

08_2 Throwing & Catching Exceptions

Object-Oriented Programming

Exception의 Throwing과 Catching에 대해 강의하겠습니다.

Exception Handling Code

- try-catch-finally block

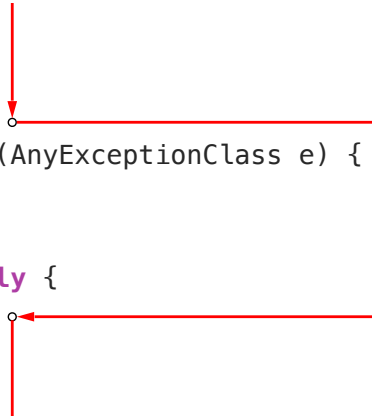
```
try {  
    // Code that can throw an exception  
}  
catch (AnyExceptionClass e) {  
    // Exception handling  
}  
finally {  
    // Code that is always executed  
    // whether the exception is thrown or not  
}
```

Exception handling의 기본 code는 try-catch-finally block 입니다.
try block 안에는 exception이 throwing 될 가능성이 있는
즉 exception이 일어날 가능성이 있는 code를 둡니다.
catch block에서는 일어난 (throwing된) exception이 있는 경우
그 exception을 handling하는 code를 둡니다.
catch block에서는 발생한 exception을
parameter 형태로 받아서 이용할 수 있음을 눈여겨 보기 바랍니다.
finally block은 exception이 일어난 여부와 상관없이
항상 실행되어야 하는 부분을 포함합니다.
finally block은 꼭 있어야 하는 부분은 아니라서
많은 경우에 생략되기도 합니다.

Try-Catch-Finally Block

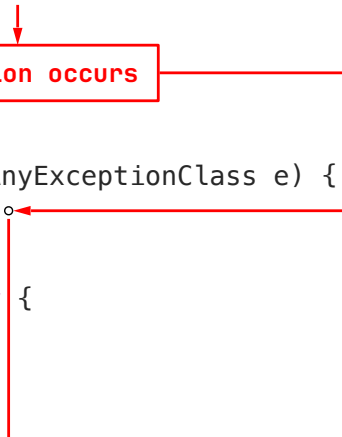
// normal execution

```
try {  
      
} catch (AnyExceptionClass e) {  
      
} finally {  
      
}
```



// exception case

```
try {  
      
} catch (AnyExceptionClass e) {  
      
} finally {  
      
}
```




Exception이 발생하지 않는 상태에서
try-catch-finally block의 실행 순서는
왼쪽 그림과 같습니다.
try block의 시작부터 끝까지 실행 후
catch block은 건너뛰고
finally block을 모두 실행한 후
계속 다음으로 실행해 나가는 것입니다.
반면 Exception이 발생하는 경우에는
try block 안에서 exception이 발생하는 지점에서
실행이 끊어지고 catch block으로 이동하여
catch block을 처음부터 끝까지 모두 실행하며
finally block을 모두 실행하고
계속 다음으로 실행해 나가는 것입니다.
특히 exception이 일어나는 케이스에서는
try block 내에서
exception 이 일어난 지점 이후의 명령어들은
실행되지 않고 catch block으로 jump한다는 점에
유의하기 바랍니다.

Example: Flow in try-catch Block (1/2)

No Exception case

```
class ExceptionEx02 {  
    public static void main(String args[]) {  
        System.out.println(1);  
        System.out.println(2);  
        try {  
            System.out.println(3);  
            System.out.println(4);  
        } catch (Exception e) {  
            System.out.println(5);  
        }  
        System.out.println(6);  
    }  
}
```



Output:

1
2
3
4
6

좀 더 구체적으로 try catch block의 flow에 대해 알아보죠.
이 경우는 exception이 일어나지 않는 경우입니다.
main method에서 1과 2를 print한 후
try block에서 3과 4를 print하고
exception이 발생하지 않았기 때문에
catch block은 실행되지 않습니다.
catch block을 건너뛰고 6을 print하고
프로그램은 종료됩니다.

Example: Flow in try-catch Block (2/2)

Exception case

```
class ExceptionEx02 {  
    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);  
        } catch (Exception e) {  
            System.out.println(5);  
        }  
        System.out.println(6);  
    }  
}
```

Division by Zero

Output:

1
2
3
5
6

5

페이지 5

이번에는 exception이 발생하는 경우를 살펴 보죠.
1과 2를 print한 후 try block에 진입합니다.
3을 print하고
0/0 을 시도하는데, 0으로 나누는 것은
ArithmeticException을 발생시킵니다.
발생된 exception은
catch block에서 catch되어
catch block으로 진입합니다.
5를 print하고 catch block을 끝낸 후
6을 print하고 프로그램이 끝나게 됩니다.

Example: TryCatchDemo

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

        String[] str = new String[]{"123", "45", "abc"};
        int[] a = new int[3];
        for (int i = 0; i < 4; i++) {
            try {
                a[i] = Integer.parseInt(str[i]);
                System.out.println("Index " + i + " parseInt done");
            } catch (ArrayIndexOutOfBoundsException e) {
                System.out.println("Array index exception at index " + i);
            } catch (NumberFormatException e) {
                System.out.println("Number format exception at index " + i);
            } catch (Exception e) {
                System.out.println("Other exception at index " + i);
            } finally {
                System.out.println("finally index " + i + " done");
            }
        }
    }
}
```

```
Index 0 parseInt done
finally index 0 done
Index 1 parseInt done
finally index 1 done
Number format exception at index 2
finally index 2 done
Array index exception at index 3
finally index 3 done
```

6

페이지 6

TryCatchDemo example 프로그램에서는 String array의 element들을 하나씩 int type으로 conversion하는 과정에서 발생할 수 있는 exception들을 try-catch-finally block을 이용하여 handling 하는 code를 살펴보겠습니다. 우리 프로그램에서 String array의 element들은 "123", "45", "abc" 이므로 세번째 element인 "abc"를 integer로 바꾸려고 한다면 runtime exception인 NumberFormatException이 발생하게 될 것입니다. 한편 for 문의 index가 i = 0 부터 3까지 반복하게 되는데 str array의 size가 3이고 index는 0부터 2까지 이므로 i = 3 일 때 ArrayIndexOutOfBoundsException이 발생하게 될 것입니다. 각 exception을 catch한 각각의 catch block에서 어떤 exception이 일어났다는 message를 print 해 주게 하였습니다. 그리고 finally block은 exception의 발생 여부와 상관없이 for문 반복 때 마다 계속 body의 마지막에 실행되게 됩니다. 이와 같이 각각의 서로 다른 exception들에 대해 다른 handling을 할 수 있도록 catch block을 여러번 사용할 수 있음에 유의하기 바랍니다.

Exception Catching Order

- All exception classes are descendants of java.lang.Exception
- Descendant exception class (specific exception) must be caught first
- Otherwise, the parent will catch all exceptions

```
try {  
    }  
    catch (Exception e) {  
    }  
    catch (ArrayIndexOutOfBoundsException e) {  
    }  
    catch (NumberFormatException e) {  
    }  
}
```

Catching all exceptions here

No chance to catch any exception

7

페이지 7

모든 exception은 java.lang.Exception의 descendant입니다.
따라서 catch block의 순서는
가장 하위의 descendant exception class를
가장 먼저 catch하게 하는 것이 좋습니다.
만약 그림과 같이 대문자 Exception class를
가장 먼저 catch하도록 해 놓는다면
polymorphism에 의해
모든 하위 descendant exception들도
이 첫번째 catch block에서 catch되고 말 것입니다.
이럴 경우 아래의 두 catch block은
하는 일이 없게 되고
존재할 필요도 없게 되는 것입니다.
따라서 이러한 경우를 방지하기 위해
상위의 exception들을
되도록 뒤쪽 catch block에서 catch 하도록 하는 것이 좋습니다.

Throwing Exception

- Two choices for exception handling code inside a method:
 - ① Use the try-catch block to handle the exception
 - ② Just throw the exception to the place where the method was called
- For throwing exception: Use **'throws'** keyword at the header of the method:

```
return_type method_name (parameters) throws SomeException1, SomeException2, ... {  
  
}
```

어떤 method 안에서 exception handling을 하는
두 가지 방법을 고려한다면
첫번째는 지금까지 보아 온 것처럼
try-catch block을 이용하는 handling 방법이 있고
두번째로는 현재 method를 call한
이전 method가 exception을 handling하도록
exception을 throw 하는 방법이 있습니다.
두번째 방법처럼 exception을 throw하기 위해서는
현재 method의 header에
'throws' keyword를 사용하여
이 method에서는 어떠한 exception을
처리하지 않고 throw한다는 것을 밝힐 필요가 있습니다.
이렇게 해 놓으면
나중에 다른 method에서 이 method를 call하는 경우
call한 method가 책임을 지고 그 throw된 method를
try-catch로 handling해 주거나
그것이 싫다면 다시 throw하여
자신을 call한 method로
handling 책임을 넘기도록 할 수 있는 것입니다.

Example: ThrowingExceptionDemo

```
public class ThrowingExceptionDemo {
    public static void main(String[] args) {
        String name = "java.lang.String2";
        try {
            // get 'Class' object having 'name'
            Class classObject = findClass(name);
        } catch (ClassNotFoundException e) {
            System.out.println("No class having name: " + name);
        }
    }

    static Class findClass(String name) throws ClassNotFoundException {
        Class classObject = Class.forName(name);
        return classObject;
    }
}
```

OUTPUT:

No class having name: java.lang.String2

9

페이지 9

예를 들어 ThrowingExceptionDemo 프로그램을 살펴 보겠습니다.
"java.lang.String2" 라는 String literal을 String name에 assign하였습니다.
이 프로그램의 의도는 이 name이라는 String의 이름을 가진 class가 존재할 때
그 class information을 대문자 Class의 object로 return받을 수 있는
Class.forName(name)을 실행하고자 하는 것입니다.
그러나 우리는 이미 String2 라는 class는 java.lang에 존재하지 않는다는 것을
잘 알고 있지요.
어쨌든 forName method call을 바로 하지 않고
findClass(name) 으로 findClass라는 method에서
forName을 call하도록 시도하였습니다.
findClass method에 가서 보니
Class.forName(name) 을 여기서 call하여
대문자 Class object를 return 받아
다시 main으로 return하도록 되어 있네요.
그런데 우리가 이미 공부했듯이
forName call을 할 때에는 ClassNotFoundException이 발생할 우려가 있고
이는 General Exception 이기 때문에
forName call이 exception을 발생할 때를 대비한
handling code를 함께 coding하지 않는다면
compile error가 나게 됩니다.
여기서 findClass method는 그 handling의 책임을
자신을 call한 main method로 다시 떠넘기기 위해
method header 뒤에 "throws ClassNotFoundException" 이라는
throws phrase를 coding하였습니다.
이 때문에 ClassNotFoundException 은
main에서 다시 catch하여 handling하여야 하며
그 때문에 try-catch block을 사용하게 된 것입니다.
그런데 이렇게 throws phrase를 method header에 추가해 주어야 하는 경우는
throw되는 exception이 General exception
즉, compiler checked exception 인 경우에만 강제성이 부과됩니다.
runtime exception, 즉, compiler unchecked exception의 경우에는
throws절을 반드시 써야 하는 것은 아니며
써도 되고 안써도 compile error는 나지 않습니다.
그러나 이 method가 exception을 handling 하지 않고
그냥 throw를 하기만 한다는 것을
다른 method들에 알리기 위해서는
throws절을 써 놓는 것이 더 좋다고 볼 수 있습니다.

이 프로그램의 output은 결국
catch block에서 print한 메시지가 되겠습니다.

Throw command

- Intentionally throwing an exception using `throw` command
- The exception can be handled inside the method or `throws` it to parent

```
public class ExceptionEx03 {  
    public static void main(String args[]) {  
        try {  
            Exception e = new Exception("My Exception");  
            throw e; // throw the exception  
        }  
        catch (Exception e) {  
            System.out.println("Error message: " + e.getMessage());  
            e.printStackTrace();  
        }  
        System.out.println("Program ended");  
    }  
}
```

10

페이지 10

이번에는 throw command를 알아보겠습니다.
주의할 점은 throw command와 앞 슬라이드에서 살펴본 throws phrase는 다른 것이라는 겁니다.
throw command는 프로그램이 고의적으로 exception을 발생시키기 위해 명령하는 것으로 error에 의해 발생하는 exception과 똑같이 try-catch block으로 handling하거나 자신을 call한 method로 throw되어 책임을 떠 넘기든지 해야 합니다.
ExceptionEx03 class 프로그램에서는 java.lang.Exception class object인 e를 생성하면서 메시지를 "MyException" 이라 주고나서 그 exception e를 throw 하였습니다.
이렇게 throw된 exception은 try block안에서 발생한 error에 의한 exception과 동일한 handling을 필요로 합니다.
따라서 catch block에서는 이 exception을 받아서 error message를 출력하였고 printStackTrace()를 call하여 method stack을 print하였습니다.
비록 이 프로그램에서 throw된 exception은 아무 이유없이 throw되었지만 이 throw exception을 잘 활용하면 여러가지 유용한 실행 sequence를 control할 수도 있습니다.

Example: ExceptionEx04

```
public class ExceptionEx04 {  
    public static void main(String[] args) {  
        try {  
            method1();  
            System.out.println(6);  
        } catch (Exception e) {  
            System.out.println(7);  
        }  
    }  
}
```

OUTPUT:

```
2  
4  
7
```

```
static void method1() throws Exception {  
    try {  
        method2();  
        System.out.println(1);  
    } catch (NullPointerException e) {  
        System.out.println(2);  
        throw e; // rethrow the exception  
    } catch (Exception e) {  
        System.out.println(3);  
    } finally {  
        System.out.println(4);  
    }  
    System.out.println(5);  
}  
static void method2() throws NullPointerException {  
    throw new NullPointerException();  
}  
}
```

11

페이지 11

throw와 throws를 이용했을 때
프로그램의 실행 순서가 어떻게 달라지는지
예를 들어 보도록 하겠습니다.
이 예제 프로그램은 ExceptionEx04라는 class인데
main, method1, method2의 세개의 class가 존재합니다.
main의 try block에서 method1을 call하였고
method1의 try block에서 다시 method2를 call하였습니다.
method2는 NullPointerException을 throws하도록 되어있고
아무 처리 없이 NullPointerException을 생성하여 throw하였습니다.
이 exception은 method2를 call한 method1에서 catch하여
handling해야 합니다.
NullPointerException을 catch한 후
2를 print합니다.
그리고 NullPointerException을 다시 throw합니다.
이렇게 catch한 exception을 다시 throw하는 경우로
'rethrow'라 부릅니다.
method1이 'throws Exception' phrase를 가지고 있기 때문에
이 rethrow된 exception은
method1을 call한 main이 handling하게 됩니다.
그런데 method1의 catch block 아래에
finally block이 있기 때문에
main으로 돌아가기 전에
finally block을 실행하게 되어
4가 프린트되게 됩니다.
한가지 주의할 점은
rethrow된 exception e를
method1의 두번째 catch block인
catch (Exception e)에서는 catch하지 못한다는 것입니다.
이것은 try-catch-finally block이 한 묶음이고
중간에 있는 catch block들 중에서는
딱 하나의 catch block만 실행된 후
finally로 진행하기 때문입니다.
결국 rethrow된 exception은 main으로 돌아오게 되고
catch block이 실행되어
7이 프린트 되게 됩니다.
최종적으로 output은 2 4 7 이 됩니다.

User Defined Exception

- Custom exception class defined by programmer
- To handle exceptions that are not provided by Java standard library
- By inheriting from the standard exception classes
 - extends Exception: compiler checked exception
 - extends RuntimeException: compiler unchecked exception
- Constructor
 - Passing more specific exception messages

12

페이지 12

Java의 exception에는 유용한 user defined exception 또는 programmer defined exception이라고 불리는 기능이 있습니다. 이것은 programmer가 자신이 새로운 custom exception class를 만들어 사용하는 기능입니다. 만약에 Java에서 이미 제공하는 library중에 찾고 있는 적절한 exception class가 없다면 새로운 exception class를 standard exception class를 inherit하여 생성해 낼 수 있습니다. 만약 대문자 Exception class를 extend한다면 compiler checked exception을 새로 만들 수 있고 RuntimeException class를 extend한다면 compiler unchecked exception을 새로 만들 수 있습니다. 한편 user defined exception class의 constructor를 이용하면 적절한 메시지를 exception class가 가질 수 있도록 프로그래밍할 수 있습니다.

Example: User Defined Exception (1/3)

```
import java.util.Scanner;

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

public class ExceptionBasedInputLoop {

    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        int number = 0;
        boolean validInput = false;
        String input = null;
    }
}
```

13

페이지 13

이제 user defined exception의 예제 프로그램을 보겠습니다.
Scanner를 사용한 input을 위해서 java.util.Scanner class를 import하였습니다.
Exception class를 inherit하여
User defined exception으로
InvalidInputException class를 define 하였습니다.
constructor에서 String message를 받아
super(message)를 call함으로써
exception 발생시 가져올 수 있는 message를 저장하였습니다.
Main class로 ExceptionBasedInputLoop를 정의합니다.
main method에서는 먼저
input을 위한 Scanner object를 생성하고
integer input을 받을 number를 0으로 초기화 하며
valid input이 들어왔는지를 알려줄
validInput boolean 변수를 false로 초기화 하고
input을 위한 String 변수 input을 null로 초기화 하였습니다.

Example: User Defined Exception (2/3)

```
while (!validInput) {  
    try {  
        System.out.print("Please enter a positive odd integer: ");  
        input = scanner.nextLine();  
        number = Integer.parseInt(input); // convert String to int  
        if (number <= 0) { // negative integer  
            throw new InvalidInputException("Negative integer");  
        }  
        else if (number % 2 == 0) { // even number  
            throw new InvalidInputException("Not odd integer");  
        }  
        validInput = true; // exit from the loop if valid input  
    } catch (InvalidInputException e) {  
        System.out.println("Invalid input: " + e.getMessage());  
    } catch (NumberFormatException e) {  
        System.out.println("Invalid input: Not a valid integer");  
    }  
}
```

14

페이지 14

이 프로그램은 양의 홀수 정수가 제대로 input될 때까지 계속 scanner input을 반복하는데 양의 홀수 정수라는 조건을 테스트하기 위해 Exception handling을 사용하고 있습니다. 이러한 방식을 Exception based input loop 라고 부릅니다. while문은 validInput이 false인 동안 계속 반복이 유지됩니다. try block 안에서 "Please enter a positive odd integer: " 라는 prompt를 프린트하여 보여주고 scanner.nextLine()으로 한 라인 전체를 String으로 input 받습니다. 이 input String을 Wrapper class인 Integer.parseInt(input) 으로 int type의 number로 conversion을 시도 합니다. 만약 이 conversion에서 input String이 int로 바뀔 수 없는 것이라면 예를 들어 "abcd" 나 "123.45" 같은 것이라면 프로그램은 NumberFormatException을 발생시키고 try block의 실행은 이 지점에서 stop한 후 NumberFormatException을 catch하는 catch block으로 건너뛰게 됩니다. 만약 input을 정수로 바꾸는데 성공했을 때 이 때 number가 0이하의 정수이면 "Negative Number" 라는 메시지를 가진 user defined exception class인 InvalidInputException을 throw합니다. throw한 후 더 진행하지 않고 catch (InvalidInputException) block을 실행하게 됩니다. negative integer, 또는 0이 아니라면 number는 양의 정수인데 우리는 그 중에서도 홀수를 입력받고 싶어하는 것이니까 number가 짝수인지를 테스트 합니다. number를 2로 나눈 나머지가 0이면 짝수입니다. 짝수 이면 "Not odd integer" 라는 message와 함께 InvalidInputException object를 생성하여 throw합니다. 그리고 나서 프로그램은 catch(InvalidInputException e) block으로 진행합니다. 만약 number가 음의정수도, 0도, 양의짝수도 아니라면 number는 우리가 원하는 양의 홀수 정수 일 것입니다. 이 때에는 validInput이 true로 assign되고 while문을 빠져나올 수 있게 됩니다.

Example: User Defined Exception (3/3)

```
        System.out.println("You entered a valid positive integer: " + number);
        scanner.close();
    }
}
```

Please enter a positive odd integer: a9832
Invalid input: Not a valid integer
Please enter a positive odd integer: -253
Invalid input: Negative integer
Please enter a positive odd integer: 2982
Invalid input: Not odd integer
Please enter a positive odd integer: 980751
You entered a valid positive integer: 980751

입력의 예를 보면 처음에 a9832를 입력했을때에는
NumberFormatException이 발생하면서
"Invalid input: Not a valid integer" 라는 메시지가 표시됩니다.
다시 받은 입력이 이번에 -253 인데
"Negative integer" 라는 메시지가 표시되고
다음 입력은 2982 인데
"Not odd integer" 라는 메시지가 표시됩니다.
980751 이라는 양의 홀수 정수가 입력되면
while loop가 끝나고 프로그램이 종료됩니다.