예외 처리

```
├── Throwable
│   ├── Exception
│   │   ├── Checked Exception
│   └── └── RuntimeException
└──
```

> [!important]
> 예외처리란? 프로그램이 실행되는 중에 예상치 못한 상황에 대응하여 적절한 조치를 취하는 것을 말합니다 예외처리는 프로그램의 안정성을 높이고 예측할 수 없는 상황에서 프로그램의 비정상 종료를 방지할 수 있습니다.

Exception

java.lang.Exception

> [!NOTE]
> 예외를 영어로 하면 Exception 이라고 하며, 클래스 기반으로 된 계층구조를 가지고 있습니다.Exception 은 모든 예외 클래스의 최상위 클래스이며, 모든 예외는 Exception 클래스를 상속하고 있습니다.

Checked Exception과 Runtime Exception 정의

> [!NOTE]
> Exception은 크게 일반 예외인 Checked Exception 과 실행 예외인 Runtime Exception 이 존재합니다. Checked Exception 은 컴파일러가 예외 처리 코드를 강제하는 것을 말하며 컴파일러에 의해 Checked 되기 때문에 Checked 예외라고 합니다. Runtime Exception 은 실행 중에 발생하기 때문에 Runtime 예외라고 하며 컴파일러가 예외 할 코드를 검사하지 않는 예외입니다. 예외 처리 코드를 강제한다는 것은 컴파일 할 때, 즉 파일을 실행할 때 에러가 발생하는 것을 말하며 컴파일러가 예외하지 않는다는 것은 컴파일을 할 때 문제없이 컴파일이 동작한다는 것을 의미합니다. Checked Exception과 Runtime Exception 으로 나뉘어져 있는 이유는 Checked Exception은 자바에서 예측 가능한 에러이기 때문에 강제할 수 있지만 Runtime Exception은 주로 개발자의 실수로 발생되므로 예측 불가능하기 때문에 강제할 수 없기 때문입니다.

Checked Exception

IOException: 파일 입출력과 관련된 작업에서 발생하는 예외입니다. 파일이 존재하지 않거나 읽기/쓰기 권한이 없는 경우 등이 해당됩니다.

SQLException: 데이터베이스와 관련된 작업에서 발생하는 예외입니다. SQL 쿼리 실행 중에 문제가 발생할 경우 이 예외가 발생합니다.

ClassNotFoundException: 클래스를 찾을 수 없는 경우에 발생하는 예외입니다. 클래스 패스에 해당 클래스가 없는 경우 등이 해당됩니다.

RuntimeException

NullPointerException: null인 참조 변수를 사용하려고 할 때 발생하는 예외입니다. 예를 들어, null인 객체의 메서드를 호출하거나 null을 다루는 연산을 수행하려고 할 때 발생합니다.

ArrayIndexOutOfBoundsException: 배열의 범위를 벗어난 인덱스에 접근하려고 할 때 발생하는 예외입니다. 배열의 길이를 넘어가는 인덱스에 접근하면 이 예외가 발생합니다.

ArithmeticException: 산술 연산 중에 발생하는 예외입니다. 예를 들어, 0으로 나누기 연산이나 정수를 0으로 나누는 경우에 발생합니다.

IllegalArgumentException: 메서드에 전달된 인수의 값이 부적절한 경우에 발생하는 예외입니다. 메서드에 대한 인수로 잘못된 값이 전달되면 이 예외가 발생합니다.

RuntimeException (일반적인 상위 클래스): 여러 Runtime Exception들의 상위 클래스로, 일반적으로 프로그래머의 실수나 잘못된 로직에 의해 발생하는 예외들을 포함합니다.

- 일반 예외 (Exception) 컴파일러가 예외 처리 코드 여부를 검사하는 예외
    - Checked Exception Exception 클래스를 상속하면서 RuntimeException 클래스를 상속하지 않는 예외
    - 실행 예외 (RuntimeException) 컴파일러가 예외 처리 코드 여부를 검사하지 않은 예외

1. 실행 예외(RuntimeException): 컴파일러의 역할: 컴파일러가 검사하지 않기 때문에 컴파일 시 강제하지 않는다.

발생 시점: 프로그램 실행 중(runtime)에 발생하는 예외로, 프로그램의 로직에 의해 발생한다.

대표적인 예외:

- NullPointerException(null 값을 가지고 있는 객체 또는 변수)
- ArrayIndexOutOfBoundsException(잘못된 배열 인덱스)
- ArithmeticException(예외적 산술 조건)

처리 방법: 프로그래머가 명시적으로 예외를 처리하지 않아도 되며, 처리할 경우도 선택사항입니다. 일반적으로 예외 발생 시의 상황을 수정하는 것이 아니라 코드 로직을 개선하여 예외가 발생하지 않도록 하는 것이 바람직합니다.

```
public class RuntimeExceptionExample {
    public static void main(String[] args) {
        int[] arr = new int[5];
        // ArrayIndexOutOfBoundsException이 발생하더라도 컴파일러는 강제로 예외 처리
를 요구하지 않음.
        System.out.println(arr[10]);
    }
}
```

2. 일반 예외(Checked Exception): 컴파일러의 역할: 컴파일러가 검사하며, 예외 처리 코드를 강제한다. 반드시 예외 처리 코드를 작성해야 한다.

발생 시점: 파일 읽기나 데이터베이스 연결 등의 작업에서 발생한다.

대표적인 예외:

- IOException(존재하지 않는 파일 등 입출력)
- SQLException(잘못된 SQL문)

처리 방법: 예외를 명시적으로 처리하거나, throws를 사용하여 예외를 상위 호출자에게 전파한다.

```
// src/main/java/org.example/Main.java
package org.example;

public class Main {
    public static void main(String[] args) {

        Test test = new Test();
        test.testExceptoin();

        System.out.println("Hello world!");
    }
}
```

```
// src/main/java/org.example/Test.java
package org.example;

// require 된 라이브러리 모듈
import java.io.FileReader;

public class Test {

    public void testExceptoin() {
        try {
            // 예외가 발생할 수 있는 코드
            FileReader fileReader = new FileReader("example.txt");

        } catch (Exception e) {
            // Exception 클래스를 잡아서 처리

            // 예외가 발생된 추적 내용
            e.printStackTrace();
        }
    }

}
```

```
java.io.FileNotFoundException: example.txt (지정된 파일을 찾을 수 없습니다)
```

FileNotFoundException은 java.io 패키지에 속한 예외로서, 파일을 찾을 수 없을 때 발생하는 예외입니다. 이는 Checked Exception에 해당합니다.

예외 처리 코드

try 블록:

try는 예외가 발생할 수 있는 코드를 포함시키며, 예외가 발생되면 catch와 finally 블록이 실행된다.

> [!note]/ Java에서 예외(Exception)가 발생하면, 예외에 대한 정보는 catch 블록에서 선언한 참조변수를 통해 접근할 수 있습니다. 이 참조변수는 catch 블록 내에서 사용되며, 보통 e라는 이름으로 사용됩니다.

```
try {
    // 예외가 발생할 수 있는 코드
} catch (ExceptionType1 e1) {
    // ExceptionType1에 대한 예외 처리
} catch (ExceptionType2 e2) {
    // ExceptionType2에 대한 예외 처리
} finally {
    // 항상 실행되어야 하는 코드 (예외 발생 여부와 상관없이)
}
```

catch 블록:

예외 처리를 한다.

```
try {
    // 예외가 발생할 수 있는 코드
} catch (IOException e) {
    // IOException에 대한 예외 처리
    e.printStackTrace();
} catch (SQLException e) {
    // SQLException에 대한 예외 처리
    e.printStackTrace();
}
```

finally 블록:

예외 발생 여부와 관계없이 항상 실행되며 선택 사항이다.

```
// 데이터베이스와 연결한 Connection 객체를 닫아야 합니다. 데이터베이스 연결은 한정된 자원이므로 연결이 더 이상 필요하지 않을 때 적절하게 닫아주어야 합니다.
Connection connection = null;
try {
    connection = DriverManager.getConnection("jdbc:mysql://localhost:3306/example", "username", "password");
    // 데이터베이스 작업 수행
} catch (SQLException e) {
    e.printStackTrace();
} finally {
    try {
        if (connection != null) {
```

```
            connection.close(); // 데이터베이스 연결 닫기
        }
    } catch (SQLException e) {
        e.printStackTrace();
    }
}
```

```
public void twoException() {
    try {
        int[] arr = new int[5];
        // 예외가 발생할 수 있는 코드: 배열의 길이를 넘어가는 인덱스에 접근
        System.out.println(arr[10]);

        // 다른 예외 발생 시도: 0으로 나누기
        int result = 10 / 0;
    } catch (ArrayIndexOutOfBoundsException e) {
        //ArrayIndexOutOfBoundsException에 대한 예외 처리
        System.err.println("ArrayIndexOutOfBoundsException: " + e.getMessage());
    } catch (ArithmeticException e) {
        // ArithmeticException에 대한 예외 처리
        System.err.println("ArithmeticException: " + e.getMessage());
    } finally {
        // 항상 실행되어야 하는 코드 (예외 발생 여부와 상관없이)
        System.out.println("Finally block executed");
    }
}
```

throws

> 메소드 내부에서 예외가 발생할 때 try-catch 블록으로 예외를 처리하는 것이 아닌, 메소드를 호출한 곳으로 예외를 넘길 수 있다.

```
리턴타입 메소드명(매개변수,...) throws 예외클래스1, 예외클래스2 ... {

}
```

예시 코드

```
package org.example;

public class Main {
    public static void main(String[] args) {

        Test test = new Test();
        //test.testExceptoin();
```

```
        //test.testRuntimeException();
        test.twoException();

        System.out.println("Hello world!");

        try {
            // 메서드에서 Throwable을 던지는 예제
            test.readFile("text");
        } catch (Throwable t) {
            // Throwable 예외 처리
            System.err.println("Throwable caught: " + t.getMessage());
        }
    }
}
```

```
package org.example;

import java.io.FileReader;

public class Test {
    public static void readFile(String filename) throws Throwable {
        // 파일을 읽는 작업 수행
        // 예제에서는 파일을 열려고 하지만 실제 파일이 존재하지 않으면
FileNotFoundException이 발생
        throw new Throwable("File not found: " + filename);
    }
}
```

결과

```
Finally block executed
Hello world!
Throwable caught: File not found: text
```

사용자 정의 예외

> [!NOTE]
> 자바 라이브러리에 존재하지 않는 예외를 발생시키고 싶을 때 사용자 정의 예외를 사용한다. 사용자 정의 예외는 일반 예외와 실행 예외 모두 선언할 수 있다.

예시

```
public class XXXException extesnds [Exception | RuntimeException] {
    public XXXException() {}
```

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

	상속	예외 확인시점	예외처리	장점	단점
Checked Exception	Exception	컴파일 단계	O	개발자의 실수 낮음	레이어간 의존성이 높고 Steam 내에서 사용하지 않음
UnChecked Exception	Runtime Exception	실행 단계	X	레이어간 의존성이 앚음	개발자의 실수가 높음