



아이템 30 이왕이면 제네릭 메서드로 만들라

TOPIC : generic method, generic singleton factory, recursive type bound

- **제네릭 메서드**의 대표적 예로 매개변수화 타입을 받는 정적 유틸리티 메서드가 있다.

Collections의 '알고리즘'메서드 (binarySearch, sort 등)는 모두 제네릭이다.

▼ binarySearch와 sort 코드

```
public static <T>
int binarySearch(List<? extends Comparable<? super T>> list, T key) {
// 모든 원소가 T의 슈퍼타입과 비교 가능한 Comparable을 상속하고 있는 list와 T(타입) key를 비교한다
    if (list instanceof RandomAccess || list.size()<BINARYSEARCH_THRESHOLD)
        return Collections.indexedBinarySearch(list, key);
    else
        return Collections.iteratorBinarySearch(list, key);
}
```

```
@SuppressWarnings("unchecked")
public static <T extends Comparable<? super T>> void sort(List<T> list) {
    list.sort(null);
}
```

문제있는 코드

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

```
Union.java:9: warning: [unchecked] unchecked call to HashSet(Collection<? extends E>) as a member of the raw type HashSet
    Set result = new HashSet(s1);
                   ^
where E is a type-variable:
  E extends Object declared in class HashSet
Union.java:10: warning: [unchecked] unchecked call to addAll(Collection<? extends E>) as a member of the raw type Set
    result.addAll(s2);
               ^
```

2가지 경고가 발생한다.

경고를 없애려면 ? 메서드를 타입 안전하게 만들어야 한다

- 메서드 선언에서의 3개의 Set(s1, s2, result)의 원소타입을 타입 매개변수로 명시하고, 메서드 안에서도 이 타입 매개변수만 사용하게 수정하면 된다.
- (타입 매개변수들을 선언하는) 타입 매개변수 목록은 메서드의 제한자와 반환 타입 사이에 온다.

```
// 타입 매개변수 목록은 <E> 이고 반환 타입은 Set<E>이다.
public static <E> Set<E> union(Set<E> s1, Set<E> s2) {
    Set<E> result = new HashSet<>(s1);
    result.addAll(s2);
    return result;
}
```

위 코드는 경고 없이 컴파일 되며, 타입 안전하고 쓰기도 쉽다.

불변 객체를 여러 타입으로 활용할 수 있게 만들어야 할 때는 ?

- 제네릭은 런타임에 타입 정보가 소거되므로 하나의 객체를 어떤 타입으로든 매개변수화 할 수 있다.
- 요청한 타입 매개변수에 맞게 **매번 그 객체의 타입을 바꿔주는 정적 팩터리**를 만들어야한다.

💡 제네릭 싱글턴 팩터리

- Collections.reverseOrder(함수객체)나 Collections.emptySet (컬렉션용)으로 사용한다.

▼ reverseOrder, emptySet 코드

```
@SuppressWarnings("unchecked")
public static <T> Comparator<T> reverseOrder() {
    // T 타입으로 캐스팅 된 싱글턴 ReverseComparator 를 반환, 타입 정보는 소거돼서 매번 캐스팅만 된다
    return (Comparator<T>) ReverseComparator.REVERSE_ORDER;
}
```

```
private static class ReverseComparator Complexity is 3 Everything is cool!
    implements Comparator<Comparable<Object>>, Serializable {

    private static final long serialVersionUID = 7207038068494060240L;

    static final ReverseComparator REVERSE_ORDER
        = new ReverseComparator();

    public int compare(Comparable<Object> c1, Comparable<Object> c2) { return c2.compareTo(c1); }

    private Object readResolve() { return Collections.reverseOrder(); }

    @Override
    public Comparator<Comparable<Object>> reversed() { return Comparator.naturalOrder(); }
}
```

```
List<String> ex1 = Arrays.asList("a", "1", "가");
ex1.sort(Collections.reverseOrder()); // 함수 내부에 들어가는 객체를 함수객체라 한다.
System.out.println(ex1); // [가, a, 1]
```

```
@SuppressWarnings("unchecked")
public static final <T> Set<T> emptySet() {
    // T 타입으로 캐스팅해서 반환하는 제네릭 싱글턴 팩터리
    return (Set<T>) EMPTY_SET;
}
//
@SuppressWarnings("rawtypes")
public static final Set EMPTY_SET = new EmptySet<>();
```

- 책에 나온 예제 - 항등 함수

```
private static UnaryOperator<Object> IDENTITY_FN = (t) -> t;
/*
    항등함수란 입력값을 수정없이 그대로 반환하는 함수
    이므로 T가 어떤 타입이든 UnaryOperator<T>를 사용해도 타입안전하다.
*/
@SuppressWarnings("unchecked") // 비검사 형변환 경고를 숨기자.
public static <T> UnaryOperator<T> identityFunction() {
    return (UnaryOperator<T>) IDENTITY_FN;
}
```

```
GenericSingletonFactory.java:11: warning: [unchecked] unchecked cast
    return (UnaryOperator<T>) IDENTITY_FN;
           ^
required: UnaryOperator<T>
found:    UnaryOperator<Object>
where T is a type-variable:
  T extends Object declared in method <T>identityFunction()
1 warning
```

@SuppressWarnings("unchecked") 없을 시,
IDENTITY_FN 을 UnaryOperator<T> 로 형변환 하면 비검사 형변환 경고가 발생한다.
T 가 어떤 타입이든 UnaryOperator<Object>는 UnaryOperator<T>가 아니기 때문이다.

```

public static void main(String[] args) {
    String[] strings = { "삼베", "대마", "나일론" };
    UnaryOperator<String> sameString = identityFunction();
    for (String s : strings)
        System.out.println(sameString.apply(s));

    Number[] numbers = { 1, 2.0, 3L };
    UnaryOperator<Number> sameNumber = identityFunction();
    for (Number n : numbers)
        System.out.println(sameNumber.apply(n));
}

```

▼ 연습 코드

```

public class GenericSingletonFactoryEx {
    private static BinaryOperator<Object> FIRST = (t1, t2) -> t1;

    @SuppressWarnings("unchecked")
    public static <T> BinaryOperator<T> firstFunction() {
        return (BinaryOperator<T>) FIRST;
    }

    public static void main(String[] args) {
        String[] strings = {"a", "b"};
        String c = "c";
        BinaryOperator<String> binaryOperator = firstFunction();
        for (String s : strings)
            System.out.println(binaryOperator.apply(s, c));

        int[] numbers = {1,2,3};
        int other = 4;
        BinaryOperator<Integer> binaryOperatorNum = firstFunction();
        for(int n : numbers)
            System.out.println(binaryOperatorNum.apply(n, other));
    }
}

```

- 입력 인수와 반환 타입이 같다, 두 인수는 타입이 같아야 한다.
- 첫번째 인수를 반환하는 제네릭 싱글턴 팩터리 함수

```

> Task :GenericSingletonFactoryEx.main()
a
b
1
2
3

```

💡 재귀적 타입 한정 (recursive type bound)

- **재귀적 타입 한정** : 자기 자신이 들어간 표현식을 사용해 타입 매개변수의 허용범위를 한정

주로 타입의 자연적 순서를 정하는 Comparable 인터페이스와 함께 쓰인다.

```

public interface Comparable<T>{
    int compareTo(T o)
}
// 타입 매개변수 T는 Comparable<T>를 구현한 타입이 비교할 수 있는 원소의 타입을 정의한다.
// 즉, T는 비교 가능한 타입이다.(자신과 같은 타입의 원소와 비교)

```

- Comparable을 구현한 컬렉션의 모든 원소가 상호 비교되어야 하는데, 그 제약을 <E extends Comparable<E>> 라고 표현한다.

```

public static <E extends Comparable<E>> E max(Collection<E> c);

```

```

public static <E extends Comparable<E>> E max(Collection<E> c) {
    if (c.isEmpty())
        throw new IllegalArgumentException("Empty collection"); // Optional<E> 를 반환하도록 고치는 편이 낫다 (아이템 55)

    E result = null;
}

```

```
for (E e : c)
    if (result == null || e.compareTo(result) > 0)
        result = Objects.requireNonNull(e);

return result;
}
```

핵심 정리

- 제네릭 타입과 마찬가지로 클라이언트에서 입력 매개변수와 반환 값을 명시적으로 형변환해야 하는 메서드보다 제네릭 메서드가 더 안전하며 사용하기 쉽다. 타입과 마찬가지로, 메서드도 형변환 없이 사용할 수 있는 편이 좋으며, 많은 경우 그렇게 하려면 제네릭 메서드가 되어야 한다.
- 형변환을 해줘야 하는 기존 메서드는 제네릭하게 만들자.