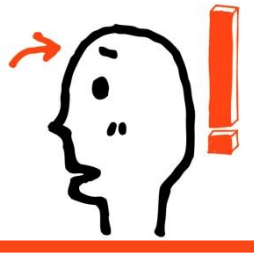
Android Java

17장 예외처리(Exception Handling)

예외처리(Exception Handling)

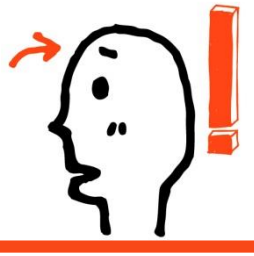


01 예외처리에 대한 이해와 try~chatch문의 기본

02 프로그래머가 직접 정의하는 예외의 상황

03 예외 클래스의 계층도

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.

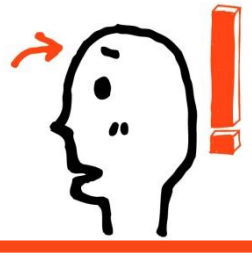
이렇듯 프로그램의 실행 도중에 발생하는 문제의 상황을 가리켜 예외라 한다.

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 사용해온 예외의 처리 방식이다. 이는 if문이 프로그램의 주 흐름인지, 아니면 예외의 처리인지 구분이 되지 안된다는 단점이 있다.

프로그램 실행 중 발생하는 예외



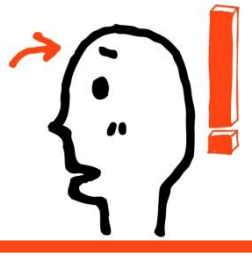
- **컴파일 에러** : 자바 코드로 이뤄진 '.java' 파일을
  실행 가능한 바이너리 파일인 '.class'로 만드는 과정에서 발생하는 에러

- 소스 파일에 잘못된 구문 오류가 있거나
  JVM이 인식할 수 없는 클래스를 사용하여 코딩을 했을 때 발생

- **런타임 에러** : 컴파일은 되었지만 프로그램 실행 도중(runtime)
  논리적인 오류나 외부 요인에 의해 발생하는 에러를 의미

- 런타임 에러는 프로그램 외부의 문제인지 논리적인 오류인지 등
  여러 가지 상황을 고려해야 하므로 에러의 원인을 찾아내기 힘든 경우도 많음

try~catch문



```
try
{
   //try 영역
}
```

try 영역에서 발생한 AAA 예외상황은

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

이어서 등장하는 catch 영역에서 처리된다.

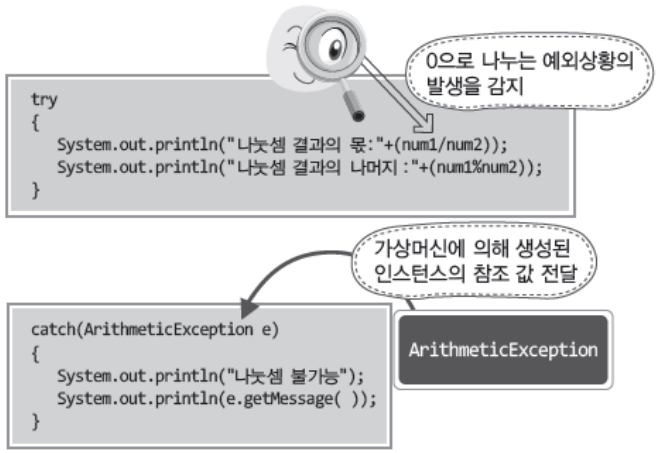
try는 예외발생의 감지 대상을 감싸는 목적으로 사용된다. 그리고 catch는 발생한 예외상황의 처리를 위한 목적으로 사용된다.

try~catch의 장점 중 하나는!

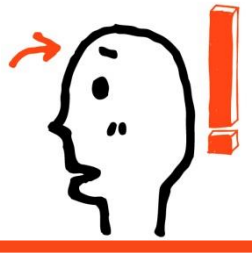
- try 영역을 보면서.. 아! 예외발생 가능지역이구나!
- catch 영역을 보면서.. 아! 예외처리 코드이구나!

```
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 실행한다는 특징으로 인해서 트랜잭션(Transcation)의 구성이 용이하다.

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

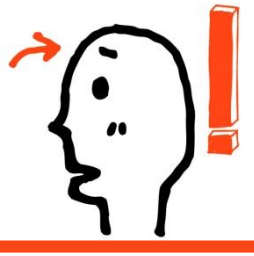
예외상황이 발생해도 실행이 된다.

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

일의 단위가 구성! 함께 실행되거나, 모두 실행되지 않거나!

오류발생시 실행하면 안 되는 부분을 효율적으로 건너 뛸 수 있다!

e.getMessage( )



• ArithmaticException 클래스와 같이 예외상황을 알리기 위해 정의된 클래스를 가리켜 예외 클래스라 한다.

• 모든 예외 클래스는 Throwable 클래스를 상속하며, 이 클래스에는 getMessage 메소드가 정의되어 있다.

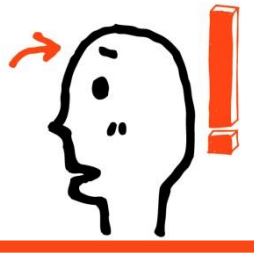
• getMessage 메소드는 예외가 발생한 원인정보를 문자열의 형태로 반환한다.

```
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.");
```

실행의 예

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

예외 클래스는 모두 정의가 되어 있는가?



대표적인 예외 클래스들

- 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

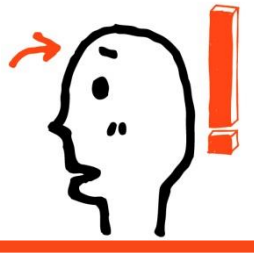
```
try
{
    int[] arr=new int[3];
    arr[-1]=20;
}
catch(ArrayIndexOutOfBoundsException e)
{
    System.out.println(e.getMessage());
}
```

```
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 예외 클래스는 대부분 정의가 되어 있다. 그리고 프로그램에 따라서 별도로 표현해야 하는 예외 상황에서는 예외 클래스를 직접 정의하면 된다.

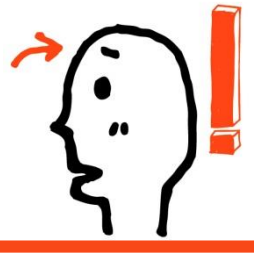
자주 발생하는 예외



예외 종류	예외 발생 경우
ArithmeticException	정수를 0으로 나눌 때 발생
NullPointerException	null 레퍼런스를 참조할 때 발생
ClassCastException	변환할 수 없는 타입으로 객체를 변환할 때 발생
OutOfMemoryException	메모리가 부족한 경우 발생
ArrayIndexOutOfBoundsException	배열의 범위를 벗어난 접근 시 발생
IllegalArgumentException	잘못된 인자 전달 시 발생
IOException	입출력 동작 실패 또는 인터럽트 시 발생
NumberFormatException	문자열이 나타내는 숫자와 일치하지 않는 타입의 숫자로 변환 시 발생

OutOfMemoryError

실행 예외(RuntimeException)



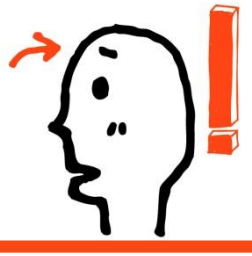
- **NullPointerException**
  - 객체 참조가 없는 상태
    - null 값 갖는 참조변수로 객체 접근 연산자인 도트(.) 사용했을 때 발생

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

- **ArrayIndexOutOfBoundsException**
  - 배열에서 인덱스 범위 초과하여 사용할 경우 발생

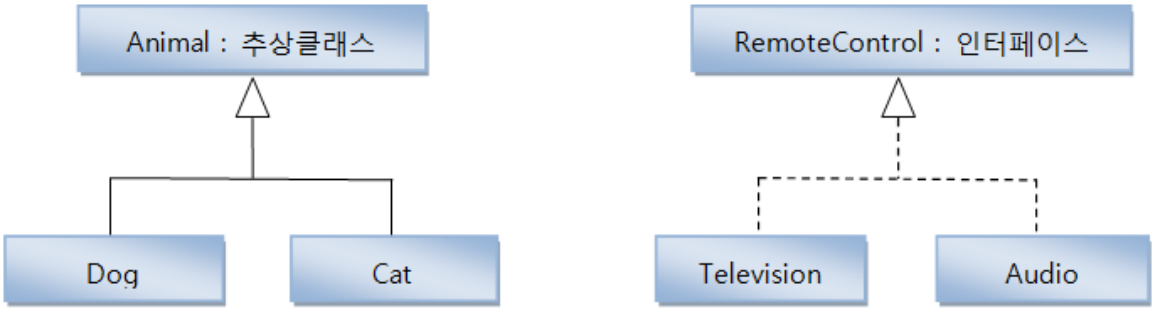
```
public static void main(String[] args) {
   String data1 = args[0];
   String data2 = args[1];

   System.out.println("args[0]: " + data1);
   System.out.println("args[1]: " + data2);
}
```

실행 예외(RuntimeException)



ClassCastException

- 타입 변환이 되지 않을 경우 발생



- 정상 코드

```
Animal animal = new Dog();
Dog dog = (Dog) animal;
```

```
RemoteControl rc = new Television();
Television tv = (Television) rc;
```

- 예외 발생 코드

```
Animal animal = new Dog();
Cat cat = (Cat) animal;
```

```
RemoteControl rc = new Television();
Audio audio = (Audio) rc;
```

정수가 아닌 문자열을 정수로 변환할 때 예외 발생

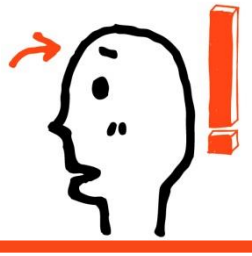
문자열을 정수로 변환할 때 발생하는 NumberFormatException을 처리하는 프로그램을 작성하라.

```
public class NumException {
   public static void main (String[] args) {
      String[] stringNumber = {"23", "12", "998", "3.141592"};
      try {
         for (int i = 0; i < stringNumber.length; i++) {
            int j = Integer.parseInt(stringNumber[i]);
            System.out.println("숫자로 변환된 값은 " + j);
         }
      }
      catch (NumberFormatException e) {
         System.out.println("정수로 변환할 수 없습니다.");
      }
   }
}
```

"3.141592"를 정수로 변환할 때 NumberFormatException 예외 발생

```
숫자로 변환된 값은 23
숫자로 변환된 값은 12
숫자로 변환된 값은 998
정수로 변환할 수 없습니다.
```

범위를 벗어난 배열의 접근



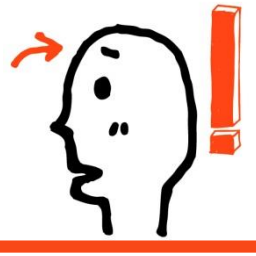
배열의 인덱스가 범위를 벗어날 때 발생하는 ArrayIndexOutOfBoundsException을 처리하는 프로그램을 작성하시오.

```
public class ArrayException {
    public static void main (String[] args) {
        int[] intArray = new int[5];
        intArray[0] = 0;
        try {
            for (int i = 0; i < 5; i++) {
                intArray[i+1] = i+1 + intArray[i];
                System.out.println("intArray["+i+"]"+"="+intArray[i]);
            }
        }
        catch (ArrayIndexOutOfBoundsException e) {
            System.out.println("배열의 인덱스가 범위를 벗어났습니다.");
        }
    }
}
```

i가 4일 때 ArrayIndexOutOfBoundsException 예외 발생

```
intArray[0]=0
intArray[1]=1
intArray[2]=3
intArray[3]=6
배열의 인덱스가 범위를 벗어났습니다.
```

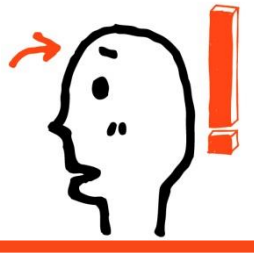
try~catch의 또다른 장점



```
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 잇기 때문에 예외처리와 관련된 부분을 완전히 별도로 떼어 놓을 수 있다!

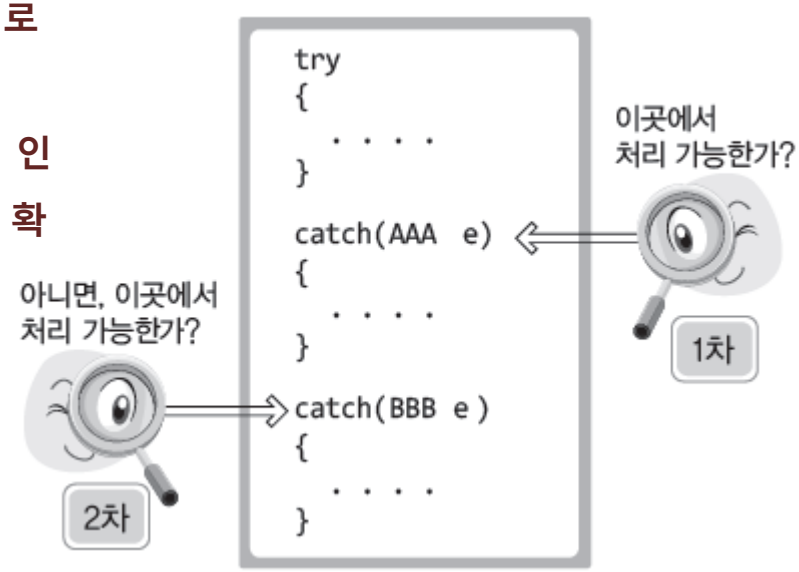
catch가 결정되는 방법



첫 번째 catch 블록에서부터 순서대로 찾아 내려온다.

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

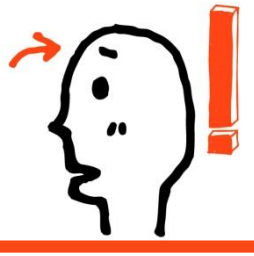
```
try
{
    . . . .
}
catch(Throwable e)
{
    . . . .
}
catch(ArithmeticException e)
{
    . . . .
}
```



따라서 이렇게 catch문을 구성하면 에러발생!

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

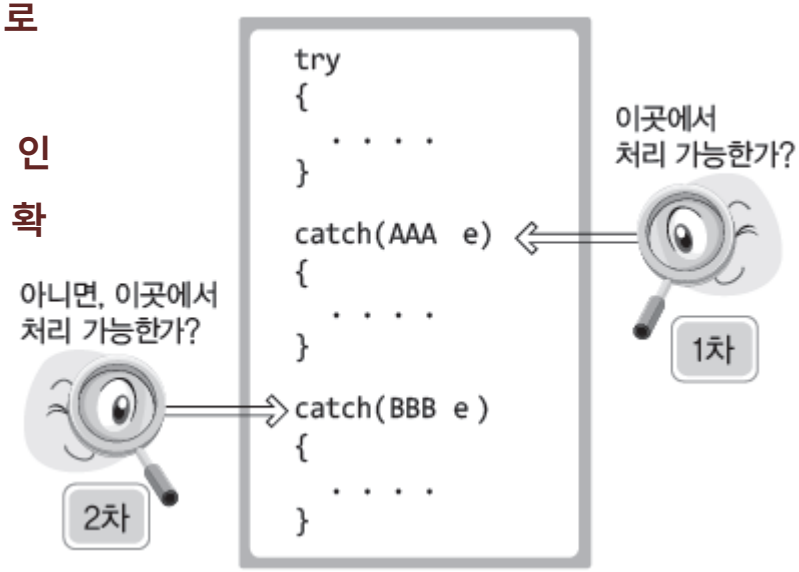
catch가 결정되는 방법



첫 번째 catch 블록에서부터 순서대로 찾아 내려온다.

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

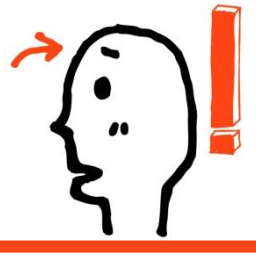
```
try
{
    . . . .
}
catch(Throwable e)
{
    . . . .
}
catch(ArithmeticException e)
{
    . . . .
}
```



따라서 이렇게 catch문을 구성하면 에러발생!

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

catch가 결정되는 방법

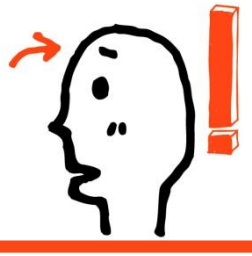


❖ 멀티(multi) catch

- 자바 7부터는 하나의 catch 블록에서 여러 개의 예외 처리 가능
  - 동일하게 처리하고 싶은 예외를 |로 연결

```
try {
        ArrayIndexOutOfBoundsException 또는 NumberFormatException 발생
        다른 Exception 발생
} catch(ArrayIndexOutOfBoundsException | NumberFormatException e) {
        예외 처리 1
} catch(Exception e) {
        예외 처리 2
}
```

try-catch-finally 구문만 알면 된다



try-catch-finally 구문 사용법

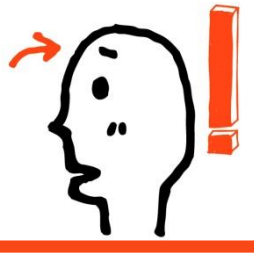
- **try 영역** : 예외가 발생할 수 있는 구문을 지정
  try 영역이 아닌 곳에서 발생된 예외에 대해서는 예외 처리 불가

- **catch 영역** : try 영역에서 발생한 예외를 받고, 처리하는 영역
  catch 키워드 다음에 위치한 괄호 내부에는 어떤 예외를 받을지에 대해서 명시
  괄호 내부에는 변수를 선언하듯 클래스명과 변수 이름을 적음

- **finally 영역** : 반드시 실행해야 할 구문을 처리하는 구문을 다룸
  예외가 발생 여부와 상관 없이 이 영역 안에 있는 구문들은 반드시 실행

- try-catch 영역은 반드시 동시에 존재해야 하지만 finally 영역은 상황에 따라서 생략 가능

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");
}
```

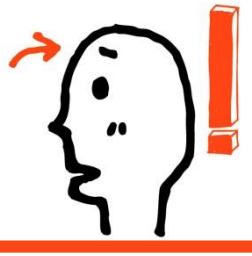
try 블록 내에 진입하면,

finally 블록은 무조건 실행된다!

·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!

예외 클래스의 정의와 throw



- 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 예외 클래스를 직접 정의해야 하고, 예외의 발생도 직접 명시해야 한다.

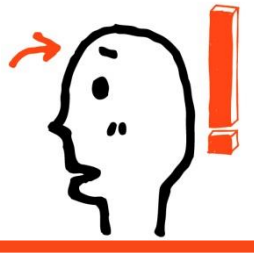
예외 클래스는 Throwable의 하위 클래스인 Exception 클래스를 상속해서 정의한다.

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

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

프로그래머가 직접 정의하는 예외의 상황

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( ));
   }
}
```

throws에 의해 이동된 예외처리 포인트!

```
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;
}
```

AgeInputException 예외는 던져버린다

예외상황의 발생지점 예외처리 포인트!

예외처리 메커니즘 가동!

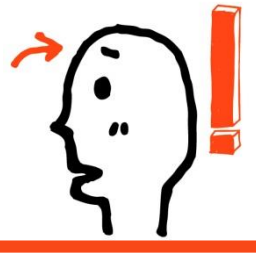
예외상황이 메소드 내에서 처리되지 않으면, 메소드를 호출한 영역으로 예외의 처리가 넘어간다!

예외가 처리되지 않고 넘어감을 명시해야 한다.

throw는 예외 인스턴스의 참조변수를 기반으로 구성을 한다.

프로그래머가 직접 정의하는 예외의 상황

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)
```

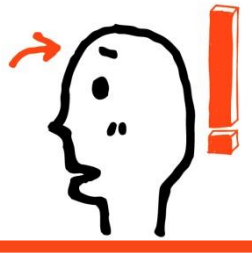
메소드의 호출관계(예외의 전달 흐름)을 보여주는 printStackTrace 메소드의 호출결과

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

가상머신의 예외처리 방식

프로그래머가 직접 정의하는 예외의 상황

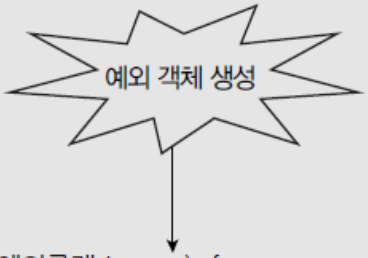
예외를 처리하지 않으면?



❖ printStackTrace()

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

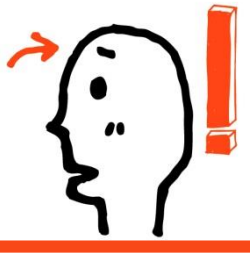
```
try {
```

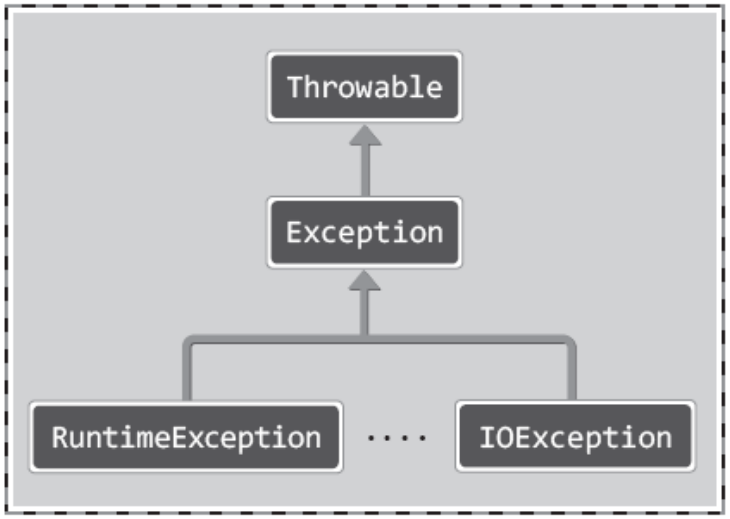



```
} catch(예외클래스    e) {
    //예외가 가지고 있는 Message 얻기
    String message = e.getMessage();

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

프로그래머가 직접 정의하는 예외의 상황

예외 클래스의 계층도와 Error 클래스





앞으로는 호출하고자 하는 메소드의 throws 절을 API 문서에서 확인해야 한다.

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

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

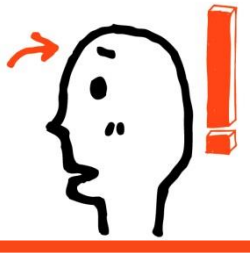
따라서 clone 메소드를 호출하려면 try~catch 또는 throws 절을 통해서 예외의 처리를 명시해야 한다.

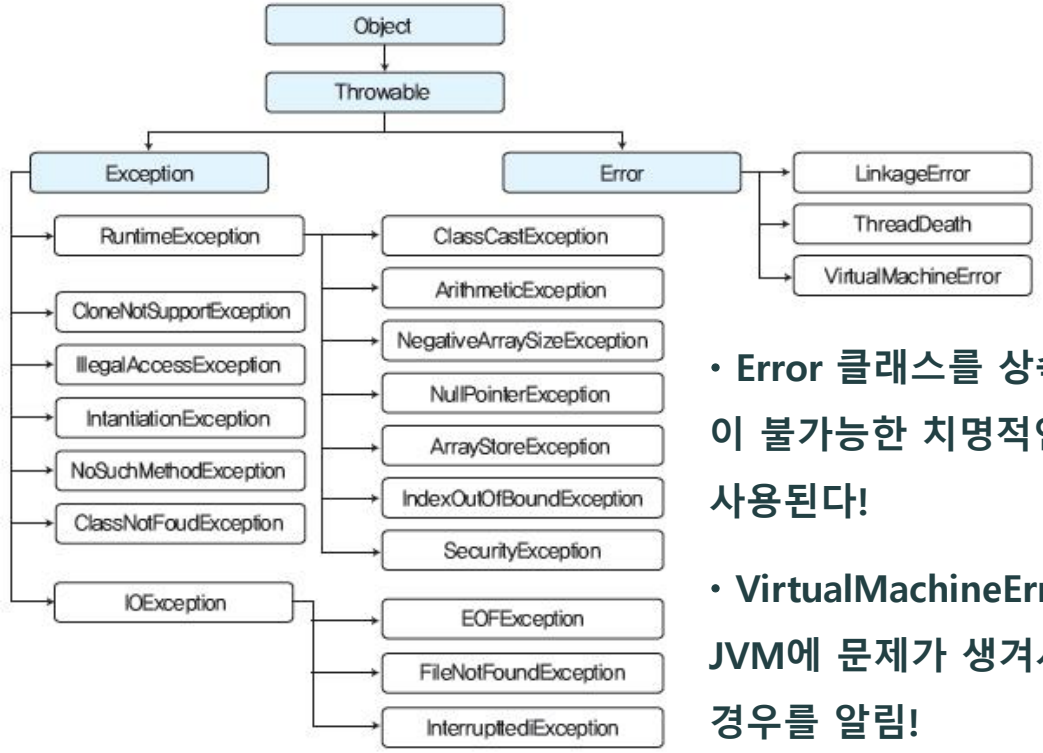
clone (Object 클래스의 인스턴스 메소드)

```
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.

Exception과 API 문서



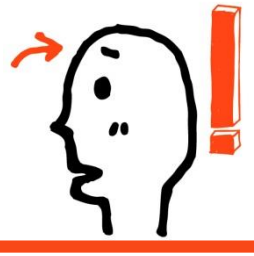


우리의 관심사는 Error 클래스가 아닌,
Exception 클래스에 두어야 한다!

- 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.

예외 클래스의 계층도

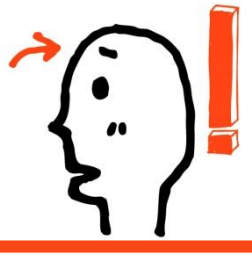
Throwable 클래스와 계층도 이해하기



자바에서 제공하는 오류 관련 클래스

- 오류 관련 클래스들은 기본적으로 Object 클래스를 상속
- 모든 예외 클래스의 부모 클래스는 Throwable 클래스
- Throwable 클래스를 상속받은 클래스만이 try-catch-finally 구문을 사용하여 예외 처리를 실행할 수 있음
- Throwable 클래스를 상속 받는 클래스는 오직 Exception과 Error 클래스뿐
- 다수의 클래스들이 Exception 클래스를 상속받는 구조

처리하지 않아도 되는 RuntimeException



- Exception 클래스의 하위 클래스이다.
- Error를 상속하는 예외 클래스만큼 치명적인 예외상황의 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.

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!

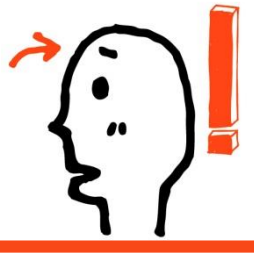
THANK YOU

실무에서 알아야 할 기술은 따로 있다! 자바를 다루는 기술

Android Java

18장 Object 클래스

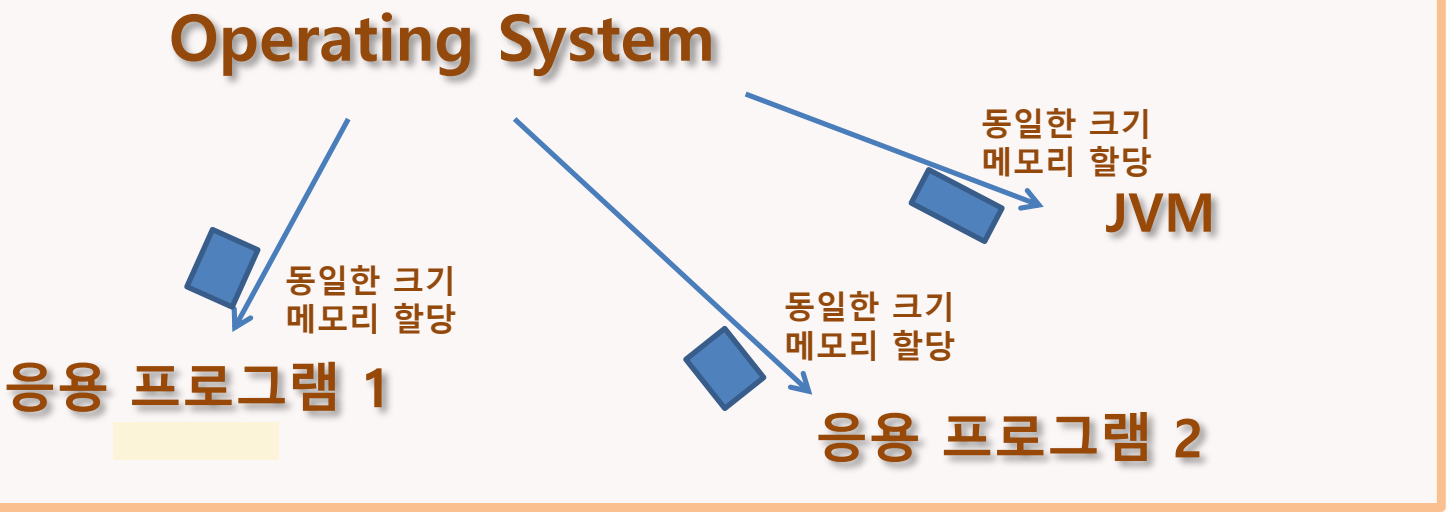
JVM은 운영체제 위에서 동작한다



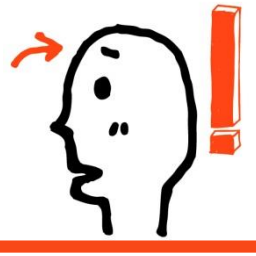
· 운영체제가 JVM을 포함해서 모든 응용 프로그램에게 동일한 크기의 메모리 공간을 할당할 수 있는 이유는 가상 메모리 기술에서 찾을 수 있다.

·JVM은 운영체제로부터 할당받은 메모리 공간을 기반으로 자바 프로그램을 실행해야 한다.

· JVM은 운영체제로부터 할당받은 메모리 공간을 이용해서 자기 자신도 실행을 하고, 자바 프로그램도 실행을 한다.



JVM의 메모리 살림살이

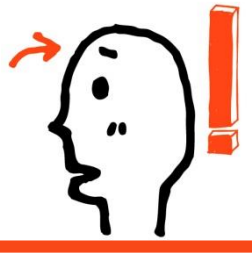


JVM의 메모리 구분 및 관리 기준

- 메소드 영역 (method area) — 메소드의 바이트코드, static 변수
- 스택 영역 (stack area) — 지역변수, 매개변수
- 힙 영역 (heap area) — 인스턴스

메모리 공간을 용도에 따라서 별도로 나누는 이유는, 서랍장의 칸을 구분하고, 칸 별로 용도를 지정하는 이유와 차이가 없다!

메소드 영역과 스택의 특성



메소드 영역에 대한 설명

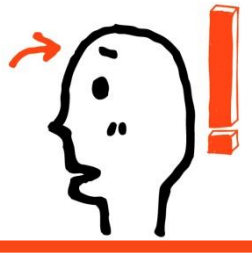
- 자바 바이트코드(bytecode) : 자바 가상머신의 의해서 실행되는 코드
- 메소드의 자바 바이트코드는 JVM이 구분하는 메모리 공간 중에서 메소드 영역에 저장된다.
- static으로 선언된 클래스 변수도 메소드 영역에 저장된다.
- 정리하면, 클래스의 정보가 JVM의 메모리 공간에 LOAD 될 때 할당 및 초기화되는 대상은 메소드 영역에 할당된다.

참고로, 메소드의 바이트코드는 실행에 필요한 바이트코드 전부를 의미한다. 자바 프로그램의 실행은 메소드 내에 정의된 문장들의 실행으로 완성되기 때문이다.

스택 영역에 대한 설명

- 매개변수, 지역변수가 할당되는 메모리 공간
- 프로그램이 실행되는 도중에 임시로 할당되었다가 바로 이어서 소멸되는 특징이 있는 변수가 할당된다.
- 메소드의 실행을 위한 메모리 공간으로도 정의할 수 있다.

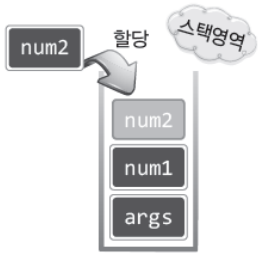
스택의 흐름



Ⅰ

```
public static void main(String[ ] args)
{
  int num1=10;
  int num2=20;
  adder(num1, num2);
  . . . .
}

public static void adder(int n1, int n2)
{
  int result=n1+n2;
  return result;
}
```

현재 실행위치

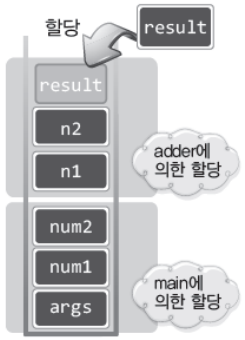


Ⅱ

```
public static void main(String[ ] args)
{
  int num1=10;
  int num2=20;
  adder(num1, num2);
  . . . .
}

public static void adder(int n1, int n2)
{
  int result=n1+n2;
  return result;
}
```

메소드가 호출됨

현재 실행위치



- 지역변수는 스택에 할당된다.
- 스택에 할당된 지역변수는 해당 메소드를 빠져나가면 소멸된다.
- 할당 및 소멸의 특성상 그 형태가 접시를 쌓는 것과 유사하다 따라서 스택이라 이름 지어졌다.

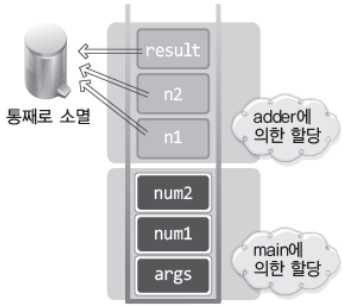
할당 및 소멸의 특성상

메소드 별 스택이 구분이 된다!

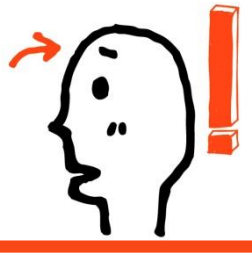
Ⅲ

```
public static void main(String[ ] args)
{
  int num1=10;
  int num2=20;
  adder(num1, num2);
  . . . .
}

public static void adder(int n1, int n2)
{
  int result=n1+n2;
  return result;
}
```

현재 실행위치

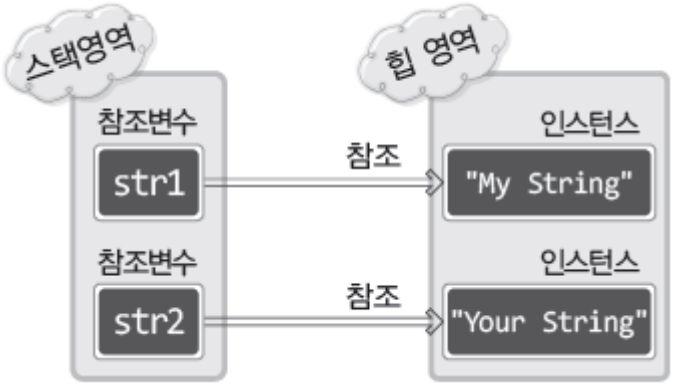


힙 영역



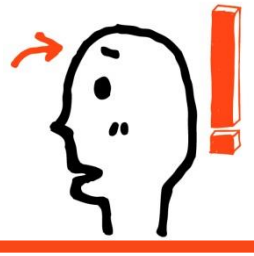
힙 영역에 대한 설명

- 인스턴스가 생성되는 메모리 공간
- JVM에 의한 메모리 공간의 정리(Garbage Collection)가 이뤄지는 공간
- 할당은 프로그래머가 소멸은 JVM이.
- 참조변수에 의한 참조가 전혀 이뤄지지 않는 인스턴스가 소멸의 대상이 된다. 따라서 JVM은 인스턴스의 참조관계를 확인하고 소멸할 대상을 선정한다.



메소드 내에서 인스턴스를 생성한다면 위의 그림에서 설명하듯이 참조변수는 스택에 인스턴스는 힙에 저장된다.

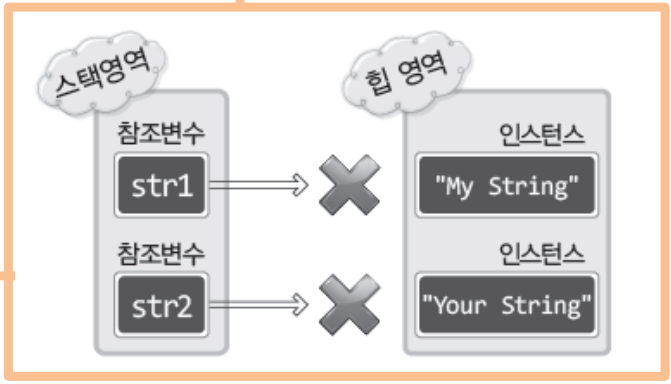
힙 영역

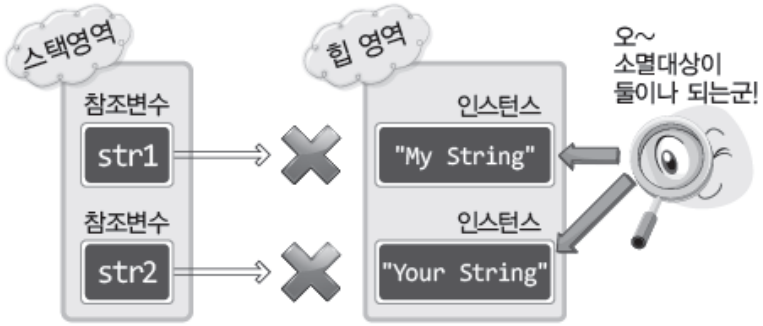


```
public staic void simpleMethod()
{
    String str1=new String("My String");
    String str2=new String("Your String");
    . . . .
    str1=null;
    str2=null;
    . . . .
}
```

참조가 이뤄지지 않으면

소멸의 대상이 된다!

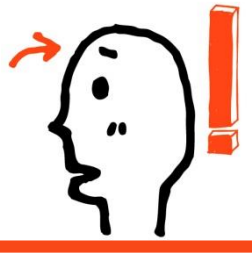




JVM은 인스턴스의 참조관계를

통해서 소멸 대상을 결정한다!

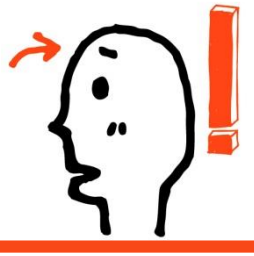
Object 클래스



클래스의 조상, Object 클래스

- 자바의 최상위 클래스는 **Object**이며 모든 클래스들은 Object 클래스를 상속받음
- extends 키워드를 선언하지 않아도 자바로 개발된 모든 클래스들은 최상위 클래스인 Object를 기본적으로 상속
- **트리(Tree) 구조, 계층 구조(Hierarchy structure)** : 하나의 뿌리에서 여러 개의 가지(branch)를 치는 형상

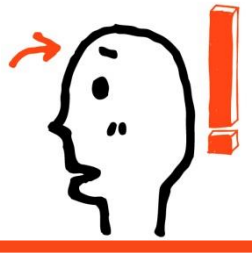
Object 클래스



클래스의 조상, Object 클래스

메소드 이름	설명
equals	• 선언부 : public boolean equals(Object obj) • 설명 : 매개변수로 받는 obj 객체와 메소드가 포함된 객체가 서로 같은지 비교한다. equals( ) 메소드는 JVM 메모리의 주소가 같은지 확인해서 같으면 true, 다르면 false를 반환한다.
toString	• 선언부 : public String toString( ) • 설명 : 객체의 정보를 문자열로 반환하는 기능을 한다. 보통은 [클래스 이름]@[16진수 코드값]과 같은 형태로 반환한다. System.out.println( ) 메소드에 객체를 매개변수로 입력하면 println( ) 메소드가 자동으로 toString( ) 값을 호출해서 반환된 문자열을 출력한다.
hashCode	• 선언부 : public int hashCode( ) • 설명 : JVM 메모리에 저장된 객체 고유의 값 즉, 해시 코드를 반환하는 메소드다.
clone	• 선언부 : protected Object clone( ) • 설명 : 객체를 복사하여 새로운 객체를 반환한다.

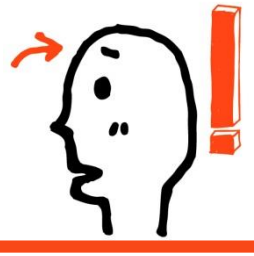
Object 클래스



클래스의 조상, Object 클래스

finalize	• 선언부 : protected void finalize( ) • 설명 : JVM에서 사용하지 않는 메모리를 청소하는 가비지 컬렉터에 의해서 호출되는 메소드다. 그러므로 객체를 종료할 때 실행할 구문이 선언되어 있다.
getClass	• 선언부 : public Class getClass( ) • 설명 : 객체에서 클래스를 받아올 때 사용하는 메소드다.
notify	• 선언부 : public void notify( ) • 설명 : 대기 중인 스레드를 시작한다(스레드와 관련된 내용은 스레드 프로그래밍에서 설명한다).
notifyAll	• 선언부 : public void notifyAll( ) • 설명 : 대기 중인 모든 스레드를 시작한다.
wait	• 선언부 : public void wait( ) • 설명 : 현재의 스레드가 동작하는 모든 일을 멈추고, 대기 상태로 만든다.

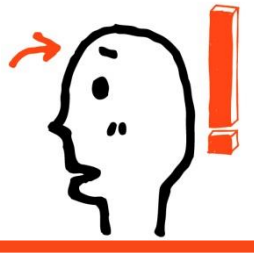
예제를 통한 getClass()의 이해



```
class Point {
   int x, y;
   public Point(int x, int y) {
      this.x = x;
      this.y = y;
   }
}

public class ObjectProperty {
   public static void main(String [] args) {
      Point p = new Point(2,3);
      System.out.println(p.getClass().getName());
   }
}
```

```
Point
```

예제를 통한 toString()의 이해

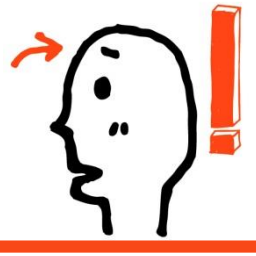


```
class Point {
   int x, y;

   public Point(int x, int y) {
      this.x = x;
      this.y = y;
   }
   public String toString() {
      return "Point(" + x + "," + y+ ")";
   }
}

public class ObjectProperty {
   public static void main(String [] args) {
      Point a = new Point(2,3);
      System.out.println(a.toString());
   }
}
```

System.out.println(a); 라고 해도 동일

```
Point(2,3)
```

인스턴스 소멸 시 호출되는 finalize 메소드



```
protected void finalize( ) throws Throwable
```

인스턴스가 완전히 소멸되기 직전 호출되는 메소드,

Object 클래스의 멤버이므로 모든 인스턴스에는 이 메소드가 존재한다.

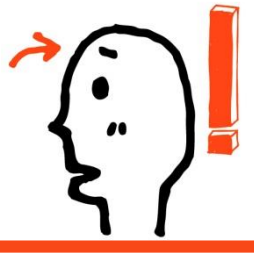
```
class MyName
{
    String objName;
    public MyName(String name)
    {
        objName=name;
    }
    protected void finalize() throws Thr
    {
        super.finalize();
        System.out.println(objName+"이 소멸되었습니다.");
    }
}
```

```
public static void main(String[] args)
{
    MyName obj1=new MyName("인스턴스1");
    MyName obj2=new MyName("인스턴스2");
    obj1=null;
    obj2=null;

    System.out.println("프로그램을 종료합니다.");
    // System.gc();
    // System.runFinalization();
}
```

위 예제의 실행과정에서 finalize 메소드는 호출되지 않을 수 있다.

Garbage Collection이 실행되는 시기와 인스턴스의 완전한 소멸의 시기는 차이가 날 수 있기 때문이다.

finalize 메소드의 오버라이딩의 예



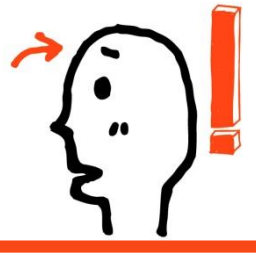
```
protected void finalize() throws Throwable
{
    super.finalize();
    System.out.println(objName+"이 소멸되었습니다.");
}
```

이것은 모범이 되는 메소드 오버라이딩의 예! 이다.

super.finalize( )

- Object 클래스에 정의되어 있는 finalize 메소드에 중요한 코드가 삽입되어 있는지 확인한 바 없다!
- 만약에 중요한 코드가 삽입되어 있다면? 단순한 오버라이딩으로 인해서 중요한 코드의 실행을 방해할 수 있다!
- 따라서! 대상 메소드에 대한 정보가 부족한 경우에는 오버라이딩 된 메소드도 호출이 되도록 오버라이딩을 하자! 이것이 오버라이딩의 기본 원칙이다.

인스턴스 비교



```
class IntNumber
{
    int num;
    public IntNumber(int num) { this.num=num; }
    public boolean isEquals(IntNumber numObj)
    {
        if(this.num==numObj.num)
            return true;
        else
            return false;
    }
}
```

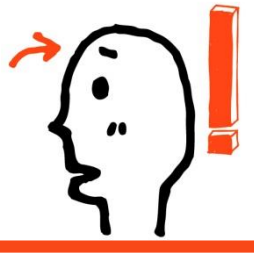
이전에 언급했듯이 == 연산자는 참조 값 비교를 한다. 따라서 인스턴스간 내용비교를 위해서는 내용 비교 기능의 메소드가 필요하다.

```
public static void main(String[] args)
{
    IntNumber num1=new IntNumber(10);
    IntNumber num2=new IntNumber(12);
    IntNumber num3=new IntNumber(10);

    if(num1.isEquals(num2))
        System.out.println("num1과 num2는 동일한 정수");
    else
        System.out.println("num1과 num2는 다른 정수");

    if(num1.isEquals(num3))
        System.out.println("num1과 num3는 동일한 정수");
    else
        System.out.println("num1과 num3는 다른 정수");
}
```

실행 결과

```
num1과 num2는 다른 정수
num1과 num3는 동일한 정수
```

equals 메소드

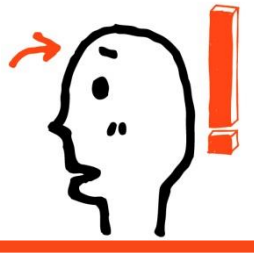


```
class IntNumber
{
    int num;
    public IntNumber(int num)
    {
        this.num=num;
    }
    public boolean equals(Object obj)
    {
        if(this.num==((IntNumber)obj).num)
            return true;
        else
            return false;
    }
}
```

JAVA에서는 인스턴스간의 내용 비교를 목적으로 Object 클래스에 equals 메소드를 정의해 놓았다.
따라서 새로 정의되는 클래스의 내용 비교가 가능하도록 이 메소드를 오버라이딩 하는 것이 좋다!

Object 클래스의 equals 메소드를 인스턴스의 내용비교 메소드로 지정해 놓았기 때문에, 처음 접하는 클래스의 인스턴스라 하더라도 equals 메소드의 호출을 통해서 인스턴스간 내용 비교를 할 수 있다.

Rect 클래스 만들고 equals() 만들기



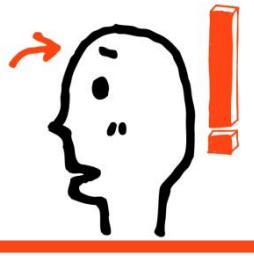
int 타입의 width, height의 필드를 가지는 Rect 클래스를 작성하고, 두 Rect 객체의 width, height 필드에 의해 구성되는 면적이 같으면 두 객체가 같은 것으로 판별하도록 equals()를 작성하라. Rect 생성자에서 width, height 필드를 인자로 받아 초기화한다.

```
class Rect {
   int width;
   int height;
   public Rect(int width, int height) {
      this.width = width;
      this.height = height;
   }
   public boolean equals(Rect p) {
      if (width*height == p.width*p.height)
         return true;
      else
         return false;
   }
}
```

```
public class EqualsEx {
   public static void main(String[] args) {
      Rect a = new Rect(2,3);
      Rect b = new Rect(3,2);
      Rect c = new Rect(3,4);
      if(a.equals(b)) System.out.println("a is equal to b");
      if(a.equals(c)) System.out.println("a is equal to c");
      if(b.equals(c)) System.out.println("b is equal to c");
   }
}
```

```
a is equal to b
```

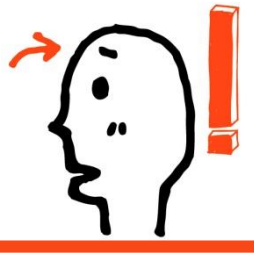
자주 사용하는 Object 슈퍼 클래스 활용하기



코드 13-1 package com.gilbut.chapter13;

```
1   public class ObjectSample1
2   {
3       private int value;
4
5       public static void main(String[] args)
6       {
7           ObjectSample1 sample1 = new ObjectSample1(1);
8           ObjectSample1 sample2 = new ObjectSample1(2);
9           ObjectSample1 sample3 = new ObjectSample1(1);
10
11          System.out.println("HashCode sample1 : " + sample1.hashCode());
12          System.out.println("HashCode sample2 : " + sample2.hashCode());
13          System.out.println("HashCode sample3 : " + sample3.hashCode());
14
15          System.out.println("sample1 and sample2 is same : " + sample1.
            equals(sample2));
16          System.out.println("sample1 and sample3 is same : " + sample1.
            equals(sample3));
17      }
```

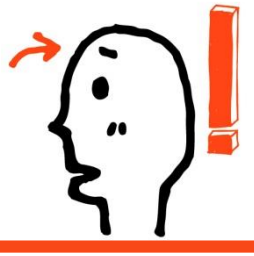
자주 사용하는 Object 슈퍼 클래스 활용하기



```
18
19      @Override
20      public boolean equals(Object obj)
21      {
22         boolean rt = false;
23         if (obj instanceof ObjectSample1)
24         {
25            ObjectSample1 sample = (ObjectSample1) obj;
26            rt = (sample.getValue() == value);
27         }
28         return rt;
29      }
30
31      public ObjectSample1(int value)
32      {
33         this.value = value;
34      }
35
```

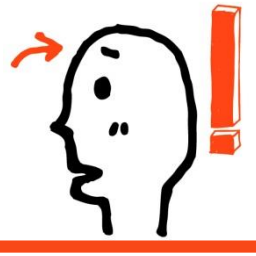
Object 클래스의 equal( ) 메소드를 오버라이딩한 구문으로, 개발자가 원하는 방법으로 비교하기 위해서 종종 재정의한다는 것 기억하세요. 마찬가지로 toString( ) 메소드 또한 자주 오버라이딩합니다.

자주 사용하는 Object 슈퍼 클래스 활용하기



```
36      public int getValue()
37      {
38          return this.value;
39      }
40
41      public void setValue(int value)
42      {
43          this.value = value;
44      }
45  }
```

자주 사용하는 Object 슈퍼 클래스 활용하기

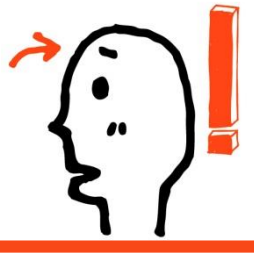


실행결과

```
HashCode sample1 : 190454046
HashCode sample2 : 1210517092
HashCode sample3 : 2048177213
sample1 and sample2 is same : false
sample1 and sample3 is same : true
```

- ObjectSample1 클래 스와 같이 equals( ) 메소드를 재정의해서 사용하는 예제
- equals( ) 메소드의 기본 기능은 JVM의 참조 위치를 서로 비교하는 것이지만 JVM의 위치가 아닌 실제값을 비교하거나 다른 방법으로 비교하기 위해서 equals( ) 메소드를 오버라이딩해서 사용

인스턴스의 복사(복제): clone 메소드

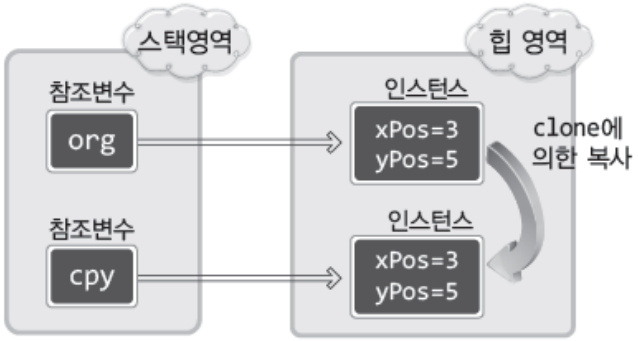


- Object 클래스에는 인스턴스의 복사를 목적으로 clone이라는 이름의 메소드가 정의되어 있다.
- 단, 이 메소드는 Cloneable 인터페이스를 구현하는 클래스의 인스턴스에서만 호출될 수 있다.
- Cloneable 인터페이스의 구현은 다음의 의미를 지닌다.
  "이 클래스의 인스턴스는 복사를 해도 됩니다."
- 사실 인스턴스의 복사는 매우 민감한 작업이다. 따라서 클래스를 정의할 때 복사의 허용여부를 결정하도록 Cloneable 인터페이스를 통해서 요구하고 있다.

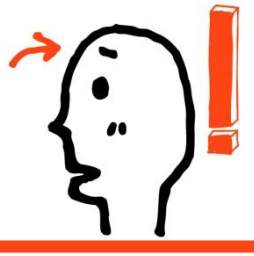
```
class Point implements Cloneable
{
    private int xPos;
    private int yPos;
    . . . . .
    public Object clone() throws CloneNotSupportedException
    {
        return super.clone();
    }
}
```

clone 메소드는 protected로 선언되어 있다.
따라서 외부 호출이 가능하도록 public으로 오버라이딩!

인스턴스를 통째로 복사한다!

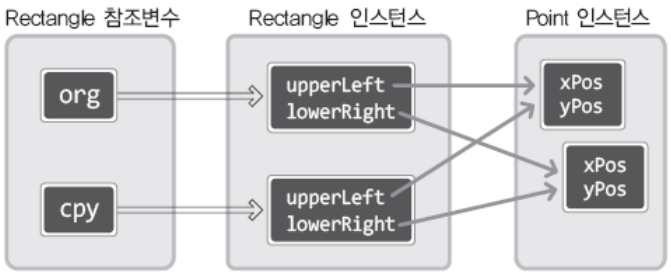


얕은(Shallow) 복사의 예



```
class Rectangle implements Cloneable
{
    Point upperLeft, lowerRight;

    public Rectangle(int x1, int y1, int x2, int y2)
    {
        upperLeft=new Point(x1, y1);
        lowerRight=new Point(x2, y2);
    }

    . . . . . .

    public Object clone() throws CloneNotSupportedException
    {
        return super.clone();
    }
}
```

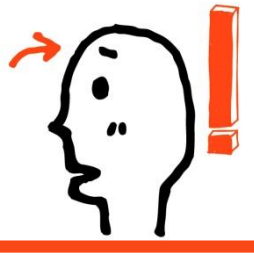
```
public static void main(String[] args)
{
    Rectangle org=new Rectangle(1, 1, 9, 9);
    Rectangle cpy;

    try
    {
        cpy=(Rectangle)org.clone();
        org.changePos(2, 2, 7, 7);
        org.showPosition();
        cpy.showPosition();
    }
    catch(CloneNotSupportedException e)
    {
        e.printStackTrace();
    }
}
```



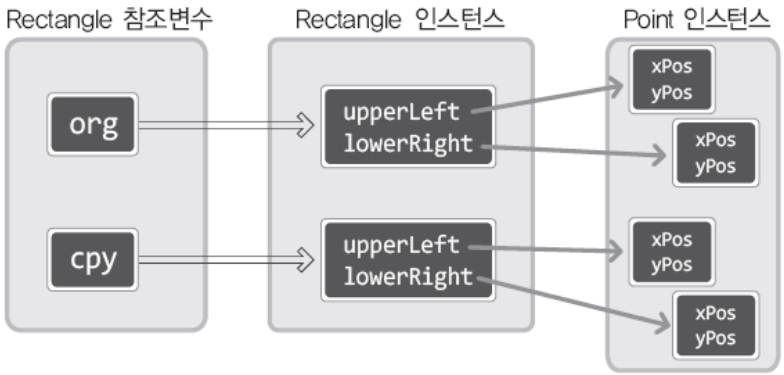
복사결과!
이러한 유형의 복사를 가리켜 얕은 복사라 한다!

Object 클래스의 clone 메소드는 인스턴스를 통째로 복사는 하지만, 인스턴스가 참조하고 있는 또 다른 인스턴스까지 복사하지는 않는다. 단순히 참조 값만을 복사할 뿐이다!

깊은(Deep) 복사의 예



```
class Rectangle implements Cloneable
{
    // clone 메소드를 제외한 나머지는 ShallowCopy.java와 동일하므로 생략

    public Object clone() throws CloneNotSupportedException
    {
        Rectangle copy=(Rectangle)super.clone();
        copy.upperLeft=(Point)upperLeft.clone();
        copy.lowerRight=(Point)lowerRight.clone();
        return copy;
    }
}
```



복사결과!
이러한 유형의 복사를 가리켜 깊은 복사라 한다!

THANK YOU

실무에서 알아야 할 기술은 따로 있다! 자바를 다루는 기술