

Exception I - try ~ catch

2016년 9월 29일 목요일 오전 1:00

Error vs Exception

- Error : 컴파일 에러
 - Exception : 프로그램 문법상에는 문제가 없으나, 실행 과정 중에 발생하는 문제의 상황
(논리적 오류, 컴파일 오류와는 의미가 다르다)
-

if문을 이용한 예외상황의 처리

```
public static void main(String[] args) {  
    Scanner scan = new Scanner(System.in);  
    int num1=0, num2=0;  
    int i = 0;  
  
    System.out.println("피제수 입력 : ");  
    scan.nextInt();  
    System.out.println("제수 입력 : ");  
    scan.nextInt();  
  
    // 예외의 감지  
    if(num2 == 0){  
        // 예외의 처리  
        System.out.println("제수는 0이 될 수 없습니다.");  
        i -= 1;  
    }  
}
```

if문으로 예외처리를 했을 때의 단점

- if문이 프로그램의 흐름을 담당하는 건지, 예외를 처리하는 건지 주석을 달지 않는 이상 한눈에 분석하기 어렵다.
 - 너무 많이 사용하면 프로그램 전체흐름을 이해하는데 방해가 된다.
-

Java의 예외처리 메커니즘 - try ~ catch문

- 예외처리 로직을 구분하겠다.
- try ~ catch 문의 장점
 - try문을 보면서.. 아! 예외발생 가능지역이구나!
 - catch문을 보면서.. 아! 예외처리 코드이구나!
 - 직관적인 판단과 예외의 감지와 처리의 영역 구분이 가능하다.

```
try
{
    // try 영역 : 프로그램의 정상적인 흐름 중에서 예외발생이 예상가능한 영역
    (try 영역도 프로그램의 정상적인 흐름의 일부다.)
}
catch(Exception e)
{
    // catch 영역 : try 영역에서 Exception 예외가 발생했을 때 처리하는 영역
}
```

```
import java.util.Scanner;
```

```
public class ExceptionHandler {
    public static void main(String[] args) {
        Scanner scan = new Scanner(System.in);
        int num1 = 0;
        int num2 = 0;

        System.out.println("제수를 입력하세요 : ");
        num2 = scan.nextInt();
        System.out.println("피제수를 입력하세요 : ");
        num1 = scan.nextInt();

        try {
```

① JVM이 연산예외 발생을 감지, 그리고 해당예외를 표현할 수 있는 인스턴스를 생성한다.

(표준으로 예외 Class가 많이 정의되어 있다. 아래의 예외는 `ArithmeticException`)

```
System.out.println("나눗셈 결과의 몫 : "+num1/num2);
System.out.println("나눗셈 결과의 나머지 : "+num1%num2); // 예외가 발생한 로직 아래의 로직들은 다 건너뛰는다.
```

```
/*
 * 컴파일러는 결과 중심으로 예외를 판단한다. 그래서 num2 = 0 연산은 정상으로 판단하나
 * num2를 가지고 나눗셈 연산을 하는 시점을 예외로 판단한다.
 */
}
```

② JVM이 try문에서 생성된 예외의 인스턴스를 catch문으로 전달하면서 catch문 실행

(try문에서 전달받은 인스턴스의 자료형과 catch문의 매개변수 자료형이 일치해야 한다.)

```
catch(ArithmeticException e) {

    System.out.println("나눗셈 불가능");
    System.out.println(e.getMessage());

}

System.out.println("프로그램을 종료합니다.");
}
}
```

적절한 try 블록의 구성

```
try {

    int result = num1/num2;

} catch(ArithmeticException e){
    . . . . .
}
```

try문에서 예외가 발생하면 result값이 적절한 값을 갖지 못하게 된다.
예외상황이 발생해도 아래의 로직이 실행된다.

```
System.out.println("정수형 나눗셈이 정상적으로 진행되었습니다.");  
System.out.println("나눗셈 결과 : "+result);
```

```
try {
```

하나의 일의 단위(Transaction)로 구성, 모두 실행되거나, 모두 실행되지 않거나

```
int result = num1/num2; // 이 문장에서 예외가 발생하면 나머지 아래의 문장 두 개는 다 건너뛴다.  
System.out.println("정수형 나눗셈이 정상적으로 진행되었습니다.");  
System.out.println("나눗셈 결과 : "+result);
```

```
} catch(ArithmeticException e){  
    예외 발생 시 실행하면 안 되는 부분을 건너 뛴 수 있다.  
}
```

e.getMessage()

- ArithmeticException Class와 같이 예외상황을 알리기 위해 정의된 Class를 가리켜 Exception Class (예외클래스)라고 한다.
- 모든 Exception Class는 Throwable Class를 상속하며, 이 클래스에 getMessage() 메소드가 정의되어 있다.
- getMessage() 메소드는 예외가 발생한 원인정보를 문자열의 형태로 반환한다.

```
try {
```

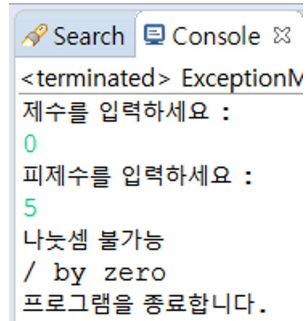
```
    System.out.println("나눗셈 결과의 몫 : "+num1/num2);  
    System.out.println("나눗셈 결과의 나머지 : "+num1%num2);
```

```
} catch(ArithmeticException e) {
```

```
    System.out.println("나눗셈 불가능");
```

```
        System.out.println(e.getMessage());
    }
    System.out.println("프로그램을 종료합니다.");
```

실행결과



```
<terminated> ExceptionIn
제수를 입력하세요 :
0
피제수를 입력하세요 :
5
나눗셈 불가능
/ by zero
프로그램을 종료합니다.
```

예외클래스는 모두 정의되어 있는가?- No!

- 모든 경우에 있어서 예외로 인정되는 상황을 표현하기 위한 예외클래스는 대부분 정의가 되어있다.
- 하지만 프로그램상의 모든 논리적인 예외는 미리 구현할 수가 없다.
- 그래서 프로그램에 따라 별도로 표현해야 하는 예외 상황에서는 별도로 예외클래스를 직접 정의하면 된다.
- 대표적인 예외클래스

ArrayIndexOutOfBoundsException	배열의 접근에 잘못된 인덱스 값을 사용하는 예외상황
ClassCastException	허용할 수 없는 Casting 연산을 진행하는 예외상황
NegativeArraySizeException	배열선언 과정에서 배열의 크기를 음수로 지정하는 예외상황
NullPointerException	<u>참조변수가 null로 초기화 된 상황에서 메소드를 호출하는 예외상황</u>

try ~ catch문의 또다른 장점

- **하나의 try 블록에 둘 이상의 catch 블록을 구성할 수 있기 때문에, 다양한 예외처리 관련된 부분을 완전히 별도로 떼어 놓을 수 있다!**

```
try {
    . . . . .
} catch(ArithmeticException e){
    . . . . .
} catch(ArrayIndexOutOfBoundsException e){
    . . . . .
}
```

catch가 결정되는 방법

- JVM이 첫번째 catch 블록에서 부터 순서대로 내려와 예외상황에 맞는 예외클래스 참조변수를 찾는다.
- Catch 블록의 매개변수가 해당 예외 인스턴스의 참조값을 받을 수 있는지 확인하며 위에서 아래로 내려온다.
- Method Overloading의 관점에서 생각하면 안 된다.

```
try {
    . . . . .
} catch(ArithmeticException e){ ← 1차 : 이곳에서 처리가 가능한가?
    . . . . .
} catch(ArrayIndexOutOfBoundsException e){ ← 2차 : 이곳에서 처리가 가능한가?
    . . . . .
}
```

```
try {
    . . . . .
}
```

Throwable class는 모든 예외클래스의 부모이기 때문에 다형성에 의해 모든 예외클래스의 인스턴스를 참조할 수 있으므로 이렇게 선언하면 안 된다.

```
catch(Throwable e){
    . . . . .
}
```

```
catch(ArithmeticException e){  
    . . . . .  
}
```

항상 실행되는 finally

```
try  
{  
    int result = num1/num2;  
    System.out.println("나눗셈 결과는 "+result);  
    return true;  
}  
catch(ArithmeticException e)  
{  
    System.out.println(e.getMessage());  
    return false;  
}  
  
finally  
{  
    System.out.println("finally 영역 실행");  
}
```

- 예외가 발생하던 안 하던 실행되는 영역
 - try문으로 진입하면 100% 무조건 실행이 된다.
 - try문 중간에 return문을 실행하더라도, finally 블록이 실행된 다음 메소드를 빠져나간다.
 - try문에서 예외가 발생하면 catch문으로 진입해 catch문을 실행하고 난 후 finally문 실행
 - try문에서 예외가 없어도 catch문을 건너뛰어 finally문 실행
-