예외(Exception)

프로그램을 실행하다가 보면 어떤 원인때문에 비정상적인 동작을 일으키며 프로그램이 종료되는 상황을 보신적 있으실 겁니다. 이때 우리는 프로그램이 오류가 발생했다고 합니다. 에러의 종류는 우리가 컴파일 할 때 발생할 수 있는 컴파일 오류와 실행 중 발생되는 런타임 오류 두 종류가 있지요. 컴파일 오류는 우리가 잡기가 쉽지만, 런타임 오류는 잡기가 까다롭습니다. 자바에서는 런타임 오류를 두 종류로 보고 있습니다. 바로 에러(Error)와 예외(Exception)으로 말이죠.

에러는 프로그램이 코드로 복구될 수 없는 오류를 의미하고 예외는 프로그래머가 직접 예측하여 막을 수 있는 처리가능한 오류라고 보시면 됩니다. 예를 들어 메모리가 부족한 경우 프로그래머가 직접 제어할 수 없으므로 이런 경우는 메모리 부족(OutOfMemoryError) 에러가 발생하고 함수 호출이 많아 스택이 쌓일 경우에는 StackOverFlowError가 발생할 수 있습니다.

그런데 아래의 코드처럼 어떤 수를 0으로 나눈다면 어떤 상황이 발생할까요?

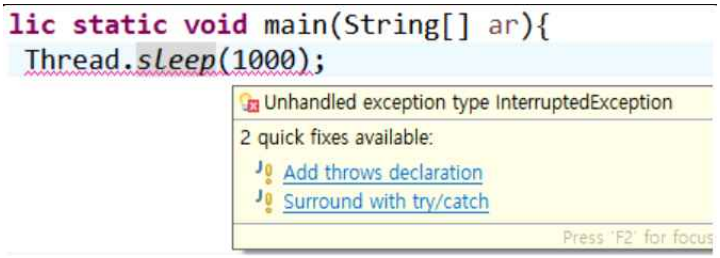
```
int a,b;
a=10;
b=0;

int c=a/b;
System.out.println(c);
```

어떤 수를 0으로 나눌 수는 없기 때문에 오류를 내보내게 됩니다.

```
Exception in thread "main" java.lang.ArithmeticException: / by zero
        at aa.Main.main(Main.java:11)
```

하지만 조건문을 통해서 우리는 0으로 못 나누게 할 수 있죠. 이처럼 우리가 예측 가능한 상황에서 오류를 제어할 수 있는 것이 예외입니다.

예외는 Compile 시에 발견할 수 있는 예외와 프로그램 실행시에 발생하는 예외 두 종류가 있습니다. Compile시에 발생할 수 있는 예외는 아래의 사진과 같이 IntelliJ 또는 Eclipse와 같은 IDE를 쓰신다면 빨간줄로 예외를 처리하라고 욕합니다.



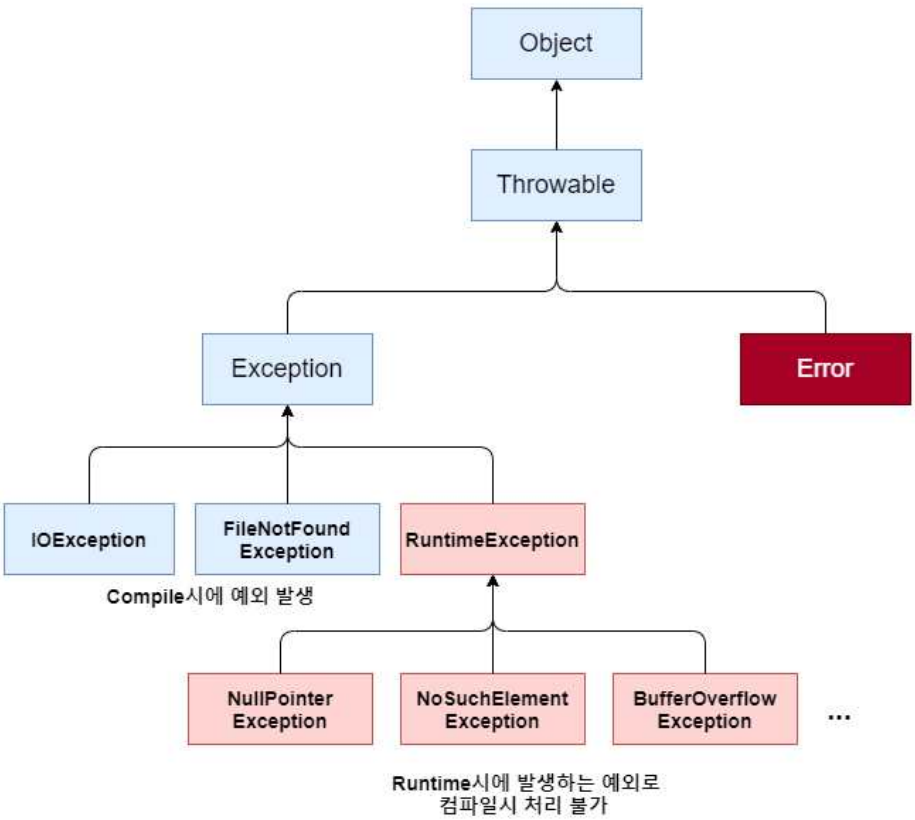
하지만 위에서의 예처럼 Compile시에 발견하지 못하는 에러를 Runtime에러라고 하는데, 이때는 프로그래머가 예측하여 처리해주어야합니다.

그리고 그런 예외가 발생했을때 어떤 동작을 처리해야 하는지를 우리는 예외 처리라고 합니다.

예외 처리

1. try, catch

예외가 발생했을때 우리는 try ... catch ... finally 라는 키워드로 예외를 처리할 수 있거나 메소드를 호출한 곳으로 던질 수 있습니다. 한 가지 중요한 점은 자바에서 모든 예외는 Exception이라는 클래스를 상속받습니다. Exception의 상속 트리를 아래에 간략하게 나타내었습니다.



예외 처리하는 방식은 이렇습니다.

```
try{
        //예외가 발생될만한 코드
}catch(FileNotFoundException e){  //FileNotFoundException이 발생했다면

}catch(IOException e){ //IOException이 발생했다면

}catch(Exception e){  //Exception이 발생했다면

}finally{
        // 어떤 예외가 발생하던 말던 무조건 실행
}
```

try 블록 : 이 블록에서 예외가 발생할만한 코드가 쓰여집니다.

catch (예외 종류) 블록 : 이 부분에서 예외가 발생되었을때 처리하는 동작을 명시합니다. catch 블록은 여러 개가 있을 수 있습니다. 맨 처음 catch 블록에서 잡히지 않는 예외라면 다음 catch의 예외를 검사합니다. 이때 상속관계에 있는 예외 중 부모가 위의 catch, 그리고 그 자식 예외 클래스가 아래의 catch로 놓일 순 없습니다. 예를 들어 아래와 같이말이죠.

```
try{
        //.. 중략 ..//
} catch (Exception e){
        //컴파일 오류 발생
} catch (IOException e){

}
```

Exception 클래스는 모든 예외의 부모이기 때문에 Exception을 IOException보다 위에서 처리할 수는 없다는 뜻입니다. 왜냐면 IOException의 catch블록은 도달할 수 없는 코드이기 때문이죠.

finally 블록 : 여기서는 예외가 발생하건 발생하지 않건 공통으로 수행되어야할 코드가 쓰여집니다. 임시 파일의 삭제 등 뒷정리 코드가 쓰입니다.

이것을 이용해서 우리는 위의 코드를 예외처리할 수 있습니다.

```
public static void main(String[] ar){
        int a, b;
        a=10;
        b=0;
        try {
                int c=a/b;
```

```
            System.out.println(c);        //예외발생으로 실행 불가한 코드
        }catch(ArithmeticException e) {
            System.out.println("ArithmeticException 발생");
            System.out.println("0으로 나눌 수는 없습니다");
            e.printStackTrace();
        }finally {
            System.out.println("finally 실행");
        }
    }
```

printStackTrace()라는 메소드는 어느 부분에서 예외가 발생했는지 알려주는 추적 로그를 보여줍니다. Exception이 발생했을 때 기본 동작이죠. 결과는 아래와 같은 것을 알 수 있습니다.

```
ArithmeticException 발생
java.lang.ArithmeticException: / by zero
        at aa.Main.main(Main.java:11)
                    //Main.java에서 11번째 줄에서 발생했다는 printStackTrace
finally 실행
```

2. throws

아까 전에 예외를 그냥 던질 수 있다고 했죠? 그 의미가 어떤 의미냐면 예외를 여기서 처리하지 않을 테니 나를 불러다가 쓰는 녀석에게 에러 처리를 전가하겠다는 의미이며 코드를 짜는 사람이 이 선언부를 보고 어떤 예외가 발생할 수 있는지도 알게 해줍니다. 어떤 뜻인지 모르겠다구요? 아래의 코드를 통해서 알아보도록 합니다.

```
    public static void divide(int a,int b) throws ArithmeticException {
        if(b==0) throw new ArithmeticException("0으로 나눌 수는 없다니까?");
        int c=a/b;
        System.out.println(c);
    }
    public static void main(String[] ar){
        int a=10;
        int b=0;

        divide(a,b);
    }
```

divide()메소드는 a와 b를 나눈 후에 출력하는 역할을 하는데, 이 나누기 부분에서 우리는 예외가 발생할

수 있음을 알았습니다. 그래서 try, catch로 예외 처리를 해야하지만, divide()를 호출하는 부분에서 처리하기를 원합니다. 왜냐면 divide()를 호출한 곳에서 예외가 발생한 다음의 처리를 하는 것이 올바르기 때문입니다. 예를 들어 main 메소드에서는 예외가 발생하면 다시 divide()를 호출하거나, 프로그램을 끝내거나, b의 값을 다시 입력받거나 해야하기 때문인데, divide() 메소드는 그 결정을 할 수 없다는 의미입니다. 그래서 throws ArithmeticException을 divide를 호출한 main에다가 던지는 것(throw)입니다. 여기서 예외를 던지는 방법은 아래와 같습니다.

(아, 참고로 Exception 생성자 중에서 메시지를 받는 생성자가 있는데, 메시지를 보려면 getMessage()메소드를 이용할 수 있습니다. 아래에서 그 메소드를 사용합니다.)

throw 예외객체
ex) throw new Exception("예외 발생!")

예외를 발생시키는 키워드는 throw입니다. 이때 main은 그 예외를 처리하기 위해 try, catch블록을 쓰면 됩니다. 아래처럼 말이죠.

```
try {
        divide(a,b);
}catch(ArithmeticException e) {
        e.getMessage();
        e.printStackTrace();
}
```

throws 키워드로 처리되어야 할 예외가 여러 개가 존재한다면 쉼표로 끊어서 예외를 넘겨줄 수 있습니다. 그 결과는 아래와 같습니다.

```
java.lang.ArithmeticException Create breakpoint : 0으로 나눌 수는 없다니까?
    at exception.ExceptionTest.divide(ExceptionTest.java:5)
    at exception.ExceptionTest.main(ExceptionTest.java:13)
```