

# 이펙티브 자바 CP.4

⌚ 작성 일시	@2023년 1월 27일 오후 7:25
⌚ 최종 편집 일시	@2023년 1월 29일 오후 9:04
📄 유형	이펙티브 자바
👤 작성자	
👥 참석자	

## 4 제네릭

선행 내용 (inpa님 블로그)

- 26. raw(로) 타입은 사용하지 말라
- 27. 비검사 경고를 제거하라
- 28. 배열보다는 리스트를 사용하라
- 29. 이왕이면 제네릭 타입으로 만들라
- 30. 이왕이면 제네릭 메서드로 만들라
- 31. 한정적 와일드카드를 사용해 API 유연성을 높이라
- 32. 제네릭과 가변인수를 함께 쓸 때는 신중하라
- 33. 타입 안정 이중 컨테이너를 고려하라

## 4 제네릭

### ▼ 선행 내용 (inpa님 블로그)

- [JAVA] 제네릭(Generics) 개념 & 문법 정복하기
- [JAVA] 제네릭 - 공변성 & 와일드카드 완벽 이해하기
- [JAVA] 제네릭 - 힙 오염 (Heap Pollution) 이란?
- [JAVA] 제네릭 타입 소거 컴파일 과정 알아보기
- 용어 정리
  - 클래스와 인터페이스 선언에 타입 매개변수가 쓰이면, 이를 **제네릭 클래스** 혹은 **제네릭 인터페이스**라 부른다.  
제네릭 클래스와 제네릭 인터페이스를 통틀어 **제네릭 타입**이라고 부른다.
  - 제네릭 타입은 일련의 **매개변수화 타입**을 정의한다.

- `List<String>` 은 원소의 타입이 `String` 인 리스트를 뜻하는 매개변수화 타입이다.
- 여기서 `String` 이 정규 타입 매개변수 `E` 에 해당하는 **실제 타입 매개변수**이다.
- 제네릭 타입을 하나 정의하면 그에 딸린 **로 타입** 도 함께 정의 된다.
  - 로 타입이란 제네릭 타입에서 타입 매개변수를 전혀 사용하지 않을 때를 말한다.

한글 용어	영문 용어	예	아이템
매개변수화 타입	parameterized type	<code>List&lt;String&gt;</code>	아이템 26
실제 타입 매개변수	actual type parameter	<code>String</code>	아이템 26
제네릭 타입	generic type	<code>List&lt;E&gt;</code>	아이템 26, 29
정규 타입 매개변수	formal type parameter	<code>E</code>	아이템 26
비한정적 와일드카드 타입	unbounded wildcard type	<code>List&lt;?&gt;</code>	아이템 26
로 타입	raw type	<code>List</code>	아이템 26
한정적 타입 매개변수	bounded type parameter	<code>&lt;E extends Number&gt;</code>	아이템 29
재귀적 타입 한정	recursive type bound	<code>&lt;T extends Comparable&lt;T&gt;&gt;</code>	아이템 30
한정적 와일드카드 타입	bounded wildcard type	<code>List&lt;? extends Number&gt;</code>	아이템 31
제네릭 메서드	generic method	<code>static &lt;E&gt; List&lt;E&gt; asList(E[] a)</code>	아이템 30
타입 토큰	type token	<code>String.class</code>	아이템 33

## ▼ 26. raw(로) 타입은 사용하지 말라

- raw 타입 (제네릭을 지원하지전) (jdk 1.5 이전)
  - **로 타입**은 타입 선언에서 제네릭 타입 정보가 전부 지워진 것처럼 동작하는데, 제네릭이 도래하기 전 코드와 호환되도록 하기 위한 궁여지책(야매)이라 할 수 있다.
  - raw 타입 컬렉션

```
// raw 타입
private final Collection stamps = ...;
```

```
stamps.add(new Coin(...));
// 아무런 오류 없이 컴파일되고 실행되고, 컴파일러가 "unchecked call" 경고를 내뱉는다.

for (Iterator i = stamps.iterator(); i.hasNext(); ) {
    Stamp stamp = (Stamp) i.next(); // ClassCastException 을 던진다.
    ...
}
// 위 에서 element 를 꺼내기 전에는 오류를 알지 못한다.
// 즉, 해당 코드의 오류를 런타임시에 알 수 있는 것이다.
```

## ○ 매개변수화 타입 컬렉션

```
private final Collection<Stamp> stamps = ...;
```

- 컴파일러는 `stamps` 에는 `Stamp` 의 인스턴스만 넣어야 함을 컴파일러가 인지하게 된다. 따라서 아무런 경고 없이 컴파일된다면 의도대로 동작할 것임을 보장한다.
- `stamps` 에 엉뚱한 타입의 인스턴스를 넣으려 하면 컴파일 오류가 발생하며 무엇이 잘못 되었는지 알려준다.
- 컴파일러는 컬렉션에서 원소를 꺼내는 모든 곳에 보이지 않는 형변환을 추가하여 절대 실패하지 않음을 보장한다. (컴파일러 경고를 숨기지 않음 - 아이템 27)

## ○ raw 타입을 사용하게 되면 제네릭이 안겨주는 안정성과 표현력을 모두 잃게 된다.

- raw 타입이 있는 이유는 호환성 때문이다. (제네릭 없이 짠 코드 에서 기존 코드를 모두 수용하면서, 제네릭을 맞물려 돌아가게 하기 위해서는 raw 타입을 사용하는 메서드에 매개변수화 타입의 인스턴스를 (그 반대도) 넘겨도 동작해야만 했다.)
- 마이그레이션 호환성을 위해 raw 타입을 지원하고 제네릭 구현에는 소거 (아이템 28) 방식을 사용하기로 했다.

## ○ `List`, `List<Object>` 의 차이

- `List` 는 제네릭 타입에서 완전히 발 뻗 것을 의미.
- `List<Object>` 는 모든 타입을 허용한다는 의사를 컴파일러에 명확히 전달한 것이다.
  - 매개변수로 `List` 를 받는 메서드에 `List<String>` 을 넘길 수 있지만, `List<Object>` 를 받는 메서드에는 넘길 수 없다. (제네릭의 하위 타입 규칙)
  - `List<String>` 은 raw 타입인 `List` 의 하위 타입이지만, `List<Object>` 의 하위 타입이 아니다. (아이템 28)

- `List<Object>` 같은 매개변수화 타입을 사용할 때와 달리 `List` 같은 raw 타입을 사용하면 타입 안전성을 잃게 된다.
- 런타임 실패

```
public static void main(String [] args) {
    List<String> strings = new ArrayList<>();
    unsafeAdd(strings, Integer.valueOf(42));
    String s = strings.get(0); // 컴파일러가 자동으로 형변환 코드를 넣어줌
}
private static void unsafeAdd(List list, Object o) {
    list.add(o);
}
```

- 위 코드는 컴파일 되지만 raw 타입인 `List` 를 사용해 경고가 발생된다.
  - `strings.get(0)` 의 결과를 형변환시 `ClassCastException` 을 던진다.
  - `Integer` 를 `String` 으로 변환하려 시도 함.
- 컴파일 실패

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

- 제네릭으로 변환시 - 오류 메시지를 출력되며 컴파일 조차 되지 않는다.
  - 컴파일 시점에 오류를 반환해 보다 안전.
- `List<?>`, `List` 의 차이
  - 와일드카드 타입은 안전하고, raw 타입은 안전하지 않다.
  - raw 타입 컬렉션에는 아무 원소를 넣을 수 있으니 타입 불변식을 훼손하기 쉽다.
  - 반면, 와일드카드 타입은 `null` 을 제외하고는 어떤 원소도 넣을 수 없다.
  - 와일드카드 타입을 사용하면 컴파일러가 제 역할을 할 수 있으며, 컬렉션의 불변식을 훼손하지 않게 막을 수 있다.
- raw 타입을 사용 해야 하는 상황
  - class 리터럴에는 raw 타입을 써야 한다.

- 자바 명세는 class 리터럴에 매개변수화 타입을 사용하지 못하게 했다.
- `List.class`, `String[].class`, `int.class` 허용
- `List<String>.class`, `List<?>.class` 허용 하지 않음.
- `instanceof` 연산자
  - 런타임에는 제네릭 타입 정보가 지워지므로 `instanceof` 연산자는 비한정적 타입 이외의 매개변수화 타입에는 적용 할 수 없다.
  - raw 타입이든 비한정적 와일드카드 타입이든 `instanceof` 는 완전히 똑같이 동작한다.
- 정리
  - raw 타입으로 사용하면 런타임에서 예외가 날 수 있으니, 사용하면 안된다. raw 타입은 제네릭이 도입되기 이전 코드와 호환성을 위해 제공될 뿐이다.
  - `Set<Object>` 는 어떤 타입의 객체도 저장할 수 있는 매개변수화 타입이다. `Set<?>` 는 모종의 타입 객체(`null`) 만 저장할 수 있는 와일드카드 타입이다. 그리고 이들의 raw 타입인 `Set` 은 제네릭 타입 시스템에 속하지 않는다.

## ▼ 27. 비검사 경고를 제거하라

- 제네릭 사용시 보이는 컴파일 경고
  - 비검사 형변환 경고
  - 비검사 메서드 호출 경고
  - 비검사 매개변수화 가변인수 타입 경고
  - 비검사 변환 경고
  - 제네릭에 익숙해질수록 마주치는 경고 수는 줄겠지만, 새로 작성된 제네릭 코드가 한 번에 깨끗하게 컴파일 되리라 기대하지는 말자.
- 컴파일 방법
  - javac 명령줄 인수에 `-Xlint:unchecked` 옵션을 추가한다.

```
// 이 코드를 컴파일 시킴
Set<Lark> exaltation = new HashSet();

// 컴파일 내용
____.java(file):4(Line): warning(level): [unchecked] unchecked conversion
    Set<Lark> exaltation = new HashSet();
                        ^
required: Set<Lark>
found:    HashSet
```

- 컴파일러가 알려준 대로 수정한다면, 경고는 사라진다.
- 사실 컴파일러가 알려준 타입 매개변수를 명시하지 않고, 자바 7부터 지원하는 다이아몬드 연산자(<>)만으로 해결할 수 있다.  
그러면 컴파일러가 올바른 실제 타입 매개변수 (코드에 경우 Lark)를 추론해 줌.

```
Set<Lark> exaltation = new HashSet<>();
```

- 제거하기 어려운 경고도 있으며, 앞으로 이번 장을 진행하면서 그런 예제를 볼 수 있다.
  - **할 수 있는 한 모든 비검사 경고를 제거하라. (코드 안전성이 보장됨)**
- 경고를 제거할 수는 없지만, 타입이 안전하다고 확신할 수 있다면 `@SuppressWarnings("unchecked")` 애너테이션을 달아 경고를 숨기자.
  - 단, 타입 안전함을 검증하지 않은 채 경고를 숨기면 스스로에게 잘못된 보안 인식을 심어주는 꼴이다.
  - 경고 없이 컴파일은 되겠지만, 런타임에는 여전히 `ClassCastException`을 던질 수 있다.
- `@SuppressWarnings` 애너테이션
  - 해당 애너테이션은 개별 지역변수 선언부터 클래스 전체까지 어떤 선언에도 달 수 있다. 하지만, `@SuppressWarnings` 애너테이션은 항상 가능한 한 좁은 범위에 적용하자.
    - 변수 선언, 아주 짧은 메서드, 혹은 생성자가 될 것이다.  
자칫 심각한 경고를 놓칠 수 있으니 절대로 클래스 전체로 적용해선 안 된다.
  - 한 줄이 넘는 메서드나 생성자에 달린 애너테이션을 발견하면, 지역변수 선언 쪽을 옮기자.
    - 이를 위해 지역변수를 새로 선언하는 수고를 해야 할 수 도 있지만, 그만한 값어치가 있다.
      - ex) `ArrayList`의 `toArray` 메서드

```
public <T> T[] toArray(T[] a) {
    if (a.length < size)
        return (T[]) Arrays.copyOf(elements, size, a.getClass());
    System.arraycopy(elements, 0, a, 0, size);
    if (a.length > size)
        a[size] = null;
    return a;
}
```

```
// 컴파일시
```

```
ArrayList.java:305: warning: [unchecked] unchecked cast
    return (T[]) Arrays.copyOf(elements, size, a.getClass());
                        ^
required: T[]
found:    Object[]
```

- 애너테이션은 선언에만 달 수 있기 때문에 `return` 문에는 `@SuppressWarnings` 를 다는게 불가능하다.  
메서드 전체에 달고 싶겠지만, 범위가 필요 이상으로 넓어지니 하지 않는다.
  - 그 대신 반환값을 담을 지역변수를 하나 선언하고 그 변수에 애너테이션을 달아주자.
- ex) 변경된 코드

```
public <T> T[] toArray(T[] a) {
    if (a.length < size) {
        @SuppressWarnings("unchecked")
        T[] result = (T[]) Arrays.copyOf(elements, size, a.getClass());
        return result;
    }
    System.arraycopy(elements, 0, a, 0, size);
    if (a.length > size)
        a[size] = null;
    return a;
}
```

- 이 코드는 깔끔하게 컴파일되고 비검사 경고를 숨기는 범위도 최소로 좁혔다.
  - `@SuppressWarnings("unchecked")` 애너테이션을 사용할 때면 경고를 무시해도 안전한 이유를 항상 주석으로 남겨야한다.
    - 코드의 이해에 도움이 되며,  
다른 사람이 코드를 잘 못 수정하여 타입 안정성을 잃은 상황도 줄여준다.
- 정리
    - 비검사 경고는 중요하니 무시하지 말자.
    - 모든 비검사 경고는 런타임에 `ClassCastException` 을 일으킬 잠재적 가능성을 뜻하니 최대한 제거해라.
    - 경고를 없앨 방법을 찾지 못하겠다면, 그 코드가 타입 안전함을 증명하고 가능한 한 범위를 좁혀 `@SuppressWarnings("unchecked")` 애너테이션으로 경고를 숨기고, 근거를 주석으로 남겨라.

- ▼ 28. 배열보다는 리스트를 사용하라
- ▼ 29. 이왕이면 제네릭 타입으로 만들라
- ▼ 30. 이왕이면 제네릭 메서드로 만들라
- ▼ 31. 한정적 와일드카드를 사용해 API 유연성을 높이라
- ▼ 32. 제네릭과 가변인수를 함께 쓸 때는 신중하라
- ▼ 33. 타입 안정 이중 컨테이너를 고려하라