

이펙티브 자바

Item 32 ~ 33

2024/03/05

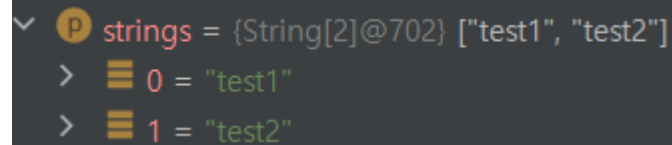
Item 32 제네릭과 가변인수를 함께 쓸 때는 신중하라

가변인자: 자바 5에 추가된 매개변수의 개수를 동적으로 지정

오버로딩의 수고를 줄임

가변인자 내부적으로 배열(Object[])을 생성해서 사용

```
public void testArgs(String...strings){  
    for (String string : strings) {  
        System.out.println("string = " + string);  
    }  
}
```



A screenshot from a Java IDE showing a variable declaration and its contents. The variable `strings` is of type `String[]` and holds the values `"test1"` and `"test2"`. Below the declaration, the array elements are indexed: `0 = "test1"` and `1 = "test2"`.

```
✓ p strings = {String[2]@702} ["test1", "test2"]  
> 0 = "test1"  
> 1 = "test2"
```

Item 32 제네릭과 가변인수를 함께 쓸 때는 신중하라

- 가변인자는 마지막 매개변수가 되어야 한다.
- 해당 변수의 종료지점을 알 수 없기 때문
- 하나의 가변인자만 사용할 수 있다.

```
public void testArgs(int ...arr,String...strings){
    for (String string : strings) {
        System.out.println("string = " + string);
    }
}

public void testArgs(String...strings,int arr){
    for (String string : strings) {
        System.out.println("string = " + string);
    }
}
```

Item 32 제네릭과 가변인수를 함께 쓸 때는 신중하라

- 제네릭은 실체화 불가 타입이다.
- 런타임에 컴파일보다 타입 관련정보가 적다
- 배열은 공변, 제네릭은 불공변

Item 32 제네릭과 가변인수를 함께 쓸 때는 신중하라

- 제네릭 가변인자 배열은 타입 안정성을 해칠 수 있다.

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

- ClassCastException 발생

Item 32 제네릭과 가변인수를 함께 쓸 때는 신중하라

- 제네릭 배열은 금지되어 있다. 제네릭 가변인수는 경고수준
- 파라미터로는 사용가능

```
public void testArgs(List<String>...lists){  
    //something  
}
```

```
public static void testArgs(List<String>lists[]){  
    //something  
}
```

Item 32 제네릭과 가변인수를 함께 쓸 때는 신중하라

- 제네릭 가변인수의 유용성
- 타입안정성이 보장되는 경우 사용할 수 있음
- `Arrays.asList(T.. A)`
- `Collections.addAll(Collection<? super T> c, T... elements)`
- `EnumSet.of(E first, E...rest)`

Item 32 제네릭과 가변인수를 함께 쓸 때는 신중하라

```
List<String[]>list=new ArrayList<>();  
list.add(new String[]{"1","2","3"});  
list.add(new String[]{"4","5"});
```

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

1. `List<List<String[]>> list3 = Arrays.asList(list);`
2. `List<String[]> list3 = Arrays.asList(list);`
3. `List<String> list3 = Arrays.asList(list);`

Item 32 제네릭과 가변인수를 함께 쓸 때는 신중하라

```
List<String[]> list3 = Arrays.asList(new String[]{"1", "2", "3"}, new String[]{"1", "2", "3"});
```

```
List<String> list3 = Arrays.asList(new String[]{"1", "2", "3"}, new String[]{"1", "2", "3"});
```

- 자바7 부터 @SafeVarargs
- 경고를 제거하고 타입 안전을 보장한다.
- 컴파일타임에서 안정성을 검사한다. (final)

Item 32 제네릭과 가변인수를 함께 쓸 때는 신중하라

- 제네릭을 가변인수로 받는 모든 메서드에 @SafeVarargs를 선언할 것
- 그렇지 않으면 해당 메서드를 사용하지 말 것
- @SafeVarage를 사용하지 않는다면 List으로 해결해볼것

```
@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;
}
```

```
public 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;
}
```

Item 33 타입 안전 이중 컨테이너를 고려하라

- 컨테이너에 매개변수화 할 수 있는 타입의 수는 제한적이다.

```
public class Favorites {
```

- class의 클래스는 제네릭이다

```
    private Map<Class<?>, Object> map = new HashMap<>();
```

```
    public <T> void put(Class<T> clazz, T value) {  
        this.map.put(Objects.requireNonNull(clazz), value);  
    }
```

```
    public <T> T get(Class<T> clazz) {  
        return clazz.cast(this.map.get(clazz));  
    }
```

```
    public static void main(String[] args) {  
        Favorites favorites = new Favorites();  
        favorites.put(String.class, "pizza");  
        favorites.put(Integer.class, 2);  
    }
```

```
}
```

Item 33 타입 안전 이중 컨테이너를 고려하라

- Class 클래스
- 모든 클래스와 인터페이스는 컴파일 후 class 파일로 생성(메타데이터로 관리)
- Class 클래스에서 class 파일에 접근가능
- `Class<String> String.class`와 같은 리터럴을 타입 토큰이라함

Item 33 타입 안전 이중 컨테이너를 고려하라

- Favorite 인스턴스는 타입 안전
- 여러가지 타입이 원소가 될 수 있음
- 타입 안전 이중 컨테이너

```
public class Favorites {  
    private Map<Class<?>, Object> map = new HashMap<>();  
  
    public <T> void put(Class<T> clazz, T value) {  
        this.map.put(Objects.requireNonNull(clazz), value);  
    }  
  
    public <T> T get(Class<T> clazz) {  
        return clazz.cast(this.map.get(clazz));  
    }  
  
    public static void main(String[] args) {  
        Favorites favorites = new Favorites();  
        favorites.put(String.class, "pizza");  
        favorites.put(Integer.class, 2);  
    }  
}
```

Item 33 타입 안전 이중 컨테이너를 고려하라

- 키와 값 사이 타입 관계를 보증하지 않아 모든 값이 키로 명시한 타입임이 보증되지 않는다.

- Get메소드에서 관계를 회복한다.

```
public class Favorites {  
  
    private Map<Class<?>, Object> map = new HashMap<>();  
  
    public <T> void put(Class<T> clazz, T value) {  
        this.map.put(Objects.requireNonNull(clazz), value);  
    }  
  
    public <T> T get(Class<T> clazz) {  
        return clazz.cast(this.map.get(clazz));  
    }  
  
    public static void main(String[] args) {  
        Favorites favorites = new Favorites();  
        favorites.put(String.class, "pizza");  
        favorites.put(Integer.class, 2);  
    }  
}
```

Item 33 타입 안전 이중 컨테이너를 고려하라

- 로타입으로 Class 객체를 넘기는 경우 문제발생(애러를 못잡음)

- List<Integer>.class 불가

```
favorites.put(List.class, List.of(1, 2, 3));  
favorites.put(List.class, List.of("a", "b", "c"));
```

- 런타임에 확인할 수 없음

```
public<T> void put(Class clazz, T value) {  
    this.map.put(Objects.requireNonNull(clazz), clazz.cast(value));  
}
```

- 런타임에 타입 확인

Item 33 타입 안전 이중 컨테이너를 고려하라

- `List<Integer>.class`와 같이 실체화 불가 타입을 사용하려면?
- 슈퍼타입 토큰
- 한정적 타입 토큰(와일드 카드)

Item 33 타입 안전 이중 컨테이너를 고려하라

- 슈퍼타입 토큰
- 상속을 받는 경우 타입정보가 런타임에 유지됨
- 익명 클래스를 만들어 제네릭 타입을 기억한다.

Item 33 타입 안전 이중 컨테이너를 고려하라

- 한정적 타입 토큰(와일드 카드)
- 비한정적 타입 토큰(와일드 카드)를 애노테이션 서브타입이라고 가정
- Annotation 클래스의 하위 타입으로 변환(해당 타입이 아닐 경우 null)

```
Class<?> annotationType = null;
```

```
annotationType.asSubclass(Annotation.class)
```