아이템 32
제네릭과 가변인수를 함께 쓸 때는 신중하라

실체화 불가 타입으로
가변인수 매개변수를 쓰면 위험하다.

실체화 불가 타입(제네릭 등)은 런타임에 컴파일타임보다 타입 정보가 적다.

메서드의 선언 시 실체화 불가 타입으로 가변인수 매개변수 선언

or

메서드 호출 시 가변인수 매개변수가 실체화 불가 타입으로 추론

-> warning: [unchecked] Possible heap pollution from parameterized vararg type ...

가변인수 + 제네릭의 힙 오염

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

형변환이 없는데도
ClassCastException!

Why? 마지막 줄에서 컴파일러가 임의로 형변환, 컴파일은 통과하지만 타입이 다르다
-> 타입 안정성이 깨진다.

힙 오염

아이템 28로 돌아가보자

배열은 실체화(reify)된다.

배열은 런타임에도 자신의 원소 타입을 인지하고 확인한다. 하지만 제네릭은 런타임에는 소거된다. **컴파일시 확인하면 런타임에는 알 수 없는 것**이다. 소거는 제네릭이 지원되기 전의 레거시 코드와 제네릭 타입을 함께 사용할 수 있게 해주는 메커니즘을 위해서 존재한다. (아이템 26)

이로 인해 배열과 제네릭은 잘 어우러지지 못한다. 배열은 제네릭 타입, 매개변수화 타입, 타입 매개변수로 사용할 수 없다. 즉, 코드를 `new List<E>[]`, `new List<String>[]`, `new E[]` 식으로 작성하면 컴파일할 때 제네릭 배열 생성 오류가 난다.

그럼 제네릭 배열을 왜 막아놨을까? 타입 안전하지 않기 때문이다. 이를 허용하면 컴파일러가 자동 생성한 형변환 코드에서 런타입에 ClassCaseException이 발생할 수 있다. 그러면 런타임에 ClassCaseException이 발생하는 일을 막아주겠다는 제네릭의 취지에 어긋난다.

만약 제네릭 배열을 허용한다고 가정해보자.

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

1이 허용한다고 가정해보자. 3은 1에서 생성한 `List<String>`의 배열을 `Object` 배열에 할당한다. 배열은 공변이니 문제 없다. 4는 2에서 생선한 `List<Integer>` 인스턴스를 `Object` 배열의 첫 원소로 저장한다. 제네릭은 소거 방식으로 구현되어서 런타임에는 `List<Integer>` 인스턴스 타입은 단순히 List가 되고, `List<Integer>[]` 인스턴스 타입은 `List[]` 가 된다. 따라서 4도 문제 없다. 결국 `List<String>` 인스턴스만 담겠다고 선언한 `stringLists`에 `List<Integer>` 인스턴스가 저장되어있다. 그 원소를 `String`으로 캐스팅 하려하니 오류가 발생한다.

`E`, `List<E>`, `List<String>` 같은 타입을 실체화 불가 타입(non-relifiable type)이라 한다. 쉽게 말해, 실체화 되지 않아 컴파일타임보다 런타임에 타입 정보를 적게 가지는 것이다.

제네릭 컬레션에서 자신의 원소 타입을 담은 배열을 반환하는게 불가능하다 (아이템 33을 통해 우회할 수는 있다). 또 제네릭 타입과 가변인수 메서드를 함께 쓰면 해석하기 어려운 경고 메시지를 받게된다. 가변인수 메서드를 호출할 때마다 가변인수 매개변수를 담을 배열이 하나 만들어지는데, 이때 그 배열의 원소가 실체화 불가 타입이라면 경고가 발생하는 것이다.

배열로 형변환할 때 오류나 경고가 뜨는 경우 `E[]` 대신 컬렉션 `List<E>`를 사용하면 해결된다. 코드가 복잡해지고 성능이 살짝 나빠질 수 있지만, 타입 안정성과 상호운용성은 좋아진다.

일반적인 상황이라면 아이템 28처럼
이전의 코드에서도 컴파일 오류가 발생

But 유용성을 위해 가변인수로는 허용

예를 들면?

Arrays.asList(T... a), Collections.addAll(Collection<? super T> c, T... elements)...

@SafeVarargs

```java
@SafeVarargs
@SuppressWarnings("varargs")
static <E> List<E> of(E... elements) {
    switch (elements.length) { // implicit null check of elements
        case 0:
            return ImmutableCollections.emptyList();
        case 1:
            return new ImmutableCollections.List12<>(elements[0]);
        case 2:
            return new ImmutableCollections.List12<>(elements[0], elements[1]);
        default:
            return new ImmutableCollections.ListN<>(elements);
    }
}
```

타입 안전하지 않으면
절대 사용 X!!!

원래는 제네릭 가변인수 메서드를 작성하면 클라이언트에 경고를 주지만
메서드 작성자가 타입 안전을 보장하여 컴파일러가 경고를 발생시키지 않는다.

메서드가 안전한지 어떻게 확신할 건데?

- 메서드가 가변인수를 담는 배열에 아무것도 저장하지 않을 것
- 가변인수를 담은 배열의 참조가 밖으로 노출되지 않을 것
  (신뢰할 수 없는 코드가 배열에 접근하지 못하도록 할 것)

- 즉, varargs 매개변수 배열이 순수하게 인수를 전달하는 역할만 할 것

주의!

매개변수 배열에 아무것도 저장하지 않고도 타입 안정성을 깰 수 있다.

```
public static void main(String[] args) {
    String[] argsArray = pickTwo("a" , "b", "c");
    System.out.println(argsArray);
}

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);
        default: throw new IllegalStateException();
    }
}

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

전혀 문제 없어 보이는 메서드
컴파일도 정상적으로 완료

ClassCastException 발생

왜 그럴까?

pickTwo 메서드는 Object[]를 반환

-> 제네릭을 담기에 가장 구체적인 타입이 Object 이므로

Object[]로 반환된 pickTwo의 반환값이 String[] 으로 자동 형변환

-> Object[]는 String[]의 하위 타입이 아니므로 형변환 불가

Q. toArray만 호출해도 예외가 발생하나요?

A. Nope! toArray 자체는 제네릭 배열을 그대로 반환하므로 X

단, 예외도 존재

@SafeVarargs가 제대로 사용된 또다른 가변인수 메서드로 넘기는 경우

배열의 일부를 가변인수 메서드를 받지 않는 일반 메서드에 넘기는 경우

-> 이 경우는 안전하다.

단, 예외도 존재

```
@SafeVarargs
public 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;
}
```

@SafeVarargs 결론

제네릭이나 매개변수화 타입의 가변인수 매개변수를 받는 모든 메서드에
@SafeVarargs를 달아야만 한다.

@SafeVarargs는 타입 안전한 메서드에만 사용해야 하므로
모든 가변인수 메서드를 타입 안전하도록 작성해야 한다는 말과 같다.

List를 사용하는 방법

어노테이션 사용 대신, 배열을 List로 변환해서 사용하는 방법

아이템 28. 배열보다는 리스트를 사용하라

List를 사용하는 방법

```
public static <T> List<T> pickTwo(T a, T b, T c) {
    switch(ThreadLocalRandom.current().nextInt(3)) {
        case 0: return List.of(a, b);
        case 1: return List.of(a, c);
        case 2: return List.of(b, c);
        default:
            throw new IllegalStateException();
    }
}
```

List.of는 대표적인 타입 안전한
가변인수 메서드

장점

컴파일러가 메서드의 타입 안정성을 검증해준다.

어노테이션을 직접 달 필요가 없다.

사용자의 실수로 안전하지 않은데 안전하다고 판단할 일이 없다.

단점

클라이언트 코드가 지저분해진다.

속도가 배열 사용에 비해서는 느리다.