Ch5. 제네릭

태그 2주차

Item26 로 타입은 사용하지 마라

- 로 타입: List, Set과 같이 어떤 매개변수를 담을지 선언하지 않은 타입

로 타입은 이전 자바 버전에서 사용하던 것으로 지금은 절대 사용하면 안되며, 이전 코드들과의 호환을 위해 존재하는 것이다.

```
int sum(Collection c) {
    int sum = 0;
    Iterator i = c.iterator();
    for (k = 0; k < c.size(); k++) {
        sum += Integer.parseInt(i.next());
    }
    return sum;
}
```

→ 컴파일 시점에 에러를 찾을 수 없음. 모든 타입을 담을 수 있으므로 런타임시 캐스팅 오류 등이 발생할 확률이 높음.

- **비한정 와일드카드 타입(unbounded wildcard type)** : 제네릭 타입을 쓰고 싶지만, 실제 타입 매개변수가 무엇인지 신경쓰고 싶지 않을 때 사용한다. `Set<E>`의 비한정 와일드카드 타입은 `Set<?>`

```
void printCollection(Collection<?> c) {
    for (Object e : c) {
        System.out.println(e);
    }
}

void genericTest() {
    Collection<?> c = new ArrayList<String>();
```

```
    c.add(new Object()); // 컴파일 에러
}
```

→ 요소를 추가하는 연산을 허용하면 로 타입과 다를게 없으므로 이를 허용하지 않는다.

만약 모든 요소를 추가하는 연산도 허용하고 싶으면 Object 타입을 이용하자 (Set<Object>)

용어정리

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

⭐핵심정리

💡 로 타입을 사용하면 런타임에 예외가 일어날 수 있으니 사용하면 안된다. 단지 이전 코드들과 호환성을 위해 남겨둔 것이다. 모든 요소 추가가 가능하게 사용하려면 매개변수로 Object타입을 넣고, 타입을 신경쓰기 싫다면 와일드카드를 사용하자.

Item27 비검사 경고를 제거하라

```
Set<Lark> ecaltation = new HashSet(); // 컴파일 에러
Set<Lark> ecaltation = new HashSet<>();
```

다이아몬드 연산자를 통해 직접 타입을 명시하지 않아도 컴파일러에게 추론시켜 컴파일 에러 없애줄 수 있음

제네릭에서는 원소의 타입정보가 소거되어 런타임에는 무슨 타입인지 알 수 없다!!

제네릭을 사용하기 시작했을 때 볼 수 있는 수많은 컴파일러 경고

- 비검사 형변환 경고
- 비검사 메서드 호출 경고
- 비검사 매개변수화 가변인수 타입 경고
- 비검사 변환 경고

→ 타입을 명시하지 않았을 때 나오는 경고

⭐핵심정리

> 💡 비검사 경고는 중요하니 무시하지 말자. 모든 비검사 경고는 런타임에 ClassCaseException을 일으킬 수 있는 가능성을 뜻하니 최선을 다해 제거하자. 단, 경고를 없앨 방법이 없고 타입이 안전함을 보장하면 @SuppressWarnings("unchecked") 에너테이션으로 경고를 숨기고 주석을 남겨 근거를 남겨두자

Item28 배열보다는 리스트를 사용하라

배열과 제네릭의 차이

배열

- 공변 (convariant) - `Sub`가 `Super`의 하위타입이라면 배열 `Sub[]`는 배열 `Super[]`의 하위타입이다. (함께 변한다)
- 배열에서는 실수를 런타임에 타입 오류를 알 수 있다

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

실체화(reify)된다 : 런타임에도 자신이 담기로 한 원소의 타입을 인지하고 확인한다.

제네릭

- 불공변 (invariant) - 서로 다른 타입 `Type1`, `Type2`가 있을 때 `List<Type1>` 은 `List<Type2>` 의 하위타입도 아니고 상위타입도 아니다
- 리스트에서는 컴파일할 때 타입 오류를 바로 알 수 있다.

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

소거(erasure)된다 : 원소 타입을 컴파일 타임에만 검사하며 런타임에는 알 수 없다.

제네릭 지원 전과 제네릭 타입을 함께 사용할 수 있게 해주는 메커니즘이다.

에러는 컴파일 시점에 알아차리는게 가장 좋다!!

⭐핵심정리

> 💡 배열과 제네릭에는 매우 다른 타입 규칙이 적용된다. 배열은 공변이고 실제화 되는 방면 제네릭은 불공변이며, 런타임 시점에 타입 정보가 소거된다. 그 결과 배열은 컴파일 시점에 안전하고, 런타임때 에러가 발생하고 제네릭은 그 반대다. 따라서 둘을 섞어쓰다 오류가 발생하면 가장 먼저 배열을 리스트로 대체하는 방법을 적용해보자.

Item29 이왕이면 제네릭 타입으로 만들라

제네릭 타입으로 만들 수 있는 두 가지 방법

1. Object 배열을 생성한 다음 제네릭 배열로 형변환

```
elements = (E[]) new Object[DEFAULT_INITIAL_CAPACITY];
```

오류는 없겠지만, 경고를 내보낼 것이고 타입 안전하지 않다.

따라서 타입 안전성을 확인해보자. 문제의 배열 elements는 private 필드에 저장되고, 클라이언트로 반환되거나 다른 메서드에 전달되는 일이 전혀 없다. push 메서드를 통해 배열에 저장되는 원소의 타입은 항상 E다. 따라서 이 비검사 형변환은 안전하다.

안전을 확인했으므로 `@SuppressWarning` 애너테이션으로 경고를 숨긴다.

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

2. elements 필드의 타입을 `E[]`에서 `Object[]`로 바꾸는 것이다. 생성자에서 `Object[]`로 선언하고, `pop()`에서 형변환을 한다.

```
elements = new Object[DEFAULT_INITIAL_CAPACITY];
```

또한 이를 통한 경고는 아래와 같이 숨길 수 있을 것이다.

```
    // 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;
 }
```

둘의 차이는 무엇일까?

첫번째 방법

- 가독성이 더 좋다.
  - 배열의 타입을 `E[]`로 선언해 오직 `E` 타입 인스턴스만 받음을 확실히 어필
  - 더 짧음
- 형변환 한번 - 배열 생성시
- 현업에서 선호
- 배열의 런타임 타입이 컴파일타임 타입과 달라 힙오염발생 위험 있음 → 컴파일 타임엔 타입이 E이나, 런타임시엔 타입이 `Object[]`이다.

```
    public Stack() {
     String simpleName = this.elements.getClass().getSim
pleName();
     System.out.println("element's class = " + simpleNam
e);
 }
```

두번째 방법

- 배열에서 원소를 읽을 때 마다 형변환

대다수의 제네릭 타입은 타입 매개변수에 아무런 제약을 두지 않는다. 따라서 어떤 참조 타입으로도 Stack을 만들 수 있지만, **기본 타입**은 사용할 수 없다.

O : `Stack<Object>, Stack<int[]>, Stack<List<String>>`

X : `Stack<int>, Stack<double>`

⭐핵심 정리

> 💡 제네릭 타입이 클라이언트가 형변환하는 것 보다 안전하고 쓰기 편하다. 기존 타입 중 제네릭이었어야 하는게 있다면(형변환 해줘야 하는 타입) 제네릭 타입으로 변경하자. 기존 클라이언트에 아무 영향을 주지 않으면서 새로운 사용자를 훨씬 편하게 해주는 길이다!

Item30 이왕이면 제네릭 메서드로 만들라

제네릭 메서드 만들기

메서드도 제네릭으로 만들 수 있다. ex) Collections의 '알고리즘' 메서드

- 메서드 선언에서 원소타입을 타입 매개변수로 지정한다.
- 메서드 안에서 이 타입 매개변수를 사용하게 수정한다.
- 타입 매개변수 목록은 메서드의 제한자와 반환타입 사이에 온다.
- 한정적 와일드 카드 타입을 사용하면, 반환타입 입력타입 등을 좀더 유연하게 개선할 수 있다.

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

제네릭 싱글톤 팩토리

제네릭으로 타입설정 가능한 인스턴스를 만들어두고, 반환 시에 제네릭으로 받은 타입을 이용해 타입을 결정하는 것이다.

```
public class GenericFactoryMethod {
public static final Set EMPTY_SET = new HashSet(); //싱글톤
인스턴스

    public static final <T> Set<T> emptySet() {
        return (Set<T>) EMPTY_SET;
    }
}

@Test
public void genericTest() {
    Set<String> set = GenericFactoryMethod.emptySet();
    Set<Integer> set2 = GenericFactoryMethod.emptySet();
```

```
    Set<Elvis> set3 = GenericFactoryMethod.emptySet();

    set.add("ab");
    set2.add(123);
    set3.add(Elvis.INSTANCE);

    String s = set.toString();
    System.out.println("s = " + s);
}
```

```
s = [ab, item3.Elvis@3439f68d, 123]
```

위와 같이 여러 타입으로 내부 객체를 받아도 에러가 나지 않는다.

큰 유연성을 제공한다.

재귀적 타입 한정

자기 자신이 들어간 표현식을 사용하여 타입 매개변수의 허용범위를 한정한다.

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

위 제네릭 코드는 모든 타입 E에 대해 자기 자신과 같은 타입인 원소 모두와 비교 가능하다를 뜻한다.

⭐핵심정리

> 💡 제네릭 타입과 마찬가지로, 클라이언트에서 입력 매개변수와 반환값을 명시적으로 형변환해야 하는 메서드보다 제네릭 메서드가 더 안전하며 사용하기도 쉽다. 타입과 마찬가지로, 메서드도 형변환 없이 사용할 수 있는 편이 좋으며, 많은 경우 그렇게 하려면 제네릭 메서드가 되어야한다!

Item31 한정적 와일드카드를 사용해 API 유연성을 높이라

매개변수화 타입의 불공변

매개변수화 타입은 불공변이다(invariant) : 서로 다른 타입 `Type1`, `Type2`가 있을 때 `List<Type1>` 은 `List<Type2>` 의 하위타입도 아니고 상위타입도 아니다. (리스코프 치환 원칙에 어긋난다.)

이때 불공변 방식보다 유연한 방식 : **한정적 와일드카드 타입**

한정적 와일드 카드 타입을 이용한 확장

유연성 극대화를 위해 원소의 생산자나 소비자용 입력 매개변수에 왈일드카드 타입을 사용하라.

입력 매개변수가 생산자와 소비자 역할을 동시에 한다면 와일드 카드 타입을 써도 좋을 게 없다.

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

```
public void popAll(Collection<? super E> dst) {// 소비자
  dst.addAll(list);
  list.clear();
}
```

PECS : producer-extends, consumer-super (= Get and Put Princlple)

매개변수화 타입 T가 생산자면 <? extends T>를 사용하고 소비자라면 <? super T>를 사용하라

반환타입에서의 한정적 와일드 카드 타입

반환타입에는 한정적 와일드 카드 타입을 사용하면 안된다

유연성을 높여주지 않고 클라이언트 코드에서도 와일드 카드 타입을 사용하게 하기 때문이다.

`public static <E> Set<E> union(Set<? extends E> s1, Set<? extends E> s2)`

클래스 사용자가 와일드 카드 타입을 신경써야 한다면 그 API에 문제가 있을 가능성이 크다

⭐핵심정리

> 💡 널리 쓰일 라이브러리라면 와일드 카드 타입을 적절히 사용해줘야 한다..
> PECS를 기억하자.. 생산자 extends 소비자 super
> Comparable과 Comparator는 모두 소비자

Item32 제네릭과 가변인수를 함께 쓸 때는 신중하라

가변인수와 제네릭을 함께 사용할 때의 헛점

가변인수 메서드를 호출하면 가변인수를 담기위한 배열이 자동으로 하나 만들어진다.

내부로 감춰야했을 배열을 클라이언트에 노출해서 문제가 생겼다.

> 제네릭타입의 가변인수 메서드를 호출하면, 제네릭 타입의 배열이 생성되며, 제네릭 타입의 배열은 item28에서 말한 것 처럼, 타입을 런타임에 체크하기 때문에 클래스 캐시팅 에러가 날 가능성이 있다.

메서드 선언시, 실체화 불가 타입으로 varargs 매개변수를 선언하면 컴파일러가 경고를 보낸다.

제네릭과 varages를 혼용하면 **타입 안정성**이 깨진다. 따라서 제네릭 varargs 배열 매개변수에 값을 저장하는 것은 안전하지 않다.

```
static void dangerous(List<String>...stringLists){
  List<Integer> intList = List.of(42);
  Object[] objects = stringLists;
  objects[0] = intList;// 힙오염 발생String s = stringLists
[0].get(0)// ClassCastException
}
```

이런 위험에도 varargs 매개변수를 받으면 메서드가 실무에서 매우 유용하다.

`Arrays.asList(T... a)`, `Collections.addAll(Collection<? superT> c, T... elements)`, `EnumSet.of(E first, E... rest)`

제네릭 varargs 매개변수 배열에 다른 메서드가 접근하도록 허용하지 말자

자신의 제네릭 매개변수 배열의 참조를 노출하므로 안전하지 않다.

- 힙 오염을 메서드를 호출한 쪽의 콜스택으로까지 전이하는 결과를 낳는다.

```
static <T> T[] toArray(T... args){
  return args;// 참조가 밖으로
}
```

⭐핵심정리

💡 가변인수와 제네릭은 궁합이 좋지 않다. 가변인수 기능은 배열을 노출하여 추상화가 완벽하지 못하고, 배열과 제네릭의 타입 규칙이 서로 다르기 때문이다. 제네릭 varargs(...) 매개변수는 타입 안전하지 않지만 허용된다. varargs 매개변수를 사용하고자 한다면, 먼제 타입 안전한지 확인 이후 @SafeVarargs 어노테이션과 함께 사용하자.

Item33 타입 안전 이종 컨테이너를 고려하라

⭐핵심정리

💡 컬렉션 API로 대표되는 일반적인 제네릭 형태에서는 한 컨테이너가 다룰 수 있는 타입 매개변수의 수가 고정되어 있다.

컨테이너 자체가 아닌 키를 타입 매개변수로 바꾸면 이런 제약이 없는 타입 안전 이종 컨테이너를 만들 수 있다.

타입 안전 이종 컨테이너는 Class를 키로 쓰며, 이런 식으로 쓰이는 Class 객체를 타입 토큰이라한다.