

[아이템 32] 제네릭과 가변인수를 함께 쓸 때는 신중하라

요약

1. 가변인자를 갖는 메서드는 배열을 사용해서 인자들을 받아온다

2. 비 구체화 타입을 가변인자로 사용하면 warning 이 뜬다

우선, 비 구체화 타입을 배열의 원소로 갖는 배열 생성식은 컴파일 에러가 발생한다

그렇다면 가변인자로 비 구체화 타입을 사용하게 되는 경우는 어떻게 되느냐

warning 메시지를 없애는 방법 - @SafeVarargs

3. 가변인자로 제네릭을 사용하고 있지만 type-safe 한 메서드란?

type-safe 하지 않는 메서드의 예

위의 메서드를 type-safe 한 메서드로 만들어보자

요약

- 🧑 제네릭과 가변인수는 함께 사용하면 타입 안정성이 깨질 수 있다.

배열은 공변이면서 구체적인 타입인 반면, 제네릭은 불변이면서 런타임 시에 타입 정보가 없어지는 비 구체적 타입이다 → 물과 기름 같은 놈들이라 잘 혼용되지 않는다

- 타입 안정성을 깨지 않는 경우에 한해, 제네릭과 가변인수를 잘 쓰면 유용하긴 해서 컴파일 에러 대신에 에러 메시지를 띄우는 식으로 처리된다.

- 🍀 제네릭과 가변인수를 함께 쓰지만 메소드가 타입 안전한 경우라면?

`@SafeVarargs` 어노테이션을 붙여라. 이로써 warning 메시지를 없애라

- 타입 안전한 메서드란?

메서드 내에서 어떤 배열에 아무것도 저장하지 않고,
배열의 참조가 밖으로 노출되지 않는다면 타입 안전하다고 한다

- 제네릭과 가변인수를 함께 써야 한다면?

가변인수를 List 로 대체하는건 어떨지 고민해라

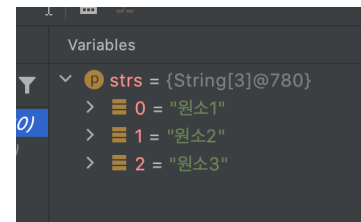
1. 가변인자를 갖는 메서드는 배열을 사용해서 인자들을 받아온다

```
public class Sample {
    public static void display(String... strs) {
        for (String s : strs) {
            System.out.println("str:" + s);
        }
    }

    public static void main(String[] args) {
        display("원소1", "원소2", "원소3");
    }
}
```



```
3 public class Sample {
4     public static void display(String... strs) { strs: {"원소1", "원소2", "원소3"}
5         for (String s : strs) { strs: {"원소1", "원소2", "원소3"}
6             System.out.println("str:" + s);
7         }
8     }
9
10    public static void main(String[] args) {
11        display(...strs: "원소1", "원소2", "원소3");
12    }
13 }
14
```



Variables	
strs	= {String[3]@780}
> 0	= "원소1"
> 1	= "원소2"
> 2	= "원소3"

가변인자를 갖는 메소드를 매번 호출할 때마다 가변인자를 저장할 배열이 생성된다.

2. 비 구체화 타입을 가변인자로 사용하면 warning 이 뜬다

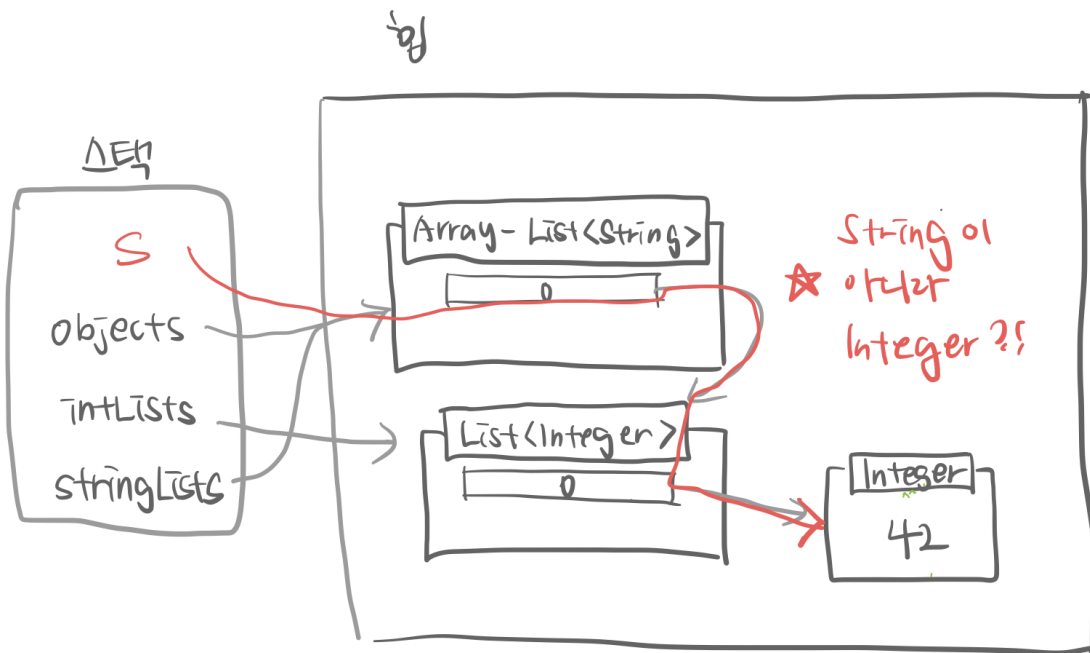
- 배열은 구체화 타입(reified type) 제네릭의 경우 비 구체화 타입(non-reified type)이다
- 비 구체화 타입? 컴파일 시보다 런타임 시에 더 적은 정보를 갖는 것
ex) E, List<E>, List<String> - 비 구체화 타입

우선, 비 구체화 타입을 배열의 원소로 갖는 배열 생성식은 컴파일 에러가 발생한다

```
6 ▶ public class ErrorSample {
7 ▶   public static void main(String[] args) {
8       /*1*/ List<String>[] stringLists = new List<String>[1];
9       /*2*/ List<Integer> intList = Arrays.asList(42);
10      /*3*/ Object[] objects = stringLists;
11      /*4*/ objects[0] = intList;
12      /*5*/ String s = stringLists[0].get(0);
13   }
14 }
```

Generic array creation

배열은 구체화 타입이지만, 제네릭은 비 구체화 타입이다. 따라서 둘은 잘 혼용되지 않으며 `new List<E>[]`, `new List<String>[]`, `new E[]` 와 같은 배열 생성식을 사용할 수 없다. 컴파일 시에 제네릭 배열 생성 에러가 발생한다.



- 만약 컴파일 에러가 발생하지 않는다면?

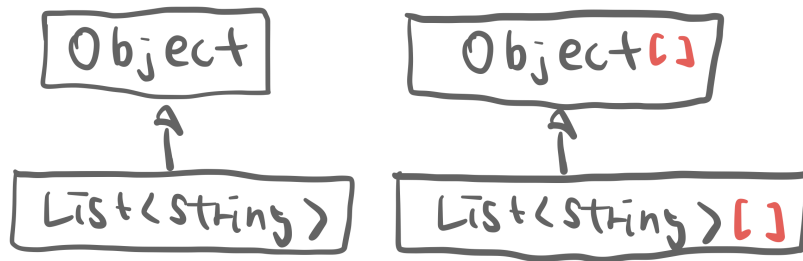
1. 원소로 비 구체화 타입 `List<String>` 을 사용하는 배열을 생성 (원래는 여기서 컴파일 에러 발생)
2. Integer List 생성, 초기화
3. `Object[]` 타입의 참조변수로 `List<String>[]` 인스턴스를 가리킴

이게 가능한 이유? 배열은 불변이다 & 업캐스팅의 경우 명시적인 타입 캐스팅이 필요 없다

→ `List<String>` 은 `Object` 의 자식

→ `List<String>` 배열 은 `Object` 배열 의 자식

→ 서브 클래스의 객체인 `List<String>` 의 인스턴스 가 수퍼 클래스 타입 `Object` 배열 으로 형변환되는 것(=업캐스팅)은 명시적인 타입 캐스팅이 필요 없음



`Object[]` 는 `List<String>[]` 의 부모 타입이다. 왜냐? 배열이 공변이니까

`Object[]` objects
 = `StringLists`
 = `new List<String>[]`
 → 배열은 공변이니까

4. `List<Integer>` 를 `Object[]` 의 요소로 저장

이게 가능한 이유? 제네릭이 소거자에 의해 구현되는 비 구체화 타입이기 때문

objects[0] = intList

List<String>[0] List<Integer>

↓ ↓

List[] List

List[] 요소로 List 인스턴스를 추가하는 상황
왜냐? 배열은 비구체화 타입이니까

5. 컴파일러 입장에서 `stringLists[0]` 는 `List<String>` 이고 `stringLists[0].get(0)` 은 `String` 이 나올 것이라고 생각함. 그래서 컴파일러는 읽어들인 요소를 `String` 으로 캐스팅하는 코드를 생성하게 되는데, 실제로 읽은 요소는 `Integer` 객체이기 때문에 `ClassCastException` 런타임 예외가 발생하게 된다.

명시적인 캐스트가 없는 라인에서 `ClassCastException` 예외가 발생해버릴 수 있다.

이러한 위험성 때문에 비 구체화 타입을 배열로 원소로 가지는 배열 생성식을 사용하는 경우 컴파일 에러가 발생한다.

그렇다면 가변인자로 비 구체화 타입을 사용하게 되는 경우는 어떻게 되느냐

비 구체화 타입을 배열의 원소로 사용하게 되면 컴파일 에러가 발생한다.

가변 인자를 가지는 메소드는 내부적으로 배열을 생성한다고 했는데,

그렇다면 가변 인자로 비 구체화 타입인 제네릭을 넘길때도 컴파일 에러가 발생하는가? →

LL warning 만 발생

```
// It is unsafe to store a value in a generic varargs array parameter (Page 146)
public class Dangerous {
    // Mixing generics and varargs can violate type safety!
    @SuppressWarnings("unchecked")
    static void dangerous(List<String>... stringLists) {
        List<Integer> intList = List.of(42);
        Object[] objects = stringLists;
        objects[0] = intList; // Heap pollution
        String s = stringLists[0].get(0); // ClassCastException
    }

    public static void main(String[] args) { dangerous(List.of("There be dragons!")); }
}
```

가변인자로 비 구체화 타입을 사용하게 될 때, 컴파일 에러가 발생하지는 않는다. 하지만 위의 코드는 `ClassCastException` 이 발생한다.

- 가변인자로 제네릭을 사용하게 되면, 이 경우는 warning 이 발생한다

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

- 힙 오염이란?

매개변수화 타입(Parameterize Type)의 변수가 타입이 다른 객체를 참조하면 힙 오염 이 발생한다

- 왜 이때는 컴파일 에러를 발생시키지 않을까?

가변인자로 제네릭 타입(`List<E>`) 이나 매개변수화 타입(`List<String>`) 을 사용하는 경우 유용하게 쓰일 수 있기 때문

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

`Arrays.asList(T... a)`

```
@SafeVarargs
public static <T> boolean addAll( @NotNull Collection<? super T> c, @NotNull T... elements) {
    boolean result = false;
    for (T element : elements)
        result |= c.add(element);
    return result;
}
```

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

```
@SafeVarargs
public static <E extends Enum<E>> EnumSet<E> of(E first, E... rest) {
    EnumSet<E> result = noneOf(first.getDeclaringClass());
    result.add(first);
    for (E e : rest)
        result.add(e);
    return result;
}
```

EnumSet.of(E first, E... reset)

즉, 메소드가 typesafe 한 경우에 한해서 잘 쓰면 유용한데,
잘못 쓰면 ClassCastException 을 발생시킬 수 있다는 것.

warning 메시지를 없애는 방법 - @SafeVarargs

Arrays.asList(T... a), Collections.addAll(Collection<? super T> c, T... elements), EnumSet.of(E first, E... reset) 와 같은 메서드는 가변인자로 제네릭을 사용하고 있지만 type-safe 한 메서드이다.

어떤 메서드가 type-safe 함을 확신할 수 있다면 @SafeVarargs 어노테이션을 붙여서 warning 메시지가 뜨지 않도록 할 수 있다.

3. 가변인자로 제네릭을 사용하고 있지만 type-safe 한 메서드란?

1. 가변인자를 사용함으로써 만들어지는 배열에 어떠한 것도 덮어쓰지 않고
2. 그 배열을 가리키는 참조변수를 직간접적으로 만들지 않는다면

이러한 메서드를 type-safe 하다고 이야기한다.

즉, 가변인자가 여러 인자들을 넘겨오는데에만 사용되는 메서드를 일컫는다.

그리고 이러한 메서드에는 안심하고 @SafeVarargs 어노테이션을 붙여 줄 수 있다.

type-safe 하지 않는 메서드의 예

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

▼ 이 메서드는 왜 type-safe 하지 않을까?

가변 인자로 넘어 온 여러 데이터들을 하나의 배열로 묶어주는 메서드

이 메서드는 가변인자 배열을 가리키는 참조변수를 리턴하고 있기 때문에 힙 오염을 발생시킬 가능성이 있다.

```
public class PickTwo {  
    // UNSAFE - Exposes a reference to its generic parameter array!  
    static <T> T[] toArray(T... args) {  
        return args;  
    }  
  
    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(); // Can't get here  
    }  
  
    public static void main(String[] args) {  
        String[] attributes = pickTwo("Good", "Fast", "Cheap");  
        System.out.println(Arrays.toString(attributes));  
    }  
}
```

```
Exception in thread "main" java.lang.ClassCastException: class [Ljava.lang.Object; can  
not be cast to class [Ljava.lang.String; ([Ljava.lang.Object; and [Ljava.lang.String;
```



```
are in module java.base of loader 'bootstrap')
at effectivejava.chapter5.item32.PickTwo.main(PickTwo.java:23)
```

Object[] 인스턴스를 String[] 타입으로 캐스팅 할 수 없어서 ClassCastException 이 발생하게 된다.

toArray() 메서드를 호출 하면 가변인자들을 담고 있는 메서드가 필요함

→ T 타입이기 때문에 어떤 놈이던 담을 수 있는 Object[] 로 만들어 리턴함

→ pickTwo("Good", "Fast", "Cheap") 앞에 컴파일러는 Object[] 코드를 추가하여 캐스팅함

→ Object[] 를 String[] 에 넣을 수 없다

→ ✗ ClassCastException

위의 메서드를 type-safe 한 메서드로 만들어보자

pickTwo() 에서 type-safe 하지 못한 toArray() 메서드를 사용함으로써 힙 오염이 전파되었다.

가변인자를 받아올 때 사용되는 배열을 바로 사용하지 말고

가변인자들을 List 로 묶어서 사용하는게 이에 대한 해결책이 될 수 있다.

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

```
public class SafePickTwo {
    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);
    }
    throw new AssertionError();
}

public static void main(String[] args) {
    List<String> attributes = pickTwo("Good", "Fast", "Cheap");
    System.out.println(attributes);
}
}
```