● 프로그램 오듂

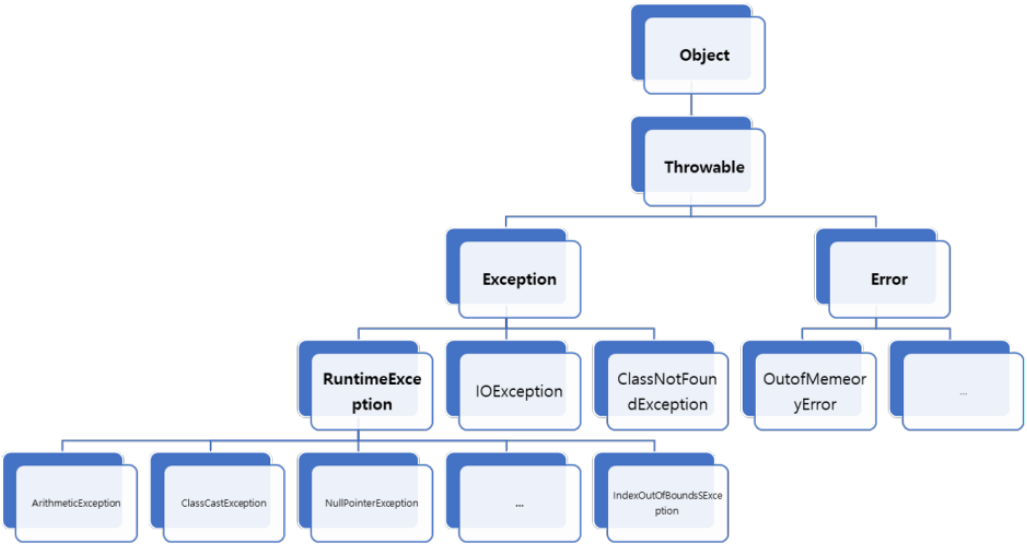
▶ 컴파일 에러(compile-time error): 컴파일 시에 발생하는 에러.

▶ 런타임 에러(run-time error): 프로그램의 실행도중, 즉, 실행 시에 발생하는 에러
☞ 에러(error): 일단 발생하면 복구할 수 없는 심각한 오듂. 프로그램 코드에 의해서 수습될 수 없다.
(ex. 메모리 부족(OutOfMemory Error), 스택오버플로우(StackOverflowError)
☞ 예외(exception): 발생하더라도 미리 작성한 프로그램 코드에 의해서 수습될 수 있는 다소 미약한 오듂.

▶ 논리적 에러(logical error): 컴파일과 실행은 잘 되지만, 의도와 다르게 동작하는 것

소스파일을 컴파일하면 컴파일러가 소스코드(.java)에 대해 오타나 잘못된 구문, 자료형 체크 등의 기본적인 검사를 수행하여 오듂가 있는지를 알려 준다. 컴파일러가 알려 준 에러들을 모두 수정해서 컴파일을 성공적으로 마치고 나면, 클래스 파일(.class)이 생성되고, 생성된 클래스 파일을 실행할 수 있게 되는 것이다. 하지만 컴파일이 에러없이 성공적으로 마쳤다고 해서 프로그램 실행 시에도 에러가 발생하지 않는 것은 아니다. 컴파일러가 실행도중에 발생할 수 있는 잠자적인 오듂까지 검사할 수 없기 때문에 컴파일은 잘 되어있어도 실행 중에 에러에 의해서 잘못된 결과를 얻거가 프로그램이 비정상적으로 종료될 수 있다. 런타임 에러 중에 예외(exception)에 해당되는 것들은 프로그래머가 코드를 짤 때 미리 적절한 코드를 작성하여 프로그램의 비정상적인 종료를 방지할 수 있다.

● 예외 클래스의 계층 구조: 자바에서는 실행 시 발생할 수 있는 오듂(Exception과 Error)를 클래스로 정의하였다. 모든 클래스의 조상은 Object이므로 Exception과 Error클래스 역시 Object 클래스의 자손들이다. 모든 예외의 조상은 Exception이며, 예외 클래스들은 두 그둡으로 나뉜다.



▶ Exception (예외)

☞ RuntimeException클래스들: RuntimeException클래스와 그 자손들. 프로그래머의 실수로 발생하는

예외들로 자바의 프로그래밍 요소들과 관계가 깊다.

Ex) ArrayIndexOutOfBoundsException: 배열의 범위를 벗어난다

Ex) NullPointerException: 값이 null인 참조변수의 멤버를 호출하려 했다

Ex) ClassCastException: 클래스간의 형변환을 잘못했다

Ex) ArithmeticException: 정수를 0으로 나누려 했다 (실수를 0으로 나누는 것은 허용된다)

☞ Exception클래스들: Exception클래스와 그 자손들(RuntimeException과 자손들 제외).
사용자의 실수와 같은 외적인 요인에 의해 발생하는 예외. 주로 외부의 영향으로 발생할 수 있는 것들로서, 프로그램의 사용자들의 동작에 의해서 발생하는 경우가 많다.

Ex) FileNotFoundException: 존재하지 않는 파일의 이름을 입력했다

Ex) ClassNotFoundException: 클래스의 이름을 잘못 적었다

Ex) DataFormatExcpetion: 입력한 데이터 형식이 잘못됐다

● 예외처리(exception handling)

정의	프로그램 발생할 수 있는 예외에 대비한 코드를 작성하는 것
목적	예외의 발생으로 인한 실행 중인 프로그램의 비정상 종료를 막고, 정상적인 실행상태를 유지하는 것

발생한 예외를 처리하지 못하면 프로그램은 비정상적으로 종료되며, 처리되지 못한 예외(uncaught exception)은 JVM의 예외처리기(UncaughtExceptionHandler)가 받아서 예외의 원인을 화면에 출력한다. 결과에 나타난 메시지를 보면 예외의 발생원인과 위치를 알 수 있다.

▶ try-catch문을 이용한 예외처리

```
try {
    // 예외가 발생할 가능성이 있는 문장들을 넣는다.
} catch (예외이름 참조변수) {
    // 예외가 발생했을 경우, 이를 처리하기 위한 문장을 적는다.
} catch (ArithmeticException ae) { // 예를 들어,
    // ArithmeticException 예외가 발생했을 경우, 이를 처리하기 위한 문장을 넣는다.
} finally {
    // 예외의 발생여부에 관계없이 실행될 코드.
}
```

하나의 try블럭 다음에 여러 종듀의 예외를 처리할 수 있도록 하나 이상의 catch블럭을 사용할 수 있으며, 이 중 발생한 예외의 종듀와 일치하는 단 한 개의 catch블럭만 수행된다. 발생한 예외의 종듀와 일치하는 catch블럭이 없으면 예외는 처리되지 않는다. 또한 하나의 메서드 내에 여러 개의 try-catch문이 사용될 수 있으며, try블럭 또는 catch블럭에 또다든 try-catch문이 포함될 수 있다. catch블럭 괄호 내에 선언된 변수는 catch블럭 내에서만 유효하기 때문에, 모든 catch 블럭에 참조변수 'e' 하나만을 사용해도 된다. 그러나 catch블럭 내에 또 하나의 try-ctach문이 포함될 경우, 같은 이름의 참조변수를 사용해서는 안 된다. 각 catch블럭에 선언된 두 참조변수의 영역이 서로 겹치므로 다든 이름을 사용해야만 서로 구별되기 때문이다.

▶ try-catch문에서의 흐름: 예외가 발생한 경우와 발생하지 않았을 때의 흐름(문장 실행순서)가 달라진다.

☞ try블럭 내에서 예외가 발생한 경우:
1) 발생한 예외와 일치하는 catch블럭이 있는지 확인한다.

2) 일치하는 catch블럭을 찾으면 그 catch블럭 내의 문장들을 수행하고 전체 try-catch문을 빠져나가서 그 다음 문장을 계속해서 수행한다.
3) 일치하는 catch블럭을 찾지 못하면, 예외는 처리되지 못한다.

☞ try블럭 내에서 예외가 발생하지 않은 경우:
1) catch블럭을 거치지 않고 전체 try-catch문을 빠져나가서 수행을 계속한다.

try블럭에서 예외가 발생하면, 예외가 발생한 위치 이후에 있는 try블럭의 문장들은 수행되지 않는다. 그러므로 try블럭에 포함시킬 코드의 범위를 잘 선택해야 한다.

```
class ExceptionEx5 {
    public static void main(String args[]) {
            System.out.println(1);
            System.out.println(2);
            try {
                System.out.println(3);
                System.out.println(0/0);
                System.out.println(4);  // 실행되지 않는다.
            } catch (ArithmeticException ae)    {
                System.out.println(5);
            }   // try-catch의 끝
            System.out.println(6);
    }   // 1 2 3 5 6
}
```

▶ 예외 발생과 catch블럭: catch블럭은 괄호()와 블럭{} 두 부분으로 나누어져 있고, 괄호()내에는 처리하고자 하는 예외와 같은 타입의 참조변수 하나를 선언해야 한다. 예외가 발생하면, 발생한 예외에 해당하는 클래스의 인스턴스가 만들어진다. 그리곤 첫번째 catch블럭부터 차례로 내려가면서 catch블럭 괄호()내에 선언된 참조변수의 종류와 생성된 예외클래스의 인스턴스에 instanceof연산자를 이용해서 검사하게 되는데, 검사결과가 true인 catch블럭을 만날 때까지 검사는 계속된다. 모든 예외 클래스는 Exception클래스의 자손이므로, catch블럭의 괄호()에 Exception클래스 타입의 참조변수를 선언하면 어떤 종류의 예외가 발생하더라도 이 catch블럭에 의해서 처리된다. 예를 들면 ArithmeticException클래스는 Exception클래스의 자손이므로 ArithmeticException인스턴스와 Exception클래스와의 instanceof연산결과가 true가 되어 Exception클래스 타입의 참조변수를 선언한 catch블럭의 문장들이 수행되고 예외는 처리되는 것이다.

```
class ExceptionEx7 {
    public static void main(String args[]) {
        System.out.println(1);
        System.out.println(2);
        try {
            System.out.println(3);
            System.out.println(0/0);
            System.out.println(4);      // 실행되지 않는다.
        } catch (ArithmeticException ae)    {
//          if (ae instanceof ArithmeticException)
                System.out.println("true");
            System.out.println("ArithmeticException");
        } catch (Exception e)   { // ArithmeticException을 제외한 모든 예외가 처리된다
            System.out.println("Exception");
        }   // try-catch의 끝
        System.out.println(6);
    }   // main메서드의 끝
}
```

위의 예를 보면 첫번째 검사에서 일치하는 catch블럭을 찾았기 때문에 두번째 catch블럭은 검사하지 않게 된다. 만약 try블럭 내에서 ArithmeticException이 아닌 다른 종류의 예외가 발생한 경우에는 두번째 catch블럭인 Exception클래스 타입의 참조변수를 선언한 곳에서 처리되었을 것이다. 이처럼 try-catch문의 마지막에 Exception클래스 타입의 참조변수를 선언한 catch블럭을 사용하면, 어떤 종류의 예외가 발생하더라도 이 catch 블럭에 의해 처리되도록 할 수 있다.

▶ printStackTrace()와 getMessage(): 예외의 발생원인을 알게 해 주는 메서드들. 예외가 발생했을 때 생성되는 예외 클래스의 인스턴스에는 발생한 예외에 대한 정보가 담겨 있으며, getMessage()와 printSetackTrace()을 통해서 이 정보들을 얻을 수 있다. catch블럭 괄호()에 선언된 참조변수를 통해 이 인스턴스에 접근할 수 있다. 이 참조변수는 선언된 catch블럭 내에서만 사용 가능하다. (참고로 printStackTrace(PrintStream s) 또는 PrintStackTrace(PrintWriter s)를 사용하면, 발생한 예외에 대한 정보를 파일에 저장할 수도 있다).

printStackTrace()	예외발생 당시의 호출스택(Call Stack)에 있었던 메서드의 정보와 예외 메세지를 화면에 출력한다.	Ex) java.lang.ArithmeticException : / by zero at ExceptionEx8.main(ExceptionEx8.java:7)
getMessage()	발생한 예외클래스의 인스턴스에 저장된 메시지를 얻을 수 있다.	Ex) / by zero

```
class ExceptionEx8 {
    public static void main(String args[]) {
        System.out.println(1);
        System.out.println(2);
        try {
            System.out.println(3);
            System.out.println(0/0); // 예외발생!!!
            System.out.println(4);   // 실행되지 않는다.
        } catch (ArithmeticException ae)    {
            ae.printStackTrace();
            System.out.println("예외메시지 : " + ae.getMessage());
        }   // try-catch의 끝
        System.out.println(6);
    }   // main메서드의 끝
}
```

▶ 멀티 catch블럭: 여러 catch블럭을 '|'기호를 이용해서 하나의 catch블럭으로 합쳐 중복된 코드를 줄일 수 있으며, 이를 '멀티 catch블럭'이라 한다. '|' 기호로 연결할 수 있는 예외 클래스의 개수에는 제한이 없다. 다만 연결된 예외 클래스가 조상과 자손의 관계에 있다면 컴파일 에러가 발생한다. 왜냐하면, 이런 경우 그냥 조상 클래스만 써주는 것과 똑같기 때문이다. 또한 멀티 catch는 하나의 catch블럭으로 여러 예외를 처리하는 것이기 때문에, 발생한 예외를 멀티 catch블럭으로 처리하게 되었을 때, 멀티 catch블럭 내에는 실제로 어떤 예외가 발생했는지 알 수 없다. 그래서 참조변수 e로 멀티 catch블럭에 '|'기호로 연결된 예외 클래스들의 공통 분모인 조상 예외 클래스의 선언된 멤버만 사용할 수 있다. 마지막으로 멀티 catch블럭에 선언된 참조변수 e는 상수이므로 값을 변경할 수 없다는 제약이 있는데, 이것은 여러 catch블럭이 하나의 참조변수를 공유하기 때문에 생기는 제약으로 실제 참조변수의 값을 바꿀 일은 없을 것이다.

```
try {
    ...
} catch (ExceptionA | ExceptionB e) {
    e.methodA(); // 에러. ExceptionA에 선언된 methyodA()는 호출 불가.
    if (e instanceof ExceptionA) {
        ExceptionA e1 = (ExceptionA)e;
        e1.methodA(); //하지만 이렇게까지 하면서 catch블럭을 합칠 일은 거의 없다.
    }
}
```

▶ 예외 발생시키기

1. 먼저, 연산자 new를 이용해서 발생시키려는 예외 클래스의 객체를 만든다. Ex) Exception e = new Exception("고의로 발생시켰음"); Exception인스턴스를 생성할 때 생성자에 String을 넣어주면, 이 String이 Exception인스턴스에 메세지로 저장된다. 이 메시지는 getMessage()를 이용해서 얻을 수 있다.
2. 키워드 throw를 이용해서 예외를 발생시킨다. throw e;

```
class ExceptionEx9 {
    public static void main(String args[]) {
        try {
            Exception e = new Exception("고의로 발생시켰음.");
            throw e;     // 예외를 발생시킴
        //  throw new Exception("고의로 발생시켰음.");

        } catch (Exception e)   {
            System.out.println("에러 메시지 : " + e.getMessage());
             e.printStackTrace();
        }
        System.out.println("프로그램이 정상 종료되었음.");
    }
}
```

☞ Exception예외를 발생시키고 예외처리를 하지 않은 경우: 컴파일이 완료되지 않는다. Exception클래스들(Exception클래스와 그 자손들)이 발생할 가능성이 있는 문장들에 대해 예외처리를 하지 않으면 컴파일조차 되지 않는다. 컴파일러가 예외처리를 확인하는 Exception클래스들은 'checked예외'라고 부른다.

☞ RuntimeException예외를 발생시키고 예외처리를 하지 않는 경우: 성공적으로 컴파일되지만 실행하면 RuntimeException이 발생하여 비정상적으로 종료된다. RuntimeException클래스들에 해당하는 예외는 프로그래머에 의해 실수로 발생하는 것들이기 때문에 예외처리를 강제하지 않기 때문이다. 컴파일러가 예외처리를 확인하지 않는 RunTimeException클래스들은 'unchecked예외'라고 부른다.

Resource: 자바의 정석