Java프로그래밍

7강. 패키지와 예외처리 (교재 6장)

컴퓨터과학과 김희천 교수

한국방송통신대학교

오늘의 학습목차

1. 패키지
2. 예외와 예외처리

1. 패키지

1. 패키지

1) 패키지의 의미

- 관련이 있는 클래스와 인터페이스의 묶음
  - ✓ 클래스와 인터페이스는 패키지의 멤버로 존재
- 전체적으로 계층 구조의 클래스 라이브러리
  - ✓ 패키지(폴더와 유사) 단위로 계층적으로 분류됨

패키지의 용도

- 쉽게 찾아 사용하기 위해
- 이름 충돌을 피하기 위해
  - ✓ graphics.Rectangle와 java.awt.Rectangle는 구분됨
- 접근 제어를 위해

2) 시스템 패키지

- JDK가 제공하는 클래스 라이브러리
  - ✓ JDK와 함께 설치됨
  - ✓ 클래스 파일들은 기능에 따라 패키지로 묶여 분류됨
  - ✓ 일반적으로 jar 파일로 압축되어 있음
  - ✓ C:₩Program Files₩Java₩jdk-15.0.1₩lib₩jrt-fs.jar
  - ✓ C:₩Program Files₩Java₩jdk-15.0.1₩lib₩src.zip 에서 소스를 확인할 수도 있음

3) 시스템 패키지의 사용

- 가장 기본이 되는 최상위 시스템 패키지는 java임
  - ✓ 대부분의 시스템 패키지는 java.으로 시작됨
- Java 프로그램에서 상위 패키지와 하위 패키지의 구분을 위해 도트(.)를 사용
  - ✓ 예: java.lang, java.io, java.awt, java.awt.color, java.util 등
  - ✓ Java 언어에서 가장 기본적 클래스는 java.lang 패키지에 존재
  - ✓ 프로그램에서 클래스를 사용할 때는 java.io.IOException과 같이 표현하는 것이 원칙임

4) 사용자 정의 패키지(1)

◆ 패키지 정의 문법

```
package  패키지이름 ;
// 1개 이상의 클래스나 인터페이스 정의가 나옴
```

✓ package 구문은 소스 코드 맨 앞에 위치해야 함
✓ 패키지 이름은 관례상 모두 소문자로 작명
✓ 도트(.)로 구분하여 계층적으로 정의할 수 있음
✓ 컴파일하면 패키지가 만들어지고(또는 기존 패키지에) 클래스 파일(.class)이 패키지에 저장됨

4) 사용자 정의 패키지(2)

◆ 패키지 정의 예

```
package com.vehicle;
public class Car {
   String szType = "승용차";
}
```

✓ 컴파일 결과로 Car.class가 만들어짐

✓ Car.class는 com.vehicle 패키지에 저장됨

✓ com.vehicle은 어디에?

✓ 컴파일 할 때 -d 옵션 사용하여 지정함

✓ 예: > javac Car.java -d D:₩javaClasses

✓ 이 경우 D:₩javaClasses₩com₩vehicle₩Car.class

5) Eclipse를 사용한 패키지 정의

- 메뉴 [File/New/Package]를 선택
  - ✓ 패키지에 해당하는 폴더가 만들어짐
  - ✓ 생성된 패키지에서 클래스를 만들면 됨
- 메뉴 [File/New/Class]를 선택하여 클래스 이름과 함께 패키지 이름을 입력함

6) 패키지와 클래스의 사용(1)

- 다른 패키지에 존재하는 public 클래스를 사용하려면 기본적으로 패키지 경로를 포함한 완전한 클래스 이름을 사용해야 함
  - ✓ 프로그램에서 자주 사용한다면 import 구문을 사용하는 게 좋음
- 예
  - ✓ graphics.Rectangle myRect
    = new graphics.Rectangle( );
  - ✓ java.util.Scanner s
    = new java.util.Scanner(System.in);

6) 패키지와 클래스의 사용(2)

- import 문
  - ✓ 1개 클래스 또는 패키지에 있는 클래스 전체를 import 할 수 있음

```
import  패키지 이름.클래스 이름;
import  패키지 이름.* ;
```

  - ✓ import 구문은 소스 코드 맨 앞에 위치함
  - ✓ 단, package 구문이 있다면 그 다음에 위치함
  - ✓ 프로그램에서 패키지 경로를 생략하고, 이름만 가지고 클래스나 인터페이스를 사용할 수 있게 함
- Java 프로그램에서 import java.lang.*; 구문은 자동 포함됨

7) 패키지의 사용과 접근 제어

◆ 아래 프로그램에서 package 구문이 없다면 패키지 접근 수준의 Car 클래스를 사용할 수 없음

```
// package com.vehicle;

import com.vehicle.*;

class MyBus extends Bus {  }

public class PackageTest {
   public static void main(String args[]) {
        Bus bus = new Bus( );
        Car car = new Car( ); // 오류
   }
}
```

```
package com.vehicle;

class Car { … }
```

```
package com.vehicle;

public class Bus extends Car { … }
```

8) 클래스 찾기(1)

- 컴파일하거나 실행할 때, 필요한 클래스(A)를 찾아야 함
  - ✓ 컴파일러가 A.class가 위치한 경로 또는 A.class를 포함하고 있는 jar 파일의 존재를 알아야 함
- JVM은 기본 패키지나 확장 패키지 외에 사용자 클래스도 찾을 수 있음
  - ✓ 이때 방법이 필요함

8) 클래스 찾기(2)

- 컴파일러는 환경 변수 CLASSPATH에 지정된 경로에서 사용자 클래스를 찾을 수 있음

- 환경변수 CLASSPATH
  - ✓ CLASSPATH의 경로는 jar 파일을 포함할 수 있음
  - ✓ 예: 프로그램에서 graphics.Circle 클래스를 사용
    - CLASSPATH=경로1;경로2;a.jar라고 가정
    - 이때, 경로1₩graphics₩Circle.class 또는 경로2₩graphics₩Circle.class 가 있거나 a.jar에 ₩graphics₩Circle.class가 있어야 함

2. 예외와 예외처리

2. 예외와 예외처리

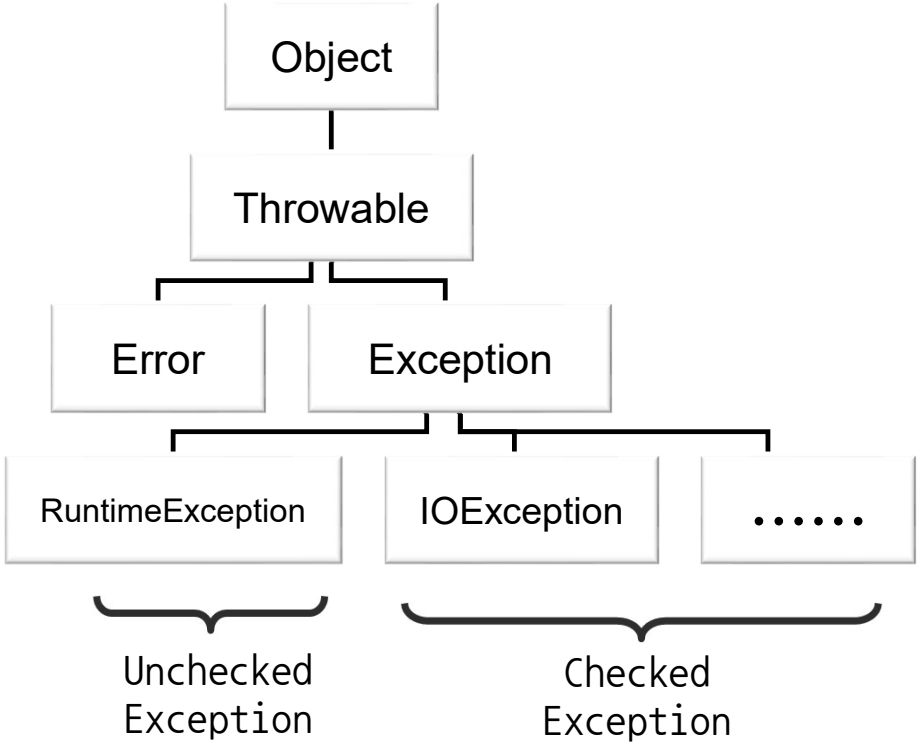
1) 예외와 에러

- 에러(Error)는 심각한 오류로 더 이상의 실행이 불가함
- 예외(Exception)는 경미한 오류로 복구가 가능함
  - ✓ 예외는 정상적 실행 흐름을 방해하는 예외적 사건

예외 발생과 처리

- 메소드를 수행할 때 예외가 발생하면 예외 객체를 만들어 던짐
- 예외처리 코드(exception handler)가 없으면, 오류 메시지가 출력되면서 프로그램이 즉시 종료됨
- 예외처리 코드가 있으면, 예외 객체를 잡아 처리한 뒤, 프로그램은 계속 수행됨
- 예외 객체는 Exception 클래스(또는 하위 클래스)로 표현되며 예외 발생 정보를 가지고 있음

2) 예외 클래스의 계층 구조



3) 예외처리(Exception handling)

- 예외가 발생했을 때 이 상황을 바로잡아 계속 수행하도록 하는 것
- 예외 발생 시, Exception 객체를 생성하고 throw함
  - ✓ throw new MyException( );
- throw된 예외 객체를 예외처리 코드가 catch하여 예외를 처리함

예외의 종류

- checked Exception이 발생할 수 있는 경우, 반드시 명시적인 예외처리가 필요함(예외처리 코드가 없으면 컴파일 오류)
- RuntimeException의 경우, 예외처리를 안 해도 됨
  - ✓ 프로그램을 정확하게 작성하지 않은 경우 발생됨
  - ✓ ArithmeticException, NullpointerException, IndexOutOfBoundsException 등

4) 예외처리 방법

- 직접 처리
  - ✓ 예외가 발생한 곳에서 예외 객체를 잡아서 처리하는 것
  - ✓ try-catch 구문 또는 try-catch-finally 구문을 사용하여 예외를 처리함
  - ✓ 일반 코드와 예외 처리가 분리되어 가독성이 좋아짐

- 간접 처리(예외의 전파)
  - ✓ 예외 발생 가능성이 있는 메소드의 선언에서 괄호 다음에 throws 예외이름을 사용
  - ✓ 그 메소드를 호출한 메소드에게 예외처리를 전달 또는 위임하는 것

5) try-catch-finally 구문

◆ 문법

```
try { … }
catch(ExceptionType1 ex1) { … }
catch(ExceptionType2 ex2) { … }
finally { … }
```

✓ 예외 객체를 throw하는 문장 또는
예외 발생 가능성이 있는 메소드의 호출 부분을 try 블록에 둠

✓ catch 블록은 1개의 예외 유형 인자를 가지는 메소드와 유사

- 처리해야 하는 예외 유형이 여럿이면 catch 블록도 여럿이 됨

✓ finally 블록은 생략 가능

6) try-catch-finally 구문의 실행

- 예외가 발생하면 try 블록은 즉시 종료됨
- catch 블록이 여럿이면, 가장 적합한
  (발생된 예외 자료형과 일치하거나 상위 유형) 하나만 실행됨
- 예외가 발생하지 않으면 catch 블록은 실행되지 않음
- finally 블록은 예외 발생과 무관하게 try 블록이 종료된 후
  항상 실행됨
  - ✓ 할당 받아 사용했던 리소스를 원상복구하기 위해
    finally 블록을 주로 사용함
  - ✓ 예: try 블록에서 open 했던 파일을 close하는 코드를
    finally 블록에 둠

7) 예외의 직접 처리 예

```
public class A {
  public void problem( ) throws RuntimeException  {
            throw new RuntimeException( );
  }
  public void tryThis( ) {
     try {
                problem( );
                System.out.print("1");
     } catch (RuntimeException x) {
                System.out.print("2");
     } catch (Exception x) {
                System.out.print("3");
     } finally {
                System.out.print("4");
     }
     System.out.print("5");
  }
  public static void main(String[ ] args){
            A a = new A( );
            a.tryThis( );
  }
}
```

```
2
4
5
```

8) 예외의 간접 처리(1)

- '예외를 발생시킬 수 있는 메소드'를 호출하는 쪽에 예외 처리를 위임하는 것(예외의 전파)
  - ✓ 메소드 선언에서 발생시킬 수 있는 예외유형을 표시함
  - ✓ 즉, 메소드 선언에서 괄호 다음에 throws 예외유형을 사용

```
public char getInput( ) throws IOException {
        nInput = System.in.read( ) ; //예외 발생 가능
}
```

- 메소드 선언에서 throws 절이 표시된 메소드를 호출하는 메소드는 예외 처리를 해야 함

```
try {
        c = obj.getInput( );
}
catch(IOException ex ) {
}
```

8) 예외의 간접 처리(2)

- 예외를 발생시킬 수 있는 메소드
  - ✓ public FileInputStream(String name)
    throws FileNotFoundException
    - FileInputStream 클래스의 생성자
  - ✓ public int read( ) throws IOException
    - InputStream(또는 Reader) 클래스의 메소드
- 위와 같은 메소드를 호출할 때는 반드시 예외 처리가 필요함
  - ✓ 위에서 발생 가능한 예외 유형은 checked Exception의 예

9) 예외 처리 프로그램 예(1)

```
import java.io.*;

public class ExceptionTest1 {
   public static void main(String args[ ]) {
     int b = 0;
     try {
         b = System.in.read( );
     } catch (IOException ex) {
         System.out.println(ex);
     }
     System.out.println((char)b);
   }
}
```

9) 예외 처리 프로그램 예(2)

```
import java.io.*;
class CharInput {
   int nInput = 0;
   public char getInput( ) throws IOException {
     nInput = System.in.read( );
     return (char)nInput;
   }
}
public class ExceptionTest4 {
   public static void main(String args[ ]) {
     CharInput charInput = new CharInput( );
     try {
         System.out.println(charInput.getInput( ));
     } catch (IOException ex) {
         System.out.println(ex);
     }
   }
}
```

10) 사용자 정의 예외

- 사용자가 직접 예외 클래스를 작성할 수 있음
- 일반적으로 Exception 클래스를 상속받음
- throw 구문을 사용하여, 필요할 때 예외 객체를 던질 수 있음

```
class MyException extends Exception{
   public MyException( ) { super( ); }
   public String toString( ) { return "MyException"; }
}

class MyExceptionTest {
   public void testFunc(int x) throws MyException {
     if (x > 10) throw new MyException( );
   }
}
```

Java프로그래밍

다음시간안내

8강. java.lang 패키지