Java의 정석

Chapter 12

지네릭스(Generics)

❑ 지네릭스

- 다양한 타입의 객체들을 다루는 메서드나 컬렉션 클래스에 컴파일 시의 타입 체크를 해주는 기능
- 컴파일 시 타입이 결정되어 보다 안전한 프로그래밍이 가능하다
- 타입 체크와 형 변환을 생략할 수 있으므로 코드가 간결해진다.

```
class Box{
    String item;

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

지네릭스(Generics)

```
class Box{
    String item;

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

위 코드에서 box1이라는 Box 객체를 만든다고 하자

- box1에 item과 setItem()의 파라미터로는 string 타입을 줘야한다.
- 이때 2가지 번거로운 점이 있는데
- 1) 해당 값들은 String만 줄 수 있다는 사실은 인지하고 String 값만 전달해야 한다
- 2) 만약 int값을 넣고 싶다면 클래스를 하나 더 오버로딩해야 한다.

- 하지만 만약 String 대신 자유로운 값을 넣을 수 있는 “T”라는 임의의 값을 설정한다면 T에 들어갈 것을 자유자재로 바꾸어 작업이 가능하다.
- Type을 의미하는 “T” 대신 E(=Element), Map<K,V>(타입 여러개일 때) 등 아무 기호나 사용 가능하다.

지네릭스(Generics)

❑ Box<T>

- 지네릭 클래스, ‘T의 박스’ 또는 ‘T 박스 ' 라고 읽는다

❑ T

- 타입 변수 또는 타입 매개뱐수

❑ Box

- 원시 타입

❑ 지네릭스의 재한

- Static멤버에 타입 변수 T를 사용할 수 없다.
- T는 인스턴스 변수로 간주되기 때문인데, static멤버는 인스턴스 변수를 참조할 수 없다.
- 지네릭 타입 배열도 생성이 불가능하다(new 연산자는 컴파일 시점에 T가 뭔지 알아야하기 때문)

지네릭스(Generics)

Public String toString()

- 직접 호출하지 않아도 자동으로 어떤 객체를 System.out.print()로 호출 시 자동으로 toString()이 호출된다.

제한된 지네릭 클래스

- 박스에 과일류만 넣고 싶다면 Box<T> 부분을 Box<T extends Fruit>로 바꿔 쓰면 된다

```
3  class Fruit               { public String toString() { return "Fruit";}}
4  class Apple extends Fruit { public String toString() { return "Apple";}}
5  class Grape extends Fruit { public String toString() { return "Grape";}}
6  class Toy                 { public String toString() { return "Toy"  ;}}
7
8  class FruitBoxEx1 {
9      public static void main(String[] args) {
10         Box<Fruit> fruitBox = new Box<Fruit>();
11         Box<Apple> appleBox = new Box<Apple>();
12         Box<Toy>   toyBox   = new Box<Toy>();
13 //      Box<Grape> grapeBox = new Box<Apple>(); // ????. Ÿ?? ????ġ
14
15         fruitBox.add(new Fruit());
16         fruitBox.add(new Apple()); // OK. void add(Fruit item)
17
18         appleBox.add(new Apple());
19         appleBox.add(new Apple());
20 //      appleBox.add(new Toy()); // ????. Box<Apple>???? Apple?? ???? ?? ????
21
22         toyBox.add(new Toy());
23 //      toyBox.add(new Apple()); // ????. Box<Toy>???? Apple?? ???? ?? ????
24
25         System.out.println(fruitBox);
26         System.out.println(appleBox);
27         System.out.println(toyBox);
28     }  // main?? ??
29 }
30
31 class Box<T> {
32     ArrayList<T> list = new ArrayList<T>();
33     void add(T item)  { list.add(item); }
34     T get(int i)      { return list.get(i); }
35     int size() { return list.size(); }
36     public String toString() { return list.toString();}
37 }
38
```

와일드카드

❑ 와일드 카드

- 아래 코드의 경우 Fruit 대신 T를 쓸 수 없다(static이기 때문)
- Fruit 대신 Apple을 넣고 싶으면 오버로딩 해야함.
- 이러한 비효율을 해결하기 위한 것이 와일드 카드

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

와일드카드

❑ **<? Extends T>**

- 와일드 카드의 상한 제한. T와 그 자손들만 가능

❑ **<? Super T>**

- 와일드 카드의 하한 제한. T와 그 조상들만 가능

❑ **<?> = <? Extends Object>**

- 제한 없음. 모든 타입 가능

지네릭 메소드

❑ 지네릭 메소드

- 반환타입(void 등) 바로 앞에 선언한다
- 클래스 타입 매개변수와는 다른 것이다.
- Static멤버에는 타입 매개 변수를 사용할 수 없지만 메소드에 지네릭 타입을 선언하고 사용하는 것은 가능하다
- 코드를 간결하게 해줄 수 있다.

```
class FruitBox<T>{

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

}
```

```
static Juice makeJuice(Fruitbox<? extends Fruit> box) {
    // codes //
}

static <T extends Fruit> Juice makeJuice(Fruitbox<T> box) {
    // codes //
}
```

지네릭 타입의 형변환

❑ **지네릭 타입 형변환 가능한 경우**

- 지네릭 타입과 지네릭 타입이 아닌 것 가이 형변환 가능
- 와일드 카드와 형변환
- Ex1) Fruitbox<? Extends Fruit> box = new Fruitbox<Fruit>
- Ex2) Fruitbox<? Extends Fruit> box = new Fruitbox<Apple>
- Ex3) Fruitbox<? Extends Fruit> box = new Fruitbox<Grape>

- **단, 지네릭 타입 간에는 형변환이 안된다.**

열거형(enums)

❑ 열거형(enums)

- 서로 관련된 상수를 편리하게 선언하기 위한 것
- 순서에 따라 열거되거나 여러 개가 하나의 묶음으로 의미가 있을 때 주로 사용한다
- 대문자로 표기하며, 값을 부여하는 경우가 아니라면 마지막에 “;”를 생략한다

Ex1) “월요일~일요일”은 순서가 있으며, 요일로 묶인다

Ex2) “동,서,남,북”은 순서는 없을지라도, 방향으로 묶을 수 있다.

쉽게 정리하면 (아래 내용은 필수는 아니고 주로 이러한 특징을 갖는다 정도로 이해할 것!)

1. 묶음으로 특성을 가질 때
2. 순서의 효과를 갖고 싶을 때
3. 상수별로 특정한 조건을 주고 싶을 때

주로 사용한다고 보자

열거형(enums)

```
3  enum Day { MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY }
4
5  public class ex_0628 {
6
7      public static void main(String[] args) {
8
9          System.out.println(Day.MONDAY);
10
11     }
12 }
13
```

- 결과값: MONDAY

열거형(enums)

❑ 주요 메소드

메소드	설명
String name()	- 열거형 상수의 이름을 문자열로 반환
Int ordinal()	- 열거형 상수가 정의된 순서 반환 (0~)
T values()	- 열거형 상수의 목록을 리스트로 반환 - For문 등을 통해 읽어온다
T valueOf(Class<T> enumType, String name)	- 지정된 열거형에서 name과 일치하는 열거형 상수를 반환

열거형(enums)

❑ 메소드 실습

```
3  enum Day { MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY }
4
5  public class ex_0628 {
6
7      public static void main(String[] args) {
8
9          System.out.println(Day.MONDAY);
10         System.out.println();
11
12         for(Day d: Day.values()) {
13             System.out.print(d);
14             System.out.println(d.ordinal());
15         }
16
17         System.out.println();
18         System.out.println(Day.MONDAY.name());
19
20
21     }
22 }
23
24
25
```

Problems @ Javadoc Declaration Console

```
<terminated> ex_0628 [Java Application] /Library/Java/JavaVirtualMachines/jdk-11.0.11.jc
MONDAY

MONDAY0
TUESDAY1
WEDNESDAY2
THURSDAY3
FRIDAY4

MONDAY
```

열거형(enums)

❑ 값을 넣어주기

```
enum Day { MONDAY("월"), TUESDAY("화"), WEDNESDAY("수"), THURSDAY("목"), FRIDAY("금");

    final String day;

    Day(String day){ this.day = day; }

    String getDay() {return day;}

    }

public class ex_0628 {

    public static void main(String[] args) {

        for (Day d: Day.values()) {
            System.out.println(d.getDay());
        }

    }
}
```

Problems @ Javadoc Declaration Console

```
<terminated> ex_0628 [Java Application] /Library/Java/JavaVirtualMachines/jdk-11.0.11.jdk/Contents/Home/bin/java  (2021. 8. 10.
월
화
수
목
금
```

애너테이션(annotation)

❑ 애너테이션(annotation)

- 한국어로 “언급”이라고 이해하면 된다.
- 해당 클래스/메소드가 어떤 역할을 하는지, 어떤 특성이 있는지를 링크를 통해 각주(주석) 형식으로 알려준다고 이해하면 쉽다.
- 안써도 큰 문제는 없지만, 경우에 따라 실수를 인지할 수 있다는 장점이 있다.

EX) **@Override**

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


class Child extends Parent{

//  @Override
    void parentmethod() {}
}
```

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


class Child extends Parent{

    @Override
    void parentmethod() {}
}
```

메소드를 오버라이딩 할 계획이었다고 하자.

좌측의 경우 m을 실수로 소문자로 썼다.
하지만, 새로운 메소드로 인지할 뿐 오류로 보지 않는다.

우측의 경우 오버라이딩을 한 것이라고 인지시킴에 따라 빨간줄이 뜬다