➢ 학습개요

프로그램은 실행하는 동안 어떤 원인에 의해서든 오작동을 하거나 비정상적으로 종료되는 경우도 있다. 일반적으로 실행 오류라고 하며 Java 에서는 실행 오류를 “에러(error)” 와 “예외(exception)” 으로 나누고 있다. 이 중에서 예외에 대해서는 프로그램을 개발할 때 미리 예측하고 대비하는 코드를 작성할 수 있으며 이 때 사용되는 구문이 바로 예외 처리 구문이다.

➢ 학습목표

- “에러(error)” 와 “예외(exception)”의 차이점 그리고 Java의 예외처리 메카니즘을 파악한다.
- Exception과 RuntimeException의 차이점을 이해한다.
- try-catch-finally 구문의 구현 방법과 처리 과정을 파악한다.
- 예외 선언 구문인 throws 절의 기능을 파악하고 활용 방법을 익힌다.
- 예외의 발생 구문과 예외 클래스의 생성 방법을 익힌다.

11. 예외처리

프로그램 실행 중에 예외가 발생하여 프로그램이 비정상적으로 종료되는 것을 막고 예외에 대비한 적절한 코드를 소스상에 작성하는 예외처리 구문에는 예외를 잡는 구문 try-catch-finally, 예외를 발생시키는 구문 throw, 예외를 선언하는 구문 throws 그리고 Java SE 1.4 에서 추가된 assert 구문 등이 사용되고 있다.

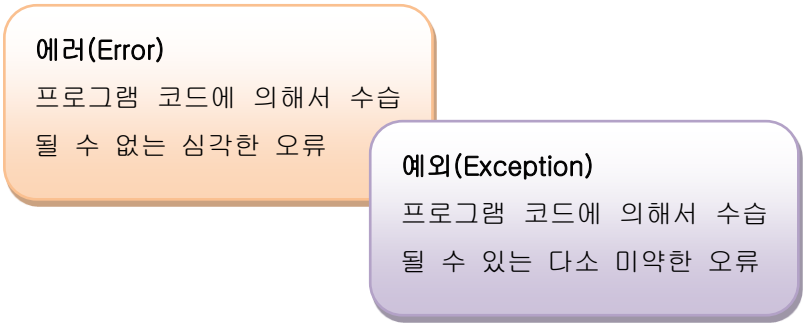
Java 프로그램의 실행 오류

프로그램 실행 중에 발생할 수 있는 실행 오류는 “에러(error)”와 “예외(exception)”로 나뉘며 이 중에서 Java 프로그램 구현 시 대비할 수 있는 실행 오류는 바로 예외이다. 예외도 발생 원인에 따라서 Exception과 RuntimeException으로 나뉘며 어떠한 예외냐에 따라 예외 처리 구문이 다르게 적용된다.

■ “에러(error)” 와 “예외(exception)”

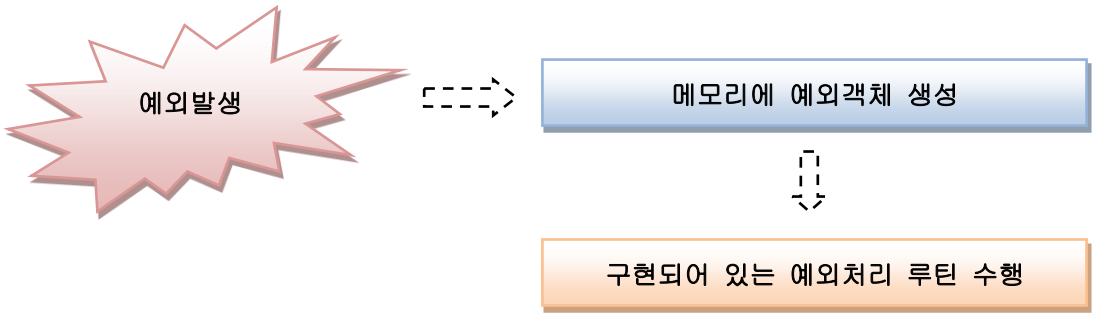
Java에서는 실행 시(runtime) 발생할 수 있는 프로그램 오류를 '에러(Error)'와 '예외(Exception)', 두 가지로 구분하였다. 에러는 메모리 부족(OutOfMemoryError)이나 스택 오버플로우(StackOverflowError)와 같이 일단 발생하면 코드에 의해서 수습될 수 없는 심각한 오류이고, 예외는 프로그램 코드에 의해서 수습될 수 있는 다소 미약한 오류이다.

에러가 발생하면, 프로그램의 비정상적인 종료를 막을 길이 없지만, 예외는 발생하더라도 프로그래머가 이에 대한 적절한 코드를 미리 작성해 놓음으로써 프로그램의 비정상적인 종료를 막을 수 있다. 발생 가능한 예외에 대비한 코드를 작성하는 것을 예외처리라고 한다.



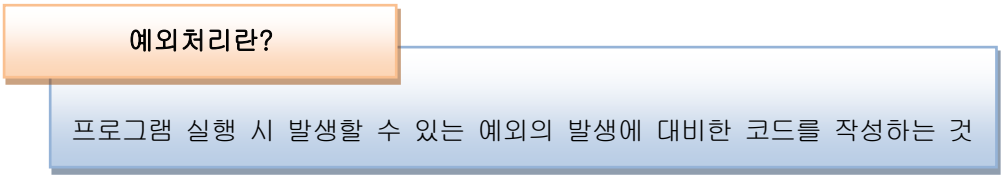
■ Java의 예외 메커니즘

프로그램 실행 도중에 예외가 발생한다는 것은 메모리 상에 예외 클래스의 객체가 생성되고 현재의 수행 흐름을 중단한 후에 예외처리의 구현 내용대로 그 다음 수행 위치로 분기가 일어나 수행을 하게 된다.



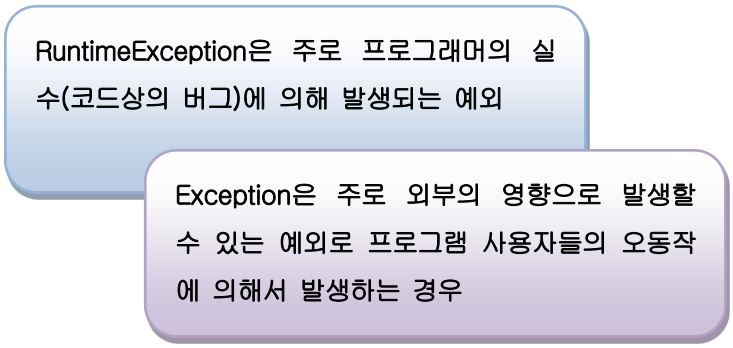
프로그램의 실행도중에 발생하는 에러는 어쩔 수 없더라도 예외의 경우에는 프로그래머가 이에 대한 처리를 프로그램 코드상에 미리 구현하는 것이 좋으며 Java에서는 예외처리를 하도록 구문적으로 요구하고 있다.

예외처리(Exception Handling)란, 프로그램 실행 시 발생할 수 있는 예기치 못한 예외의 발생에 대비한 코드를 작성하는 것이며, 예외처리의 목적은 예외의 발생으로 인한 실행 중인 프로그램의 갑작스런 비정상 종료를 막고, 정상적인 실행상태를 유지할 수 있도록 하거나 원하는 오류메시지를 출력하고 정상 종료를 하도록 하기 위해서이다.



■ Exception과 RuntimeException

예외 클래스들은 Exception과 RuntimeException으로 나누며 Exception은 Checked Exception이고 RuntimeException은 Unchecked Exception이다. Exception과 RuntimeException은 기능적으로 다음과 같은 차이가 있다.



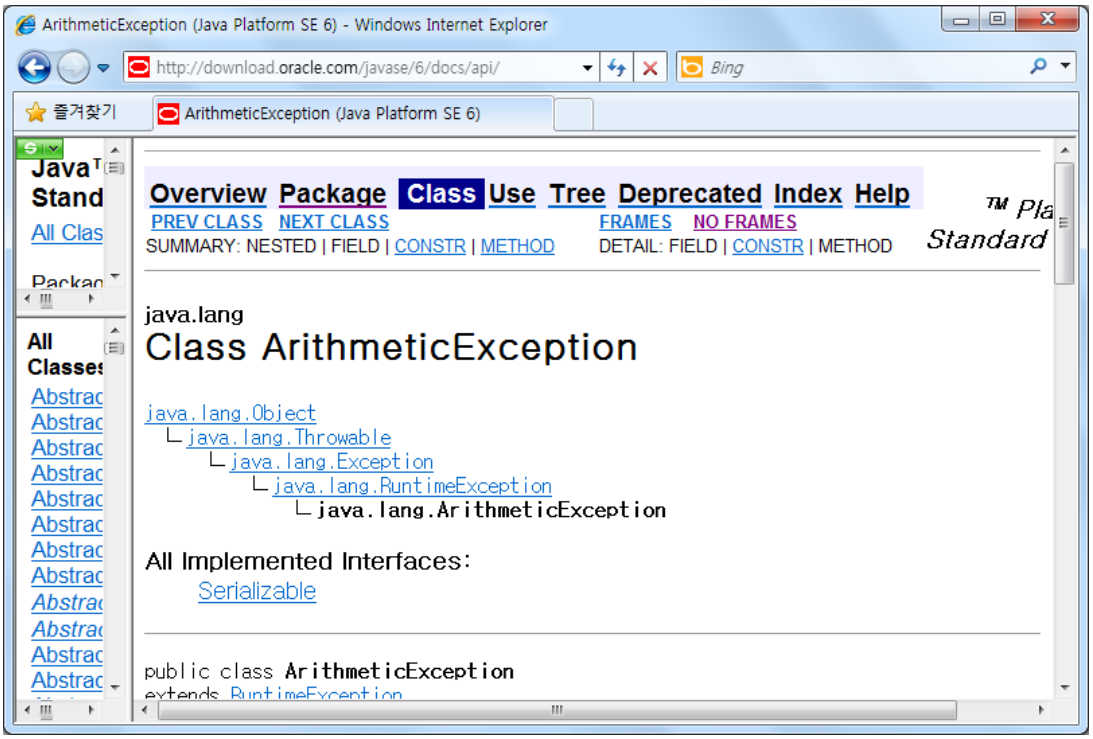
[RuntimeException의 예]
- 배열의 범위를 벗어나는 인덱스 문제 – ArrayIndexOutOfBoundsException
- null 값을 가지고 있는 참조 변수로 멤버를 접근하려고 할 때 – NullPointerException
- 참조 형간에 잘못된 형 변환을 할 때 –ClassCastException
- 숫자를 0으로 나누려 할 때 – ArithmeticException

[Exception의 예]

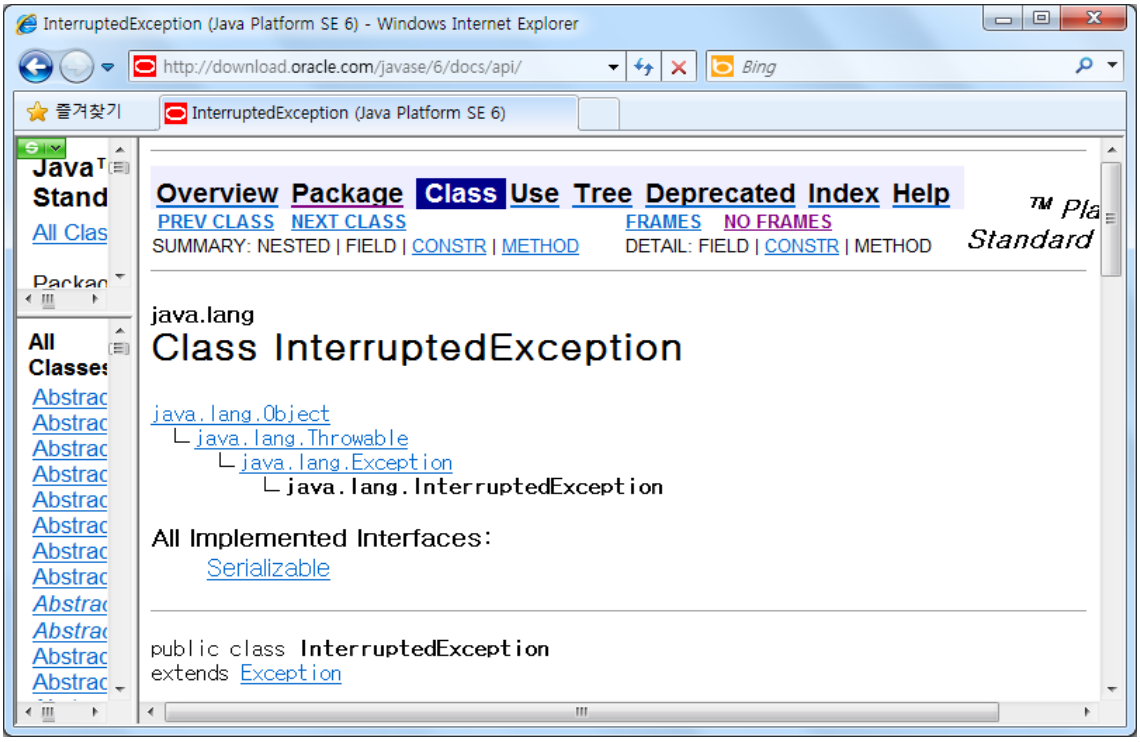
- 존재하지 않는 파일을 읽기모드로 오픈하려고 할 때 – FileNotFoundException
- 실행시키고자 하는 클래스의 이름을 잘못 입력했을 때 – ClassNotFoundException
- 지원되지 않는 문자 인코딩을 사용했을 때 – UnsupportedEncodingException

기능적으로 구분하는 것 외에도 또 다른 구분 방법으로는 해당 예외 클래스의 조상 클래스 중에 RuntimeException 클래스가 있으면 RuntimeException 계열이 되는 것이며 없으면 Exception 계열이 된다.

[RuntimeException 계열의 예외 클래스의 예]



[Exception 계열의 예외 클래스의 예]



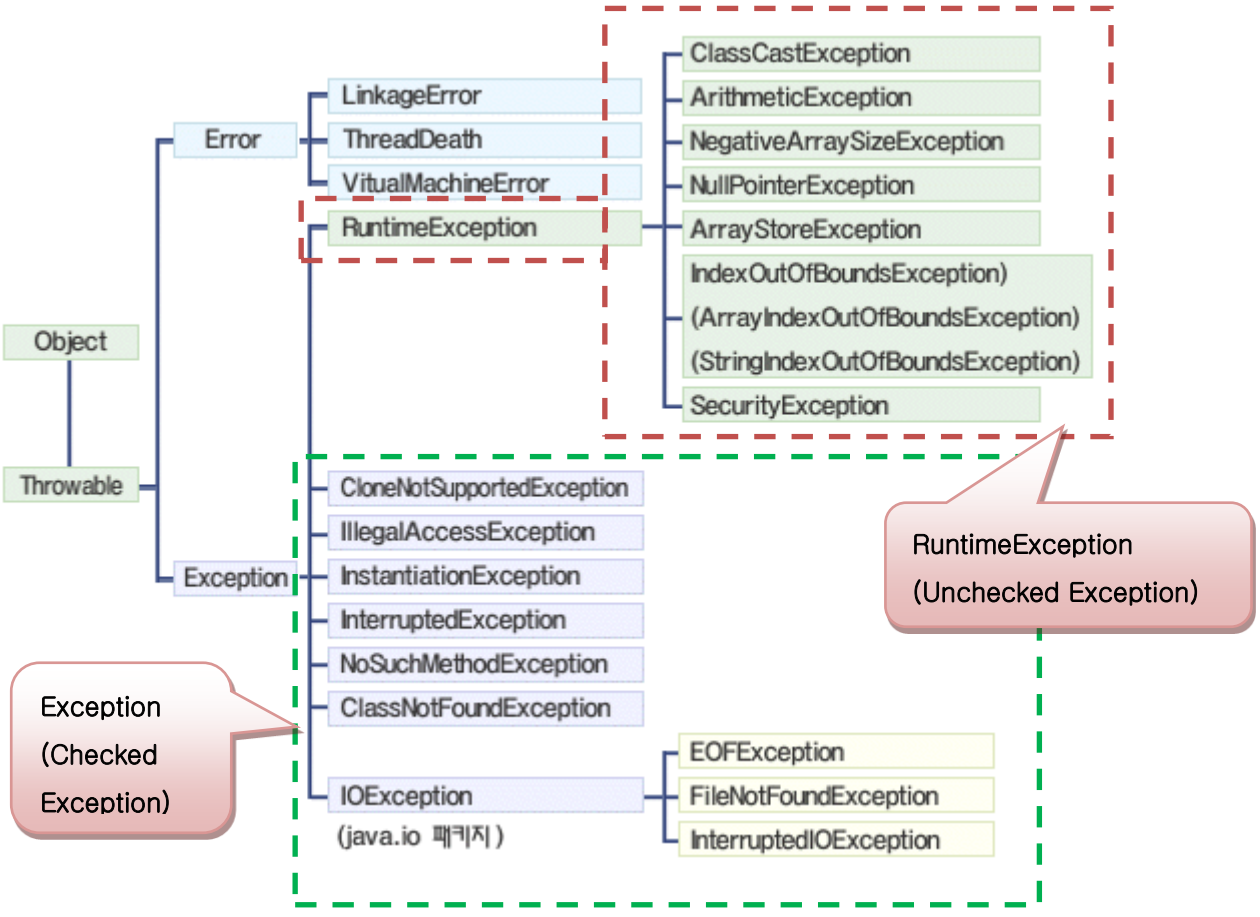
▶ CheckedException과 UncheckedException

CheckedException은 컴파일 시 예외처리를 구현했는지 점검하는 예외로 try-catch-finally 나 throws 구문을 이용하여 예외처리를 구현해야만 하는 예외들이며 UncheckedException 은 컴파일 시 예외처리를 구현했는지 점검하지 않는 예외로 예외 처리를 구현하지 않으면 인터프리터(JVM)이 예외 처리를 대신 하게 되는 예외들이다.

Exception에 속하는 예외들은 CheckedException이며 RuntimeException에 속하는 예외들은 UncheckedException이다.

▶ 예외와 에러 관련 클래스들

다음 그림은 예외와 에러 관련 대표적인 클래스들로서 RuntimeException의 자손 클래스들이 모두 RuntimeException 클래스에 속하며 이외의 Exception 클래스들은 모두 Exception 클래스에 속한다.



예외의 처리와 발생

예외 처리 구문이란 발생할 수 있는 예외에 대하여 대비하는 코드를 작성할 수 있는 구문으로서 try-catch-finally와 throws중에서 선택하여 적용할 수 있다. 예외가 발생하였을 때 어떠한 처리가 되기를 원하는가에 따라 선택 가능하다. 예외 발생 구문으로는 throw구문이 있으며 Java의 표준 API에도 다양한 예외 클래스가 제공되고 있지만 직접 예외 클래스를 만들어서도 활용할 수도 있다.

■ try-catch-finally

예외를 처리하기 위해서는 try-catch-finally문을 사용하며 구조는 다음과 같이 다양하게 작성할 수 있다. 하나의 try블록에는 catch블록과 finally블록이 선택적으로 정의될 수 있으며 catch 블록의 경우에는 처리할 예외가 여러 개인 경우 2개 이상 정의되는 것도 가능하다.

```
try {
        // 예외가 발생할 가능성이 있는 문장들
} catch (Exception1 e1) {
        // Exception1이 발생한 경우,
        //Exception1을 처리하기 위한 문장
}
next행
```

catch블록을 수행하는 동안
Exception1 예외 발생시 - catch 블록 수행한 후 next행 수행으로 넘어간다.
Exception1 예외 미 발생시 - 바로 next행 수행으로 넘어간다.

```
try {
        // 예외가 발생할 가능성이 있는 문장들
} catch (Exception1 e1) {
        // Exception1이 발생한 경우,
        // Exception1을 처리하기 위한 문장
} catch (Exception2 e2) {
        // Exception2가 발생했을 경우,
        // Exception2를 처리하기 위한 문장
}
next행
```

catch블록을 수행하는 동안
Exception1 예외 발생시 - 첫 번째 catch블록 수행한 후 next행 수행으로 넘어간다.
Exception2 예외 발생시 - 두 번째 catch블록 수행한 후 next행 수행으로 넘어간다.
예외 미 발생시 - 바로 next행 수행으로 넘어간다.

try블록에는 한 개의 finally블록이 올 수 있으며 예외 발생 여부와 관계없이 마지막에 수행시켜야 하는 문장을 정의하는 블록이다. 생략가능하며 정의하는 경우에는 마지막 위치에 해야 한다.

```
try {
        // 예외가 발생할 가능성이 있는 문장들
} finally {
        // 예외 발생 여부와 관계없이
        // 수행할 문장
}
next행
```

catch블록을 수행하는 동안 예외 발생시 – finally블록을 수행한 후 이 메서드를 호출한 메서드의 구현되어 있는 예외 처리 루틴을 따라 수행한다.
예외 미 발생시 – finally블록을 수행한 후에 바로 next행 수행으로 넘어간다.

하나의 try블록 다음에는 여러 종류의 예외를 처리할 수 있도록 하나 이상의 catch블록이 올 수 있으며, 이 중 발생한 예외의 종류와 일치하는 단 한 개의 catch블록 만이 실행된다.
발생한 예외의 종류와 일치하는 catch블록이 없으면, 이곳에서 예외는 처리되지 못하고 이 메서드를 호출한 곳에서 예외 처리 코드 찾아 구현되어 있는 내용대로 수행한다.

```
try {
        // 예외가 발생할 가능성이 있는 문장들
} catch (Exception1 e1) {
        // Exception1이 발생했을 경우, Exception1을 처리하기 위한 문장
} catch (Exception2 e2) {
        // Exception2가 발생했을 경우, Exception2를 처리하기 위한 문장
...
} catch (ExceptionN eN) {
        // ExceptionN이 발생했을 경우, ExceptionN을 처리하기 위한 문장
}finally {
        // 예외 발생 여부와 관계없이 수행할 문장
}
```

try블록, catch블록 그리고 finally블록은 수행 문장이 하나라 하더라도 블록으로 구성해야 하며

try-catch-fanilly를 중첩되게 구성하는 것도 가능하다.

▶ catch 블록

catch블록에는 괄호를 사용하여 변수를 하나 선언하고 있어야 하며 이 변수를 처리하고자 하는 예외 클래스 타입이거나 처리하고자 하는 예외 클래스의 조상 클래스 타입이어야 한다.

catch블록에 선언되는 변수의 기능은 메모리상에 생성되는 예외객체에 대한 참조값을 전달받아 발생되는 예외에 대한 정보 즉, 오류 메시지, 예외가 발생되기 까지의 호출 흐름 등을 추출하는 용도이다.

참조하게 되는 예외객체에 대하여 자주 사용되는 메서드는 getMessage()와 printStackTrace()이다. getMessage()는 예외객체가 생성될 때 초기화된 메시지를 추출하는 기능이고 printStackTrace()는 예외가 발생하기까지의 호출 흐름을 출력하는 기능이다.

printStackTrace() - 예외 발생 당시의 호출스택(Call Stack)에 있었던 메서드의 정보와 예외 메세지를 화면에 출력한다.
getMessage() - 발생한 예외클래스의 객체에 저장된 예외메세지를 얻을 수 있다.

```
try {

} catch (예외클래스타입  변수) {

}
```

```
try {

} catch (IOException  e) {
    Systrm.out.println("메시지" + e.getMessage());
    e.printStackTrace();
}
```

catch블록에 선언되는 변수는 생성되는 예외 클래스의 객체를 전달받는 역할이므로 처리하려는 예외 클래스 타입 외에도 처리하려는 예외클래스의 조상 타입으로 변수를 선언해도 된다. 이때에도 다형성이 적용될 수 있기 때문이다.

예외 클래스들로 예외 클래스들간에 상속 관계가 적용되어 있으므로 어떠한 예외 타입의 변수

로 선언한 경우인가에 따라서 그 변수로 참조할 수 있는 예외 객체들은 해당 예외 클래스뿐만 아니자 자손 예외 객체들도 포함된다. 이것은 곧 자손 예외도 처리할 수 있다는 것을 뜻한다.

위의 예제의 catch블록에 선언된 e변수는 IOException 객체 뿐만 아니라 EOFException, FileNotFoundException 등을 모두 참조할 수 있으며 이것은 곧 IOException, EOFException, 그리고 FileNotFoundException 등을 모두 처리할 수 있는 catch블록을 선언하는 것을 의미한다.

그러므로 모든 예외 클래스들의 최상위 클래스인 Exception 클래스 타입으로 선언한 경우에는 모든 예외를 처리할 수 있는 catch 블록을 선언하게 되는 결과가 된다.

다음은 나눗셈 연산을 수행하면서 발생될 수 있는 ArithmeticException에 대한 try-catch구문을 적용하여 예외를 처리하고 있는 부분 소스이다. 나눗셈 연산을 수행할 때 ArithmeticException 이 발생되는 경우는 제수에 해당되는 숫자의 값이 0이 되는 경우이다.

```
try {
    int num1 = Integer.parseInt(s1);
    int num2 = Integer.parseInt(s2);
    result = num1 / num2;
} catch (ArithmeticException e) {
    System.out.println("연산 오류 발생");
    return;
}
```

try블록의 문장들을 수행하는 동안 ArithmeticException이 발생한 경우에 한해서 예외처리를 구현한 예제이다.

```
try {
    int num1 = Integer.parseInt(s1);
    int num2 = Integer.parseInt(s2);
    result = num1 / num2;
} catch (ArithmeticException e) {
    System.out.println("연산 오류 발생");
    return;
} catch (NumberFormatException e) {
    System.out.println("숫자 변환 오류 발생");
    return;
}
```

try블록의 문장들을 수행하는 동안 ArithmeticException이 발생한 경우, NumberFormatException 이 발생한 경우 각각에 대하여 예외처리를 구현한 예제이다.

```
try {
    int num1 = Integer.parseInt(s1);
    int num2 = Integer.parseInt(s2);
    result = num1 / num2;
} catch (ArithmeticException e) {
    System.out.println("연산 오류 발생");
    return;
} catch (NumberFormatException e) {
    System.out.println("숫자 변환 오류 발생");
    return;
} finally {
    System.out.println("finally 블록 수행");
}
```

try블록의 문장들을 수행하는 동안 예외의 발생 여부에 관계없이 마지막에 수행되는 블록인 finally블록을 정의한 예제이다. catch블록에 return문이 있더라도 finally블록은 수행하고 분기가 일어난다.

```
try {
    int num1 = Integer.parseInt(s1);
    int num2 = Integer.parseInt(s2);
    result = num1 / num2;
} catch (Exception e) {
    System.out.println("오류 발생");
    return;
}finally {
    System.out.println("finally 블록 수행");
}
```

catch블록에 선언하는 변수가 어떠한 예외클래스 타입의 변수인가에 따라서 해당 catch블록에 의해 처리하게 되는 예외의 종류가 달라진다. 이 예제처럼 Exception이라는 조상 예외 클래스 타입으로 변수를 선언한 경우에는 어떠한 예외가 발생하든 이 catch블록에서 예외를 처리하게 되는 결과가 된다.

```
try {
    int num1 = Integer.parseInt(s1);
    int num2 = Integer.parseInt(s2);
    result = num1 / num2;
} catch (Exception e) {
    System.out.println("오류 발생");
    return;
} catch (ArithmeticException e) {   // 컴파일 오류
    System.out.println("연산 오류 발생");
    return;
}finally {
    System.out.println("finally 블록 수행");
}
```

이 부분 소스는 컴파일 오류가 발생하게 된다. 이유는 모든 예외를 처리할 수 있는 catch블록에 위에 있고 특정 예외를 처리할 수 있는 catch블록이 아래에 정의되어 있기 때문이다. catch블록은 정의 순서에 의해 처리 코드로 우선 선택되어지기 때문에 이와 같이 구현하게 되면 아래에 있는 catch블록은 무의미한 구현이 되기 때문이다.

▶ try-catch 블록의 처리 흐름

1. try블록 내에서 예외가 발생하지 않은 경우.

try블록 내의 마지막 문장까지 다 실행할 때까지도 예외가 발생하지 않으면, 어떠한 catch블록도 거치지 않고 전체 try-catch문을 빠져나가서 그 다음 문장을 계속해서 실행한다.

2. try블록 내에서 예외가 발생한 경우.

곧바로 try블록을 벗어나 제일 첫 번째 위치한 catch블록부터 차례대로 내려오면서, 발생한 예외와 일치하는 catch블록이 있는지 확인한다. 일치하는 catch블록을 찾게 되면, 그 catch블록 내의 문장들을 실행하고 전체 try-catch문을 빠져나가서 그 다음 문장을 계속해서 실행한다. catch블록의 괄호()내에 선언된 참조변수의 종류와 생성된 예외클래스의 객체에 instanceof 연산자를 이용해서 검사하게 되는데, 검사결과가 true인 catch블록을 만날 때까지 검사는 계속된다. 검사결과가 true인 catch블록을 찾게 되면, 블록에 있는 문장들을 모두 실행한 후에 try-catch문을 빠져나가고 예외는 처리되지만, 검사결과가 true인 catch블록이 하나도 없으면 예외는 처리되지 않는다.

[예외가 발생하지 않을 때]

```
System.out.println(1);
System.out.println(2);
try {
    System.out.println(3);
    System.out.println(4);
} catch (Exception e){
    System.out.println(5);
}
System.out.println(6);
```

1
2
3
4
6
이 출력된다.

[예외가 발생할 때]

```
System.out.println(1);
System.out.println(2);
try {
    System.out.println(3);
    System.out.println(0/0);
    System.out.println(4);
} catch (Exception e){
    System.out.println(5);
}
System.out.println(6);
```

1
2
3
5
6
이 출력된다.

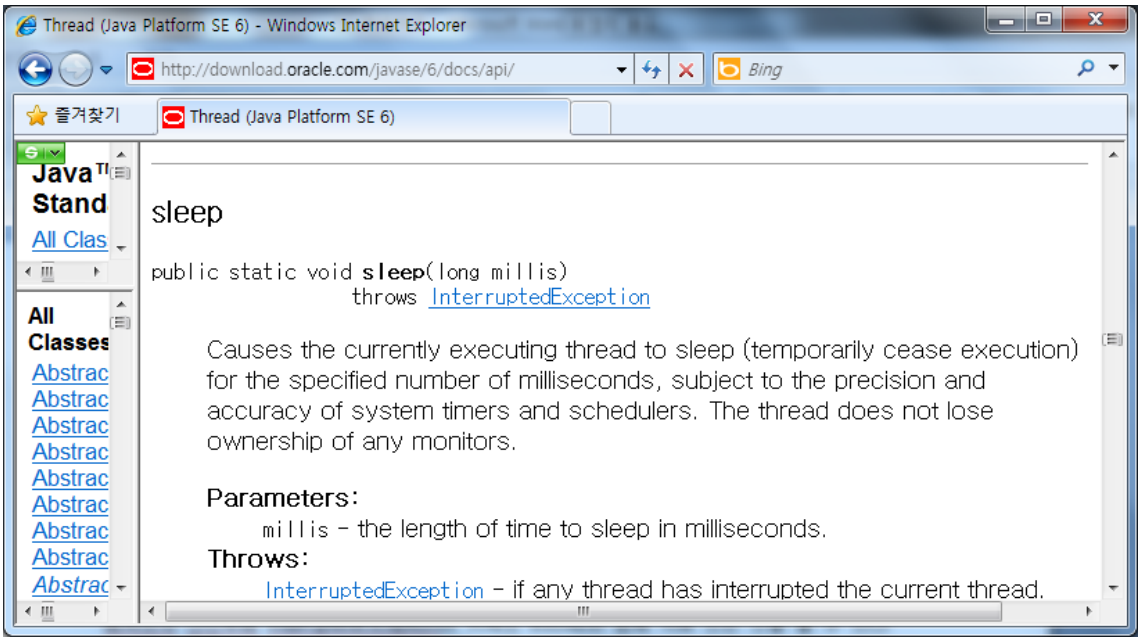
■ throws

예외를 처리하는 방법에는 try-catch-finally문을 사용하는 것 이외에, throws절을 이용하여 예외를 메서드에 선언하는 방법이 있다. 예외를 잡아서 처리하는 것 대신 이 메서드를 호출한 쪽에 예외 발생을 알리고자 할 때 사용되는 예외 처리구문이다.

메서드에 예외를 선언하려면, 메서드의 선언부에 키워드 throws절를 사용해서 메서드 내에서 발생할 수 있는 예외를 적어주면 된다. 그리고, 예외가 여러 개일 경우에는 쉼표(,)로 구분한다.

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

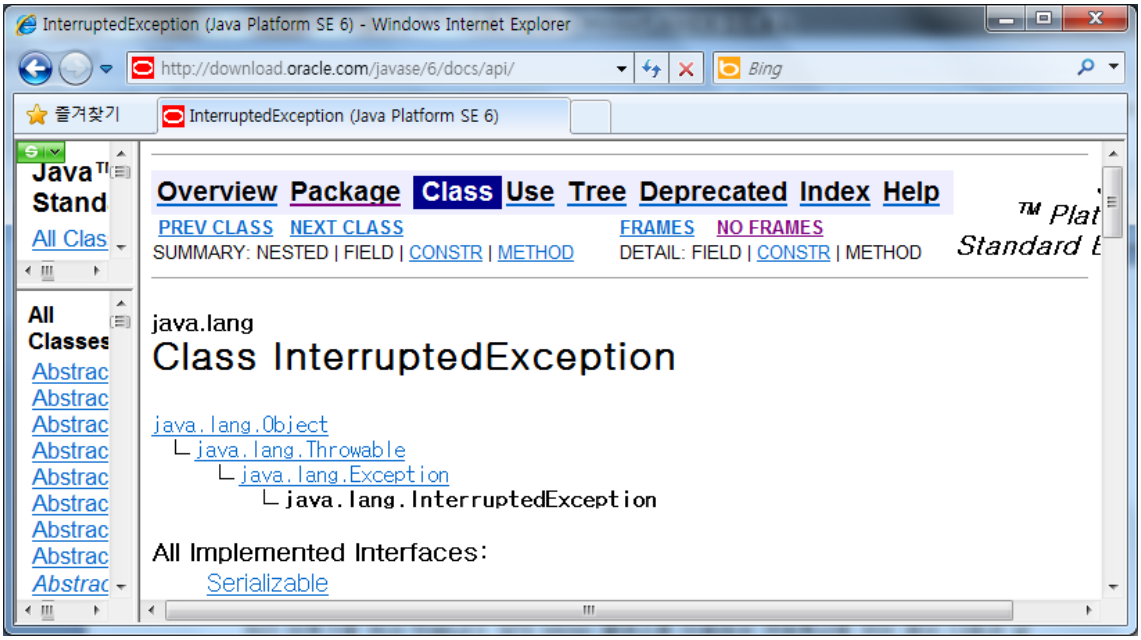
메서드의 선언부에 예외를 선언함으로써, 메서드를 호출하려는 시점에서 이 메서드를 사용하기 위해서는 어떠한 예외들이 처리되어져야 하는지 쉽게 알 수 있다. 메서드를 작성할 때 메서드 내에서 발생할 가능성이 있는 예외를 메서드의 선언부에 명시하여, 이 메서드를 호출하는 쪽에서는 이에 대한 처리를 하도록 하므로 보다 견고한 프로그램 코드를 작성할 수 있도록 도와준다.



위의 그림은 Java API 도큐멘테이션 문서에서 찾아본 java.lang.Thread클래스의 sleep()메서드에 대한 설명이다. 메서드의 선언부에 InterruptedException이 키워드 throws와 함께 적혀 있는 것을 볼 수 있다. 이 것이 의미하는 바는 이 메서드에서는 InterruptedException이 발생할 수 있

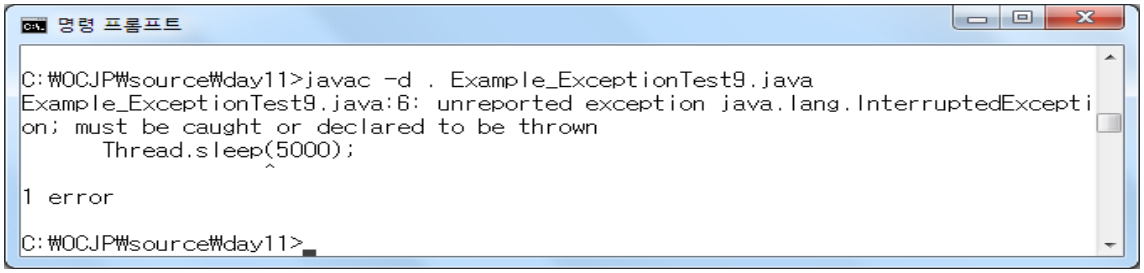
으니, 이 메서드를 호출하고자 하는 메서드에서는 InterruptedException을 처리 해주어야 한다는 것이다.

InterruptedException에 밑줄이 있는 것으로 보아 링크가 걸려 있음을 알 수 있을 것이다. 이 링크를 클릭하면, InterruptedException에 대한 설명을 볼 수 있다.



위의 그림에서 볼 수 있는 것처럼. InterruptedException은 Exception클래스의 자손임을 알 수 있다. 따라서 InterruptedException은 반드시 처리해주어야 하는 예외임을 알 수 있다. 그래서 sleep()메서드의 선언부에 키워드 throws와 함께 선언되어져 있는 것이다.
Java API 도큐멘테이션 문서의 sleep()메서드 설명의 아래쪽에 있는 'Throws:'를 보면, sleep()메서드에서 발생할 수 있는 예외의 리스트와 언제 발생하는가에 대한 설명이 덧붙여져 있다.

이렇게 Exception 계열의 예외가 발생할 수 있는 메서드에서는 try-catch-finally문으로 예외를 해당 메서드 안에서 직접 처리하든지 throws절을 이용하여 메서드 선언부에 예외선언을 하여 이 메서드를 호출하는 쪽에서 예외 처리를 구현할 것을 알려야 한다. 그렇지 않으면 다음과 같이 컴파일시 예외가 처리되지 않았다는 오류 메시지가 발생한다.



만일 프로그램내에서 throws절만으로 예외를 처리한 경우에는 최종적으로 JVM이 예외 처리를 대신하게 되는데 JVM이 수행하게 되는 예외 처리는 프로그램 수행을 종료하고 예외가 발생하기까지의 호출 스택 정보를 화면에 출력한다. 다른 결과를 원한다면 try-catch-finally구문을 사용해서 예외처리를 해야한다.

처리해야할 예외가 여러가지인 경우에는 throws절 뒤에 예외 클래스명을 여러 개 나열해도 되며 조상 예외 클래스 하나를 선언하여 대신 처리해도 된다.

```
void method() throws   FileNotFoundException, InterruptedException{
        // 메서드의 내용
}
```

```
void method() throws Exception{
        // 메서드의 내용
}
```

RuntimException 계열의 예외는 예외처리 구현을 생략한 경우 throws절로 예외를 처리한 결과와 동일하다.

■ **예외 발생**

키워드 throw를 사용해서 프로그래머가 직접 예외를 발생시킬 수 있으며, 방법은 아래의 순서를 따르면 된다.

1. 예외를 발생시키려는 목적에 부합되는 명칭의 예외 클래스를 찾는다.
   없으면 직접 예외 클래스를 생성할 수도 있다.
2. 선택된 예외 클래스를 객체 생성하여 throw 구문 뒤에 지정한다.

```
throw  new  IllegalArgumentException();
```

다음과 같이 변수에 담아서 지정해도 된다.

```
IllegalArgumentException e = new IllegalArgumentException();
throw e;
```

예외를 발생시키는 구문이 정의된 메서드에서는 발생시키는 예외가 RuntimeException계열이라면 관계없지만 Exception계열이라면 반드시 try-catch문으로 예외처리 구문을 같이 구현하든지 아니면 throws절을 이용하여 예외 발생을 알리는 선언문을 메서드 헤더에 지정해주어야 한다.

▶ 사용자정의 예외 만들기

기존의 정의된 예외 클래스 외에 필요에 따라 프로그래머가 새로운 예외 클래스를 정의하여 사용할 수 있다. 보통 Exception클래스로부터 상속받는 클래스를 만들지만, 필요에 따라서 알맞은 예외 클래스를 부모 클래스로 선택할 수 있다.

```
class MyException extends Exception {
    MyException(String msg) {
        super(msg);
    }
}
```