

이펙티브 자바 CP.4

🕒 작성 일시	@2023년 1월 27일 오후 7:25
🕒 최종 편집 일시	@2023년 2월 4일 오후 11:48
📄 유형	이펙티브 자바
👤 작성자	
👥 참석자	

4 제네릭

선행 내용 (inpa님 블로그)

- 26. raw(로) 타입은 사용하지 말라
- 27. 비검사 경고를 제거하라
- 28. 배열보다는 리스트를 사용하라
- 29. 이왕이면 제네릭 타입으로 만들라
- 30. 이왕이면 제네릭 메서드로 만들라
- 31. 한정적 와일드카드를 사용해 API 유연성을 높이라
- 32. 제네릭과 가변인수를 함께 쓸 때는 신중하라
- 33. 타입 안정 이중 컨테이너를 고려하라

4 제네릭

▼ 선행 내용 (inpa님 블로그)

- [\[JAVA\] 제네릭\(Generics\) 개념 & 문법 정복하기](#)
- [\[JAVA\] 제네릭 - 공변성 & 와일드카드 완벽 이해하기](#)
- [\[JAVA\] 제네릭 - 힙 오염 \(Heap Pollution\) 이란?](#)
- [\[JAVA\] 제네릭 타입 소거 컴파일 과정 알아보기](#)
- 용어 정리
 - 클래스와 인터페이스 선언에 타입 매개변수가 쓰이면, 이를 **제네릭 클래스** 혹은 **제네릭 인터페이스**라 부른다.
제네릭 클래스와 제네릭 인터페이스를 통틀어 **제네릭 타입**이라고 부른다.
 - 제네릭 타입은 일련의 **매개변수화 타입**을 정의한다.
 - `List<String>` 은 원소의 타입이 `String` 인 리스트를 뜻하는 매개변수화 타입이다.
 - 여기서 `String` 이 정규 타입 매개변수 `E` 에 해당하는 **실제 타입 매개변수**이다.
 - 제네릭 타입을 하나 정의하면 그에 딸린 **로 타입** 도 함께 정의 된다.

- 로 타입이란 제네릭 타입에서 타입 매개변수를 전혀 사용하지 않을 때를 말한다.

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

▼ 26. raw(로) 타입은 사용하지 말라

- raw 타입 (제네릭을 지원하지전) (jdk 1.5 이전)
 - 로 타입은 타입 선언에서 제네릭 타입 정보가 전부 지워진 것처럼 동작하는데, 제네릭이 도래하기 전 코드와 호환되도록 하기 위한 궁여지책(야매)이라 할 수 있다.
 - raw 타입 컬렉션

```
// raw 타입
private final Collection stamps = ...;

stamps.add(new Coin(...));
// 아무런 오류 없이 컴파일되고 실행되고, 컴파일러가 "unchecked call" 경고를 내뱉는다.

for (Iterator i = stamps.iterator(); i.hasNext(); ) {
    Stamp stamp = (Stamp) i.next(); // ClassCastException 을 던진다.
    ...
}
// 위 에서 element 를 꺼내기 전에는 오류를 알지 못한다.
// 즉, 해당 코드의 오류를 런타임시에 알 수 있는 것이다.
```

- 매개변수화 타입 컬렉션

```
private final Collection<Stamp> stamps = ...;
```

- 컴파일러는 `stamps` 에는 `Stamp` 의 인스턴스만 넣어야 함을 컴파일러가 인지하게 된다. 따라서 아무런 경고 없이 컴파일된다면 의도대로 동작할 것임을 보장한다.
- `stamps` 에 엉뚱한 타입의 인스턴스를 넣으려 하면 컴파일 오류가 발생하며 무엇이 잘못 되었는지 알려준다.
- 컴파일러는 컬렉션에서 원소를 꺼내는 모든 곳에 보이지 않는 형변환을 추가하여 절대 실패하지 않음을 보장한다. (컴파일러 경고를 숨기지 않음 - 아이템 27)
- raw 타입을 사용하게 되면 제네릭이 안겨주는 안정성과 표현력을 모두 잃게 된다.
 - raw 타입이 있는 이유는 호환성 때문이다. (제네릭 없이 짠 코드 에서 기존 코드를 모두 수용하면서, 제네릭을 맞물려 돌아가게 하기 위해서는 raw 타입을 사용하는 메서드에 매개변수와 타입의 인스턴스를 (그 반대로) 넘겨도 동작해야만 했다.)
 - 마이그레이션 호환성을 위해 raw 타입을 지원하고 제네릭 구현에는 소거 (아이템28) 방식을 사용하기로 했다.
- `List`, `List<Object>` 의 차이
 - `List` 는 제네릭 타입에서 완전히 발 뻗 것을 의미.
 - `List<Object>` 는 모든 타입을 허용한다는 의사를 컴파일러에 명확히 전달한 것이다.
 - 매개변수로 `List` 를 받는 메서드에 `List<String>` 을 넘길 수 있지만, `List<Object>` 를 받는 메서드에는 넘길 수 없다. (제네릭의 하위 타입 규칙)
 - `List<String>` 은 raw 타입인 `List` 의 하위 타입이지만, `List<Object>` 의 하위 타입이 아니다. (아이템 28)
 - `List<Object>` 같은 매개변수와 타입을 사용할 때와 달리 `List` 같은 raw 타입을 사용하면 타입 안전성을 잃게 된다.
 - 런타임 실패

```
public static void main(String [] args) {
    List<String> strings = new ArrayList<>();
    unsafeAdd(strings, Integer.valueOf(42));
    String s = strings.get(0); // 컴파일러가 자동으로 형변환 코드를 넣어줌
}
private static void unsafeAdd(List list, Object o) {
    list.add(o);
}
```

- 위 코드는 컴파일 되지만 raw 타입인 `List` 를 사용해 경고가 발생된다.
 - `strings.get(0)` 의 결과를 형변환시 `ClassCastException` 을 던진다.
 - `Integer` 를 `String` 으로 변환하려 시도 함.
- 컴파일 실패

```
private static void unsafeAdd(List<Object> list, Object o) {
    list.add(o);
}
```

- 제네릭으로 변환시 - 오류 메시지를 출력되며 컴파일 조차 되지 않는다.
 - 컴파일 시점에 오류를 반환해 보다 안전.
 - `List<?>`, `List` 의 차이
 - 와일드카드 타입은 안전하고, raw 타입은 안전하지 않다.
 - raw 타입 컬렉션에는 아무 원소를 넣을 수 있으니 타입 불변식을 훼손하기 쉽다.
 - 반면, 와일드카드 타입은 `null` 을 제외하고는 어떤 원소도 넣을 수 없다.
 - 와일드카드 타입을 사용하면 컴파일러가 제 역할을 할 수 있으며, 컬렉션의 불변식을 훼손하지 않게 막을 수 있다.
- raw 타입을 사용 해야 하는 상황
 - class 리터럴에는 raw 타입을 써야 한다.
 - 자바 명세는 class 리터럴에 매개변수화 타입을 사용하지 못하게 했다.
 - `List.class`, `String[].class`, `int.class` 허용
 - `List<String>.class`, `List<?>.class` 허용 하지 않음.
 - `instanceof` 연산자
 - 런타임에는 제네릭 타입 정보가 지워지므로 `instanceof` 연산자는 비한정적 타입 이외의 매개변수화 타입에는 적용 할 수 없다.
 - raw 타입이든 비한정적 와일드카드 타입이든 `instanceof` 는 완전히 똑같이 동작한다.
- 정리
 - raw 타입으로 사용하면 런타임에서 예외가 날 수 있으니, 사용하면 안된다.
raw 타입은 제네릭이 도입되기 이전 코드와 호환성을 위해 제공될 뿐이다.
 - `Set<Object>` 는 어떤 타입의 객체도 저장할 수 있는 매개변수화 타입이다.
`Set<?>` 는 모종의 타입 객체(`null`) 만 저장할 수 있는 와일드카드 타입이다.
그리고 이들의 raw 타입인 `Set` 은 제네릭 타입 시스템에 속하지 않는다.

▼ 27. 비검사 경고를 제거하라

- 제네릭 사용시 보이는 컴파일 경고
 - 비검사 형변환 경고
 - 비검사 메서드 호출 경고
 - 비검사 매개변수화 가변인수 타입 경고
 - 비검사 변환 경고
 - 제네릭에 익숙해질수록 마주치는 경고 수는 줄겠지만, 새로 작성된 제네릭 코드가 한 번에 깨끗하게 컴파일 되리라 기대하지는 말자.
- 컴파일 방법

- javac 명령줄 인수에 -Xlint:unchecked 옵션을 추가한다.

```
// 이 코드를 컴파일 시킴
Set<Lark> exaltation = new HashSet();

// 컴파일 내용
____.java(file):4(Line): warning(level): [unchecked] unchecked conversion
    Set<Lark> exaltation = new HashSet();
                        ^
required: Set<Lark>
found:      HashSet
```

- 컴파일러가 알려준 대로 수정한다면, 경고는 사라진다.
- 사실 컴파일러가 알려준 타입 매개변수를 명시하지 않고, 자바 7부터 지원하는 다이아몬드 연산자(<>)만으로 해결할 수 있다.
그러면 컴파일러가 올바른 실제 타입 매개변수 (코드에 경우 Lark)를 추론해줌.

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

- 제거하기 어려운 경고도 있으며, 앞으로 이번 장을 진행하면서 그런 예제를 볼 수 있다.

- **할 수 있는 한 모든 비검사 경고를 제거하라. (코드 안전성이 보장됨)**

- 경고를 제거할 수는 없지만, 타입이 안전하다고 확신할 수 있다면 `@SuppressWarnings("unchecked")` 애너테이션을 달아 경고를 숨기자.
 - 단, 타입 안전함을 검증하지 않은 채 경고를 숨기면 스스로에게 잘못된 보안 인식을 심어주는 꼴이다.
 - 경고 없이 컴파일은 되겠지만, 런타임에는 여전히 `ClassCastException`을 던질 수 있다.
- `@SuppressWarnings` 애너테이션
 - 해당 애너테이션은 개별 지역변수 선언부터 클래스 전체까지 어떤 선언에도 달 수 있다. 하지만, `@SuppressWarnings` 애너테이션은 항상 가능한 한 좁은 범위에 적용하자.
 - 변수 선언, 아주 짧은 메서드, 혹은 생성자가 될 것이다.
자칫 심각한 경고를 놓칠 수 있으니 절대로 클래스 전체로 적용해선 안 된다.
 - 한 줄이 넘는 메서드나 생성자에 달린 애너테이션을 발견하면, 지역변수 선언 쪽을 옮기자.
 - 이를 위해 지역변수를 새로 선언하는 수고를 해야 할 수 도 있지만, 그만한 값어치가 있다.
- ex) `ArrayList`의 `toArray` 메서드

```
public <T> T[] toArray(T[] a) {
    if (a.length < size)
        return (T[]) Arrays.copyOf(elements, size, a.getClass());
    System.arraycopy(elements, 0, a, 0, size);
    if (a.length > size)
        a[size] = null;
    return a;
}
```

```

}

// 컴파일시

ArrayList.java:305: warning: [unchecked] unchecked cast
    return (T[]) Arrays.copyOf(elements, size, a.getClass());
                        ^
required: T[]
found:    Object[]

```

- 애너테이션은 선언에만 달 수 있기 때문에 `return` 문에는 `@SuppressWarnings` 를 다는게 불가능하다.
메서드 전체에 달고 싶겠지만, 범위가 필요 이상으로 넓어지니 하지 않는다.
- 그 대신 반환값을 담을 지역변수를 하나 선언하고 그 변수에 애너테이션을 달아주자.
- ex) 변경된 코드

```

public <T> T[] toArray(T[] a) {
    if (a.length < size) {
        @SuppressWarnings("unchecked")
        T[] result = (T[]) Arrays.copyOf(elements, size, a.getClass());
        return result;
    }
    System.arraycopy(elements, 0, a, 0, size);
    if (a.length > size)
        a[size] = null;
    return a;
}

```

- 이 코드는 깔끔하게 컴파일되고 비검사 경고를 숨기는 범위가 최소로 좁혔다.
- `@SuppressWarnings("unchecked")` 애너테이션을 사용할 때면 경고를 무시해도 안전한 이유를 항상 주석으로 남겨야한다.
 - 코드의 이해에 도움이 되며,
다른 사람이 코드를 잘 못 수정하여 타입 안정성을 잃은 상황도 줄여준다.
- 정리
 - 비검사 경고는 중요하니 무시하지 말자.
 - 모든 비검사 경고는 런타임에 `ClassCastException` 을 일으킬 잠재적 가능성을 뜻하니 최대한 제거해라.
 - 경고를 없앨 방법을 찾지 못하겠다면, 그 코드가 타입 안전함을 증명하고 가능한 한 범위를 좁혀 `@SuppressWarnings("unchecked")` 애너테이션으로 경고를 숨기고, 근거를 주석으로 남겨라.

▼ 28. 배열보다는 리스트를 사용하라

- 배열과 제네릭의 차이
 1. 공변, 불공변

- 배열은 공변이다.
 - `Sub` 가 `Super` 의 하위 타입이라면, 배열 `Sub[]` 는 배열 `Super[]` 의 하위 타입이 된다.
- 제네릭은 불공변이다.
 - 서로 다른 타입 `Type1`, `Type2` 가 있을 때, `List<Type1>` 은 `List<Type2>` 의 하위 타입도 아니고 상위 타입도 아니다.
- Ex) `Object` 배열, 재네릭

```
Object[] objectArray = new Long[1];
objectArray[0] = "문자열"; // ArrayStoreException 을 던진다. (런타임시!!!)

List<Object> ol = new ArrayList<Long>(); // 호환되지 않는다. (컴파일 시)
ol.add("문자열");
```

- 어느 쪽이든 `Long` 용 저장소에 `String` 값을 넣을 수 없다.
다만 배열에서는 그 실수를 런타임에 알 수 있지만,
리스트의 경우 컴파일 시 바로 알 수 있다.

2. 타입 정보

- 배열은 런타임에도 자신이 담기로 한 원소의 타입을 인지하고 확인한다.
그래서 위 코드에서 보듯 `Long` 배열에 `String` 을 넣으려 하면 `ArrayStoreException` 이 발생된다.
- 반면, 제네릭은 타입 정보가 런타임에는 소거된다.
원소 타입을 컴파일 타임에만 검사하며 런타임에는 알 수조차 없다는 뜻이다. (아이템 26)

3. 그 외..

- 배열은 제네릭 타입, 매개변수화 타입, 타입 매개변수로 사용할 수 없다.
 - `new List<E>[]`, `new List<String>[]`, `new E[]` 식으로 작성하면 컴파일시 제네릭 배열 생성 오류를 발생한다.
 - 제네릭 배열은 타입 안전하지 않기 때문에, 만들지 못한다.
이를 허용한다면, 컴파일러가 자동 생성한 형변환 코드에서 런타임 `ClassCastException` 이 발생할 수 있다.
런타임에 `ClassCastException` 이 발생 하는 일을 막아주겠다는 제네릭 타입 시스템의 취지에 어긋난다.
 - ex) 제네릭 배열이 된다는 가정

```
List<String>[] stringLists = new List<String>[1]; // 1
List<Integer> intList = List.of(42); // 2
Object[] objects = stringLists; // 3
objects[0] = intList; // 4
String s = stringLists[0].get(0) // 5
```

- 1. 제네릭 배열 생성이 허용된다고 하자.
- 2. 는 원소 하나인 `List<Integer>` 를 생성한다
- 3. 1. 에서 생성한 `List<String>` 의 배열을 `Object` 배열에 할당한다.
(배열은 공변이니 아무 문제없다.)
- 4. 2.에서 생성한 `List<Integer>` 의 인스턴스를 `Object` 배열의 첫 원소로 저장한다. (제네릭은 소거 방식으로 구현되어서 이 역시 성공한다.)
- 즉, 런타임에는 `List<Integer>` 인스턴스 타입은 단순히 `List` 가 되고
`List<Integer>[]` 인스턴스의 타입은 `List[]` 가 된다.
따라서 4.에서도 `ArrayStoreException` 을 일으키지 않는다.
- 이제부터 문제이다. `List<String>` 인스턴스만 담겠다고 선언한 `stringLists` 배열에는 지금 `List<Integer>` 인스턴스가 저장되어 있다.
- 5. 원소를 꺼내는 순간 자동적으로 `String` 형변환을 하는데, 이 원소는 `Integer` 이므로 런타임에 `ClassCastException` 이 발생한다.
 - 이런 일을 방지하고자 제네릭 배열 생성 에서 컴파일 오류를 내야한다.
- 실체화 불가 타입
 - `E, List<E>, List<String>`
 - 실체화되지 않아서 런타임에는 컴파일 타임보다 타입 정보를 적게 가지는 타입이다.
 - 소거 메커니즘 때문에 매개변수화 타입 가운데 실체화 될 수 있는 타입은 `List<?>`, `Map<?, ?>` 과 같은 비한정적 와일드 카드 타입 뿐이다.(아이템 26)
 - 배열을 비한정적 와일드카드 타입으로 만들 수 있지만, 유용하게 쓰일 일이 없다.
- 배열로 형변환 시, 제네릭 배열 생성 오류나 비검사 형변환 경고가 뜨는 경우
 - 배열인 `E[]` 대신 컬렉션인 `List<E>` 를 사용하면 해결된다.
 - 코드가 조금 복잡해지고 성능이 살짝 나빠질 수도 있지만, 그 대신 타입 안전성과 상호운용성은 좋아진다.
 - Ex) Chooser 클래스 (배열)

```
public class Chooser {
    private final Object[] choiceArray;

    public Chooser(Collection choices) {
        choiceArray = choices.toArray();
    }

    public Object Choose() {
        Random rnd = ThreadLocalRandom.current();
        return choiceArray[rnd.nextInt(choiceArray.length)];
    }
}
```

- `choose` 메서드 호출할 때마다 반환된 `Object` 를 원하는 타입으로 형변환해야 한다.
- 혹시, 타입이 다른 원소가 들어 있었다면 런타임에 형변환 오류가 날 것이다.

◦ Ex) 제네릭으로 수정한 Chooser 클래스 (배열)

```
public class Chooser<T> {
    private final T[] choiceArray;

    public Chooser(Collection<T> choices) {
        choiceArray = choices.toArray();
    }

    public Object Choose() {
        Random rnd = ThreadLocalRandom.current();
        return choiceArray[rnd.nextInt(choiceArray.length)];
    }
}

// 컴파일시 오류 메시지
Chooser.java:9: error: incompatible types: Object[] cannot be
converted to T[]
        choiceArray = (T[]) choices.toArray();
                        ^
where T is a type-variable:
  T extends Object declared in class Chooser

// Object 배열을 T 배열로 형변환 한다.
choiceArray = (T[]) choices.toArray();

// 이번엔 경고가 뜬다.
Chooser.java:9: warning: [unchecked] unchecked cast
        choiceArray = (T[]) choices.toArray();
                        ^
required: T[], found: Object[]
where T is a type-variable:
  T extends Object declared in class Chooser
```

- 마지막 경고는, T가 무슨 타입인지 알 수 없으니 컴파일러는 이 형변환이 런타임에도 안전한지 보장 할 수 없다는 메시지이다.
- 제네릭에서는 원소의 타입 정보가 소거되어 런타임에는 무슨 타입인지 알 수 없음을 기억하자.
- 위 코드는 동작하지만, 컴파일러에 안전을 보장하지 못한다.
코드를 작성하는 사람이 안전하다고 확신하면 주석을 남기고 애너테이션을 달아도 되지만, 경고의 원인을 제거하는 편이 훨씬 낫다. (아이템 27)
- Ex) 제네릭 Chooser 클래스 (리스트) - 타입 안전성 확보

```
public class Chooser<T> {
    private final List<T> choiceList;

    public Chooser(Collection<T> choices) {
        choiceArray = new ArrayList<>(choices);
    }

    public T Choose() {
        Random rnd = ThreadLocalRandom.current();
        return choiceList.get(rnd.nextInt(choiceArray.length));
    }
}
```

- 코드양이 조금 늘었고, 성능도 조금 더 느릴 테지만, 런타임에 `ClassCastException` 을 만날리 없으니 그만한 가치가 있다.
- 앞으로..
 - 배열을 제네릭을 만들 수 없어 귀찮을 때가 있다.
 - 제네릭 컬렉션에서는 자신의 원소 타입을 담은 배열을 반환하는 게 보통 불가능하다. (완벽 하지는 않지만 대부분의 상황에서 이 문제를 해결해주는 방법을 아이템 33 에서 설명한다.)
 - 또한 제네릭 타입과 가변인수 메서드 (varargs method, 아이템 53)를 함께 쓰면 해석하기 어려운 경고 메시지를 받게 된다.
 - 가변인수 메서드를 호출할 때마다 가변인수 매개변수를 담은 배열이 하나 만들어지는 데, 이때 그 배열의 원소가 실체화 불가 타입이라면 경고가 발생한다.
 - 이 문제는 `@SafeVarargs` 애너테이션으로 대처할 수 있다. (아이템 32)
- 정리
 - 배열과 제네릭에는 매우 다른 타입 규칙이 적용된다.
 - 배열은 공변이고 실체화되는 반면
 - 제네릭은 불공변이고 타입 정보가 소거된다.
 - 결과적으로, 배열은 런타임에는 타입 안전하지만, 컴파일타임에는 그렇지 않다. 제네릭 은 반대이다.

▼ 29. 이왕이면 제네릭 타입으로 만들라

- 제네릭이 필요한 코드 (아이템 7 Stack 클래스)

```
public class Stack {
    private Object[] elements;
    private int size = 0;
    private static final int DEFAULT_INITIAL_CAPACITY = 16;

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

    public void push(Object e) {
        ensureCapacity();
        elements[size++] = e;
    }

    public Object pop() {
        if (size == 0)
            throw new EmptyStackException();
        Object result = elements[--size];
        elements[size] = null; // 다 쓴 참조 해제
        return result;
    }

    public boolean isEmpty() {
        return size == 0;
    }
}
```

```

    }

    private void ensureCapacity() {
        if (elements.length == size)
            elements = Arrays.copyOf(elements, 2 * size + 1);
    }
}

```

- 이 클래스는 원래 제네릭 타입이어야 마땅하다. (앞으로 제네릭으로 변경한다.)
- 일반 클래스를 제네릭 클래스로 만드는 법
 1. 클래스 선언에 타입 매개 변수를 추가하는 일이다. (타입 이름은 보통 **E** (아이템 68))
- 제네릭 **Stack**

```

public class Stack<E> {
    private E[] elements;
    private int size = 0;
    private static final int DEFAULT_INITIAL_CAPACITY = 16;

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

    public void push(E e) {
        ensureCapacity();
        elements[size++] = e;
    }

    public E pop() {
        if (size == 0)
            throw new EmptyStackException();
        E result = elements[--size];
        elements[size] = null; // 다 쓴 참조 해제
        return result;
    }

    public boolean isEmpty() {
        return size == 0;
    }

    private void ensureCapacity() {
        if (elements.length == size)
            elements = Arrays.copyOf(elements, 2 * size + 1);
    }
}

```

- 대체로 이 과정에서 하나 이상의 오류가 발견된다.
 - 여기서 `new E[DEFAULT_INITIAL_CAPACITY]` 에서 오류가 났다.
 - **E** 와 같은 실체화 불가 타입으로는 배열을 만들 수 없다. (아이템 28)
 - 해결책
 1. 제네릭 배열 생성을 금지하는 제약을 대놓고 우회하는 방법.
 - **Object** 배열을 생성한 다음 제네릭 배열로 형변환.
- 이제 컴파일러는 오류대신 경고를 내보낼 것이다.

이렇게 할 수는 있지만, (일반적으로) 타입 안전하지 않다.

```
Stack.java:8: warning: [unchecked] unchecked cast
found: Object[], required: E[]
    elements = (E[]) new Object[DEFAULT_INITIAL_CAPACITY];
               ^
```

- 컴파일러는 이 프로그램이 타입이 안전한지 증명할 방법이 없지만, 우리는 할 수 있다. 따라서 이 비검사 형변환 프로그램의 타입 안전성을 해치지 않음을 우리 스스로 확인해야 한다.
- 문제의 배열 `elements` 는 `private` 필드에 저장되고, 클라이언트로 반환되거나 다른 메서드에 전달되는 일이 전혀 없다. `push` 메서드를 통해 배열에 저장되는 원소의 타입은 항상 `E` 다. 따라서 이 비검사 형변환은 확실히 안전하다.
- 비검사 형변환이 안전함을 직접 증명했다면 범위를 최소로 좁혀 `@SuppressWarnings` 애너테이션으로 해당 경고를 숨긴다. (아이템 27)

```
// 배열 elements 는 push(E)로 넘어온 E 인스턴스만 담는다.
// 따라서 타입 안전성을 보장한다.
// 이 배열의 런타임 타입은 E[] 가 아닌 Object[] 이다.
@SuppressWarnings("unchecked")
public Stack() {
    elements = (E[]) new Object[DEFAULT_INITIAL_CAPACITY];
}
```

- 위 예에서는 생성자가 비검사 배열 생성 말고는 하는 일이 없으므로 생성자 전체에 경고를 숨겨도 된다.
- 애너테이션을 달면 `Stack` 은 깔끔히 컴파일되고, 명시적으로 형 변환하지 않아도 `ClassCastException` 걱정 없이 사용 할 수 있다.

2. `elements` 필드의 타입을 `E[]` 에서 `Object[]` 로 바꾸는 것이다.

- (당연히 생성자에서도 `new Object[...]` 로 변경한다.)
- 이렇게 하면 첫 번째와는 다른 오류가 발생한다.

```
Stack.java:19 incompatible types
found: Object, required: E
    E result = elements[--size];
               ^
// 배열이 반환한 원소를 E로 형변환하면 오류 대신 경고가 뜬다.
Stack.java:19 [unchecked] unchecked cast
found: Object, required: E
    E result = (E) elements[--size];
               ^
```

- `E` 는 실체화 불가 타입이므로 컴파일러는 런타임에 이뤄지는 형변환이 안전한지 증명할 방법이 없다. 이번에도 마찬가지로 우리가 직접 증

명하고 경고를 숨길 수 있다. `pop` 메서드 전체에서 경고를 숨기지 말고 비검사 형변환을 수행하는 할당문에만 적용 (아이템 27)

```
public E pop() {
    if (size == 0)
        throw new EmptyStackException();
    // push 에서 E 타입만 허용하므로 이 형변환은 안전하다.
    @SuppressWarnings("unchecked") E result = (E) elements[--size];
    elements[size] = null; // 다 쓴 참조 해제
    return result;
}
```

◦ 첫 번째 방법과 두 번째 방법의 비교

- 첫 번째 방법은 가독성이 더 좋다.

배열의 타입을 `E[]` 로 선언하여 오직 `E` 타입 인스턴스만 받음을 확실 코드도 더 짧다.

보통의 제네릭 클래스라면 코드 이것저곳에서 이 배열을 자주 사용할 것이다

첫 번째 방식에서는 형변환을 배열 생성 시 단 한 번만 해주면 되지만, 두 번째 방식에서는 배열에서 원소를 읽을 때마다 해야한다.

현업에서는 첫 번째 방식을 더 선호한다. 하지만 (`E` 가 `Object` 가 아닌 한) 배열의 런타임 타입이 컴파일 타임 타입과 힙 오염(아이템 32)을 일으킨다. 힙 오염이 맘에 걸리는 프로그래머는 두 번째 방식을 고수하기도 한다.

- 지금까지 설명한 `Stack` 의 예는 “배열보다 리스트를 우선하라” (아이템 28) 과 는 모순돼어 보인다.

사실 제네릭 타입 안에서 리스트를 사용하는 게 항상 가능하지도, 꼭 더 좋은 것도 아니다. 자바가 리스트를 기본 타입으로 제공하지 않으므로 `ArrayList` 같은 제네릭 타입도 결국 기본 타입인 배열을 사용해 구현해야 한다.

`HashMap` 같은 제네릭 타입은 성능을 높일 목적으로 배열을 사용하기도 한다.

- `Stack` 에 처럼 대다수의 제네릭 타입은 타입 매개변수에 아무런 제약을 두지 않는다.

`Stack<Object>`, `Stack<int[]>`, `Stack` 등 어떤 참조 타입으로 `Stack` 으로 만들 수 있다.

단, 기본타입은 사용할 수 없다. (`Stack<int>`, `Stack<double>`)

이는 자바 제네릭 타입 시스템의 근본적인 문제이나, 박싱된 기본 타입(아이템 61)을 사용해 우회할 수 있다.

- 정리

- 클라이언트에서 직접 형변환해야 하는 타입보다 제네릭 타입이 더 안전하고 쓰기 편하다
- 새로운 타입을 설계할 때는 형변환 없이도 사용할 수 있도록 하라. 그렇게 하려면 제네릭 타입으로 만들어야 할 경우가 많다. 기존 타입 중 제네릭이었어야 하는 게 있다면 제네릭 타입으로 변경하자
- 기존 클라이언트에는 아무영향을 주지 않으면서, 새로운 사용자를 훨씬 편하게 해준다. (아이템 26)

▼ 30. 이왕이면 제네릭 메서드로 만들라

- 매개변수화 타입을 받는 정적 유틸리티 메서드는 보통 제네릭이다
 - ex) 두 집합의 합집합을 반환하는, raw 타입 사용한 메서드이다.

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

// 컴파일은 되지만 경고가 두 개 발생한다.
Union.java:5: warning: [unchecked] unchecked call to
HashSet(Collection<? extends E>) as a member of raw type HashSet
    Set result = new HashSet(s1);
                  ^
Union.java:6: warning: [unchecked] unchecked call to
addAll(Collection<? extends E>) as a member of raw type Set
    result.addAll(s2);
              ^
```

- 경고를 없애려면 이 메서드를 타입 안전하게 만들어야 한다.
 - 메서드 선언에서의 세 집합 (입력 2, 반환 1)의 원소 타입을 타입 매개변수로 명시하고 메서드 안에서도 이 타입 매개변수만 사용하게 수정하면 된다.
 - (타입 매개변수들을 선언하는) 타입 매개변수 목록은 메서드의 제한자와 반환 타입 사이에 온다.

- ex) 제네릭 메서드

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

- 경고 없이 컴파일되며, 타입 안전하고, 쓰기도 쉽다.
- 집합 3개 (입력 2, 반환 1) 타입이 모두 같아야 한다.
 - 한정적 와일드 카드 타입 (아이템 31)을 사용하여 더 유연하게 개선할 수 있다.
- 때로는 불변 객체를 여러 타입으로 활용할 수 있게 만들어야 할 때가 있다.
 - 제네릭은 런타임에 타입 정보가 소거 (아이템 28) 되므로 하나의 객체를 어떤 타입으로든 매개변수화 할 수 있다.
 - 하지만 이렇게 하려면 요청한 타입에 맞게 매번 그 객체의 타입을 바꿔주는 정적 팩터리를 만들어야 한다. (**제네릭 싱글턴 팩터리**)
`Collections.reverseOrder` 같은 함수 객체 (아이템 42) 나 `Collections.emptySet` 같은 컬렉션용으로 사용한다.

- ex) 항등함수를 담은 클래스

- **항등함수** 객체는 상태가 없다. 따라서 요청할 때마다 새로 생성하는 것은 낭비다.

- 제네릭은 소거 방식을 사용하기 때문에 **제네릭 싱글턴** 하나면 항등함수를 만들기에 충분하다.

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

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

- IDENTITY_FN을 `UnaryOperator<T>` 로 형변환하면 비검사 형변환 경고가 발생한다. `T`가 어떤 타입이든 `UnaryOperator<Object>` 는 `UnaryOperator<T>` 가 아니기 때문이다.
- 하지만 항등함수란 입력 값을 수정 없이 그대로 반환하는 특별한 함수이므로, `T`가 어떤 타입이든 `UnaryOperator<T>` 를 사용해도 타입 안전하다.
- `@SuppressWarnings` 애너테이션을 추가하여 오류나 경로를 없이 컴파일한다.
 - ex) 위에 함수(제네릭 싱글턴)를 사용하는 예제

```
String [] strings = { "삼베", "대마", "나일론" };
UnaryOperator<String> sameString = identityFunction();
for (String s : strings)
    System.out.print(sameString.apply(s));

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

- 형변환하지 않아도 컴파일 오류나 경고가 발생지 않는다.

• 재귀적 타입 한정

- 자기 자신이 들어간 표현식을 사용하여 타입 매개변수의 허용 범위를 한정할 수 있다.
- 주로 타입의 자연적 순서를 정하는 `Comparable` 인터페이스 (아이템 14) 와 함께 쓰인다.
- ex) `Comparable` 인터페이스

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

- 타입 매개변수 `T` 는 `Comparable<T>` 를 구현한 타입이 비교할 수 있는 원소의 타입을 정의한다. 실제로 거의 모든 타입은 자신과 같은 타입의 원소와만 비교할 수 있다.
- 재귀적 타입 한정을 이용해 상호 비교 할 수 있음.

```
public static <E extends Comparable<E>> E max(Collection<E> c) {
    if (c.isEmpty())
        throw new IllegalArgumentException("컬렉션이 비어 있습니다.");
}
```

```

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

    return result;
}

```

- 타입 한정인 `<E extends Comparable<E>>` 는 모든 타입 E는 자신과 비교 할 수 있다 상호 비교 가능하다는 뜻을 아주 정확히 표현했다.
- tmi. 이 메서드에 빈 컬렉션을 건네면 `IllegalArgumentException` 을 던지니, `Optional<E>` 를 반환하는 편이 좋다. (아이템 55)

• 정리

- 제네릭 타입(제네릭 클래스)과 마찬가지로, 클라이언트에서 입력 매개변수와 반환값을 명시적으로 형변환 해야 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.
- 역시 타입과 마찬가지로, 형변환을 해줘야 하는 기존 메서드는 제네릭하게 만들자.

▼ 31. 한정적 와일드카드를 사용해 API 유연성을 높이라

- 매개변수화 타입은 불공변이다. (아이템 28에서 나옴)
 - `Type1`, `Type2` 가 있을 때, `List<Type1>` 은 `List<Type2>` 의 하위 타입도 상위 타입도 아니다.
 - `List<String>` 은 `List<Object>` 의 하위 타입이 아니라는 뜻인데, 곰곰히 따지면 사실이 이쪽이 말이 된다.
 - `List<String>` 에는 문자열만 넣을 수 있다. 즉 `List<String>` 은 `List<Object>` 가 하는 일에 제대로 수행하지 못하니 하위 타입이 될 수 없다. (리스코프 치환 원칙 위반)
 - 하지만 때론 불공변 방식보다 유연한 방식이 필요하다.
 - ex) 아이템 29의 Stack 클래스 의 public API

```

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

```

- 일련의 원소를 스택에 넣는 메소드를 추가해보자.

```

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

```


- 이 메서드는 깨끗이 컴파일 되지만, 완벽하지 않다. `Iterable` `src` 의 원소 타입이 스택의 원소 타입과 일치하면 잘 동작한다.
- `Stack<Number>` 로 선언한 후 `pushAll(intVal)` 을 호출하면 `Integer(intVal)` 의 하위 타입이니 논리적으로는 잘 동작 해야 할 것 같다. **하지만 에러가난다.**

```
Stack<Number> numberStack = new Stack<>();
Iterable<Integer> integers = ...;
numberStack.pushAll(integers);

// 매개변수화 타입이 불공변이기 때문에, 오류가 뜬다.
StackTest.java: 7: error: incompatible types: Iterable<Integer>
cannot be converted to Iterable<Number>
    numberStack.pushAll(integers);
                        ^
```

- 해결책으로 한정적 와일드카드 타입이라는 특별한 매개변수화 타입을 지원한다.

- `Iterable<? extends E>`

`pushAll` 의 입력 매개변수 타입은 `E` 의 `Iterable` 이 아니라 `E` 의 하위 타입의 `Iterable` 이어야 한다는 뜻이다.

- 수정된 `pushAll` 메서드 (와일드카드 타입 적용)

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

- `popAll` 메서드 (`Stack` 의 모든 원소를 주어진 컬렉션으로 옮겨 담는다.)

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

- `pushAll` 가 마찬가지로 컬렉션의 원소 타입이 스택의 원소 타입과 일치한다면 컴파일되고 문제없이 동작한다. 하지만 이번에도 완벽하지는 않다.
- `Stack<Number>` 의 원소를 `Object` 용 컬렉션으로 옮긴다면, 잘 동작 해야 할 것 같다. **하지만 에러가난다.**

```
Stack<Number> numberStack = new Stack<>();
Collection<Object> objects = ...;
numberStack.popAll(objects);

// 에러 Collection<Object>는 Collection<Number>의 하위 타입이 아니다.
numberStack.popAll(objects);
                        ^
```

- 위와 똑같이 와일드카드 타입으로 해결 할 수 있다.

- `Collection<? super E>`

`popAll`의 입력 매개변수의 타입이 `E`의 `Collection`이 아니라 `E`의 상위 타입의 `Collection`이어야 한다. (모든 타입은 자기 자신의 상위 타입이다.)

- 수정된 `popAll` 메서드 (와일드카드 타입 적용)

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

- 유연성을 극대화하려면 원소의 생산자의 소비자용 입력 매개변수에 와일드카드 타입을 사용하라

- 입력 매개변수가 생산자와 소비자 역할을 동시에 한다면 와일드카드 타입을 써도 좋을 게 없다. (타입을 정확히 지정해야 하는 상황으로, 와일드카드 타입을 쓰지 말아야 한다.)

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

- 매개변수와 타입 `T`가 생산자라면 `<? extends T>`를 사용하고, 소비자라면 `<? super T>`를 사용하라

- `Stack` 예에서

- `pushAll`의 `src` 매개변수는 `Stack`이 사용할 `E` 인스턴스를 생산하므로 `src`의 적절한 타입은 `Iterable<? extends E>`이다.
- `popAll`의 `dest` 매개변수는 `Stack`으로부터 `E` 인스턴스를 소비하므로 `dest`의 적절한 타입은 `Collection<? super E>`이다

- 예시

```
// PECS 적용 이전
public static <E> Set<E> union(Set<E> s1, Set<E> s2)
// PECS 적용 이후
public static <E> Set<E> union(Set<? extends E> s1, Set<? extends E> s2)
```

- 반환 타입에는 한정적 와일드 카드 타입을 사용하면 안된다.
유연성을 높여주는 커널 클라이언트 코드에서 와일드카드 타입을 사용해야 하기 때문이다.

```
// 클라이언트 코드 (Set.of 자바 9)
Set<Integer> integers = Set.of(1,3,5);
Set<Double> doubles = Set.of(2.0, 4.0, 6.0);
Set<Number> numbers = union(integers, doubles);
```

- 와일드카드가 제대로 사용된다면 사용자는 와일드 카드가 사용된지도 모른다. 만약 사용자가 와일드카드 타입을 신경써야 한다면 그 API는 문제가 있을 수 있다.

- 자바 버전 7 이전이라면 위 클라이언트 코드에서 명시적 타입 변수를 사용해야한다. 자바 8 이전까지는 타입 추론 능력이 충분하지 못해 반환타입을 명시해야 한다.

```
Set<Number> numbers = Union.<Number>union(integers,doubles);
```

- 타입 매개변수와 와일드카드에는 공통되는 부분이 있어서, 메서드를 정의할 때 둘 중 어느 것을 사용해도 괜찮을 때가 많다
 - 기본 규칙: 메서드 선언에 타입 매개변수가 한번만 나오면 와일드 카드로 대체하라

```
// 비한정적 타입 매개변수
public static <E> void swap(List<E> list, int i, int j);
// 비한정적 와일드 카드
public static void swap(List<?> list, int i, int j);
```

- 이때 한정적 타입 매개변수라면 한정적 와일드카드로
비한정적 타입 매개변수라면 비한정적 와일드카드로 변경하면 된다.
- 하지만 아래 코드는 컴파일 하면 그다지 도움이 되지 않는 오류 메시지가 나온다.

```
public static void swap(List<?> list, int i, int j) {
    list.set(i, list.set(j, list.get(i)));
}
```

- 이 코드는 `list.get(i)` 로 꺼낸 코드를 다시 리스트에 넣을 수 없다는 오류를 발생시킨다 (List<?> 은 null 만 넣을 수 있음)
- 다행히 (런타임 오류를 낼 가능성이 있는) 형변환이나 리스트의 raw 타입을 사용하지 않고도 해결할 방법이 있다.
- 바로 와일드카드 타입의 실제 타입을 알려주는 메서드를 `private` 도우미 메서드로 따로 작성하여 활용하는 방법이다.

```
public static void swap(List<?> list, int i, int j) {
    swapHelper(i, list.set(j, list.get(i)));
}
public static <E> void swapHelper(List<E> list, int i, int j) {
    list.set(i, list.set(j, list.get(i)));
}
```

- `swapHelper` 메서드는 `List<E>` 에서 꺼낸 타입이 항상 `E` 이고 `E` 타입의 값은 해당 `List` 에 다시 넣어도 안전함을 알고 있다.

• 정리

- 조금 복잡하더라도 **와일드카드 타입**을 적용하면 API 가 훨씬 유연해진다.
- 그러니 널리 쓰일 라이브러리를 작성한다면 반드시 와일드카드 타입을 적절히 사용해주어야 한다.

- PECS 공식을 기억하자. producer 는 extends, consumer 는 super
- Comparable 과 Comparator 는 모두 소비자 이다.

▼ 32. 제네릭과 가변인수를 함께 쓸 때는 신중하라

• 가변인수

- 메서드에 넘기는 인수의 개수를 클라이언트가 조절할 수 있게 해줌 (`String... s`)
- 가변인수 메서드를 호출하면 가변인수를 담기 위한 배열이 자동으로 하나 만들어진다.
- 그런데, 내부로 감춰야 했을 이 배열을 그만 클라이언트에 노출하는 문제가 발생함.
- 그로 인해, `varargs` 매개변수에 제네릭이나 매개변수화 타입이 포함되면 알기 어려운 컴파일 결과 발생한다. (힙 오염)
 - 메서드를 선언할 때, 실체화 불가 타입으로 `varargs` 매개변수를 선언하면 컴파일러가 경고를 보낸다. 가변인수 메서드를 호출 할 때도 `varargs` 매개변수가 실체화 불가 타입으로 추론되면, 경고를 낸다. (`List<E>... list`)
 - 매개변수화 타입의 변수가 타입이 다른 객체를 참조하면 힙 오염이 발생한다.

```
warning: [unchecked] Possible heap pollution from
parameterized vararg type List<String>
```

■ 위 오류 예시 코드

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

- 이 메서드에서는 형변환하는 곳이 보이지 않는데도 인수를 건네 호출하면 `ClassCastException` 을 던진다. 마지막 줄에 컴파일러가 생성한 (보이지 않는) 형변환이 숨어 있기 때문이다.
- 이처럼 타입 안정성이 깨짐으로 제네릭 `varargs` 배열 매개변수에 값을 저장하는 것은 **안전하지 않다.**
- 재미난 질문 - 제네릭 배열을 프로그래머가 직접 생성하는 건 허용하지 않지만, 제네릭 `varargs` 매개변수를 받는 메서드를 선언할 수 있게 한 이유는 무엇일까?
 - 제네릭이나 매개변수화 타입의 `varargs` 매개변수를 받는 메서드가 실무에서 매우 유용하기 때문이다.
 - ex) `Arrays.asList(T... a)`, `Collections.addAll(Collection<? super T> c, T... elements)`, `EnumSet.of(E first, E... rest)` 등
 - 위 메서드들은 타입 안전하다.

- java 7 이전에는 제네릭 가변인수 메서드의 작성자가 호출자 쪽에서 발생하는 경고에 대해 할 게 없었다. (그래서 사용자는 `@SuppressWarnings("unchecked")` 달고 경고를 숨기 기만 해야했다. (아이템 27))
- java 7 이후에는 `SafeVarargs` 애너테이션이 추가되어 제네릭 가변인수 메서드 작성자가 클라이언트 측에서 발생하는 경고를 숨길 수 있게되었다.
- 즉, `@SafeVarargs` 애너테이션은 메서드 작성자가 그 메서드가 타입 안전함을 보장하는 장치이다.

○ `Arrays.asList` (java 11)

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

• 제네릭 가변인수 메서드가 안전한지 확인하는 방법 (`@SafeVarargs` 를 사용할 수 있을 때)

- 메서드가 제네릭 배열에 아무것도 저장하지 않아야 한다. (매개변수들을 덮어 쓰지 말라)
- 제네릭 배열의 참조가 밖으로 노출되지 않아야 한다. (신뢰할 수 없는 코드가 배열에 접근 할 수 없다)
- 위, 두 개를 확인하면 해당 제네릭 가변인수 메서드는 안전하다.
- 제네릭 매개변수 배열의 참조를 노출한다. (안전하지 않다.)

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

- 반환하는 배열의 타입은 이 메서드에 인수를 넘기는 컴파일 타임에 결정된다.
그 시점에는 컴파일러에게 충분한 정보가 주어지지 않아 타입을 잘못 판단할 수 있다.
따라서 힙 오염이 발생되어 호출한 쪽의 call stack 까지 전이될 수 있다.
- `T` 타입 인수 3개를 받아 그 중 2개를 무작위로 골라 담은 배열을 반환
(`toArray` 는 위에 예제 (안전하지 않은) 메서드 이다.)

```
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, b);
        case 2: return toArray(a, b);
    }
    throw new AssertionError(); // 도달할 수 없다.
}
```

- 컴파일러는 `toArray` 에 넘길 `T` 인스턴스 2개를 담은 `varargs` 매개변수 배열을 만드는 데, 배열 타입은 `Object` 이다. (어떤 객체를 넘기더라도 담을 수 있는 가장 구체적인 타입)

- `toArray` 메서드가 돌려준 이 배열이 그대로 `pickTwo` 를 호출한 클라이언트까지 전달된다. 즉, 만약 호출하는 클라이언트 코드에서 `Object[]` 배열이 아닌 다른 타입의 배열이라면 `classCastException` 을 던진다.
 - 위에 설정된 제네릭 `varargs` 매개변수 배열에 다른 메서드가 접근하도록 허용하면 안전하지 않다는 것을 상기 시킨다.
 - 예외
 1. `@SafeVarargs` 로 제대로 애노테이트된 또 다른 `varargs` 메서드에 넘기는 것은 안전하다.
 2. 이 배열 내용의 일부 함수를 호출만 하는 (`varargs` 를 받지 않는) 일반 메서드에 넘기는 것도 안전하다.
- `@SafeVarargs` 애너테이션이 유일한 정답은 아니다. `varargs` 매개변수를 `List` 매개변수로 바꿀 수 있다. (아이템 28)

```
@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;
}

static <T> List<T> flatten(List<List<? extends T>> lists) {
    List<T> result = new ArrayList<>();
    for (List<? extends T> list : lists)
        result.addAll(list);
    return result;
}
```

- `@SafeVarargs` 애너테이션을 우리가 직접 달지 않아도 된다.
 - 실수로 안전하다고 걱정할 일도 없다.
 - 단점이면 클라이언트 코드가 살짝 지저분하고, 속도가 조금 느려질 수 있다. (`List` 객체는 어차피 배열임)
- 정리
 - 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.
 - 메서드에 제네릭 (또는 매개변수화 된) `varargs` 매개변수를 사용하고자 한다면, 먼저 그 메서드가 타입 안전한지 확인한 다음 `@SafeVarargs` 애너테이션을 달아 사용하는데 불편함 없게하자.

▼ 33. 타입 안정 이중 컨테이너를 고려하라