callCatch()`에서 잡는 것을 확인할 수 있다.

```
public void callCatch() {
    try {
        client.call();
    } catch (MyCheckedException e) {
        //예외 처리 로직
        System.out.println("예외 처리, message=" + e.getMessage());
    }
    System.out.println("정상 흐름");
}
```

실행 결과

```
예외 처리, message=ex
System.out.println("정상 흐름");
```

- 이 부분의 실행 결과를 보면 `catch` 부분이 작동한 것을 확인할 수 있다.
- `catch`로 예외를 잡아서 처리하고 나면 코드가 `catch` 블럭의 다음 라인으로 넘어가서 정상 흐름으로 작동한다.

체크 예외를 잡아서 처리하는 코드 - callCatch()

```
try {
    client.call();
} catch (MyCheckedException e) {
    //예외 처리 로직
}
System.out.println("정상 흐름");
```

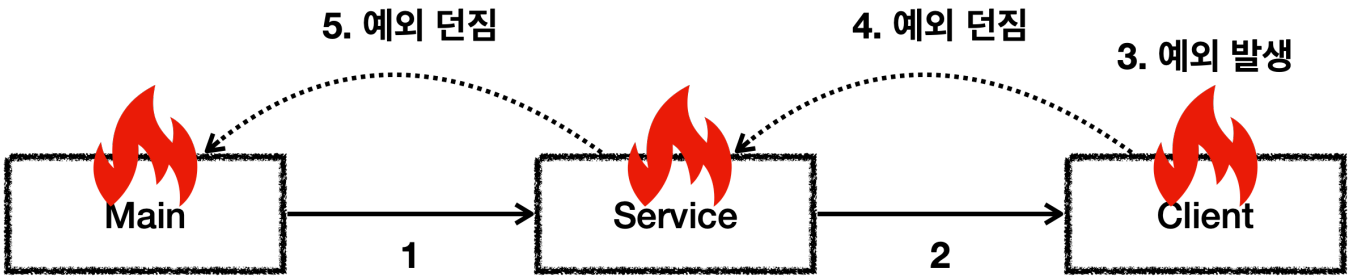
- 예외를 잡아서 처리하려면 `try ~ catch(..)`를 사용해서 예외를 잡으면 된다.
- `try` 코드 블럭에서 발생하는 예외를 잡아서 `catch`로 넘긴다.
  - 만약 `try`에서 잡은 예외가 `catch`의 대상에 없으면 예외를 잡을 수 없다. 이때는 예외를 밖으로 던져야 한다.
- 여기서는 `MyCheckedException` 예외를 `catch`로 잡아서 처리한다.

catch는 해당 타입과 그 하위 타입을 모두 잡을 수 있다

```
public void callCatch() {
    try {
        client.call();
    } catch (Exception e) {
        //예외 처리 로직
    }
    System.out.println("정상 흐름");
}
```

- `catch`에 `MyCheckedException`의 상위 타입인 `Exception`을 적어주어도 `MyCheckedException`을 잡을 수 있다.
- `catch`에 예외를 지정하면 해당 예외와 그 하위 타입 예외를 모두 잡아준다.
- 물론 정확하게 `MyCheckedException`만 잡고 싶다면 `catch`에 `MyCheckedException`을 적어주어야 한다.
- 예외도 객체이기 때문에 다형성이 적용된다.

예외를 처리 하지 않고 던지기



이번에는 예외를 처리하지 않고, 밖으로 던지는 코드를 살펴보자.

```
package exception.basic.checked;

public class CheckedThrowMain {

    public static void main(String[] args) throws MyCheckedException {
        Service service = new Service();
        service.callThrow();
        System.out.println("정상 종료");
    }
}
```

실행 결과

```
Exception in thread "main" exception.basic.checked.MyCheckedException: ex
    at exception.basic.checked.Client.call(Client.java:5)
    at exception.basic.checked.Service.callThrow(Service.java:28)
    at exception.basic.checked.CheckedThrowMain.main(CheckedThrowMain.java:7)
```

- `Service.callThrow()` 안에서 예외를 처리하지 않고, 밖으로 던졌기 때문에 예외가 `main()` 메서드까지 올라온다.
- `main()` 의 `service.callThrow()` 를 호출하는 곳에서 예외를 잡아서 처리하지 못했기 때문에 여기서 예외가 `main()` 밖으로 던져진다. 따라서 `main()` 에 있는 `service.callThrow()` 메서드 다음에 있는 "정상 종료"가 출력되지 않는다.
- 예외가 `main()` 밖으로 던져지면 예외 정보와 스택 트레이스(Stack Trace)를 출력하고 프로그램이 종료된다.
  - 스택 트레이스 정보를 활용하면 예외가 어디서 발생했는지, 그리고 어떤 경로를 거쳐서 넘어왔는지 확인할

수 있다.

실행 순서를 분석해보자.

1. `main()` ➡ `service.callThrow()` ➡ `client.call()` **[예외 발생, 던짐]**
2. `main()` ⬅ `service.callThrow()` **[예외 던짐]** ⬅ `client.call()`
3. `main()` **[예외 던짐]** ⬅ `service.callThrow()` ⬅ `client.call()`

체크 예외를 밖으로 던지는 코드

```
public void callThrow() throws MyCheckedException {
    client.call();
}
```

- 체크 예외를 처리할 수 없을 때는 `throws` 키워드를 사용해서, `method() throws 예외`와 같이 밖으로 던질 예외를 필수로 지정해주어야 한다. 여기서는 `MyCheckedException`을 밖으로 던지도록 지정해주었다.

체크 예외를 밖으로 던지지 않으면 컴파일 오류 발생

```
public void callThrow() {
    client.call();
}
```

- `throws`를 지정하지 않으면 컴파일 오류가 발생한다.
  - `java: unreported exception exception.basic.checked.MyCheckedException; must be caught or declared to be thrown`
  - `client.call()`을 보면 `throws MyCheckedException`가 선언되어 있다. 따라서 `client.call()`을 호출하는 쪽에서 예외를 잡아서 처리하든, 던지든 선택해야 한다. 참고로 `MyCheckedException`는 체크 예외이다.
- 체크 예외는 잡아서 직접 처리하거나 또는 밖으로 던지거나 둘중 하나를 개발자가 직접 명시적으로 처리해야한다.
- 체크 예외는 `try ~ catch`로 잡아서 처리하거나 또는 `throws`를 지정해서 예외를 밖으로 던진다는 선언을 필수로 해주어야 한다.

참고로 체크 예외를 밖으로 던지는 경우에도 해당 타입과 그 하위 타입을 모두 던질 수 있다

```
public void callThrow() throws Exception {
    client.call();
}
```

- `throws`에 `MyCheckedException`의 상위 타입인 `Exception`을 적어주어도 `MyCheckedException`을

던질 수 있다.
- `throws`에 지정한 타입과 그 하위 타입 예외를 밖으로 던진다.
- 물론 정확하게 `MyCheckedException`만 밖으로 던지고 싶다면 `throws`에 `MyCheckedException`을 적어주어야 한다.
- 예외도 객체이기 때문에 다형성이 적용된다.

체크 예외의 장단점

체크 예외는 예외를 잡아서 처리할 수 없을 때, 예외를 밖으로 던지는 `throws 예외`를 필수로 선언해야 한다. 그렇지 않으면 컴파일 오류가 발생한다. 이것 때문에 장점과 단점이 동시에 존재한다.

- **장점**: 개발자가 실수로 예외를 누락하지 않도록 컴파일러를 통해 문제를 잡아주는 훌륭한 안전 장치이다. 이를 통해 개발자는 어떤 체크 예외가 발생하는지 쉽게 파악할 수 있다.
- **단점**: 하지만 실제로는 개발자가 모든 체크 예외를 반드시 잡거나 던지도록 처리해야 하기 때문에, 너무 번거로운 일이 된다. 크게 신경쓰고 싶지 않은 예외까지 모두 챙겨야 한다.

정리

체크 예외는 잡아서 직접 처리하거나 또는 밖으로 던지거나 둘중 하나를 개발자가 직접 명시적으로 처리해야한다. 그렇지 않으면 컴파일 오류가 발생한다.

자바 예외 처리4 - 언체크 예외

- `RuntimeException`과 그 하위 예외는 언체크 예외로 분류된다.
- 언체크 예외는 말 그대로 컴파일러가 예외를 체크하지 않는다는 뜻이다.
- 언체크 예외는 체크 예외와 기본적으로 동일하다. 차이가 있다면 예외를 던지는 `throws`를 선언하지 않고, 생략할 수 있다. 생략한 경우 자동으로 예외를 던진다.

체크 예외 VS 언체크 예외

- **체크 예외**: 예외를 잡아서 처리하지 않으면 항상 `throws` 키워드를 사용해서 던지는 예외를 선언해야 한다.
- **언체크 예외**: 예외를 잡아서 처리하지 않아도 `throws` 키워드를 생략할 수 있다.

언체크 예외 전체 코드

먼저 전체 코드를 작성하고 그 다음에 하나씩 설명하겠다.

```
```

```java
package exception.basic.unchecked;

/**
 * RuntimeException을 상속받은 예외는 언체크 예외가 된다.
 */
public class MyUncheckedException extends RuntimeException {
    public MyUncheckedException(String message) {
        super(message);
    }
}
```

```java
package exception.basic.unchecked;

public class Client {
    public void call() {
        throw new MyUncheckedException("ex");
    }
}
```

```java
package exception.basic.unchecked;

/**
 * UnChecked 예외는
 * 예외를 잡거나, 던지지 않아도 된다.
 * 예외를 잡지 않으면 자동으로 밖으로 던진다.
 */
public class Service {

    Client client = new Client();

    /**
     * 필요한 경우 예외를 잡아서 처리하면 된다.
     */
    public void callCatch() {
        try {
            client.call();
        } catch (MyUncheckedException e) {
            //예외 처리 로직
            System.out.println("예외 처리, message=" + e.getMessage());
```

```
        }
        System.out.println("정상 로직");
    }

    /**
     * 예외를 잡지 않아도 된다. 자연스럽게 상위로 넘어간다.
     * 체크 예외와 다르게 throws 예외 선언을 하지 않아도 된다.
     */
    public void callThrow() {
        client.call();
    }
}
```

체크 예외와 실행 결과는 완전히 동일하다.

```
package exception.basic.unchecked;

public class UncheckedCatchMain {

    public static void main(String[] args) {
        Service service = new Service();
        service.callCatch();
        System.out.println("정상 종료");
    }
}
```

실행 결과

```
예외 처리, message=ex
정상 로직
정상 종료
```

```
package exception.basic.unchecked;

public class UncheckedThrowMain {

    public static void main(String[] args) {
        Service service = new Service();
```

```
        service.callThrow();
        System.out.println("정상 종료");
    }
}
```

실행 결과

```
Exception in thread "main" exception.basic.unchecked.MyUncheckedException: ex
    at exception.basic.unchecked.Client.call(Client.java:5)
    at exception.basic.unchecked.Service.callThrow(Service.java:29)
    at
exception.basic.unchecked.UncheckedThrowMain.main(UncheckedThrowMain.java:7)
```

언체크 예외를 잡아서 처리하는 코드 - callCatch()

```
try {
    client.call();
} catch (MyUncheckedException e) {
    //예외 처리 로직
    log.info("error", e);
}
System.out.println("정상 로직");
```

- 언체크 예외도 필요한 경우 예외를 잡아서 처리할 수 있다.

언체크 예외를 밖으로 던지는 코드

```
public void callThrow() {
    client.call();
}
```

- 언체크 예외는 체크 예외와 다르게 `throws 예외`를 선언하지 않아도 된다.
- 말 그대로 **컴파일러가 이런 부분을 체크하지 않기 때문에 언체크 예외**이다.

언체크 예외를 밖으로 던지는 코드 - 선언

```
public class Client {
```

```
    public void call() throws MyUncheckedException {
        throw new MyUncheckedException("ex");
    }
}
```

- 참고로 언체크 예외도 `throws 예외`를 선언해도 된다. 물론 생략할 수 있다.
- 언체크 예외는 주로 생략하지만, 중요한 예외의 경우 이렇게 선언해두면 해당 코드를 호출하는 개발자가 이런 예외가 발생한다는 점을 IDE를 통해 좀 더 편리하게 인지할 수 있다. (언체크 예외를 던진다고 선언한다고 해서 체크 예외 처럼 컴파일러를 통해서 체크할 수 있는 것은 아니다. 해당 메서드를 호출하는 개발자가 IDE를 통해 호출 코드를 보고, 이런 예외가 발생한다고 인지할 수 있는 정도이다.)

언체크 예외의 장단점

언체크 예외는 예외를 잡아서 처리할 수 없을 때, 예외를 밖으로 던지는 `throws 예외`를 생략할 수 있다. 이것 때문에 장점과 단점이 동시에 존재한다.

- **장점**: 신경쓰고 싶지 않은 언체크 예외를 무시할 수 있다. 체크 예외의 경우 처리할 수 없는 예외를 밖으로 던지려면 항상 `throws 예외`를 선언해야 하지만, 언체크 예외는 이 부분을 생략할 수 있다.
- **단점**: 언체크 예외는 개발자가 실수로 예외를 누락할 수 있다. 반면에 체크 예외는 컴파일러를 통해 예외 누락을 잡아준다.

정리

체크 예외와 언체크 예외의 차이는 예외를 처리할 수 없을 때 예외를 밖으로 던지는 부분에 있다. 이 부분을 필수로 선언해야 하는가 생략할 수 있는가의 차이다.