Item 31

한정적 와일드카드를 사용해 API 유연성을 높이라

목차

목차

불공변

서로 다른 타입 Type1, Type2 존재

List<Type1>, List<Type2>는 아무 관계 X

불공변

상위 타입 Type1, 하위 타입 Type2 존재

List<Type1>, List<Type2>는 아무 관계 X

불공변

제네릭 타입은 불공변 !

불공변의 장점

- 타입 안전성
- 예측 가능한 동작
- ...

불공변

제네릭 타입이 **불공변이라 불편한점** 존재!

목차

불공변 불편함 1

```
public class Stack<E>{
   ...
   public void pushAll(Iterable<E> src){
     for(E e : src)
          push(e);
   }

}
```

불공변 불편함 1

2

```
public class Stack<Number>{
   ...
   public void pushAll(Iterable<Number> src){
     for(E e : src)
         push(e);
   }

}
```

1

```
Stack<Number> stack = new Stack<>();

List<Integer> list = new ArrayList<>();

stack.pushAll(list);
```

3

4

Iterable<Number>에 Iterable<Integer> 삽입

불공변 불편함 1

Iterable<Number>에 Iterable<Integer> 삽입

불공변 불편함 1 - 해결책

한정적 와일드카드 타입

- 타입 매개변수 범위 제한하는 방법
- 특정 타입 기준으로 상한, 하한 범위 지정
- extends, super 키워드로 타입의 범위 지정

불공변 불편함 1 - 해결책

Iterable<Number>에 Iterable<Integer> 넣기

Iterable<E> -> Iterable<E의 하위 타입>

Iterable<Number> -> Iterable<Number의 하위 타입>

Iterable<E> -> Iterable<? extends E>

불공변 불편함 1 - 해결책

```
public static void main(String[] args) {
    Stack<Number> stack = new Stack<>();

    Iterable<Integer> test1 = new ArrayList<>();
    Iterable<Number> test2 = new ArrayList<>();
    Iterable<Object> test3 = new ArrayList<>();

    stack.pushAll(test1);
    stack.pushAll(test2);
    stack.pushAll(test3);     < < 컴파일 에러 발생
}
```

메서드 파라미터로

E, E의 하위 타입 가능,

E의 상위 타입 불가능

Integer<Number<Object

상한 경계

불공변 불편함 1 - 해결책

```
public void pushAll(Iterable<? extends Number> src){

    for (Integer number : src) {     <<  컴파일 에러 발생
    }

    for (Number number : src) {
    }

    for (Object number : src) {
    }
}
```

꺼내서 사용할 때

E, E의 상위 타입 가능,

E의 하위 타입 불가능

Integer<Number<Object

상한 경계

불공변 불편함 1 - 해결책

```java
public void test(ArrayList<? extends Number> dst){
        Integer integer = 3;
        Number number = 3;
        Object object = 3;

        dst.add(integer);       << 컴파일 에러 발생
        dst.add(number);        << 컴파일 에러 발생
        dst.add(object);        << 컴파일 에러 발생
}
```

넣으려고 할 때

E, E의 상위, 하위 타입 불가능

E나 E의 하위 타입 중 어떤 것?

-> 넣기 불가능

목차

불공변 불편함 2

```
public class Stack<E>{
    ...
    public void popAll(Collection<E> dst){
      while(!isEmpty()){
          dst.add(pop());
      }
    }
}
```

불공변 불편함 2

```
public class Stack<Number>{
    ...
    public void popAll(Collection<Number> dst){
      while(!isEmpty()){
          dst.add(pop());
      }
    }
}
```

```
Stack<Number> stack = new Stack<>();

List<Object> list = ArrayList<>();

stack.popAll(list);
```

Collection<Object> 받아서 여기에 Number 넣기

불공변 불편함 2


public class Stack<Number>{
...
public void popAll(Collection<
while(!isEmpty()){
dst.add(pop());
}
}
}
Number> stack = new Stack<>();
List<Object> list = ArrayList<>();
stack.popAll(list);
Collection<Object> 받아서 여기에 Number 넣기

불공변 불편함 2 - 해결책

Collection<Number>에 Collection<Object> 넣기

Collection<E> -> Collection<E의 상위 타입>

Collection<Number> -> Collection<Number의 상위 타입>

Collection<E> -> Collection<? super E>

불공변 불편함 2 - 해결책

```
Stack<Number> stack = new Stack<>();

Collection<Integer> test11 = new ArrayList<>();
Collection<Number> test22 = new ArrayList<>();
Collection<Object> test33 = new ArrayList<>();

stack.popAll(test11);          <<  컴파일 에러 발생
stack.popAll(test22);
stack.popAll(test33)
```

메서드 파라미터로

E, E의 상위 타입 가능,

E의 하위 타입 불가능

Integer<Number<Object

하한 경계

불공변 불편함 2 - 해결책

```
public void test(Collection<? super Number> dst){
    for (Object o : dst) {
    }

    for (Number o : dst) {      <<  컴파일 에러 발생
    }

    for (Integer o : dst) {     <<  컴파일 에러 발생
    }
}
```

꺼내서 사용할 때

최상위 타입 Object 가능,

E, E의 하위 타입 불가능

Integer<Number<Object

하한 경계

불공변 불편함 2 - 해결책

```java
public void test(Collection< ? super Number> dst){
    Integer integer = 3;
    Number number = 3;
    Object object = 3;

    dst.add(integer);
    dst.add(number);
    dst.add(object);      <<  컴파일 에러 발생
}
```

넣으려고 할 때

E, E의 하위 타입 가능

E의 상위 타입 불가능

Integer<Number<Object

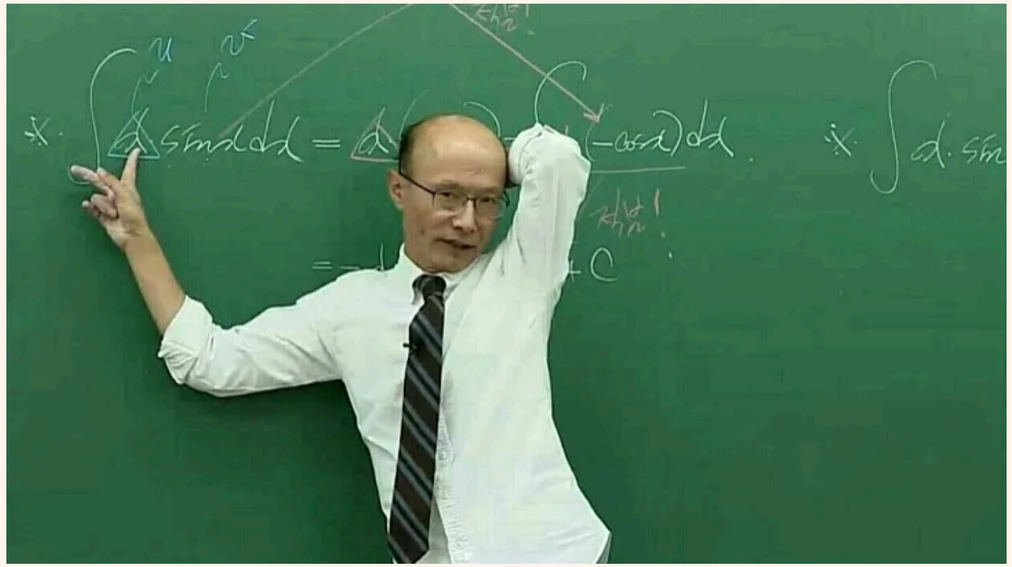
하한 경계

목차

와일드카드 타입 사용 공식

한정적 와일드카드 타입 언제 사용?

한정적 와일드카드 타입 사용 공식 존재!



와일드카드 타입 사용 공식

PECS : producer-extends, consumer-super

컬렉션, 제네릭 타입 T가 생산자 : < ? extends T>

컬렉션, 제네릭 타입 T가 소비자 : < ? super T>

와일드카드 타입 사용 공식

```
public class Stack<E>{
    ...
    public void pushAll(Iterable<? extends E> src) {
        for (E e : src)
            push(e);
    }
}
```

src는 Stack이 사용할

E 인스턴스를 생산 = 꺼내주니

생산자 (producer)

→ extends 사용

와일드카드 타입 사용 공식

```
public class Stack<E>{
    ...
    public void popAll(Collection<? super E> dst) {
        while (!isEmpty())
            dst.add(pop());
    }
}
```

dst는 Stack으로부터

E 인스턴스를 소비 = 받으니

소비자 (consumer)

→ super 사용

Tip

- 반환 타입에는 한정적 와일드카드 타입 사용 X
- Comparable, Comparator는 언제나 소비자
  - Comparable<E> 보단 Comparable<? super E> 사용

결론!

한정적 와일드카드 타입을 적용하면 **API 유연**해진다.

한정적 와일드카드 타입 사용 시 **PECS 공식** 기억하자.