

Effective Java

Generic 기초

Item 26 : 로 타입은 사용하지 말라

Item 27 : 비검사 경고를 제거하라

Item 28 : 배열보다는 리스트를 사용하라

Generic 기초

제네릭이란?

- Java 5부터 제네릭(Generic) 타입이 추가됨
- 컴파일 과정에서 타입 체크를 해준다.
 - 이후 소거(erasure)

```
public class className<T> {...}  
public interface interfaceName<T> {...}  
public <T> T genericMethod(T o) {...}
```

제네릭의 장점

1. 컴파일 시 강한 타입 체크
2. 타입 변환(casting)을 제거

```
List<String> stringList = new ArrayList<String>();  
stringList.add(1);           // 에러  
stringList.add(3.14);        // 에러  
stringList.add("건국대");    // 정상
```

```
String str = stringList.get(0);
```

다양한 타입 매개변수

T – Type

E – Element

K – Key

V – Value

N – Number

S, U, V – 2nd, 3rd, 4th types

단지 관례이므로, 자신이 원하는 대로 표기하여 사용해도 문제되지 않음

제네릭 타입(class<T>, interface<T>)

```
public class Box<T> {  
    private T data;  
  
    public T get() {  
        return data;  
    }  
  
    public void set(T data) {  
        this.data = data;  
    }  
}
```

자바 7부터 다이아몬드 연산자를 지원

```
Box<String> box = new Box<>();  
box.set("Hello");  
String str = box.get();
```

멀티 탑 매개변수

```
public class Box<T, E> {  
    private T data1;  
    private E data2;  
  
    public T getData1() { return data1; }  
  
    public E getData2() { return data2; }  
  
    public void setData1(T data1) { this.data1 = data1; }  
  
    public void setData2(E data2) { this.data2 = data2; }  
}
```

제네릭 메소드(<T, R> R method(T t))

주로 static 메서드에 사용됨

public <타입파라미터, ...> 리턴타입 메소드명(매개변수, ...) { ... }

선언

```
public <T> void test1(T data1) {...}
```

```
public <T, E> E test2(T data1) {...}
```

```
public <T, S, E> E test3(T data1, S data2) {...}
```

사용

```
Box<Integer> box = <Integer>boxing(100); // 명시적 지정  
Box<Integer> box = boxing(100);
```

제네릭 메소드(<T, R> R method(T t))

제네릭 클래스와 타입 문자가 겹치면, 제네릭 메소드가 우선이다.
하지만 일반적으로 혼란을 줄이기 위해 **서로 다른 문자 사용** 권장

```
public class Box<T> {

    public <T> void test1(T data1) {
        System.out.println(data1.getClass().getName());
    }

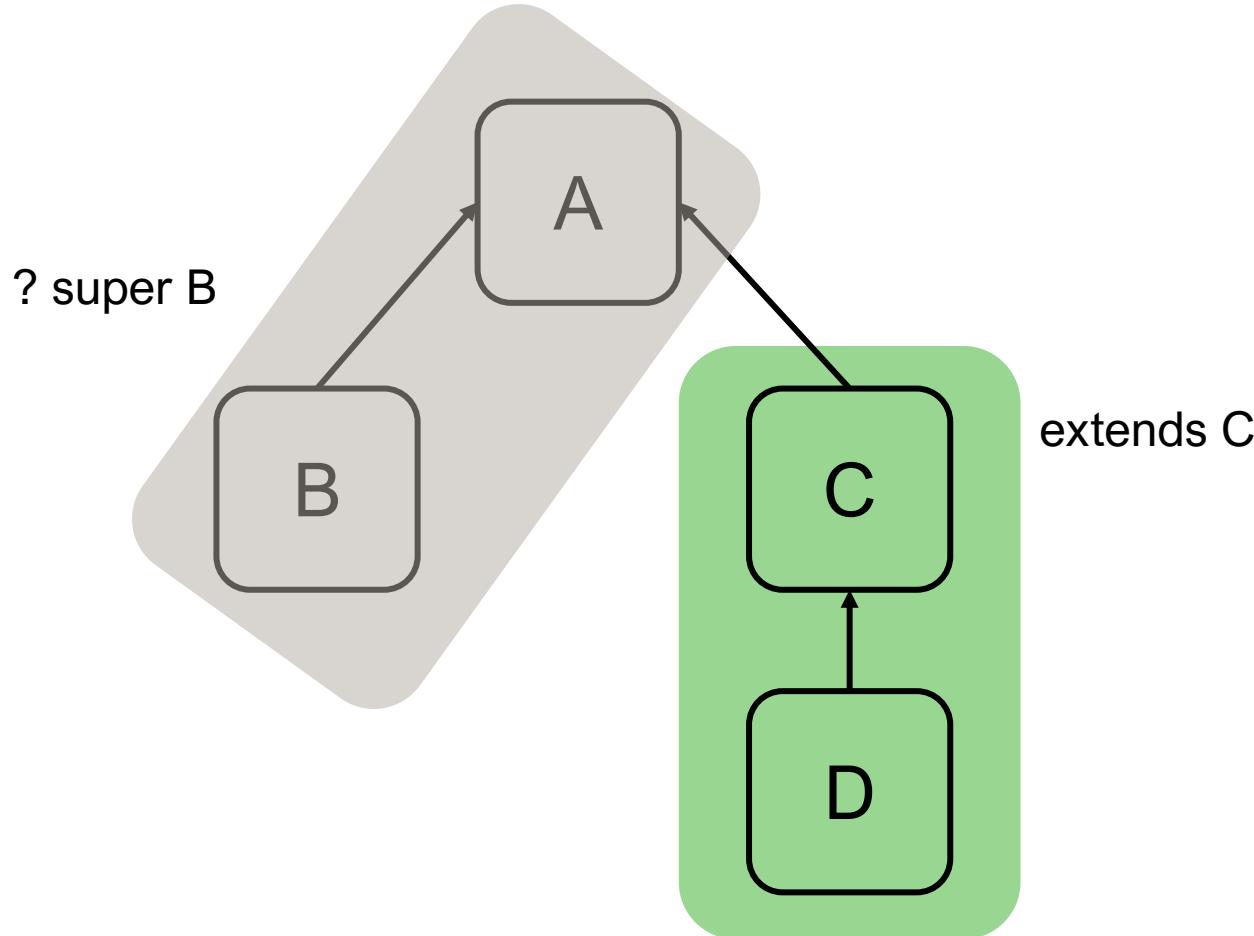
    public static void main(String[] args) {
        Box<Integer> box = new Box<>();
        box.test1("Hello");
    }
}
```

java.lang.String

종료 코드 0(으)로 완료된 프로세스

제한된 타입 매개변수

`T extends {타입}` : T는 명시된 타입과 그 **하위 타입**만 가능하다.
`? super {타입}` : T는 명시된 타입과 그 **상위 타입**만 가능하다.



CF) `<T super {타입}>` 은 없다

제한된 타입 파라미터

```
public <T> T foo(List<T> list) { }
```

제네릭 메서드에서 전달되는 타입 매개변수의 범위를 제한하고 싶다면?

```
public <T> T foo(List<T extends Fruit> list) { } X
```

```
public <T extends Fruit> T foo(List<T> list) { } O
```

제한된 타입 파라미터

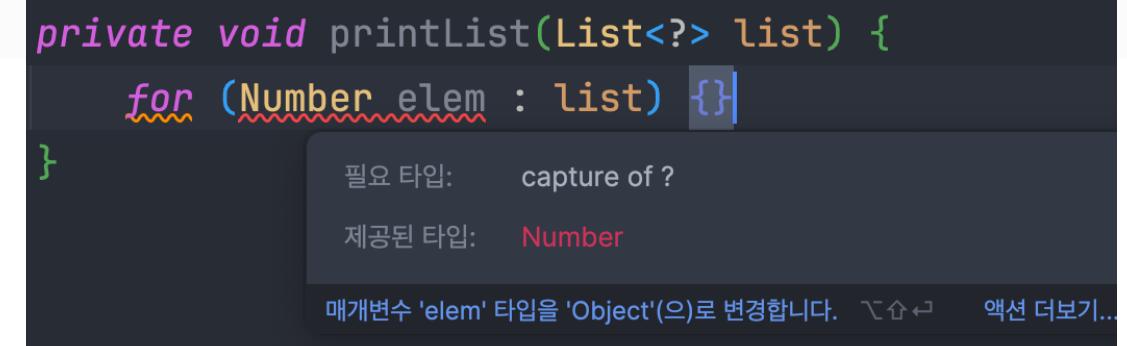
```

public static <T extends Number, E, K extends Number> K test1(List<T> data1, E data2) {}

public <T extends Number> void printList(List<? extends T> list){
    for (Number num : list) {}
}

public void printElemAndSum(List<? extends Number> list){
    for (Number num : list) {}
}

```



Item 26

로 타입은 사용하지 말라

로 타입(raw type)

제네릭 타입을 정의하면 자동으로 로 타입(raw type)도 함께 정의된다.

```
public static void main(String[] args) {
    System.out.println("Hello World!");
    List<String> list = new ArrayList<>();
    unsafeAdd(list, Integer.valueOf(42));
    String s = list.get(0);
}
                                         ClassCastException
private static void unsafeAdd(List list, Object o) {
    list.add(o);
}
```

런타임 에러

```
public static void main(String[] args) {
    System.out.println("Hello World!");
    List<String> list = new ArrayList<>();
    unsafeAdd(list, Integer.valueOf(42));
    String s = list.get(0);
}

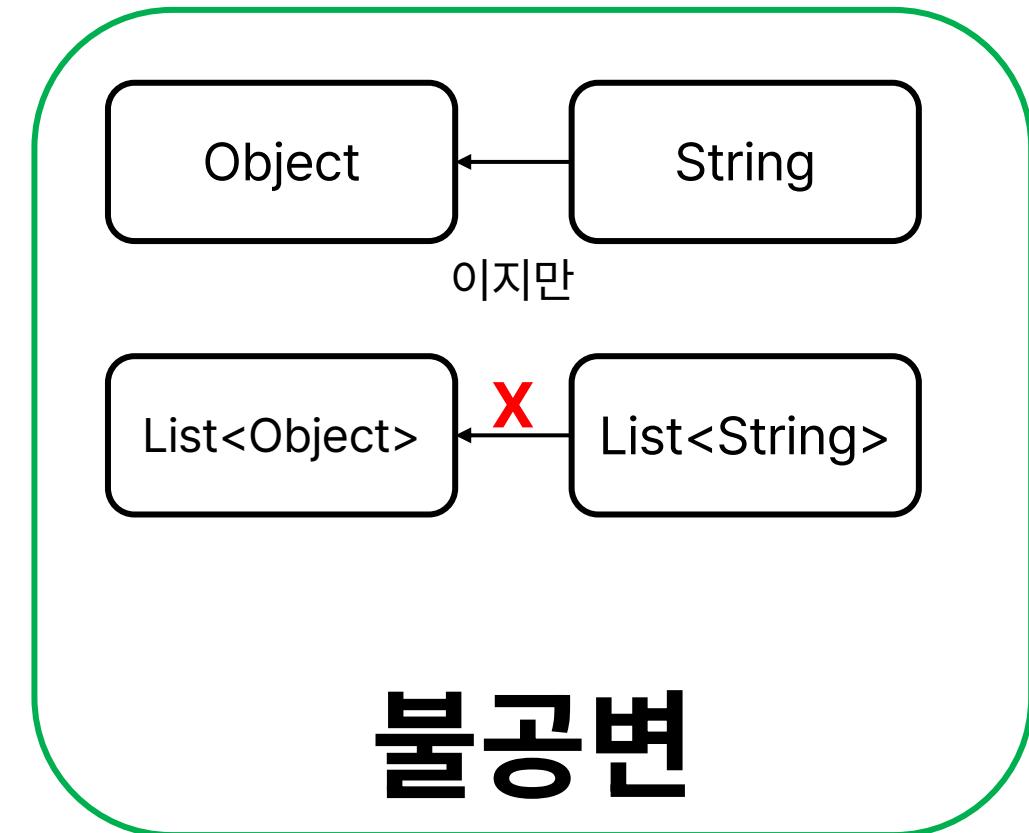
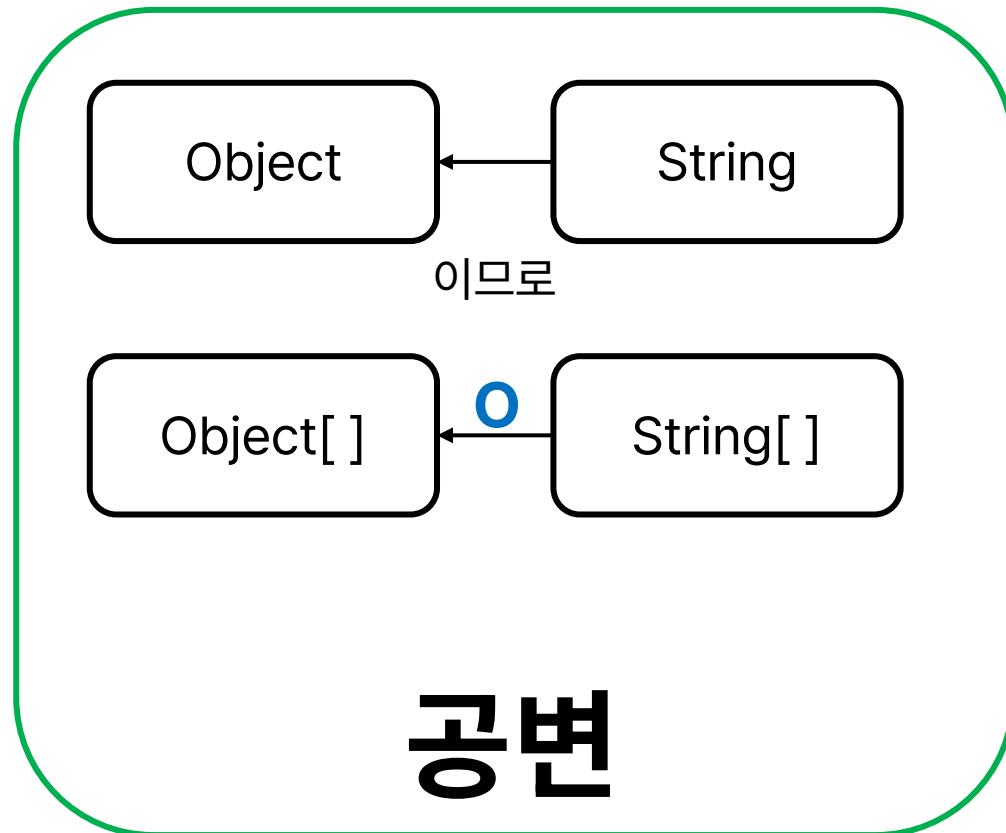
private static void unsafeAdd(List<Object> list, Object o) {
    list.add(o);
}
```

컴파일 에러

가변성

공변(covariant) : A가 B의 하위 타입일 때, T <A> 가 T의 하위 타입이면 T는 공변

불공변(invariant) : A가 B의 하위 타입일 때, T <A> 가 T의 하위 타입이 아니면 T는 불공변



Quiz

```

public static void main(String[] args) {
    List<Integer> list = List.of(1, 2, 3);
    printList(list);
}

private static <E> void printList(List<E> list) {
    for (Object integer : list) {
        System.out.println(integer);
    }
}

```

```

public static void main(String[] args) {
    List<Integer> list = List.of(1, 2, 3);
    printList(list);
}

private static void printList(List<?> list) {
    for (Object integer : list) {
        System.out.println(integer);
    }
}

```

컴파일 에러

```

public static void main(String[] args) {
    List<Integer> list = List.of(1, 2, 3);
    printList(list);
}

private static void printList(List<Object> list) {
    for (Object integer : list) {
        System.out.println(integer);
    }
}

```

```

public static void main(String[] args) {
    List<Integer> list = List.of(1, 2, 3);
    printList(list);
}

private static void printList(List list) {
    for (Object integer : list) {
        System.out.println(integer);
    }
}

```

와일드카드

모든 타입을 대신할 수 있는 와일드카드 타입(<?>)

타입이 정해지지 않은 **unknown type**이다. (**any type**이 아니다)

목적 : 하나의 참조변수로 서로 다른 타입이 대입된 여러 제네릭 객체를 다루기 위해

```
List<?> list = new ArrayList<Integer>();
```

.add() 는 타입체크를 수행한다. – unknown이므로 불가능

```
list.add(1); // 컴파일 에러
```

```
Object o = list.get(0); // 정상동작
```

어떤 타입의 자식인지 중요하지 않고, 심지어 적어도 Object 타입임을 보장

와일드카드 제한

<? extends T> 상한 제한, T와 그 자손들만 가능

<? super T> 하한 제한, T와 그 조상들만 가능

<?> 제한 없음. 모든 타입이 가능. **<? extends Object>**와 동일

```
ArrayList<Fruit> list = new ArrayList<Apple>(); // 에러
```

```
ArrayList<? extends Fruit> list = new ArrayList<Apple>();  
ArrayList<? extends Fruit> list = new ArrayList<Orange>(); // OK
```

```
void makeJuice(FruitBox<? extends Fruit> box) {  
    ...  
}
```

```
makeJuice(new FruitBox<Fruit>());  
makeJuice(new FruitBox<Apple>());
```

결론

로 타입은 안전하지 않다. 대신 제네릭이나 와일드카드를 사용하자

로 타입을 사용하는 상황

1. class 리터럴

List.class, String[].class, int.class

2. instanceof

```
if (o instanceof Set) {      // 로 타입
    Set<?> s = (Set<?>) o; // 와일드카드 타입
}
```

Item 27

비검사 경고를 제거하라

컴파일러 경고

컴파일 경고는 중요하므로, 가능한 경고를 제거하라

```
public static void main(String[] args) {
    List<String> list = new ArrayList();
}
```



```
public static void main(String[] args) {
    @SuppressWarnings("unchecked")
    List<String> list = new ArrayList();
}
```

경고를 없앨 방법을 찾지 못하겠다면, 애너테이션으로 경고를 숨겨라

```
▶ javac Box.java -Xlint:unchecked
Box.java:18: warning: [unchecked] unchecked conversion
      List<String> list = new ArrayList();
                           ^
required: List<String>
found:    ArrayList
1 warning
```

@SuppressWarnings

@SuppressWarnings("unused") : 사용하지 않는 코드

@SuppressWarnings("deprecation") : 권장되지 않는 기능

@SuppressWarnings("null") : null 분석

@SuppressWarnings("unchecked") : 미확인 오퍼레이션

@SuppressWarnings("all") : 모든 경고

@SuppressWarnings({"unused", "null"}) : 여러 종류 사용

[IBM 문서] SuppressWarnings 어노테이션 내부에서 사용할 수 있는 토큰 목록

@SuppressWarnings

가능한 좁은 범위에서 사용하자

클래스 전체, 한 줄이 넘는 메서드나 생성자에 사용 금지
지역변수로 분리하여 선언하자

```
public <T> T[] toArray(T[] a) {
    if (a.length < size) {
        // 생성한 배열과 매개변수로 받은 배열의 타입이 모두 T[]로 같으므로
        // 올바른 형변환이다.
        @SuppressWarnings("unchecked") + 경고 무시 이유를 주석으로 알려주기
        T[] result = (T[]) Arrays.copyOf(elements, size, a.getClass());
        return result;
    }
    System.arraycopy(a, 0, a, 0, size);
    if (a.length > size) a[size] = null;
    return a;
}
```

결론

비검사 경고는 런타임에 ClassCastException을 발생시킬 잠재적 가능성을 뜻한다.
따라서 최선을 다해 제거하라

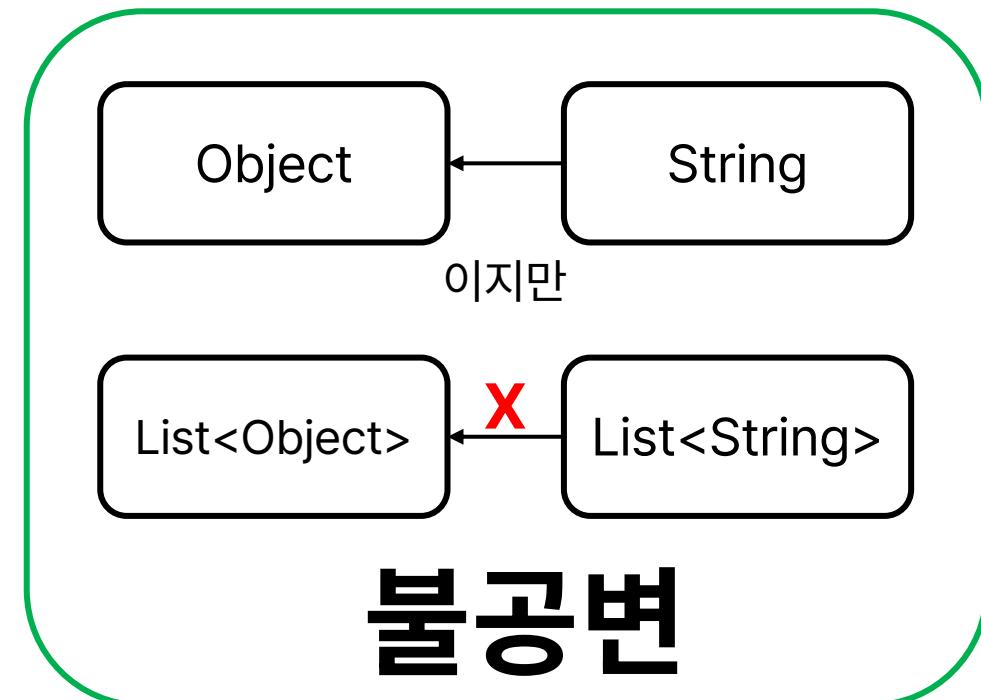
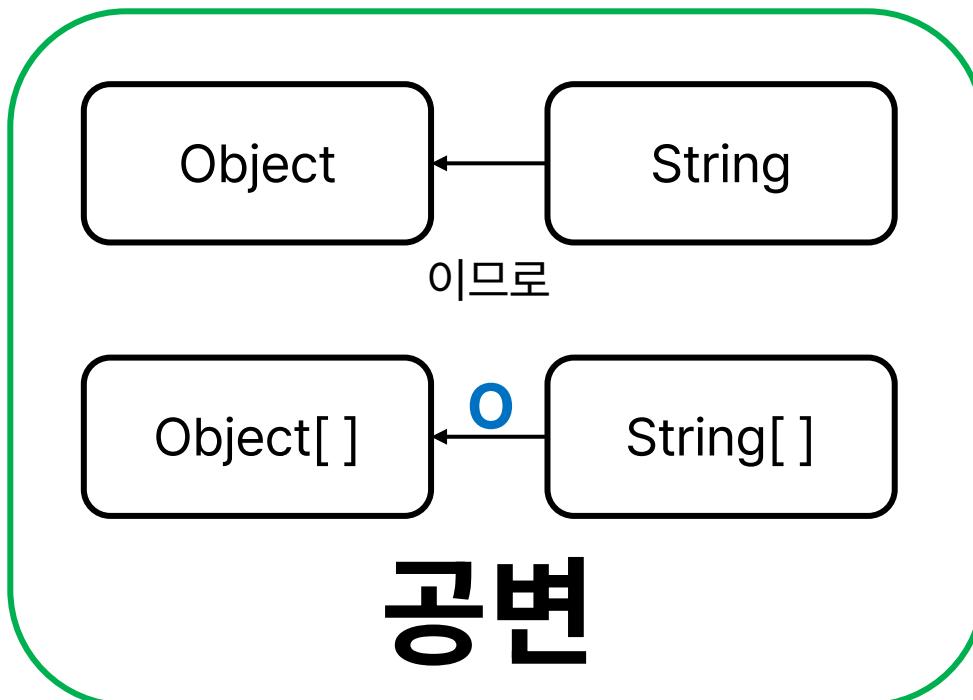
만일 경고를 없앨 방법을 찾지 못하겠다면
안전함을 증명하고 @SuppressWarnings로 경고를 숨겨라(+주석)

Item 28

배열보다는 리스트를 사용하라

배열과 제네릭 타입

배열	제네릭
공변(covariant)	불공변(invariant)
런타임에서 타입 인지	런타임에서 타입 소거



배열과 제네릭 타입

배열	제네릭
공변(covariant)	불공변(invariant)
런타임에서 타입 인지	런타임에서 타입 소거

```
Object[] objectArray = new Long[1];
objectArray[0] = "ArrayStoreException"
```

런타임 에러

```
List<Object> ol = new ArrayList<Long>();
ol.add("Compile error")
```

컴파일 에러

제네릭 배열 생성 오류

new List<E>[], new List<String>[], new E[]: 컴파일 에러

```
List<String>[] lists = new List<String>[3];
```

제네릭 배열 생성

```
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); ← ??
```

실체화 불가 타입(non-reifiable type)

E, List<E>, List<String> : 런타임에는 컴파일타임보다 타입 정보를 적게 가짐(소거)

매개변수화 타입 가운데 실체화될 수 있는 타입은 비한정적 와일드카드 타입뿐이다.

List<?>, Map<?, ?>

배열을 비한정적 와일드카드로 만들 수는 있지만, 유용하게 쓰일 일이 거의 없다.

배열보다는 컬렉션을 사용하자

코드 복잡성과 성능을 조금 포기하고, 타입 안정성을 챙기자

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

    public Chooser(final Object[] choiceArray) {
        this.choiceArray = choiceArray;
    }

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

return 데이터 형변환 필요
(런타임 에러가 발생할 여지 존재)

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

    public ListChooser(final Collection<T> choices) {
        this.choiceList = new ArrayList<>(choices);
    }

    public T choose(){
        Random random = ThreadLocalRandom.current();
        return choiceList[random.nextInt(choiceList.size())];
    }
}
```

타입 안정성 확보
(코드 양 증가, 속도 감소)