



Item 30.

이왕이면 제네릭 메서드로 만들라





INDEX

목차

1. 제네릭 타입 메서드
2. 와일드 카드 활용
3. 제네릭 싱글턴 팩토리 패턴
4. 재귀적 타입 한정





01

제네릭 타입 메서드



제네릭 타입 메서드 타입의 매개변수를 가질 수 있는 메서드

장점

- 경고 없는 코드
- 타입의 안정성
- 유연한 설계
- 클라이언트의 코드 재사용



```
1 Set<String> ex1 = Set.of("새로이", "비타", "라젤");
2 Set<String> ex2 = Set.of("월슨", "밍트");
3 Set<Integer> ex3 = Set.of(1, 2, 3);
4 Set<List> ex4 = Set.of(List.of(1.0, 3), List.of("밍트", "월
5 슨"));
6
7 System.out.println("결과 : " + union(ex1, ex2));
8 System.out.println("결과 : " + union(ex1, ex3));
9 System.out.println("결과 : " + union(ex1, ex4));
```



```
1 private static Set union(Set s1, Set s2)
2 {   Set result = new HashSet(s1);
3     result.addAll(s2);
4     return result;
5 }
```

```
public static void main(String[] args) {  
    Set<String> ex1 = Set.of("새로이", "비타", "라젤");  
    Set<String> ex2 = Set.of("월슨", "밍트");  
    Set<Integer> ex3 = Set.of(1, 2, 3);  
    Set<List> ex4 = Set.of(List.of(1.0, 3), List.of("밍트", "월슨"));  
  
    System.out.println("결과 : " + union(ex1, ex2));  
    System.out.println("결과 : " + union(ex1, ex3));  
    System.out.println("결과 : " + union(ex1, ex4));  
}
```

```
private static Set union(Set s1, Set s2) {  
    Set result = new HashSet(s1);  
    result.addAll(s2);  
    return result;  
}
```

결과 : [월슨, 새로이, 비타, 밍트, 라젤]

결과 : [1, 2, 3, 새로이, 비타, 라젤]

결과 : [[1.0, 3], [밍트, 월슨], 새로이, 비타, 라젤]

✗ 경고 없는 코드

✗ 타입의 안정성

- 유연한 설계
- 클라이언트의 코드 재사용



```
1 Set<String> ex1 = Set.of("새로이", "비타", "라  
2 Set<String> ex2 = Set.of("월슨", "밍트");  
3  
4 Set<Integer> ex3 = Set.of(1, 2, 3);  
5 Set<Integer> ex4 = Set.of(1, 2, 3, 4, 5);  
6  
7 union(ex1, ex2);  
8 union(ex3, ex4);
```



```
1 private static Set<String> union(Set<String> s1, Set<String> s2) {  
2     Set<String> result = new HashSet<>(s1);  
3     result.addAll(s2);  
4     return result;  
5 }
```



```
public static void main(String[] args) {  
    Set<String> ex1 = Set.of("새로이", "비타", "라젤");  
    Set<String> ex2 = Set.of("월슨", "밍트");  
  
    Set<Integer> ex3 = Set.of(1, 2, 3);  
    Set<Integer> ex4 = Set.of(1, 2, 3, 4, 5);  
  
    Set<String> result = union(ex1, ex2);  
    union(ex3, ex4); // 컴파일 에러 발생  
}  
  
private static Set<String> union(Set<String> s1, Set<String> s2) {  
    Set<String> result = new HashSet<>(s1);  
    result.addAll(s2);  
    return result;  
}
```

- 경고 없는 코드
- 타입의 안정성
- ✗ 유연한 설계
- ✗ 클라이언트의 코드 재사용



**적절한 트레이드오프가
최선인가..?**



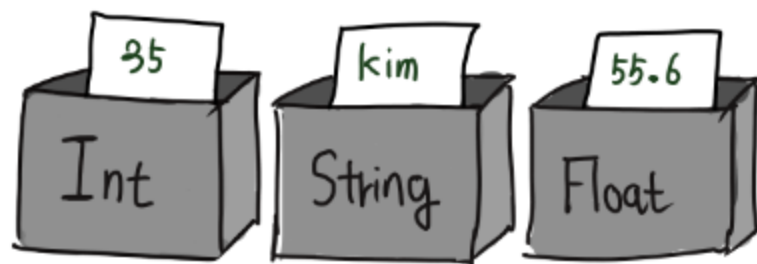
```
1 Set<String> ex1 = Set.of("새로이", "비타", "라  
2 Set<String> ex2 = Set.of("월슨", "밍트");  
3  
4 Set<Integer> ex3 = Set.of(1, 2, 3);  
5 Set<Integer> ex4 = Set.of(1, 2, 3, 4, 5);  
6  
7 union(ex1, ex2);  
8 union(ex3, ex4);
```



```
1 private static <E> Set<E> union(Set<E> s1, Set<E> s2)  
2 { Set<E> result = new HashSet<>(s1);  
3   result.addAll(s2);  
4   return result;  
5 }
```

- ⦿ 경고 없는 코드
- ⦿ 타입의 안정성
- ⦿ 유연한 설계
- ⦿ 클라이언트의 코드 재사용

제네릭 타입 메서드



타입 매개변수 목록은 **제한자**와 **반환 타입** 사이에 선언
타입 매개변수 **명명 규칙**은 클래스와 동일



```
1 private static <E> Set<E> union(Set<E> s1, Set<E> s2)
2 {   Set<E> result = new HashSet<>(s1);
3     result.addAll(s2);
4     return result;
5 }
```

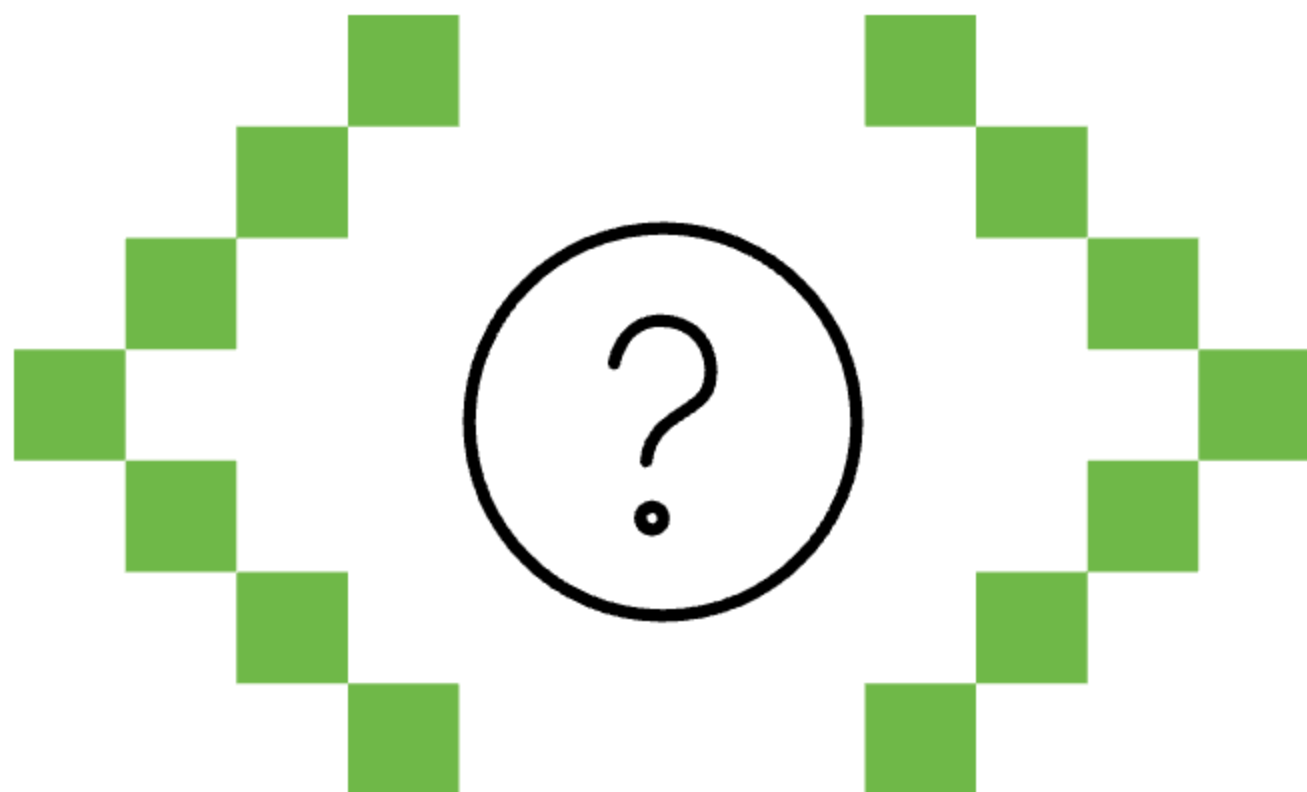
제네릭 명명 규칙

- T : Type(일반적인 타입)
- E : Element(컬렉션의 요소)
- K : Key(키)
- V : Value(값)
- N : Number(숫자)
- R : Result(결과)



02

와일드 카드 활용



와일드 카드

제네릭 타입에서 "정확한 타입을 모르거나,
신경 쓰지 않을 때" 사용하는 타입 표현


```

public static void main(String[] args) {
    Set<String> ex1 = Set.of("새로이", "비타", "라젤");
    Set<String> ex2 = Set.of("월슨", "밍트");

    Set<Integer> ex3 = Set.of(1, 2, 3);
    Set<Double> ex4 = Set.of(1.0, 2.0, 3.0, 4.1, 5.2);

    union(ex1, ex2);
    union(ex3, ex4); // 컴파일 에러
}

private static <E> Set<E> union(Set<E> s1, Set<E> s2) {
    Set<E> result = new HashSet<>(s1);
    result.addAll(s2);
    return result;
}

```

예외 

	Required type	Provided
s1:	Set<E>	Set<Integer>
s2:	Set<E>	Set<Double>

```
public static void main(String[] args) {
    Set<String> ex1 = Set.of("새로이", "비타", "라젤");
    Set<String> ex2 = Set.of("월슨", "밍트");

    Set<Double> ex3 = Set.of(1.0, 2.1, 3.5);
    Set<Integer> ex4 = Set.of(1, 2, 3, 4, 5);

    union(ex1, ex2);
    union(ex3, ex4);
}

private static <E> Set<E> union(Set<? extends E> s1, Set<? extends E> s2) {
    Set<E> result = new HashSet<>(s1);
    result.addAll(s2);
    return result;
}
```

와일드 카드 활용 개선

제네릭 타입의 매개변수를 와일드 카드 활용으로
더 유연한 설계 가능

03

제네릭 싱글톤 팩토리 패턴





제네릭 싱글톤 팩토리 패턴

불변 객체를 여러 타입으로 활용하는 상황

- 제네릭은 하나의 객체를 어떤 타입으로든 매개변수화 가능
- 요청한 타입 매개변수에 알맞는 타입으로 바꿔주는 정적 팩토리 필요

항등 함수

입력값을 그대로 반환하는 함수

제네릭 사용 ❌

```
1 public static class IdentityFactory {  
2     private static final UnaryOperator<Object> IDENTITY = o -> o;  
3  
4     public static UnaryOperator<Object> get() {  
5         return IDENTITY;  
6     }  
7 }
```

항등 함수

입력값을 그대로 반환하는 함수

제네릭 사용 ○

```
1 public static class IdentityFunction {  
2     private static final UnaryOperator<Object> IDENTITY = t -> t;  
3  
4     public static <T> UnaryOperator<T> get() {  
5         return (UnaryOperator<T>) IDENTITY;  
6     }  
7 }
```

제네릭 싱글톤 패턴

특정 타입을 유동적으로 추출 가능

```
UnaryOperator<Object> general = IdentityFactory.get(); // 제네릭 ❌  
UnaryOperator<String> generic = IdentityFunction.get(); // 제네릭 ✅  
  
String ex1 = general.apply(t: "hello");  
String ex2 = generic.apply(t: "hello");
```



```
1 UnaryOperator<String> stringIdentity = s -> s;
2 UnaryOperator<Integer> intIdentity = i -> i;
3
4 System.out.println(stringIdentity.apply("hello"));
5 System.out.println(stringIdentity);
6
7 System.out.println(intIdentity.apply(123));
8 System.out.println(intIdentity);
```

hello

GenericSingleEx1\$\$Lambda/0x00000003010031f8@4e50df2e

123

GenericSingleEx1\$\$Lambda/0x0000000301003440@1d81eb93



1. 불필요한 객체 지속적인 생성
2. 중복 코드 발생
3. 클라이언트 재사용 불가능



```
1 List<String> strings = List.of("hello", "world");
2 List<Number> integers = List.of(1, 2.0, 3L);
3
4 UnaryOperator<String> stringIdentity = IdentityFunction.get();
5 UnaryOperator<Number> intIdentity = IdentityFunction.get();
6
7 System.out.println(stringIdentity);
8 for (String string : strings) {
9     System.out.println(stringIdentity.apply(string));
10 }
11
12 System.out.println(intIdentity);
13 for (Number integer : integers) {
14     System.out.println(intIdentity.apply(integer));
15 }
```



```
GenericSingleEx2$IdentityFunction$$Lambda/0x00000003010033f0@2f4d3709
변경 : hello
타입 : class java.lang.String

변경 : world
타입 : class java.lang.String

GenericSingleEx2$IdentityFunction$$Lambda/0x00000003010033f0@2f4d3709
변경 : 1
타입 : class java.lang.Integer

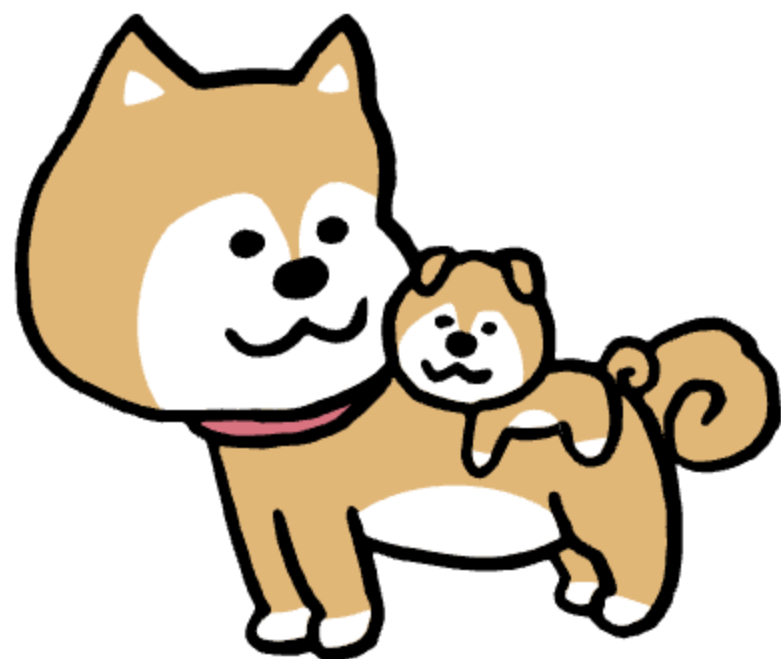
변경 : 2.0
타입 : class java.lang.Double

변경 : 3
타입 : class java.lang.Long
```



04


재귀적 타입 한정






재귀적 타입 한정


- 타입 매개변수 허용 범위를 한정
 - 타입 안전한 비교 가능
 - 정확한 타입 보장
 - 실수 방지



```
1 public static class User {  
2  
3     private final String name;  
4     private final int age;  
5  
6     public User(final String name, final int age) {  
7         this.name = name;  
8         this.age = age;  
9     }  
10 }
```



```
1 User 새로이 = new User("새로이", 50);  
2 User 밍트 = new User("밍트", 11);  
3 System.out.println("나이 높은 사람 \n" + max(새로이, 밍트));
```



```
1     public static <E extends Comparable<E>> E max(E a, E b) {  
2         return a.compareTo(b) > 0 ? a : b;  
3     }
```

```
1 public static class User {  
2  
3     private final String name;  
4     private final int age;  
5  
6     public User(final String name, final int age) {  
7         this.name = name;  
8         this.age = age;  
9     }  
10 }
```

```
1 User 새로이 = new User("새로이", 50);  
2 User 밍트 = new User("밍트", 11);  
3 System.out.println("나이 높은 사람 \n" + max(새로이, 밍트));
```

```
1     public static <E extends Comparable<E>> E max(E a, E b) {  
2         return a.compareTo(b) > 0 ? a : b;  
3     }
```

```

public static void main(String[] args) {
    User 새로이 = new User( name: "새로이", age: 50);
    User 밉트 = new User( name: "밉트", age: 11);
    System.out.println("나이 높은 사람 \n" + max(새로이, 밉트));
}

```

	Required type	Provided
a:	E	User
b:	E	User

reason: no instance(s) of type variable(s) E exist so that User conforms to Comparable<E>


```

1 public static class User implements Comparable<User> {
2
3     private final String name;
4     private final int age;
5
6     public User(final String name, final int age) {
7         this.name = name;
8         this.age = age;
9     }
10
11     @Override
12     public int compareTo(final User o) {
13         return Integer.compare(age, o.age);
14     }
15 }

```

```

1 public static <E extends Comparable<E>> E max(E a, E b) {
2     return a.compareTo(b) > 0 ? a : b;
3 }

```

나이 높은 사람
이름 : 새로이
나이 : 50



00

정리

- 범용성
 - 여러 타입에 대해 유연하게 작동하는 메서드 생성
- 클래스 오염 방지
 - 클래스 전체를 제네릭으로 만들 필요 없이, 필요한 메서드만 제네릭 처리 가능
- 재사용성 향상
 - 다양한 곳에서 재사용 가능 (유틸, 정렬, 반환 등)
- 타입 안정성
 - 컴파일 과정에서 잘못된 타입을 추출 가능

★ ★ 요약정리!!
보기

Item 30.

이왕이면 제네릭 메서드로 만들라

