예외 처리

강사 : 강병준

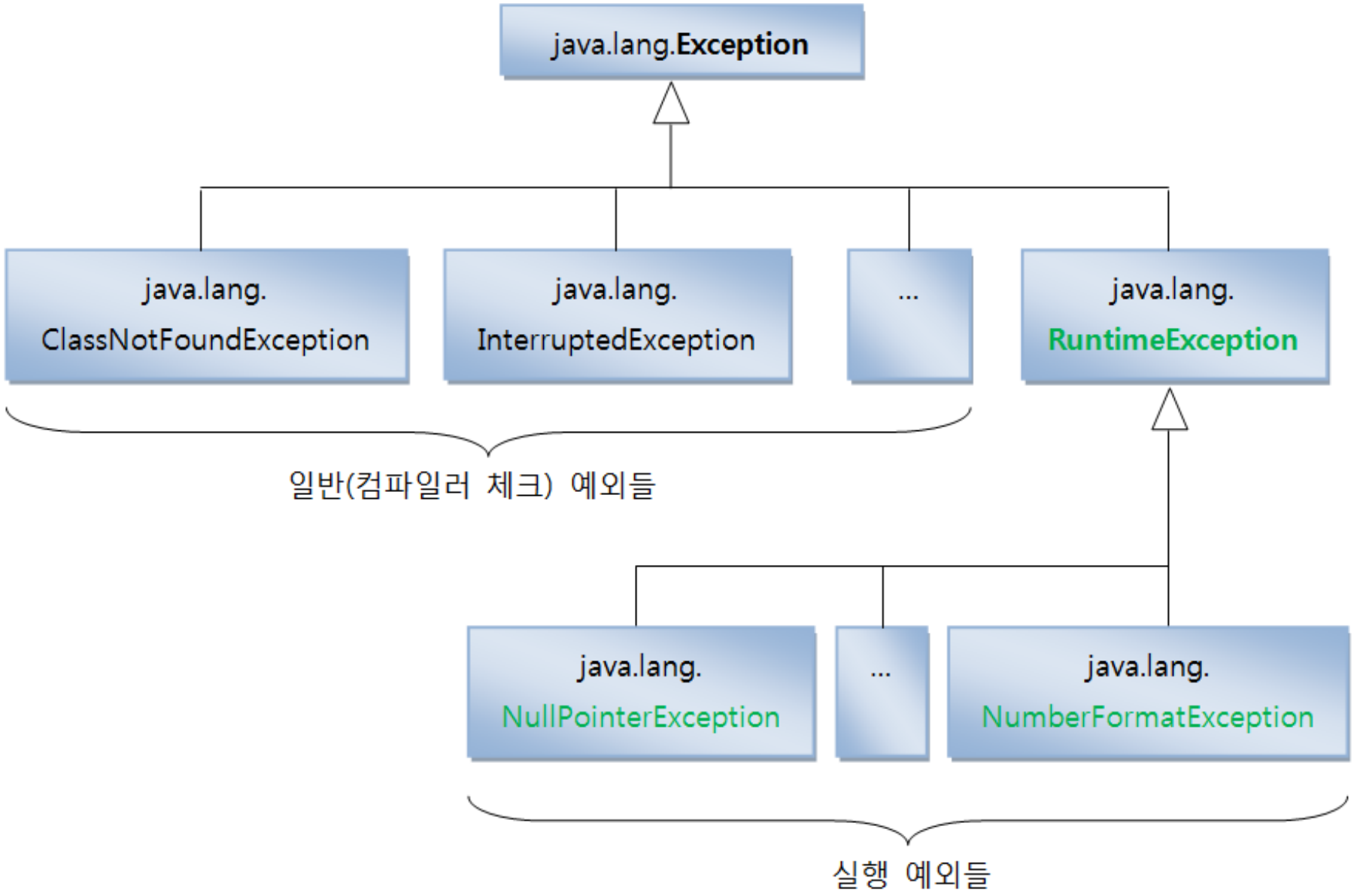
예외와 예외 클래스

- 오류의 종류
  - 에러(Error)
    - 하드웨어의 잘못된 동작 또는 고장으로 인한 오류
    - 에러가 발생되면 프로그램 종료
    - 정상 실행 상태로 돌아갈 수 없음
  - 예외(Exception)
    - 사용자의 잘못된 조작 또는 개발자의 잘못된 코딩으로 인한 오류
    - 예외가 발생되면 프로그램 종료
    - 예외 처리 추가하면 정상 실행 상태로 돌아갈 수 있음

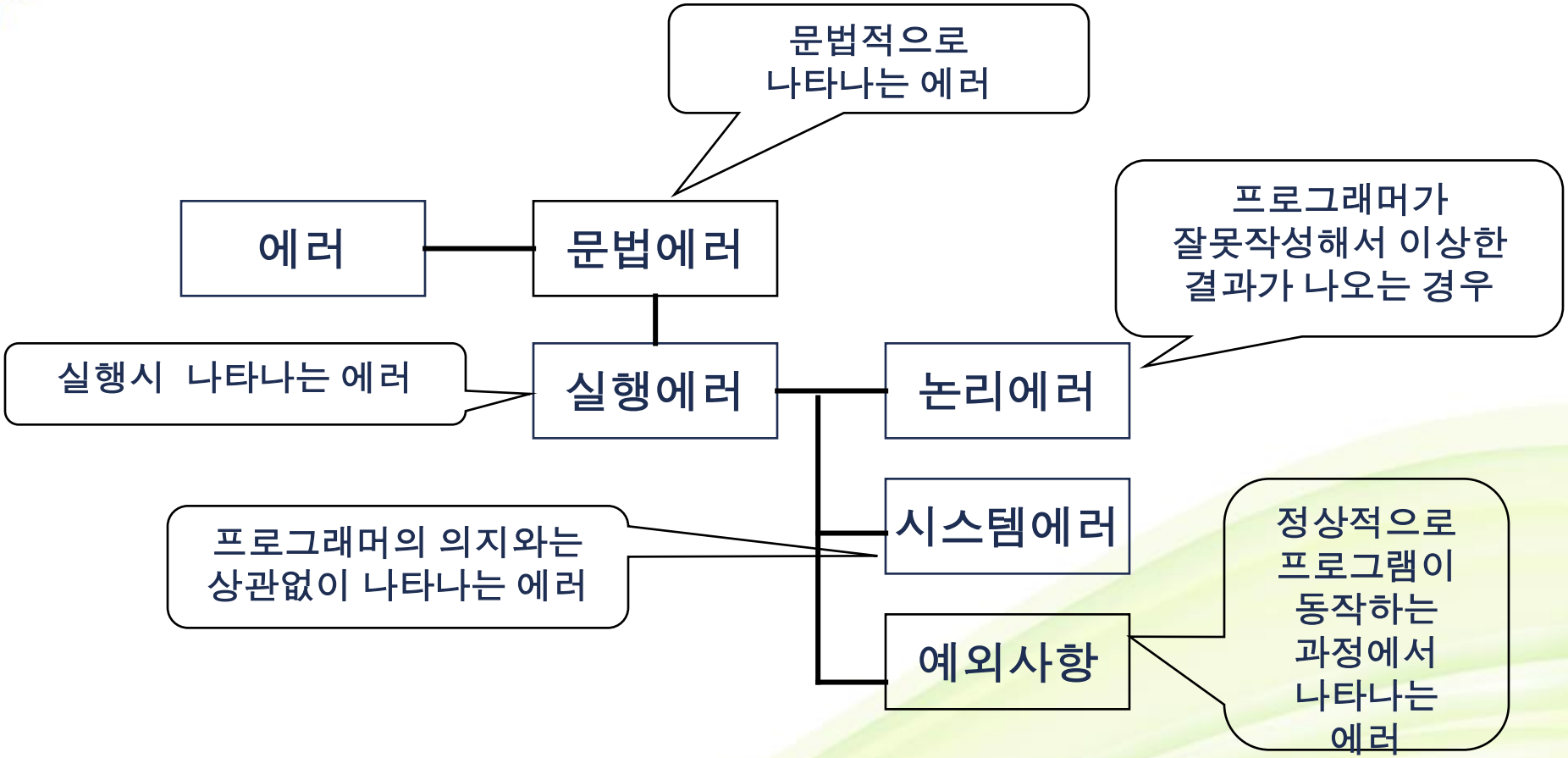
예외와 예외 클래스

- 예외의 종류
  - 일반(컴파일 체크) 예외(Exception)
    - 예외 처리 코드 없으면 컴파일 오류 발생
  - 실행 예외(RuntimeException)
    - 예외 처리 코드를 생략하더라도 컴파일이 되는 예외
    - 경험 따라 예외 처리 코드 작성 필요

예외와 예외 클래스


java.lang.Exception
java.lang.
ClassNotFoundException
java.lang.
InterruptedException
...
java.lang.
RuntimeException
일반(컴파일러 체크) 예외들
java.lang.
NullPointerException
...
java.lang.
NumberFormatException
실행 예외들

에러


문법적으로
나타나는 에러
에러
문법에러
프로그래머가
잘못작성해서 이상한
결과가 나오는 경우
실행시 나타나는 에러
실행에러
논리에러
시스템에러
프로그래머의 의지와는
상관없이 나타나는 에러
정상적으로
프로그램이
동작하는
과정에서
나타나는
에러
예외사항

예외의 개요

예외 : 프로그램 실행 중에 발생하는 예기치 않은 사건

예외가 발생하는 예

- 정수를 0으로 나누는 경우
- 배열의 첨자가 음수 또는 범위를 벗어나는 경우
- 부적절한 형변환이 일어나는 경우
- 입출력을 위한 파일이 없는 경우 등

자바 언어는 프로그램에서 예외를 처리할 수 있는 기법을 제공

자바는 예외를 객체로 취급

예외 관련 클래스를 **java.lang** 패키지에서 제공

자바 프로그램에서는 **Error, RuntimeException** 클래스의 하위 클래스들을 제외한 모든 예외를 처리하여야 한다

일반적으로 **Error, RuntimeException** 클래스(하위 클래스 포함)들과 연관된 예외는 프로그램에서 처리하지 않는다

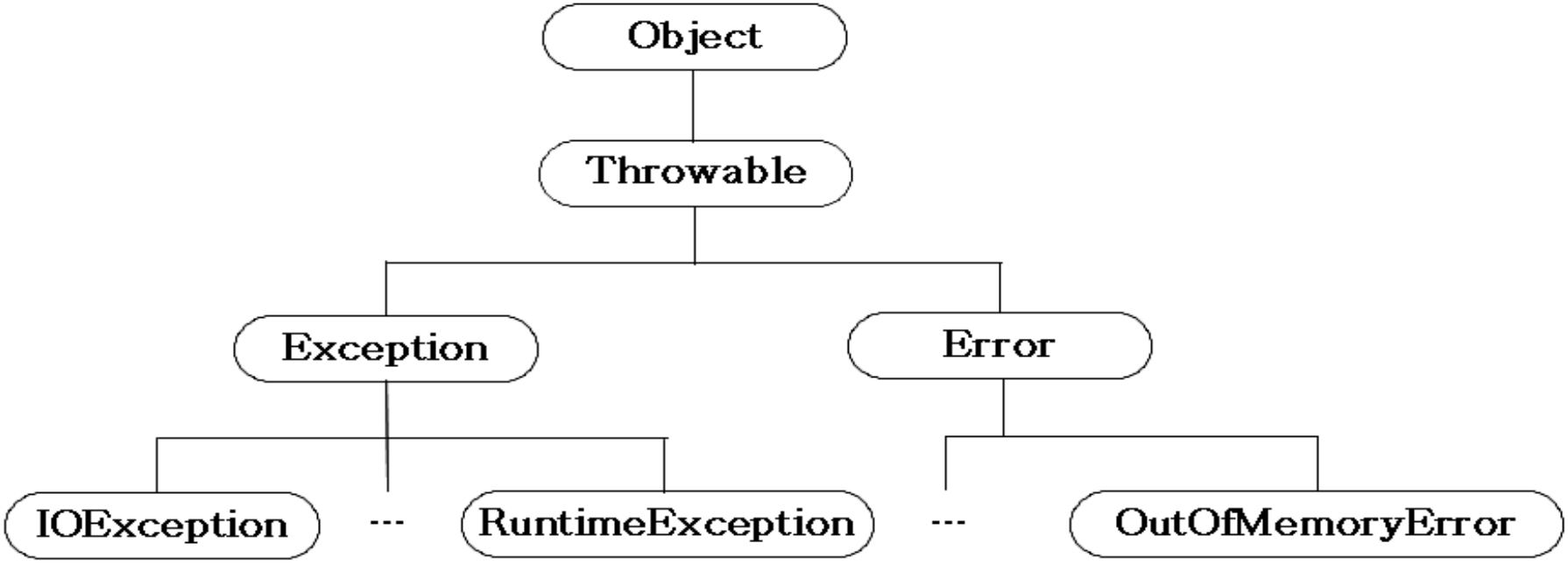
- 이유 : 예외를 처리하여 얻는 이득보다 예외를 처리하기 위한 노력이 너무 크기 때문

예외 클래스의 계층구조

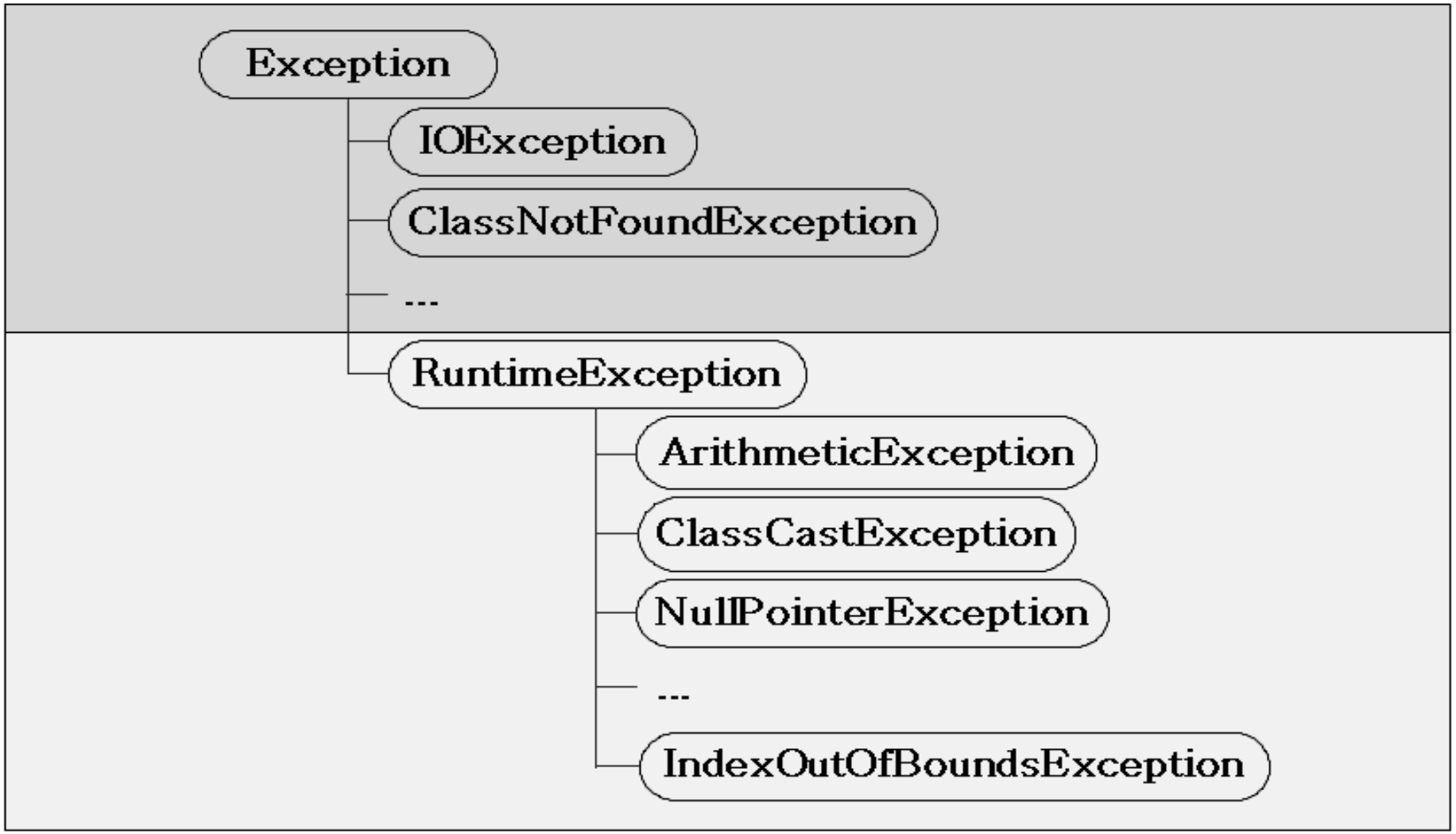
- 예외 클래스는 크게 두 그룹으로 나뉜다.

RuntimeException클래스들 - 프로그래머의 실수로 발생하는 예외 ← 예외처리 필수

Exception클래스들 - 사용자의 실수와 같은 외적인 요인에 의해 발생하는 예외 ← 예외처리 선택



예외클래스 계층도



Exception클래스와 RuntimeException클래스 중심의 상속계층도

예외관련 클래스

❑ Exception 클래스의 하위 클래스

NoSuchMethodException 메소드가 존재하지 않을 때

ClassNotFoundException 클래스가 존재하지 않을 때

CloneNotSupportedException 객체의 복제가 지원되지 않는 상황에서의 복제 시도

IllegalAccessException 클래스에 대한 부정 접근

InstantiationException 추상클래스나 인터페이스로부터 객체 생성하려 할 때

InterruptedException 스레드가 인터럽트 되었을 때

RuntimeException 실행시간 예외가 발생할 때

예외관련 클래스

❑ RuntimeException 클래스의 하위 클래스

ArithmeticException	0으로 나누는 등의 산술적인 예외
NegativeArraySizeException	배열의 크기를 지정할 때 음수의 사용
NullPointerException	null 객체의 메소드나 멤버 변수에 접근할 때
IndexOutOfBoundException	배열이나 스트링의 범위를 벗어난 접근. 하위클래스로 ArrayIndexOutOfBoundsException 클래스와 StringIndexOutOfBoundsException 클래스를 제공
SecurityException	보안을 이유로 메소드를 수행할 수 없을 때

```
class ExceptionEx2 {

    public static void main(String args[]) {
        int number = 100;
        int result = 0;

        for(int i=0; i < 10; i++) {
                result = number / (int)(Math.random() * 10);
                // 7번째 라인
                System.out.println(result);
        }

    }
}
```

예외를 처리하는 방법

예외를 처리하는 방법은 두 가지

- 예외가 발생된 메소드 내에서 처리하는 방법**(try, catch** 절 사용**)**
- 예외가 발생된 메소드를 호출한 메소드에게 예외의 처리를 넘겨주는 방법**(throws** 절 사용**)**

Try/catch문에 의한 예외처리

형식

```
try{
   예외가 발생할 만한 코드
}catch(해당 Exception){
   예외처리를 위한 루틴
}
```

☞ **try문 : try문 바로 다음은 반드시 블록{ }있어야 하고 예외가 발생할 만한 코드를 기술한다.**

☞ **catch문 : try 블록 안에 코드를 실행하다가 예외가 발생하면 처리할 수행문을 기술한다. catch문은 예외유형에 따라 여러개를 기술할수 있다.**

```
public class Except1 {
	public static void main(String[] args) {
		int number = 100, result = 0;
		for (int i = 0; i < 10; i++) {
			//			0 ~ 9
			try {
				result = number /(int)(Math.random()* 10);
			}catch (ArithmeticException e) {
				System.out.println("0으로 못나눠" +
e.getMessage());
			}
			System.out.println("나눗셈 결과 : " + result);
		}
	}
}
```

```
class ExceptionEx5 {
    public static void main(String args[]) {
    // 0으로 나눠서 고의로 ArithmeticException을 발생시킨다.
                System.out.println(1);
                System.out.println(2);

                try {
                        System.out.println(3);
                        System.out.println(0/0);
                        System.out.println(4);    // 실행되지 않는다.
                } catch (ArithmeticException ae) {
                        System.out.println(5);
                }       // try-catch의 끝

                System.out.println(6);

    }     // main메서드의 끝
}
```

```
public class Except2 {
    public static void main(String[] args) {
            Scanner sc = new Scanner(System.in);
            int num1 = 0, num2 = 0; // num1, num2 사용가능한 지역을 확장
            while(true) {
                    System.out.println("첫번째 숫자를 입력하세요");
                    String str = sc.nextLine();
                    if (str.equals("x")) break;
//                  try catch로 예외하면 그 경우를 제외하고 계속 진행할 수 있다
                    try {
                            num1 = Integer.parseInt(str);
                            System.out.println("두번째 숫자를 입력하세요");
                            num2 = Integer.parseInt(sc.nextLine());
                            System.out.printf("%d/%d=%d\n",num1,num2,num1/num2);
                    }catch (ArithmeticException e) {
                            System.out.println("영으로 나눌 수 없습니다");
                    }catch (NumberFormatException e) {
                            System.out.println("숫자로 변경할 수 없는 문자입니다");
                    }catch (Exception e) { // 범위가 넓은 처리를 아래에 배치
                            System.out.println(e.getMessage());
                    }
            }
            System.out.println("프로그램 종료");
            sc.close();
    }
}
```

실행 예외(RuntimeException)

- NullPointerException
  - 객체 참조가 없는 상태
    - null 값 갖는 참조변수로 객체 접근 연산자인 도트(.) 사용했을 때 발생

```
String data = null;
System.out.println(data.toString());
```

```
public class NullPointerExceptionExample {
        public static void main(String[] args) {
                String data = null;
                System.out.println(data.toString());
        }
}
```

실행 예외(RuntimeException)

ArrayIndexOutOfBoundsException

배열에서 인덱스 범위 초과하여 사용할 경우 발생

```
public class ArrayIndexOutOfBoundsExceptionExample {
        public static void main(String[] args) {
                if(args.length == 2) {
                        String data1 = args[0];
                        String data2 = args[1];
                        System.out.println("args[0]: " + data1);
                        System.out.println("args[1]: " + data2);
                } else {
                        System.out.println("[실행 방법]");
                        System.out.print("java
ArrayIndexOutOfBoundsExceptionExample  ");
                        System.out.print("값1  값2");
                }
        }
}
```

다중Try/catch문에 의한 예외처리

형식

```
try{
   예외가 발생할 만한 코드
}catch(해당 Exception){
   예외처리를 위한 루틴
} catch(해당 Exception){
   예외처리를 위한 루틴
} catch(해당 Exception){
   예외처리를 위한 루틴
}
```

finally문

▶ **finally문**

☞ 예외의 발생 여부에 상관없이 반드시 실행이 되어야 할 때 사용한다.

형식

```
try{
   예외가 발생할 만한 코드
}catch(해당 Exception){
   예외처리를 위한 루틴
}finally{
   실행될 문장
}
```

finally문-예제

```
import java.io.FileNotFoundException;
import java.io.FileReader;    import java.io.IOException;

public class Ex03 {
   public static void main(String[] args) {
        FileReader reader;
       char[] buffer = new char[100];;
        String file_name = ".classpath";
        try{
              reader = new FileReader(file_name);
              reader.read(buffer,0,100);
              String str = new String(buffer);
              System.out.println("읽은건 " + str);
              reader.close();
        } catch(FileNotFoundException e){
               System.out.println("그런 파일 없습당");
       }catch(IOException e){
              System.out.println("읽다가 에러났슈");
      }
       finally{
               System.out.println("어쨌거나 읽었어요");
     }
   }
}
```

throw / throws/사용자 정의 Exception

▶ **throw**는 프로그래머가 임의로 예외를 발생시킬 때 사용된다.

형식) **throw** 예외객체;

throw new 예외객체타입(매개변수);

예) **throw new MyExceptionOne();**

▶ **throws**는 예외를 직접처리하지 않고 자신을 호출한 메소드에게 예외를 넘겨주는 방법에 사용

형식) 메소드(매개변수) **throws** 해당**Exception,.....**

예) **public int add(int a) throws XXException, YYException{....}**

▶ 사용자 정의 **Exception**은 상위 클래스에서 **Exception**을 상속받으면 된다.

형식) **class** 클래스명 **extends Exception{ }**

```
class ExceptionEx6   {

    public static void main(String args[])          {
            try {
                    Exception e = new Exception("고의로 발생시켰음.");
                    throw e;   // 예외를 발생시킴
                    //  throw new Exception("고의로 발생시켰음.");
                     // 위의 두 줄을 한 줄로 줄여 쓸 수 있다.

            } catch (Exception e)          {
                    System.out.println("에러 메시지 : " + e.getMessage());
                    e.printStackTrace();
            }
            System.out.println("프로그램이 정상 종료되었음.");
     }
}
```

메서드에 예외 선언하기

- 예외를 처리하는 또 다른 방법
- 사실은 예외를 처리하는 것이 아니라, 호출한 메서드로 전달해주는 것
- 호출한 메서드에서 예외처리를 해야만 할 때 사용

```
void method() throws Exception1, Exception2, ... ExceptionN {
    // 메서드의 내용
}
```

[참고] 예외를 발생시키는 키워드 throw와 예외를 메서드에 선언할 때 쓰이는 throws를 잘 구별하자.

```
public final void wait()
            throws InterruptedException
```

Causes current thread to wait until another thread invokes the
() method for this object. In other words, this method behaves
the call wait(0).

Throws:

IllegalMonitorStateException - if the current thread is not the owner
InterruptedException - if another thread interrupted the current thre
current thread was waiting for a notification. The *interrupted statu*
cleared when this exception is thrown.

See Also:

notify(), notifyAll()

java.lang
Class IllegalMonitorStateException

java.lang.Object
  └ java.lang.Throwable
    └ java.lang.Exception
      └ java.lang.RuntimeException
        └ **java.lang.IllegalMonitorStateException**

java.lang
Class InterruptedException

java.lang.Object
  └ java.lang.Throwable
    └ java.lang.Exception
      └ **java.lang.InterruptedException**

```
class ExceptionEx18 {

        public static void main(String[] args) throws Exception {
                method1();
            // 같은 클래스내의 static멤버이므로 객체생성없이 직접 호출가능.
        }           // main메서드의 끝

        static void method1() throws Exception {
                method2();
        }           // method1의 끝

        static void method2() throws Exception {
                throw new Exception();
        }           // method2의 끝

}
```

예외 되 던지기(re-throwing)

- 예외를 처리한 후에 다시 예외를 생성해서 호출한 메서드로 전달하는 것
- 예외가 발생한 메서드와 호출한 메서드, 양쪽에서 예외를 처리해야 하는 경우에 사용.

```
class ExceptionEx23 {
  public static void main(String[] args)          {
          try  {      method1();
          } catch (Exception e)        {
             System.out.println("main메서드에서 예외가 처리");
          }
  }       // main메서드의 끝
  static void method1() throws Exception {
          try {
                  throw new Exception();
          } catch (Exception e) {
              System.out.println("method1메서드에서 예외가 처리");
                  throw e;  // 다시 예외를 발생시킨다.
          }
   }      // method1메서드의 끝
}
```

사용자정의 예외 만들기

- 기존의 예외 클래스를 상속받아서 새로운 예외 클래스를 정의할 수 있다.

```
class MyException extends Exception {
     MyException(String msg) {  // 문자열을 매개변수로 받는 생성자
       super(msg); // 조상인 Exception클래스의 생성자를 호출한다.
     }
}
```

- 에러코드를 저장할 수 있게 ERR_CODE와 getErrCode()를 멤버로 추가

```
class MyException extends Exception {
    // 에러 코드 값을 저장하기 위한 필드를 추가 했다.
    private final int ERR_CODE;

    MyException(String msg, int errCode) {  // 생성자.
       super(msg);
       ERR_CODE = errCode;
    }

    MyException(String msg) {  // 생성자.
       this(msg,100);            // ERR_CODE를 100(기본값)으로 초기화한다.
    }

    public int getErrCode() {  // 에러 코드를 얻을 수 있는 메서드도 추가했다.
         return ERR_CODE;     // 이 메서드는 주로 getMessage()와 함께 사용될 것이다.
    }
}
```

throw / throws/사용자정의 예제

```
class MyExceptionOne extends Exception{}
class MyExceptionTwo extends Exception{}
public class ExcepTest{
        public void check(int i) throws MyExceptionOne, MyExceptionTwo{
                if(i==2)
                        throw new MyExceptionOne();
                else if(i==3)
                        throw new MyExceptionTwo();
        }
        public static void main(String[] arg){
                ExcepTest e = new ExcepTest();
                try{
                        e.check(2);
                }catch(MyExceptionOne one){
                        System.out.println("MyExceptionOne happen!!");
                }catch(MyExceptionTwo two){
                        System.out.println("MyExceptioTwo");
                }
        }
}
```

예외 정보 얻기

- getMessage()
  - 예외 발생시킬 때 생성자 매개값으로 사용한 메시지 리턴

```
throw new XXXException("예외  메시지");
```

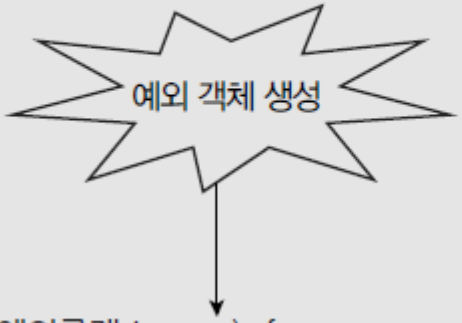
  - 원인 세분화하기 위해 예외 코드 포함(예: 데이터베이스 예외 코드)

```
} catch(Exception  e) {
    String message  = e.getMessage();
}
```

예외 정보 얻기

printStackTrace()

- 예외 발생 코드 추적한 내용을 모두 콘솔에 출력
- 프로그램 테스트하면서 오류 찾을 때 유용하게 활용




```
try {

} catch(예외클래스    e) {
    //예외가 가지고 있는 Message 얻기
    String message = e.getMessage();

    //예외의 발생 경로를 추적
    e.printStackTrace();
}
```

예외처리 권장 사항

❖가능하면 예외처리 구문을 사용하지 않고 논리적으로 해결하고 특히 반복문 내에서의 예외처리 구문을 피하는 것이 좋다.

❖가능하면 표준 예외 클래스를 이용해서 예외를 던지는 것이 좋다.

❖catch 처리 구문에서 여러 종류의 예외를 한꺼번에 처리하고자 하는 경우에는 매개변수를 Throwable 보다는 Exception으로 처리하는 것이 좋다.

연습문제

아래의 코드가 수행되었을 때의 실행결과를 적으시오.

```
class Ex01 {
    static void method(boolean b) {
        try {
            System.out.println(1);
            if(b) throw new ArithmeticException();
            System.out.println(2); // 예외가 발생하면 실행되지 않는 문장
        } catch(RuntimeException r) {
            System.out.println(3);
            return; // 메서드를 빠져나간다.(finally블럭을 수행한 후에)
        } catch(Exception e) {
            System.out.println(4);
            return;
        } finally {
            System.out.println(5); // 예외발생여부에 관계없이 항상 실행되는 문장
        }
        System.out.println(6);
    }

    public static void main(String[] args) {
        method(true);
        method(false);
    } // main
}
```

2. 아래의 코드가 수행되었을 때의 실행결과를 적으시오

```
class Ex02 {
    static void method(boolean b) {
        try {
            System.out.println(1);
            if(b) System.exit(0);
            System.out.println(2);
        } catch(RuntimeException r) {
            System.out.println(3);
            return;
        } catch(Exception e) {
            System.out.println(4);
            return;
        } finally {
            System.out.println(5);
        }
        System.out.println(6);
    }

    public static void main(String[] args) {
        method(true);
        method(false);
    } // main
}.
```