

📁

🔗 main ▾

curriculum300H / 1.JAVA(84시간)  
/ 16일차(3h) - 열거형, 애너테이션 /

📄

🔍


Go to file

t

+

Add file ▾

⋮

 yonggyo1125 동영상 강의 URL 업데이트

7005896 · 2 years ago

🕒 History

⋮

Name		Name	Last commit date
📁 ..			
📁 day16	용어 변경		2 years ago

📁

🔗 main ▾

curriculum300H / 1.JAVA(84시간) / 16일차(3h) - 열거형, 애너테이션 /

↑ Top

README.md

✎⋮

# 강의 동영상 링크

## [동영상 링크](#)

# 열거형(enums)

## 열거형이란?

- 자바는 C언어와 달리 열거형라는 것이 존재하지 않았으나 JDK1.5부터 추가되었다.
- 자바의 열거형은 C언어의 열거형보다 더 향상된 특징을 가진다.
- 값뿐만 아니라 **타입까지 체크하여 타입에 안전하다**
- 상수의 값이 바뀌면 해당 상수를 참조하는 모든 소스를 다시 컴파일 해야 한다. 그러나 열거형 상수를 사용하면, 기존의 소스를 다시 컴파일하지 않아도 된다.**

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

📄

```
final Value value;
```

```
}
```

## 열거형 정의와 사용

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

- 사용하는 방법은 **열거형 이름.상수명**이다
- 클래스의 **static** 변수를 참조하는 것과 동일

```
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 상수간의 비교에는 **'=='**를 사용할 수 있다.(상수의 **주소를 비교**하므로)
- '<', '>'와 같은 비교연산자는 사용할 수 없고 **compareTo()**는 사용가능하다.

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

- switch문 조건식에도 열거형을 사용할 수 있다.
- 주의할 점은 case문에 열거형의 이름은 적지 않고 상수의 이름만 적어야 한다.

```
void move() {  
    switch(dir) {  
        case EAST: x++; // Direction.EAST라고 쓰면 안된다.  
            break;  
        case WEST: x--;  
            break;  
        case SOUTH: y++;  
            break;  
        case NORTH: y--;  
            break;  
    }
```

```
}  
}
```

## 모든 열거형의 상위 클래스 - java.lang.Enum

```
Direction[] dArr = Direction.values();  
  
for(Direction d : dArr) {  
    System.out.printf("%s = %d\n", d.name(), d.ordinal());  
}
```



- values()는 열거형의 모든 상수를 배열에 담아서 반환한다.
- 이 메서드는 모든 열거형이 가지고 있는 것으로 컴파일러가 자동으로 추가해 준다.
- ordinal()은 모든 열거형의 상위 클래스인 java.lang.Enum 클래스에 정의된 것으로 열거형 상수가 정의된 순서(0부터 시작)를 정수로 반환한다

### Enum 클래스에 정의된 메서드

메서드	설명
Class<E> getDeclaringClass()	열거형의 Class 객체를 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 name값과 일치하는 열거형 상수를 반환한다.

### 컴파일러가 자동으로 추가해주는 메서드

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

- 열거형 상수의 이름으로 문자열 상수에 대한 참조를 얻을 수 있게 해준다.

```
Direction d = Direction.valueOf("WEST");  
  
System.out.println(d); // WEST  
System.out.println(Direction.WEST == Direction.valueOf("WEST")); // true
```



### day16/EnumEx1.java

```
package day16;  
  
enum Direction { EAST, SOUTH, WEST, NORTH };  
  
public class EnumEx1 {  
    public static void main(String[] args) {  
        Direction d1 = Direction.EAST;
```



```

Direction d2 = Direction.valueOf("WEST");
Direction d3 = Enum.valueOf(Direction.class, "EAST");

System.out.println("d1=" + d1);
System.out.println("d2=" + d2);
System.out.println("d3=" + d3);

System.out.println("d1==d2 ? " + (d1 == d2)); // false
System.out.println("d1==d3 ? " + (d1 == d3)); // true
System.out.println("d1.equals(d3) ? " + d1.equals(d3)); // true
// System.out.println("d2 > d3 ? " + (d1 > d3)); // 에러
System.out.println("d1.compareTo(d3) ? " + (d1.compareTo(d3))); //

0

System.out.println("d1.compareTo(d2) ? " + (d1.compareTo(d2)));

switch(d1) {
    case EAST: // Direction.EAST라고 쓸 수 없다.
        System.out.println("The direction is EAST.");
break;

    case SOUTH:
        System.out.println("The direction is SOUTH.");
break;

    case WEST:
        System.out.println("The direction is WEST.");
break;

    case NORTH:
        System.out.println("The direction is NORTH");
break;

    default:
        System.out.println("Invalid direction.");
}

Direction[] dArr = Direction.values();

for(Direction d : dArr) {
    System.out.printf("%s=%d\n", d.name(), d.ordinal());
}
}

```

실행 결과

```

d1=EAST
d2=WEST
d3=EAST
d1==d2 ? false
d1==d3 ? true
d1.equals(d3) ? true
d1.compareTo(d3) ? 0
d1.compareTo(d2) ? -2
The direction is EAST.
EAST=0
SOUTH=1
WEST=2
NORTH=3

```

# 열거형에 멤버 추가하기

- Enum 클래스에 정의된 `ordinal()`이 열거형 상수가 정의된 순서를 반환하지만, 이 값을 열거형 상수의 값으로 사용하지 않는 것이 좋다. 이 값은 내부적인 용도로만 사용되기 위한 것이다.
- 열거형 상수의 값이 불연속적인 경우에는 다음과 같이 열거형 상수의 이름 옆에 원하는 값을 괄호()와 함께 적어주면 된다.

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



- 지정된 값을 값을 저장할 수 있는 인스턴스 변수와 생성자를 새로 추가해 주어야 한다.
- 주의할 점
  - 열거형 상수를 모두 정의한 다음에 다른 멤버들을 추가해야 한다.
  - 열거형 상수의 마지막에 ';'를 추가해야 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; } // 외부에서 이 값을 얻을 수 있도록  
    추가  
}
```



- 열거형 생성자는 외부에서 호출이 불가하다. 열거형의 생성자는 제어자가 묵시적으로 `private` 이다

```
Direction d = new Direction(1); // 에러, 열거형의 생성자는 외부에서 호출 불가
```



```
enum Direction {  
    ...  
    Direction(int value) { // private Direction(int value)와 동일  
        this.value = value;  
    }  
    ...  
}
```



- 열거형 상수에 여러값을 지정할 수 있다. 다만 그에 맞게 인스턴스 변수와 생성자 등을 추가해야 한다.

```
enum Direction {  
    EAST(1, ">"), SOUTH(2, "V"), WEST(3, "<"), NORTH(4, "^");  
  
    private final int value;  
    private final String symbol;  
  
    Direction(int value, String symbol) { // 접근 제어자 private이 생략됨
```



```

        this.value = value;
        this.symbol = symbol;
    }

    public int getValue() { return value; }
    public String getSymbol() { return symbol; }
}

```

## day16/EnumEx2.java

```

package day16;

enum Direction2 {
    EAST(1, ">"), SOUTH(2, "V"), WEST(3, "<"), NORTH(3, "^");

    private static final Direction2[] DIR_ARR = Direction2.values();
    private final int value;
    private final String symbol;

    Direction2(int value, String symbol) { // 접근 제어자 private이 생략됨
        this.value = value;
        this.symbol = symbol;
    }

    public int getValue() { return value; }
    public String getSymbol() { return symbol; }

    public static Direction2 of(int dir) {
        if (dir < 1 || dir > 4) {
            throw new IllegalArgumentException("Invalid value : " +
dir);
        }

        return DIR_ARR[dir - 1];
    }

    // 방향을 회전시키는 메서드. num의 값만큼 90도씩 시계방향으로 회전한다.
    public Direction2 rotate(int num) {
        num = num % 4;

        if (num < 0) num += 4; // num이 음수일 때는 시계반대 방향으로 회전

        return DIR_ARR[(value - 1 + num) % 4];
    }

    public String toString() {
        return name() + getSymbol();
    }
}

public class EnumEx2 {
    public static void main(String[] args) {
        for(Direction2 d : Direction2.values()) {
            System.out.printf("%s=%d\n", d.name(), d.getValue());
        }
    }
}

```



```

        Direction2 d1 = Direction2.EAST;
        Direction2 d2 = Direction2.of(1);

        System.out.printf("d1=%s, %d%n", d1.name(), d1.getValue());
        System.out.printf("d2=%s, %d%n", d2.name(), d2.getValue());

        System.out.println(Direction2.EAST.rotate(1));
        System.out.println(Direction2.EAST.rotate(2));
        System.out.println(Direction2.EAST.rotate(-1));
        System.out.println(Direction2.EAST.rotate(-2));
    }
}

```

실행결과

```

EAST=1
SOUTH=2
WEST=3
NORTH=3
d1=EAST, 1
d2=EAST, 1
SOUTHV
WEST<
NORTH^
WEST<

```

## 열거형에 추상메서드 추가하기

- 열거형에 추상 메서드를 선언하면 각 열거형 상수가 이 추상 메서드를 반드시 구현해야 한다.
- 추상클래스나 인터페이스를 가지고 추상 메서드를 구현함으로써 익명 클래스를 작성하는 것과 유사하다

```

enum Transportation {
    BUS(100) { int fare(int distance) { return distance * BASIC_FARE; }},
    TRAIN(150) { int fare(int distance) { return distance * BASIC_FARE; }},
    SHIP(100) { int fare(int distance) { return distance * BASIC_FARE; }},
    AIRPLANE(300) { int fare(int distance) { return distance * BASIC_FARE; }},
};

abstract int fare(int distance); // 거리에 따른 요금을 계산하는 추상 메서드

protected final int BASIC_FARE; // protected로 해야 각 상수에서 접근가능

Transportation(int basicFare) {
    BASIC_FARE = basicFare;
}

public int getBasicFare() { return BASIC_FARE; }
}

```



```
enum Transportation {
    BUS(100) { int fare(int distance) { return distance * BASIC_FARE; }},
    TRAIN(150) { int fare(int distance) { return distance * BASIC_FARE; }},
    SHIP(100) { int fare(int distance) { return distance * BASIC_FARE; }},
    AIRPLANE(300) { int fare(int distance) { return distance * BASIC_FARE;
}};

    abstract int fare(int distance); // 거리에 따른 요금을 계산하는 추상 메서드

    protected final int BASIC_FARE; // protected로 해야 각 상수에서 접근가능

    Transportation(int basicFare) {
        BASIC_FARE = basicFare;
    }

    public int getBasicFare() { return BASIC_FARE; }
}

public class EnumEx3 {
    public static void main(String[] args) {
        System.out.println("bus fare=" + Transportation.BUS.fare(100));
        System.out.println("train fare=" +
Transportation.TRAIN.fare(100));
        System.out.println("ship fare=" + Transportation.SHIP.fare(100));
        System.out.println("airplane fare=" +
Transportation.AIRPLANE.fare(100));
    }
}
```

실행결과

```
bus fare=10000
train fare=15000
ship fare=10000
airplane fare=30000
```

## 애너테이션(annotation)

- 프로그램의 소스코드 안에 다른 프로그램을 위한 정보를 미리 약속된 형식으로 포함시킨 것이 애너테이션이다.
- 애너테이션은 주석(comment)처럼 프로그래밍 언어에 영향을 미치지 않으면서도 다른 프로그램에게 유용한 정보를 제공할 수 있다는 장점이 있다.
- 애너테이션(annotation)의 뜻은 주석, 주해, 메모이다.

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



'@Test'는 이 메서드를 테스트해야 한다는 것을 테스트 프로그램에게 알리는 역할을 하며, 메서드가 포함된 프로그램 자체에는 아무런 영향을 미치지 않는다. 주석처럼 존재하지 않는 것이나 다름없다.

- 표준 에너지테이션 : JDK에서 제공하며 주로 컴파일러를 위한 것으로 컴파일러에게 유용한 정보를 제공한다.
- 메타 에너지테이션 : 새로운 에너지테이션을 정의할 때 사용한다.

## 에너지테이션이란?

자바에서 기본적으로 제공하는 표준 에너지테이션(\*가 붙은 것은 메타 에너지테이션)

에너지테이션	설명
@Override	컴파일러에게 재정의 하는 메서드라는 것을 알린다.
@Deprecated	앞으로 사용되지 않을 것을 권장하는 대상에게 붙인다.
@SuppressWarnings	컴파일러의 특정 경고메세지가 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*	에너지테이션이 적용 가능한 대상을 지정하는데 사용한다.
@Documented*	에너지테이션 정보가 @javadoc으로 작성된 문서에 포함되게 한다.
@Inherited*	에너지테이션이 하위 클래스에 상속되도록 한다.
@Retention*	에너지테이션이 유지되는 범위를 지정하는데 사용한다.
@Repeatable*	에너지테이션을 반복해서 적용할 수 있게 한다.(JDK1.8)

## 표준 에너지테이션

- 메서드 앞에만 붙일 수 있는 에너지테이션
- 상위클래스의 메서드를 재정의 하는 것이라는 걸 컴파일러에게 알려주는 역할을 한다.
- 재정의할 때 메서드 앞에 '@Override'라고 에너지테이션을 붙이면, 컴파일러가 같은 이름의 메서드가 상위 클래스에 있는지 확인하고 없으면 에러메세지를 출력한다.
- 재정의할 때 메서드 앞에 '@Override'를 붙이는 것이 필수는 아니지만 알아내기 어려운 실수를 미연에 방지해 주므로 붙이는 것이 좋다.

### @Override

day16/AnnotationEx1.java

```
package day16;
```

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

```
public class AnnotationEx1 extends Parent{  
    @Override  
    void parentmethod() {} // 상위클래스 메서드의 이름을 잘못 적음  
}
```

컴파일 결과

AnnotationEx1.java:8: error: method does not override or implement a method from a supertype

```
    @Override  
    ^
```

1 error

## @Deprecated

- '@Deprecated' 애너테이션이 붙은 대상은 다른 것으로 대체되었으니 더 이상 사용하지 않을 것을 권한다는 의미
- 예) java.util.Date 클래스의 대부분의 메서드에서는 '@Deprecated'가 붙어 있다.

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

이 메서드 대신에 JDK1.1 부터 추가된 Calendar 클래스의 get()을 사용하라는 것

## day16/AnnotationEx2.java

```
package day16;
```

```
class NewClass {  
    int newField;  
  
    int getNewField() { return newField; }  
  
    @Deprecated  
    int oldField;  
  
    @Deprecated  
    int getOldField() { return oldField; }  
}
```

```
public class AnnotationEx2 {  
    public static void main(String[] args) {  
        NewClass nc = new NewClass();  
    }  
}
```

```
nc.oldField = 10; // @Deprecated가 붙은 대상을 사용
System.out.println(nc.getOldField()); // @Deprecated가 붙은 대상을
```

사용

```
}
}
```

컴파일 결과

```
D:\javaEx\lecture\src\day16>javac AnnotationEx2.java
```

```
Note: AnnotationEx2.java uses or overrides a deprecated API.
```

```
Note: Recompile with -Xlint:deprecation for details.
```

## @FunctionalInterface

- '함수형 인터페이스(functional interface)'를 선언할 때, 이 애너테이션을 붙이면 컴파일러가 '함수형 인터페이스'를 올바르게 선언했는지 확인하고, 잘못된 경우 에러를 발생시킨다.
- 필수는 아니지만, 붙이면 실수를 방지할 수 있으므로 '함수형 인터페이스'를 선언할 때는 이 애너테이션을 붙이는 것이 좋다.
- 함수형 인터페이스는 추상 메서드가 하나뿐이어야 한다는 제약이 있다(18일차 - 랴다식 참조)

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



## @SuppressWarnings

- 컴파일러가 보여주는 경고메시지가 나타나지 않게 억제해 준다.
- 주로 사용되는 것은 "deprecation", "unchecked", "rawtypes", "varargs" 정도 이다.
  - deprecation : "@Deprecated"가 붙은 대상을 사용해서 발생하는 경고를 억제
  - unchecked : 지네릭스로 타입을 지정하지 않았을 때 발생하는 경고를 억제
  - rawtypes : 지네릭스를 사용하지 않아서 발생하는 경고를 억제
  - varargs : 가변인자의 타입이 제네릭 타입일 때 발생하는 경고를 억제

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



- 둘 이상의 경고를 동시에 억제하려면 배열에서처럼 중괄호{}를 추가로 사용해야 한다.

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



## day16/AnnotationEx3.java

```
package day16;

import java.util.ArrayList;
```



```

class NewClass2 {
    int newField;

    int getNewField() {
        return newField;
    }

    @Deprecated
    int oldField;

    @Deprecated
    int getOldField() {
        return oldField;
    }
}

public class AnnotationEx3 {
    @SuppressWarnings("deprecation") // deprecation 관련 경고를 억제
    public static void main(String[] args) {
        NewClass2 nc = new NewClass2();

        nc.oldField = 10;
        System.out.println(nc.getOldField());

        @SuppressWarnings("unchecked") // 지네릭스 관련 경고를 억제
        ArrayList<NewClass2> list = new ArrayList(); // 타입을 지정하지 않음
        list.add(nc);
    }
}

```

## @SafeVarargs

메서드에 선언된 가변인자의 타입이 non-reifiable 타입일 경우, 해당 메서드를 선언하는 부분과 호출하는 부분에서 "unchecked" 경고가 발생한다. 해당 코드에 문제가 없다면 이 경고를 억제하기 위해 "@SafeVarargs"를 사용해야 한다.

- 이 애너테이션은 static이나 final이 붙은 메서드에만 붙일 수 있다. 즉 오버라이드 될 수 있는 메서드에는 사용할 수 없다.
- 지네릭스에서 살펴본 것과 같이 어떤 타입들은 컴파일 이후에 제거된다. 컴파일 후에도 제거되지 않는 타입을 reifiable 타입이라 하고, 제거되는 타입을 non-reifiable 타입이라고 한다.
- 지네릭 타입들은 대부분 컴파일 시에 제거되므로 non-reifiable 타입이다.

## day16/AnnotationEx4.java

```

package day16;

import java.util.Arrays;

class MyArrayList<T> {
    T[] arr;
}

```



```

@SafeVarargs
MyArrayList(T... arr) {
    this.arr = arr;
}

@SafeVarargs
public static <T> MyArrayList<T> asList(T... a) {
    return new MyArrayList<>(a);
}

public String toString() {
    return Arrays.toString(arr);
}
}

public class AnnotationEx4 {
    public static void main(String[] args) {
        MyArrayList<String> list = MyArrayList.asList("1", "2", "3");

        System.out.println(list);
    }
}

```

## 메타 애너테이션

- 애너테이션을 위한 애너테이션
- 애너테이션을 붙이는 애너테이션으로 애너테이션을 정의할 때 애너테이션의 적용대상(target)이나 유지기간(retention)등을 지정하는데 사용된다.
- 메타 애너테이션은 "java.lang.annotation" 패키지에 포함되어 있다.

### @Target

애너테이션이 적용 가능한 대상을 지정하는데 사용된다.

"@target"으로 지정할 수 있는 애너테이션 적용대상의 종류

대상 타입	의미
ANNOTATION_TYPE	애너테이션
CONSTRUCTOR	생성자
FIELD	필드(멤버변수, enum상수) - 기본자료형에 사용
LOCAL_VARIABLE	지역변수
METHOD	메서드
PACKAGE	패키지
PARAMETER	매개 변수
TYPE	타입(클래스, 인터페이스, enum)

대상 타입	의미
TYPE_PARAMETER	타입 매개변수
TYPE_USE	타입이 사용되는 모든 곳 - 참조자료형에 사용

```
import static java.lang.annotation.ElementType.*;

@Target({FIELD, TYPE, TYPE_USE}) // 적용대상이 FIELD, TYPE, TYPE_USE
public @interface MyAnnotation { // MyAnnotation을 정의

}

@MyAnnotation // 적용대상이 TYPE인 경우
class MyClass {
    @MyAnnotation // 적용대상이 FIELD인 경우
    int i;

    @MyAnnotation // 적용대상이 TYPE_USE인 경우
    MyClass mc;
}
```



## @Retention

애너테이션이 유지되는 기간을 지정하는데 사용된다.

### 애너테이션 유지정책(retention policy)의 종류

유지 정책	의미
SOURCE	소스 파일에만 존재, 클래스파일에는 존재하지 않음
CLASS	클래스 파일에 존재, 실행시에 사용불가, 기본값
<b>RUNTIME</b>	<b>클래스 파일에 존재, 실행시에 사용가능</b>

- SOURCE
  - "@Override"나 "@SuppressWarnings"처럼 컴파일러가 사용하는 애너테이션은 유지 정책이 SOURCE이다. 컴파일러를 직접 작성할 것이 아니면 이 유지정책은 필요 없다.

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



- **RUNTIME**
  - 실행 시에 '리플렉션(reflection)'을 통해 클래스 파일에 저장된 애너테이션의 정보를 읽어 서 처리할 수 있다.
  - "@FunctionalInterface"는 "@Override"처럼 컴파일러가 체크해주는 애너테이션이지만, **실행 시에도 사용되므로 유지 정책이 "RUNTIME"으로 되어 있다.**

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



## • CLASS

- 컴파일러가 애너테이션의 정보를 클래스 파일에 저장할 수 있기는 하지만, 클래스 파일이 JVM에 로딩될 때는 애너테이션의 정보가 무시되어 실행 시에 애너테이션의 정보를 얻을 수 없다.
- CLASS가 기본값이지만 상기 사유로 잘 사용되지 않는다.
- 지역변수에 붙은 애너테이션은 컴파일러만 인식할 수 있으므로, 유지정책이 RUNTIME인 애너테이션을 지역변수에 붙여도 실행 시에는 인식되지 않는다. (지역변수는 함수가 호출될때 스택에 올라갈때 인식이 가능하므로)

## @Documented

애너테이션에 대한 정보가 javadoc으로 작성한 문서에 포함되도록 한다.

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



## @Inherited

- 애너테이션을 하위 클래스에 상속되도록 한다,
- "@Inherited"가 붙은 애너테이션을 상위 클래스에 붙이면, 하위 클래스도 이 애너테이션이 붙은 것과 같이 인식된다.

```
@Inherited // @SuperAnno가 하위클래스까지 영향 미치게
@interface SuperAnno {}
```

```
@SuperAnno
class Parent {}
```

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



## @Repeatable

"@Repeatable"이 붙은 애너테이션은 여러 번 붙일 수 있다.

### day16/AnnotationEx5.java

```
package day16;

import java.lang.annotation.*;
```



```

@interface Todos {
    Todo[] value();
}

@Repeatable(Todos.class)
@interface Todo {
    String value();
}

@ToDo("자바 공부")
@ToDo("자바스크립트 공부")
@ToDo("데이터베이스 공부")
public class AnnotationEx5 {
}

```

## @Native

- 네이티브 메서드(native method)에 의해 참조되는 '상수 필드(constant field)'
- 네이티브 메서드는 JVM이 설치된 OS의 메서드를 말한다.
- 네이티브 메서드는 보통 C언어로 작성되어 있는데, 자바에서는 메서드의 선언부만 정의하고 구현은 하지 않는다.
- 그래서 추상 메서드 처럼 선언부만 있고 몸통이 없다.
- 네이티브 메서드는 자바로 정의되어 있기 때문에 호출하는 방법은 자바의 일반 메서드와 다르지 않지만 실제로 호출하는 것은 OS의 메서드이다.

java.lang.Long 클래스에 정의된 상수 예



```
@Native public static final Long MIN_VALUE = .....
```

```

public class Object {
    private static native void registerNatives();

    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();
    ...
}

```





# 애너테이션 타입 정의하기

- "@기호를 붙이는 것을 제외하면 인터페이스를 정의하는 것과 동일하다.
- 엄밀히 말하면 "@Override"는 애너테이션이고 'Override'는 '애너테이션'의 타입이다.

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



## 애너테이션의 요소

- 애너테이션 내에 선언된 메서드를 '애너테이션의 요소(element)'라고 한다.
- 애너테이션에도 인터페이스처럼 상수를 정의할 수 있지만 디폴트 메서드는 정의할 수 없다.

```
@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 yymdd();  
    String hhmmss();  
}
```

- 애너테이션의 요소는 반환값이 있고 매개변수는 없는 추상 메서드의 형태를 가지며, 상속을 통해 구현하지 않아도 된다.
- 애너테이션을 적용할 때 이 요소들의 값을 빠짐없이 지정해주어야 한다.
- 요소이름을 같이 적어서 적용하므로 순서는 상관 없다.

```
@TestInfo(  
    count = 3, testedBy="Kim",  
    testTools={"JUnit", "AutoTester"},  
    testType=TestType.FIRST,  
    testDate=@DateTime(yymdd="160101", hhmmss="235959")  
)  
public class NewClass {  
    ..  
}
```



- 애너테이션의 각 요소는 기본값을 가질 수 있으며, 기본값이 있는 요소는 애너테이션을 적용할 때 값을 지정하지 않으면 기본값이 사용된다.

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

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

- 애너테이션 요소가 오직 하나뿐이고 이름이 value인 경우, 애너테이션을 적용할 때 요소의 이름을 생략하고 값만 적어도 된다.

```
@interface TestInfo {
    String value();
}
```

```
@TestInfo("passed") // @TestInfo(value = "passed")와 동일
class NewClass { ... }
```

- 요소의 타입이 배열인 경우, 괄호{}를 사용해서 여러 개의 값을 지정할 수 있다.

```
@interface TestInfo {
    String[] testTools();
}
```

```
@Test(testTools={"JUnit", "AutoTester"}) // 값이 여러 개인 경우
@Test(testTools="JUnit") // 값이 하나일 때는 괄호{} 생략가능
@Test(testTools={}) // 값이 없을 때는 괄호{}가 반드시 필요
```

- 기본값을 지정할 때도 마찬가지로 괄호{}를 사용할 수 있다.

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

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

- 요소의 타입이 배열일 때에도 요소의 이름이 value이면, 요소의 이름을 생략할 수 있다.

```
@interface SuppressWarnings {
    String[] value();
}
```

```
@SuppressWarnings(value={"deprecation", "unchecked"})
@SuppressWarnings({"deprecation", "unchecked"})
class NewClass { ... }
```

# java.lang.annotation.Annotation

- 모든 애너테이션의 상위 클래스는 Annotation이다. 그러나 애너테이션은 상속이 허용되지 않으므로 명시적으로 Annotation을 상위 클래스로 지정할 수 없다.

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



- Annotation은 애너테이션이 아니라 일반적인 인터페이스로 정의되어 있다.
- 모든 애너테이션의 상위 클래스인 Annotation 인터페이스가 하기와 같이 정의되어 있으므로, 모든 애너테이션 객체에 대해 equals(), hashCode(), toString()과 같은 메서드를 호출할 수 있다.

```
package java.lang.annotation;

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

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



## 마커 애너테이션(Marker Annotation)

- 값을 지정할 필요가 없는 경우, 애너테이션의 요소를 하나도 정의하지 않을 수 있다. Serializable이나 Cloneable 인터페이스처럼, 요소가 하나도 정의되지 않은 애너테이션을 마커 애너테이션이라고 한다.

```
@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.
```



## 애너테이션 요소의 규칙

애너테이션 요소를 선언할때 반드시 지켜야 하는 규칙

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

```
@interface AnnoTest {
    int id = 100; // OK, 상수 선언, public 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
}
```

## day16/AnnotationEx6.java

```
package day16;

import java.lang.annotation.*;

@Retention(RetentionPolicy.RUNTIME) // 실행시에사용가능하도록 지정
@interface TestInfo {
    int count() default 1;
    String testedBy();
    String[] testTools() default "JUnit";
    TestType testType() default TestType.FIRST;
    DateTime testDate();
}

@Retention(RetentionPolicy.RUNTIME) // 실행 시에 사용가능하도록 지정
@interface DateTime {
    String yymmdd();
    String hhmmss();
}

enum TestType { FIRST, FINAL }

@Deprecated
@TestInfo(testedBy="aaa", testDate=@DateTime(yymmdd="160101", hhmmss="235959"))
public class AnnotationEx6 {
    public static void main(String[] args) {
        // AnnotationEx6의 Class 객체를 얻는다.
        Class<AnnotationEx6> cls = AnnotationEx6.class;

        TestInfo anno = (TestInfo)cls.getAnnotation(TestInfo.class);
        System.out.println("anno.testedBy()" + anno.testedBy());
        System.out.println("anno.testDate().yymmdd()" +
anno.testDate().yymmdd());
        System.out.println("anno.testDate().hhmmss()" +
anno.testDate().hhmmss());

        for(String str : anno.testTools()) {
            System.out.println("testTools=" + str);
        }

        System.out.println();

        // AnnotationEx6에 적용된 모든 애너테이션을 가져온다.
        Annotation[] annoArr = cls.getAnnotations();
    }
}
```

```
        for (Annotation a : annoArr) {  
            System.out.println(a);  
        }  
    }  
}
```

실행결과

```
anno.testedBy()=aaa  
anno.testDate().yymmdd()160101  
anno.testDate().hhmmss()=235959  
testTools=JUnit
```

```
@java.lang.Deprecated(forRemoval=false, since="")  
@day16.TestInfo(count=1, testType=FIRST, testTools={"JUnit"}, testedBy="aaa",  
testDate=@day16.DateTime(yymmdd="160101", hhmmss="235959"))
```