이펙티브 자바 제네릭

item26. 로 타입은 사용하지 말라

클래스 혹은 인터페이스 선언에 타입 매개변수가 쓰이면 이를 제네릭 타입이라고 한다.

제네릭 타입은 일련의 매개변수화 타입을 정의한다.

제네릭 타입을 정의하면 그에 딸린 raw type을 정의한다.

raw type은 제네릭 타입에서 타입 매개변수를 사용하지않은 것 ex)List, Set

```
// 제네릭을 지원하기 전엔 컬렉션을 아래와 같이 선언
private final Collection stamps = ...;

stamps.add(new Coin(...));

for (Iterator it = stamps.iterator(); i.hasNext();) {
  Stamp stamp = (Stamp) i.next(); // ClassCastException
  stamp.cancel();
}
```

raw type을 쓰면 제네릭이 주는 안정성과 표현력을 모두 잃는다.

List<Object>는 사용해도 괜찮지만, List는 사용하면 안된다.

명확하게 Object의 하위타입을 받겠다고 선언한것이지만, List는 제네릭을 버린 것이다.

모든 타입을 받기 위해서는 비 한정적 와일드카드 타입을 사용하자.

```
private void unsafeAdd(List<?> list, Object o) {
    list.add(o); // 컴파일 에러! null 외에 넣을 수 없음
```

```
}
```

null만 넣을 수 있는 제약을 벗어나고 싶다면 ? extends class 와 같이 한정적 와일드카드 타입을 이용해 어떤 클래스의 하위 클래스를 받을 것인지 명시하면 된다.

로 타입을 쓸 수 밖에 없는 경우

class 리터럴을 사용할 때

- 자바 명세는 class 리터럴에 매개변수화 타입을 사용하지 못하게 했다.
- List.class, int.class는 허용하고, List<String>.class는 허용하지 않음

instanceof 연산자를 사용할 때

- 런타임에는 제네릭 정보가 지워져서 instanceof 연산자는 비 한정적 와일드 카드 타입 이외의 매개변수화 타입에 적용이 불가하다.
- raw type과 비한정적 와일드 카드 타입의 instanceof는 완전히 똑같이 동작한다.

```
if (o instanceof Set) {  // raw type
  Set<?> s = (Set<?>) o; // 와일드카드 타입
  ...
}
```

o가 Set타입인지 확인후 형변환을 진행해야한다.

item27. 비검사 경고를 제거하라

비검사 경고가 발생하는 경우

```
Set<Lark> exaltation = new HashSet();
```

- 비검사 경고가 발생하는 코드이다.
- raw type을 사용해서 경고가 발생한다.
- 자바 7부터는 <> 연산자를 뒤에 적어주면 자동으로 타입 추론을 해준다.
- 이걸해줘야 타입 안정성 보장 및 런타임에 ClassCastException 확률이 줄어들음

만약 비검사 경고를 제거할 수 없다면?

경고를 제거할 수는 없지만 안전하다는 확신이 있다면 @SuppressWarnings("unchecked")를 달아서 경고를 숨길 수 있다.

→ 가장 작은 범위로 설정해야한다. ex) 변수 선언, 짧은 메서드, 생성자..

item28. 배열보다는 리스트를 사용하라

배열은 공변(함께 변한다)이 적용된다.

리스트는 불공변(함께 변하지 않는다.)이 적용된다.

```
@Test
public void covariantTest() {
    Object[] objects = new Long[1];
    objects[0] = "String"; // ArrayStoreException
}
```

Object[]의 공변 특성을 이용해 objects를 선언 후에 Long[] 저장소를 만들어 할당했다.

object[] 타입만 보고 문자열을 넣은 경우 ArrayStoreException이 발생한다.

```
@Test
public void invariantTest() {
```

```
    List<Object> ol = new ArrayList<Long>(); // 불공변이라 타입 자체
    ol.add("아아");
    // 제네릭의 타입 정보가 런타임에는 이미 사라져버리기 때문에 컴파일 타임에
    // 런타임에 제네릭 타입을 소거시키는 이유는 레거시 코드와 제네릭 타입을 함
}
```

불공변 특성을 가지는 리스트에서는 하위 타입과 같은 관계 자체가 존재하지 않고, 모든 관계가 그저 다른 타입일 뿐이므로 에러가 발생한다.

배열은 실체화된다

- 배열이 실체화된다는 것은 자신이 담기로 한 원소의 타입을 인지하고 확인한다는 뜻이다.
- 제네릭은 런타임에 타입정보가 소거된다.
- 배열은 제네릭 타입, 매개변수화 타입, 타입 매개변수로 사용할 수 없다.

→ new List<E>[], new E[] 이런식으로 작성하면 컴파일 시 제네릭 배열 생성 오류가 발생

제네릭은 실체화되지 않는다

E, List<E> 같은 타입은 실체화 불가 타입이다.

런타임에 컴파일타임보다 타입 정보를 적게 가지는 타입이다.

소거 매커니즘 때문에, 매개변수화 타입 가운데 실체화될 수 있는 타입은 List<?>와 Map<?, ?> 같은 비한정적 와일드카드 타입 뿐이다.

배열로 형변환하는 경우 배열인 E[] 대신 컬렉션인 List<E>를 사용하면 해결된다.

item29. 이왕이면 제네릭 타입으로 만들라

Object배열을 필드로 갖는 클래스를 제네릭 배열로 만들면 생성시 에러가발생한다.

```
private E[] elements;

 public Generic() {
        elements = new E[DEFAULT_INITIAL_CAPACITY];
 }
```

위 생성자는 generic array creation이 발생함

제네릭 타입 배열만들기1 : Object 배열 생성 후 (E[])로 형변환하기

```
@SuppressWarnings("unchecked")
public Generic() {
    elements = (E[]) new Object[DEFAULT_INITIAL_CAPACITY];
}
```

- Object 배열을 생성한 뒤에 (E[]) 와 같이 제네릭 배열로 캐스팅하는 방법이다.
- 배열에 값을 추가하는 작업을 할때만 elements 배열의 타입을 알아야한다.

제네릭 타입 배열만들기2 : 새 제네릭 타입 변수에 형변환하여 할당하기

```
public E pop() {
  if(size == 0) {
    throw new EmptyStackException();
  }

  //새 제네릭 타입 변수에 형변환하여 할당
  @SuppressWarnings("unchecked")
  E result = (E) elements[--size];

  elements[size] = null;
```

```
  return result;
}
```

제네릭 타입 배열을 사용하는 두 가지 방법 정리

- 첫번째 방식은 형변환을 배열 생성 시 단 한 번만 해주면 된다.
- 두번째 방식은 배열에서 원소를 읽을 때마다 해주어야 한다.
- 첫번째 방식이 더 자주 사용되지만, 배열의 런타임 타입이 컴파일타임 타입과 달라 힙 오염을 일으킨다.
    - 힙 오염이란 타입 캐스팅 시 컴파일러가 타입 캐스팅 객체를 진짜 캐스팅할 수 있는지 검사하지 않고, 캐스팅했을 때 대입되는 참조변수에 저장할 수 있느냐만 검사하기 때문에 일어나는 현상이다.
        - 컴파일러는 컴파일 이후에 제네릭 타입 파라미터를 전혀 신경쓰지 않는다.
        - 즉, ArrayList<Integer>, ArrayList<String> 모두 컴파일 이후에는 ArrayList<Object>가 된다.
    - 힙 오염은 잠재적으로 ClassCastException을 일으킬 수 있다.

item30. 이왕이면 제네릭 메서드로 만들라

제네릭 메서드

```
public static <E/*타입 매개변수 목록*/> Set<E/*반환 타입*/> union(Set
    Set<E> result = new HashSet<>(s1);
    result.addAll(s2);
    return result;
}
```

사용 예시

```
@Test
public void unionTest() {
    Set<String> A = Set.of("A", "B", "C");
    Set<String> B = Set.of("D", "E", "F");
    Set<String> C = union(guys, stooges);
    System.out.println("C = " + C);
}
```

- 제네릭은 런타임에 타입 정보가 소거되므로 하나의 객체를 어떤 타입으로든 매개변수화 할 수 있다.
- 이를 이용해 불변 객체를 여러 타입으로 이용할 수 있게 만들 수도 있다.

```
private static final UnaryOperator<Object> IDENTITY_FN = (t) ->

@SuppressWarnings("unchecked")
public static <T> UnaryOperator<T> identityFunction() {
    return (UnaryOperator<T>) IDENTITY_FN;
}
```

- 위는 제네릭 싱글턴 팩터리로 항등함수를 작성한 예이다.
- 제네릭으로 수용 가능한 어떤 타입이든 작동하는 항등함수이다.
- UnaryOperator를 어떤 타입이든 계속해서 이용할 수 있다.

재귀적 타입한정 이용하기

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

- E로 받을 타입은 오직 Comparable<E>를 구현한 타입만 가능하다는 뜻이다.
- 즉, Comparable을 구현한 타입만 가능하다는 뜻이다.

item31. 한정적 와일드카드를 사용해 API 유연성을 높이라

와일드 카드 타입을 사용하는 이유

- 제네릭은 불공변이다
- 하위 혹은 상위 타입을 기본적으로 포용하지 않음
- 필요하다면, extends 혹은 super를 이용

```
static class Gneric<E> {
      private E[] elements;

      public Generic() {
        elements = (E[])  new Object[DEFAULT_INITIAL_CAPACITY];
    }


         /* 새로 추가된 메서드 */
    public void pushAll(Iterable<E> src) {
        for (E e : src) {
            push(e);
        }
    }

    @Override
    public String toString() {
        return "Gneric{" +
                "elements=" + Arrays.toString(elements) +
                ", size=" + size +
                '}';
```

```
    }
}
```

```
@Test
public void Test() {
    Generic<Number> generic = new Generic<>();
    List<Integer> integers = List.of(1, 2, 3);
    generic.pushAll(integers);
    System.out.println("generic = " + generic);
}
```

위 코드는 에러가 난다. Number 타입의 스택에 Integer는 하위타입이기에 들어갈 수 있다고 생각하지만, 제네릭이 불공변이어서 에러가 난다.

생산자 버전

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

- pushAll()을 위와 같이 바꾸면, 더이상 에러가 나지 않는다.
  - 단순히 E만 이용했을 때는 Number만 받을 수 있지만, 이제 Integer도 받을 수 있다.
- ? 와일드카드 타입으로 E를 상속한 아무 타입이나 받아줄 수 있다.
  - 제네릭의 불공변 때문에 이렇게 한정적 와일드카드 타입(? extends E)을 이용해주는 것이 좋다.

소비자 버전

```
public void popAll(Collection<E> dst) {
    while(size > 0) {
        dst.add(pop());
    }
}
```

- popAll()도 위의 pushAll() 케이스와 비슷하게, 제네릭의 불공변 특성 때문에 에러가 나는 케이스이다.
- 불공변덕에 Generic의 제네릭이 Number였어서 유연성 없이 Collection<Number> 타입만을 인수로 받을 수 있는데, 상위 타입인 Object가 와서 문제인 것이다.
- 소비자의 경우, 생산자와 다르게 상위 클래스로만 유연하게 확장해야 한다.
- 클래스가 가진 컬렉션 필드에 수용할 때는 많은 정보 중 필요한 정보만 취사선택 하면 된다. (생산자 관점)
    - 정보가 더 많아서 나쁠 게 없다. (상위 타입은 하위 타입을 수용할 수 있다. 상속 덕에 필요한 정보가 이미 다 있다.)
- 클래스가 가진 컬렉션 필드를 꺼내올 때는 해당 컬렉션이 가진 정보를 받아줄 객체가 필요하다. (소비자 관점)
    - 상위 객체만이 하위 객체를 받아줄 수 있다. (하위 타입은 상위 타입을 수용할 수 없다. 더 많은 정보가 필요하다.)
- 그러므로 와일드카드에 super 키워드를 사용해야 한다.

```
public void popAll(Collection<? super E> dst) {
    while(size > 0) {
        dst.add(pop());
    }
}
```

- 상위 타입일수록 더 적은 정보를 가지고 있다. 하위 타입일수록 더 구체적인 다른 정보를 많이 가지고 있다.
- 생산자일 때는 하위 타입을 받아도 객체는 필요한 정보만 쓰면 되니, extends를 쓴다.
- 소비자일 때는 소비자의 정보를 받아줄 타입이 필요 하니, super를 쓴다.
- 만일 매개변수가 생산자와 소비자 둘의 역할을 모두 하면, 어느것을 써도 좋을 것이 없기 때문에 그냥 정확한 타입을 써야 한다.
    - 생산을 해야 하는데 상위 객체가 온다면 정보가 부족하고, 소비를 해야 하는데 하위 객체가

> 위와 같이 생산자(producer)에는 extends, 소비자(consumer)에는 super를 쓴다고 해서 PECS라고 부른다.

item32. 제네릭과 가변인수를 함께 쓸 때는 신중하라

가변인수 메서드의 허점

- 가변인수 메서드를 호출하면 가변인수를 담기 위한 배열이 자동으로 하나 만들어진다.
- 이 배열은 내부로 감춰져야 하는데, 클라이언트에 공개되면서 문제가 발생할 수 있다.
- 특히 varargs 변수에 제네릭이나 매개변수화 타입이 포함되면 알기 어려운 컴파일 경고가 발생한다.

```
@Test
public void unableToReifyTest() {
    Assertions.assertThrows(ClassCastException.class, () -> {
        reifyExampleMethod(List.of("A","B","C"));
    });
}
```

```
public static void reifyExampleMethod(List<String>... stringList
    List<Integer> intList = List.of(42);
    Object[] objects = stringLists;
    objects[0] = intList;
    String s = stringLists[0].get(0); // ClassCastException 발생
}
```

자바 표준 API에서의 제네릭 가변인수 활용

```
@SafeVarargs
@SuppressWarnings("varargs")
public static <T> List<T> asList(T... a) {
    return new ArrayList<>(a);
}
```

- 가변 인수를 외부에 노출하는 행위는 위험
- 다른 배열 값에 저장하여 조작하는 등의 행위는 위험

위험한 제네릭 가변인수 활용

```
public static <T> T[] toArray(T... args) {
    // 가변인자를 그대로 반환하는 것은 외부 메서드에서 사용돼선 안된다.
    // 가변인자를 그대로 반환하여 외부에 노출하지 말자!
    return args;
}
```

- 가변인자를 그대로 외부에 노출해서 위험하다.
- 이 메서드가 반환하는 배열의 타입은 이 메서드에 인수를 넘기는 컴파일 타임에 결정되는데, 그 시점에 컴파일러에게 충분한 정보가 주어지지 않아 타입을 잘못 판단할 수 있다.
- 잘못된 타입을 그대로 반환하면, 힙 오염을 클라이언트의 콜스택까지 전달하는 결과를 낳을 수도 있다

```
/**
 * 가변인자 제네릭 인자를 반환하는 `toArray()`의 결과를 이용하기 때문에
 * 항상 `Object[]` 타입을 반환한다.
 * **중요**, 제네릭 varargs 배열에 다른 메서드가 접근하도록 허용하면 안전히
 * 제네릭 varargs 배열은 사용하는 해당 메서드에서만 접근하는 것이 좋을 것이다
 */
public static <T> T[] pickTwo(T a, T b, T c) {
    switch(ThreadLocalRandom.current().nextInt(3)) {
        case 0: return toArray(a, b);
        case 1: return toArray(a, c);
        case 2: return toArray(b, c);
    }

    throw new AssertionError(); // 도달할 수 없다.
}
```

위 메서드는 가변 제네릭 인자를 반환한 toArray()의 결과를 이용하기 때문에 위험하다.

```
@Test
public void test() {
    String[] strings = toArray("일", "이", "삼");
    System.out.println("strings = " + Arrays.toString(string
s));

    Assertions.assertThrows(ClassCastException.class, () -> {
// 만일, `String[] pickTwo`와 같이 코드를 작성하면,// 여기서 컴파일러
는 `String[]`로 `pickTwo()`의 결과를 캐스팅하려고 한다.
        String[] pickTwo = pickTwo("일", "이", "삼");
        System.out.println("pickTwo = " + Arrays.toString(pic
kTwo));
    });
}
```

- 위의 클라이언트 코드를 이용하여 실행했을 때, pickTwo()의 결과는 Object[] 타입으로 반환된다. 그러나, String[] 타입으로 받기 때문에 자바의 묵시적 캐스팅 때문에 ClassCastException이 날 것이다.
- 힙 오염을 발생시킨 진짜 원인인 toArray()와 매우 떨어져있어서 진짜 원인을 찾는데도 오래걸린다.

안전한 제네릭 가변인수 활용

```
@SafeVarargs
static <T> List<T> flatten(List <? extends T>... lists) {
    List<T> result = new ArrayList<>();

    for (List<? extends T> list : lists) {
        result.addAll(list);
    }

    return result;
}
```

- 위의 메서드는 위험한 행위 2개를 하지 않았기에 안전하다.
  - 제네릭 가변인수를 외부로 노출시키지 않았다.
  - 제네릭 가변인수를 다른 배열에 담아서 조작하지 않았다.
- 또한 @SafeVarargs 애너테이션을 이용하여 안전함을 표시했고, 그래서 컴파일 에러도 뜨지 않는다.

> 단, @SafeVarargs는 재정의할 수 없는 메서드에만 달아야 한다. 상속할 때도 애노테이션이 이어지기 때문에 하위 타입의 구현이 정말로 안전한 가변인수인지는 알기 힘들다.

item33. 타입 안전 이종 컨테이너를 고려하라

타입 안전 이종 컨테이너란?

```
static class Favorites {
    private final Map<Class<?>, Object> favorites = new HashM
ap<>();

    public <T> void putFavorite(Class<T> type, T instance) {
        favorites.put(Objects.requireNonNull(type), instanc
e);
    }

    public <T> T getFavorite(Class<T> type) {
        return type.cast(favorites.get(type));
    }
}
```

- 하나의 프로그래밍 패턴이다.
- Map에서 key를 특정 클래스로 받고, value를 Object의 형태로 받는다.
- 데이터를 넣을 때, key의 클래스 타입과 instance의 타입이 동일해야 하므로, 타입 안전이 보장된다.
    - 특정 클래스 타입의 key를 불렀을때 해당 타입의 인스턴스가 나올 것이라 예측 가능하다.
- 이 방식이 동작하는 이유는 Class 클래스가 기본적으로 제네릭 타입을 받는 클래스기 때문이다.
    - 컴파일타임 타입 정보와 런타임 타입 정보를 알아내기 위해 메서드들이 주고받는 class 리터럴을 타입토큰이라 한다.
- Map<Class<?>, Object>에서는 와일드카드가 중첩되어 맵이 아니라 key가 와일드 타입임을 인지해야 한다.
    - 그래서 모든 키가 서로 다른 매개변수화 타입일 수 있다는 의미가 된다.

> 흔히 제네릭 타입을 쓰는 클래스들은 매개변수화 대상을 컨테이너 자신으로 하는 반면, 타입 안전 이종 컨테이너는 매개변수화 대상을 키로 사용하는 Class에 둔다.

제약 1 : 클라이언트의 악의적인 raw type 사용

- 악의적인 클라이언트가 Class 객체를 제네릭이 아닌 로 타입으로 넘기면 Favorites 인스턴스의 타입 안전성이 쉽게 깨진다.
    - 하지만, 컴파일할 때 비검사 경고가 뜰 것이다.
- 위와 같이 악의적으로 (Class)와 같은 캐스팅을 통해 로타입을 사용하는 것에 대한 취약점이 있다.
- 그냥 실행하면, 런타임에 ClassCastException을 만나게 될 것이다.

제약 2: 실체화 불가 타입에는 사용할 수 없다.

- List<String>.class와 같이 실체화 되지 않는 타입에는 사용 불가능하다.

슈퍼 타입 토큰

위의 제약을 해결하려 슈퍼 타입 토큰(super type token)을 사용하려는 시도도 있었다. 스프링에서는 ParameterizedTypeReference라는 클래스로 미리 구현해놓았다.

```
Favirotes f = new Favorites();

List<String> alpa = Arrays.asList("A", "B", "C");

f.putFavorite(new TypeRef<List<String>>(){}, alpa);
List<String> listofStrings = f.getFavorite(new TypeRef<List<S
tring>>(){});
```

위와 같은 코드를 사용하여 해결한다.

한정적 타입 토큰을 이용한 타입 제한

- Favorites가 사용하는 타입 토큰은 기본적으로 비 한정적이다. 한정적 타입 매개변수나 한정적 와일드카드를 이용하여 표현 가능한 타입을 제한하는 것을 한정적 타입 토큰이라고 한다.
  - 애너테이션 API에서는 이를 적극 활용한다.
  - 애너테이션 API는 대상 요소에 달려있는 애너테이션을 런타임에 읽어오는 기능을 한다.
  - 아래는 애너테이션 API인 AnnotatedElement 코드 예이다.

```
public <T extends Annotation>
  T getAnnotation(Class<T> annotationType);
```

- annotationType 인수는 애너테이션 타입을 뜻하는 한정적 타입 토큰이다.
- 이 메서드에서는 토큰으로 명시한 타입의 애너테이션이 해당 요소에 달려있다면 애너테이션을 반환하고 없으면 null을 반환한다.
- 애너테이션된 요소는 그 키가 애너테이션 타입인 타입 안전 이종 컨테이너인 것이다.

한정적 타입 토큰을 받는 메서드에 Class<?> 타입의 객체를 넘기는법

- Class 내부 메서드인 asSubclass 메서드를 활용하면 된다.
  - 호출된 인스턴스 자신의 Class 객체를 인수가 명시한 클래스로 형변환한다.
  - 형변환에 성공하면 인수로 받은 클래스 객체를 반환하고 실패하면 ClassCastException을 던진다.

```
static Annotation getAnnotation(AnnotatedElement element, Str
ing annotationTypeName) {
  Class<?> annotationType = null;// 비 한정적 타입 토큰try {
    annotationType = Class.forName(annotationTypeName);
  } catch (Exception ex) {
    throw new IllegalArgumentException(ex);
```

```
  }

  return element.getAnnotation(annotationType.asSubclass(Anno
tation.class));
}
```