10. 예외 처리

예외는 어떤 방법으로든지 해결되어야 합니다. 직접 해결하기 위해서는 try~catch 구문을 사용할 수도 있으며 throws 구문을 통해 예외가 발생했을 때 호출한 메서드로 예외를 넘겨줄 수도 있습니다. 이 장에서는 예외의 종류와 예외처리 방법, 사용자 정의 예외 클래스와 예외를 발생시키는 방법들에 대해 설명합니다. assert 키워드를 통해 특정 조건을 만족하는지 여부를 체크할 수 있는 기능에 대해서도 알아보겠습니다. 그리고 예외처리와 Assertion 기능을 통해서 요구사항 정의단계에서 선행 조건과 후행조건을 올바른 코드로 옮기기 위한 지침을 소개합니다.

10장의 주요 내용입니다.

- try~catch
- throws
- 사용자 정의 예외
- throw 문
- Assertion 코드 작성
- Assertion 코드 컴파일 및 실행

10.1. 예외 처리

예외 처리(Exception Handling)는 오류발생에 대한 대처 방법 중의 하나입니다. 예외 처리는 시스템 스스로 오류를 복구 하는 것이 아니고 오류발생 가능성이 있는 부분에 대한 처리를 미리 프로그램 해주는 것입니다. 오류의 종류는 심각한 오류(serious error)와 심각하지 않은 오류(mild error)로 나눌 수 있습니다.

심각한 오류는 메모리가 부족할 때 발생하는 OutOfMemoryError 등이 있습니다. 심각한 오류가 발생하면 오류는 복구될 수 없습니다. 심각한 오류가 발생할 경우 프로그램은 중지됩니다. 자바에서는 이런 경우에만 오류(Error)라고 합니다.

가벼운 오류는 예외(Exception)라고 부르며, 오류로 취급은 되지만 프로그램이 중지되지는 않게 할 수 있는 오류를 말합니다. 예를 들면 존재하지 않는 파일을 읽을 때 발생하는 FileNotFoundException, 배열의 범위를 넘어서는 ArrayIndexOutOfBoundsException 등이 있습니다. 존재하지 않는 파일을 읽으려고 할 때에 프로그램은 멈추지 않고 파일이 없음을 알리는 오류 메시지만 출력하고 계속 진행되게 할 수 있을 것입니다. 이와 같은 예외가 발생했을 경우 처리하는 것을 예외처리라고 합니다.

다음 프로그램은 실행 시 예외가 발생하도록 만든 예 입니다.

ExceptionProblemExample.java

```
1:  public class ExceptionProblemExample {
2:     public static void main (String[] args) {
3:        int i = 0;
4:        String[] greetings = {"안녕하세요.", "반갑습니다.", "또 오세요."};
5:
6:        while (i < 4) {
7:           System.out.println (greetings[i]);
8:           i++;
9:        }
10:
11:       System.out.println("메인의 마지막입니다.");
12:    }
13: }
```

이 프로그램은 오류가 발생하는데 컴파일 발생하는 오류가 아니고 실행 시 발생하는 예외입니다. while 문장은 i값이 3이 될 때까지 greeting[i] 데이터를 출력하는 코드 블록을 실행합니다. 그러나 배열 greetings는 greetings[2]까지 데이터가 저장되어 있기 때문에 i값이 3일 경우 배열의 범위를 넘어 참조하게 됩니다. 이렇게 배열의 범위를 넘어선 데이터를 참조하려 할 경우 자바에서는 예외가 발생하여 프로그램 실행이 중단 됩니다.

프로그램의 실행결과는 다음과 같습니다.

실행 결과를 보면, greetings[2]까지 출력하고 greetings[3]을 실행할 때에 ArrayIndexOutOfBoundsException이 발생했음을 알 수 있습니다. 이들 예외상황들은 자바 프로그램을 개발하는 도중에 종종 만나게 됩니다.

앞의 프로그램을 수정하여 오류가 발생하지 않게 하려면 5라인을 다음과 같이 기술할 수 있습니다.

```
5:          while (i < 3) {
```

그런데 7라인을 수정하지 않고 문제를 해결할 수 있는 방법이 있습니다. 즉 배열의 참조되는 인덱스를 수정하지 않고 프로그램이 정상 실행되도록 하는 방법에 대하여 알아보겠습니다.

예외처리가 발생하는 이유는 프로그램을 잘못 작성 했을 수도 있지만, 그렇지 않을 경우도 있습니다. 예외는 여러 상황에서 발생할 수 있으며, 예외처리는 예외사항을 찾아내 이를 처리한 다음 계속 프로그램이 진행되도록 하는 것입니다.

예외처리 방법은 크게 두 가지로 나눌 수 있습니다. 첫 번째는 try ~ catch 문을 사용하는 방법과, 두 번째는 throws 선언문을 사용하는 방법이 있습니다. try ~ catch 문을 이용하여 예외 상황을 처리하기 위해 받아들이는 것을 '예외를 잡는다(catch)'라고 표현하고, throws 선언문을 이용하여 예외 처리를 자신을 호출한 메서드에게 넘겨주는 것을 '예외를 던진다(throws)'라고 표현합니다. try ~ catch 문장은 예외를 직접 처리하는 경우에 사용하며, throws 선언문은 메서드 선언부에 표기하여 메서드를 호출하는 호출자에게 예외를 처리하도록 합니다. 이후 설명하는 내용들은 try ~ catch와 throws 에 대해서 더 자세하게 설명하고 있습니다.

예외 처리에서 기억해야할 것이 있습니다. 예외처리를 사용하면 예외가 발생했을 때 JVM에서 예외처리를 담당하는 부분을 자동적으로 호출해 주지만, 예외를 처리하기 위한 부분은 프로그래머가 직접 작성해야 합니다.

10.1.1. try ~ catch

예외를 처리하는 방법 중에서 try~catch 문을 이용한 예외처리 방법을 살펴보겠습니다.

다음 코드는 앞에서 배열의 범위를 넘어선 데이터를 출력할 때 오류가 발생했던 코드를 try~catch문을 이용하여 예외처리를 하는 예 입니다. while 문의 조건문은 변경하지 않은 상태에서 배열의 범위를 넘어 참조했을 경우 예외가 발생하고 이를 처리하는 예입니다.

TryCatchExample.java

```
1:  public class TryCatchExample {
2:     public static void main (String[] args) {
3:        int i = 0;
4:        String[] greetings = {"안녕하세요.", "반갑습니다.", "또 오세요."};
5:        while (i < 4) {
6:           try {
7:              System.out.println(greetings[i]);
8:           }
9:           catch (ArrayIndexOutOfBoundsException e) {
10:             System.out.println("예외 발생했습니다.");
11:             System.out.println("예외가 발생한 원인은 " + e.getMessage());
12:             System.out.println("예외 처리를 완료했습니다.");
13:          }
14:          finally {
15:             System.out.println("finally 문은 항상 실행됩니다.");
16:          }
17:          i++;
18:       }
19:       System.out.println("메인의 마지막입니다.");
20:    }
21: }
```

```
Console
<terminated> TryCatchExample [Java Ap
안녕하세요.
finally 문은 항상 실행됩니다.
반갑습니다.
finally 문은 항상 실행됩니다.
또 오세요.
finally 문은 항상 실행됩니다.
예외 발생했습니다.
예외가 발생한 원인은 3
예외 처리를 완료했습니다.
finally 문은 항상 실행됩니다.
메인의 마지막입니다.
```

try~catch 문장은 크게 try, catch, finally 블록으로 구성됩니다.

첫 번째, try 블록으로 예외가 발생할 가능성이 있는 문장을 포함합니다.

```
6:           try {
7:              System.out.println(greetings[i]);
8:           }
```

try 블록에서 greeting[3]의 내용을 출력하는 도중에 예외상황이 발생할 것입니다. 이처럼 예외가 발생할 것이라고 예상되는 문장들은 try 블록 안에 기술합니다. try 블록은 단독으로 사용할 수 없습니다. 반드시 catch 블록 또는 finally블록과 함께 사용해야 합니다.

두 번째, catch 블록은 try 블록에서 발생한 예외상황을 감지하여 예외가 발생했을 때 적절한 조치를 취하기 위해 사용하는 곳입니다.

```
9:          catch (ArrayIndexOutOfBoundsException e) {
10:             System.out.println("예외 발생했습니다.");
11:             System.out.println("예외가 발생한 원인은 " + e.getMessage());
12:             System.out.println("예외 처리를 완료했습니다.");
13:         }
```

catch 블록의 실행은 try블록에서 예외가 발생했을 때입니다. try블록에서 예외가 발생하지 않으면 catch 블록은 실행되지 않습니다. catch 블록에서 기술한 예외가 아닌 다른 예외가 발생하면 catch 블록의 예외는 처리되지 않습니다. 위의 프로그램에서는 ArrayIndexOutOfBoundsException이 발생했을 경우에 catch 블록이 실행됩니다. 이렇게 try블록 안에서 예외상황이 발생하면 catch 블록이 수행되며, 이때 발생한 예외는 ArrayIndexOutOfBoundsException 유형의 인스턴스(instance) 입니다. 따라서 catch 블록에서 예외 객체를 이용할 수 있습니다. catch 블록에서는 예외가 발생한 원인을 출력하고 있습니다.

세 번째, finally 블록은 catch 블록과는 다르게 예외상황의 발생여부에 관계없이 무조건 실행됩니다.

```
14:         finally {
15:             System.out.println("finally 문은 항상 실행됩니다.");
16:         }
```

try 문이 실행되면 finally 블록은 항상 실행됩니다. 그래서 “finally 문은 항상 실행됩니다.” 라는 문장은 try 문이 실행될 때마다 계속 출력됩니다. 프로그램 실행 도중에 예외가 발생해도 프로그램이 계속 실행될 수 있는 이유는 프로그래머가 예외상황을 직접 다룰 수 있기 때문입니다.
finally 블록은 try 블록에서 할당받은 자원을 반납해야 하는 코드[35] 등을 포함할 수 있습니다.

finally는 try 블록 또는 catch 블록에서 return문을 만나더라도 실행됩니다. 다음의 경우에만 finally 블록이 실행되지 않습니다.

- System.exit() 구문을 호출했을 때.
- 전원이 꺼져 시스템이 멈추었을 때.
- finally 블록 내부에서 예외상황이 발생했을 때.
- try 블록을 실행시키는 스레드가 죽었을 때.

try 블록에서 한 가지 예외만 발생하는 것은 아닙니다. catch 블록에서 처리하려고 하는 예외가 아닌 다른 예외가 try 블록에서 발생하면 그 예외는 처리되지 않습니다. 즉, 위의

35) 파일 입/출력 스트림이나 데이터베이스 커넥션 등을 닫아주는 코드가 있습니다.

예에서 try블록에서 NullPointerException이 발생하면 그 예외는 처리 되지 않습니다. 예외가 여러 가지가 발생할 경우에는 catch블록을 중복해서 사용할 수도 있습니다. 그럴 경우에 일반적인 예외(Exception 계층 구조에서 상위 예외 클래스)가 먼저 처리될 수 있으므로 Exception 계층 구조에서 상위 클래스가 하위 클래스보다 먼저 catch블록의 인자로 선언되어서는 안 됩니다.

다음 코드는 try블록에서 여러 개의 예외가 발생되는 예입니다.

CatchExample.java

```
1:  public class CatchExample {
2:     public static void main(String[] args) {
3:        int a = (int)(Math.random() * 100);
4:        int b = (int)(Math.random() * 4);
5:        try {
6:           System.out.printf("a=%d, b=%d, result=%d\n", a, b, divide(a, b));
7:           doSomething(a);
8:        }catch(ArithmeticException ae) {
9:           System.out.println("0으로 나누려고 하고 있습니다.");
10:          System.out.println(ae.getMessage());
11:       }catch(Exception e) {
12:          System.out.println("예외가 발생했습니다.");
13:          System.out.println("예외 발생원인 : "+e.getMessage());
14:       }
15:    }
16:
17:    public static int divide(int a, int b) {
18:       return a/b;
19:    }
20:
21:    public static void doSomething(int a) throws Exception{
22:       if(a>50) {
23:          throw new Exception("a값이 50보다 큽니다.");
24:       }
25:       System.out.println("50%확률로 실행됩니다.");
26:    }
27: }
```

Console
<terminated> CatchExample [Java
a=37, b=3, result=12
50%확률로 실행됩니다.

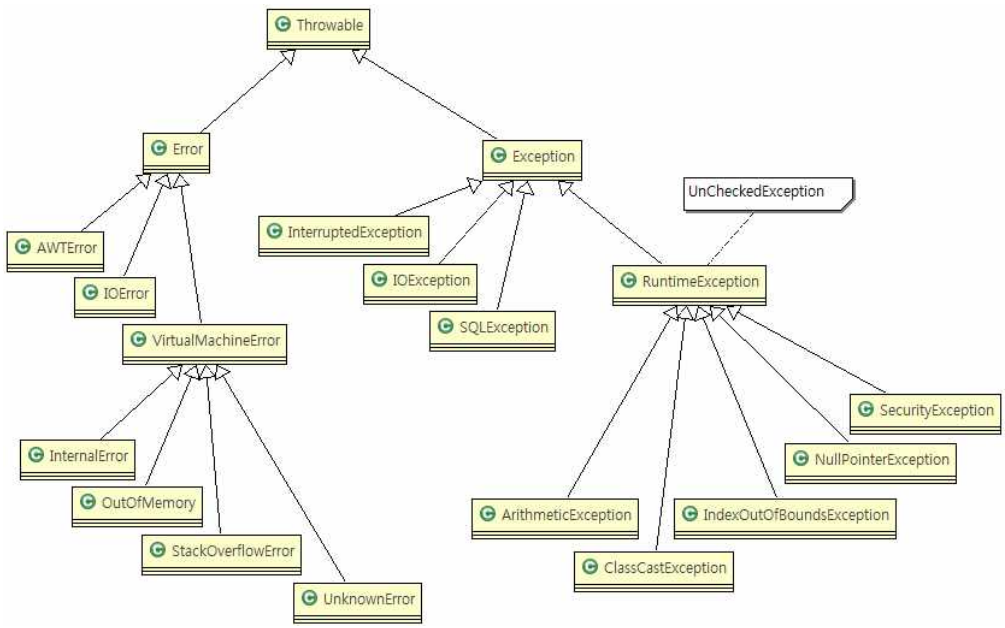
Console
<terminated> CatchExample [Java Ap
0으로 나누려고 하고 있습니다.
/ by zero

Console
<terminated> CatchExample [Java Application]
a=66, b=3, result=22
예외가 발생했습니다.
예외 발생원인 : a값이 50보다 큽니다.

이 코드에서 ArithmethicException catch 블록과 Exception catch 블록 위치를 바꾸면 "Unreachable catch block for ArithmeticException" 컴파일 오류가 발생합니다. 이 코드는 실행할 때마다 결과가 다를 수 있습니다. doSomething() 메서드는 인자 값이 50보다 크면 따라서 예외를 발생합니다. throw 구문은 예외를 발생시키기 위한 키워드입니다. doSomething() 메서드에서 예외를 발생시키면 Exception catch 블록이 실행되며, 발생한 난수 b 값이 0일 경우 ArithmethicException catch 블록이 실행됩니다.

10.1.2. 예외의 종류

다음 그림은 예외클래스들의 상속관계를 클래스다이어그램으로 나타낸 것입니다. 빈 삼각형을 갖는 실선은 클래스 일반화 관계, 즉 상속관계를 나타냅니다. 그림은 예외와 관련된 클래스들을 개략적으로 나타낸 것입니다. 여기에 나타나 있지 않은 클래스들도 많이 있으므로 예외 클래스들에 대해서 더 많이 알고 싶으면 API문서를 참조해야 합니다.



오류 또는 예외와 관련된 클래스는 Throwable의 하위 클래스들 이며, 이 중 Error클래스와 그 하위 클래스들에서 정의되어 있는 오류 상황이 발생하면 프로그램이 종료되는 것이 당연합니다. 그러나 Exception과 그 하위 클래스들에 정의되어 있는 것과 관련 있는 예외 상황이 발생하면 이들 예외는 처리의 대상 될 수 있습니다.

예외 클래스 계층 구조에서 Exception의 하위 클래스 들 중에서 RuntimeException은 설계상의 문제 또는 구현 코드 문제들과 관련된 예외 클래스들 입니다. 즉, 프로그램이 정상적으로 실행되고 있는 상황에서 발생해서는 안 되는 실행 도중의 상태를 알려주는 예외 클래스들 입니다. 이런 실행중의 예외들의 예를 들면, 나누려고 하는 숫자가 0인 경우, 배열의 인덱스를 벗어난 참조, null인 레퍼런스 변수 참조, 형 변환 예외, 보안과 관련된 예외 상황 등이 있습니다. 올바르게 설계되어 구현된 프로그램에서는 절대로 이런 종류의 예외가 발생하지 않으므로 처리하지 않고 그대로 두는 것이 보통입니다. 이들 예외가 발생하면 실행 중 메시지가 나오고 프로그램을 종료하므로 예외를 수정할 수 있습니다. 그리고 RuntimeException과 그 하위 예외들은 예외처리를 하지 않아도 컴파일타임에는 예외가

발생할 가능성이 있는지 체크 하지 않습니다. 그러나 실행 중에 예외가 발생하면 프로그램의 실행은 중단될 수 있습니다. 이들 RuntimeException과 그 하위 클래스들을 컴파일 시 예외 발생 유/무를 판단하지 않으므로 비검증 예외(Unchecked Exception)라고 부릅니다.

비검증 예외(Unchecked Exception)가 아닌 다른 예외들이 발생할 가능성이 있는 메서드 또는 생성자를 포함하는 구문들은 모두 컴파일타임에 예외처리 여부를 검사하기 때문에 반드시 예외처리를 해 주어야 합니다. 예를 들면 파일을 찾을 수 없거나 URL을 잘못 지정한 경우가 그러한 경우 입니다. 이런 것은 사용자가 잘못 입력할 때 발생하며 프로그래머가 처리해야 합니다. 예외들 중에서 InterruptedException, IOException, SQLException 등과 그 하위 예외들이 발생할 가능성이 있는 문장을 작성할 경우 반드시 예외 처리 코드를 포함 해 주어야 합니다.

개발자는 API에서 제공되는 메서드를 사용할 때는 사용하고자 하는 메서드가 어떤 예외 클래스를 throws 하고 있는지 확인하고 그에 맞는 예외 처리를 해 주어야 할 것입니다.

다음 코드는 검증(Checked)예외의 예입니다. 검증예외를 발생시키는 메서드 또는 생성자를 사용할 경우에 반드시 예외처리를 해 줘야 컴파일 오류가 발생하지 않습니다.

CheckedExample.java

```
1:  import java.io.IOException;
2:
3:  public class CheckedExample {
4:
5:      public static void main(String[] args) {
6:          System.out.println("값을 입력받는 프로그램입니다.");
7:          byte[] data = new byte[100]; //한번에 100byte씩 읽습니다.
8:          try {
9:              System.in.read(data);
10:         } catch (IOException e) {
11:             e.printStackTrace();
12:         }
13:         System.out.print("입력한 문자열은 : ");
14:         System.out.println(new String(data).trim());
15:     }
16: }
```

```
Console
<terminated> CheckedExample [Java Application] C:₩
값을 입력받는 프로그램입니다.
안녕하세요. 반갑습니다.
입력한 문자열은 : 안녕하세요. 반갑습니다.
```

위의 프로그램은 사용자로부터 입력받은 문자열을 화면에 다시 출력하는 예입니다. 사용자로부터 문자열을 입력받기 위해 read() 메서드를 사용했습니다. read() 메서드는 java.io.InputStream 클래스에 정의되어 있는 메서드입니다. API 문서에서 read() 메서드를 참고하면 Throws 항목에 IOException과 NullPointerException 이 있는 것을 확인할 수 있습니다. Throws 항목에 있는 예외 클래스들 중에서 비검증(UnChecked)예외가 아닌 예외가 있을 경우에는 반드시 예외처리를 해줘야 컴파일 오류가 발생하지 않습니다. 비검증 예외가 아닌 다른 예외들은 모두 검증(Checked) 예외입니다.

read

```
public int read(byte[] b)
        throws IOException
```

Reads some number of bytes from the input stream and stores them into the buffer array b. The number of bytes actually read is returned as an integer. This method blocks until input data is available, end of file is detected, or an exception is thrown.

If the length of b is zero, then no bytes are read and 0 is returned; otherwise, there is an attempt to read at least one byte. If no byte is available because the stream is at the end of the file, the value -1 is returned; otherwise, at least one byte is read and stored into b.

The first byte read is stored into element b[0], the next one into b[1], and so on. The number of bytes read is, at most, equal to the length of b. Let *k* be the number of bytes actually read; these bytes will be stored in elements b[0] through b[*k*-1], leaving elements b[*k*] through b[b.length-1] unaffected.

The read(b) method for class InputStream has the same effect as:

```
read(b, 0, b.length)
```

Parameters:

b - the buffer into which the data is read.

Returns:

the total number of bytes read into the buffer, or -1 if there is no more data because the end of the stream has been reached.

Throws:

IOException - If the first byte cannot be read for any reason other than the end of the file, if the input stream has been closed, or if some other I/O error occurs.

NullPointerException - if b is null.

See Also:

read(byte[], int, int)

다음은 자바 언어에서 미리 정의된 몇 가지 예외 중에서 발생빈도가 높은 비검증 예외 클래스를 설명한 것입니다.

- ArithmethicException : 일반적으로 정수를 0(zero)로 나눗셈을 할 때 발생합니다.
- NullPointerException : 인스턴스를 만들기 전에 객체나 메서드를 액세스하면 발생합니다.

```
Image[] img= new Image[4];
System.out.println(img[0].toString());
```

- ArrayIndexOutOfBoundException : 배열의 인덱스를 벗어난 구성요소를 액세스할 때 발생합니다.

10.1.3. throws

예외 상황을 다루는 방법으로 try~catch 구문 외에 throws 구문이 있습니다. try~catch 문이 예외가 발생했을 때 직접 해결을 하고자 하는 코드라면 throws는 메서드를 호출한 곳으로 예외가 발생했음을 알리는 코드입니다. 즉 예외 처리를 직접 수행하지 않고 호출자에게 예외를 던지는(throws) 방법입니다. 예외가 발생하는 원인이 실행되는 메서드에 있지 않고 호출하는 메서드에게 있을 경우 사용하는 방법입니다. 그래야만 예외가 발생되는 근본 원인을 해결할 수 있을 것입니다.

간혹 main() 메서드에서 throws 해 놓은 코드를 보실 수도 있습니다. main() 메서드에서 throws 하는 것은 예외 처리를 JVM에게 넘기겠다는 의미입니다. 그러나 JVM이 그 예외를 받아서 처리해 주지는 않습니다. 예외가 발생하면 JVM은 예외 메지를 출력하고 프로그램을 종료 시킬 것입니다. main()에서 throws 하는 코드가 있었다면 테스트를 위해 사용

한 코드에서 검증(Checked)예외를 발생시키는 구문을 포함하기 때문이라고 생각하시기 바랍니다.

throws 구문에서 예외상황이 여러 개 있을 때는 ","(comma)로 구분하여 나열하여 표기합니다.

```
modifier return-type methodName() throws Exception1, Exception2 ...{
    method-body;
}
```

먼저 다음의 코드를 보겠습니다.

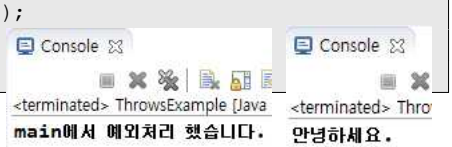
```
public void exceptionOccur() throws ArrayIndexOutOfBoundsException {
    ... // 이 안에서 ArrayIndexOutOfBoundsException이 발생했다고 가정.
}
```

exceptionOccur() 메서드 안에서 ArrayIndexOutOfBoundsException이 발생하면, try~catch문을 처리할 수 있지만 try~catch문을 사용하지 않고 메서드의 선언부에 throws를 사용해도 처리가 가능합니다. 이 경우는 exceptionOccur()를 호출한 메서드에게 발생한 예외를 던지는 것입니다.

다음 코드는 throws 구문을 이용해 예외상황을 처리하는 예입니다.

ThrowsExample.java

```
1:  public class ThrowsExample {
2:
3:      static String[] greetings = {"안녕하세요.", "반갑습니다.", "또 오세요."};
4:
5:      public static void main(String[] args) {
6:          int i = (int)(Math.random()*4); //0, 1, 2, 3중 하나가 랜덤하게 생성
7:
8:          try {
9:              doIt(i);
10:         }catch(Exception e) {
11:             System.out.println("main에서 예외처리 했습니다.");
12:         }
13:     }
14:
15:     public static void doIt(int index) throws ArrayIndexOutOfBoundsException {
16:         System.out.println(greetings[index]);
17:     }
18: }
```



소스코드를 보면 main() 메서드에서 doIt() 메서드를 호출했습니다. doIt() 메서드 호출시 인자의 값은 랜덤하게 발생되어 적용됩니다. 그런데 만일 랜덤하게 발생된 숫자가 3일 경

우에는 doIt() 메서드에서 예외가 발생합니다. 그런데, doIt() 메서드의 선언부에서 예외를 throws를 사용하고 있으므로 ArrayIndexOutOfBoundsException이 발생할 경우, 자신을 호출한 메서드에게 이 예외를 던져라(throws)라는 의미로 해석할 수 있습니다. 따라서 이 경우, doIt() 메서드를 호출한 main() 메서드로 예외가 던져지게 되는데, main() 메서드의 입장에서는 결국, doIt() 메서드를 호출하는 곳에서 예외가 발생한 것이고, 그래서 이를 try ~ catch를 이용해서 해결하고 있습니다.

처음부터 doIt() 메서드 안에서 try ~ catch 블록을 이용해 예외를 처리할 수 있습니다. 그러나 doIt() 메서드에서 예외가 발생하는 이유가 인수로 넘어오는 index 값이기 때문에 예외가 발생된 근본 원인을 해결하기 위해 메서드를 호출하는 곳에 예외를 보내주는 것입니다.

참고로 재정의(Overriding)와 관련해서 한 가지 더 알고 있어야 할 것이 있습니다. throws로 선언된 메서드를 재정의 할 때에 재정의 하는 자식클래스의 메서드에서는 부모클래스의 메서드에서 throws한 예외가 아닌 새로운 예외는 throws 할 수 없습니다. 즉, 재정의 되는 자식 메서드에서는 부모의 메서드에서 throws 하는 예외에 비하여 더 적은 개수의 예외를 throws 하거나, 부모의 메서드에서 throws 하는 예외클래스의 하위클래스 예외가 throws 될 경우만 가능합니다. 자세한 내용은 “메서드 재정의와 throws”절에서 배웁니다.

10.1.4. 사용자 정의 예외

앞에서 살펴본 바와 같이 예외는 시스템에서 자동으로 발생시켜주기 때문에 발생된 예외를 try~catch 혹은 throws를 이용해 처리하면 됩니다. 뿐만 아니라 프로그래머가 직접 예외클래스를 만들 수 있고, 이를 원하는 시점에 발생시킬 수도 있습니다.

사용자 정의 예외(user defined exception) 클래스를 만든다는 것은 보통의 클래스를 만드는 것과 동일합니다. 그러나 유의할 점은 사용자 정의 예외 클래스는 반드시 Exception 클래스 또는 그 하위 클래스를 상속[36]받아서 만들어야 한다는 점입니다.

다음 프로그램은 예외 클래스를 직접 만들어 사용하는 예를 보인 것입니다.

ServerTimedOutException.java

```
1:  public class ServerTimedOutException extends Exception {
2:      private int port;
3:
4:      public ServerTimedOutException(String message, int port) {
```

36) 반드시 Exception 클래스를 상속받을 필요는 없습니다. Exception 클래스뿐만 아니라 그 하위 클래스이면 어떤 클래스라도 가능합니다.

```
5:          super(message);
6:          this.port = port;
7:      }
8:
9:      public int getPort() {
10:         return port;
11:     }
12: }
```

클래스 선언부에서 Exception 클래스를 상속받은 후 ServerTimedOutException 클래스를 정의하고, 나머지는 일반적인 클래스선언과 동일합니다. 그리고 예외 클래스들도 보통의 클래스들처럼 멤버 변수를 포함할 수 있습니다. 예외 클래스도 객체를 생성할 때 멤버 변수의 값을 초기화하기 위해 생성자를 포함할 수 있습니다. super(message); 구문은 사용자 정의 예외를 발생시킬 때 예외의 원인을 예외 객체가 저장할 수 있도록 합니다. 예외 클래스도 보통 클래스들처럼 메서드를 포함할 수 있습니다.

10.1.5. throw

사용자가 직접 선언한 예외 클래스 또는 API에서 제공하는 예외클래스들을 이용해서 예외를 발생시키려면 new 키워드를 이용하여 객체를 생성하는 것만으로는 안 됩니다. 다음과 같이 발생시키고 싶은 예외 클래스의 인스턴스를 만든 후, "throw"(throws가 아니라 throw입니다.) 키워드를 이용하여 예외가 발생되도록 해야 합니다. throw는 사용자가 만든 예외를 발생시킬 때만 사용할 수 있는 것은 아닙니다. 예외클래스로 정의된 예외라면 어떤 예외든지 throw를 이용해 예외를 강제로 발생시킬 수 있습니다.

```
throw new ServerTimedOutException("Could not connect",
80);
```

다음 프로그램은 사용자가 예외를 정의하고 발생시키는 예입니다. 이 전 사용자 정의 예외에서 설명했던 클래스(ServerTimedOutException)가 작성되어 있어야 합니다.

TestUserException.java

```
1:  public class UserExceptionExample {
2:      String defaultServer = "javaspecialist.co.kr";
3:      String alternativeServer = "consolbook.com";
4:
5:      public void connectMe(String serverName) throws ServerTimedOutException {
6:          int success;
7:          int portToConnect = 80;
8:          success = open(serverName, portToConnect);
9:          if (success == 0) {
10:             throw new ServerTimedOutException("Could not connect", 80);
```

```
11:         }
12:     }
13:     public void findServer() {
14:         try {
15:             connectMe(defaultServer);
16:             System.out.println(defaultServer + "에 연결되었습니다.");
17:         }
18:         catch (ServerTimedOutException e) {
19:             System.out.println( "Server timed out, trying alternative...");
20:             try {
21:                 connectMe(alternativeServer);
22:                 System.out.println(alternativeServer + "에 연결되었습니다.");
23:             }
24:             catch (ServerTimedOutException e1) {
25:                 System.out.println("서버 연결에 실패했습니다.");
26:                 System.out.println( "Error : " + e1.getMessage() +
                        "connecting to port " + e1.getPort());
27:             }
28:         }
29:     }
30:     public int open (String serverName, int port) {
31:         return (int)(Math.random() * 2);//0 또는 1이 리턴됩니다.
32:     }
33:     public static void main (String[] args) {
34:         UserExceptionExample ue = new UserExceptionExample();
35:         ue.findServer();
36:     }
37: }
```

Console
<terminated> UserExceptionExample [Java Application]
javaspecialist.co.kr에 연결되었습니다.

Console
<terminated> UserExceptionExample [Java Application] C:₩Prograı
Server timed out, trying alternative...
consolbook.com에 연결되었습니다.

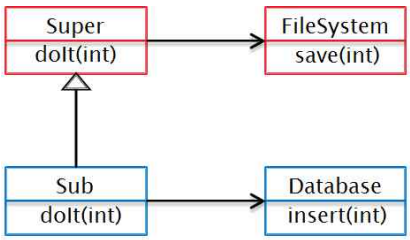
Console
<terminated> UserExceptionExample [Java Application] C:₩Program Files₩Java
Server timed out, trying alternative...
서버 연결에 실패했습니다.
Error : Could not connectconnecting to port 80

이 예제는 실제 서버에 접속하는 프로그램은 아닙니다. 사용자 정의 예외와 throw 구문을 이해시키기 위한 용도로 만들어진 예외입니다. 예제 코드에서 프로그램 시작점은 main() 메서드입니다. 이 예제는 서버에 접속을 시도하다가 접속에 문제가 발생하면 이미 정의한 ServerTimedOutException을 발생시킵니다. 주의할 부분은 connectMe() 메서드에서 open() 메서드를 통해 서버에 접속을 시도하는 코드입니다. 이미 정의되어 있는 open() 메서드는 0또는 1값을 반환합니다. 0을 리턴하면 접속에 실패한 것으로 간주하여 예외를 만들고 이를 throw합니다. throw 구문은 예외를 강제로 발생시킬 때 사용하는 구문입니다. if 블록에서 ServerTimedOutException이 발생할 수 있으므로 connectMe() 메서드 선언부에 throws 구문을 포함시켰습니다. 예외의 발생 원인이 서버 주소에 기인한 것 일 수 있기 때문에 예외를 try ~ catch 블록으로 처리하지 않고 throws 선언문으로 처리 한 것입니다. 예외가 발생되면 try ~ catch문을 이용 예외상황을 처리하게 됩니다. catch 블록에는 다른 서버로 접속시도를 합니다.

10.1.6. 메서드 재정의와 throws

메서드를 재정의 할 때 부모에서 던지지 않은 새로운 예외는 던질 수 없다고 했습니다. 메서드 재정의와 throws 대해 좀 더 알아보겠습니다.

아래 그림은 4개 클래스를 보여주고 있습니다. 닫힌 화살표를 갖는 실선은 일반화(Generalization)를 의미합니다. 즉, Super와 Sub 클래스는 상속 관계가 있다는 의미입니다. 열린 화살표를 갖는 실선은 연관관계(Association)를 의미하며 화살표를 받는 클래스의 인스턴스가 화살표를 주는 클래스의 멤버로 정의되어 있음을 의미합니다. 즉, Super 클래스의 멤버변수로 FileSystem 객체가 선언되어 있다는 것입니다. 전체적으로 Super 클래스는 FileSystem 클래스의 기능을 사용하고, Sub 클래스는 Database 클래스의 기능을 사용하고 있음을 보여주고 있습니다.



시스템이 초기 개발되었을 때에는 Super 클래스와 FileSystem 클래스만 존재하였고, doIt() 메서드는 FileSystem 클래스를 이용해 데이터를 저장하고 있었다고 가정합니다. 이후 시스템 업그레이드가 필요하여 기존을 확장할 필요가 있어 Database 클래스가 추가된 것입니다. Database의 기능을 사용하는 클래스인 Sub는 Super를 상속받아 doIt() 메서드를 재정의 해야 하는 상황이 발생한 것입니다. 그런데 FileSystem 클래스의 save() 메서드는 IOException을 throws 하도록 정의되어 있었는데, Database 클래스의 insert() 메서드는 SQLException을 throws 하도록 정의되어 있다는 것입니다. 이런 상황에서 Sub의 doIt() 메서드를 어떻게 재정의 해야 하는 것이 주어진 과제입니다.

다음 그림은 Super 클래스를 상속받은 Sub 클래스에서 doIt() 메서드가 SQLException을 throws 할 때 발생하는 오류 메시지입니다.

이런 상황을 해결하는 방법은 재정의 하는 메서드에서 try 블록을 이용해 예외가 발생한 코드를 작성합니다. 그리고 catch 블록에서 부모의 예외를 throw 하도록 하면 재정의 하는 메서드에서는 부모의 메서드에서 throws 한 예외를 throws 할 수 있습니다.

```
        try {
            db.insert(num);
        } catch (SQLException e) {
            throw new IOException(e.getMessage());
        }
```

다음 설명하는 코드들은 메서드 재정의시 예외처리 방법을 보여주기 위한 클래스들입니다.

FileSystem 클래스는 파일에 데이터를 저장하는 것을 시뮬레이션하기 위한 코드입니다. save() 메서드의 인자로 전달된 num 값이 0보다 작으면 예외를 발생시키도록 하고 있습니다.

FileSystem.java

```
1:  import java.io.IOException;
2:
3:  public class FileSystem {
4:      public void save(int num) throws IOException  {
5:          if(num<0) {
6:              throw new IOException("num이 0보다 작습니다.");
7:          }
8:          System.out.println("File에 저장됐습니다.");
9:      }
10: }
```

Super 클래스는 FileSystem 클래스의 save() 메서드를 호출하고 있습니다. save() 메서드에서 IOException이 발생할 가능성이 있으므로 throws를 이용하여 예외처리 하였습니다. try~catch문을 이용하지 않는 이유는 save() 메서드 인자로 전달될 값을 doIt() 메서드를 호출하는 곳으로부터 전달받기 때문입니다.

Super.java

```
1:  import java.io.IOException;
2:
3:  public class Super {
4:      FileSystem fs = new FileSystem();
5:
6:      public void doIt(int num) throws IOException {
7:          System.out.println("Super.doIt");
8:          fs.save(num);
9:      }
10: }
```

Database 클래스는 데이터베이스에 데이터를 저장하는 것을 시뮬레이션하기 위한 코드입니다. insert() 메서드에서 num 값이 100보다 크면 SQLException을 발생시킵니다.

Database.java

```
1:  import java.sql.SQLException;
2:
3:  public class Database {
4:     public void insert(int num) throws SQLException {
5:        if(num>100) {
6:           throw new SQLException("num이 너무 큽니다.");
7:        }
8:        System.out.println("데이터베이스에 저장되었습니다.");
9:     }
10: }
```

Sub 클래스는 Super 클래스를 상속받았습니다. 그리고 doIt() 메서드를 재정의 하고 있습니다. insert() 메서드가 SQLException을 발생시키기 때문에 예외처리를 해야 합니다. 그렇다고 해서 아래 코드처럼 작성하면 오류가 발생합니다. 그 이유는 앞에서 언급했던 것처럼 부모에서 던지지 않는 새로운 예외는 재정의 하는 메서드에서 던질 수 없기 때문입니다.

```
    public void doIt(int num) throws SQLException {//Error
       System.out.println("Sub.doIt");
       db.insert(num);
    }
```

다음 코드는 바르게 작성된 Sub 클래스입니다.

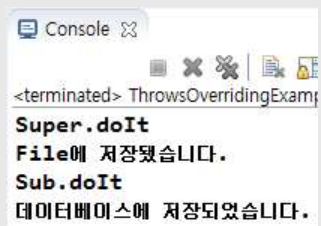
Sub.java

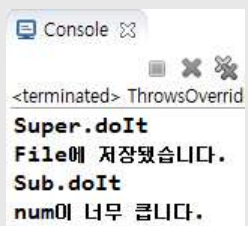
```
1:  import java.io.IOException;
2:  import java.sql.SQLException;
3:
4:  public class Sub extends Super {
5:     Database db = new Database();
6:
7:     public void doIt(int num) throws IOException {
8:        System.out.println("Sub.doIt");
9:        try {
10:          db.insert(num);
11:       } catch (SQLException e) {
12:          throw new IOException(e.getMessage());
13:       }
14:    }
15: }
```

다음 코드는 위의 코드들을 테스트하기 위한 코드입니다. Super로 선언된 객체변수를 이용해 호출하는 doIt() 메서드는 생성하고 참조되는 객체의 유형에 따라 다르게 동작할 것입니다. Super 클래스의 인스턴스를 참조하면 FileSystem 클래스를 통해 데이터를 저장하며, Sub 클래스의 인스턴스를 참조하면 Database 클래스를 통해 데이터를 저장하는 것을 시뮬레이션 할 수 있습니다.

ThrowsOverridingExample.java

```
1:  import java.io.IOException;
2:  import java.util.Random;
3:
4:  public class ThrowsOverridingExample {
5:
6:      public static void main(String[] args) {
7:          Super[] works = new Super[2];
8:          works[0] = new Super();
9:          works[1] = new Sub();
10:
11:         Random rand = new Random();
12:         int data = rand.nextInt(200);
13:         for(Super s : works) {
14:             try {
15:                 s.doIt(data);
16:             } catch (IOException e) {
17:                 System.out.println(e.getMessage());
18:             }
19:         }
20:     }//end main
21: }//end class
```





10.1.7. 자원관리 자동화와 멀티캐치

자원관리 자동화(automatic resource management)와 멀티캐치(Multi-catch)는 Java SE 7(JDK 1.7) 버전에 추가된 기능입니다.

자원관리 자동화는 try문 뒤에 괄호를 치고, 변수를 선언할 수 있는데 이렇게 하면 try가 끝날 때 해당 변수에 대한 close가 자동으로 호출됩니다. 사실 좀 더 정확하게는 자원관리 자동화 기능을 사용하면 컴파일러는 자동으로 finally 블록에 자원 반납 코드를 넣어줍니다. 역 컴파일(decompile) 해보면 알 수 있습니다. 자원관리 자동화 기능은 finally 블록에서 자원해제를 위한 코드를 넣지 않아도 된다는 장점은 있지만 catch문에서 해당 변수에 접근이 안 되는 단점이 있습니다. 또한 try 구문에서 선언된 클래스는 AutoCloseable 인터페이스를 구현한 클래스 이어야 합니다.

```
try(FileInputStream in = new FileInputStream("a.txt"))
```

멀티캐치 기능은 하나의 catch 블록에 다수의 예외를 기술하는 것입니다. 하나의 catch 문으로 다수의 예외를 처리해줄 수 있습니다. 멀티캐치의 예외들은 계층구조가 없어야 합니다.

```
catch(NullPointerException | IOException ex)
```

다음은 자원관리 자동화 예외처리 구문 예입니다. 예에서 9라인은 자원관리 자동화의 예를 보여주고 있으며, 11라인은 멀티캐치 예를 보여주고 있습니다.

AutoResourceManagerExample.java

```
1:  import java.io.*;
2:
3:  public class AutoResourceManagerExample {
4:
5:      public static void main(String[] args) {
6:
7:          try(FileInputStream in = new FileInputStream("a.txt")) {
8:              System.out.println("read data : " + in.read());
9:          }catch(NullPointerException | IOException ex) {
10: //          System.out.println(in);  //in 변수 참조 못함
11:             System.out.println("예외 처리합니다");
12:             System.out.println(ex.toString());
13:         }
14:
15:     }
16: }
```

이 예에서 try 블록에 괄호를 이용해서 객체를 생성하는 코드를 넣을 것을 볼 수 있습니다. 이렇게 하면 컴파일러는 자동으로 finally 블록에서 해당 리소스 자원을 반납하는 코드를 넣어줍니다. 이것을 보고 예외처리 시 자원관리 자동화라 부릅니다.

하나의 catch문에서 다수의 예외를 처리해 줄 수 있습니다. 이렇게 하면 여러 예외를 처리하기 위해 catch 블록을 반복해서 작성하지 않아도 됩니다. 이러한 문장을 멀티캐치 문장이라 부릅니다. 멀티 캐치 문장 작성 시 주의할 사항이 있습니다. 하나의 catch 문에 다수의 예외를 처리할 경우 기술되는 예외들은 계층 구조가 없어야 합니다. 예를 들면 IOException과 FileNotFoundException을 동시에 잡을 수는 없습니다.
자원관리 자동화와 멀티캐치 구문은 반드시 사용할 필요는 없습니다. 필요한 경우 사용할 수 있습니다.

Java SE 7(JDK 1.7)에 추가된 예외에 대해 더 자세하게 알고 싶으면 아래 주소를 참고하세요.
http://www.oracle.com/technetwork/articles/java/java7exceptions-486908.html

10.2. Assertion

이 기능은 JDK 1.4 버전에 추가되었습니다. Assertion을 영어 사전에서 찾아보면 '단정', '단언', '주장' 이라고 되어 있습니다. 자바 프로그램에서 assertion 은 특정 내용을 단정 짓는 것입니다. Assertion을 굳이 우리말로 변역할 필요는 없습니다. 그냥 '어써션'이라고 읽으면 됩니다. assertion은 오류가 없는 프로그램을 작성하기 위한 하나의 기술입니다.

프로그램 내에서, 특정 조건이 성립해야 하는 장소에 검증용 코드를 넣어 그 조건에 위반하고 있는 경우는 오류를 출력해 프로그램 상태를 체크할 수 있도록 합니다. 이러한 방법은 프로그램의 버그의 수정을 도와, 보다 견고한 프로그램을 작성하는 것이 가능해집니다. assert 구문은 프로그래머가 자신의 프로그램에 대한 가정을 확신시키는 문장으로 boolean 수식을 가지고 주어진 조건을 만족하지 않을 경우 즉, 수식의 결과가 false 일 경우 예외를 발생시키는 키워드입니다.

assertion 기능이 없었을 때 개발자들은 검증용 코드를 디버그메시지(println) 또는 예외를 사용했었습니다. 아래의 코드들을 예를 들어 설명하겠습니다. 입력 값이 0일 때 사용자 인증 처리를 하는 코드입니다.

다음 코드는 검증 구문을 포함하지 않았습니다.

```
int x = new Scanner(System.in).nextInt();
if(x==0) {
    System.out.println("인증되었습니다.");
}
```

위 코드는 x값으로 0이 들어갈 경우에는 if 블록이 정상적으로 실행되지만 그렇지 않을 경우에는 if 블록은 실행되지 않습니다. 어떤 값이 입력이 확인할 필요가 있거나 0이 아닌 값이 들어갈 경우에 예외를 발생시킬 필요가 있을 것입니다.

그래서 x값을 확인하기 위한 첫 번째 방법으로 다음과 같이 디버그메시지를 사용할 수 있습니다.

```
int x = new Scanner(System.in).nextInt();

System.out.println("x=" + x);

if(x==0) {
    System.out.println("인증되었습니다.");
}
```

두 번째 방법으로 x에 0이 입력되지 않았을 경우 예외를 발생시킬 수도 있습니다.

```
int x = new Scanner(System.in).nextInt();

if(x==0) {
    System.out.println("인증되었습니다.");
}else {
    throw new Exception("인증에 실패했습니다. x=" + x);
}
```

그런데 위 두 가지 방법들 중에서 첫 번째 방법은 항상 x의 값이 출력되고, 두 번째 방법은 x가 0이 아니면 예외가 발생하게 됩니다. 이는 고객에게 이 프로그램의 신뢰도를 떨어뜨리게 될 것입니다. 물론 첫 번째 방법의 출력문을 다음 코드처럼 사용한다면 프로젝트가 종료될 시점에 Util 클래스의 DEBUG 상수 값만 변경하여 메시지를 출력하지 않도록 할 수도 있습니다.

```
if(Util.DEBUG==true) System.out.println("x=" + x);
```

그런데 assertion 기능을 사용하면 위 두 가지를 한 번에 해결할 수 있습니다. 다음은 assertion이 적용된 코드입니다.

```
int x = new Scanner(System.in).nextInt();

assert x==0 : "인증에 실패했습니다. x=" + x;

if(x==0) {
    System.out.println("인증되었습니다.");
}
```

위의 코드처럼 작성해 놓으면 개발자는 개발 할 때 x의 값이 0이 아닐 경우 그 값을 화면에 출력되도록 할 수 있습니다. 그리고 시스템이 운영될 대 사용자는 x의 값이 0이더라도 메시지나 예외 없이 프로그램을 사용할 수 있을 것입니다.

assertion 기능을 사용하기 위해서는 몇 가지 요구사항을 만족해야 합니다. 첫 번째 컴파일러의 버전입니다. 컴파일러의 버전이 최소 JDK 1.4이상이어야 합니다. 그리고 코드에 assert 문장이 포함되어야 합니다. 마지막으로 세 번째 실행 시 assertion 기능을 사용해야 할 경우(개발 시) -ea 옵션으로 실행시켜야 합니다.

assertion 문장은 assert 키워드를 이용하며 두 가지 사용법이 있습니다.

```
assert Expression1 ;
assert Expression1 : Expression2 ;
```

다음과 같은 형태에서 *Expression*$_1$의 결과는 boolean 형이어야 합니다. 만일 *Expression*$_1$의 결과 값이 false이면 실행 시 AssertionError를 발생시킵니다. 이 때 예외의 메시지는 없습니다.

```
assert Expression1 ;
```

다음과 같은 형태 즉, 두 개의 Expression을 가질 때는 먼저 *Expression*$_1$의 결과는 boolean형이어야 하며, *Expression*$_1$의 결과 값이 false이면 실행 시 *Expression*$_2$의 결과를 메시지로 갖는 AssertionError를 발생시킵니다. 여기서 주의해야 할 점은 '*Expression*$_2$에는 void형 메서드 호출이 올 수 없다'는 것입니다.

```
assert Expression1 : Expression2 ;
```

10.2.1. assertion 코드 컴파일

assert문이 있는 프로그램은 버전이 1.4임을 알려주기 위해 컴파일 시 다음과 같이 옵션을 넣어서 컴파일 해야 합니다. 그러나 이클립스를 사용하신다면 컴파일 걱정은 하지 않아도 됩니다.

```
javac -source 1.4 FileName.java
```

10.2.2. assertion 코드 실행

실행시킬 때 -enableassertions 또는 -ea 옵션을 사용해야 합니다. assertion 코드 실행을 해제하려면 -disableassertion 또는 -da 옵션을 사용해야 합니다. 옵션이 클래스 이름 앞에 들어가는 것을 주의하세요.

```
java -ea FileName
```

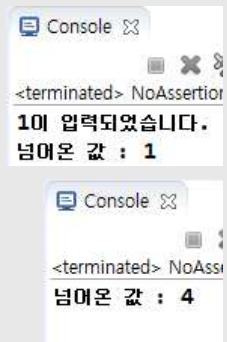
10.2.3. assertion 코드 예

다음 프로그램은 assert키워드의 사용법을 알아보기 위한 assert 키워드가 사용되지 않은

예입니다.

NoAssertionExample.java

```
1:  public class NoAssertionExample {
2:     public static void main(String[] args) {
3:        int i = (int)(Math.random() * 4) + 1;//1,2,3,4 값이 랜덤하게 발생됨
4:        System.out.println("넘어온 값 : " + doIt(i));
5:     }
6:     public static int doIt(int a) {
7:        switch(a) {
8:           case 1:
9:              System.out.println("1이 입력되었습니다.");
10:             break;
11:          case 2:
12:             System.out.println("2가 입력되었습니다.");
13:             break;
14:          case 3:
15:             System.out.println("3이 입력되었습니다.");
16:             break;
17:       }
18:       return a;
19:    }
20: }
```



앞의 프로그램에서 switch문이 사용되었는데 조건을 만족하는 case문이 없으면 실행되는 문장이 없습니다. 따라서 프로그램을 작성하면서 처리할 문장이 없음을 간과할 수 있습니다.

다음 프로그램은 앞의 예제에서 조건을 만족하지 않으면 실행되는 문장이 없는 예외상황을 처리하기 위한 기능을 부여한 예입니다.

AssertionExample.java

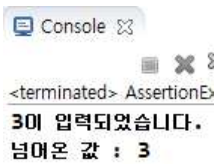
```
1:  public class AssertionExample {
2:     public static void main(String[] args) {
3:        int i = (int)(Math.random() * 4) + 1;//1,2,3,4값이 랜덤하게 발생됨
4:        System.out.println("넘어온 값 : " + doIt(i));
5:     }
6:     public static int doIt(int a) {
7:        switch(a) {
8:           case 1:
9:              System.out.println("1이 입력되었습니다.");
10:             break;
11:          case 2:
12:             System.out.println("2가 입력되었습니다.");
13:             break;
14:          case 3:
15:             System.out.println("3이 입력되었습니다.");
16:             break;
17:          default:
```

```
18:                 assert false : a;
19:         }
20:         return a;
21:     }
22: }
```

이 프로그램에서는 assert문이 사용되었습니다. 실행할 때 입력 값이 case문의 값과 일치하는 값이 없을 경우에는 실행시의 옵션에 따라 예외를 발생시킬 수도 있고, 반대로 그렇지 않게 할 수도 있습니다. 이렇게 하면 사용자가 메서드 호출시 예외 상황에 대하여 더 다양한 방식으로 프로그램을 작성할 수 있습니다.

다음은 앞의 예제를 컴파일하고 실행한 결과를 나타낸 것입니다. 랜덤 값이 4가 발생되었을 경우 각각의 실행결과를 보여주고 있습니다.

-ea 옵션을 주지 않고 실행할 경우

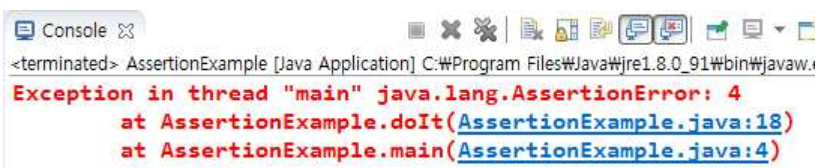


정상적으로 실행되는 경우입니다.

랜덤값이 4일 경우라도 정상 실행됩니다.

-ea 옵션을 주고 실행할 경우

정상적으로 실행되는 경우입니다.



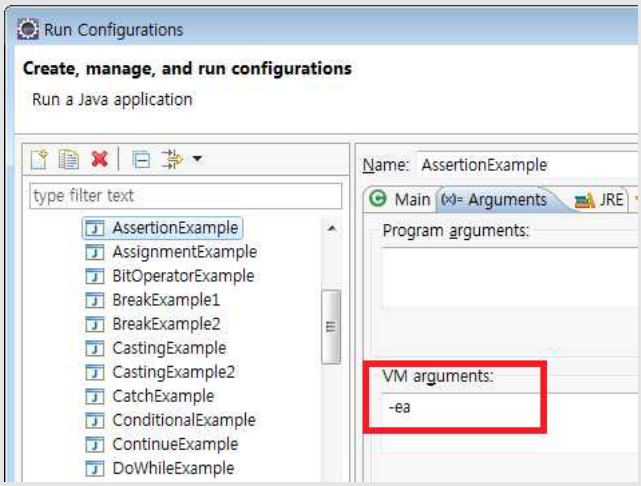
-ea 옵션을 주고 실행할 경우 랜덤 값이 4일 때 AssertionError를 발생시켜 사용자에게 예외를 알릴 수 있습니다.
프로그램 개발 단계에는 -ea 옵션을 주고 실행하여 테스트 하면 비정상적인 상황들에 대해 모니터링이 가능합니다. 프로그램 운영 단계에는 -ea 옵션을 주지 않고 실행하여 예외 상황이 발생하더라도 사용자 화면에 오류가 나타나는 일이 없을 것입니다.

이클립스에서 Assertion기능 사용하기

이클립스는 컴파일러의 버전이 1.5이상이면 자동으로 Assertion 컴파일을 수행합니다. 실행 시 옵션으로 Assertion 기능을 사용할 수 있습니다.

- Run -> Run Configuration...

1. 실행시 Assertion 기능을 활성화 시키고 싶은 클래스를 선택합니다.
2. Arguments 탭을 클릭합니다.
3. VM arguments: 항목에 -ea라고 입력합니다.

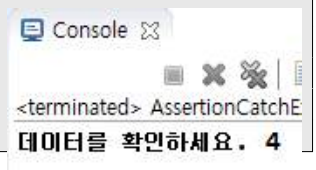


다음 코드는 앞의 코드의 assert문장을 throw문으로 수정할 수 있습니다. AssertionError가 발생한다면 try~catch 문으로 처리할 수 있습니다.

AssertionCatchExample.java

```
1:  public class AssertionCatchExample {
2:     public static void main(String[] args) {
3:        int i = (int)(Math.random() * 4) + 1;//1,2,3,4값이 랜덤하게 발생됨
4:        try {
5:           System.out.println("넘어온 값 : " + doIt(i));
6:        }catch(AssertionError ae) {
7:           System.out.println("데이터를 확인하세요. " + ae.getMessage());
8:        }
9:     }
10:    public static int doIt(int a) {
11:       switch(a) {
12:          case 1:
13:             System.out.println("1이 입력되었습니다.");
14:             break;
15:          case 2:
16:             System.out.println("2가 입력되었습니다.");
17:             break;
18:          case 3:
19:             System.out.println("3이 입력되었습니다.");
20:             break;
```

```
21:           default:
22:              assert false : a;
23:        }
24:        return a;
25:     }
26: }
```



-ea 옵션을 주고 실행했을 경우 랜덤값이 4이면 AssertionError가 발생하고 이를 try ~ catch 블록을 이용해 처리한 결과입니다.

10.2.4. assertion과 exception

이 내용은 분석/설계 시 선행조건과 후행조건을 정의했을 경우 이를 구현하기 위한 과정에 대한 설명입니다. 난이도에 비하여 자주 사용되는 기능은 아닙니다. 내용이 어렵다 생각되면 다음 장으로 넘어가도 됩니다.

Assertion 기능을 후행 조건 확인 및 클래스 불변조건에 사용할 수 있습니다. 후행 조건은 리턴문장 바로 위에 놓여 메서드의 처리 결과를 확인하기 위한 조건입니다. 클래스 불변조건은 클래스의 내부 상태를 체크하여 클래스가 가지는 멤버변수가 변경이 일어났을 경우 처리되도록 하는 코드를 포함할 수 있습니다.

메서드가 수행되기 전에 값의 유효성을 체크하는 선행 조건에는 사용하는 것은 바람직하지 않습니다. 선행 조건은 메서드 블록이 수행되기 위해 만족되어야 할 필요조건입니다. 선행 조건이 만족되지 않으면 이후 메서드 블록은 시작되면 안 됩니다.

후행조건은 메서드 종료 시 만족해야 하는 조건으로 메서드의 정상 동작 여부에 대한 최소한의 판단 기준으로 사용될 수 있습니다. 후행 조건이 만족되지 않으면 메서드가 정상적으로 동작하지 않았다고 할 수 있습니다. 그러나 후행 조건을 만족 하더라도 메서드가 올바르게 수행되었다고 판단할 수는 없습니다.

다음 코드는 선행 조건과 후행 조건을 Assertion으로 처리한 예입니다.

PreAndPostConditionExample.java

```
1:  public class PreAndPostConditionExample {
2:     public static void main(String[] args) {
3:        Account myAccount = new Account("홍길동", 100);
4:  //    myAccount.save(-10);
5:        System.out.println(myAccount);
6:        myAccount.withdraw(10);
7:        System.out.println(myAccount);
8:     }
9:  }
10:
```

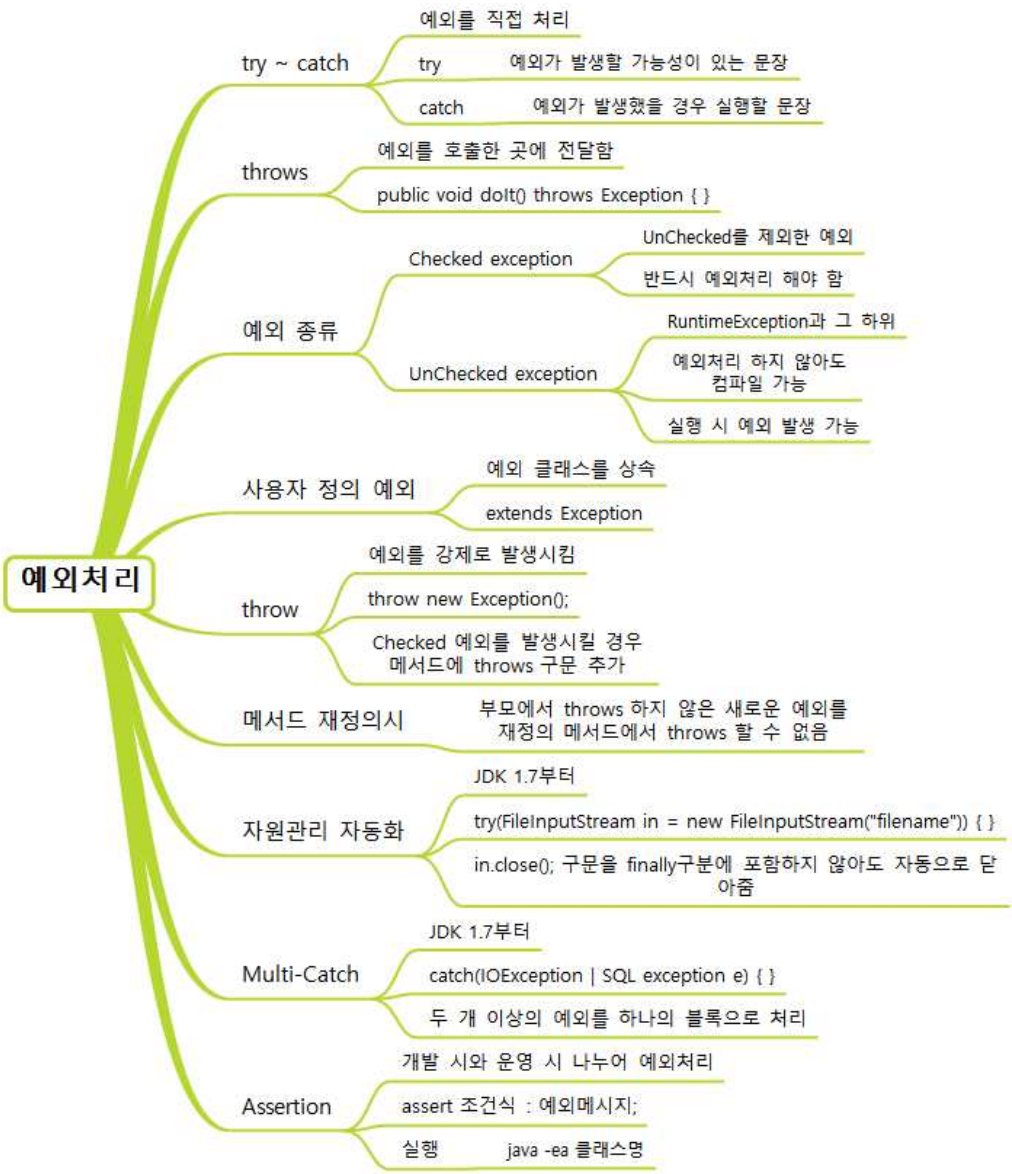
```
11: class Account {
12:     String user;
13:     int balance;
14:
15:     public Account(String user, int balance) {
16:         super();
17:         this.user = user;
18:         this.balance = balance;
19:     }
20:
21:     public void save(int amount) {
22:         if(amount<=0) {
23:             //선행 조건은 확실하게 체크해야 합니다.
24:             throw new IllegalArgumentException("입금액 오류 - " + amount);
25:         }
26:         int preBalance = balance;
27:         System.out.println(amount + "원이 입금되었습니다.");
28:         this.balance = this.balance + amount;
29:
30:         assert (this.balance == preBalance + amount);
31:     }
32:     public void withdraw(int amount) {
33:         if(amount<=0) {
34:             //선행 조건은 확실하게 체크해야 합니다.
35:             throw new IllegalArgumentException("출금액 오류 - " + amount);
36:         }
37:         int preBalance = balance;
38:         System.out.println(amount + "원이 출금되었습니다.");
39:         this.balance = this.balance + amount;
40:
41:         assert (this.balance == preBalance + amount);
42:     }
43:
44:     public String toString() {
45:         return user +"님의 잔고는 " + balance + "입니다.";
46:     }
47: }
```

위 코드에서 save() 메서드와 withdraw() 메서드에서는 선행조건을 명시적으로 비교한 후 예외를 발생시켰습니다. 그러므로 -ea 옵션과는 무관하게 amount가 0 이하일 경우에는 메서드가 실행되지 않게 하였습니다.

후행 조건의 경우 save()메서드와 withdraw() 메서드에 assertion 기능을 사용했습니다. 입금 후행 조건은 입금 후 잔액=입금 전 잔액+입금액 이며 출금 후행조건은 출금 후 잔액=출금 전 잔액-출금액 입니다. 후행조건을 만족하지 못하면 시스템은 정상적으로 동작했다고 판단하기 어렵습니다. 그러나 후행 조건이 충족되었다고 메서드가 올바르게 수행되었다고 판단할 수는 없습니다.

시스템의 요구사항을 구현할 때 선행조건과 후행조건이 있다면 Exception 처리와 Assertion 처리를 통해 구현할 수 있습니다. Assertion은 현업에서 자주 사용되는 기능은 아니므로 참고만하고 넘어가도 됩니다.

10.3. 마인드맵 정리


예외처리
try ~ catch
예외를 직접 처리
try
예외가 발생할 가능성이 있는 문장
catch
예외가 발생했을 경우 실행할 문장
throws
예외를 호출한 곳에 전달함
public void doIt() throws Exception { }
예외 종류
Checked exception
UnChecked를 제외한 예외
반드시 예외처리 해야 함
UnChecked exception
RuntimeException과 그 하위
예외처리 하지 않아도 컴파일 가능
실행 시 예외 발생 가능
사용자 정의 예외
예외 클래스를 상속
extends Exception
throw
예외를 강제로 발생시킴
throw new Exception();
Checked 예외를 발생시킬 경우 메서드에 throws 구문 추가
메서드 재정의시
부모에서 throws 하지 않은 새로운 예외를 재정의 메서드에서 throws 할 수 없음
자원관리 자동화
JDK 1.7부터
try(FileInputStream in = new FileInputStream("filename")) { }
in.close(); 구문을 finally구분에 포함하지 않아도 자동으로 닫아줌
Multi-Catch
JDK 1.7부터
catch(IOException | SQL exception e) { }
두 개 이상의 예외를 하나의 블록으로 처리
Assertion
개발 시와 운영 시 나누어 예외처리
assert 조건식 : 예외메시지;
실행
java -ea 클래스명