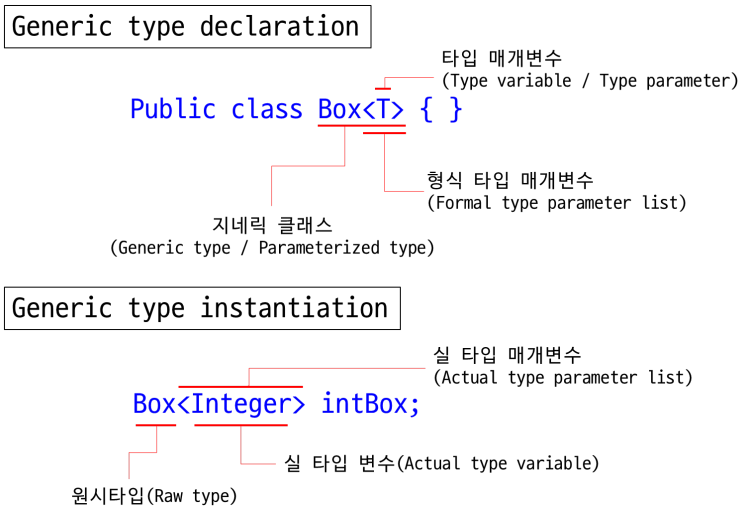
Chapter 12 지네릭스, 열거형, 애너테이션

◈ 지네릭스
- 다양한 타입의 객체들을 다루는 메서드나 컬렉션 클래스에 컴파일 시의 타입 체크(compile-time type check)를 해주는 기능
- run-time에서 발생하는 각종 exception을 compile-time으로 가져온다.
- 쉽게 말해, parameter등의 type을 미리 정해주고 검사하여 각종 오류가 없게 만드는 목적으로 사용한다.

◈ 지네릭스의 장점
1. 의도하지 않은 타입의 객체가 저장되는 것을 막고, 저장된 객체를 꺼내올 때 원래의 타입과 다른 타입으로 잘못 형변환 되어 발생할 수 있는 오류 방지
2. 별도의 타입 체크가 불필요하며, 형변환을 생략하여 코드의 간결화

◈ 지네릭스 용어



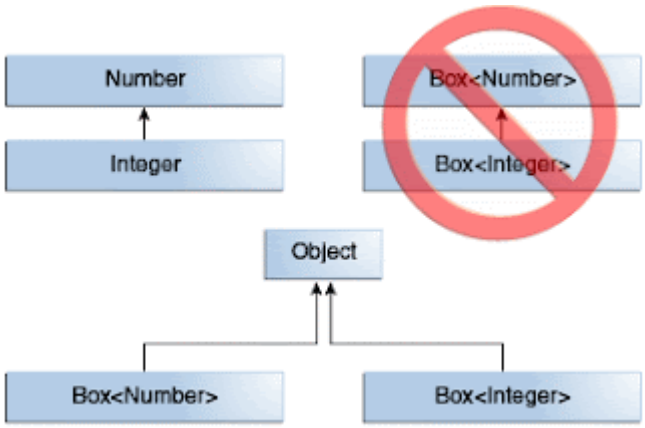
- Generic = Parameterized
- T는 generic class Box<T>의 type variable 혹은 type parameter라고 한다.
- T를 <>로 감싼 형태인 <T>를 Formal type parameter list라고 한다.
- type variable로 표현되는 T 대신 다른 문자를 사용해도 된다. 예를 들어, E는 element, K는 key, V는 value를 뜻한다.
- compile 과정을 거치면 BOX<T>, BOX<Integer> 등은 BOX로 바뀐다.

Term	*Example*
Parameterized type	List<String>
Actual type parameter	String
Generic type	List<E>
Formal type parameter	E
Unbounded wildcard type	List<?>
Raw type	List
Bounded type parameter	<E extends Number>
Recursive type bound	<T extends Comparable<T>>
Bounded wildcard type	List<? extends Number>
Generic method	static <E> List<E> asList(E[] a)
Type token	String.class

◈ 지네릭스의 제한
- 객체의 parameter 별로 다른 타입을 선언여 사용하는 것이 가능하다.
  . Box<Apple> appleBox = new Box<Apple>(); // OK. Apple 객체만 저장 가능
  . Box<Grape> appleBox = new Box<Grape>(); // OK. Grape 객체만 저장 가능
- static 멤버는 지네릭스를 사용할 수 없다. static 멤버에 붙이는 type variable이 instance variable로 간주되기 때문에 static이 선언된 상황에서는 이 instance variable이 존재하지 않기 때문. 그리고 static 멤버는 타입 변수에 지정된 타입, 즉 대입된 타입의 종류에 관계없이 동일한 것이어야 하기 때문
- 하지만, 지네릭 메서드에서는 static에서도 사용 가능
- 지네릭 타입의 배열을 생성하는 것도 불가능

◈ **Unbounded wildcard type**
- Unbounded는 무제한, wildcard는 어떤 타입이든 들어올 수 있는 형태
- 즉, ? 뒤에 조건을 붙여서 해당 조건을 만족한 타입을 받아들일 수 있는 형태

<? 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



◈ **Bounded wildcard type**
- parameter를 특정 타입으로 제한하여 받을 수 있는 형태 (extends, super 사용)

◈ 지네릭 메서드
- 선언 위치는 return type 바로 앞으로 지정되어 있다.
  . static <T> void sort(List<T> list, Comparator<? super T> c)
- Generaic class에 정의된 type parameter와 Generic method에 정의된 type parameter는 전혀 별개의 것이므로 구분해야 한다.
- 메서드에서는 지네릭 타입을 선언하고 사용하는 것이 가능. 메서드에 선언된 지네릭 타입은 지역 변수를 선언한 것과 같다고 생각하면 이해하기 쉬우며, 이 타입 매개변수는 메서드 내에서만 지역적으로 사용될 것이므로 메서드가 static이건 아니건 상관이 없다.
- 결론적으로, 지네릭 타입은 static class, static variable에서는 사용 불가하나, static method에서는 사용 가능하다.
- 지네릭 메서드를 호출할 때에는 참조 변수나 클래스 이름을 생략할 수 없다.
  . FruitBox<Furit> furitBox = new FruitBox<Fruit>();
  . System.out.println(<Fruit>makeJuice(fruitBox)); // 에러, class명 생략 불가
  . System.out.println(this.<Fruit>makeJuice(fruitBox)); // OK
  . System.out.println(Juicer.<Fruit>makeJuice(fruitBox)); // OK

◈ 지네릭 기타
- 가급적이면 raw type 사용은 지양하고, generic type으로 사용하는 것을 습관화 해야 한다.
- 지네릭 타입은 compile 과정을 통해 제거가 된다.

◈ 열거형 **(enums)**
- 서로 관련된 상수를 묶어서 편리하게 선언하기 위한 클래스
- type-safe enum이므로, 값 뿐만 아니라 type도 체크하여 실제의 값이 같아도 type이 다르면 compile 에러 발생
- 열거형 상수간의 비교에는 ==은 사용 가능하나 비교연산자(<, >)는 불가하다. 단, compareTo()는 가능하다.

◈ 열거형에 멤버 추가하기
- Enum클래스에 정의된 ordinal()이 열거형 상수가 정의된 순서를 반환하지만, 이를 상수값으로 매칭하여 사용하는 것은 안좋다. 이는 내부적인 용도로 사용되기 때문이다.
- 열거형 상수의 값이 불연속적일 경우에는 열거형 상수의 이름 옆에 원하는 값을 괄호와 함께 적어주면 된다.
- 주의할 점은 먼저 열거형 상수를 모두 정의한 다음에 다른 멤버들을 추가해야 하며, 열거형 상수의 마지막에 ;를 붙여야 한다.
- 열거형의 생성자는 묵시적으로 private이기 때문에 외부에서 호출이 불가하다.

```
enum Direction {
    EAST(1), SOUTH(5), WEST(-1), NORTH(10);  // 끝에 ;를 추가해야 한다.
    private final int value;    // 정수를 저장할 필드(instance 변수)를 추가
    Direction(int value) { }  // 생성자 추가
    public int getValue() { }
}
```

- 필요에 의해 아래와 같이 하나의 열거형 상수에 여러 값을 지정할 수도 있다.

```
enum direction {
    EAST(1, ">"), SOUTH(2, "V"), WEST(3, "<"), NORTH(4, "^");
    private final int value;    // 정수를 저장할 필드(instance 변수)를 추가
    private final String symbol;
    Direction(int value, String symbol) { // access modifier private이 생략됨
        this.value = value; this.symbol = symbol }
    public int getValue() { }
}
```

◈ 애너테이션**(annotation)**
- 주석(comment)처럼 프로그래밍 언어에 영향을 미치지 않으면서도 다른 프로그램에게 유용한 정보를 제공하는데 사용

@Override	compiler에게 override하는 메서드라는 것을 알림
@Deprecated	앞으로 사용하지 않을 것을 권장하는 대상에 붙임
@SuppressWarnings	compiler의 특정 경고 메시지가 나타나지 않게함
@SafeVarargs	- 지네릭스 타입의 가변인자에 사용 - static, final이 붙은 메서드만 가능(overriding 불가) - parameter가 non-reifiable 타입일 경우 unchecked 경고가 발생하며, 이를 억제하기 위해 사용

- reifiable[re-a-ify-able]: 컴파일 후에도 타입 정보가 유지되는 타입
- non-reifiable: 컴파일 후에 타입 정보가 제거되는 타입 (대부분)

◈ 메타 애너테이션
- 애너테이션을 위한 애너테이션으로 주로 애너테이션을 정의할 때 사용

◈ 애너테이션 타입 정의하기

```
@interface 애너테이션 이름 {
    타입 요소이름();   // 애너테이션의 요소를 선언한다.
    …
}
```

- 타입 요소이름 뒤에 붙은 ()는 메서드를 의미하지 않고, variable을 의미한다.

```
@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
public class NewClass { … }
```

```
// 요소의 타입이 배열인 경우, 괄호 {} 를 사용해서 여러 개의 값 지정 가능

@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
```

◈ 애너테이션 요소의 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
    String major(int i, int j);            // 에러. 매개변수 선언 불가
    String minor() throws Exception;       // 에러. 예외 선언 불가
    ArrayList<T> list();                   // 에러. 요소의 타입에 타입 매개변수 불가
}
```