<5장> 제네릭

> 제네릭(generic)은 자바 5부터 사용할 수 있다. 제네릭을 지원하기 전에는 컬렉션에서 객체를 꺼낼 때마다 형변환을 해야 했다. 그래서 누군가 실수로 엉뚱한 타입의 객체를 넣어두면 런타임에 형변환 오류가 나곤 했다. 반면, 제네릭을 사용하면 컬렉션이 담을 수 있는 타입을 컴파일러에 알려주게 된다. 그래서 컴파일러는 알아서 형변환 코드를 추가할 수 있게 되고, 엉뚱한 타입의 객체를 넣으려는 시도를 컴파일 과정에서 차단하여 더 안전하고 명확한 프로그램을 만들어준다. 꼭 컬렉션이 아니더라도 이러한 이점을 누릴 수 있으나, 코드가 복잡해진다는 단점이 따라온다.
>
> 이번 장에서는 제네릭의 이점을 최대로 살리고 단점을 최소화하는 방법을 이야기한다.

📌 **5장 용어 사전**

한글 용어	영문 용어	예	아이템
매개변수화 타입	parameterized type	List<String>	아이템 26
실제 타입 매개변수	actual type parameter	String	아이템 26
제네릭 타입	generic type	List<E>	아이템 26, 29
정규 타입 매개변수	formal type parameter	E	아이템 26
비한정적 와일드카드 타입	unbounded wildcard type	List<?>	아이템 26
로 타입	raw type	List	아이템 26
한정적 타입 매개변수	bounded type parameter	<E extends Number>	아이템 29
재귀적 타입 한정	recursive type bound	<T extends Comparable<T>>	아이템 30
한정적 와일드카드 타입	bounded wildcard type	List<? extends Number>	아이템 31
제네릭 메서드	generic method	static <E> List<E> asList(E[] a)	아이템 30
타입 토큰	type token	String.class	아이템 33

▼ [아이템 26] 로 타입은 사용하지 말라

📕 **제네릭 클래스, 제네릭 인터페이스**

선언에 타입 매개변수(type parameter)가 쓰인 클래스와 인터페이스.

이들을 통틀어 **제네릭 타입(generic type)**이라 한다.

제네릭 타입

1. 각각의 제네릭 타입은 일련의 매개변수화 타입(parameterized type)을 정의한다

- 먼저 클래스(혹은 인터페이스) 이름이 나오고, 이어서 꺽쇠괄호 안에 실제 타입 매개변수들을 나열한다.

2. 제네릭 타입을 하나 정의하면 그에 딸린 로 타입(raw type)도 함께 정의된다

- 로 타입은 타입 선언에서 제네릭 타입 정보가 전부 지워진 것 처럼 동작하는데, 제네릭이 도래하기 전 코드와 호환되기 하도록 위한 궁여지책이다

> 📕 **로 타입(raw type)**
>
> 제네릭 타입에서 타입 매개변수를 전혀 사용하지 않을 때의 타입
>
> ex) List list = new ArrayList();

로 타입의 위험성

1. 로 타입을 쓰면 제네릭이 안겨주는 안전성과 표현력을 모두 잃게 된다

- 로 타입은 단지 호환성 때문에 만들어졌다
- 단, List<Object>와 같이 모든 타입을 허용한다는 의사를 컴파일러에 명확히 전달하는 방식은 괜찮다

2. 실제 타입 매개변수를 신경쓰지 않으려면 비한정적 와일드카드 타입을 사용하자

- Set<E>의 비한정적 와일드카드 타입은 Set<?>이다

> 📕 **비한정적 와일드카드 타입(unbounded wildcard type)**
>
> 어떤 타입이라도 담을 수 있는 가장 범용적인 매개변수화 타입이다.
>
> Collection<?>에는 (null 외에는) 어떤 원소도 넣을 수 없다.

로 타입을 써야하는 예외 상황

1. class 리터럴에는 로 타입을 써야 한다

- 자바 명세는 class 리터럴에 매개변수화 타입을 사용하지 못하게 했다(배열과 기본 타입은 허용한다)
    - List.class, String[].class, int.class는 허용했으나, List<String>.class, List<?>.class는 허용하지 않는다

2. instanceof 연산자는 비한정적 와일드카드 타입 이외의 매개변수화 타입에는 적용할 수 없다

- 그러나 이 때의 `<?>`는 아무런 역할 없이 코드를 지저분하게 하므로 로 타입을 쓰는 편이 깔끔하다
    - 예시

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

▼ [아이템 27] 비검사 경고를 제거하라

비검사 경고

1. 할수 있는 한 모든 비검사 경고를 제거하라

- 모두 제거한다면 그 코드는 타입 안정성이 보장된다
    - 런타임에 ClassCastException이 발생할 일이 없다

2. 경고를 제거할 수는 없지만 타입 안전하다고 확신한다면 @SuppressWarnings(”unchecked”) 어노테이션을 달아 경고를 숨기자

- 단, 타입 안전함을 검증하지 않은 채 경고를 숨기면 스스로에게 잘못된 보안 인식을 심어주는 꼴이다
- 한편, 안전하다고 검증된 비검사 경고를 그대로 두면, 진짜 문제를 알리는 새로운 경고가 나와도 눈치채지 못할 수 있다

2-1. @SuppressWarnings 어노테이션은 항상 가능한 한 좁은 범위에 적용하자

- 보통은 변수 선언, 아주 짧은 메소드, 혹은 생성자가 될 것이다
- **절대로 클래스 전체에 선언해서는 안된다!**
- 한 줄이 넘는 메소드나 생성자에 달린 어노테이션을 발견하면 지역변수 선언쪽으로 옮기자

2-2. @SuppressWarnings 어노테이션을 사용할 때면 그 경고를 무시해도 안전한 이유를 항상 주석으로 남겨야 한다

- 다른 사람이 그 코드를 이해하는 데 도움이 된다
- 다른 사람이 그 코드를 잘못 수정하여 타입 안전성을 잃는 상황을 줄여준다

▼ [아이템 28] 배열보다는 리스트를 사용하라

배열과 리스트의 차이

1. 배열은 공변(covariant; 共變)이며, 제네릭은 불공변(invariant; 不共變)이다.

- 배열의 경우, Super의 하위 타입 Sub에 대해 Sub[]은 Super[]의 하위 타입이다
- 그러나 제네릭의 경우, List<Sub>은 List<Super>의 하위 타입이 아니다

```
Object[] objectArray = new Long[1];
objectArray[0] = "타입이 달라 넣을 수 없다" //ArrayStoreExce
```

```
List<Object> ol = new ArrayList<Long>(); //호환되지 않는 티
ol.add("타입이 달라 넣을 수 없다");
```

2. 배열은 실체화(relify)된다

- 배열은 런타임에도 자신이 담기로 한 원소의 타입을 인지하고 확인한다
  - 그래서 위 코드에서 Long타입 배열에 String을 넣으려 하면 ArrayStoreException이 발생한다
- 반면, 제네릭은 타입 정보가 런타임에는 소거(erasure)된다.
  - 원소 타입을 컴파일타임에만 검사하며, 런타임에는 알 수조차 없다

⚠ 위 두 차이로 배열은 제네릭 타입, 매개변수화 타입, 타입 매개변수로 사용할 수 없다. 즉, 코드를 `new List<E>[]`, `new List<String>[]`, `new E[]` 식으로 작성하면 제네릭 배열 생성 오류를 일으킨다.

이는 타입 안전하지 않기 때문이며, 만약 허용한다면 컴파일러가 자동 생성한 형변환 코드에서 런타임에 ClassCastException이 발생할 수 있다. 런타임에 해당 Exception을 막아주겠다는 제네릭 타입 시스템의 취지에 어긋나는 것이다.

이를 방지하려면, 배열 대신 제네릭을 쓰면 된다.

▼ [아이템 29] 이왕이면 제네릭 타입으로 만들라

예시: 스택 구현

배열을 사용한 코드를 제네릭으로 만드는 방법 1

```
//배열 elements는 push(E)로 넘어온 E 인스턴스만 담는다.
//따라서 타입 안전성을 보장하지만, 이 배열의 런타임 타입은 E[]가 아닌 O
@SuppressWarnings("unchecked")
public Stack() {
        elements = (E[]) new Object[DEFAULT_INITIAL_CAPACIT
}
```

- 가독성이 좋다
- 코드가 짧다
- 현업에서는 이 방식을 더 선호한다
- 하지만 (E가 Obejct가 아닌 한) 배열의 런타임 타입이 컴파일 타입과 달라 힙 오염(heap pollution; 아이템 32)을 일으킨다
    - 위 예시에서는 해가 되지 않으나, 이가 걸리는 프로그래머는 두 번째 방식을 고수하기도 한다

배열을 사용한 코드를 제네릭으로 만드는 방법 2

```
//비검사 경고를 적절히 숨긴다
public E pop() {
        if (size == 0)
                throw new EmptyStackException();

        //push에서 E 타입만 허용하므로 이 형변환은 안전하다.
        @SuppressWarnings("unchecked")
        E result = (E) elements[--size];

        elements[size] = null; // 다 쓴 참조 해제
        return result;
}
```

타입 매개변수에 제약을 두는 제네릭 타입

한정적 타입 매개변수(bounded type parameter)

- java.util.concurrent.DelayQueue는 다음과 같이 선언되어 있다.

  `class DelayQueue<E extends Delayed> implements BlockingQueue<E>`

  이러한 경우에는 클라이언트가 DelayQueue의 원소에서 (형변환 없이) 곧바로 Delayed 클래스의 메소드를 호출할 수 있다.

☝🏻 클라이언트에서 직접 형변환해야 하는 타입보다 제네릭 타입이 더 안전하고 쓰기 편하다. 그러므로 새로운 타입을 설계할 때는 형변환 없이도 사용할 수 있도록 하라.

기존 타입 중 제네릭이어야 하는 게 있다면 제네릭 타입으로 변경하자. 기존 클라이언트에는 아무 영향을 주지 않으면서 새로운 사용자를 훨씬 편하게 해주는 길이다(아이템 26)

▼ [아이템 30] 이왕이면 제네릭 메소드로 만들라

☝🏻 매개변수화 타입을 받는 정적 유틸리티 메소드는 보통 제네릭이다

- 예시: Collections의 알고리즘 메소드(binarySearch, sort 등)

제네릭 메소드

(타입 매개변수를 선언하는) 타입 매개변수 목록은 메소드 제한자와 반환 타입 사이에 온다

- 다음 코드에서 타입 매개변수 목록은 <E>이고, 반환 타입은 Set<E>이다
  - 타입 매개변수의 명명 규칙은 제네릭 메소드나 제네릭 타입이나 똑같다(아이템 29, 68)

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

```
//내 이해를 위한 예시
public static <E, K extends E, D extends K> Set<Object>
    Set<Object> result = new HashSet<>(s1);
    s1.addAll(s2);
    s1.addAll(s3);
    return result;
}
```

불변 객체를 여러 타입으로 활용할 때에는 그 객체의 타입을 바꿔주는 정적 팩토리를 활용해라

- 제네릭 싱글톤 패턴이라 하며, Collections.reverseOrder와 같은 함수 객체(아이템 42)나 Collections.emptySet 같은 컬렉션용으로 사용한다.

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

@SuppressWarnings("unchecked")
public static <T> UnaryOperator<T> identityFunction() {
```

```
        return (UnaryOperator<T>) IDENTITY_FN;
}
```

```
public static void main(String[] args){
        //예시1
        String[] strings = {"삼베", "대마", "나일론"};
        UnaryOperator<String> sameString = identityFunct
        for (String s : strings)
                System.out.println(sameString.apply(s));

        //예시2
        Number[] numbers = {1, 2.0, 3L};
        UnaryOperator<Number> sameNumber = identityFunct
        for (Number n : numbers)
                System.out.println(sameNumber.apply(n));
}
```

재귀적 타입 한정(recursive type bound)

자기 자신이 들어간 표현식을 사용해 타입 매개변수의 허용 범위를 한정할 수 있다

- Comparable 인터페이스(아이템 14)와 함께 쓰인다

```
public interface Comparable<T> {
        int compareTo(T o);
}
```

- Comparable을 구현한 원소의 컬렉션을 입력받는 메소드들은 주로 그 원소들을 정렬/검색하거나 최솟값/최댓값을 구하는 식으로 사용된다.
    - 이 기능을 수행하려면 컬렉션에 담긴 모든 원소가 상호 비교될 수 있어야 한다. 아래는 그 제약을 코드로 표현한 모습이다

      ```
      public static <E extends Comparable<E>> E max(Collect
      ```

    - 아래는 위에서 선언한 메소드의 구현이다

```
public static <E extends Comparable<E>> E max(Collect
        if (c.isEmpty())
                throw new IllegalArgumentException("
        E result = null;
        for (E e : c)
                if (result == null || e.compareTo(res
                        result = Objects.requireNonNu
        return result;
}
```

☝🏻 이 메소드에 빈 컬렉션을 건네면 IllegalArgumentException을 던지니, Optional<E>를 반환하도록 고치는 편이 나을 것이다(아이템 55)

▼ [아이템 31] 한정적 와일드카드를 사용해 API 유연성을 높이라

때론 불공변 방식보다 유연한 무언가가 필요하다

```
public class Stack<E>{
        public Stack();
        public void push(E e);
        public E pop();
        public boolean isEmpty();
}
```

- 여기에 일련의 원소를 스택에 넣는 메소드를 추가해야한다고 해보자

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

- Iterable src의 원소 타입이 스택의 원소 타입과 일치하면 잘 작동한다. 그러나 Stack<Number>로 선언한 후 pushAll(intVal) - intVal은 Integer 타입이다 - 을 호출하면 오류가 발생한다
    - 매개변수화 타입이 불공변이기 때문이다
- 이러한 경우를 대비해 자바는 한정적 와일드카드 타입이라는 특별한 매개변수화 타입을 지원한다

한정적 와일드카드 타입

- 위 예시의 pushAll의 입력 매개변수 타입은 'E의 Iterable'이 아니라 'E의 하위 타입의 Iterable'이어야 한다
- 이 때, 한정적 와일드카드 타입 `Iterable<? extends E>`를 사용한다

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

- 마찬가지로, 'E의 상위 타입의 콜렉션'을 원할 때에는 `Collection<? super E>`를 사용한다

유연성을 극대화하려면 원소의 생산자나 소비자용 입력 매개변수에 와일드카드 타입을 사용하라

- 단, 입력 매개변수가 생산자와 소비자 역할을 동시에 한다면 와일드카드 타입을 쓰는게 좋을건 없다
    - 타입을 확실히 지정해야 하는 상황이기 때문이다

☝🏻 **와일드카드 타입을 써야하는 상황 공식**

펙스(PECS): producer-extends, consumer-super

즉, 매개변수화 타입 T가 생산자라면 <? extends T>를, 소비자라면 <? super T>를 사용하라

- Comparable, Comparator는 항상 소비자이므로, Comparable<E>보다는 Comparable<? super E>를 쓰자

반환 타입에는 한정적 와일드카드 타입을 사용하면 안된다

- 클래스 사용자가 와일드카드 타입을 신경써야 한다면 그 API에 무슨 문제가 있을 가능성이 크다

메소드 선언에 타입 매개변수가 한 번만 나오면 와일드카드로 대체하라

- 비한정적 타입 매개변수라면 비한정적 와일드카드로, 한정적 타입 매개변수라면 한정적 와일드카드로 바꿔라

▼ [아이템 32] 제네릭과 가변인수를 함께 쓸 때는 신중하라

가변인수(varargs)

- 가변인수 메소드(아이템 53)와 제네릭은 자바 5때 함께 추가되었으나, 서로 어우러지지 않는다
- 가변인수는 메소드에 넘기는 인수의 개수를 클라이언트가 조절할 수 있게 해주는데, 구현 방식에 허점이 있다
    - 내부적으로 가변인수를 담기 위한 배열이 생성되는데, 이 배열을 클라이언트에 노출하는 문제가 생겼다
    - 그 결과로 varargs 매개변수에 제네릭이나 매개변수화 타입이 포함되면 알기 어려운 컴파일 경고가 발생한다

제네릭 varargs 배열 매개변수에 값을 저장하는 것은 안전하지 않다

```
static void dangerous(List<Stirng>... stringLists){
        List<Integer> intList = List.of(42);
        Object[] objects = stringLists;
        objects[0] = intList;             //힙 오염 발생
        String s = stirngLists[0].get(0)  //ClassCastExcept
}
```

@SafeVarargs 어노테이션

- 메소드 작성자가 그 메소드가 타입 안전함을 보장하는 장치이다
    - 메소드가 안전한게 확실하지 않다면 절대 사용하지 말자

- 제네릭이나 매개변수화 타입의 varargs 매개변수를 받는 모든 메소드에 @SafeVarargs를 달아라
- 재정의할 수 없는 메소드에만 달아라
    - 자바 8에서는 오직 static, final 메소드에만 붙일 수 있고, 자바 9부터는 private에도 허용된다

> ☝🏻 **안전한 제네릭 varargs 메소드의 기준**
>
> - varargs 매개변수 배열에 아무것도 저장하지 않는다
> - 그 배열(혹은 복제본)을 신뢰할 수 없는 코드에 노출하지 않는다

제네릭 varargs 매개변수 배열에 다른 메소드가 접근하도록 허용하면 안전하지 않다

- 단, @SafeVarargs로 제대로 어노테이트된 또 다른 varargs 메소드에 넘기는 것은 안전하다
- 또, 그저 이 배열 내용의 일부 함수를 호출만 하는(varargs를 받지 않는) 일반 메소드에 넘기는 것도 안전하다

▼ [아이템 33] 타입 안전 이종 컨테이너를 고려하라

타입 안전 이종 컨테이너 패턴

- API

```
public class Favorites {
        public <T> void putFavorite(Class<T> type, T ins
        public <T> T getFavorite(Class<T> type);
}
```

- 클라이언트

```
public static void main(String[] args){
        Favorites f = new Favorites();
```

```
        f.putFavorite(String.class, "Java");
        f.putFavorite(Integer.class, 0xcafebabe);
        f.putFavorite(Class.class, Favorites.class);

        String favoriteString = f.getFavorite(String.cla
        int favoriteInteger = f.getFavorite(Integer.clas
        Class<?> favoriteClass = f.getFavorite(Class.cla

        System.out.printf("%s %x %s%n", favoriteString,
}
```

- 구현

```
public class Favorites{
        private Map<Class<?>, Object> favorites = new Ha

        public <T> void putFaborite(Class<T> type, T ins
                favorites.put(Objects.requireNonNull(typ
        }

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

  - Class의 cast 메소드는 동적 형변환을 한다