예외처리

☞ if문을 이용한 예외의 처리

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

이렇듯 프로그램의 실행 도중에 발생하는 문제의 상황을 가리켜 예외라 한다.
예외는 컴파일 오류와 같은 문법의 오류와는 의미가 다르다.

```
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 사용해 온 예외의 처리방식이다. 이는 if 문이 프로그램의 주 흐름인지, 아니면 예외의 처리인지 구분이 되지 않는다는 단점이 있다.

☞ try~catch문

```
try
{
   //try 영역
}
```

try 영역에서 발생한 AAA 예외상황은

```
catch(AAA e)
{
   //catch 영역
}
```

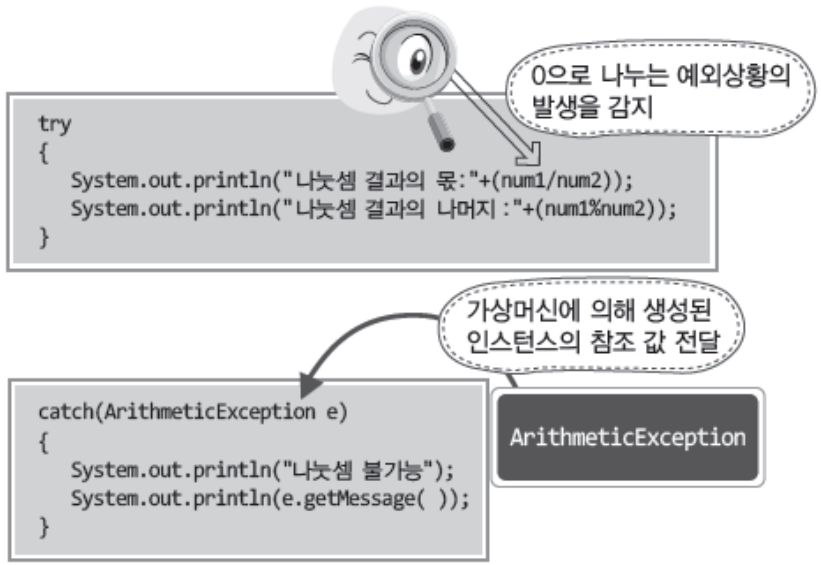
이어서 등장하는 catch 영역에서 처리된다.

try는 예외발생의 감지 대상을 감싸는 목적으로 사용된다. 그리고 catch는 발생한 예외상황의 처리를 위한 목적으로 사용된다.

try~catch의 장점중 하나는!
try 영역을 보면서.. 아! 예외발생 가능 지역이구나!
catch 영역을 보면서.. 아! 예외처리코드이구나!

```
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. 예외발생
2. 참조 값 전달하면서 catch 영역실행
3. catch 영역실행 후, try~catch 다음 문장을 실행



☞ 적절한 try 블록의 구성

try문 내에서 예외상황이 발생하고 처리된 다음에는, 나머지 try문을 건너뛰고, try~catch의 이후를 실행하는 특징을 가짐.

```
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)
{
     . . . .
}
```

☞ e.getMessage( )

- ArithmaticException 클래스와 같이 예외상황을 알리기 위해 정의된 클래스를 가리켜 예외 클래스라 한다.
- 모든 예외클래스는 Throwable 클래스를 상속하며, 이 클래스에는 getMessage 메소드가 정의되어 있다.
- getMessage 메소드는 예외가 발생한 원인정보를 문자열의 형태로 반환한다.

```
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.");
```

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

☞ 예외클래스는 모두 정의가 되어 있는가?

- 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
  → 예외 클래스 : NegativeArraySizeException
- 참조변수가 null로 초기화 된 상황에서 메소드를 호출하는 예외상황
  → 예외 클래스 : NullPointerException

```
try
{
    int[] arr=new int[3];
    arr[-1]=20;
}
catch(ArrayIndexOutOfBoundsException e)
{
    System.out.println(e.getMessage());
}
```

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

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

모든 경우에 있어서 예외로 인정되는 상황을 표현하기 위한 예외클래스는 대부분 정의가 되어있다. 그리고 프로그램에 따라서 별도로 표현해야 하는 예외상황에서는 예외클래스를 직접 정의하면 된다.

☞ 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)
{
    System.out.println("제수는 0이 될 수 없습니다.");
    i-=1;
    continue;
}
catch(ArrayIndexOutOfBoundsException e)
{
    System.out.println("유효하지 않은 인덱스 값입니다.");
    i-=1;
    continue;
}
```

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

☞ 항상 실행되는 finally

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

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

☞ 프로그래머가 직접 정의하는 예외의 상황

- 나이를 입력하라고 했더니, -20살을 입력했다.
- 이름 정보를 입력하라고 했더니, 나이 정보를 입력했다.

이와 같은 상황은 프로그램의 논리적 예외상황이다! 즉, 프로그램의 성격에 따라 결정이 되는 예외상황이다! 따라서 이러한 경우에는 예외클래스를 직접 정의해야 하고, 예외의 발생도 직접 명시해야 한다.

예외 클래스는 Throwable의 하위클래스인 Exception 클래스를 상속해서 정의한다.

```
class AgeInputException extends Exception
{
    public AgeInputException()
    {
        super("유효하지 않은 나이가 입력되었습니다.");
    }
}
```

Exception 클래스의 생성자로 전달되는 문자열이 getMessage 메소드 호출 시 반환되는 문자열이다!

☞ 프로그래머 정의 예외 클래스의 핸들링

```
public static void main(String[ ] args)
{
   System.out.print("나이를 ...
   try
   {
      int age=readAge( );
      System.out.println("당신은 "+age+"세입니다.");
   }
   catch(AgeInputException e)
   {
      System.out.println(e.getMessage( ));
   }
}
```

throws 에 의해 이동된 예외처리 포인트!

예외 상황이 메소드 내에서 처리되지 않으면, 메소드를 호출한 영역으로 예외의 처리가 넘어간다

AgeInputException 예외는 던져버린다

```
public static int readAge() throws AgeInputException
{
   Scanner keyboard=new Scanner(System.in);
   int age=keyboard.nextInt( );
   if(age<0)
   {
      AgeInputException excpt=new AgeInputException();
      throw excpt;
   }
   return age;
}
```

예외가 처리되지 않고 넘어감을 명시해야 한다.

예외상황의 발생지점 예외처리 포인트!

예외처리 메커니즘 가동!

throw는 예외인스턴스의 참조변수를 기반으로 구성을 한다.

☞ 예외를 처리하지 않으면?

```
public static void main(String[] args) throws AgeInputException
{
    System.out.print("나이를 입력하세요 : ");
    int age=readAge();
    System.out.println("당신은 "+age+"세입니다.");
}

public static int readAge() throws AgeInputException
{
    Scanner keyboard=new Scanner(System.in);
    int age=keyboard.nextInt();
    if(age<0)
    {
        AgeInputException excpt=new AgeInputException();
        throw excpt;
    }
    return age;
}
```

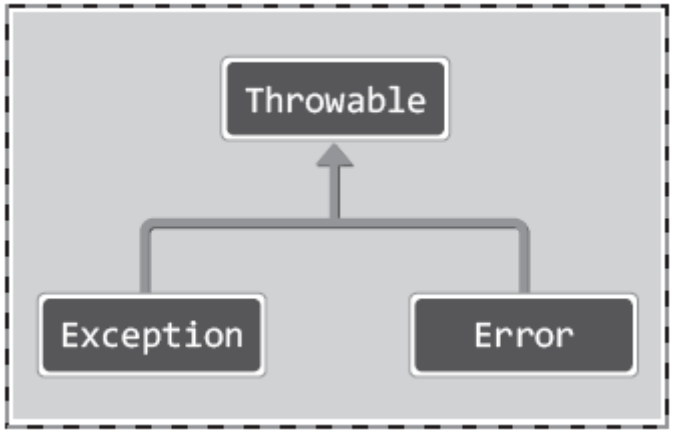
예외가 발생은 되었는데, 처리하지 않으면, 계속해서 반환이 되어 main 메소드를 호출한 가상머신에게 전달이 된다.

```
나이를 입력하세요 : -2
Exception in thread "main" AgeInputException : 유효하지 않은 나이가 입력되었습니다.
        at ThrowsFromMain.readAge(ThrowsFromMain.java : 26)
        at ThrowsFromMain.main(ThrowsFromMain.java : 16)
```

메소드의 호출관계(예외의 전달 흐름)을 보여주는 printStackTrace 메소드의 호출 결과

- 가상머신의 예외처리 1 — getMessage 메소드를 호출한다.
- 가상머신의 예외처리 2 — 예외상황이 발생해서 전달되는 과정을 출력해준다.
- 가상머신의 예외처리 3 — 프로그램을 종료한다

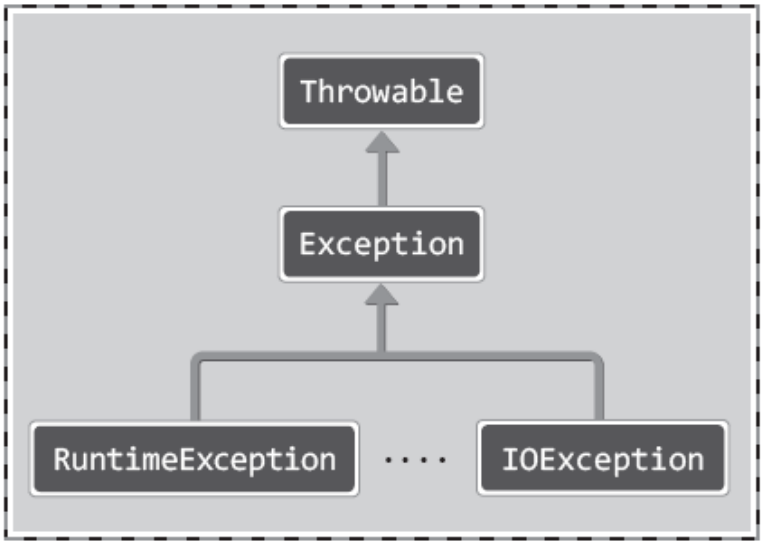
☞ 예외 클래스의 계층도와 Error 클래스



우리의 관심사는 Error 클래스가 아닌,
Exception 클래스에 두어야 한다!

- Error 클래스를 상속하는 예외 클래스는 프로그램 내에서 해결이 불가능한 치명적인 예외 상황을 알리는 예외 클래스의 정의에 사용된다!
- VirtualMachineError 클래스가 대표적인 예! VME 클래스는 JVM에 문제가 생겨서 더 이상 프로그램의 흐름을 이어갈 수 없는 경우를 알림!
- Error 클래스는 try~catch로 처리가 불가능한 예외. JVM에 발생한 문제를 프로그램 내에서 해결할 수 있겠는가? 따라서 이러한 유형의 예외는 JVM에게 전달되도록 두어야 한다.

☞ Exception과 API 문서



앞으로는 호출하고자 하는 메소드의 throws 절을 API 문서에서 확인해야 한다.

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

따라서 clone 메소드를 호출하려면 try~catch 또는 throws 절을 통해서 예외의 처리를 명시해야 한다.

clone (Object 클래스의 인스턴스 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 예외상황의 표현에 사용되지 않는다.
- 때문에t ry~catch문을 통해서 처리하기도 한다.
- 그러나 Error 클래스를 상속하는 예외클래스와 마찬가지로, try~catch문 또는 throws 절을 명시하지 않아도 된다.
- 이들이 명시하는 예외의 상황은 프로그램의 종료로 이어지는 것이 자연스러운 경우가 대부분이기 때문이다.

RuntimeException을 상속하는 대표적인 예외 클래스

- ArrayIndexOutOfBoundsException
- ClassCastException
- NegativeArraySizeException
- NullPointerException

이들은 try~catch문, 또는 throws절을 반드시 필요로 하지 않기 때문에 지금까지 예외처리 없이 예제를 작성할 수 있었다!