Java의 정석

제 12 장

지네릭스, 열거형, 애너테이션

2017. 8. 23

남궁성 강의

castello@naver.com

1. 지네릭스(Generics)

1.1 지네릭스(Generics)란?

- 컴파일시 타입을 체크해 주는 기능(compile-time type check) – JDK1.5
- 객체의 타입 안정성을 높이고 형변환의 번거로움을 줄여줌
  (하나의 컬렉션에는 대부분 한 종류의 객체만 저장)

지네릭스의 장점

1. 타입 안정성을 제공한다.
2. 타입체크와 형변환을 생략할 수 있으므로 코드가 간결해 진다.

1.2 지네릭 클래스의 선언

- 클래스를 작성할 때, Object타입 대신 T와 같은 타입변수를 사용

```
class Box {
    Object item;

    void setItem(Object item) { this.item = item; }
    Object getItem() { return item; }
}
```

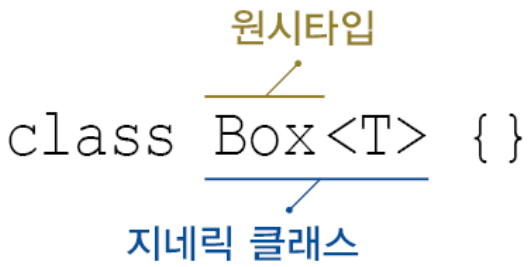
```
class Box<T> { // 지네릭 타입 T를 선언
    T item;

    void setItem(T item) { this.item = item; }
    T getItem() { return item; }
}
```

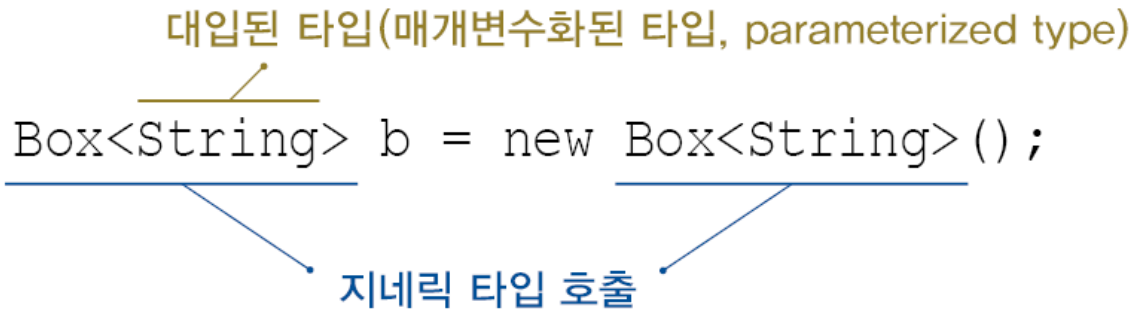
- 참조변수, 생성자에 T대신 실제 타입을 지정하면 형변환 생략가능

```
Box<String> b = new Box<String>();   // 타입 T 대신, 실제 타입을 지정
b.setItem(new Object());             // 에러. String이외의 타입은 지정불가
b.setItem("ABC");                    // OK. String타입이므로 가능
String item = (String) b.getItem(); // 형변환이 필요없음
```

1.3 지네릭스의 용어

Box<T>	지네릭 클래스. 'T의 Box' 또는 'T Box'라고 읽는다.
T	타입 변수 또는 타입 매개변수.(T는 타입 문자)
Box	원시 타입(raw type)





1.4 지네릭스의 제한

- static멤버에는 타입 변수 T를 사용할 수 없다.

```
class Box<T> {
    static T item; // 에러
    static int compare(T t1, T t2) { ... } // 에러
        ...
}
```

- 지네릭 타입의 배열 T[]를 생성하는 것은 허용되지 않는다.

```
class Box<T> {
    T[] itemArr;  // OK. T타입의 배열을 위한 참조변수
        ...
    T[] toArray() {
        T[] tmpArr = new T[itemArr.length]; // 에러. 지네릭 배열 생성불가
        ...
        return tmpArr;
    }
        ...
}
```

1.5 지네릭 클래스의 객체 생성과 사용

- 지네릭 클래스 Box<T>의 선언

```
class Box<T> {
    ArrayList<T> list = new ArrayList<T>();

    void add(T item)             { list.add(item);          }
    T get(int i)                 { return list.get(i);      }
    ArrayList<T> getList()       { return list;             }
    int size()                   { return list.size();      }
    public String toString()     { return list.toString(); }
}
```

- Box<T>의 객체 생성. 참조변수와 생성자에 대입된 타입이 일치해야 함

```
Box<Apple> appleBox = new Box<Apple>(); // OK
Box<Apple> appleBox = new Box<Grape>(); // 에러  대입된 타입이 다르다.
Box<Fruit> appleBox = new Box<Apple>(); // 에러. 대입된 타입이 다르다.
```

- 두 지네릭 클래스가 상속관계이고, 대입된 타입이 일치하는 것은 OK

```
Box<Apple> appleBox = new FruitBox<Apple>(); // OK. 다형성
Box<Apple> appleBox = new Box<>();   // OK. JDK1.7부터 생략가능
```

- 대입된 타입과 다른 타입의 객체는 추가할 수 없다.

```
Box<Apple> appleBox = new Box<Apple>();
appleBox.add(new Apple()); // OK.
appleBox.add(new Grape()); // 에러. Box<Apple>에는 Apple객체만 추가가능
```

1.6 제한된 지네릭 클래스

- 지네릭 타입에'extends'를 사용하면, 특정 타입의 자손들만 대입할 수 있게 제한할 수 있다.

```
class FruitBox<T extends Fruit> {   // Fruit의 자손만 타입으로 지정가능
    ArrayList<T> list = new ArrayList<T>();
    void add(T item)              { list.add(item);          }
    ...
}
```

- add()의 매개변수의 타입 T도 Fruit와 그 자손 타입이 될 수 있다.

```
FruitBox<Fruit> fruitBox = new FruitBox<Fruit>();
fruitBox.add(new Apple());  // OK. Apple이 Fruit의 자손
fruitBox.add(new Grape());  // OK. Grape가 Fruit의 자손
```

- 인터페이스의 경우에도 'implements'가 아닌, 'extends'를 사용

```
interface Eatable {}
class FruitBox<T extends Eatable> { ... }
class FruitBox<T extends Fruit & Eatable> { ... }
```

1.7 와일드 카드 '?'

- 지네릭 타입에 와일드 카드를 쓰면, 여러 타입을 대입가능
  단, 와일드 카드에는 <? extends T & E>와 같이'&'를 사용불가

〈? extends T〉	와일드 카드의 상한 제한. T와 그 자손들만 가능
〈? super T〉	와일드 카드의 하한 제한. T와 그 조상들만 가능
〈?〉	제한 없음. 모든 타입이 가능. 〈? extends Object〉와 동일

```
static Juice makeJuice(FruitBox<? extends Fruit> box) {
    String tmp = "";
    for(Fruit f : box.getList()) tmp += f + " ";
    return new Juice(tmp);
}
```

- makeJuice()의 매개변수로 FruitBox<Apple>, FruitBox<Grape> 가능

```
FruitBox<Fruit> fruitBox = new FruitBox<Fruit>();
FruitBox<Apple> appleBox = new FruitBox<Apple>();
    ...
System.out.println(Juicer.makeJuice(fruitBox)); // OK. FruitBox<Fruit>
System.out.println(Juicer.makeJuice(appleBox)); // OK. FruitBox<Apple>
```

1.6 지네릭 메서드

- 반환타입 앞에 지네릭 타입이 선언된 메서드

```
static <T> void sort(List<T> list, Comparator<? super T> c)
```

- 클래스의 타입 매개변수<T>와 메서드의 타입 매개변수 <T>는 별개

```
class FruitBox<T> {
        ...
    static <T> void sort(List<T> list, Comparator<? super T> c) {
        ...
    }
}
```

- 지네릭 메서드를 호출할 때, 타입 변수에 타입을 대입해야 한다.

  (대부분의 경우 추정이 가능하므로 생략할 수 있음.)

```
FruitBox<Fruit> fruitBox = new FruitBox<Fruit>();
FruitBox<Apple> appleBox = new FruitBox<Apple>();
        ...
System.out.println(Juicer.<Fruit>makeJuice(fruitBox));
System.out.println(Juicer.makeJuice(appleBox)); // 대입된 타입 생략가능
```

1.7 지네릭 타입의 형변환

- 지네릭 타입과 원시 타입간의 형변환은 불가능

```
Box          box    = null;
Box<Object> objBox = null;

box    = (Box)objBox;       // OK. 지네릭 타입  → 원시 타입. 경고 발생
objBox = (Box<Object>)box; // OK. 원시 타입     → 지네릭 타입. 경고 발생
```

- 와일드 카드가 사용된 지네릭 타입으로는 형변환 가능

```
Box<? extends Object> wBox = new Box<String>();

FruitBox<? extends Fruit> box = null;
FruitBox<Apple> appleBox = (FruitBox<Apple>)box; // OK. 미확인 타입으로 형변환 경고
```

- <? extends Object>를 줄여서 <?>로 쓸 수 있다.

```
Optional<?> EMPTY = new Optional<?>();  // 에러. 미확인 타입의 객체는 생성불가
Optional<?> EMPTY = new Optional<Object>(); // OK.
Optional<?> EMPTY = new Optional<>();       // OK. 위의 문장과 동일
```

[주의] class Box<T extends Fruit>의 경우 Box<?> b = new Box<>;는 Box<?> b = new Box<Fruit>;이다.

1.8 지네릭 타입의 제거

- 컴파일러는 지네릭 타입을 제거하고, 필요한 곳에 형변환을 넣는다.

① 지네릭 타입의 경계(bound)를 제거

```
class Box<T extends Fruit> {
    void add(T t) {
        ...
    }
}
```

→

```
class Box {
    void add(Fruit t) {
        ...
    }
}
```

② 지네릭 타입 제거 후에 타입이 불일치하면, 형변환을 추가

```
T get(int i) {
return list.get(i);
}
```

→

```
Fruit get(int i) {
return (Fruit)list.get(i);
}
```

③ 와일드 카드가 포함된 경우, 적절한 타입으로 형변환 추가

```
static Juice makeJuice(FruitBox<? extends Fruit> box) {
    String tmp = "";
    for(Fruit f : box.getList()) tmp += f + " ";
    return new Juice(tmp);
}
```

```
static Juice makeJuice(FruitBox box) {
    String tmp = "";
    Iterator it = box.getList().iterator();
    while(it.hasNext()) {
        tmp += (Fruit)it.next() + " ";
    }
    return new Juice(tmp);
}
```

2. 열거형(enums)

2.1 열거형이란?

- 관련된 상수들을 같이 묶어 놓은 것. Java는 타입에 안전한 열거형을 제공

```
class Card {
    static final int CLOVER = 0;
    static final int HEART = 1;
    static final int DIAMOND = 2;
    static final int SPADE = 3;

    static final int TWO = 0;
    static final int THREE = 1;
    static final int FOUR = 2;

    final int kind;
    final int num;
}
```

```
class Card {
    enum Kind   { CLOVER, HEART, DIAMOND, SPADE }   // 열거형 Kind를 정의
    enum Value  { TWO, THREE, FOUR }                // 열거형 Value를 정의

    final Kind  kind;    // 타입이 int가 아닌 Kind임에 유의하자.
    final Value value;
}
```

2.2 열거형의 정의와 사용

- 열거형을 정의하는 방법

```
enum 열거형이름 { 상수명1, 상수명2, ... }
```

- 열거형 타입의 변수를 선언하고 사용하는 방법

```
enum Direction { EAST, SOUTH, WEST, NORTH }

class Unit {
    int x, y;    // 유닛의 위치
        Direction dir;       // 열거형을 인스턴스 변수로 선언

    void init() {
        dir = Direction.EAST;  // 유닛의 방향을 EAST로 초기화
    }
}
```

- 열거형 상수의 비교에 ==와 compareTo() 사용가능

```
if(dir==Direction.EAST) {
    x++;
} else if (dir > Direction.WEST) { // 에러. 열거형 상수에 비교연산자 사용불가
    ...
} else if (dir.compareTo(Direction.WEST)>0) { // compareTo()는 가능
    ...
}
```

2.3 모든 열거형의 조상 – java.lang.Enum

- 모든 열거형은 Enum의 자손이며, 아래의 메서드를 상속받는다.

메서드	설명
Class<E> getDeclaringClass( )	열거형의 Class객체를 반환한다.
String name( )	열거형 상수의 이름을 문자열로 반환한다.
int ordinal( )	열거형 상수가 정의된 순서를 반환한다.(0부터 시작)
T valueOf(Class<T> enumType, String name)	지정된 열거형에서 name과 일치하는 열거형 상수를 반환한다.

- 컴파일러가 자동적으로 추가해주는 values()도 있다.

```
static E values()
static E valueOf(String name)

Direction d = Direction.valueOf("WEST");
```

2.4 열거형에 멤버 추가하기

- 불연속적인 열거형 상수의 경우, 원하는 값을 괄호()안에 적는다.

```
enum Direction { EAST(1), SOUTH(5), WEST(-1), NORTH(10) }
```

- 괄호()를 사용하려면, 인스턴스 변수와 생성자를 새로 추가해 줘야 한다.

```
enum Direction {
    EAST(1), SOUTH(5), WEST(-1), NORTH(10);  // 끝에 ';'를 추가해야 한다.

    private final int value;  // 정수를 저장할 필드(인스턴스 변수)를 추가
    Direction(int value) { this.value = value; } // 생성자를 추가

    public int getValue() { return value; }
}
```

- 열거형의 생성자는 묵시적으로 private이므로, 외부에서 객체생성 불가

```
Direction d = new Direction(1); // 에러. 열거형의 생성자는 외부에서 호출불가
```

2.4 열거형의 이해

- 열거형 Direction이 아래와 같이 선언되어 있을 때,

```
enum Direction { EAST, SOUTH, WEST, NORTH }
```

- 열거형 Direction은 아래와 같은 클래스로 선언된 것과 유사하다.

```
class Direction {
    static final Direction EAST  = new Direction("EAST");
    static final Direction SOUTH = new Direction("SOUTH");
    static final Direction WEST  = new Direction("WEST");
    static final Direction NORTH = new Direction("NORTH");

    private String name;

    private Direction(String name) {
        this.name = name;
    }
}
```

3. 애너테이션(Annotation)

3.1 애너테이션이란?

- 주석처럼 프로그래밍 언어에 영향을 미치지 않으며, 유용한 정보를 제공

```
/**
 * The common interface extended by all annotation types.  Note that an
 * interface that manually extends this one does <i>not</i> define
 * an annotation type.  Also note that this interface does not itself
 * define an annotation type.
    ...
 * The {@link java.lang.reflect.AnnotatedElement} interface discusses
 * compatibility concerns when evolving an annotation type from being
 * non-repeatable to being repeatable.
 *
 * @author  Josh Bloch
 * @since   1.5
 */
public interface Annotation {
    ...
```

- 애너테이션의 사용예

```
@Test    // 이 메서드가 테스트 대상임을 테스트 프로그램에게 알린다.
public void method() {
        ...
}
```

3.2 표준 애너테이션

- Java에서 제공하는 애너테이션

애너테이션	설명
@Override	컴파일러에게 오버라이딩하는 메서드라는 것을 알린다.
@Deprecated	앞으로 사용하지 않을 것을 권장하는 대상에 붙인다.
@SuppressWarnings	컴파일러의 특정 경고메시지가 나타나지 않게 해준다.
@SafeVarargs	지네릭스 타입의 가변인자에 사용한다.(JDK1.7)
@FunctionalInterface	함수형 인터페이스라는 것을 알린다.(JDK1.8)
@Native	native메서드에서 참조되는 상수 앞에 붙인다.(JDK1.8)
@Target*	애너테이션이 적용가능한 대상을 지정하는데 사용한다.
@Documented*	애너테이션 정보가 javadoc으로 작성된 문서에 포함되게 한다.
@Inherited*	애너테이션이 자손 클래스에 상속되도록 한다.
@Retention*	애너테이션이 유지되는 범위를 지정하는데 사용한다.
@Repeatable*	애너테이션을 반복해서 적용할 수 있게 한다.(JDK1.8)

▲ 표12-2 자바에서 기본적으로 제공하는 표준 애너테이션(*가 붙은 것은 메타 애너테이션)

3.2 표준 애너테이션 - @Override

- 오버라이딩을 올바르게 했는지 컴파일러가 체크하게 한다.
- 오버라이딩할 때 메서드이름을 잘못적는 실수를 하는 경우가 많다.

```
class Parent {
    void parentMethod() { }
}

class Child extends Parent {
    void parentmethod() { } // 오버라이딩하려 했으나 실수로 이름을 잘못적음
}
```

- 오버라이딩할 때는 메서드 선언부 앞에 @Override를 붙이자.

```
class Child extends Parent {
   void parentmethod(){}
}
```

→

```
class Child extends Parent {
   @Override
   void parentmethod() {}
}
```

▼ 컴파일 결과

```
AnnotationEx1.java:6: error: method does not override or implement a method
from a supertype
       @Override
       ^
1 error
```

3.2 표준 애너테이션 - @Deprecated

- 앞으로 사용하지 않을 것을 권장하는 필드나 메서드에 붙인다.
- @Deprecated의 사용 예, Date클래스의 getDate()

```
int                 getDate()
                    Deprecated.
                    As of JDK version 1.1, replaced by
                    Calendar.get(Calendar.DAY_OF_MONTH).
```

```
@Deprecated
public int getDate() {
    return normalize().getDayOfMonth();
}
```

- @Deprecated가 붙은 대상이 사용된 코드를 컴파일하면 나타나는 메시지

```
Note: AnnotationEx2.java uses or overrides a deprecated API.
Note: Recompile with -Xlint:deprecation for details.
```

3.2 표준 애너테이션 - @FunctionalInterface

- 함수형 인터페이스에 붙이면, 컴파일러가 올바르게 작성했는지 체크

  함수형 인터페이스에는 하나의 추상메서드만 가져야 한다는 제약이 있음

```
@FunctionalInterface
public interface Runnable {
    public abstract void run(); // 추상 메서드
}
```

3.2 표준 애너테이션 - @SuppressWarnings

- 컴파일러의 경고메시지가 나타나지 않게 억제한다.
- 괄호()안에 억제하고자하는 경고의 종류를 문자열로 지정

```
@SuppressWarnings("unchecked")       // 지네릭스와 관련된 경고를 억제
ArrayList list = new ArrayList();    // 지네릭 타입을 지정하지 않았음.
list.add(obj);                       // 여기서 경고가 발생
```

- 둘 이상의 경고를 동시에 억제하려면 다음과 같이 한다.

```
@SuppressWarnings({"deprecation", "unchecked", "varargs"})
```

- '-Xlint'옵션으로 컴파일하면, 경고메시지를 확인할 수 있다.

  괄호[]안이 경고의 종류. 아래의 경우 rawtypes

```
C:\jdk1.8\work\ch12>javac -Xlint AnnotationTest.java
AnnotationTest.java:15: warning: [rawtypes] found raw type: List
    public static void sort(List list) {
                            ^
  missing type arguments for generic class List<E>
  where E is a type-variable:
    E extends Object declared in interface List
```

3.2 표준 애너테이션 - @SafeVarargs

- 가변인자의 타입이 non-reifiable인 경우 발생하는 unchecked경고를 억제
- 생성자 또는 static이나 final이 붙은 메서드에만 붙일 수 있다.
  (오버라이딩이 가능한 메서드에 사용불가)
- @SafeVarargs에 의한 경고의 억제를 위해 @SuppressWarnings를 사용

```
@SafeVarargs                        // 'unchecked'경고를 억제한다.
@SuppressWarnings("varargs") // 'varargs'경고를 억제한다.
public static <T> List<T> asList(T... a) {
    return new ArrayList<>(a);
}
```

3.3 메타 애너테이션 - @Target

- 메타 애너테이션은 '애너테이션을 위한 애너테이션'
- 애너테이션을 정의할 때, 적용대상이나 유지기간의 지정에 사용
- @Target은 애너테이션을 적용할 수 있는 대상의 지정에 사용

```
@Target({TYPE, FIELD, METHOD, PARAMETER,CONSTRUCTOR, LOCAL_VARIABLE})
@Retention(RetentionPolicy.SOURCE)
public @interface SuppressWarnings {
    String[] value();
}
```

대상 타입	의미
ANNOTATION_TYPE	애너테이션
CONSTRUCTOR	생성자
FIELD	필드(멤버변수, enum상수)
LOCAL_VARIABLE	지역변수
METHOD	메서드
PACKAGE	패키지
PARAMETER	매개변수
TYPE	타입(클래스, 인터페이스, enum)
TYPE_PARAMETER	타입 매개변수(JDK1.8)
TYPE_USE	타입이 사용되는 모든 곳(JDK1.8)

3.3 메타 애너테이션 - @Retention

- 애너테이션이 유지(retention)되는 기간을 지정하는데 사용

유지 정책	의미
SOURCE	소스 파일에만 존재. 클래스파일에는 존재하지 않음.
CLASS	클래스 파일에 존재. 실행시에 사용불가. 기본값
RUNTIME	클래스 파일에 존재. 실행시에 사용가능.

▲ 표12-4 애너테이션 유지정책(retention policy)의 종류

- 컴파일러에 의해 사용되는 애너테이션의 유지 정책은 SOURCE이다.

```
@Target(ElementType.METHOD)
@Retention(RetentionPolicy.SOURCE)
public @interface Override {}
```

- 실행시에 사용 가능한 애너테이션의 정책은 RUNTIME이다.

```
@Documented
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.TYPE)
public @interface FunctionalInterface {}
```

3.3 메타 애너테이션 - @Documented, @Inherited

- javadoc으로 작성한 문서에 포함시키려면 @Documented를 붙인다.

```
@Documented
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.TYPE)
public @interface FunctionalInterface {}
```

- 애너테이션을 자손 클래스에 상속하고자 할 때, @Inherited를 붙인다.

```
@Inherited  // @SupperAnno가 자손까지 영향 미치게
@interface SupperAnno {}

@SuperAnno
class Parent {}

class Child extends Parent {}  // Child에 애너테이션이 붙은 것으로 인식
```

3.3 메타 애너테이션 - @Repeatable

- 반복해서 붙일 수 있는 애너테이션을 정의할 때 사용

```
@Repeatable(ToDos.class) // ToDo애너테이션을 여러 번 반복해서 쓸 수 있게 한다.
@interface ToDo {
    String value();
}
```

- @Repeatable이 붙은 애너테이션은 반복해서 붙일 수 있다.

```
@ToDo("delete test codes.")
@ToDo("override inherited methods")
class MyClass {
    ...
}
```

3.3 메타 애너테이션 - @Native

- native메서드에 의해 참조되는 상수에 붙이는 애너테이션

```
@Native public static final long MIN_VALUE = 0x8000000000000000L;
```

- native메서드에 JVM이 설치된 OS의 메서드이다.

```
public class Object {
    private static native void registerNatives();  // 네이티브 메서드

    static {
        registerNatives(); // 네이티브 메서드를 호출
    }

    protected native Object clone() throws CloneNotSupportedException;
    public final native Class<?> getClass();
    public final native void notify();
    public final native void notifyAll();
    public final native void wait(long timeout) throws InterruptedException;
    public native int hashCode();
        ...
}
```

3.4 애너테이션 타입 정의하기

- 애너테이션을 직접 만들어 쓸 수 있다.

```
@interface 애너테이션이름 {
    타입 요소이름();   // 애너테이션의 요소를 선언한다.
    ...
}
```

- 애너테이션의 메서드는 추상메서드이며, 애너테이션을 적용할 때 모두 지정해야한다.(순서 상관없음)

```
@interface TestInfo {
    int       count();
    String    testedBy();
    String[]  testTools();
    TestType  testType(); // enum TestType { FIRST, FINAL }
    DateTime  testDate(); // 자신이 아닌 다른 애너테이션(@DateTime)을 포함할 수 있다.
}

@interface DateTime {
    String yymmdd();
    String hhmmss();
}
```

3.5 애너테이션 요소의 기본값

- 적용시 값을 지정하지 않으면, 사용될 수 있는 기본값 지정 가능(null제외)

```
@interface TestInfo {
    int count() default 1;      // 기본값을 1로 지정
}

@TestInfo   // @TestInfo(count=1)과 동일
public class NewClass { ... }
```

- 요소가 하나일 때는 요소의 이름 생략가능

```
@TestInfo(5)   // @TestInfo(count=5)와 동일
public class NewClass { ... }
```

- 요소의 타입이 배열인 경우, 괄호{}를 사용해야 한다.

```
@interface TestInfo {
    String[] info()  default {"aaa","bbb"}; // 기본값이 여러 개인 경우. 괄호{}사용
    String[] info2() default "ccc"; // 기본값이 하나인 경우. 괄호 생략가능
}

@TestInfo                 // @TestInfo(info={"aaa","bbb"}, info2="ccc")와 동일
@TestInfo(info2={})  // @TestInfo(info={"aaa","bbb"}, info2={})와 동일
class NewClass { ... }
```

3.6 모든 애너테이션의 조상 – java.lang.annotation.Annotation

- Annotation은 모든 애너테이션의 조상이지만 상속은 불가

```
@interface TestInfo extends Annotation { // 에러. 허용되지 않는 표현
    int         count();
    String      testedBy();
        ...
}
```

- 사실 Annotaion은 인터페이스로 정의되어 있다.

```
package java.lang.annotation;

public interface Annotation {  // Annotation자신은 인터페이스이다.
    boolean equals(Object obj);
    int hashCode();
    String toString();

    Class<? extends Annotation> annotationType(); // 애너테이션의 타입을 반환
}
```

3.7 마커 애너테이션 - Marker Annotation

- 요소가 하나도 정의되지 않은 애너테이션

```
@Target(ElementType.METHOD)
@Retention(RetentionPolicy.SOURCE)
public @interface Override {}   // 마커 애너테이션. 정의된 요소가 하나도 없다.

@Target(ElementType.METHOD)
@Retention(RetentionPolicy.SOURCE)
public @interface Test {}       // 마커 애너테이션. 정의된 요소가 하나도 없다.
```

3.8 애너테이션 요소의 규칙

- 애너테이션의 요소를 선언할 때 아래의 규칙을 반드시 지켜야 한다.

  - 요소의 타입은 기본형, String, enum, 애너테이션, Class만 허용됨
  - 괄호()안에 매개변수를 선언할 수 없다.
  - 예외를 선언할 수 없다.
  - 요소를 타입 매개변수로 정의할 수 없다.

- 아래의 코드에서 잘못된 부분은 무엇인지 생각해보자.

```
@interface AnnoTest {
    int id = 100;                       // OK. 상수 선언. static final int id = 100;
    String major(int i, int j);         // 에러. 매개변수를 선언할 수 없음
    String minor() throws Exception;    // 에러. 예외를 선언할 수 없음
    ArrayList<T> list();                // 에러. 요소의 타입에 타입 매개변수 사용불가
}
```