

1. 지네릭스(Generics)

JDK1.5에서 처음 도입된 지네릭스는 JDK1.8부터 도입된 람다식만큼 큰 변화였다. 그 당시만 해도 지네릭스는 선택적으로 사용하는 경우가 많았지만 이제는 지네릭스를 모르고는 Java API 문서조차 제대로 보기 어려울 만큼 중요한 위치를 차지하고 있다.

이번 장을 다 이해했다고 해서 지네릭스를 완전히 이해할 수는 없을 것이다. 이 장에서는 기본적인 개념 정도만 익히고, 다른 장을 통해서 지네릭스가 실제로 어떻게 활용되는지 배움으로써 보다 깊게 이해하게 될 것이다. 지네릭스가 잘 이해되지 않는다고 해서 이 장에만 너무 머물러 있지 않기를 당부하는 바이다.

1.1 지네릭스란?

지네릭스는 다양한 타입의 객체들을 다루는 메서드나 컬렉션 클래스에 컴파일 시의 타입 체크(compile-time type check)를 해주는 기능이다. 객체의 타입을 컴파일 시에 체크하기 때문에 객체의 타입 안정성을 높이고 형변환의 번거로움이 줄어든다.

타입 안정성을 높인다는 것은 의도하지 않은 타입의 객체가 저장되는 것을 막고, 저장된 객체를 꺼내올 때 원래의 타입과 다른 타입으로 잘못 형변환되어 발생할 수 있는 오류를 줄여준다는 뜻이다.

예를 들어, `ArrayList`와 같은 컬렉션 클래스는 다양한 종류의 객체를 담을 수 있긴 하지만 보통 한 종류의 객체를 담는 경우가 더 많다. 그런데도 꺼낼 때마다 타입체크를 하고 형변환을 하는 것은 아무래도 불편할 수밖에 없다. 게다가 원하지 않는 종류의 객체가 포함되는 것을 막을 방법이 없다는 것도 문제다. 이러한 문제들을 지네릭스가 해결해 준다.

지네릭스의 장점

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

간단히 얘기하면 다룰 객체의 타입을 미리 명시해줌으로써 번거로운 형변환을 줄여준다는 얘기다. 처음부터 너무 어렵게 생각하지 말자.

1.2 지네릭 클래스의 선언

지네릭 타입은 클래스와 메서드에 선언할 수 있는데, 먼저 클래스에 선언하는 지네릭 타입에 대해서 알아보자. 예를 들어 클래스 `Box`가 다음과 같이 정의되어 있다고 가정하자.

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

이 클래스를 지네릭 클래스로 변경하면 다음과 같이 클래스 옆에 '<T>'를 붙이면 된다. 그리고 'Object'를 모두 'T'로 바꾼다.

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

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

`Box<T>`에서 T를 '타입 변수(type variable)'라고 하며, 'Type'의 첫 글자에서 따온 것이다. 타입 변수는 T가 아닌 다른 것을 사용해도 된다. `ArrayList<E>`의 경우, 타입 변수 E는 'Element(요소)'의 첫 글자를 따서 사용했다. 타입 변수가 여러 개인 경우에는 `Map<K, V>`와 같이 콤마','를 구분자로 나열하면 된다. K는 Key(키)를 의미하고, V는 Value(값)을 의미한다. 무조건 'T'를 사용하기보다 가능하면, 이처럼 상황에 맞게 의미있는 문자를 선택해서 사용하는 것이 좋다.

이들은 기호의 종류만 다를 뿐 '임의의 참조형 타입'을 의미한다는 것은 모두 같다. 마치 수학식 ' $f(x, y) = x + y$ '가 ' $f(k, v) = k + v$ '와 다르지 않은 것처럼 말이다.

기존에는 다양한 종류의 타입을 다루는 메서드의 매개변수나 리턴타입으로 `Object`타입의 참조변수를 많이 사용했고, 그로 인해 형변환이 불가피했지만, 이젠 `Object`타입 대신 원하는 타입을 지정하기만 하면 되는 것이다.

이제 지네릭 클래스가 된 `Box`클래스의 객체를 생성할 때는 다음과 같이 참조변수와 생성자에 타입 T대신에 사용될 실제 타입을 지정해주어야 한다.

```
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
```

위의 코드에서 타입 T대신에 `String`타입을 지정해줬으므로, 지네릭 클래스 `Box<T>`는 다음과 같이 정의된 것과 같다.

```
class Box<String> { // 지네릭 타입을 String으로 지정
    String item;

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

만일 `Box`클래스에 `String`만 담을 거라면, 타입 변수를 선언하지 않고 위와 같이 직접 타입을 적어주는 것도 가능하다. 단, `Box<String>`클래스는 `String`타입만 담을 수 있다. 반면에 `Box<T>`클래스는 어떤 타입이든 한 가지 타입을 정해서 담을 수 있다.

지네릭이 도입되기 이전의 코드와 호환을 위해, 지네릭 클래스인데도 예전의 방식으로 객체를 생성하는 것이 허용된다. 다만 지네릭 타입을 지정하지 않아서 안전하지 않다는 경고가 발생한다.

```
Box b = new Box();           // OK. T는 Object로 간주된다.
b.setItem("ABC");          // 경고. unchecked or unsafe operation
b.setItem(new Object());    // 경고. unchecked or unsafe operation
```

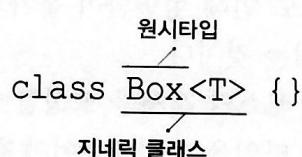
아래와 같이 타입 변수 T에 Object타입을 지정하면, 타입을 지정하지 않은 것이 아니라 알고 적은 것이므로 경고는 발생하지 않는다.

```
Box<Object> b = new Box<Object>();
b.setItem("ABC");          // 경고발생 안함
b.setItem(new Object());    // 경고발생 안함
```

지네릭스가 도입되기 이전의 코드와 호환성을 유지하기 위해서 지네릭스를 사용하지 않은 코드를 허용하는 것일 뿐, 앞으로 지네릭 클래스를 사용할 때는 반드시 타입을 지정해서 지네릭스와 관련된 경고가 나오지 않도록 하자.

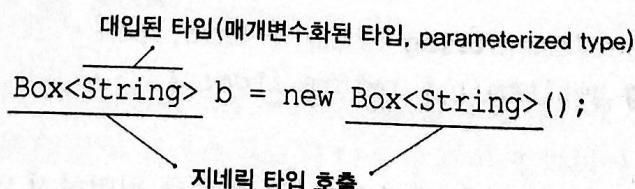
지네릭스의 용어

지네릭스에서 사용되는 용어들은 자칫 혼갈리기 쉽다. 진도를 더 나가기 전에, 지네릭스의 용어를 먼저 정리하고 넘어가자. 다음과 같이 지네릭 클래스 Box가 선언되어 있을 때,



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)

타입 문자 T는 지네릭 클래스 Box<T>의 타입 변수 또는 타입 매개변수라고 하는데, 메서드의 매개변수와 유사한 면이 있기 때문이다. 그래서 아래와 같이 타입 매개변수에 타입을 지정하는 것을 ‘지네릭 타입 호출’이라고 하고, 지정된 타입 ‘String’을 ‘매개변수화된 타입(parameterized type)’이라고 한다. 매개변수화된 타입이라는 용어가 좀 길어서, 앞으로 이 용어 대신 ‘대입된 타입’이라는 용어를 사용할 것이다.



예를 들어, Box<String>과 Box<Integer>는 지네릭 클래스 Box<T>에 서로 다른 타입을 대입하여 호출한 것일 뿐, 이 둘이 별개의 클래스를 의미하는 것은 아니다. 이는 마치 매개변수의 값이 다른 메서드 호출, 즉 add(3,5)와 add(2,4)가 서로 다른 메서드를 호출하는 것이 아닌 것과 같다.

컴파일 후에 `Box<String>`과 `Box<Integer>`는 이들의 '원시 타입'인 `Box`로 바뀐다. 즉, 지네릭 타입이 제거된다. 이에 대해서는 '1.8 지네릭 타입의 제거'에서 자세히 설명한다.

지네릭스의 제한

지네릭 클래스 `Box`의 객체를 생성할 때, 객체별로 다른 타입을 지정하는 것은 적절하다. 지네릭스는 이처럼 인스턴스별로 다르게 동작하도록 하려고 만든 기능이니까.

```
Box<Apple> appleBox = new Box<Apple>(); // OK. Apple 객체만 저장 가능
Box<Grape> grapeBox = new Box<Grape>(); // OK. Grape 객체만 저장 가능
```

그러나 모든 객체에 대해 동일하게 동작해야하는 `static`멤버에 타입 변수 `T`를 사용할 수 없다. `T`는 인스턴스변수로 간주되기 때문이다. 이미 알고 있는 것처럼 `static`멤버는 인스턴스변수를 참조할 수 없다.

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

`static`멤버는 타입 변수에 지정된 타입, 즉 대입된 타입의 종류에 관계없이 동일한 것이어야 하기 때문이다. 즉, '`Box<Apple>.item`'과 '`Box<Grape>.item`'이 다른 것이어서는 안 된다는 뜻이다. 그리고 지네릭 타입의 배열을 생성하는 것도 허용되지 않는다. 지네릭 배열 타입의 참조변수를 선언하는 것은 가능하지만, '`new T[10]`'과 같이 배열을 생성하는 것은 안 된다는 뜻이다.

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

지네릭 배열을 생성할 수 없는 것은 `new`연산자 때문인데, 이 연산자는 컴파일 시점에 타입 `T`가 뭔지 정확히 알아야 한다. 그런데 위의 코드에 정의된 `Box<T>`클래스를 컴파일하는 시점에서는 `T`가 어떤 타입이 될지 전혀 알 수 없다. `instanceof`연산자도 `new`연산자와 같은 이유로 `T`를 피연산자로 사용할 수 없다.

꼭 지네릭 배열을 생성해야 할 필요가 있을 때는, `new`연산자 대신 'Reflection API'의 `newInstance()`와 같이 동적으로 객체를 생성하는 메서드로 배열을 생성하거나, `Object` 배열을 생성해서 복사한 다음에 '`T[]`'로 형변환하는 방법 등을 사용한다.

참고! 지네릭 배열의 실제 구현을 보려면, `src.zip`에서 `Collections.toArray(T[] a)`의 소스를 찾아보자.

1.3 지네릭 클래스의 객체 생성과 사용

지네릭 클래스 Box<T>가 다음과 같이 정의되어 있다고 가정하자. 이 Box<T>의 객체에는 한 가지 종류, 즉 T타입의 객체만 저장할 수 있다. 전과 달리 ArrayList를 이용해서 여러 객체를 저장할 수 있도록 하였다.

```
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 객체를 생성할 때는 다음과 같이 한다. 참조변수와 생성자에 대입된 타입(매개 변수화된 타입)이 일치해야 한다. 일치하지 않으면 에러가 발생한다.

```
Box<Apple> appleBox = new Box<Apple>(); // OK
Box<Apple> appleBox = new Box<Grape>(); // 에러
```

두 타입이 상속관계에 있어도 마찬가지이다. Apple이 Fruit의 자손이라고 가정하자.

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

단, 두 지네릭 클래스의 타입이 상속관계에 있고, 대입된 타입이 같은 것은 괜찮다. FruitBox는 Box의 자손이라고 가정하자.

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

JDK1.7부터는 추정이 가능한 경우 타입을 생략할 수 있게 되었다. 참조변수의 타입으로부터 Box가 Apple타입의 객체만 저장한다는 것을 알 수 있기 때문에, 생성자에 반복해서 타입을 지정해주지 않아도 되는 것이다. 따라서 아래의 두 문장은 동일하다.

```
Box<Apple> appleBox = new Box<Apple>();
Box<Apple> appleBox = new Box<>(); // OK. JDK1.7부터 생략 가능
```

생성된 Box<T>의 객체에 'void add(T item)'으로 객체를 추가할 때, 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 Apple 객체만 추가 가능
```

그러나 타입 T가 'Fruit'인 경우, 'void add(Fruit item)'가 되므로 Fruit의 자손들은 이 메서드의 매개변수가 될 수 있다. Apple이| Fruit의 자손이라고 가정하였다.

```
Box<Fruit> fruitBox = new Box<Fruit>();
fruitBox.add(new Fruit()); // OK.
fruitBox.add(new Apple()); // OK. void add(Fruit item)
```

이제 예제를 통해서 지금까지 배운 내용을 직접 확인해 보자.

▼ 예제 12-1/ch12/FruitBoxEx1.java

```
import java.util.ArrayList;

class Fruit { public String toString() { return "Fruit"; } }
class Apple extends Fruit { public String toString() { return "Apple"; } }
class Grape extends Fruit { public String toString() { return "Grape"; } }
class Toy { public String toString() { return "Toy"; } }

class FruitBoxEx1 {
    public static void main(String[] args) {
        Box<Fruit> fruitBox = new Box<Fruit>();
        Box<Apple> appleBox = new Box<Apple>();
        Box<Toy> toyBox = new Box<Toy>();
//      Box<Grape> grapeBox = new Box<Apple>(); // 에러. 타입 불일치

        fruitBox.add(new Fruit());
        fruitBox.add(new Apple()); // OK. void add(Fruit item)

        appleBox.add(new Apple());
        appleBox.add(new Apple());
//      appleBox.add(new Toy()); // 에러. Box<Apple>에는 Apple만 담을 수 있음

        toyBox.add(new Toy());
//      toyBox.add(new Apple()); // 에러. Box<Toy>에는 Apple을 담을 수 없음

        System.out.println(fruitBox);
        System.out.println(appleBox);
        System.out.println(toyBox);
    } // main의 끝
}

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

▼ 실행결과

```
[Fruit, Apple]
[Apple, Apple]
[Toy]
```

1.4 제한된 지네릭 클래스

타입 문자로 사용할 타입을 명시하면 한 종류의 타입만 저장할 수 있도록 제한할 수 있지만, 그래도 여전히 모든 종류의 타입을 지정할 수 있다는 것에는 변함이 없다. 그렇다면, 타입 매개변수 T에 지정할 수 있는 타입의 종류를 제한할 수 있는 방법은 없을까?

```
FruitBox<Toy> fruitBox = new FruitBox<Toy>();  
fruitBox.add(new Toy()); // OK. 과일상자에 장난감을 담을 수 있다.
```

다음과 같이 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>();  
    ...  
}
```

여전히 한 종류의 타입만 담을 수 있지만, Fruit 클래스의 자손들만 담을 수 있다는 제한이 더 추가된 것이다.

```
FruitBox<Apple> appleBox = new FruitBox<Apple>(); // OK  
FruitBox<Toy> toyBox = new FruitBox<Toy>(); // 에러. Toy는 Fruit의 자손이 아님
```

게다가 add()의 매개변수의 타입 T도 Fruit와 그 자손 타입이 될 수 있으므로, 아래와 같이 여러 과일을 담을 수 있는 상자가 가능하게 된다.

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

다형성에서 조상타입의 참조변수로 자손타입의 객체를 가리킬 수 있는 것처럼, 매개변수화된 타입의 자손 타입도 가능한 것이다. 타입 매개변수 T에 Object를 대입하면, 모든 종류의 객체를 저장할 수 있게 된다.

만일 클래스가 아니라 인터페이스를 구현해야 한다는 제약이 필요하다면, 이때도 'extends'를 사용한다. 'implements'를 사용하지 않는다는 점에 주의하자.

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

클래스 Fruit의 자손이면서 Eatable 인터페이스도 구현해야 한다면 아래와 같이 '&' 기호로 연결한다.

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

이제 FruitBox에는 Fruit의 자손이면서 Eatable을 구현한 클래스만 타입 매개변수 T에 대입될 수 있다.

▼ 예제 12-2/ch12/FruitBoxEx2.java

```

import java.util.ArrayList;

class Fruit implements Eatable {
    public String toString() { return "Fruit"; }
}

class Apple extends Fruit { public String toString() { return "Apple"; } }
class Grape extends Fruit { public String toString() { return "Grape"; } }
class Toy { public String toString() { return "Toy"; } }

interface Eatable {}

class FruitBoxEx2 {
    public static void main(String[] args) {
        FruitBox<Fruit> fruitBox = new FruitBox<Fruit>();
        FruitBox<Apple> appleBox = new FruitBox<Apple>();
        FruitBox<Grape> grapeBox = new FruitBox<Grape>();
//        FruitBox<Grape> grapeBox = new FruitBox<Apple>(); // 에러. 타입 불일치
//        FruitBox<Toy> toyBox = new FruitBox<Toy>(); // 에러.

        fruitBox.add(new Fruit());
        fruitBox.add(new Apple());
        fruitBox.add(new Grape());
        appleBox.add(new Apple());
//        appleBox.add(new Grape()); // 에러. Grape는 Apple의 자손이 아님
        grapeBox.add(new Grape());

        System.out.println("fruitBox-"+fruitBox);
        System.out.println("appleBox-"+appleBox);
        System.out.println("grapeBox-"+grapeBox);
    } // main
}

class FruitBox<T extends Fruit & Eatable> extends Box<T> {}

class Box<T> {
    ArrayList<T> list = new ArrayList<T>();
    void add(T item) { list.add(item); }
    T get(int i) { return list.get(i); }
    int size() { return list.size(); }
    public String toString() { return list.toString(); }
}

```

▼ 실행결과

```

fruitBox-[Fruit, Apple, Grape]
appleBox-[Apple]
grapeBox-[Grape]

```

1.5 와일드 카드

매개변수에 과일박스를 대입하면 주스를 만들어서 반환하는 Juicer라는 클래스가 있고, 이 클래스에는 과일을 주스로 만들어서 반환하는 makeJuice()라는 static메서드가 다음과 같이 정의되어 있다고 가정하자.

```
class Juicer {
    static Juice makeJuice(FruitBox<Fruit> box) { // <Fruit>으로 지정
        String tmp = "";
        for(Fruit f : box.getList()) tmp += f + " ";
        return new Juice(tmp);
    }
}
```

Juicer클래스는 지네릭 클래스가 아닌데다, 지네릭 클래스라고 해도 static메서드에는 타입 매개변수 T를 매개변수에 사용할 수 없으므로 아예 지네릭스를 적용하지 않던가, 위와 같이 타입 매개변수 대신, 특정 타입을 지정해줘야 한다.

```
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)); // 에러. FruitBox<Apple>
```

이렇게 지네릭 타입을 'FruitBox<Fruit>'로 고정해 놓으면, 위의 코드에서 알 수 있듯이 'FruitBox<Apple>'타입의 객체는 makeJuice()의 매개변수가 될 수 없으므로, 다음과 같이 여러 가지 타입의 매개변수를 갖는 makeJuice()를 만들 수밖에 없다.

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

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

그러나 위와 같이 오버로딩하면, 컴파일 에러가 발생한다. 지네릭 타입이 다른 것만으로는 오버로딩이 성립하지 않기 때문이다. 지네릭 타입은 컴파일러가 컴파일할 때만 사용하고 제거해버린다. 그래서 위의 두 메서드는 오버로딩이 아니라 '메서드 중복 정의'이다. 이럴 때 사용하기 위해 고안된 것이 바로 '와일드 카드'이다. 와일드 카드는 기호 '?'로 표현하는데, 와일드 카드는 어떠한 타입도 될 수 있다.

'?'만으로는 Object타입과 다를 게 없으므로, 다음과 같이 'extends'와 'super'로 상한(upper bound)과 하한(lower bound)을 제한할 수 있다.

- | | |
|----------------------------|--|
| <? 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 |

| 참고 | 지네릭 클래스와 달리 와일드 카드에는 '&'를 사용할 수 없다. 즉, <? extends T & E>와 같이 할 수 없다.

와일드 카드를 사용해서 makeJuice()의 매개변수 타입을 FruitBox<Fruit>에서 FruitBox <? extends Fruit>으로 바꾸면 다음과 같이 된다.

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

이제 이 메서드의 매개변수로 FruitBox<Fruit>뿐만 아니라, FruitBox<Apple>와 FruitBox<Grape>도 가능하게 된다.

| 참고 | 매개변수의 타입을 'FruitBox<? extends Object>'로 하면, 모든 종류의 FruitBox가 매개변수로 가능하다.

```
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>
```

매개변수의 타입을 FruitBox<? extends Object>로 하면, 모든 종류의 FruitBox가 이 메서드의 매개변수로 가능해진다. 대신, 전과 달리 box의 요소가 Fruit의 자손이라는 보장이 없으므로 아래의 for문에서 box에 저장된 요소를 Fruit타입의 참조변수로 못받는다.

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

그러나 실제로 테스트 해보면 문제없이 컴파일되는데 그 이유는 바로 지네릭 클래스 FruitBox를 제한했기 때문이다.

```
class FruitBox<T extends Fruit> extends Box<T> {}
```

컴파일리는 위 문장으로부터 모든 FruitBox의 요소들이 Fruit의 자손이라는 것을 알고 있으므로 문제 삼지 않는 것이다.

▼ 예제 12-3/ch12/FruitBoxEx3.java

```

import java.util.ArrayList;

class Fruit { public String toString() { return "Fruit";}}
class Apple extends Fruit { public String toString() { return "Apple";}}
class Grape extends Fruit { public String toString() { return "Grape";}}

class Juice {
    String name;
    Juice(String name) { this.name = name + "Juice"; }
    public String toString() { return name; }
}

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

class FruitBoxEx3 {
    public static void main(String[] args) {
        FruitBox<Fruit> fruitBox = new FruitBox<Fruit>();
        FruitBox<Apple> appleBox = new FruitBox<Apple>();

        fruitBox.add(new Apple());
        fruitBox.add(new Grape());
        appleBox.add(new Apple());
        appleBox.add(new Apple());

        System.out.println(Juicer.makeJuice(fruitBox));
        System.out.println(Juicer.makeJuice(appleBox));
    } // main
}

class FruitBox<T extends Fruit> extends Box<T> {}

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(); }
}

```

▼ 실행결과

Apple	Grape	Juice
Apple	Apple	Juice

다음의 예제는 'super'로 와일드 카드의 하한을 제한하는 경우에 대한 예를 보여준다. 일단 예제부터 먼저 보자.

▼ 예제 12-4/ch12/FruitBoxEx4.java

```
import java.util.*;  
class Fruit {  
    String name;  
    int weight;  
    Fruit(String name, int weight) {  
        this.name = name;  
        this.weight = weight;  
    }  
    public String toString() { return name+"("+weight+");" }  
}  
class Apple extends Fruit {  
    Apple(String name, int weight) {  
        super(name, weight);  
    }  
}  
class Grape extends Fruit {  
    Grape(String name, int weight) {  
        super(name, weight);  
    }  
}  
class AppleComp implements Comparator<Apple> {  
    public int compare(Apple t1, Apple t2) {  
        return t2.weight - t1.weight;  
    }  
}  
class GrapeComp implements Comparator<Grape> {  
    public int compare(Grape t1, Grape t2) {  
        return t2.weight - t1.weight;  
    }  
}  
class FruitComp implements Comparator<Fruit> {  
    public int compare(Fruit t1, Fruit t2) {  
        return t1.weight - t2.weight;  
    }  
}  
  
class FruitBoxEx4 {  
    public static void main(String[] args) {  
        FruitBox<Apple> appleBox = new FruitBox<Apple>();  
        FruitBox<Grape> grapeBox = new FruitBox<Grape>();  
        appleBox.add(new Apple("GreenApple", 300));  
        appleBox.add(new Apple("GreenApple", 100));  
        appleBox.add(new Apple("GreenApple", 200));  
        grapeBox.add(new Grape("GreenGrape", 400));  
        grapeBox.add(new Grape("GreenGrape", 300));  
        grapeBox.add(new Grape("GreenGrape", 200));  
    }  
}
```

```

Collections.sort(appleBox.getList(), new AppleComp());
Collections.sort(grapeBox.getList(), new GrapeComp());
System.out.println(appleBox);
System.out.println(grapeBox);
System.out.println();
Collections.sort(appleBox.getList(), new FruitComp());
Collections.sort(grapeBox.getList(), new FruitComp());
System.out.println(appleBox);
System.out.println(grapeBox);
} // main
}

class FruitBox<T extends Fruit> extends Box<T> {}
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();
    }
}

```

▼ 실행결과

```
[GreenApple(300), GreenApple(200), GreenApple(100)]
[GreenGrape(400), GreenGrape(300), GreenGrape(200)]
```

```
[GreenApple(100), GreenApple(200), GreenApple(300)]
[GreenGrape(200), GreenGrape(300), GreenGrape(400)]
```

이 예제는 `Collections.sort()`를 이용해서 `appleBox`와 `grapeBox`에 담긴 파일을 무게별로 정렬한다. 이 메서드의 선언부는 다음과 같다.

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

'static' 옆에 있는 '<T>'는 메서드에 선언된 지네릭 타입이다. 이런 메서드를 지네릭 메서드라고 하는데, 다음 단원에서 배울 것이다. 첫 번째 매개변수는 정렬할 대상이고, 두 번째 매개변수는 정렬할 방법이 정의된 `Comparator`이다. `Comparator`의 지네릭 타입에 한 제한이 걸려있는 와일드 카드가 사용되었다. 먼저 다음과 같이 와일드 카드를 사용하지 않았다고 가정해보자.

App
지만
이
적용

위의

매개변
개변수
<Fruit>

Co
Co

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

만일 타입 매개변수 T에 Apple이 대입되면, 위의 정의는 아래와 같이 바뀔 것이다.

```
static void sort(List<Apple> list, Comparator<Apple> c)
```

이것은 List<Apple>을 정렬하기 위해서는 Comparator<Apple>이 필요하다는 것을 의미 한다. 그래서 Comparator<Apple>을 구현한 AppleComp클래스를 아래처럼 정의하였다.

```
class AppleComp implements Comparator<Apple> {
    public int compare(Apple t1, Apple t2) {
        return t2.weight - t1.weight;
    }
}
```

지금까지는 별문제 없어 보인다. 하지만, Apple대신 Grape가 대입된다면? List<Grape>를 정렬하려면, Comparator<Grape>가 필요하다. Comparator<Apple>로는 List<Grape>를 정렬할 수 없기 때문이다.

```
class GrapeComp implements Comparator<Grape> {
    public int compare(Grape t1, Grape t2) {
        return t2.weight - t1.weight;
    }
}
```

AppleComp와 GrapeComp는 타입만 다를 뿐 완전히 같은 코드이다. 코드의 중복도 문제지만, 새로운 Fruit의 자손이 생길 때마다 위와 같은 코드를 반복해서 만들어야 한다는 것이 더 문제다. 이 문제를 해결하기 위해서는 타입 매개변수에 하한 제한의 와일드 카드를 적용해야 한다. 앞서 살펴본 것과 같이 sort()는 원래 그렇게 정의되어 있다.

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

위의 문장에서 타입 매개변수 T에 Apple이 대입되면 다음과 같이 된다.

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

매개변수의 타입이 Comparator<? super Apple>이라는 의미는 Comparator의 타입 매개변수로 Apple과 그 조상이 가능하다는 뜻이다. 즉, Comparator<Apple>, Comparator<Fruit>, Comparator<Object> 중의 하나가 두 번째 매개변수로 올 수 있다는 뜻이다.

Comparator<? super Apple> : Comparator<Apple>, Comparator<Fruit>, Comparator<Object>

Comparator<? super Grape> : Comparator<Grape>, Comparator<Fruit>, Comparator<Object>

그래서 아래와 같이 FruitComp를 만들면, List<Apple>과 List<Grape>를 모두 정렬할 수 있다. 비교의 대상이 되는 weight는 Apple과 Grape의 조상인 Fruit에 정의되어 있기 때문에 가능한 것이기도 하다.

```
class FruitComp implements Comparator<Fruit> {
    public int compare(Fruit t1, Fruit t2) {
        return t1.weight - t2.weight;
    }
}

...
// List<Apple>과 List<Grape>를 모두 Comparator<Fruit>으로 정렬
Collections.sort(appleBox.getList(), new FruitComp());
Collections.sort(grapeBox.getList(), new FruitComp());
```

이러한 장점 때문에 Comparator에는 항상 <? super T>가 습관적으로 따라 붙는다. 와일드 카드 때문에 Comparator를 어려워하는 경우가 많은데, 그럴 때는 그냥 와일드 카드를 무시하고 Comparator<T>라고 생각하기 바란다.

1.6 지네릭 메서드

메서드의 선언부에 지네릭 타입이 선언된 메서드를 지네릭 메서드라 한다. 앞서 살펴본 것처럼, Collections.sort()가 바로 지네릭 메서드이며, 지네릭 타입의 선언 위치는 반환 타입 바로 앞이다.

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

지네릭 클래스에 정의된 타입 매개변수와 지네릭 메서드에 정의된 타입 매개변수는 전혀 별개의 것이다. 같은 타입 문자 T를 사용해도 같은 것이 아니라는 점에 주의해야 한다.

| 참고 | 지네릭 메서드는 지네릭 클래스가 아닌 클래스에도 정의될 수 있다.

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

위의 코드에서 지네릭 클래스 FruitBox에 선언된 타입 매개변수 T와 지네릭 메서드 sort()에 선언된 타입 매개변수 T는 타입 문자만 같을 뿐 서로 다른 것이다. 그리고 sort()가 static메서드라는 것에 주목하자. 앞서 설명한 것처럼, static멤버에는 타입 매개 변수를 사용할 수 없지만, 이처럼 메서드에 지네릭 타입을 선언하고 사용하는 것은 가능

메서드에 선언된 지네릭 타입은 지역 변수를 선언한 것과 같다고 생각하면 이해하기 쉬운데, 이 타입 매개변수는 메서드 내에서만 지역적으로 사용될 것이므로 메서드가 static이건 아니건 상관이 없다.

| 참고 | 같은 이유로 내부 클래스에 선언된 타입 문자가 외부 클래스의 타입 문자와 같아도 구별될 수 있다.

앞서 나왔던 makeJuice()를 지네릭 메서드로 바꾸면 다음과 같다.

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

↓

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

이제 이 메서드를 호출할 때는 아래와 같이 타입 변수에 타입을 대입해야 한다.

```
FruitBox<Fruit> fruitBox = new FruitBox<Fruit>();
FruitBox<Apple> appleBox = new FruitBox<Apple>();

...
System.out.println(Juicer.<Fruit>makeJuice(fruitBox));
System.out.println(Juicer.<Apple>makeJuice(appleBox));
```

그러나 대부분의 경우 컴파일러가 타입을 추정할 수 있기 때문에 생략해도 된다. 위의 코드에서도 fruitBox와 appleBox의 선언부를 통해 대입된 타입을 컴파일러가 추정할 수 있다.

```
System.out.println(Juicer.makeJuice(fruitBox)); // 대입된 타입을 생략할 수 있다.
System.out.println(Juicer.makeJuice(appleBox));
```

한 가지 주의할 점은 지네릭 메서드를 호출할 때, 대입된 타입을 생략할 수 없는 경우에는 참조변수나 클래스 이름을 생략할 수 없다는 것이다.

```
System.out.println(<Fruit>makeJuice(fruitBox)); // 에러. 클래스 이름 생략불가
System.out.println(this.<Fruit>makeJuice(fruitBox)); // OK
System.out.println(Juicer.<Fruit>makeJuice(fruitBox)); // OK
```

같은 클래스 내에 있는 멤버들끼리는 참조변수나 클래스이름, 즉 'this.'이나 '클래스이름.' 을 생략하고 메서드 이름만으로 호출이 가능하지만, 대입된 타입이 있을 때는 반드시 써 줘야 한다. 이것은 단지 기술적인 이유에 의한 규칙이므로 그냥 지키기만 하면 된다.

지네릭 메서드는 매개변수의 타입이 복잡할 때도 유용하다. 만일 아래와 같은 코드가 있다면 타입을 별도로 선언함으로써 코드를 간략히 할 수 있다.

```
public static void printAll(ArrayList<? extends Product> list,
                           ArrayList<? extends Product> list2) {
    for(Unit u : list) {
        System.out.println(u);
    }
}
```

```
public static <T extends Product> void printAll(ArrayList<T> list,
                                                ArrayList<T> list2) {
    for(Unit u : list) {
        System.out.println(u);
    }
}
```

이번엔 좀 복잡하게 선언된 지네릭 메서드 하나를 예를 들어보겠다. 아래의 메서드는 Collections클래스의 sort()인데, 좀 전에 소개한 sort()와 달리 매개변수가 하나짜리이다.

public static <T extends Comparable<? super T>> void sort(List<T> list)
매개변수로 지정한 List<T>를 정렬한다는 것은 알겠는데, 메서드에 선언된 지네릭 타입이
좀 복잡하다. 이럴 때는 일단 와일드 카드를 걷어내자.

public static <T extends Comparable<T>> void sort(List<T> list)
이해하기가 좀 쉬워졌는가? List<T>의 요소가 Comparable인터페이스를 구현한 것이어야 한다는 뜻이다. 앞서 살펴본 것처럼 인터페이스라고 해서 ‘implements’라고 쓰지 않는다.

이제 다시 와일드 카드를 넣고 이해해보자.

이제 다시 와일드 카드를 넣고 이해해보자.

```
public static <T extends Comparable<? super T>> void sort(List<T> list)
```

- ① 타입 T를 요소로 하는 List를 매개변수로 허용한다.
 - ② 'T'는 Comparable을 구현한 클래스이어야 하며 (<T extends Comparable>), 'T' 또는 그 조상
의 타입을 비교하는 Comparable이어야 한다는 것 (Comparable<? super T>)을 의미한다. 만
일 T가 Student이고, Person의 자손이라면, <? super T>는 Student, Person, Object가 모두
가능하다.

1.7 지네릭 타입의 형변환

지네릭 타입과 원시 타입(primitive type)간의 형변환이 가능할까? 잠시 생각해 본 다음에 아래의 코드를 보자.

```
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
```

위에서 알 수 있듯이, 지네릭 타입과 넌지네릭(non-generic) 타입간의 형변환은 항상 가능하다. 다만 경고가 발생할 뿐이다. 그러면, 대입된 타입이 다른 지네릭 타입 간에는 형변환이 가능할까?

```
Box<Object> objBox = null;
Box<String> strBox = null;

objBox = (Box<Object>) strBox; // 에러. Box<String> → Box<Object>
strBox = (Box<String>) objBox; // 에러. Box<Object> → Box<String>
```

불가능하다. 대입된 타입이 Object일지라도 말이다. 이 사실은 이미 배웠다. 아래의 문장이 안 된다는 얘기는 Box<String>이 Box<Object>로 형변환될 수 없다는 사실을 간접적으로 알려주는 것이기 때문이다.

```
// Box<Object> objBox = (Box<Object>) new Box<String>();
Box<Object> objBox = new Box<String>(); // 에러. 형변환 불가능
```

그러면 다음의 문장은 어떨까? Box<String>이 Box<? extends Object>로 형변환될까?

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

형변환이 된다. 그래서 전에 배운 makeJuice메서드의 매개변수에 다형성이 적용될 수 있었던 것이다.

```
// 매개변수로 FruitBox<Fruit>, FruitBox<Apple>, FruitBox<Grape> 등이 가능
static Juice makeJuice(FruitBox<? extends Fruit> box) { ... }

FruitBox<? extends Fruit> box = new FruitBox<Fruit>(); // OK
FruitBox<? extends Fruit> box = new FruitBox<Apple>(); // OK
FruitBox<? extends Fruit> box = new FruitBox<Grape>(); // OK
```

반대로의 형변환도 성립하지만, 확인되지 않은 형변환이라는 경고가 발생한다. 'FruitBox<? extends Fruit>'에 대입될 수 있는 타입이 여러 개인데다, FruitBox<Apple>를 제외한 다른 타입은 FruitBox<Apple>로 형변환될 수 없기 때문이다.

```

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

```

이번엔 좀 실질적인 예를 살펴보자. 다음은 `java.util.Optional` 클래스의 실제 소스의 일부이다. 지금까지 배운 내용들을 떠올리며, 아래의 코드를 자세히 살펴보자.

```

public final class Optional<T> {
    private static final Optional<?> EMPTY = new Optional<>();
    private final T value;
    ...
    public static<T> Optional<T> empty() {
        Optional<T> t = (Optional<T>) EMPTY;
        return t;
    }
    ...
}

```

`static`상수 `EMPTY`에 들어있는 `Optional` 객체를 생성해서 저장했다가 `empty()`를 호출하면 `EMPTY`를 형변환해서 반환한다. 먼저 상수를 선언하는 문장을 단계별로 분석해보면 다음과 같다. 편의상 제어자는 생략하였다.

```

Optional<?> EMPTY = new Optional<>();
→ Optional<? extends Object> EMPTY = new Optional<>();
→ Optional<? extends Object> EMPTY = new Optional<Object>();

```

`<?>`는 `<? extends Object>`를 줄여 쓴 것이며, `<>`안에 생략된 타입은 ‘?’가 아니라 ‘`Object`’이다.

```

Optional<?> EMPTY = new Optional<?>(); // 예러. 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.

위의 문장에서 `EMPTY`의 타입을 `Optional<Object>`가 아닌 `Optional<?>`로 한 이유는 `Optional<T>`로 형변환이 가능하기 때문이다.

```

Optional<?> wopt = new Optional<Object>();
Optional<Object> oopt = new Optional<Object>();
Optional<String> sopt = (Optional<String>) wopt; // OK. 형변환 가능
Optional<String> sopt = (Optional<String>) oopt; // 예러. 형변환 불가

```

`empty()`의 반환 타입이 `Optional<T>`이므로 `EMPTY`를 `Optional<T>`로 형변환해야 하는데, 위의 코드에서 알 수 있는 것처럼 `Optional<Object>`는 `Optional<T>`로 형변환이 불가능하다.

```

public static<T> Optional<T> empty() {
    Optional<T> t = (Optional<T>) EMPTY; // Optional<?> → Optional<T>
    return t;
}

```

정리하면, `Optional<Object>`를 `Optional<String>`으로 직접 형변환하는 것은 불가능하지만, 와일드 카드가 포함된 지네릭 타입으로 형변환하면 가능하다. 대신 확인되지 않은 타입으로의 형변환이라는 경고가 발생한다.

```

Optional<Object> → Optional<T>                                // 형변환 불가능.
Optional<Object> → Optional<?> → Optional<T> // 형변환 가능. 경고발생

```

마지막으로 하나만 덧붙이면, 다음과 같이 와일드 카드가 사용된 지네릭 타입끼리도 다음과 같은 경우에는 형변환이 가능하다.

```

FruitBox<? extends Object> objBox = null;
FruitBox<? extends String> strBox = null;

strBox = (FruitBox<? extends String>) objBox; // OK. 미확정 타입으로 형변환 경고
objBox = (FruitBox<? extends Object>) strBox; // OK. 미확정 타입으로 형변환 경고

```

형변환이 가능하긴 하지만, 와일드 카드는 타입이 확정된 타입이 아니므로 컴파일러는 미확정 타입으로 형변환하는 것이라고 경고한다.

1.8 지네릭 타입의 제거

컴파일러는 지네릭 타입을 이용해서 소스파일을 체크하고, 필요한 곳에 형변환을 넣어준다. 그리고 지네릭 타입을 제거한다. 즉, 컴파일된 파일(*.class)에는 지네릭 타입에 대한 정보가 없는 것이다.

이렇게 하는 주된 이유는 지네릭이 도입되기 이전의 소스 코드와의 호환성을 유지하기 위해서이다. JDK1.5부터 지네릭스가 도입되었지만, 아직도 원시 타입을 사용해서 코드를 작성하는 것을 허용한다. 그러나 앞으로 가능하면 원시 타입을 사용하지 않도록 하자. 언젠가는 분명히 새로운 기능을 위해 하위 호환성을 포기하게 될 때가 올 것이기 때문이다.

지네릭 타입의 제거 과정은 꽤 복잡하기 때문에 자세히 설명하기는 어렵다. 기본적인 제거과정에 대해서만 살펴볼 것이다.

1. 지네릭 타입의 경계(bound)를 제거한다.

지네릭 타입이 `<T extends Fruit>`라면 T는 `Fruit`로 치환된다. `<T>`인 경우는 T는 `Object`로 치환된다. 그리고 클래스 옆의 선언은 제거된다.

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



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

2. 지네릭 타입을 제거한 후에 타입이 일치하지 않으면, 형변환을 추가한다.

List의 get()은 Object타입을 반환하므로 형변환이 필요하다.

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



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

와일드 카드가 포함되어 있는 경우에는 다음과 같이 적절한 타입으로의 형변환이 추가된다.

```
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);
}
```

지금까지 지네릭스에 대해서 모두 살펴보았는데, 지네릭스의 모든 내용을 담지는 못했다. 지네릭스를 완전히 이해하는 것이 그리 쉬운 일이 아니기 때문에 설명하는데도 어려움이 많았다. 이 단원을 통해 지네릭스를 완전히 이해하기보다 우선 기본적인 내용만 이해하고, 다른 장을 공부하면서 부족하다고 느끼는 부분을 다시 복습하는 방식으로 학습하면서 이해의 폭을 넓혀가는 방법을 추천한다.

2. 열거형(Enums)

2.1 열거형이란?

이전까지 자바는 C언어와 달리 열거형이라는 것이 존재하지 않았으나 JDK1.5부터 새로 추가되었다. 자바의 열거형은 C언어의 열거형보다 더 향상된 것으로 열거형이 갖는 값뿐만 아니라 타입까지 관리하기 때문에 보다 논리적인 오류를 줄일 수 있다.

```
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;  
}
```

기존의 많은 언어들, 예를 들어 C언어에서는 타입이 달라도 값이 같으면 조건식 결과가 참(true)이였으나, 자바의 '타입에 안전한 열거형(typesafe enum)'에서는 실제 값이 같아도 타입이 다르면 조건식의 결과가 false가 된다. 이처럼 값뿐만 아니라 타입까지 체크하기 때문에 타입에 안전하다고 하는 것이다.

```
if(Card.CLOVER == Card.TWO) // true지만 false이어야 의미상 맞음.  
if(Card.Kind.CLOVER == Card.Value.TWO) // false. 값은 같지만 타입이 다름
```

그리고 더 중요한 것은 상수의 값이 바뀌면, 해당 상수를 참조하는 모든 소스를 다시 컴파일해야 한다는 것이다. 하지만 열거형 상수를 사용하면, 기존의 소스를 다시 컴파일하지 않아도 된다.

2.2 열거형의 정의와 사용

열거형을 정의하는 방법은 간단하다. 다음과 같이 팔호{}안에 상수의 이름을 나열하기만 하면 된다.

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

예를 들어 동서남북 4방향을 상수로 정의하는 열거형 Direction은 다음과 같다.

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

이 열거형에 정의된 상수를 사용하는 방법은 ‘열거형이름.상수명’이다. 클래스의 static 변수를 참조하는 것과 동일하다.

```
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. equals()가 아닌 ‘==’로 비교가 가능하다는 것은 그만큼 빠른 성능을 제공한다는 얘기다. 그러나 ‘<, >’와 같은 비교연산자는 사용할 수 없고 compareTo()는 사용가능하다. 앞서 배운 것과 같이 compareTo()는 두 비교대상이 같으면 0, 왼쪽이 크면 양수, 오른쪽이 크면 음수를 반환한다.

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

다음과 같이 switch문의 조건식에도 열거형을 사용할 수 있다.

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

이 때 주의할 점은 case문에 열거형의 이름은 적지 않고 상수의 이름만 적어야 한다는 제약이 있다. 아마도 그렇게 하는 것이 오타도 줄일 수 있고 보기에도 간결하기 때문인 것 같다.

모든 열거형의 조상 – java.lang.Enum

열거형 Direction에 정의된 모든 상수를 출력하려면, 다음과 같이 한다.

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

values()는 열거형의 모든 상수를 배열에 담아 반환한다. 이 메서드는 모든 열거형이 가지고 있는 것으로 컴파일러가 자동으로 추가해 준다. 그리고 ordinal()은 모든 열거형의 조상인 java.lang.Enum클래스에 정의된 것으로, 열거형 상수가 정의된 순서(0부터 시작)를 정수로 반환한다.

Enum클래스에는 그 밖에도 다음과 같은 메서드가 정의되어 있다.

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.

▲ 표12-1 Enum클래스에 정의된 메서드

이외에도 values()처럼 컴파일러가 자동적으로 추가해주는 메서드가 하나 더 있다.

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

이 메서드는 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
```

이제 예제를 통해 열거형을 직접 정의하고 사용해 보자.

▼ 예제 12-5/ch12/EnumEx1.java

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

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

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

▼ 실행결과

```
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
```

2.3 열거형에 멤버 추가하기

Enum클래스에 정의된 ordinal()이 열거형 상수가 정의된 순서를 반환하지만, 이 값을 열거형 상수의 값으로 사용하지 않는 것이 좋다. 이 값은 내부적인 용도로만 사용되기 위한 것이기 때문이다.

열거형 상수의 값이 불연속적인 경우에는 이때는 다음과 같이 열거형 상수의 이름 옆에 원하는 값을 팔호()와 함께 적어주면 된다.

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

그리고 지정된 값을 저장할 수 있는 인스턴스 변수와 생성자를 새로 추가해 주어야 한다. 이 때 주의할 점은, 먼저 열거형 상수를 모두 정의한 다음에 다른 멤버들을 추가해야 한다는 것이다. 그리고 열거형 상수의 마지막에 ';'도 잊지 말아야 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 인스턴스 변수는 반드시 final이어야 한다는 제약은 없지만, value는 열거형 상수의 값을 저장하기 위한 것이므로 final을 붙였다. 그리고 외부에서 이 값을 얻을 수 있게 getValue()도 추가하였다.

Direction d = new Direction(1); // 에러. 열거형의 생성자는 외부에서 호출불가
열거형 Direction에 새로운 생성자가 추가되었지만, 위와 같이 열거형의 객체를 생성할 수 없다. 열거형의 생성자는 제어자가 묵시적으로 private이기 때문이다.

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

필요하다면, 다음과 같이 하나의 열거형 상수에 여러 값을 지정할 수도 있다. 다만 그에 맞게 인스턴스 변수와 생성자 등을 새로 추가해주어야 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; }
}

```

▼ 예제 12-6/ch12/EnumEx2.java

```

enum Direction {
    EAST(1, ">"), SOUTH(2, "V"), WEST(3, "<"), NORTH(4, "^");

    private static final Direction[] DIR_ARR = Direction.values();
    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; }

    public static Direction 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 Direction 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();
    }
} // enum Direction

class EnumEx2 {
    public static void main(String[] args) {
        for(Direction d : Direction.values())
            System.out.printf("%s=%d%n", d.name(), d.getValue());
        Direction d1 = Direction.EAST;
        Direction d2 = Direction.of(1);

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

```

▼ 실행결과

```

EAST=1
SOUTH=2
WEST=3
NORTH=4
d1=EAST, 1
d2=EAST, 1
SOUTH<
WEST<
NORTH^
WEST<

```

```

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

```

열거형에 추상 메서드 추가하기

열거형 Transportation은 운송 수단의 종류 별로 상수를 정의하고 있으며, 각 운송 수단에는 기본요금(BASIC_FARE)이 책정되어 있다.

```

enum Transportation {
    BUS(100), TRAIN(150), SHIP(100), AIRPLANE(300);

    private final int BASIC_FARE;

    private Transportation(int basicFare) {
        BASIC_FARE = basicFare;
    }

    int fare() { // 운송 요금을 반환
        return BASIC_FARE;
    }
}

```

그러나 이것만으로는 부족하다. 거리에 따라 요금을 계산하는 방식이 각 운송 수단마다 다를 것이기 때문이다. 이럴 때, 열거형에 추상 메서드 'fare(int distance)'를 선언하면 각 열거형 상수가 이 추상 메서드를 반드시 구현해야 한다.

```

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 접근 가능
    Transportation(int basicFare) {
        BASIC_FARE = basicFare;
    }
    public int getBasicFare() { return BASIC_FARE; }
}

```

위의 코드는 열거형에 정의된 추상 메서드를 각 상수가 어떻게 구현하는지 보여준다. 마치 익명 클래스를 작성한 것처럼 보일 정도로 유사하다.

▼ 예제 12-7/ch12/EnumEx3.java

```
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; } };  
  
    protected final int BASIC_FARE; // protected로 해야 각 상수에서 접근 가능  
  
    Transportation(int basicFare) { // private Transportation(int basicFare)  
        BASIC_FARE = basicFare;  
    }  
  
    public int getBasicFare() { return BASIC_FARE; }  
  
    abstract int fare(int distance); // 거리에 따른 요금 계산  
}  
  
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
```

예제에서는 각 열거형 상수가 추상 메서드 fare()를 똑같은 내용으로 구현했지만, 다르게 구현될 수도 있게 하려고 추상 메서드로 선언한 것이다.

열거형에 추상 메서드를 선언할 일은 그리 많지 않으므로 가볍게 참고만 하자.

2.4 열거형의 이해

지금까지 열거형에 대해서 모두 살펴봤는데, 열거형의 이해를 돋기 위해 마지막으로 열거형이 내부적으로 어떻게 구현되었는지에 대해 설명하고자 한다.
만일 열거형 Direction이 다음과 같이 정의되어 있을 때,

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

사실은 열거형 상수 하나하나가 Direction 객체이다. 위의 문장을 클래스로 정의한다면 다음과 같을 것이다.

```

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;
    }
}

```

Direction 클래스의 static 상수 EAST, SOUTH, WEST, NORTH의 값은 객체의 주소이고, 이 값은 바꿔지 않는 값이므로 '=='로 비교가 가능한 것이다.

모든 열거형은 추상 클래스 Enum의 자손이므로, Enum을 흉내 내어 MyEnum을 작성하면 다음과 같다.

```

abstract class MyEnum<T extends MyEnum<T>> implements Comparable<T> {
    static int id = 0; // 객체에 붙일 일련번호 (0부터 시작)

    int ordinal;
    String name = "";

    public int ordinal() { return ordinal; }

    MyEnum(String name) {
        this.name = name;
        ordinal = id++; // 객체를 생성할 때마다 id의 값을 증가시킨다.

    }

    public int compareTo(T t) {
        return ordinal - t.ordinal();
    }
}

```

앞서 6장에서 배운 것과 같이 객체가 생성될 때마다 번호를 붙여서 인스턴스 변수 ordinal에 저장한다. 그리고 Comparable 인터페이스를 구현해서 열거형 상수 간의 비교가 가능하도록 되어 있다. 구현 내용은 간단하다. 두 열거형 상수의 ordinal 값을 서로 빼주기만 하면 된다. 만일 클래스를 MyEnum<T>와 같이 선언하였다면, compareTo()를 위와 같이 간단히 작성할 수 없었을 것이다. 타입 T에 ordinal()이 정의되어 있는지 확인할 수 없기 때문이다.

```

abstract class MyEnum<T> implements Comparable<T> {
    ...
    public int compareTo(T t) {
        return ordinal - t.ordinal(); // 예러. 타입 T에 ordinal()이 있나?
    }
}

```

그래서 MyEnum<T extends MyEnum<T>>와 같이 선언한 것이며, 이것은 타입 T가 MyEnum<T>의 자손이어야 한다는 의미이다. 타입 T가 MyEnum의 자손이므로 ordinal()이 정의되어 있는 것은 분명하므로 형변환 없이도 에러가 나지 않는다.

그리고 추상 메서드를 새로 추가하면, 클래스 앞에도 'abstract'를 붙여줘야 하고, 각 static 상수들도 추상 메서드를 구현해주어야 한다. 아래의 코드에서는 익명 클래스의 형태로 추상 메서드를 구현하였다.

```
abstract class Direction extends MyEnum {
    static final Direction EAST = new Direction("EAST") { // 익명 클래스
        Point move(Point p) { /* 내용 생략 */ }
    };
    static final Direction SOUTH = new Direction("SOUTH") {
        Point move(Point p) { /* 내용 생략 */ }
    };
    static final Direction WEST = new Direction("WEST") {
        Point move(Point p) { /* 내용 생략 */ }
    };
    static final Direction SOUTH = new Direction("SOUTH") {
        Point move(Point p) { /* 내용 생략 */ }
    };

    private String name;

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

    abstract Point move(Point p);
}
```

이제 왜 열거형에 추상 메서드를 추가하면 각 열거형 상수가 추상 메서드를 구현해야하는지 이해가 될 것이다.

이제 예제를 통해서 지금까지 배운 것들을 직접 확인해 보자.

▼ 예제 12-8/ch12/EnumEx4.java

```
abstract class MyEnum<T extends MyEnum<T>> implements Comparable<T> {
    static int id = 0;
    int ordinal;
    String name = "";

    public int ordinal() { return ordinal; }

    MyEnum(String name) {
        this.name = name;
        ordinal = id++;
    }

    public int compareTo(T t) {
        return ordinal - t.ordinal();
    }
}
```

```

abstract class MyTransportation extends MyEnum {
    static final MyTransportation BUS = new MyTransportation("BUS", 100) {
        int fare(int distance) { return distance * BASIC_FARE; }
    };
    static final MyTransportation TRAIN = new MyTransportation("TRAIN", 150) {
        int fare(int distance) { return distance * BASIC_FARE; }
    };
    static final MyTransportation SHIP = new MyTransportation("SHIP", 100) {
        int fare(int distance) { return distance * BASIC_FARE; }
    };
    static final MyTransportation AIRPLANE =
        new MyTransportation("AIRPLANE", 300) {
            int fare(int distance) { return distance * BASIC_FARE; }
        };
    abstract int fare(int distance); // 추상 메서드
    protected final int BASIC_FARE;
    private MyTransportation(String name, int basicFare) {
        super(name);
        BASIC_FARE = basicFare;
    }
    public String name() { return name; }
    public String toString() { return name; }
}

class EnumEx4 {
    public static void main(String[] args) {
        MyTransportation t1 = MyTransportation.BUS;
        MyTransportation t2 = MyTransportation.BUS;
        MyTransportation t3 = MyTransportation.TRAIN;
        MyTransportation t4 = MyTransportation.SHIP;
        MyTransportation t5 = MyTransportation.AIRPLANE;
        System.out.printf("t1=%s, %d\n", t1.name(), t1.ordinal());
        System.out.printf("t2=%s, %d\n", t2.name(), t2.ordinal());
        System.out.printf("t3=%s, %d\n", t3.name(), t3.ordinal());
        System.out.printf("t4=%s, %d\n", t4.name(), t4.ordinal());
        System.out.printf("t5=%s, %d\n", t5.name(), t5.ordinal());
        System.out.println("t1==t2 ? "+(t1==t2));
        System.out.println("t1.compareTo(t3)=" + t1.compareTo(t3));
    }
}

```

▼ 실행결과

```

t1=BUS, 0
t2=BUS, 0
t3=TRAIN, 1
t4=SHIP, 2
t5=AIRPLANE, 3
t1==t2 ? true
t1.compareTo(t3)=-1

```

3. 애너테이션(annotation)

3.1 애너테이션이란?

자바를 개발한 사람들은 소스코드에 대한 문서를 따로 만들기보다 소스코드와 문서를 하나의 파일로 관리하는 것이 낫다고 생각했다. 그래서 소스코드의 주석`/** ~ */`에 소스코드에 대한 정보를 저장하고, 소스코드의 주석으로부터 HTML문서를 생성해내는 프로그램(javadoc.exe)을 만들어서 사용했다. 다음은 모든 애너테이션의 조상인 Annotation 인터페이스의 소스코드의 일부이다.

```
/**  
 * 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 {  
...  
}
```

'/**'로 시작하는 주석 안에 소스코드에 대한 설명들이 있고, 그 안에 '@'이 붙은 태그들이 눈에 띌 것이다. 미리 정의된 태그들을 이용해서 주석 안에 정보를 저장하고, javadoc.exe라는 프로그램이 이 정보를 읽어서 문서를 작성하는데 사용한다.

이 기능을 응용하여, 프로그램의 소스코드 안에 다른 프로그램을 위한 정보를 미리 약속된 형식으로 포함시킨 것이 바로 애너테이션이다. 애너테이션은 주석(comment)처럼 프로그래밍 언어에 영향을 미치지 않으면서도 다른 프로그램에게 유용한 정보를 제공할 수 있다는 장점이 있다.

| 참고 | 애너테이션(annotation)의 뜻은 주석, 주해, 메모이다.

예를 들어, 자신이 작성한 소스코드 중에서 특정 메서드만 테스트하기를 원한다면, 다음과 같이 '@Test'라는 애너테이션을 메서드 앞에 붙인다. '@Test'는 '이 메서드를 테스트해야 한다'는 것을 테스트 프로그램에게 알리는 역할을 할 뿐, 메서드가 포함된 프로그램 자체에는 아무런 영향을 미치지 않는다. 주석처럼 존재하지 않는 것이나 다름없다.

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

테스트 프로그램에게 테스트할 메서드를 일일이 알려주지 않고, 해당 메서드 앞에 애너테이션만 붙이면 된다니 얼마나 편리한가. 그렇다고 모든 프로그램에게 의미가 있는 것은 아니고, 해당 프로그램에 미리 정의된 종류와 형식으로 작성해야만 의미가 있다. '@Test'는 테스트 프로그램을 제외한 다른 프로그램에게는 아무런 의미가 없는 정보일 것이다.

애너테이션은 JDK에서 기본적으로 제공하는 것과 다른 프로그램에서 제공하는 것들이 있는데, 어느 것이든 그저 약속된 형식으로 정보를 제공하기만 하면 될 뿐이다. 애너테이션이 제공한 정보를 이용해서 내부적으로 어떻게 처리하는 지까지 지금의 학습 단계에서 고민하지 않기 바란다.

JDK에서 제공하는 표준 애너테이션은 주로 컴파일러를 위한 것으로 컴파일러에게 유용한 정보를 제공한다. 그리고 새로운 애너테이션을 정의할 때 사용하는 메타 애너테이션을 제공한다.

| 참고 | JDK에서 제공하는 애너테이션은 'java.lang.annotation' 패키지에 포함되어 있다.

3.2 표준 애너테이션

자바에서 기본적으로 제공하는 애너테이션들은 몇 개 없다. 그나마 이들의 일부는 '메타 애너테이션(meta annotation)'으로 애너테이션을 정의하는데 사용되는 애너테이션의 애너테이션이다. 아직 여러분들은 대부분 새로운 애너테이션을 정의하기보다는 이미 작성된 애너테이션을 사용하는 경우가 많을 것으로 가벼운 마음으로 읽으면 좋을 것 같다.

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 적용 가능한 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)

▲ 표 12-2 자바에서 기본적으로 제공하는 표준 애너테이션(*가 붙은 것은 메타 애너테이션)

메타 애너테이션은 잠시 후에 소개하기로 하고, 메타 애너테이션을 제외한 애너테이션에 대해서 먼저 알아보자.

@Override

메서드 앞에만 붙일 수 있는 애너테이션으로, 조상의 메서드를 오버라이딩하는 것이라는 걸 컴파일러에게 알려주는 역할을 한다. 아래의 코드에서와 같이 오버라이딩할 때 조상 메서드의 이름을 잘못 써도 컴파일러는 이것이 잘못된 것인지 알지 못한다.

```

class Parent {
    void parentMethod() { }

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

```

오버라이딩할 때는 이처럼 메서드의 이름을 잘못 적는 경우가 많는데, 컴파일러는 그저 새로운 이름의 메서드가 추가된 것으로 인식할 뿐이다. 게다가 실행 시에도 오류가 발생하지 않고 조상의 메서드가 호출되므로 어디서 잘못되었는지 알아내기 어렵다.

```

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

```

```

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

```

그러나 위의 오른쪽 코드와 같이 메서드 앞에 '@Override'라고 애너테이션을 붙이면, 컴파일러가 같은 이름의 메서드가 조상에 있는지 확인하고 없으면, 에러메시지를 출력한다. 오버라이딩할 때 메서드 앞에 '@Override'를 붙이는 것이 필수는 아니지만, 알아내기 어려운 실수를 미연에 방지해주므로 반드시 붙이도록 하자.

▼ 예제 12-9/ch12/AnnotationEx1.java

```

class Parent {
    void parentMethod() { }

class Child extends Parent {
    @Override
    void parentmethod() { } // 조상 메서드의 이름을 잘못 적었음.
}

```

▼ 컴파일 결과

```

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

```

이 예제를 컴파일하면 위와 같은 에러메시지가 나타난다. 오버라이딩을 해야 하는데 하지 않았다는 뜻이다. '@Override'를 메서드 앞에 붙이지 않았다면 나타나지 않았을 메시지이다. Child클래스에서 메서드의 이름을 'parentMethod'로 변경하고 다시 컴파일해보자. 이번에 에러메시지가 나타나지 않을 것이다.

@Deprecated

새로운 버전의 JDK가 소개될 때, 새로운 기능이 추가될 뿐만 아니라 기존의 부족했던 기능들을 개선하기도 한다. 이 과정에서 기존의 기능을 대체할 것들이 추가되어도, 이미 여러 곳에서 사용되고 있을지 모르는 기존의 것들을 함부로 삭제할 수 없다.

그래서 생각해낸 방법이 더 이상 사용되지 않는 필드나 메서드에 '@Deprecated'를 붙이는 것이다. 이 애너테이션이 붙은 대상은 다른 것으로 대체되었으니 더 이상 사용하지 않을 것을 권한다는 의미이다. 예를 들어 java.util.Date 클래스의 대부분의 메서드에는 '@Deprecated'가 붙어 있는데, Java API에서 Date 클래스의 getDate()를 보면 아래와 같이 적혀 있다.

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

이 메서드 대신에 JDK1.1부터 추가된 Calendar 클래스의 get()을 사용하라는 얘기다. 기존의 것 대신 새로 추가된 개선된 기능을 사용하도록 유도하는 것이다. 굳이 기존의 것을 사용하겠다면, 아무도 못 말리겠지만 가능하면 '@Deprecated'가 붙은 것들은 사용하지 않아야 한다.

만일 '@Deprecated'가 붙은 대상을 사용하는 코드를 작성하면, 컴파일할 때 아래와 같은 메시지가 나타난다.

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

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

해당 소스파일이 'deprecated'된 대상을 사용하고 있으며, '-Xlint:deprecation' 옵션을 붙여서 다시 컴파일하면 자세한 내용을 알 수 있다는 뜻이다.

▼ 예제 12-10/ch12/AnnotationEx2.java

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

▼ 실행결과

```
c:\jdk1.8\work\ch12>javac AnnotationEx2.java  
Note: AnnotationEx2.java uses or overrides a deprecated API.  
Note: Recompile with -Xlint:deprecation for details.  
c:\jdk1.8\work\ch12>java AnnotationEx2
```

이 예제는 '@Deprecated'가 붙은 대상을 사용해서 고의적으로 위의 메시지가 나타나도록 했다. 메시지가 나타나기는 했지만 컴파일도 실행도 잘되었다. '@Deprecated'가 붙은 대상을 사용하지 않도록 권한 뿐 강제성은 없기 때문이다.

메시지에 나온 대로 '-Xlint:deprecation' 옵션을 붙여서 다시 컴파일하면, 아래와 같이 자세한 내용을 보여준다.

```
C:\jdk1.8\work\ch12>javac -Xlint:deprecation AnnotationEx2.java
AnnotationEx2.java:21: warning: [deprecation] oldField in NewClass has been
deprecated
    nc.oldField = 10;
                           ^
AnnotationEx2.java:22: warning: [deprecation] getOldField() in NewClass has
been deprecated
        System.out.println(nc.getOldField());
                           ^
2 warnings
```

단지 'oldField'와 'getOldField()'가 'deprecated'된 대상인데 사용했다고 알려주는 것일 뿐 별다른 내용은 없다.

@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 반드시 붙이도록 하자.

| 참고 | 함수형 인터페이스는 추상 메서드가 하나뿐이어야 한다는 제약이 있다. p.797을 참고

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

@SuppressWarnings

컴파일러가 보여주는 경고메시지가 나타나지 않게 억제해준다. 이전 예제에서처럼 컴파일러의 경고메시지는 무시하고 넘어갈 수도 있지만, 모두 확인하고 해결해서 컴파일 후에 어떠한 메시지도 나타나지 않게 해야 한다.

그러나 경우에 따라서는 경고가 발생할 것을 알면서도 묵인해야 할 때가 있는데. 이 경고를 그대로 놔두면 컴파일할 때마다 메시지가 나타난다. 이전 예제에서 확인한 것과 같이 '-Xlint' 옵션을 붙이지 않으면 컴파일러는 경고의 자세한 내용은 보여주지 않으므로 다른 경고들을 놓치기 쉽다. 이 때는 묵인해야하는 경고가 발생하는 대상에 반드시 '@SuppressWarnings'을 붙여서 컴파일 후에 어떤 경고 메시지도 나타나지 않게 해야 한다.

'@SuppressWarnings'로 억제할 수 있는 경고 메시지의 종류는 여러 가지가 있는데, JDK의 버전이 올라가면서 앞으로도 계속 추가될 것이다. 이 중에서 주로 사용되는 것은 "deprecation", "unchecked", "rawtypes", "..."이다.

“deprecation”은 앞서 살펴본 것과 같이 ‘@Deprecated’가 붙은 대상을 사용해서 발생하는 경고를, “unchecked”는 지네릭스로 타입을 지정하지 않았을 때 발생하는 경고를, “rawtypes”는 지네릭스를 사용하지 않아서 발생하는 경고를, 그리고 “varargs”는 가변인자의 타입이 지네릭 타입일 때 발생하는 경고를 억제할 때 사용한다.
억제하려는 경고 메시지를 애너테이션의 뒤에 괄호()안에 문자열로 지정하면 된다.

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

만일 둘 이상의 경고를 동시에 억제하려면 다음과 같이 한다. 배열에서처럼 괄호{}를 추가로 사용해야한다는 것에 주의하자.

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

‘@SuppressWarnings’로 억제할 수 있는 경고 메시지의 종류는 JDK의 버전이 올라가면서 계속 추가될 것이기 때문에, 이전 버전에서는 발생하지 않던 경고가 새로운 버전에서는 발생할 수 있다. 새로 추가된 경고 메시지를 억제하려면, 경고 메시지의 종류를 알아야 하는데, -Xlint옵션으로 컴파일해서 나타나는 경고의 내용 중에서 대괄호[] 안에 있는 것이 바로 메시지의 종류이다.

| 참고 | -Xlint:unchecked와 같이 하면 unchecked와 관련된 경고만 표시된다.

```
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
```

위의 경고 메시지를 보면 대괄호[] 안에 “rawtypes”라고 적혀 있으므로, 이 경고 메시지를 억제하려면 다음과 같이 하면 된다.

```
@SuppressWarnings("rawtypes")
public static void sort(List list) {
}
```

▼ 예제 12-11/ch12/AnnotationEx3.java

```
import java.util.ArrayList;
class NewClass {
    int newField;
    int getNewField() {
        return newField;
    }
}
```

```

@Deprecated
int oldField;

@Deprecated
int getOldField() {
    return oldField;
}

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

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

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

```

▼ 실행결과

```

c:\jdk1.8\work\ch12>javac AnnotationEx3.java
c:\jdk1.8\work\ch12>java AnnotationEx3
10

```

main메서드 앞에 '@SuppressWarnings("deprecation")'을 붙여서, 이 메서드 내에서 일어나는 'deprecation'과 관련된 모든 경고가 나타나지 않게 했다. 그리고 타입을 지정하지 않은 ArrayList객체를 생성한 곳에 '@SuppressWarnings("unchecked")'를 붙여서 경고를 억제했다. 두 애너테이션을 합쳐서 아래와 같이 main메서드에서 두 종류의 경고를 모두 억제하도록 할 수도 있다.

```

// main메서드 내의 "deprecation"과 "unchecked" 관련 경고를 모두 억제한다.
@SuppressWarnings({"deprecation", "unchecked"})
public static void main(String args[]) {
    NewClass nc = new NewClass();

    nc.oldField = 10;                // @Depreacted가 붙은 대상을 사용
    System.out.println(nc.getOldField()); // @Depreacted가 붙은 대상을 사용
    ArrayList<NewClass> list = new ArrayList(); // 타입을 지정하지 않음.
}

```

그러나 그렇게 하면 나중에 추가된 코드에서 발생할 수도 있는 경고까지 억제될 수 있으므로 바람직하지 않다. 해당 대상에만 애너테이션을 붙여서 경고의 억제범위를 최소화하는 것이 좋다.

@SafeVarargs

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

이 애너테이션은 생성자와 static이나 final이 붙은 메서드에만 붙일 수 있다. 즉, 오버라이드될 수 있는 메서드에는 사용할 수 없다는 뜻이다.

지네릭스에서 살펴본 것과 같이 어떤 타입들은 컴파일 이후에 제거된다. 컴파일 후에도 제거되지 않는 타입을 reifiable타입이라고 하고, 제거되는 타입을 non-reifiable타입이라고 한다. 지네릭 타입들은 대부분 컴파일 시에 제거되므로 non-reifiable타입이다.

| 참고 | reifiable은 're(다시)' + '-ify(~화 하다)' + '-able(~할 수 있는)'의 합성어로 직역하면, '다시 ~화(化) 할 수 있는'이라는 뜻이다. '리어화이어블'이라고 읽는다. 컴파일 후에도 타입정보가 유지되면 reifiable타입이다.

예를 들어, java.util.Arrays의 asList()는 다음과 같이 정의되어 있으며, 이 메서드는 매개변수로 넘겨받은 값들로 배열을 만들어서 새로운 ArrayList객체를 만들어서 반환하는데 이 과정에서 경고가 발생한다.

| 주의 | 아래의 코드에 사용된 ArrayList는 Arrays클래스의 내부 클래스이다. java.util.ArrayList와 혼동하지 말자.

```
public static <T> List<T> asList(T... a) {  
    return new ArrayList<T>(a); // ArrayList(E[] array)를 호출. 경고발생  
}
```

asList()의 매개변수가 가변인자인 동시에 지네릭 타입이다. 메서드에 선언된 타입 T는 컴파일 과정에서 Object로 바뀐다. 즉, Object[]가 되는 것이다. Object[]에는 모든 타입의 객체가 들어있을 수 있으므로, 이 배열로 ArrayList<T>를 생성하는 것은 위험하다고 경고하는 것이다. 그러나 asList()가 호출되는 부분을 컴파일러가 체크해서 타입 T가 아닌 다른 타입이 들어가지 못하게 할 것이므로 위의 코드는 아무런 문제가 없다.

이럴 때는 메서드 앞에 '@SafeVarargs'를 붙여서 '이 메서드의 가변인자는 타입 안정성이 있다.'고 컴파일러에게 알려서 경고가 발생하지 않도록 해야 한다.

메서드를 선언할 때 @SafeVarargs를 붙이면, 이 메서드를 호출하는 곳에서 발생하는 경고도 억제된다. 반면에 @SafeVarargs대신, @SuppressWarnings("unchecked")로 경고를 억제하려면, 메서드 선언뿐만 아니라 메서드가 호출되는 곳에도 애너테이션을 붙여야한다.

그리고 @SafeVarargs로 'unchecked'경고는 억제할 수 있지만, 'varargs'경고는 억제할 수 없기 때문에 습관적으로 @SafeVarargs와 @SuppressWarnings("varargs")를 같이 붙인다.

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

@SuppressWarnings("varargs")를 붙이지 않아도 경고 없이 컴파일 된다. 그러나 -Xlint옵션을 붙여서 컴파일 해보면, 'varargs'경고가 발생한 것을 확인할 수 있다. 그래서 가능하면 이 두 애너테이션을 항상 같이 사용하는 것이 좋다.

▼ 예제 12-12/ch12/AnnotationEx4.java

```
import java.util.Arrays;

class MyArrayList<T> {
    T[] arr;

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

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

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

class AnnotationEx4 {
    // @SuppressWarnings("unchecked")
    public static void main(String args[]) {
        MyArrayList<String> list = MyArrayList.asList("1", "2", "3");
        System.out.println(list);
    }
}
```

▼ 실행결과
[1, 2, 3]

▼ 컴파일 결과

```
C:\jdk1.8\work\ch12>javac AnnotationEx4.java
```

```
C:\jdk1.8\work\ch12>javac -Xlint AnnotationEx4.java
AnnotationEx4.java:15: warning: [varargs] Varargs method could cause heap
pollution from non-reifiable varargs parameter a
    return new MyArrayList<>(a);
                           ^
1 warning
C:\jdk1.8\work\ch12>
```

위의 컴파일 결과를 보면, 옵션없이 컴파일했을 때는 아무런 경고가 없지만, -Xlint옵션을 붙여서 컴파일했을 때는 'varargs' 경고가 발생한 것을 알 수 있다. 아래와 같이 애너테이션을 추가하면, -Xlint옵션으로 컴파일 하여도 경고가 나타나지 않을 것이다.

```
@SafeVarargs
@SuppressWarnings("varargs") // 애너테이션을 추가
// @SuppressWarnings("unchecked")
public static <T> MyArrayList<T> asList(T... a) {
    return new MyArrayList<>(a);
}
```

그리고 예제에서 `asList()`의 `@SafeVarargs`를 주석처리하고, `asList()`와 `main()`에 붙은 `@SuppressWarnings("unchecked")`를 주석해제해보자. `@SafeVarargs`가 이 두 개의 애너테이션과 같은 효과를 얻는다는 것을 확인할 수 있다.

```
//      @SafeVarargs      ← 주석처리
//      @SuppressWarnings("unchecked") ← 주석해제
public static <T> MyArrayList<T> asList(T... a) {
    return new MyArrayList<>(a);
}
...
class AnnotationEx3 {
    @SuppressWarnings("unchecked") ← 주석해제
    public static void main(String args[]) {
```

3.3 메타 애너테이션

앞서 설명한 것과 같이 메타 애너테이션은 ‘애너테이션을 위한 애너테이션’, 즉 애너테이션에 붙이는 애너테이션으로 애너테이션을 정의할 때 애너테이션의 적용대상(target)이나 유지기간(retention) 등을 지정하는데 사용된다.

| 참고 | 메타 애너테이션은 'java.lang.annotation' 패키지에 포함되어 있다.

@Target

애너테이션이 적용 가능한 대상을 지정하는데 사용된다. 아래는 '`@SuppressWarnings`'를 정의한 것인데, 이 애너테이션에 적용할 수 있는 대상을 '`@Target`'으로 지정하였다. 앞서 언급한 것과 같이 여러 개의 값을 지정할 때는 배열에서처럼 괄호 {}를 사용해야 한다.

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

'`@Target`'으로 지정할 수 있는 애너테이션 적용대상의 종류는 아래와 같다.

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)

▲ 표 12-3 애너테이션 적용대상의 종류

'TYPE'은 타입을 선언할 때, 애너테이션을 붙일 수 있다는 뜻이고 'TYPE_USE'는 해당 타입의 변수를 선언할 때 붙일 수 있다는 뜻이다. 표12-3의 값들은 'java.lang.annotation.ElementType'이라는 열거형에 정의되어 있으며, 아래와 같이 static import 문을 쓰면 'ElementType.TYPE'을 'TYPE'과 같이 간단히 할 수 있다.

```
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;
}
```

'FIELD'는 기본형에, 그리고 'TYPE_USE'는 참조형에 사용된다는 점에 주의하자.

@Retention

애너테이션이 유지(retention)되는 기간을 지정하는데 사용된다. 애너테이션의 유지 정책(retention policy)의 종류는 다음과 같다.

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

'@Override'나 '@SuppressWarnings'처럼 컴파일러가 사용하는 애너테이션은 유지 정책이 'SOURCE'이다. 컴파일러를 직접 작성할 것이 아니면, 이 유지정책은 필요없다.

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

유지 정책을 'RUNTIME'으로 하면, 실행 시에 '리플렉션(reflection)'을 통해 클래스 파일에 저장된 애너테이션의 정보를 읽어서 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 {}
```

유지 정책 'CLASS'는 컴파일러가 애너테이션의 정보를 클래스 파일에 저장할 수 있게는 하지만, 클래스 파일이 JVM에 로딩될 때는 애너테이션의 정보가 무시되어 실행 시에 애너테이션에 대한 정보를 얻을 수 없다. 이것이 'CLASS'가 유지정책의 기본값임에도 불구하고 잘 사용되지 않는 이유이다.

| 참고 | 지역 변수에 붙은 애너테이션은 컴파일러만 인식할 수 있으므로, 유지정책이 'RUNTIME'인 애너테이션을 지역변수에 붙여도 실행 시에는 인식되지 않는다.

@Documented

애너테이션에 대한 정보가 javadoc으로 작성한 문서에 포함되도록 한다. 자바에서 제공하는 기본 애너테이션 중에 '@Override'와 '@SuppressWarnings'를 제외하고는 모두 이 메타 애너테이션이 붙어 있다.

@Documented

```
@Retention(RetentionPolicy.RUNTIME)
```

```
@Target(ElementType.TYPE)
```

```
public @interface FunctionalInterface {}
```

@Inherited

애너테이션이 자손 클래스에 상속되도록 한다. '@Inherited'가 붙은 애너테이션을 조상 클래스에 붙이면, 자손 클래스도 이 애너테이션이 붙은 것과 같이 인식된다.

```
@Inherited // @SuperAnno가 자손까지 영향 미치게
```

```
@interface SuperAnno {}
```

```
@SuperAnno
```

```
class Parent {}
```

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

위의 코드에서 Child클래스는 애너테이션이 붙지 않았지만, 조상인 Parent클래스에 붙은 '@SuperAnno'가 상속되어 Child클래스에도 '@SuperAnno'가 붙은 것처럼 인식된다.

@Repeatable

보통은 하나의 대상에 한 종류의 애너테이션을 붙이는데, '@Repeatable'이 붙은 애너테이션은 여러 번 붙일 수 있다.

| 참고 | 팔호안의 'ToDos.class'는 반복된 애너테이션을 저장할 애너테이션을 지정한 것으로 곧 설명할 것이다.

```
@Repeatable(Todos.class) // ToDo애너테이션을 여러 번 반복해서 쓸 수 있게 한다.
```

```
@interface ToDo {
```

```
} String value();
```

예를 들어 '@ToDo'라는 애너테이션이 위와 같이 정의되어 있을 때, 다음과 같이 MyClass클래스에 '@ToDo'를 여러 번 붙이는 것이 가능하다.

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

일반적인 애너테이션과 달리 같은 이름의 애너테이션이 여러 개가 하나의 대상에 적용될 수 있기 때문에, 이 애너테이션들을 하나로 묶어서 다룰 수 있는 애너테이션도 추가로 정의해야 한다.

```
@interface Todos { // 여러 개의 ToDo애너테이션을 담을 컨테이너 애너테이션 Todos  
    ToDo[] value(); // ToDo애너테이션 배열타입의 요소를 선언. 이름이 반드시 value이어야 함  
}  
  
@Repeatable(Todos.class) // 괄호 안에 컨테이너 애너테이션을 지정해 줘야한다.  
@interface ToDo {  
    String value();  
}
```

@Native

네이티브 메서드(native method)에 의해 참조되는 '상수 필드(constant field)'에 붙이는 애너테이션이다. 아래는 java.lang.Long클래스에 정의된 상수이다.

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

네이티브 메서드는 JVM이 설치된 OS의 메서드를 말한다. 네이티브 메서드는 보통 C언어로 작성되어 있는데, 자바에서는 메서드의 선언부만 정의하고 구현은 하지 않는다. 그래서 추상 메서드처럼 선언부만 있고 몸통이 없다.

```
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 native void wait(long timeout) throws InterruptedException;  
    public native int hashCode();  
    ...  
}
```

이처럼 모든 클래스의 조상인 Object클래스의 메서드들은 대부분 네이티브 메서드이다. 네이티브 메서드는 자바로 정의되어 있기 때문에 호출하는 방법은 자바의 일반 메서드와 다르지 않지만 실제로 호출되는 것은 OS의 메서드이다. 그냥 아무런 내용도 없는 네이티브 메서드를 선언해 놓고 호출한다고 되는 것은 아니고,

자바에 정의된 네이티브 메서드와 OS의 메서드를 연결해주는 작업이 추가로 필요하다. 이 역할은 JNI(Java Native Interface)가 하는데, JNI는 이 책의 범위를 벗어나므로 자세한 설명은 생략한다.

3.4 애너테이션 타입 정의하기

지금까지 애너테이션을 사용하는 방법에 대해서 살펴봤는데, 이제 직접 애너테이션을 만들어서 사용해볼 것이다. 새로운 애너테이션을 정의하는 방법은 아래와 같다. '@' 기호를 붙이는 것을 제외하면 인터페이스를 정의하는 것과 동일하다.

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

엄밀히 말해서 '@Override'는 애너테이션이고 'Override'는 '애너테이션의 타입'이다.

애너테이션의 요소

애너테이션 내에 선언된 메서드를 '애너테이션의 요소(element)'라고 하며, 아래에 선언된 TestInfo 애너테이션은 다섯 개의 요소를 갖는다.

| 참고 | 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 yymmdd();  
    String hhmmss();  
}
```

애너테이션의 요소는 반환값이 있고 매개변수는 없는 추상 메서드의 형태를 가지며, 상속을 통해 구현하지 않아도 된다. 다만, 애너테이션을 적용할 때 이 요소들의 값을 빠짐없이 지정해주어야 한다. 요소의 이름도 같이 적어주므로 순서는 상관없다.

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

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

| 참고 | 기본값으로 null을 제외한 모든 리터럴이 가능하다.

```
@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{} 생략 가능  
@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 때도 요소의 이름이 value이면, 요소의 이름을 생략할 수 있다. 예를 들어, '@SuppressWarnings'의 경우, 요소의 타입이 String 배열이고 이름이 value이다.

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

그래서 애너테이션을 적용할 때 요소의 이름을 생략할 수 있는 것이다.

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

java.lang.annotation.Annotation

모든 애너테이션의 조상은 Annotation이다. 그러나 애너테이션은 상속이 허용되지 않으므로 아래와 같이 명시적으로 Annotation을 조상으로 지정할 수 없다.

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

게다가 아래의 소스에서 볼 수 있듯이 Annotation은 애너테이션이 아니라 일반적인 인터페이스로 정의되어 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
```

모든 애너테이션의 조상인 Annotation 인터페이스가 위와 같이 정의되어 있기 때문에, 모든 애너테이션 객체에 대해 equals(), hashCode(), toString()과 같은 메서드를 호출하는 것이 가능하다.

```
Class<AnnotationTest> cls = AnnotationTest.class;  
Annotation[] annoArr = AnnotationTest.class.getAnnotations();  
for(Annotation a : annoArr) {  
    System.out.println("toString():" + a.toString());  
    System.out.println("hashCode():" + a.hashCode());  
    System.out.println("equals():" + a.equals(a));  
    System.out.println("annotationType():" + a.annotationType());  
}
```

위의 코드는 AnnotationTest 클래스에 적용된 모든 애너테이션에 대해 toString(), hashCode(), equals()를 호출한다.

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

애너테이션의 요소를 선언할 때 반드시 지켜야 하는 규칙은 다음과 같다.

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

아래의 코드에서 오른쪽 주석을 가지고 무엇이 잘못되었는지 잠시 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
}
```

▼ 예제 12-13/ch12/AnnotationEx5.java

```
import java.lang.annotation.*;
@Deprecated
@SuppressWarnings("1111") // 유효하지 않은 애너테이션은 무시된다.
@TestInfo(testedBy="aaa", testDate=@DateTime(ymmd="160101", hhmmss="235959"))
class AnnotationEx5 {
    public static void main(String args[]) {
        // AnnotationEx5의 Class 객체를 얻는다.
        Class<AnnotationEx5> cls = AnnotationEx5.class;
        TestInfo anno = (TestInfo)cls.getAnnotation(TestInfo.class);
        System.out.println("anno.testedBy()=" + anno.testedBy());
        System.out.println("anno.testDate().ymmd()=" + anno.testDate().ymmd());
        System.out.println("anno.testDate().hhmmss()=" + anno.testDate().hhmmss());
```

```

        for(String str : anno.testTools())
            System.out.println("testTools="+str);
        System.out.println();
        // AnnotationEx5에 적용된 모든 애너테이션을 가져온다.
        Annotation[] annoArr = cls.getAnnotations();
        for(Annotation a : annoArr)
            System.out.println(a);
    } // main의 끝
}

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

enum TestType { FIRST, FINAL }

```

▼ 실행 결과

```

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

@java.lang.Deprecated()
@TestInfo(count=1, testType=FIRST, testTools=[JUnit], testedBy=aaa,
testDate=@DateTime(yymmdd=160101, hhmmss=235959))

```

애너테이션을 직접 정의하고, 애너테이션의 요소의 값을 출력하는 방법을 보여주는 예제이다. AnnotationEx5 클래스에 적용된 애너테이션을 실행시간에 얻으려면, 아래와 같이 하면 된다.

```

Class<AnnotationEx5> cls = AnnotationEx5.class;
TestInfo anno = (TestInfo)cls.getAnnotation(TestInfo.class);

```

'AnnotationEx5.class'는 클래스 객체를 의미하는 리터럴이다. 앞서 9장에서 배운 것과 같이, 모든 클래스 파일은 클래스로더(Classloader)에 의해 메모리에 올라갈 때, 클래스에 대한 정보가 담긴 객체를 생성하는데 이 객체를 클래스 객체라고 한다. 이 객체를 참조 할 때는 '클래스이름.class'의 형식을 사용한다.

클래스 객체에는 해당 클래스에 대한 모든 정보를 가지고 있는데, 애너테이션의 정보도 포함되어 있다.

클래스 객체가 가지고 있는 getAnnotation()이라는 메서드에 매개변수로 정보를 얻고자하는 애너테이션을 지정해주거나 getAnnotations()로 모든 애너테이션을 배열로 받아올 수 있다.

```
TestInfo anno = (TestInfo)cls.getAnnotation(TestInfo.class);  
System.out.println("anno.testedBy() = "+anno.testedBy());  
  
// AnnotationEx5에 적용된 모든 애너테이션을 가져온다.  
Annotation[] annoArr = cls.getAnnotations();
```

| 참고 | Class 클래스를 Java API에서 찾아보면 클래스의 정보를 제공하는 다양한 메서드가 정의되어 있는 것을 확인할 수 있다.

| 참고 | 연습문제는 코드초보스터디(<http://cafe.naver.com/joonggakpostudy.cafe>)에서 PDF파일로 제공

클래스 객체에는 해당 클래스에 대한 모든 정보를 가지고 있는데, 애너테이션의 정보도 포함되어 있다.

클래스 객체가 가지고 있는 `getAnnotation()`이라는 메서드에 매개변수로 정보를 얻고자하는 애너테이션을 지정해주거나 `getAnnotations()`로 모든 애너테이션을 배열로 받아올 수 있다.

```
TestInfo anno = (TestInfo)cls.getAnnotation(TestInfo.class);
System.out.println("anno.testedBy() = "+anno.testedBy());
// AnnotationEx5에 적용된 모든 애너테이션을 가져온다.
Annotation[] annoArr = cls.getAnnotations();
```

| 참고 | Class 클래스를 Java API에서 찾아보면 클래스의 정보를 제공하는 다양한 메서드가 정의되어 있는 것을 확인할 수 있다.

| 참고 | 연습문제는 코드초보스터디 (<http://cafe.naver.com/javachobostudy.cafe>)에서 PDF 파일로 제공