

객체지향 프로그래밍 및 실습

9주차. Exception Handling

1. What is Exception

■ Exception

- The term *exception* is shorthand for the phrase “exceptional event”
- Exception은 이벤트의 일종으로써, 프로그램이 실행되는 도중에 발생하여 프로그램의 명령 실행을 방해한다.
- Exception이 발생하면 *Exception object*를 생성하여 런타임 시스템에 전달한다.
 - 사용자의 잘못된 조작 또는 개발자의 잘못된 코딩으로 인해 발생하는 프로그램 오류
 - 예외가 발생했을 때 예외를 처리하지 않으면 프로그램은 곧바로 종료
 - 그러나 예외를 처리를 통해 프로그램을 종료하지 않고 정상 실행 상태를 유지할 수 있다.

1. What is Exception

■ Exception

- The term *exception* is shorthand for the phrase “exceptional event”
- Exception은 이벤트의 일종으로써, 프로그램이 실행되는 도중에 발생하여 프로그램의 명령 실행을 방해한다.

```
1 int val = 9 / 0;    // ← Exception occurs
2
3 System.out.println(val);
```


Exception이 발생한 Thread	발생한 Exception (Exception object type)	Exception message
Exception in thread "main"	java.lang.ArithmeticException:	/ by zero
at arith.ArithTest.main(ArithTest.java:5)		

Exception을 유발한 명령의 위치

1. What is Exception

■ 실습 문제 1

- ExceptionFindTest 클래스의 main메서드에 아래 코드를 작성해보자.
- 프로그램을 실행시켜보고, 발생한 에러메시지를 분석해보자.
- 주석으로 발생한 Exception의 종류, 위치 등등을 적어보도록 한다.

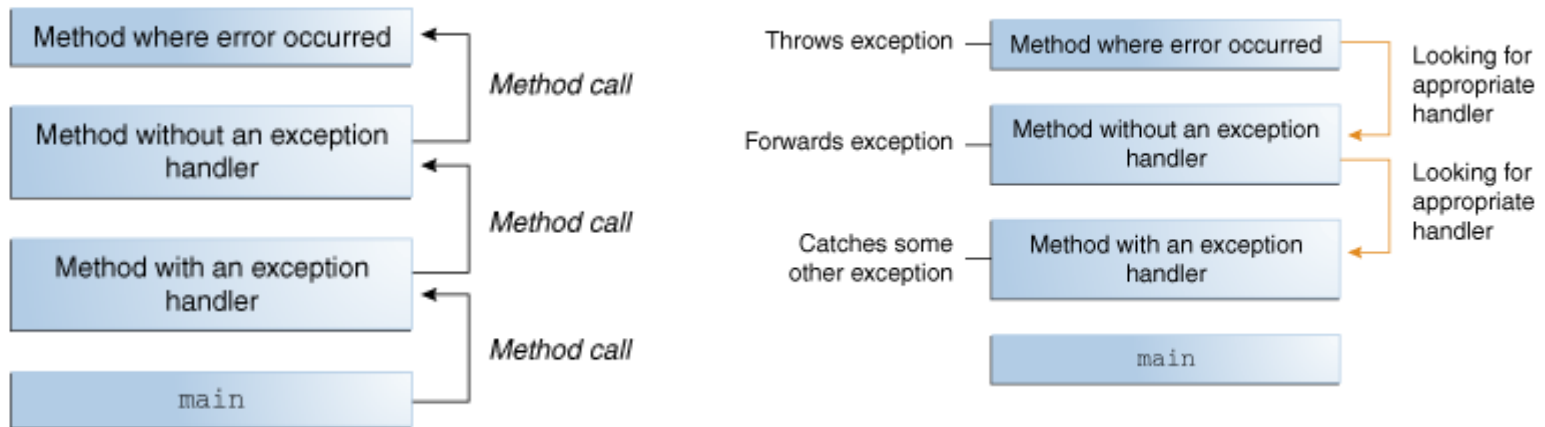


```
1 System.out.println(Integer.parseInt("190"));
2 System.out.println(Integer.parseInt("-55"));
3 System.out.println(Integer.parseInt(new String("Hello")));
4 System.out.println(Integer.parseInt(new String("50")));
```

1. What is Exception

■ Exception Stack Trace

- Exception이 발생하면, Exception Object가 생성된다.
- Exception Object는 에러에 대한 정보를 담고 있다. - 에러 타입, 에러 발생 당시의 프로그램 상대 등
- 런타임 시스템은 Exception을 핸들링하는 부분을 찾으려고 한다.



```
Exception in thread "main" java.lang.NumberFormatException: For input string: "Hello"
at java.base/java.lang.NumberFormatException.forInputString(NumberFormatException.java:65)
at java.base/java.lang.Integer.parseInt(Integer.java:652)
at java.base/java.lang.Integer.parseInt(Integer.java:770)
at arith.IATest.main(IATest.java:7)
```

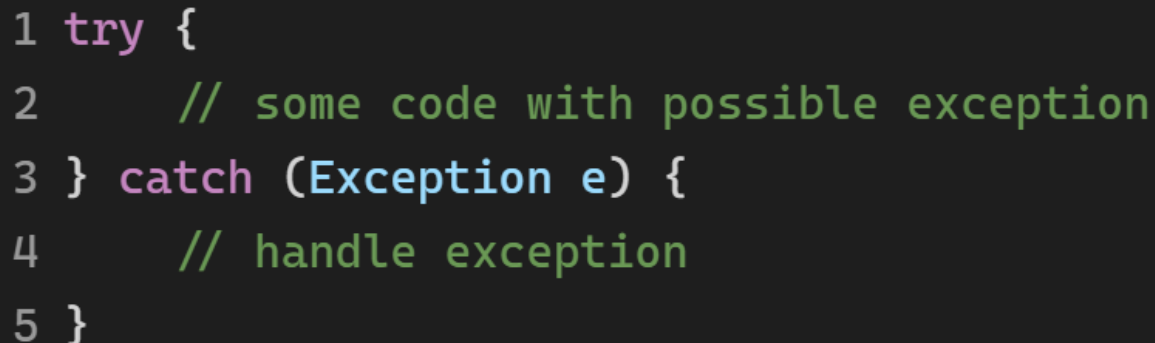
Call stack

2. Exception Handling

- Exception Handler

- Try-Catch

- Try: 예외가 발생할 수 있는 코드가 위치
 - 만약 Try 블록 실행 도중에 exception이 발생하면, catch 블록으로 흐름이 이전됨
 - Catch: 예외를 핸들링하기 위한 코드가 위치
 - Catch 옆의 괄호 -> 처리할 Exception의 타입
 - e -> Exception Object



```
1 try {  
2     // some code with possible exception  
3 } catch (Exception e) {  
4     // handle exception  
5 }
```

2. Exception Handling

- 실습 문제 2

- 실습 문제1의 코드를 try-catch 통해 handling 해보자.
- Catch block에서는 Exception이 발생했습니다!를 출력한다.

2. Exception Handling

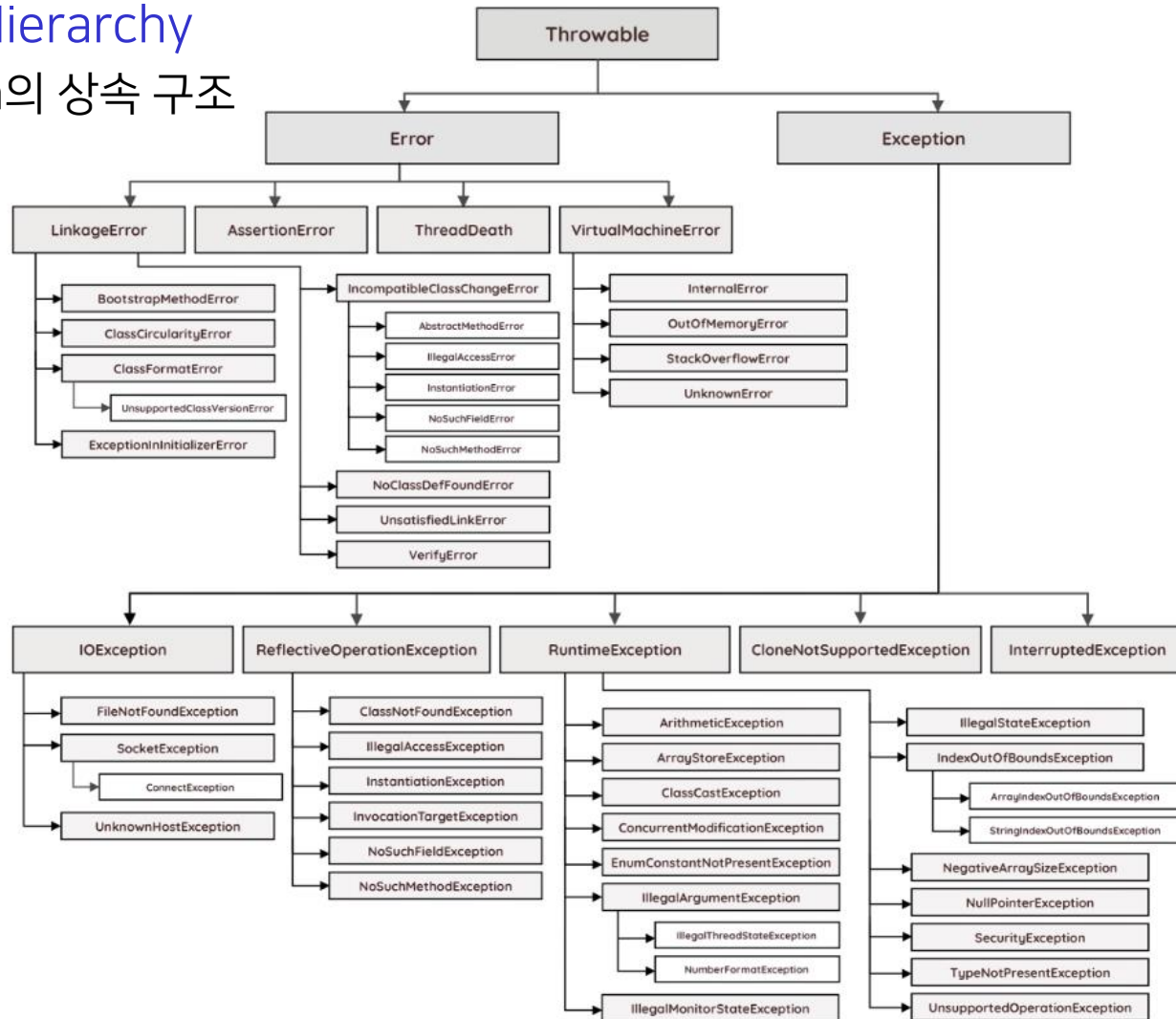
■ 실습 문제 3

- 실습 문제2의 코드에서 Exception Object를 활용해 error message를 출력하고, stack trace를 출력해보자!

Modifier and Type	Method	Description
void	<code>addSuppressed(Throwable exception)</code>	Appends the specified exception to the exceptions that were suppressed in order to deliver this exception.
Throwable	<code>fillInStackTrace()</code>	Fills in the execution stack trace.
Throwable	<code>getCause()</code>	Returns the cause of this throwable or null if the cause is nonexistent or unknown.
String	<code>getLocalizedMessage()</code>	Creates a localized description of this throwable.
String	<code>getMessage()</code>	Returns the detail message string of this throwable.
StackTraceElement[]	<code>getStackTrace()</code>	Provides programmatic access to the stack trace information printed by <code>printStackTrace()</code> .
Throwable[]	<code>getSuppressed()</code>	Returns an array containing all of the exceptions that were suppressed, typically by the try-with-resources statement, in order to deliver this exception.
Throwable	<code>initCause(Throwable cause)</code>	Initializes the <i>cause</i> of this throwable to the specified value.
void	<code>printStackTrace()</code>	Prints this throwable and its backtrace to the standard error stream.
void	<code>printStackTrace(PrintStream s)</code>	Prints this throwable and its backtrace to the specified print stream.
void	<code>printStackTrace(PrintWriter s)</code>	Prints this throwable and its backtrace to the specified print writer.
void	<code>setStackTrace(StackTraceElement[] stackTrace)</code>	Sets the stack trace elements that will be returned by <code>getStackTrace()</code> and printed by <code>printStackTrace()</code> and related methods.
String	<code>toString()</code>	Returns a short description of this throwable.

2. Exception Handling

- Exception Hierarchy
 - Exception의 상속 구조



2. Exception Handling

■ Exception Hierarchy

- Catch block의 인자 타입에 따라 catching할 수 있는 Exception의 종류가 달라짐.
- (발생한 exception 타입) is-a (인자 타입)인 경우에만 Catching 할 수 있다.
- 따라서 모든 예외를 catch 하고 싶다면 "Exception" 타입
- 특정 예외만 catch 하고 싶다면 특정 타입을 사용하는 것이 옳다.

```
1 try {  
2     // some code with possible exception  
3 } catch (Exception e) {  
4     // handle exception  
5 }
```

2. Exception Handling

■ 실습 문제 4

- 아래 코드에서 의도적으로 예외 2가지를 일으킬 수 있다.
 - 사례 1) 0으로 나누도록 유도하는 경우
 - 사례 2) 정수가 아닌 문자를 입력하는 경우
- 위 두가지 예외 중 사례 1만을 핸들링할 수 있도록 개발해보자.
- HandlingTest 클래스의 main메서드 이용.

```
1 Scanner scan = new Scanner(System.in);
2
3 String a = scan.nextLine();
4 String b = scan.nextLine();
5
6 int one = Integer.parseInt(a);
7 int two = Integer.parseInt(b);
8
9 System.out.println(one / two);
10
11 scan.close();
```

2. Exception Handling

- Different ways to use Catching

- 각각 다른 예외를 처리하는 catch 블록을 구성할 수도 있음.

```
1 int[] array = {5, 1, 3, 7};
2 Scanner scan = null;
3
4 try {
5     // array[15] = 100;
6     // scan.nextLine();
7 } catch (ArrayIndexOutOfBoundsException e) {
8     System.out.println(e.getMessage());
9 } catch (NullPointerException e) {
10     System.out.println(e.getMessage());
11 }
```

2. Exception Handling

- Different ways to use Catching
 - 여러 예외를 한번에 묶어서 처리할 수도 있음

```
1 try {  
2     // array[15] = 100;  
3     // scan.nextLine();  
4 } catch (ArrayIndexOutOfBoundsException | NullPointerException e) {  
5     System.out.println(e.getMessage());  
6 }
```

2. Exception Handling

■ 실습 문제 5

- 실습 문제 4의 코드를 catch block을 여러 개 사용하여 사례 1과 사례 2를 각각 처리할 수 있도록 프로그램을 작성해보자.

```
1 Scanner scan = new Scanner(System.in);
2
3 String a = scan.nextLine();
4 String b = scan.nextLine();
5
6 int one = Integer.parseInt(a);
7 int two = Integer.parseInt(b);
8
9 System.out.println(one / two);
10
11 scan.close();
```

2. Exception Handling

■ 실습 문제 6

- 실습 문제 5의 코드를 주석처리 하고, 그 아랫줄에 새로운 try-catch를 생성해보자.
- 이때, catch block을 단 한 개만 사용하여 사례 2가지를 처리할 수 있도록 개발한다.
- Exception 타입을 제외한 다른 타입을 사용해야한다.

```
1 Scanner scan = new Scanner(System.in);
2
3 String a = scan.nextLine();
4 String b = scan.nextLine();
5
6 int one = Integer.parseInt(a);
7 int two = Integer.parseInt(b);
8
9 System.out.println(one / two);
10
11 scan.close();
```

2. Exception Handling

■ Finally block

- Finally block은 try 블록을 빠져나갈 때 언제나 무조건 실행된다.
- 따라서 Exception이 발생하더라도 finally 블록이 실행된다.
- Exception handling의 측면 외에서도 cleaning up 코드가 return, continue, break 키워드를 통해 bypass되더라도 실행된다는 것을 보장할 수 있음.

```
1 public int getIntInput(Scanner scan) {
2     try {
3         int key = scan.nextInt();
4         return key;
5     } catch (InputMismatchException e) {
6         System.out.println("정수를 입력해 주세요!");
7     } finally {
8         scan.close();
9     }
10 }
```

```
finally {
    if (out != null) {
        System.out.println("Closing PrintWriter");
        out.close();
    } else {
        System.out.println("PrintWriter not open");
    }
    if (f != null) {
        System.out.println("Closing FileWriter");
        f.close();
    }
}
```


2. Exception Handling

- Checked & Unchecked Exception
 - Checked Exception은 반드시 handle 되어야 하는 예외를 의미함
 - `IOException`, `InterruptedException` 등..
 - Unchecked Exception은 handle 될 필요가 없는 예외를 의미함
 - `RuntimeException`
 - Unchecked Exception은 `RuntimeException` 타입
 - Checked Exception은 `RuntimeException`을 제외한 Exception 타입
 - `FileReader fr = new FileReader("./Hello.txt");`
 - `fr`은 `Hello.txt` 파일을 읽어오는 Reader 객체.
 - 위 코드를 직접 이클립스에 작성해보자.
 - 빨간 줄이 뜨며, 예외를 처리하라는 메시지가 보일 것임

2. Exception Handling

■ Exceptions thrown by method

- Exception을 내가 책임질 필요가 없는 경우에는
- Exception을 try-catch로 handle 하지 않고, 떠넘길 수 있음.

```
1 public void writeList() throws IOException, ArrayIndexOutOfBoundsException {  
2     PrintWriter out = new PrintWriter(new FileWriter("OutFile.txt"));  
3     for (int i = 0; i < SIZE; i++) {  
4         out.println("Value at: " + i + " = " + list.get(i));  
5     }  
6     out.close();  
7 }
```

- new FileWriter()는 IOException - Checked Exception을 throws 한다.
- 이를 try-catch로 처리하지 않고, throws 키워드로 예외를 넘길 수 있다.

2. Exception Handling

■ Exception Throwing

- 필요에 따라 내가 직접 예외를 발생시켜야 할 상황도 있다!
- Throw 키워드 뒤에, 발생시킬 exception 객체를 위치시킨다.
- Exception 생성자의 String 인자는 exception의 message 역할
- 만약 throw 할 예외가 Checked exception이라면 throws 키워드도 필수

```
1 public String getParsedPhoneNumber(String phoneNumber) throws SomeException {
2     if (phoneNumber.length() != 13) {
3         throw new IllegalArgumentException("Phone Number는 13자리여야합니다.");
4     }
5
6     if (!phoneNumber.startsWith("010")) {
7         throw new SomeException("Phone Number는 010으로 시작해야합니다!");
8     }
9
10    String parsed = null;
11
12    // Parse Phone number ...
13    return parsed;
14 }
```

2. Exception Handling

■ Chained Exception

- A라는 예외가 발생했을 때, 이것이 원인이 되어 B라는 예외가 발생할 수 있다.
- 이를 Chained Exception이라고 하며, A를 B의 cause exception이라고 부름.

```
1 public String getDataFromInternet(String uri) throws CannotDownloadException {
2     String data = null;
3     try {
4         InternetDownloader id = new InternetDownloader();
5         data = id.getTextData();
6     } catch (InternetDisconnetcedException e) {
7         throw new CannotDownloadException("Internet 연결이 불가능합니다.", e);
8     }
9     return data;
10 }
```

3. User Defined Exception

- Exception 타입을 생성하는 방법
 - 필요에 따라서 exception 타입을 생성할 수도 있다.
 - 기본적으로 제공되는 exception으로는 우리가 만드는 프로그램의 모든 예외를 표현할 수 없다.
 - 아래 사항 중 해당되는 부분이 있다면, 새로운 예외 타입을 생성하는 것을 추천.
 - 만약 아니라면, 기본적으로 정의된 타입을 사용하는 것을 추천.
 - Do you need an exception type that isn't represented by those in the Java platform?
 - Would it help users if they could differentiate your exceptions from those thrown by classes written by other vendors?
 - Does your code throw more than one related exception?
 - If you use someone else's exceptions, will users have access to those exceptions?
A similar question is, should your package be independent and self-contained?

3. User Defined Exception

■ 적당한 Super Class 선택하기

- 사용자 정의 예외는 예외 클래스를 상속함으로써 만들 수 있다.
- 다만! 단순히 "Exception" 클래스를 상속하는 것 보다는 알맞은 예외 클래스를 선정하는 것이 중요하다.
- `PrinterNotFoundException` - example
 - 프린터로 문서를 출력하려 했는데, 프린터와 연결되지 않았을 때 발생하는 예외
 - I/O 상황에 발생한 예외이니, `IOException`을 상속하는 것이 적당하다
- `DerivativeNotDefinedException` - example
 - `get미분값(double x)` 메서드에서 `x` 점에서의 미분 값이 정의되지 않았을 때 발생하는 예외
 - 굳이 `checked exception`이어야 할 정도로 꼭 처리되어야 하는 예외가 아니므로, `RuntimeException` 또는 `ArithmeticException`을 상속하는 것이 적당하다

3. User Defined Exception

■ 실습 문제 7

- 아래 코드가 작동할 수 있도록 BalanceException을 만들어보자.

```
public class Account {  
    private double balance;  
  
    public void deposit(int money) {  
        balance += money;  
    }  
  
    public void withdraw(int money) throws BalanceException {  
        if(balance < money) {  
            throw new BalanceException("잔고 부족: " + (money - balance) + "원 부족");  
        }  
        balance -= money;  
    }  
}
```

예외 발생

```
public class AccountTest {  
    public static void main(String[] args) {  
        Account account = new Account();  
  
        // 예금  
        account.deposit(10000);  
  
        // 출금  
        try {  
            account.withdraw(30000);  
        } catch (BalanceException e) {  
            e.printStackTrace();  
        }  
    }  
}
```

예외 catch

```
<terminated> AccountTest [Java Application] C:\Users\김성준\Downloads\ eclipse-jee-2020-12-R-wi  
week8.BalanceException: 잔고 부족: 20000.0원 부족  
    at week8.Account.withdraw(Account.java:12)  
    at week8.AccountTest.main(AccountTest.java:12)
```

예외 발생 위치 출력