**Source file 구조**

1. 자바 파일의 이름은 반드시 .java 확장자로 끝나야 한다.
2. source안에 public인 top level class가 있다면 파일이름은 반드시 해당 class의 이름으로 되어야 한다.
   * public인 class가 2개 이상 하나의 파일에 존재 할 수 없으므로 이런 경우는 따로 만들어야 한다.
   * public인 class가 하나도 없다면 파일 이름은 어떤 class의 이름을 사용하여도 상관없다
   * 만약 public인 class가 존재하고 타 클래스가 main method를 가지고 있다면 public인 class의 이름으로 파일을 생성하여 main method를 가진 클래스를 호출해야한다.
3. source 파일의 구조는 문장에 따라서 순서가 정해져 있다.
   * package 선언문
   * import 문
   * class 정의문

**keyword**

1. 자바의 키워드는 모두 소문자이다.
   * abstract   do   implements   private   throw  
     boolean   double   import   protected   throws  
     break   else   instanceof   public   transient  
     byte   extends   int   return   true  
     case   false   interface   short   try  
     catch   final   long   static   void  
     char   finally   native   super   volatile  
     class   float   new   switch   while  
     continue   for   null   synchronized  
     default   if   package   this
2. C에서 많이 쓰이던 sizeof operator가 없다.
3. NULL, TRUE, FALSE와 같이 에서 keyword는 아니지만 #define 문으로 정의해서 사용하던 단어들도 소문자이다.
4. goto, const는 실제 구현은 되어있지 않으나 keyword로 등록되어 있다.

**identifier**

1. java identifier는 class name, method name, variable name, label등에 사용된다.
2. 첫번째 문저는 숫자가 아닌 단어로 시작해야 한다.
   * 특수 문자도 첫번째 문자로 올 수 있다.
3. 자바 keyword는 identifier로 사용할 수 없다.
   * keyword를 포함하는 단어는 사용할 수 있다.
4. 공백을  포함할 수 없다.
5. unicode를 지원하기 때문에 한글도 사용은 가능하다.
6. maximum length에 대한  제한이 없다.

**coding convention**

1. class & interface
   * 첫번째 문자 : 대문자
   * 단어와 단어의 연결 : 뒷 단어가 대문자로 시작
   * 명사
2. method & variable
   * 첫번째 문자 : 소문자
   * 단어와 단어의 연결 : 뒷 단어가 대문자로 시작
   * method는 동사, variable는 명사
3. constant
   * Primitive Data Type constant
     + 모두 대문자
     + 단어와 단어의 연결 : \_(underscore)로 연결
   * Reference Data Type constant
     + 대소문자 다같이 사용

**Data type**

1. Primitive Data Type
   * boolean, char, byte, short, int, long, float, double등 8개가 있다.
     + byte : 8bit (-128[-2^7]~127[2^7-1])
     + short : 16bit ( -2^15 ~ 2^15-1 )
     + int : 32bit (-2^31 ~ 2^31-1 )
     + long : 64bit (-2^63 ~ 2^63-1 )
     + float : 32bit 크기의 실수
     + double : 64bit 크기의 실수
     + boolean : true, false
     + char
       1. single quotes에 둘러쌓인 문자로 표현됨 (ex: 'e', 'g', '1'... )
       2. '\n', '\r', '\t', \u88ab' 형식 지원 (마지막은 유니코드 표현)
   * 실제 값을 갖는다.
   * 위의 Data Type은 이에 대한 정보가 Compiler에 내장되어 있어 선언과 함께 정의된 만큼의 메모리를 생성하게 된다.
2. Reference Data Type
   * 값이 저장되어 있는 메모리의 주소를 갖는다.
   * Type : Array, class, interface
   * 그때 그때마다의 정의에 따라 메모리 할당에 필요한 정보가 달라지기 때문에 선언과 생성이 분리되어 있다.
     + new operator를 사용하여 heap으로 부터 할당해서 사용할 수 있다

|  |
| --- |
|  |
| import java.util.\*;  class TestVar {      public static void main(String[] args) {          String msg;   //라인 1         msg = new String("sample 1"); //라인 2      }  } |
|  |

* + - 1번은 선언을 한 경우이며 이 경우는 메모리가 할당되지 않는다. 2의 라인에서 메모리가 할당된다.
    - 위의 선언은 new를 생략하고 간단히  "sample 1"이라고 해도 생성이 된다. 내부적으로는 new 연산이 적용이 된다.
  + array의 경우 [] 기로를 통해 해당 변수가 arrayType이라는 것을 알려주어야 하며, array는 특정 타입의 element들을 연속적으로 갖는 것이기 때문에 반드기 그 element의 type이 무엇인지 알려주어야 한다.

|  |
| --- |
|  |
| int intArray[];  String msgArray[], str; |
|  |

* + - []안에 절대 size 정보를 주면 안된다. 선언은 단지 어떤 타입의 array라는 것을 알려주는 것이므로 메모리가 할당되지 않는다.
    - String의 변수를 갖는 msgArray는 배열로 선언되었으나 str은 단지 String class type 변수이다.
    - array에 실질적인 메모리를 할당하기 위해서는 new 연산자가 사용되게 되며 이 때 size도 지정을 한다.

|  |
| --- |
|  |
| int intArray[] = new int[100]  double [] dArray, otherArray; dArray = new double[100]; otherArray = new double[200] |
|  |

* + - []가 변수 선언시 앞에 있으면 그 뒤에 나오는 변수들은 모두 배열이다.
    - 선언 후 size 지정을 통한 생성에 대한 명시를 하지않고 바로 변수를 할당 할 수 있다. 이 경우 배열의 갯수는 할당된 수 만큼 생성된다.

|  |
| --- |
|  |
| int intArray[] = {100, 200, 300}; |
|  |

* + 다차원 배열의 경우 2차 이상의 배열들은 각각 dynamic하게 size를 가질 수 있다.
  + 초기값
    - byte : 0
    - short : 0
    - int : 0
    - long : 0L
    - float : 0.0f
    - double : 0.0d
    - char : '\u0000'
    - boolean : false
    - Reference Data Type : null

**멤버변수와 지역변수**

* 멤버변수
  1. 해당 class 내에 method와 동등한 위치에 선언된 변수
  2. 해당 class 내 어디서든 참조 가능하고 한 class 안에서는 global하게 값이 유지된다.
  3. 단. static으로 선언된 method에서는 non static인 멤버 변수를 access 할 수 없다.
  4. 멤버 변수의 경우 선언된 후 별도로 초기값이 지정되지 않으면 선언한 에따라 default 값을 초기값으로 갖게 된다.
* 지역변수
  1. 해당 Class의 method 내에 선연된 변수
  2. 해당 method가 호출되었을 때 stack에 생성되었다가 수행이 완료되면 없어진다.
  3. 지역변수는 초기값이 자동적으로 주어지지 않는다. 따라서 초기화를 하지 않고 호출하면 지역 변수는 전부 error로 처리합니다.

위와 같은 차이가 나는 이유는 {}로 싸여진 변수의 범위가 다르기 때문이다.

{} 내에서 선언된 지역 변수는 {} 지역을 벗어나면 자동으로 삭제된다.

**매개변수 전달방식**

매개변수란?

1. method를 호출할 때 메소드 내로 값을 전달하기 위한 매개로 선언된 변수.
2. 무제한으로 매개변수를 선언 할 수 있다.
3. 매개 변수의 데이터 형을 맞추지 않으면 method는 호출할 수 없다.

* Primitive Data Type의 변수를 매개변수로 이용하는 경우
  1. call by value방식으로 전달이 된다.
  2. 변수에 할당된 메모리의 주소를 전달하는 것이 아니라 이와 똑같은 정보의 메모리 복사본이 새롭게 만들어져 호출되는 쪽에서 사용하도록 하는 것
* Reference Data Type의 변수를 매개변수로 이용하는 경우
  1. call by ref 방식으로 전달이 된다. (주소값이 복사되므로 실제적인 메모리에 대한 정보가 전달이 됨.)

**연산자**

a++, ++a

아래와 같은 예제가 있다.

|  |
| --- |
|  |
| int a = 1;  int b, c; b = ++a;  c = a++; |
|  |

위의 경우

b는 2,  a에도 2

c는 2, a는 3이 저장이 된다.

++이 앞에 있으면 그 뒤의 변수에 대해 우선 연산이 이루어진다.

++이 뒤에 있으면 그 앞부분의 연산이 먼저 이루어진다.

연산식

* int type보다 크기가 작은 숫자 type들은 연산식을 처리할 때 int로 Type Conversion을 하고 연산식을 처리한다.

따라서 아래의 경우는 compile error가 발생한다.

|  |
| --- |
|  |
| byte b1 = 64;  byte b2 = 4;  byte b3 = b1 \* b2; |
|  |

이유는 b1 \* b2의 결과가 int값의 결과값을 리턴하게 되는데 이 결과값을 byte type인 b3에 대입하려고 하기 때문이다.

이런경우 type Conversion을 해주어야 한다.

|  |
| --- |
|  |
| byte b1 = 64;  byte b2 = 4; byte b3 = byte(b1 \* b2); |
|  |

그러나 위의 경우는 b1과 b2의 곱이 byte의 값을 over했기 때문에 올바른 값이 b3에 들어가지 않게 된다.

* + 연산자는, 만약 한 operand만 Stirng이고 나머지가 String이 아닌 모든 경우(Primitive Data Type, Reference Data Type 모두), String이 아닌 다른 쪽이 무조건 String으로 conversion 된다.

instanceof 연산자

Reference Data Type에만 적용해서 쓸 수 있는 연산자로 그 object의 run-time type을 알아낼 수 있게 해준다.

&, |, &&, || 연산자

두 쌍의 연산자는 각각 같은 역할을 한다.

다만 &&와 ||는 해당 연산자의 왼쪽 조건이 부합되지 않는 조건이면 연산을 끝낸다.

따라서 일반적으로는 &&와 ||의 연산이 좀더 효율적이다.

**Type Conversion**

서로 다른 Data Type을 사용한 연산식은 연산을 실행하기 전, 둘 중 하나를 다른 하나의 Data Type에 맞게 전환한다.

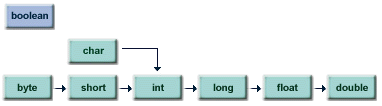
이는 컴파일 때 뿐만이 아니라 run-time일 때에도 올바르게 되었는지 check를 한다.

강제 변환(Explicit Type Conversion)과 수동 변환 (Implicit Type Conversion)이 있다.

강제 변환은 변환될 type을 지정해주어 형을 변환하는 것이다.

수동변환

1. Primitive  Data Type은 컴파일할 때에 이미 Type Conversion이 일어난다.
2. int type보다 크기가 작은 숫자 type들은 연산식을 처리 할 때 int로 Type Conversion을 하고 연산식을 처리한다.
3. Primitive Data Type 중 boolean type을 제외한 모든 다른 type (char, byte, short, int, long, float, double)들은 boolean type으로의 전환이 불가능, (자바에서는 1=true라는 공식이 성립하지 않는다.)
4. 서로 다른 Data Type에 있어 Type Conversion은 보다 더 넓은 Data Type으로 전환이 자동으로 일어난다.
5. Primitive Data Type 간에 있어 자신보다 좁은 type으로의 Implicit Type Conversion은 절대 일어나지 않는다.



1. 위와 같은 크기 관계이지만 char와 short타입은 둘다 16bit로 크기는 같으나 char type은 unsigned인 관계로, 둘 사이는 Implicit Type Conversion이 일어나지 않는다.

**Class의 Type Conversion**

Class 간의 Type Conversion이 일어나기 위해서는 상위 클래스 중에 공통인 클래스가 존재해야 한다.

derived(sub) class의 type으로 선언된 변수가 base class의 type으로 생성된 object를 참조(포인트) 하도록 하는 문장은 일단 컴파일시 무사히 통과가 된다.

그러나 run-time일 때 유요한 Type Conversion인지 확인할 때에는 경우에 따라 다르다.

이를 통과하기 위한 조건은 base class type의 변수가 전환될 type의 class나 전환될 type의 class로부터 파생된 class의 instance를 정확히 갖고 있어야 한다.

**instanceof 연산자**

해당 클래스가 해당 object 타입인지 결과를 true, false로 리턴해준다.

|  |
| --- |
|  |
| [object] instanceof [object type] |
|  |

**Java에서의 modifier들**

선언을 통해 identifier를 compiler에게 알려주는 역할을 할 때 단순히 type 정보뿐만 아니라 그 이외에 좀 더 다양한 정보를 modfier를 통해 알려 줄 수 있다.

자바에서 제공하는 modifier는 크게 access modifier와 그 외 modifier로 나뉘어진다.

* access modifier
  + public, private, friendly, protected
* 그 외 modifier
  + final, static, abstract, native, synchronized, volatile, transient

modifier는 선언문에 쓰여지는데, 반드시 **Data Type 앞 쪽에 와야 한다.**

access modifier

주로 method나 member variables 선언시 사용하는 modifier

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | class 내부 | 같은 package의 다른 class | 다른 package의 상속 class | 다른 package의 다른 class |
| public | O | O | O | O |
| protected | O | O | O | X |
| (friendly) | O | O | X | X |
| private | O | X | X | X |

가장 헷갈리기 쉬운 것이 friendly와 private의 관계인데 friendly는 modifier의 명명자가 아닌 modifier를 지정하지 않았을 경우 기본적으로 적용되는 범위 지정을 뜻한다.

friendly와 private 둘다 다른 package의 상속된 class에서는 사용할 수 없으나

friendly는 같은 package내의 상속된 class는 사용할 수 있다.

그 외의 modifier들

* final
  1. class : class를 상속하는 것은 불가능
  2. method : 해당 method가 존재하는 class를 상속받은 class 내에 해당 method의 verride 불가
  3. member variable : 상수 선언
* abstract
  1. abstract로 정의 해야하는경우
     + class가 한 개 이상의 abstract 메소드를 갖고 있을 때
     + abstract class로부터 파생된 base class에 있는 abstract 메소드를 override로 구현하지 않았을 때
     + class가 interface로부터 파생되었는데 interface에 있는 전체 메소드를 구현하지 않았을 때
     + 해당 class를 상속받은 sub class에서는 객체를 생성할 수 있지만 기존의 class는 객체를 생성하지 않으려는 경우
* static
  1. member variable
     + class의 일반 멤버 변수는 new 연산자를 통해 instance를 만들 때 마다 (즉, object를 생성할 때 마다) 각각 메모리가 할당되어진다.
     + 그러나 class의 어떤 멤버 변수가 static으로 선언되어 있다면 그 멤버 변수는 정의된 class가 load되는 시점에 이미 유일한 메모리를 할당받게 된다.
     + 즉, 하나의 class에 대해 new 연산자를 이용하여 아무리 많은 instance를 생성한다고 해도 static으로 선언된 멤버 변수는 new 연산을 수행하기 전에 이미 메모리를 할당 받았으므로 static으로 선언된 멤버 변수에 대해 object 생성시 추가적인 메모리 할당은 없는 것이다.
     + 이러한 변수를 class 변수라고 한다.
     + 이렇게 static으로 정의된  멤버 변수는, instance와 상관없이 만들어지고 유지되기 때문에 자바에서 class wide global 변수로서의 역할을 한다.
  2. method
     + 해당 메소드가 각 클래스마다 새로운 메소드 호출이 아닌 동일한 메소드가 호출되게 되어진다.
     + 따라서 static으로 선언된 method는 instance를 생성하지 않아도 사용이 가능하다.
     + 이와 같은 특성상 주의해야 할 사항이 있다.
       1. static으로 선언된 메소드에서는 non static인 멤버 변수를 access할 수 없다. 일반적인 멤버 변수는 new 연산자에 의해 instance가 생성될 때에야 비로소 메모리에 할당 받기 때문이다.
       2. this란 class안에서 자신의 reference를 의미하는 keyword이다. 따라서 this는 생성된 instance가 없으면 존재할 수 없으므로 static으로 선언된 method에서는 this라는 keyword를 사용할 수 없다.
       3. static으로 선언된 메소드를 파생된 class에서 override를 통해 일반 메소드로 바꿀 수 없다.
  3. static initializer block
     + block  '**{**' 와 '**}**' 앞에 static modifier를 붙인 형태를 말한다.
     + class 내에 위치하며 method 외에 위치한다.
     + method처럼 호출될 수 있는 형태가 아니며 class가 load되는 시점에 한번 실행되게 되어있다. 즉 instance 생성과 무관하다.
     + static initializer block이 여러 개 존재한다면 class가 load되는 시점에 나타나는 순서대로 차례로 실행이 된다.
     + 실행 순서
       1. 멤버 변수
       2. static initializer block
       3. main method 호출
          1. main method에서 자신의 instance 생성시 또는 외부 class에서 해당 class의 instance 생성시 constructor 호출
  4. native modifier
     + method를 선언할 때 사용
     + native로 선언된 메소드는 abstract처럼 선언부만 가지며 이 메소드의 body 부분은 통상 자바가 아닌 다른 언어(주로 C언어)로 만들어진다.
     + 이러한 개념은 platform dependent한 코드 부분을 자바와 연결시키는 방법으로 많이 이용되고 있다.
  5. transient modifier
     + Java에는 serialize를 하게 해주는 ObjectInputStream과 ObjectOutputStream 같은 특별한 stream이 있다. 만약 어떤 메소드가 serializable 하다면, ObjectOutputStream이 제공하고 있는 writeObject 메소드를 이용하여 이 class의 object 상태를 persistent한 위치에 object 단위로 저장할 수 있다. transient modifier는 이러한 과정중에 serializable한 변수에서 제외 시키는 기능을 한다.

|  |
| --- |
|  |
| class Account implements Serializable {      String name;      int balance;      transient String passwd;  } |
|  |

* + - 위와 같은 경우 name과 balance 변수는 멤버변수로서 그 변수가 직렬화(serializable) 과정에 저장이 되지만 passwd는 저장이 되지 않는다.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | class | method | member variable | local variable |
| public | O | O | O | X |
| protected | X | O | O | X |
| (friendly) | O | O | O | X |
| priavte | X | O | O | X |
| final | O | O | O | O |
| abstract | O | O | X | X |
| static | X | O | O | X |
| native | X | O | X | X |
| transient | X | X | O | X |
| synchronized | X | O | X | X |

modifier 사용대상

**조건 분기문**

자바에서는 if구문을 사용할 때 true = 1, false = 0으로 인식을 하지 않으므로 true와 false에 대한 결과값에 대해 명시를 해주어야 한다. 따라서 다음과 같은 구문은 오류가 발생한다.

|  |
| --- |
|  |
| int count = 0;  if (count) {      System.out.println("true state");  } |
|  |

위와 같은 구문은 아래와 같이 수정해주어야 한다.

|  |
| --- |
|  |
| int count = 0;  if (count != 0) {      System.out.println("true state");  } |
|  |

switch 구문은 다음과 같다.

|  |
| --- |
|  |
| switch (expression) {      case value1 : 문장(들);          break;      case value2 : 문장(들);          break;      case valueN : 문장(들);          break;      default : 문장(들);          break;  } |
|  |

for 구문은 다음과 같다.

|  |
| --- |
|  |
| for (init\_expr; boolean expr; alter\_expr)      문장 or block (중괄호{}로 감싸인 여러 문장들); |
|  |

whlie 구문은 다음과 같다.

|  |
| --- |
|  |
| while (boolean\_exp)      문장 or block (중괄호{}로 감싸인 여러 문장들); |
|  |

do whlie 구문은 다음과 같다.

|  |
| --- |
|  |
| do 문장 or block (중괄호{}로 감싸인 여러 문장들);      while (boolean\_exp); |
|  |

while 문과 do while문의 차이는 while 문은 boolean\_exp의 조건에 따라 한번도 실행되지 않을 수 있는 반면 do while문은 무조건 실행이 된 후 조건을 비교하게 된다는 점이다.

이외의 구문으로

break문은 현재 위치에서 가장 가까운 block을 벗어나기 위해 사용되는 문장이며

continnue 문은 자신이 포함된 반복문의 시작위치로 jump하여 새롭게 조건을 비교하도록 하고자 할 때 사용된다.

**예외처리**

자바에서는 프로그래밍에서 발생할 수 있는 문제를 두가지로 구분하여 처리를 한다.

1. 화면에 잘못된 메세지를 보내주소 실행을 멈출 만큼 프로그램 자체에 문제가 있는 것은 **error**로 분류.
   * Error class에서 파생된 class 형태로 error를 처리. (OutOfMemoryError, StackOverflowError 등)
   * error에 대한 처리코드가 명확하게 정의되어 있지 않아도 컴파일 할 때 문제삼지 않음
2. 문제가 발생될 것이 예측되어 프로그램 과정 중에 잡아낼 수 있는 문제들은 **exception**(예외)로 간주
   * exception또한 class로 다루어진다.
   * 자바의 모든 예외 class들은 java.lang package의 Exception class에서 파생되며 java.lang package가 자동으로 import되며 Exception class도 자동 import 된다.

Exception의 종류

1. RuntimeException class에서 파생된 예외들
   * 주로 프로그래머의 부주의로 발생되는 예외 (array bound를 벗어나거나 null pointer를 사용하거나 등등...)
2. Exception class에서 파생된 예외들
   * 주로 사용자가 잘못 사용할 수도 있기 때문에 발생하는 예외 (파일명을 잘못 입력하는 식의 오류)

Exception의 처리

1. try-catch구문에 의해 처리되거나
2. 발생된 메소드의 밖으로 처리를 던질수 있다.
   * 예외 처리를 던진다는 것은 곧 인위적 예외 발생도 가능하다는 말이다.
   * 또한 던질 대상이 될 예외 클래스를 지정할 수 있으므로 사용자 지정 예외 처리 클래스도 존재한다는 말이다.

try, catch, finally 구문

|  |
| --- |
|  |
| try {      구문;  }  catch (ExceptionType1 e) {      복구 routine1;  }  catch (ExceptionType2 e) {      복구 routine2;  }  catch (ExceptionType3 e) {      복구 routine3;  }  finally {      구문;  } |
|  |

* + try 블럭 안쪽의 구문을 실행하는 동안 아무런 예외(exception)이 발생하지 않았다면, catch 블럭은 실행되지 않는다.
  + try 블럭 안쪽의 구문에서 예외가 발생되면 catch문 중 해당 발생된 예외를 잡는 블럭 안으로 흐름이 이동되며 해당 catch 블럭의 구문이 수행이 된 후 다시 흐름이 외부로 이동된다.
  + 예외가 catch에서 중복 해당하는 경우 가장 순서가 위쪽에 나열된 catch 블럭을 수행하며 더이상의 catch 블럭의 처리는 없다. (하나의 예외에 대해서 수행되는 catch 블럭은 하나이다.)
  + finally 블럭은 생략가능하며, 존재하는 경우 예외 발생 여부와 상관없이 무조건 실행이 된다.
    - 혹 발생하지 않는 경우가 있을 수는 있다.
      1. try 블럭을 실행시키는 thread가 dead상태가 되었을 때
      2. try 블럭에서 System.exit()가 호출되어 프로그램이 종료될 때
      3. CPU 전원이 끊겼을 때
      4. finally 블럭이 끝나기 전에 안에서 또 다른 예외가 발생했을 때

인위적으로 예외 발생하기.

위에서 언급한 방법은 프로그램 실행중 try 블럭 내에서 예외가 발생시 catch로 예외 상황에 대한 처리를 맡기는 것인데 객체를 생성한 대상 클래스 내의 메소드에서 예외가 발생한 경우 해당 예외에 대한 처리를 해당 객체를 생성한 클래스와 같은 상위 클래스로 던질 수 있다.

이때 사용하는 구문이 throws, throw 이다.

**throw**는 인위적으로 예외를 발생시키는 keyword이며 만약 throw문이 try문 안에서 사용되었다면 당연히 이에 매칭되는 catch 문에서 throw에 의해 던져진 예외를 잡을 것이다.

그러나 호출된 메소드에서 try, catch구문의 사용없이 throw를 통해 예외를 던졌다면 그 순간 수행이 중단되며, 어디선가 이 예외를 잡아(catch) 처리하는 루틴, 즉 try, catch문이 있는 곳까지 (호출관계가 누적된 stack의 정보를 참조하여) 예외는 상위로 전달이 될 것이다.

이 경우 **throws**는 컴파일러에게 "이 메소드에서는 이런 예외가 throw를 이용하여 던져질 수 있다."는 것을 알려주는 keyword로, 메소드의 선언부에 사용된다.

구문은 다음과 같이 사용한다.

|  |
| --- |
|  |
| class {      method() throws 예외이름 {          throw new 예외이름();      }  } |
|  |

위와 같이 사용하면 해당 예외이름의 예외를 상위 클래스에서 찾아서 처리를 하게 되며 상위 클래스에도 해당 예외처리가 없다면 그 상위로 다시 던지게 된다.

이 경우 만약 중간에 있는 메소드 내에서 실행중 오류가 발생되고 상위로 해당 오류에 대한 exception처리를 던지는 경우 상위 메소드의 exception처리 후 finally가 실행되며 중간에 있는 메소드의 오류 발생 이후의 연산은 처리가 되지 않는다.

**객체지향의 개념**

객체가 가지는 구성요소는 2가지가 있다.

1. 상태 (state, attribute)
   * 객체가 가지고 있는 속성 또는 특성. 객체의 정적인 부분
2. 행동 (behavior, action)
   * 객체가 가지고 있는 기능 또는 행동, 객체의 동적인 부분

이러한 객체를 소프트웨어적으로 표현하기 위한 방법 중의 하나가 객체지향방법이며, 이 객체지향방법에서 나타내는 몇가지 특징들을 살펴보면 다음과 같다.

* 캡슐화 (Encapsulation)
  + 상태 정보를 저장하고 있는 변수와 상태를 변경하거나 서비스를 수행하는 메소드를 하나의 소프트웨어 묶음으로 캡슐화한다.  
    캡슐화는 소프트웨어의 개발자에게 높은 모듈성(modulatiry)과 정보은닉(information hiding) 등 두가지 이득을 제공해 준다.
  + 모듈성이 높아지면 다른 객체의 내용 변경과 연관이 적어져서 재사용성과 유지보수성을 높일 수 있다.
  + 이러한 캡슐화된 부분은 접근 권한을 어떻게 부여하느냐에 따라서 다른 객체가 접근할 수 있는지를 결정하게 된다.
* 다형성 (Polymorphism)
  + 하나의 객체가 여러 형태로 나타날 수 있거나 여러개의 class가 같은 메세지에 대해서 각자의 방법으로 작용함을 의미한다.  
    즉, 같은 이름을 갖는 여러가지 형태가 존재한다는 것이다.
  + 자바에서는 연산자 overloading이 없으며 메소드 overloading과 메소드 overriding을 제공한다.
  + 자바에서는 아직 형태가 결정되지 않은 interface를 두고 이 interface를 상속받아 다른 형태가 나타나도록 하고 있다.
* 상속 (Inheritance)
  + 하나의 class로 캡슐화된 정보가 다른 class에서도 사용되어야 한다면 새롭게 정의할 class는 기존의 class에 포함된 정보와 중복된다.  
    이렇게 중복되는 정보는 새롭게 정의하는 class에서 기존의 정의된 class에 포함된 정보를 이용함으로써 그 정보를 가지게 되는데 이것을 상속이라 한다.
  + 즉 상속은 새롭게 정의할 class에서 이미 존재하는 class의 속성을 그대로 가질 수 있으며 이를 바탕으로 필요한 속성만 추가하여 정의하는 방법이다.  
    이 때 새로 정의되는 class를 sub class, 기존의 클래스를 super class라고 한다.

**class의 개념**

class란 어떤 특정 종류의 모든 객체들에 대해 일반적으로 적용할 수 있는 변수와 메소드를 정의하고 있는 소프트웨어적인 설계도 (blueprint) 또는 프로토타입 (prototype)이라 할 수 있다.

다시 말해서, 실세계에 존재하는 객체들이 가질 수 있는 상태와 행동들에 대해 소프트웨어적으로 추상화 (abstraction) 해 놓은 것이 class 이다.

객체가 갖는 상태를 변수로 정의하고, 행동을 메소드로 정의할 수 있다.

그리고 이를 하나의 묶음으로 캡술화 하기 위해 "class { }"와 같이 정의할 수 있다.

이렇게 정의된 class르르 실제 객체로 생성할 수 있고, 객체를 선언하는 방법은 기본 데이터 형에 대한 변수를 선언하는 것과 같다.

다시 말해서 class형에 대한 변수를 선언한다고 생각할 수 있다.

자바에서 객체생성은 다음과 같다.

|  |
| --- |
|  |
| class이름 객체이름 = new class이름();  또는  class이름 객체이름;  객체이름 = new class이름(); |
|  |

자바에서 class에 대한 객체 (인스턴스)를 선언하고 생성하기 위해

1. class에 대한 변수를 선언함과 동시에 생성하는 방법
2. class에 대한 변수를 선언하고 나중에 필요할 때 객체 (인스턴스)를 생성하는 방법

두가지 방법이 있다.

자바에서는 이러한 class에 대한 변수를 참조형 변수라 하고, 참조형은 반드시 new 연산자를 이용하여 메모리 공간을 할당해 주어야 한다.

자바에서 말하는 class란 object를 만들어내기 위한 틀이다.

class는 element의 집합을 정의(선언)한 것이고, 이러한 정의에 근거하여 메모리를 할당받은 것이 object이다. 틀이 아니고 틀에 의해 만들어진 실체가 object인 것이다.

보통 class를 표현하는 명사나 형용사(attribute)는 멤버 변수로, 동사(behavior)는 메소드로 class 내에 표현이 된다.

**class들 간의 관계**

1. 상속의 관계 ("is a" 관계)
   * 상속의 관계인 두 class에 있어, sub class의 object를 생성하면 extends 된 super class의 object도 함께 생성되는 것이다.
2. 포함의 관계 ("has a" 관계)
   * 포함의 관계인 경우 포함을 하고 있는 클래스의 객체가 생성이 된다고 하더라도 이에 포함된 class의 object가 만들어지는 것은 아니다.
   * 포함의 관계에서는 상위 클래스가 호출시 포함을 시킬 클래스의 object를 생성하는 형식으로 객체 내부에 포함되는 객체를 생성해준다.

**this와 super**

this는 object 안에서 자신을 참조할 때 사용

super는 상위 class의 object를 참조할 때 사용

위의 두 keyword는 생성된  object 내에서 사용되는 것이기 때문에 static으로 선언된 메소드에서는 사용될 수 없다.

1. this는 객체 자신에 대한 참조값을 가지고 super는 객체를 상속해준 super class 타입의 객체에 대한 참조값을 가진다.
2. 메소드 내에서만 사용된다.
3. 매개변수와 객체 자신이 가지고 있는 변수의 이름이 같을 경우 또는 객체 자신의 변수 이름과 객체의 superclass로부터 상속받아 가진 변수의 이름이 같을 때, 이를 구분하기 위해 자신이 가지고 있는 변수앞에 this나 super를 사용한다.
4. 객체 자신이나 super class 타입의 객체에 대한 참조값을 ㅡ메소드에 전달하거나 리턴해주기 위해 사용하기도 한다.
5. this나 super를 사용함으로써, 모호하지 않고 좀더 명확한 프로그램을 작성할 수 있다.

**package 선언**

1. class들을 하나의 묶음(그룹) 단위로 구성한다.
2. class들을 묶음 단위로 제공하여, 필요할 때만 사용할 수 있게 한다.
3. class 이름의 혼란을 막아서 충돌을 방지한다.
4. package 이름과 함께 class 이름을 사용함으로써 class를 효율적으로 관리할 수 있다.
5. class를 관련이 있는 것끼리 묶어놓음으로써, 필요한 class의 식별이 용이하게 한다.
6. package 단위의 접근 권한을 지정할 수 있고, 이를 통하여 class만 있는 경우보다 더욱 다양한 접근 권한을 제어할 수 있다.
7. 하나의 class는 하나의 package에 속해야 한다.

**자바표준 패키지 : 자바 API**

자바는 J2SDK에서 정의된 많은 연관된 class들을 package로 묶어 제공하여, 필요에 따라 사용자가 import하여 사용할 수 있다.

이러한 미리 제공된 class package들을 자바 API (Java Application Programming Interface)라 한다.

자주 사용하는 자바 API Package는 다음과 같다.

* java.applet
  + Java Applet Package. 애플릿 제작 및 브라우저와 애플릿의 상호작용 등을 제공하는 Applet class와 interface를 포함하고 있다.
* java.awt
  + Java Abstract Windowing Toolkit Package. 그래픽 사용자 인터페이스의 제작과 관리에 필요한 모든 class와 interface를 포함하고 있다.
* java.awt.event
  + Java AWT Event Package. 이벤트 처리를 해주는 class와 interface를 포함하고 있다.
* java.io
  + Java Input/Output Package. 데이터를 입력받고 출력할 수 있도록 하는 class를 포함하고 있다.
* java.lang
  + Java Language Package. 이 package는 특별히 지정하지 않아도 자동으로 import된다. 자바 프로그램들이 기본적이니 언어차원에서 필요로 하는 class와 interface를 포함하고 있다.
* java.sql
  + Java Structured Query Language Package. 프로그램이 데이터베이스와 정보를 교환하기 위한 class와 interface를 포함하고 있다.
* java.util
  + Java Utilities Package. 날짜/시간 조작, 난수 발생, 문자열의 토큰화 등과 관련된 각종 유틸리티 class와 interface를 포함하고 있다.

**Method overloading**

Method overloading은 매개변수의 type이나 개수를 다르게 하여, method 이름이 같은 또다른 메소드를 정의할 수 있는 개념이다.

실제로 method 이름이란 프로그래머에게 보여지는 것이고, 내부적으로는 package명, class명, 매개변수 type, 매개변수 개수 등의 정보들이 method를 구별짓는데 영향을 주게 되어있다.

따라서 엄밀히 말하면 단순히 이름이 똑같다고 똑같은 함수는 아닌 것이다.

하지만 return type이나 modifier의 종류, exception 선언등과 같은 것들은 method를 구별짓는데 아무 영향도 줄 수 없기에, 다른 것은 다 똑같으며 이것들이 달라진다는 것은 overloading라고 할 수 없다.

method overloading의 규칙은 다음과 같다.

1. overloading이 된 메소드를 호출할 때 넘겨주는 매개변수 값에 의해 widen promotion이 일어난다.
2. return type, access modifier, exception 선언은 overloading에 전혀 영향을 미치지 않는다.

**Method overriding**

파생된 class는 이미 super (base) class가 가지고 있는 method들을 상속받아서 가지고 있게 된다.

이러한 상황에서, super (base) class에 정의된 method와 동일한 이름의 method를 파생된 class에서 재정의할 수 있게 하는 것이 바로 메소드 overriding이다.

이 때 재정의되는 method는 (overloading과는 달리) return type과 매개변수 갯수, type들의 선언부를 완전히 똑같이 만들어야만 한다.

이 개념은 super class에서 정의해놓은 method의 body(구현) 부분으르 파생된 class에서 다시 정의하여 사용하겠다는 것이다. 즉, super class 코드를 바탕으로 새로운 기능을 추가하거나, 수정할 수 있게 해주는 것이다.

method overriding의 규칙은 다음과 같다.

1. access modifier는 파생된 class에서 access 범위가 넓어지는 쪽으로는 변경이 가능하다. 하지만 좁혀지게는 재정의 할 수 없다.
   * private < (friendly) < protected < public
2. 더 많은 예외(exception)을 던질 수 없다. 즉 줄이거나 같게만 가능하다.

**Constructor (생성자)**

생성자는 이미 method 이름이 정해져 있는 method(class 명과 동일)이며, 다른 method처럼 class의 object가 만들어진 후 필요에 따라 언제든지 호출할 수 있는 것이 아니라, object가 만들어지는 순간 딱 한번만 수행되는 method이다.

주로 object 멤버 변수들의 초기화에 관련된 작업을 수행한다.

constructor를 선언하지 않은 경우 default constructor가 수행이 되며 이 default constructor는 아무런 동작을 하지 않는다.

constructor overriding

만약 constructor가 선언이 되어있고 해당 constructor의 매개변수와 호출한 object의 매개변수의 type이나 개수가 다른 경우 error가 발생된다.

이런 경우 constructor도 overloading이 가능하므로 this()로 다른 constructor를 호출하는 것이 가능하다. 이에 대한 예제는 아래와 같다.

|  |
| --- |
|  |
| class Employee {      String name;      float salary;      Employee(String name, float f) { //constructor          this.name = name;          this.salary = f;      }      Employee() { //constructor          this("unknown", 0.0f);      }      //...  } |
|  |

위와 같이 Employee class를 정의한 경우

|  |
| --- |
|  |
| Employee someone = new Employee(); |
|  |

와 같은 문장에 의해

|  |
| --- |
|  |
| Employee("unknown", 0.0f); |
|  |

가 호출이 된다. 이 때, this()는 항상 첫번째 문장으로 불려져야 한다.

부모 클래스의 constructor 호출

파생된 object가 생성될 때 base class의 object도 같이 생성이 된다. 이러한 작업에 사용되는 것이 super()이다.

super는 원래 참조형 상수이지만 constructor를 호출하는 메소드처럼 사용할 수 있다.

파생된 object가 생성될 때, 파생된 class의 constructor에서 특별한 지정이 없었다면, base class의 object를 생성하기 위한 base class의 default constructor가 호출되어진다.

이 때에도 상위 클래스의 constructor의 호출에 대한 언급이 없다면 기본적으로 상위클래스의 default constructor(매개 변수가 없는 constructor)를 호출하게 되므로 해당 constructor가 있는지, 또는 원하는 상위 클래스의 constructor를 호출하였는지에 대한 확인이 필요하다.

이때 상위 constructor를 호출할 때 사용하는 것이 super(); 이다.

**member 변수의 초기화 순서**

new 연산자에 의해 어떤 class의 object가 만들어 질 때

1. object 크기 만큼 memory가 할당된다.
2. 할당되면서 멤버변수들에게 초기값을 주지 않았을 때 갖는 default 값이 대입된다.
   * 만약 이 object가 파생된 class에 대한 것이었다면, base class의 object도 만들어지며 위 1, 2번과 같은 진행을 거친 후, base class 부터 다음과 같은 방식으로 초기화가 진행이 된다.
     + 맴버 변수를 선언하면서 지정한 초기값이 있을 경우 이 값이 대입된다.
     + 지정된 constructor에 의해 초기화가 발생된다.
3. 멤버 변수를 선언하면서 지정한 초기값이 있을 경우 이 값이 대입된다.
4. 지정된 constructor에 의해 초기화가 발생된다.

즉, 최종적으로는 constructor에 의해 초기값이 영향을 받게 된다.

**Inheritance (상속)**

모든 객체지향 기반의 환경에서는 class를 기반으로 하는 프로그래밍이 이루어지며 각 class간의 관계는 상속과 포함으로 이루어진다고 하였다.

전혀 새로운 독립적인 class를 무조건 만들어내기보다는 기존에 존재하는 class 중 상속("is a") 관계를 갖는 class를 찾아서, 그것을 바탕으로 기본속성은 상속받고 새로운 기능만 파생된 class에 추가해서 넣는다면, 보다 효율적으로 system을 구축할 수 있다.

즉, 상속은 객체지향에서 주장하는 code의 재사용성을 제공해준다.

자바의 class들은 run-time에 동적으로 binding 되기 때문에 (base class와 derived class가 다른 파일에 저장되어 있을 경우) bass class에 새로운 멤버 변수나 메소드를 추가하거나 기존 코드를 변경시킨다 해도, 이미 컴파일 되어 있던 파생 class를 다시 컴파일 시키지 않아도 된다. (왜냐하면 run-time에 파생된 class들이 메모리를 할당받을 때 변경된 base class도 메모리를 할당받게 되므로 변경된 정보를 그대로 상속받을 수 있는 것이다.)

이것은 maintenance와 reliability라는 측면에서도 커다란 개선 효과를 가져다 준다.

자바에서 상속을 명시하는 keyword는 'extends' 이다.

자바는 single inheritance (단일 상속) 만을 허용한다.

**Polymorphism (다형성)**

같은 class type의 object가 동일한 이름의 method를 호출했음에도 불구하고, (상속과 overriding된 method로 인해) 경우에 따라 서로 다른 기능의 method가 동작되는 것을 다형성이라 한다.

이에 관련이 있는 것이 바로 binding 시점인데

* C++, Java와 같은 객체 지향 언어의 경우 run-time binding이 발생
* C의 경우 compile time binding이 발생

위와 같은 시점의 차이가 있다.

만약 아래와 같은 선언이 있다고 하자.

|  |
| --- |
|  |
| A test = new B();  test.method(); |
|  |

C의 경우 compile 단계에서 binding이 일어나므로 선언된 type인 A라는 type으로 메모리가 할당이 되어 A에 존재하는 method()가 호출이 된다.

그러나 C++, Java의 경우 run-time 단계에서 binding이 일어나므로 선언은 A로 되어 있어도 new로 정의된 정보에 대하여 메모리 할당이 발생이 되어 B에 존재하는 method()가 호출이 된다.

결국 종속된 class을 new를 통해 지정해 주게 되면 해당 class의 method가 호출이 되어 다형성이 이루어지게 된다.

다형성은 method에 대해 동작되므로 멤버변수에 발생하지는 않는다.

결국 instance 변수와 class 변수는 재정의가 되어 다형성이 이루어지지 않는 것이다.

또한 instance 변수와 class 변수, class method가 은닉된 경우는 재정의 될 수 없다.

**Heterogeneous collection class**

array 배열을 이용해 class의 선언도 가능하다.

이는 다형성이 가능하기 때문에 가능한 것이기도 하다.

|  |
| --- |
|  |
| GeoShape list[] = new GeoShape[5];  list[0] = new Rectangle();  list[1] = new GeoShape();  list[2] = new Circle();  list[3] = new Rectangle();  list[4] = new GeoShape(); |
|  |

위와 같이 배열로 부모 class 객체에 대해 선언을 하고 각 배열을 해당 class의 자식 class로 생성을 해줄 수 있다.

마찬가지로 배열의 특성을 이용해 해당 class들의 method를 호출 할 수 있다.

|  |
| --- |
|  |
| for(int i=0; i < list.length; i++) {      list[i].method();  } |
|  |

만약 자식 class 내에 해당 method()가 존재하지 않는 경우 에러가 발생하게 된다. 이런 경우를 부모 class에서 해당 method()를 abstract method로 선언을 하고 자식 class에서 구현을 하는 식으로 정해놓으면 이러한 문제가 발생할 염려가 없다.

이것처럼 불특정한 대상들을 묶어서 관리하기 위한 interface를 자바에서는 제공해주고 있다.

|  |  |  |
| --- | --- | --- |
| interface | 특징 | 구현 class |
| Set | 데이터 순서와 무관  데이터의 중복 불허 | HashSet, TreeSet |
| List | 데이터의 순서 관리  데이터의 중복 허용 | ArrayList, LinkedList, Vector |
| Map | 키(key)와 값을 묶어서 관리  키(key)의 중복 불허 | HashMap, TreeMap, Hashtable |

각 interface에 정의되어 있는 주요 method는 다음과 같다.

Set

|  |  |
| --- | --- |
| size() | Set의 요소 개수를 리턴 |
| isEmpty() | Set이 비어있는지 여부 리턴 |
| contains() | Set에 특정 원소가 포함되어 있는지 여부 리턴 |
| add() | Set에 특정 원소를 추가 |
| remove() | Set에서 특정 원소를 제거 |

List

|  |  |
| --- | --- |
| get() | 지정한 위치에 있는 데이터를 리턴 |
| set() | 지정한 위치에 있는 데이터를 변경 |
| add() | 지정한 위치에 데이터를 추가 |
| remove() | 지정한 위치에서 데이터를 제거 |
| indexOf() | 지정한 데이터가 있는 처음 위치를 리턴 |
| lastIndexOf() | 지정한 데이터가 있는 마지막 위치를 리턴 |
| subList() | 지정된 범위에 해당하는 부분 List를 리턴 |

Map

|  |  |
| --- | --- |
| put() | 키(key)와 값으로 구성된 새로운 데이터를 추가 |
| get() | 지정한 키(key)에 해당하는 데이터를 리턴 |
| remove() | 지정한 키(key)에 해당하는 데이터를 삭제 |
| containsKey() | 지정한 키(key)가 존재하는지 여부를 리턴 |
| containsValue() | 지정한 값이 존재하는지 여부를 리턴 |
| size() | Map의 요소 개수를 리턴 |
| isEmpty() | Map이 비어있는지 여부를 리턴 |

Vector class

array를 생성할 때, 그 길이가 정해지고, 저장하려는 원소(element)의 형태가 같아야 한다는 단점이 있다.

vector는 다양한 형의 원소를 저장 가능하며, 그 길이 또한 가변적이라는 장점을 가지고 있다.

vector의 주요 method는 다음과 같다.

* void add(int index, Object element): 객체를 해당 index에 추가합니다.
* boolean add(Object o): 객체를 현재 Vector의 끝에 추가합니다.
* boolean addAll(Collection c): 주어진 Collection의 모든 객체를 추가합니다.
* boolean addAll(int index, Collection c) : 주어진 Collection의 모든 객체를 index 위치에 추가합니다.
* void addElement(Object obj): 객체를 추가합니다.
* int capacity(): 이 객체의 용량을 얻습니다.
* void clear(): Vector 내의 모든 객체를 제거합니다.
* boolean contains(Object elem): 객체를 포함하고 있는지를 얻습니다.
* boolean containsAll(Collection c) : Collection 내의 객체들을 포함하고 있는지를 알아봅니다.
* Object elementAt(int index): 해당 index에 있는 객체를 얻습니다.
* Enumeration elements(): 이 Vector에 대한 Enumeration 객체를 가져옵니다.
* Object firstElement(): 첫 번째 객체를 가져옵니다.
* Object get(int index): 해당 index 위치에 있는 객체를 가져옵니다.
* int indexOf(Object elem): 객체의 위치를 가져옵니다.
* int indexOf(Object elem, int index) : 해당 index 위치에서로부터 해당 객체가 처음 나오는 index를 가져옵니다.
* void insertElementAt(Object obj, int index): 해당 index에 객체를 삽입합니다.
* boolean isEmpty(): Vector가 비어있는지를 알 수 있습니다.
* Object lastElement(): Vector내의 마지막 객체를 가져옵니다.
* int lastIndexOf(Object elem): Vector 내에서 객체의 마지막 index를 얻습니다.
* int lastIndexOf(Object elem, int index) : 해당 index로부터 마지막의 객체 index를 가져옵니다.
* Object remove(int index): 해당 index의 객체를 제거합니다.
* boolean remove(Object o): 객체를 제거합니다.
* boolean removeAll(Collection c): Collection에 있는 모든 객체를 제거합니다.
* void removeAllElements(): Vector 내의 모든 객체를 제거합니다.
* boolean removeElement(Object obj): 객체를 제거합니다.
* void removeElementAt(int index): 해당 index의 객체를 제거합니다.
* boolean retainAll(Collection c): Collection에 있는 객체들을 보존합니다.
* Object set(int index, Object element): 해당 index의 객체를 대체합니다.

Stack class

Stack은 LIFO(Last In First Out) 방식으로 객체를 추가, 삭제하는 자료구조이다.

Vector 클래스를 상속하는 class로서, stack의 고유 기능인 push, pop, top, empty등과 같은 연산기능을 가진 method를 제공해준다.

stack의 주요 method는 다음과 같다.

* boolean empty(): Stack이 현재 아무 것도 저장되어 있지 않은지를 검사합니다.
* Object peek(): Stack의 top에 있는 원소를 가져옵니다.
* Object pop() : Stack의 top에 있는 원소를 가져오고 top의 위치를 하나 감소시킵니다.
* Object push(Object item): Stack에 새로운 원소를 저장합니다.
* int search(Object o): 원소의 위치를 가져옵니다.

Enumeration Interface

여러개의 원소들을 포함하는 객체에 대해 정의하고 있는 interface이다. Vector에서 처럼 각 원소에 접근할 때 그 원소의 데이터 형으로 전환하는 코드가 없어도 된다.

Enumeration interface에서 제공하는 주요 method는 다음과 같다.

* boolean hasMoreElements() : 더 이상 원소가 존재하는지를 검사한다.
* Object nextElement() : Enumeration 내의 현재 원소 다음의 원소를 가져온다.

다음과 같이 사용한다.

|  |
| --- |
|  |
| for (Enumeration e = v.elements() ; e.hasMoreElements(); ) {      System.out.println(e.nextElement());  } |
|  |

Iterator interface

Enumeration interface를 대체하기 위한 것으로 주요 method는 다음과 같다.

* boolean hasNext() : 더 이상의 원소가 존재하는지를 얻는다.
* Object next() : 현재 위치 다음에 있는 원소를 가져온다.
* void remove() : Iterator 내의 마지막 원소를 제거한다.

다음과 같이 사용한다.

|  |
| --- |
|  |
| for (Iterator i = v.iterator(); i.hasNext(); ) {      System.out.println(i.next());  } |
|  |

**Interface**

프로그래밍 과정중에 만들어지는 method들의 정의를 표준화 시키기 위한 툴

class와 동일하게 멤버 변수와 메소드를 갖는 구조로 선언이 된다.

class와의 차이점

1. 모든 member variable은 public static final로 선언되어야만 한다.
   * interface의 member variable은 단지 상수로만 사용이 된다.
   * 선언을 해주지 않아도 자동으로 public static final로 선언이 된다.
2. 모든 method는 public abstract로 선언되어야만한다.
   * 선언을 해주지 않아도 자동으로 public abstract로 선언이 된다.

이렇게 선언된 interface는 (abstract로 선언된 class처럼)

1. 생성문을 통해 object를 만들 수 없으며, 단지 base class처럼 상속을 시켜주는 역할만 한다.
2. 일단 선언되고 나면 class들처럼 data type으로 사용할 수 있다.  
   즉, 일반 변수, 매개 변수, return 값 등의 type으로 사용할 수 있다.
3. interface도 interface로부터 상속되어질 수 있다.
4. implements라는 keyword로 상속을 선언한다.

interface는 선언된 method를 가지고 base class와 같이 상속을 시켜줌으로써, 관계없는 class간에 method를 overriding 하여 사용할 수 있게 해준다.

interface는 abstract로 선언된 class가 제공하지 못하는 다중 상속(multiple inheritance)를 지원한다.

**inner class**

inner class는 크게

* 멤버의 속성을 갖는 inner member class
* method 안에 정의하는 inner area class
* 이름을 생략할 수 있는 anonymous inner class

로 나누어 볼 수 있다.

inner member class

class 내부에 member와 같은 level로정의되는 class를 말한다.

예제는 다음과 같다.

|  |
| --- |
|  |
| class MyOuter {     private int   outVar = 1;          class MyInner {         int      inVar = 2;         static int  s1 = 3;       // 라인 1.  error !!          int innerMethod() {             inVar = outVar;       // 라인 2.                                   if (this.inVar == MyOuter.this.outVar) // 라인 3.             inVar = 0;              return inVar;         }     }       int outerMethod() {         MyInner inobj = new MyInner(); // 라인 4.          return inobj.innerMethod();       }      public static void main( String   arg[]) {         MyInner tobj = new MyInner();  // 라인 5.  error!!         MyOuter mobj = new MyOuter();         mobj.outerMethod();     } } |
|  |

위의 코드를 compile하는 경우 생성되는 파일

* 1. MyOuter.class : inner member class를 포함하는 class
  2. MyOuter$MyInner.class : inner member class

주의해야할 부분

* + inner member class 내부 member variable나 method는 static으로 선언될 수 없다. 따라서 라인1은 error이다.
  + inner member class에서는 outer class의 모든 member variable이나 method를 access를 할 수 있다.  
    outer class 입장에서 본다면 inner member class 또한 내부에 정의된 구성원이므로, 라인2 와 같이 outer class의 private로 선언된 멤버변수도 access할 수 있다.
  + inner member class 내부의 this는 inner member class를 의미한다.  
    만약 outer class를 참조해야 한다면 super를 사용하는 것이 아니라 OuterclassName.this를 사용한다.  
    라인 3이 이러한 예이다. (그냥 outVar도 가능하지만, 설명을 위해 이렇게 사용했다.)
  + outer class내에서 inner member class의 object를 생성할 수 있는 곳은 outer clas를 가리키는 this가 존재하는 곳이다. (라인 4)  
    즉 this를 사용할 수 없는, static으로 선언된 method에서는 inner member class를 생성할 수 없다. 따라서 라인 5와 같은 static method에서는 에러가 발생한다.
  + 만약 outer class 내에서가 아닌 다른 class에서 inner member class를 생성하고자 한다면, 반드시 outer class의 object를 먼저 생성하고 이 object를 통해서 inner member class를 생성해야만 한다.  
    이런 속성 때문에 inner member class가 Event Listener로 많이 사용된다.

|  |
| --- |
|  |
| MyOuter myout = new MyOuter();  MyInner myin = myout.new MyInner();  또는  MyInner myin = new MyOuter().new MyInner(); |
|  |

* + inner member class는 top level class와 달리 static, private, protected가 될 수 있다.  
    만약 inner member class가 static으로 선언되었다면 이 inner member class안에서는 outer class의 instance member(즉 모든 method와 variable)를 access할 수 없다.  
    static으로 선언된 inner class를 중첩 class라고 한다. 중첩 class가 포함하고 있는 method에서는 class의 method에서처럼 non-static variable 또는 non-static method에 접근할 수 없지만, class variable 또는 class method에는 접근할 수 있다.

inner area class

method 안에 정의되는 class를 말한다.

inner area class에서는 자신이 속한 method의 지역변수가 final로 선언된 것만 access할 수 있다.

예제는 다음과 같다.

|  |
| --- |
|  |
| public class MyOuter {    int   oVal = 0;     Runnable  oMethod() {           int         지역Val = 200;       final int  fval= 100;        class My지역 implements Runnable {            int  mem지역;            public void run() {              mem지역 = fval;         // 라인 1.                   oVal = 400;              // 라인 2.                       지역Val = 400;          // 라인 3.  error!!           }       }                         My지역 runobj = new My지역(); // 라인 4.               return runobj;    } } |
|  |

inner area class는

* 라인1과 같이 자신의 내부에서, 자신을 둘러싸고 있는 method의 final로 선언된 area variable을 access 할 수 있다.
* 라인2와 같이, 자신을 둘러싸는 method가 속한 class의 member variable을 access 할 수 있다.
* 라인3과 같이 자신의 내부에서, 자신을 둘러싸는 method의 area variable을 access 할 수 없다.  
  왜냐하면 일반 area variable은 method가 return 되면서 사라져 버리기 때문에, method 종료 후에도 살아있을 수 있는 지역 class의 object에서 참조하는 것은 위험하기 때문이다.
* 라인4와 같이 area class를 둘러싸고 있는 method 내에서 area class의 object를 생성할 수 있다.

anonymous inner class

이 형식의 class는 inner area class의 일종으로, clas의 이름이 없는 class이다.

anonymous inner class의 생성과 선언은 다음과 같이 한번에 이루어진다.

|  |
| --- |
|  |
| new 상속받을class또는 interface이름 () {      //body ...  } |
|  |

따라서, anonymous inner class는 constructor가 없다.

예제는 다음과 같다.

|  |
| --- |
|  |
| class MainGUI {     Button b = new Button( "ok" );             void registHandler() {         b.addActionListener(         // 라인 1.             new ActionListener() {    // 라인 2.                 public void actionPerformed(ActionEvent e) {                     System.out.println("pressed ok");                    }             }         );           // end of 메소드 call     }               // end of registHandler                     // ... }     interface ActionListener {     public void actionPerformed(ActionEvent e ); } |