Combine generics and varargs judiciously

varargs와 generics가 Java5에서 동시에 도입되었습니다. 그래서 두 개가 잘 어울릴 것 같으나, 잘 맞지 않습니다.

varargs 문제

varargs는 client가 method의 argument의 수를 조절할 수 있도록 합니다.

문제점

추상화가 잘못되었죠. argument를 넘기고 내부적으로는 varargs를 받는 array가 생성됩니다. 이 상세 구현이 노출되었습니다.

그래서 varargs가 generic이나 parameterized types을 받으면 compiler warning이 발생해서 혼동을 줄 수 있습니다.

method를 non-refiable type으로 선언하면 compiler가 warning을 발생시킵니다. 더불어 method를 invocation할 때도 추론된 타입이 non-reifiable이라면 또 compiler가 warning을 발생시키죠.

경고문

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

heap pollution은 parameterized type의 변수가 자신의 타입이 아닌 것을 참조할 때 발생할 수 있습니다. 이는 compiler가 casting을 실패했을 때 발생하며, 근본적으로 compiler의 type safety를 저해하는 요소라할 수 있죠.

예시

```
// Mixing generics and varargs can violate type safety!
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
}
```

그럼에도 non-refiable을 허용한다!

생각해보면 위 예제는 Item-28에서 봤던 예제와 비슷합니다. non-refiable array를 선언하는 형태죠.

아래 예제는 Item-28에서 봤던 예제입니다.

```
// Why generic array creation is illegal - won't compile!
List<String>[] stringLists = new List<String>[1];
List<Integer> intList = List.of(42);
Object[] objects = stringLists;
objects[0] = intList;
String s = stringLists[0].get(0);
```

그런데 이번 아이템의 예제는 컴파일이 되고 Item-28 예제는 컴파일이 안됩니다! 똑같이 array에 non-reifiable type을 담는데도 불구하구요.

다시말해, array 생성시엔 안되고 varargs를 담는 데는 허용합니다.

그 이유는 유용성 때문에 이 모순을 허용했다고 합니다.

그래서 Java library엔 위를 구현한 메서드들을 찾아볼 수 있습니다.

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

도입부에 소개했던 예제와 반대로 메서드들은 type safety하게 구현해놨죠.

Type safety

Java 7이전에는 method 선언시에 `@SuppressWarnings("unchecked")` 를 이용하여 경고를 잡을 수 있을 지언정 client에서 호출시 warning이 발생하는 것을 막을 수 없었습니다.

그래서 메서드를 사용하려면 찝찝하게 동일한 어노테이션을 달아주어야만 했죠.

하지만 Java 7에서 SafeVarargs annotation이 도입되면서 method 작성자가 type safety하다는 것을 보장할 수 있게 되었습니다.

언제 타입 안전하다고 보장할 수 있을까?

generic varargs를 담기 위해 array가 생성됐을 때를 생각하시면 됩니다.

1. array에 무엇도 저장하면 안되며,(덮어쓰기 금지)
2. array의 참조가 외부로 노출되서는 안됩니다.

이 두가지를 지키면 type safety하다 할 수 있습니다. 간략히 말하면 단순히 arguments를 전달해주는 varargs용도 그대로만 사용하라는 뜻이에요.

위 규약을 어긴 예

array에 아무것도 저장하지 않았지만, 참조를 외부에 노출시켜서 문제가 되는 예를 보여드리겠습니다.

```
// UNSAFE - Exposes a reference to its generic parameter array!
static <T> T[] toArray(T... args) {
  return args;
}
```

딱 봤을 때 문제 없어보이죠?

하지만 method호출해서 넘겨줄 때 type이 결정되므로 compile-time에 이 array의 type을 정확히 결정할 수 없습니다. 그럼에도 이 method는 varargs parameter array를 반환하기 때문에 call stack에서 heap pollution을 전파할 수 있습니다.

구체적인 예 - pickTwo

pickTwo method는 3가지 parameter를 받습니다. 그리고 임의로 2개의 원소를 toArray method를 사용해 배열로 반환해주는 예제입니다.

```
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
   }
```

toArray에서 컴파일러는 타입을 결정할 수 없으니까 varargs array를 Object[]로 생성합니다. 그래서 두 값을 받아 반환할 때 Object[]로 반환하죠.

```
public static void main(String[] args) {
  String[] attributes = pickTwo("Good", "Fast", "Cheap");
}
```

main method를 살펴봅시다.

pickTwo는 Object[]를 반환하죠. 이 배열을 참조할 때 attributes가 String[] 타입이므로 String[]으로 캐스팅하겠죠?

그런데 Object[]는 String[]의 서브타입이 아니므로 casting에 실패합니다.

잘 사용한 예 - flatten

flatten method는 임의 크기의 lists를 받아 노출 없이 소비만 됩니다. 즉, 오로지 값을 전달 해주는 역할만 하는 것이죠.

```
@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;
}
```

따라서 TypeSafety하다고 보장할 수 있으므로 SafeVarargs annotation이 작성되었습니다.

그래서 선언시, 호출시 경고가 발생하지 않죠.

SafeVarargs와 상속

추가로, `@SafeVarargs` 가 붙여진 메서드는 오버라이딩되면 Type safety하다고 보장하기 어렵겠죠?

그래서 Java 8에선 static method, final instance method만 이 어노테이션을 허용하고, Java 9에선 private method까지 허용했습니다.

SafeVarargs 대안

굳이 varargs로 받지말고, 애당초 List type으로 받아주면 됩니다. 그래서 flatten을 수정하면 다음과 같죠.

```
// List as a typesafe alternative to a generic varargs parameter
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;
}
```

그리고 이 메서드는 List.of() static factory method를 연결시키는데 사용될 수 있습니다. List.of는 varargs가 선언되어 있는데 `@SafeVarargs`가 달려있기 때문에 문제없죠.

```
audience = flatten(List.of(friends, romans, countrymen));
```

이 메서드의 장점은

1. typeSafety하다고 보장할 수 있습니다.
2. 지저분하게 SafeVarargs를 붙일 필요가 없죠
3. typeSafety를 제대로 판단했는지 걱정할 필요도 없습니다.

단점은

1. 클라이언트 코드에서 가독성이 떨어지고
2. 조금 더 속도가 느릴 수 있습니다.

List.of 사용

varargs method를 안전하게 사용이 불가능한 상황(앞서 본 toArray)에서도 이 트릭은 사용될 수 있습니다.

```
static <T> List<T> pickTwo(T a, T b, T c) {
  switch(rnd.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();
}
```

main method

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