



chapter **06.**

Package 및 Exception

○ 패키지란?

- 자바에서 이야기하는 패키지는 서로 관련 있는 클래스와 인터페이스를 하나의 단위로 묶는 것을 의미하며, 일종의 Library(자료실)라고 할 수 있다.

○ 패키지 선언 방법

- 패키지(package)선언은 주석문을 제외하고 반드시 소스파일의 첫 줄에 와야 한다. 다음은 패키지 선언법이다.

| | |
|--|------------------------------|
| | <code>package</code> 패키지경로명; |
|--|------------------------------|

- 다음 예제들을 C:\JavaEx\WPackTest라는 폴더에 또는 독자가 기억할 만한 곳에 저장만 해 두도록 하자!

[예제6-1] MyPackOne.java

```
01 package myPack.p1;  
02 public class MyPackOne{  
03  
04     public void one(){  
05         System.out.println("MyPackOne클래스의 one메서드");  
06     }  
07 }
```

[예제6-2] MyPackTwo.java

```
01 package myPack.p1;  
02 public class MyPackTwo{  
03  
04     public void two(){  
05         System.out.println("MyPackTwo클래스의 two메서드");  
06     }  
07 }
```

- 앞선 두 개의 파일이 저장된 곳으로 cmd창을 연다. 그리고 다음과 같이 컴파일을 수행하고 다음의 표를 참조하여 옵션에 대한 설명을 확인해 보자!

```

C:\WINDOWS\system32\cmd.exe
Microsoft Windows XP [Version 5.1.2600]
(C) Copyright 1985-2001 Microsoft Corp.

C:\Documents and Settings\Michael>cd C:\JavaEx\PackTest

C:\JavaEx\PackTest>javac -d . MyPackOne.java
C:\JavaEx\PackTest>javac -d . MyPackTwo.java
C:\JavaEx\PackTest>
  
```

[표 6-1] 패키지 컴파일 시 옵션

| 옵션 | 설명 |
|---------------------------|--|
| javac [옵션] [작업위치] [소스파일명] | -d: 디렉토리(패키지)를 의미하며 컴파일되어 생기는 클래스 파일이 저장될 위치를 말하는 것이다. 즉, 디렉토리(패키지) 작업을 의미하는 것이며 디렉토리(패키지)를 만들어야 한다면 만들라는 뜻이다. |
| | . : 디렉토리(패키지) 작업은 바로 현재 디렉토리에서 하라는 뜻이다. |

○ 패키지 사용 방법

- 패키지에 있는 특정한 클래스를 사용하려면 [import문]을 사용해야 한다. import는 수입이란 의미가 되므로 현재 객체에서 원하는 다른 객체를 가져다 사용할 때 사용한다. 자바 인터프리터는 import하는 패키지 경로의 클래스들은 1장에서 CLASSPATH 환경변수에 지정된 경로에서 검색한다. package문 아래에 오는 것이 기본 방법이며 구성은 다음과 같다.

```
import [패키지경로.클래스명]; 또는 import [패키지경로.*];
```

[예제6-3] MyPackTest.java

```
01 import myPack.p1.MyPackOne;
02 import myPack.p1.MyPackTwo;
03 class MyPackTest {
04
05     public static void main(String[] args) {
06         MyPackOne myOne = new MyPackOne();
07         myOne.one();
08         MyPackTwo myTwo = new MyPackTwo();
09         myTwo.two();
10     }
11 }
```

실행결과

```
----- Java Run -----
MyPackOne클래스의 one메서드
MyPackTwo클래스의 two메서드
Normal Termination
출력 완료 (0초 경과).
```

○ 예외

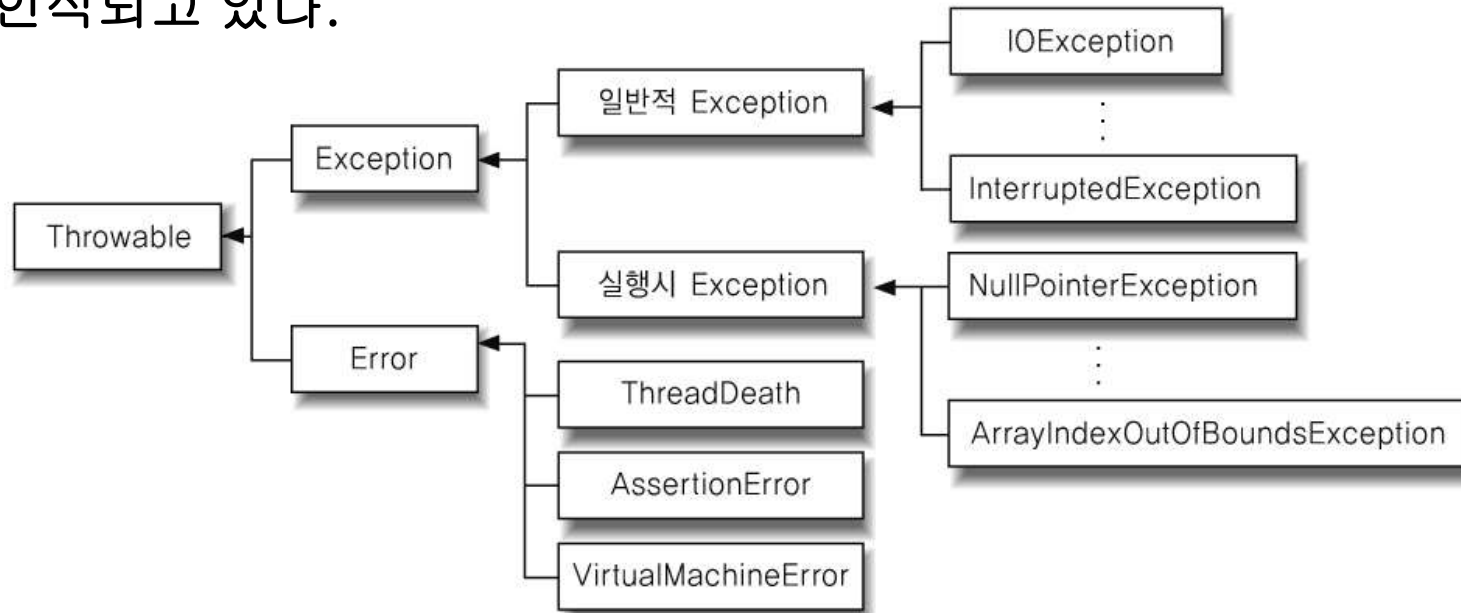
- 예를 들자면 지하철을 탈 때까지만 해도 날씨가 좋았는데 도착하여 지하철역에서 나오려니 비가 내리고 있다던가 또는 책을 보다가 종이에 손을 베었다던가 하는 뜻하지 않은 일들을 당하게 된다. 이렇게 예상하지 못한 일들을 ‘예외’라 하고 이를 대비하고 준비하는 것이 바로 ‘예외처리’다.

○ 예외처리에 대한 필요성과 이해

- 자바에서 프로그램의 실행하는 도중에 예외가 발생하면 발생한 그 시점에서 프로그램이 바로 종료가 된다. 때에 따라서는 예외가 발생 했을 때 프로그램을 종료시키는 것이 바른 판단일 수도 있다. 하지만 가벼운 예외이거나 예상을 하고 있었던 예외라면 프로그램을 종료시키는 것이 조금은 가혹(?)하다고 느껴진다. 그래서 ‘예외처리’라는 수단(mechanism)이 제안되었고 예외 처리를 통해 우선 프로그램의 비 정상적인 종료를 막고 발생한 예외에 대한 처리로 정상적인 프로그램을 계속 진행할 수 있도록 하는 것이 예외처리의 필요성이라 할 수 있다.

○예외의 종류

- 자바에서 발생하는 모든 예외는 다음과 같은 구조를 이루면서 각각 객체로 인식되고 있다.



[표 6-2] 오류의 구분

| 오류구분 | 설명 |
|---------------|---------------------------|
| 예외(Exception) | 가벼운 오류이며 프로그램적으로 처리한다. |
| 오류(Error) | 치명적인 오류이며 JVM에 의존하여 처리한다. |

- 예외 처리를 하지 않았을 때의 예제

[예제6-5] ExceptionEx1.java

```
01 import static java.lang.System.out;
02 class ExceptionEx1 {
03
04     public static void main(String[] args) {
05         int[] var = {10,200,30};
06         for(int i=0 ; i <= 3 ; i++)
07             out.println("var["+i+"] : "+var[i]);
08
09         out.println("프로그램 끝!");
10     }
11 }
```

실행결과

```
----- Java Run -----
var[0] : 10
var[1] : 200
var[2] : 30
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 3
    at ExceptionEx1.main(ExceptionEx1.java:7)
Normal Termination
출력 완료 (0초 경과).
```


- 앞선 예제가 배열의 범위를 벗어나 `ArrayIndexOutOfBoundsException`이 발생한 것을 확인할 수 있다. 이를 try~catch문을 사용하여 ‘예외처리’를 수행해 보자! 다음은 구성이다.


```
try{  
    // 예외가 발생 가능한 문장들;  
}catch(예상되는_예외객체 변수명){  
    // 해당 예외가 발생했을 시 수행할 문장들;  
}
```

- 위의 구성과 같이 try brace에 정의되는 문장은 예외가 발생 가능한 문장들을 기재하는 곳이니 주의 하기 바라며 현 try brace는 반드시 하나 이상의 catch brace 아니면 finally brace가 같이 따라 줘야 한다. 그럼 앞의 [ExceptionEx1] 예제를 수정하여 다시 작성해 보자!

[예제6-6] ExceptionEx2.java

```
01 import static java.lang.System.out;
02 class ExceptionEx2 {
03
04     public static void main(String[] args) {
05         int[] var = {10,200,30};
06         for(int i=0 ; i <= 3 ; i++){
07             try{
08                 out.println("var["+i+"] : "+var[i]);
09             }catch(ArrayIndexOutOfBoundsException ae){
10                 out.println("배열을 넘었습니다.");
11             }
12         }//for의 끝
13
14         out.println("프로그램 끝!");
15     }
16 }
```

실행결과



```
----- Java Run -----
var[0] : 10
var[1] : 200
var[2] : 30
배열을 넘었습니다.
프로그램 끝!
Normal Termination
출력 완료 (0초 경과).
```

- try ~ catch문에서의 주의 사항

아무리 try ~ catch문으로 ‘예외 처리’를 했다 해도 모든 것이 해결되는 것은 아니다. 다음 예문을 살펴 보자!

```
07 try{
08   out.println((i+1)+"번째");
09   out.println("var["+i+"] : "+var[i]);
10   out.println("~~~~~"); //수행을 못할 수도 있음!
11 }catch(ArrayIndexOutOfBoundsException ae){
12   out.println("배열을 넘었습니다.");
13 }
```

- 위의 8번 Line과 10번 Line의 문장을 추가한 후 프로그램을 다시 컴파일하고 실행해 보자!

총 반복문이 4번을 반복하게 되는데 4번째 반복 수행을 8번 행의 출력으로 알 수 있지만 예외가 발생하는 9번 행을 만나면서 10번 행을 수행하지 못하고 바로 catch영역을 수행함을 알 수 있다.

- 다중 catch문

다중 catch문은 하나의 try문 내에 여러 개의 예외가 발생 가능할 때 사용한다. 구성은 다음과 같다.

```
try{
    // 예외가 발생 가능한 문장들;
} catch(예상되는_예외객체1 변수명){
    // 해당 예외가 발생했을 시 수행할 문장들;
} catch(예상되는_예외객체2 변수명){
    // 해당 예외가 발생했을 시 수행할 문장들;
} catch(예상되는_예외객체3 변수명){
    // 해당 예외가 발생했을 시 수행할 문장들;
}
```

[예제6-7] ExceptionEx3.java

```
01 import static java.lang.System.out;
02 class ExceptionEx3 {
03
04     public static void main(String[] args) {
05         int var = 50;
06         try{
07             int data = Integer.parseInt(args[0]);
08
09             out.println(var/data);
10         }catch(NumberFormatException ne){
11             out.println("숫자가 아닙니다.");
12         }catch(ArithmeticException ae){
13             out.println("0으로 나눌 순 없죠?");
14         }
15         out.println("프로그램 종료!");
16     }
17 }
```

실행결과

```
----- Java Run -----
0으로 나눌 순 없죠?
프로그램 종료!
Normal Termination
출력 완료 (0초 경과).
```

■ 다중 catch문의 주의 사항

일반적 예외(Exception)에서 가장 상위(parent) 클래스가 Exception이다. 그러므로 가장 아래쪽에 정의 해야 한다. 이렇게 하는 이유는 예외는 상위(parent) 클래스가 모든 예외를 가지고 있으므로 가장 위에 정의를 하게 되면 모든 예외를 처리하게 되므로 두 번째 catch문부터는 절대로 비교 수행할 수 없게 된다.

```
try{
    수행문1;
    수행문2;
    수행문3;
    ...
    수행문n;
}catch(예상되는_예외객체1 변수명){
    ...;
}catch(예상되는_예외객체2 변수명){
    ...;
}catch(예상되는_예외객체3 변수명){
    ...;
}
```



- throws예약어

예외를 처리하기 보다는 발생한 예외 객체를 양도하는 것이다. 즉, 현재 메서드에서 예외처리를 하기가 조금 어려운 상태일 때 현재 영역을 호출해준 곳으로 발생한 예외 객체를 대신 처리해 달라며 양도 하는 것이다. 사용법은 다음의 구성과 같이 throws라는 예약어를 통해 메서드를 선언하는 것이다.

[접근제한] [반환형] [메서드명](인자1, ...인자n) throws 예외클래스1,...예외클래스n{}

```
04      public void setData(String n) throws NumberFormatException{
05          if(n.length() >= 1){
06              String str = n.substring(0,1);
07              printData(str);
08          }
09      }
```

- 이렇게 [throws]를 사용하여 발생한 예외객체의 양도는 것은 어디까지나 양도이지 예외에 대한 처리는 아니다. 양도를 받은 곳에서도 다시 양도가 가능하지만 언젠가는 try~catch문으로 해결을 해야 프로그램의 진행을 계속 유지 할 수 있음을 잊지 말자!

○finally의 필요성

- 예외가 발생하든 발생하지 않든 무조건 수행하는 부분이 바로 finally 영역이다. 이것은 뒤에서 Database처리나 File처리를 한다면 꼭 필요한 부분이다. 이유는 Database를 열었다거나 또는 File을 열었다면 꼭 닫아주고 프로그램이 종료되어야 하기 때문이다.
- 다음은 구성이다.

```
try{  
    // 예외가 발생 가능한 문장들;  
} catch(예상되는_예외객체1 변수명){  
    // 해당 예외가 발생했을 시 수행할 문장들;  
} finally{  
    // 예외발생 여부와 상관없이 수행할 문장들;  
}
```


[예제6-9] FinallyEx1.java

```
01 import static java.lang.System.out;
02 class FinallyEx1 {
03
04     public static void main(String[] args) {
05         int[] var = {10,200,30};
06         for(int i=0 ; i <= 3 ; i++) {
07             try{
08                 out.println((i+1)+"번째");
09                 out.println("var["+i+"] : "+var[i]);
10                 out.println("~~~~~");
11             }catch(ArrayIndexOutOfBoundsException ae){
12                 out.println("배열을 넘었습니다.");
13                 return;
14             }finally{
15                 out.println(":::::: Finally :::::");
16             }
17         } //for의 끝
18
19         out.println("프로그램 끝!");
20     }
21 }
```

실행결과

```
----- Java Run -----
1번째
var[0] : 10
~~~~~
::: Finally :::
2번째
var[1] : 200
~~~~~
::: Finally :::
3번째
var[2] : 30
~~~~~
::: Finally :::
4번째
배열을 넘었습니다.
::: Finally :::
프로그램 끝!
Normal Termination
출력 완료 (0초 경과).
```

:: 이렇게 해서 finally영역은 예외가 발생하든 하지 않든 무조건 수행함을
알아 보았다.

○ 사용자 정의 예외

- 사용자 정의 Exception이 필요한 이유는 표준예외가 발생할 때 예외에 대한 정보를 변경하거나 정보를 수정하고자 한다면 사용자가 직접 작성하여 보안된 예외를 발생시켜 원하는 결과를 얻는데 있다.
- 사용자 정의 Exception을 작성하기 위해서는 Throwable을 받지 않고 그 하위에 있으면서 보다 많은 기능들로 확장되어 있는 Exception으로부터 상속을 받는 것이 유용하다. 물론 입/출력에 관련된 예외를 작성하기 위해 IOException으로부터 상속을 받는 것이 보편적인 것이다.

[예제6-11] UserException.java

```
01 public class UserException extends Exception{
02
03     private int port = 772;
04     public UserException(String msg){
05         super(msg);
06     }
07     public UserException(String msg, int port){
08         super(msg);
09         this.port = port;
10     }
11     public int getPort(){
12         return port;
13     }
14 }
```

[예제6-12] UserExceptionTest.java

```
01 class UserExceptionTest {
02
03     public void test1(String[] n) throws UserException{
04         System.out.println("Test1");
05         if(n.length < 1)
06             throw new UserException("아무것도 없다네"); // 강제 예외 발생
07         else
08             throw new UserException("최종 예선",703); // 강제 예외 발생
09     }
10     public static void main(String[] args) {
11         UserExceptionTest ut = new UserExceptionTest();
12         try{
13             ut.test1(args);
14         }catch(UserException ue){
15             // System.out.println(ue.getMessage());
16             ue.printStackTrace();
17         }
18
19     }
20 }
```