JAVA

- 예외처리

예외처리(Exception Handling)

예외처리에 대한 이해와 try~catch문의 기본

예외처리의 정의와 목적

- 에러(error)는 어쩔 수 없지만, 예외(exception)는 처리해야 한다.

에러(error) - 프로그램 코드에 의해서 수습될 수 없는 심각한 오류
예외(exception) - 프로그램 코드에 의해서 수습될 수 있는 다소 미약한 오류

- 예외처리의 정의와 목적

예외처리(exception handling)의
정의 - 프로그램 실행 시 발생할 수 있는 예외의 발생에 대비한 코드를 작성하는 것
목적 - 프로그램의 비정상 종료를 막고, 정상적인 실행상태를 유지하는 것

[참고] 에러와 예외는 모두 실행 시(runtime) 발생하는 오류이다.

If문을 이용한 예외 처리

- 나이를 입력하라고 했는데, 0보다 작은 값이 입력이 되었다.
- 나눗셈을 위한 두 개의 정수를 입력 받는데, 제수(나누는 수)로 0이 입력이 되었다.
- 주민등록번호 13자리만 입력을 하라고 했더니, 중간에 –를 포함하여 14자리를 입력하였다.

이렇듯 프로그램의 실행 도중에 발생하는 문제의 상황을 가리켜 예외라 한다.

예외는 컴파일 오류와 같은 문법의 오류와는 의미가 다르다.

If문을 이용한 예외 처리

```
System.out.print("피제수 입력: ");
int num1=keyboard.nextInt();
System.out.print("제수 입력: ");
int num2=keyboard.nextInt();

if(num2==0)
{
        System.out.println("제수는 0이 될 수 없습니다.");
        i-=1;
        continue;
}
```

이것이 지금까지 우리가 사용해온 예외의 처리 방식이다.

이는 if문이 프로그램의 주 흐름인지, 아니면 예외의 처리인지 구분되지 안된 다는 단점이 있다.

예외처리구문 – try-catch

- 예외를 처리하려면 try-catch문을 사용해야 한다.

```
try {
      // 예외가 발생할 가능성이 있는 문장들을 넣는다.
} catch (Exception1 e1) {
      // Exception1이 발생했을 경우, 이를 처리하기 위한 문장을 적는다.
} catch (Exception2 e2) {
      // Exception2가 발생했을 경우, 이를 처리하기 위한 문장을 적는다.
...
} catch (ExceptionN eN) {
      // ExceptionN이 발생했을 경우, 이를 처리하기 위한 문장을 적는다.
}
```

[참고] if문과 달리 try블럭이나 catch블럭 내에 포함된 문장이 하나라고 해서 괄호{}를 생략할 수는 없다.

try-catch문에서의 흐름

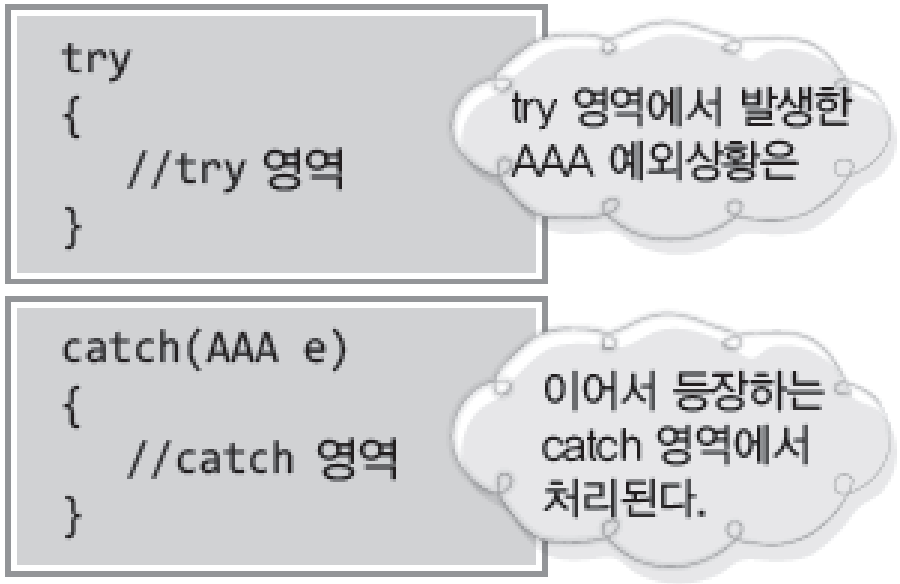
▶ try블럭 내에서 예외가 발생한 경우,

1. 발생한 예외와 일치하는 catch블럭이 있는지 확인한다.
2. 일치하는 catch블럭을 찾게 되면, 그 catch블럭 내의 문장들을 수행하고 전체 try-catch문을 빠져나가서 그 다음 문장을 계속해서 수행한다. 만일 일치하는 catch블럭을 찾지 못하면, 예외는 처리되지 못한다.

▶ try블럭 내에서 예외가 발생하지 않은 경우,

1. catch블럭을 거치지 않고 전체 try-catch문을 빠져나가서 수행을 계속한다.

try~catch문



Try는 예외발생의 감지대상을 감싸는 목적으로 사용된다.

그리고 catch는 발생한 예외상황의 처리를 위한 목적으로 사용된다.

try~catch의 장점 중 하나는!

*** try 영역을 보면서.. 아! 예외 발생 가능 지역이구나!**

*** catch 영역을 보면서.. 아! 예외처리 코드이구나!**

try~catch문

```
try
{
        System.out.println("나눗셈 결과의 몫: "+(num1/num2));
        System.out.println("나눗셈 결과의 나머지: "+(num1%num2));
}
catch(ArithmeticException e)
{
        System.out.println("나눗셈 불가능");
        System.out.println(e.getMessage());
}
System.out.println("프로그램을 종료합니다.");
```

1
2
3

1. 예외 발생 : 0으로 나누는 예외상황의 발생 감지.
2. 참조 값이 전달되면서 Catch 영역 실행
   : 가상머신에 의해 생성된 인스턴스의 참조 값 전달
3. catch 영역 실행 후, try~catch 다음 문장 실행.

try~catch문

```
import java.util.Scanner;

class DivideByZero
{
	public static void main(String[] args) {
		System.out.print("두 개의  정수 입력: ");
		Scanner keyboard=new Scanner(System.in);
		int num1=keyboard.nextInt();
		int num2=keyboard.nextInt();
		try
		{
			System.out.println("나눗셈 결과의 몫: "+(num1/num2));
			System.out.println("나눗셈 결과의 나머지: "+(num1%num2));
		}
		catch(ArithmeticException e)
		{
			System.out.println("나눗셈 불가능");
			System.out.println(e.getMessage());
		}
		System.out.println("프로그램을 종료합니다.");
	}
}
```

try~catch문

```
class ExceptionEx {
	public static void main(String args[]) {
		System.out.println("try~catch 실행 순서");
		System.out.println(1);
		System.out.println(2);

		try {
			System.out.println(3);
			System.out.println(4);
		} catch (Exception e)	{
			System.out.println(5);
		} // try-catch의 끝
		System.out.println(6);
	}	// main메서드의 끝
}
```

try~catch문

```
class ExceptionEx {
	public static void main(String args[]) { // 0으로 나눠서 고의로 ArithmeticException을 발생시킨다.
		System.out.println("try~catch 실행 순서");
		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
}
```

적절한 try 블록의 구성

Try문 내에서 예외 상황이 발생하고 처리된 다음에는,
나머지 try문을 건너뛰고, try~catch의 이후를 실행한다는 특징으로 인해서
트랜잭션(Transcation)의 구성이 용이하다.

적절한 try 블록의 구성

```
try
{
        int num = num1/num2 ;
}
catch(ArithmeticException e)
{
        .....
}
        System.out.println( " 정수형 나눗셈이 정상적으로 진행되었습니다.");
          System.out.println("나눗셈 결과: "+num);
```

```
try
{
        int num = num1/num2 ;
          System.out.println( " 정수형 나눗셈이 정상적으로 진행되었습니다.");
          System.out.println("나눗셈 결과: "+num);
}
catch(ArithmeticException e)
{
        .....
}
```

ArithmeticException 클래스

- **ArithmeticException**과 같이 예외상황을 알리기 위해 정의된 클래스들을 가리켜 “예외클래스”라 한다.
- **getMessage**는 예외상황을 알리기 위해 정의된, 모든 예외 클래스들이 상속하는 **Throwable 클래스**에 정의된 메소드 이다.
- getMessage 메소드는 예외가 발생한 원인정보를 **문자열의 형태**로 반환한다.

ArithmeticException 클래스

```
try
{
	System.out.println("나눗셈 결과의 몫: "+(num1/num2));
	System.out.println("나눗셈 결과의 나머지: "+(num1%num2));
}
catch(ArithmeticException e)
{
	System.out.println("나눗셈 불가능");
	System.out.println(e.getMessage());
}
```

```
두 개의 정수 입력 : 7 0
나눗셈 불가능
by/ zero
프로그램이 종료합니다.
```

예외 클래스

- 배열의 접근에 잘못된 인덱스 값을 사용하는 예외상황

  → 예외 클래스 : ArrayIndexOutofBoundsException
- 허용할 수 없는 형변환 연산을 진행하는 예외상황

  → 예외 클래스 : ClassCastException
- 배열선언 과정에서 배열의 크기를 음수로 지정하는 예외상황

  → 예외클래스 : NegativeArraySizeException
- 참조변수가 null로 초기화 된 상황에서 메소드를 호출하는 예외상황

  → 예외 클래스 : NullPointException

모든 경우에 있어서 예외로 인정되는 상황을 표현하기 위한 예외클래스는 대부분 정의가 되어있다. 그리고 프로그램에 따라서 별도로 표현해야 하는 예외 상황에서는 예외 클래스를 직접 정의 하면 된다

예외 클래스

```
class RuntimeExceptionCase
{
        public static void main(String[] args)
        {
                try {
                        int[] arr=new int[3];
                        arr[-1]=20;
                }
                catch(ArrayIndexOutOfBoundsException e)
                {
                        System.out.println(e.getMessage());
                }

                try {
                        Object obj=new int[10];
                        String str=(String)obj;
                }
                catch(ClassCastException e)
                {
                        System.out.println(e.getMessage());
                }
```

예외 클래스

```
                try
                {
                        int[] arr=new int[-10];
                }
                catch(NegativeArraySizeException e)
                {
                        System.out.println(e.getMessage());
                }

                try
                {
                        String str=null;
                        int len=str.length();
                }
                catch(NullPointerException e)
                {
                        System.out.println(e.getMessage());
                }
        }
}
```

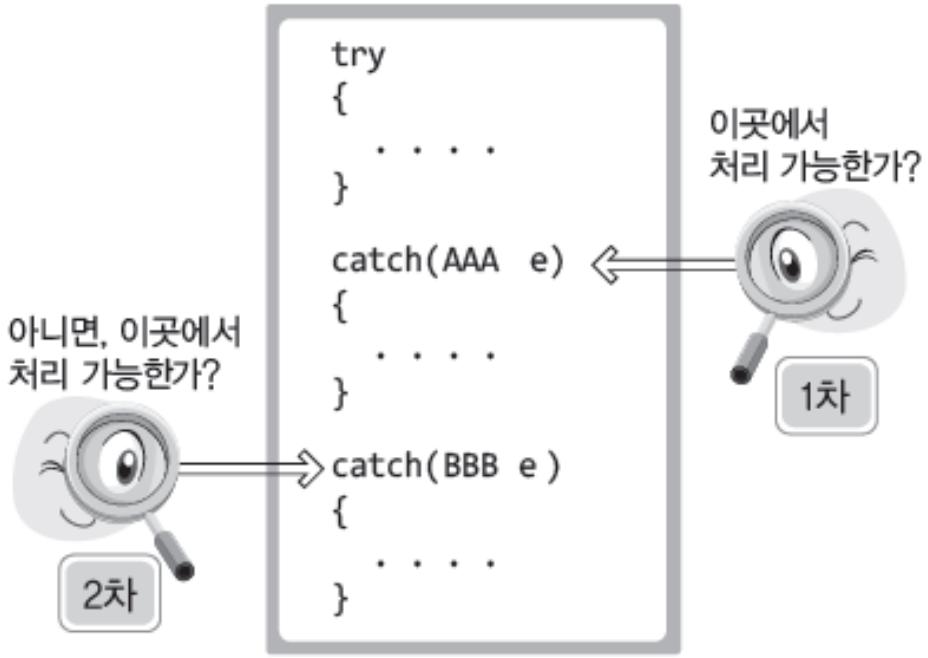
try~catch의 또 다른 장점

하나의 try 블록에 둘 이상의 catch 블록을 구성할 수 있기 때문에 예외처리와 관련된 부분을 완전히 별도로 떼어 놓을 수 있다!

try~catch의 또 다른 장점

```
try
{
        System.out.print("피제수 입력: ");
        int num1=keyboard.nextInt();
        System.out.print("제수 입력: ");
        int num2=keyboard.nextInt();
        System.out.print("연산결과를 저장할 배열의 인덱스 입력: ");
        int idx=keyboard.nextInt();
        arr[idx]=num1/num2;
        System.out.println("나눗셈 결과는 "+arr[idx]);
        System.out.println("저장된 위치의 인덱스는 "+idx);
}
catch(ArithmeticException e)
{
        …
}
catch(ArrayIndexOutOfBoundsException e)
{
        …
}
```

Catch가 결정되는 방법



첫 번째 catch 블록에서부터 순서대로 찾아 내려온다.

Catch 블록의 매개변수가 해당 예외 인스턴스의 참조 값을 받을 수 있는지 확인해 내려온다.

Catch가 결정되는 방법

```
try
{
    . . . .
}
catch(Throwable e)
{
    . . . .
}
catch(ArithmeticException e)
{
    . . . .
}
```

따라서 이렇게 catch문을 구성하면 에러발생!

어떠한 상황에서도 Throwable 클래스를 상속하는 ArithmeticException의 catch 블록이 실행되지 않으므로...

항상 실행되는 finally

finally블럭

- 예외의 발생여부와 관계없이 실행되어야 하는 코드를 넣는다.
- 선택적으로 사용할 수 있으며, try-catch-finally의 순서로 구성된다.
- 예외 발생시, try → catch → finally의 순서로 실행되고
  예외 미발생시, try → finally의 순서로 실행된다.

- try 또는 catch블럭에서 return문을 만나도 finally블럭은 수행된다.

```
try {
      // 예외가 발생할 가능성이 있는 문장들을 넣는다.
} catch (Exception1 e1) {
      // 예외처리를 위한 문장을 적는다.
} finally {
      // 예외의 발생여부에 관계없이 항상 수행되어야하는 문장들을 넣는다.
      // finally블럭은 try-catch문의 맨 마지막에 위치해야한다.
}
```

항상 실행되는 finally

- 그냥 무조건, 항상 실행되는 것이 아니라, finally와 연결되어 있는 try 블록으로 일단
  진입을 하면, 무조건 실행되는 영역이 바로 finally 블록이다.
- 중간에 return 문을 실행 하더라도 finally 블록이 실행된 다음에 메소드를 빠져 나간다!

항상 실행되는 finally

```
try             // try 블록 내에 진입하면
{
        int result=num1/num2;
        System.out.println("나눗셈 결과는 "+result);
        return true;
}
catch(ArithmeticException e)
{
        System.out.println(e.getMessage());
        return false;
}
finally         //finally 블록은 무조건 실행된다!
{
        System.out.println("finally 영역 실행");
}
```

프로그래머가 직접 정의하는 예외의 상황

예외 클래스의 정의와 throw

- 나이를 입력 하라고 했더니, -20살을 입력했다.
- 이름 정보를 입력 하라고 했더니, 나이 정보를 입력했다.

이와 같은 상황은 프로그램의 논리적 예외 상황이다!

즉, 프로그램의 성격에 따라 결정이 되는 예외 상황이다!

따라서 이러한 경우에는 예외 클래스를 직접 정의 해야 하고,

예외의 발생도 직접 명시 해야 한다.

1. 먼저, 연산자 new를 이용해서 발생시키려는 예외 클래스의 객체를 만든 다음

```
Exception e = new Exception("고의로 발생시켰음");
```

2. 키워드 throw를 이용해서 예외를 발생시킨다.

```
throw e;
```

예외 클래스의 정의와 throw

```
class ExceptionEx
{
	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.");
	}
}
```

예외 클래스의 정의와 throw

//사용자 정의 Exception 클래스

```
class AgeInputException extends Exception
// 예외클래스는 Throwable의 하위클래스인 Exception 클래스를 상속해서 정의한다.
{
        public AgeInputException()
        {
                super("유효하지 않은 나이가 입력되었습니다.");
//Exception 클래스의 생성자로 전달되는 문자열이 getMessage 메소드 호출 시 반환되는 문자열이다!
        }
}
```

예외 클래스의 정의와 throw

```
import java.util.Scanner;

class AgeInputException extends Exception
{
        public AgeInputException()
        {
                super("유효하지 않은 나이가 입력되었습니다.");
        }
}
class ProgrammerDefineException
{
        public static void main(String[] args)
        {
                System.out.print("나이를 입력하세요: ");

                try
                {
                        int age=readAge();
                            // throws에 의해 이동된 예외처리 포인트
                        System.out.println("당신은 "+age+"세입니다.");
                }
```

예외 클래스의 정의와 throw

```
                catch(AgeInputException e)
                {
                        System.out.println(e.getMessage());
                }
        }

        public static int readAge() throws AgeInputException
                                        // AgeInputException 예외는 던져 버린다.
        {
                Scanner keyboard=new Scanner(System.in);
                int age=keyboard.nextInt();
                if(age<0)
                {
                        AgeInputException excpt=new AgeInputException();
                        throw excpt; //예외상황을 알리는 문장!!!
                                        // 예외처리 메커니즘 작동!!!
                }

                return age;
        }
}
```

예외를 처리하지 않으면?

나이를 입력하세요: -20
Exception in thread "main" test.AgeInputException: 유효하지 않은 나이가 입력되었습니다.
at test.ThrowsFromMain.readAge(ThrowsFromMain.java:28)
at test.ThrowsFromMain.main(ThrowsFromMain.java:18)

- **가상머신의 예외처리 1　　getMessage 메소드를 호출한다.**
- **가상머신의 예외처리 1　　예외상황이 발생해서 전달되는 과정을 출력해 준다.**
- **가상머신의 예외처리 1　　프로그램을 종료 한다.**

예외 클래스의 정의와 throw

```
class ExceptionEx {
        public static void main(String[] args) throws Exception {
                method1();
        }       // main메서드의 끝

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

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

예외 클래스의 정의와 throw

```
class ExceptionEx {
	public static void main(String[] args) {
		method1();
	}	// main메서드의 끝

	static void method1() {
		try {
			throw new Exception();
		} catch (Exception e) {
			System.out.println("method1메서드에서 예외가 처리
되었습니다.");
			e.printStackTrace();
		}
	}	// method1의 끝
}
```

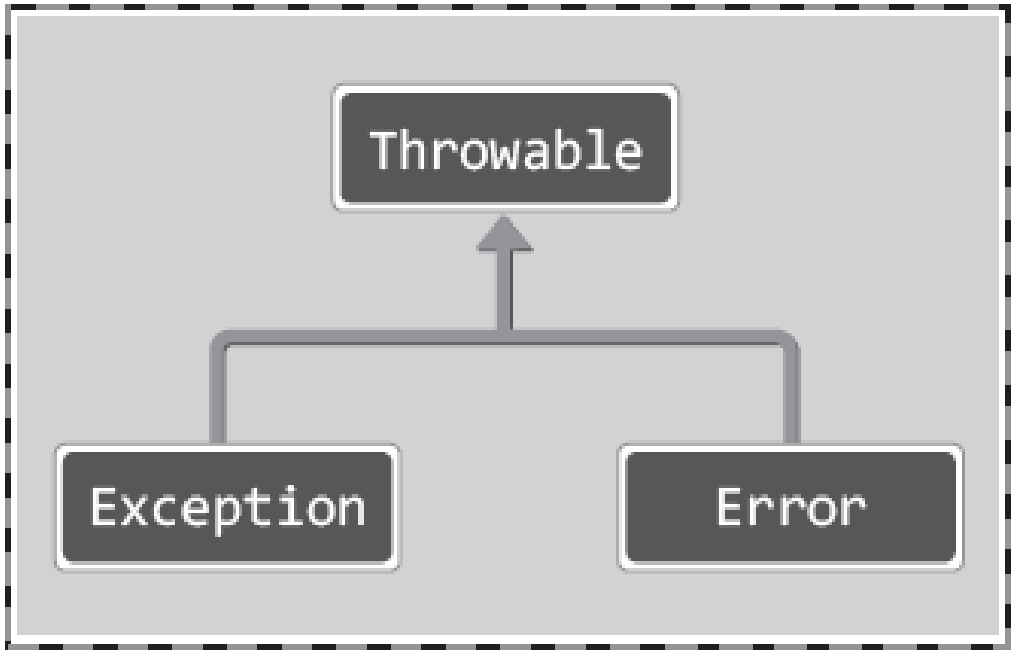
예외 클래스의 정의와 throw

```
class ExceptionEx {
	public static void main(String[] args) {
		try  {
				method1();
		} catch (Exception e)     {
				System.out.println("main메서드에서 예외가
처리되었습니다.");
				e.printStackTrace();
		}
	}	// main메서드의 끝

	static void method1() throws Exception {
		throw new Exception();
	}	// method1()의 끝
} // class의 끝
```

예외 클래스의 계층도

예외 클래스의 계층도

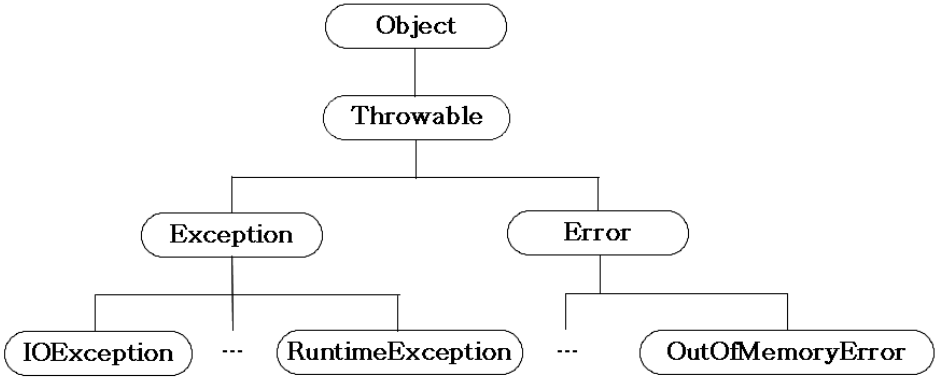


우리의 관심사는 Error 클래스가 아닌, Exception 클래스에 두어야 한다!

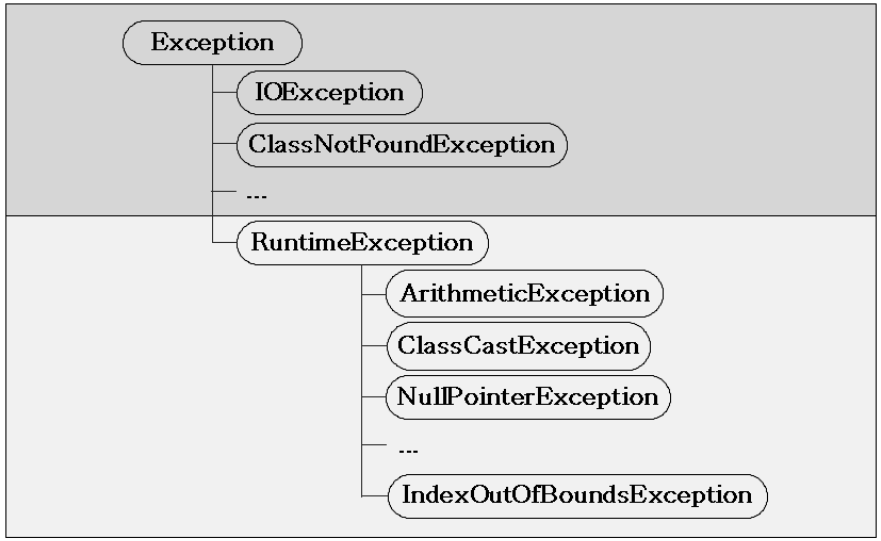
예외 클래스의 계층구조

- 예외 클래스는 크게 두 그룹으로 나뉜다.

RuntimeException클래스들 - 프로그래머의 실수로 발생하는 예외 ← 예외처리 필수
Exception클래스들 - 사용자의 실수와 같은 외적인 요인에 의해 발생하는 예외 ← 예외처리 선택



[그림8-1] 예외클래스 계층도



[그림8-2] Exception클래스와 RuntimeException클래스 중심의 상속계층도

Exception과 API 문서

호출하고자 하는 메소드가 예외를 발생시킬 수 있다면, 다음 두 가지 중 한 가지 조치를 반드시 취해야 하므로, API 문서의 참조가 필요하다.

- **try~catch문을 통한 예외의 처리**
- **throws를 이용한 예외의 전달**

clone (Object 클래스의 인스턴트 메소드)

protected Object clone()
throws CloneNotSupportedException

Creates and returns a copy of this object.
The precise meaning of "copy" may depend on the class of the object.

처리하지 않아도 되는 RuntimeException

- Exception 클래스의 하위클래스이다.
- Error를 상속하는 예외 클래스만큼 치명적인 예외 상황의 표현에 사용되지 않는다.
- 때문에 try~catch문을 통해서 처리하기도 한다.
- 그러나 Error 클래스를 상속하는 예외 클래스와 마찬가지로, try~catch문 또는 throws 절을 명시하지 않아도 된다.
- 이들이 명시하는 예외의 상황은 프로그램의 종료로 이어지는 것이 자연스러운 경우가 대부분이기 때문이다.

처리하지 않아도 되는 RuntimeException

RuntimeException을 상속하는 대표적인 예외 클래스

- ArrayIndexOutOfBoundsException
- classCastException
- NegativeArraySizeException
- NullPointerException
- NumberFormatExcpeion
- ClassNotFoundException

이들은 try~catch문, 또는 throws절을 반드시 필요로 하지 않기 때문에 지금까지 예외 처리 없이 예제를 작성할 수 있었다.

ClassNotFoundException	클래스를 찾지 못함
CloneNotSupportedException	Cloneable 인터페이스 미구현
IllegalAccessException	클래스 접근을 못함
InstantiationException	추상 클래스 또는 인터페이스를 인스턴스화 하고자 할때
InterruptedException	쓰레드가 중단 되었을때
NoSuchFieldException	지정된 필드가 없을때
NoSuchMethodException	지정된 메소드가 없을때
[IOException] CharConversionException	문자 변환에서 예외가 발생했을때
[IOException] EOFException	파일의 끝에 도달했을때
[IOException] FileNotFoundException	파일이 발견되지 않았을때
[IOException] InterruptedIOException	입출력 처리가 중단 되었을때
[IOException][ObjectStreamException] InvalidClassException	시리얼라이즈 처리에 관한 문제가 클래스 안에 있을때
[IOException][ObjectStreamException] InvalidObjectException	시리얼라이즈된 오브젝트에서 입력 검증에 실패했을때
[IOException][ObjectStreamException] NotActiveException	스트림 환경이 액티브하지 않을 때 메소드를 호출했을때
[IOException][ObjectStreamException] NotSerializableException	오브젝트를 시리얼라이즈 할 수 없을때
[IOException][ObjectStreamException] OptionalDataException	오브젝트를 읽을때 기대 이외의 데이터와 만났을때
[IOException][ObjectStreamException] StreamCorruptedException	읽은 데이터 스트림이 파손되어 있을때
[IOException][ObjectStreamException] WriteAbortedException	기록중에 예외가 발생한 스트림을 읽었을때
[IOException] SyncFailedException	시스템 버퍼를 동기시키는 FileDescriptor.sync()의 호출 실패시
[IOException] UnsupportedEncodingException	지정된 문자 부호화 형식을 지원하고 있지 않을때
[IOException] UTFDataFormatException	부정한 UTF-8방식 문자열과 만났을때
[RuntimeException] ArithmeticException	제로제산 등의 산술 예외 발생시
[RuntimeException] ArrayStoreException	배열에 부정한 형태의 오브젝트를 저장하고자 할때
[RuntimeException] [IllegalArgumentException] IllegalThreadStateException	쓰레드가 요구된 처리를 하기에 적합한 상태에 있지 않을때
[RuntimeException] [IllegalArgumentException] NumberFormatException	부적절한 문자열을 수치로 변환하고자 할때
[RuntimeException] IllegalMonitorStateException	모니터 상태가 부정일때
[RuntimeException] IllegalStateException	메소드가 요구된 처리를 하기에 적합한 상태에 있지 않을때
[RuntimeException] [IndexOutOfBoundException] ArrayIndexOutOfBoundsException	범위 밖의 배열 첨자 지정시
[RuntimeException] [IndexOutOfBoundException] StringIndexOutOfBoundsException	범위 밖의 String 첨자 지정시
[RuntimeException] NegativeArraySizeException	음의 크기로 배열 크기를 지정하였을때
[RuntimeException] NullPointerException	null 오브젝트로 접근했을때
[RuntimeException] SecurityException	보안 위반시
[RuntimeException] UnsupportedOperationException	지원되지 않는 메소드를 호출했을때

@ Error	
[LinkageError] ClassCircularityError	클래스 초기화중에 순환 참조를 검출시
[LinkageError] [ClassFormatError] UnsupportedClassVersionError	JVM이 지원되지 않는 버전의 번호를 가진 클래스 파일을 읽고자 할때
[LinkageError] ExceptionInInitializerError	정적 이니셜라이저로 예외가 발생시
[LinkageError] [IncompatibleClassChangeError] AbstracMethodError	추상 메소드를 호출했을때
[LinkageError] [IncompatibleClassChangeError] IllegalAccessError	접근할 수 없는 메소드와 필드를 사용하고자 했을때
[LinkageError] [IncompatibleClassChangeError] InstantiationError	인터페이스 또는 추상 클래스를 인스턴스화하고자 했을때
[LinkageError] [IncompatibleClassChangeError] NoSuchFieldError	지정한 필드가 존재하지 않을때
[LinkageError] [IncompatibleClassChangeError] NoSuchMethodError	지정한 메소드가 존재하지 않을때
[LinkageError] NoClassDefFoundError	클래스 정의가 발견되지 않았을때
[LinkageError] UnsatisfieldLinkError	클래스에 포함되는 링크 정보를 해결할 수 없을때
[LinkageError] VerifyError	클래스 파일안에 부적절한 부분이 있을때
ThreadDeath	쓰레드가 정지해야만 한다는 의미
[VirtualMachineError] InternalError	내부에러
[VirtualMachineError] OutOfMemoryError	메모리부족으로 메모리를 확보 못함
[VirtualMachineError] StackOverflowError	스택 오버 발생
[VirtualMachineError] UnknownError	심각한 예외발생