
이펙티브 자바

item29 ~ item 31

24/02/28

아이템 29 이왕이면 제네릭 타입으로 만들라

왜 굳이 제네릭 타입?

오브젝트를 기반으로 한 Stack 클래스는 스택에서 꺼낸 객체를
형변환하는 과정에서
런타임 오류가 날 위험성이 있다.

제네릭 코드로 리팩토링 해보자.

```
<code />

public class Stack {
    private Object[] elements;
    private int size = 0;
    private static final int DEFAULT_INITIAL_CAPACITY = 16;

    public Stack() {
        elements = new Object[DEFAULT_INITIAL_CAPACITY];
    }

    public void push(Object e) {
        ensureCapacity();
        elements[size++] = e;
    }

    public Object pop() {
        if (size == 0)
            throw new EmptyStackException();
        Object result = elements[--size];
        elements[size] = null; // 다 쓴 참조 해제
        return result;
    }

    public boolean isEmpty() {
        return size == 0;
    }

    private void ensureCapacity() {
        if (elements.length == size)
            elements = Arrays.copyOf(elements, 2 * size + 1);
    }
}
```

아이템 29 이왕이면 제네릭 타입으로 만들라

리팩토링을 완료했다.
이제 안전해졌을까?

```
<code />

public class Stack {
    private E[] elements;
    private int size = 0;
    private static final int DEFAULT_INITIAL_CAPACITY = 16;

    public Stack() {
        elements = new Object[DEFAULT_INITIAL_CAPACITY];
    }

    public void push(E e) {
        ensureCapacity();
        elements[size++] = e;
    }

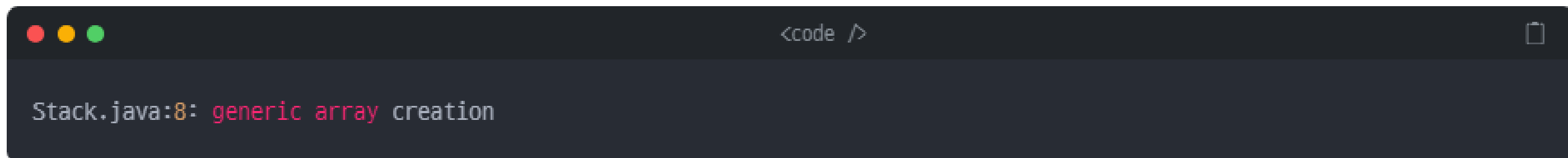
    public E pop() {
        if (size == 0)
            throw new EmptyStackException();
        E result = elements[--size];
        elements[size] = null; // 다 쓴 참조 해제
        return result;
    }

    ...
}
```

아이템 29 이왕이면 제네릭 타입으로 만들라

다음과 같은 에러가 발생한다.

바로 제네릭 같은 **실체화 불가 타입**으로는 배열을 만들 수 없다는 것이다.

A screenshot of a Java IDE window. The window has a dark theme and a title bar with three colored buttons (red, yellow, green) on the left and a '<code />' icon on the right. The main area shows a compilation error: 'Stack.java:8: generic array creation'.

```
Stack.java:8: generic array creation
```

아이템 29 이왕이면 제네릭 타입으로 만들라

첫번째 솔루션

다음 처럼 Object 배열을 생성하고 형변환하는 방식으로 생성을 한다.
이후 `@SuppressWarnings` 애너테이션으로 형변환이 안전함을 증명하자.

```
<code />

@SuppressWarnings("unchecked")
public Stack() {
    elements = (E[]) new Object[DEFAULT_INITIAL_CAPACITY];
}
```

- 가독성이 좋다.
- 코드도 짧다
- 형변환을 배열 생성 시 한 번만 해주면 된다.

아이템 29 이왕이면 제네릭 타입으로 만들라

두번째 솔루션

elements 필드의 타입만 E[]가 아닌, Object로 사용하는 것이다.

이 경우 pop 메서드에서 오류가 발생한다.

형변환을 하면 경고가 뜨는데, 다시 `@SuppressWarnings` 어노테이션으로 해결하자.

```
<code />

public E pop() {
    if (size == 0)
        throw new EmptyStackException();

    //여기서 elements의 런타임 타입은 Object[]이므로 형변환이 필요하다.
    @SuppressWarnings("unchecked")
    E result = (E) elements[--size];

    elements[size] = null; // 다 쓴 참조 해제
    return result;
}
```

아이템 29 이왕이면 제네릭 타입으로 만들라

첫번째 솔루션의 단점

E가 Object가 아닌 한 배열의 런타임 타입이 컴파일타임 타입과 달라
힙 오염을 일으킨다. (아이템32)

```
<code />  
  
@SuppressWarnings("unchecked")  
public Stack() {  
    elements = (E[]) new Object[DEFAULT_INITIAL_CAPACITY];  
}
```

아이템 29 이왕이면 제네릭 타입으로 만들라

힙 오염이란?

주로 매개변수화 타입의 변수가 타입이 다른 객체를 참조하게 되어, 힙 공간에 문제가 생기는 현상을 의미
즉 컴파일 중에 정상적으로 처리되며, 경고를 발생시키지 않고 나중에 런타임 시점에 ClassCastException이 발생하는 문제를 나타낸다.

```
@Test
@SuppressWarnings("unchecked")
void contextLoads() {
    ArrayList<String> arrayList = new ArrayList<>();

    ArrayList<Integer> arrayList2 = (ArrayList<Integer>) (Object) arrayList;
    arrayList2.add(100);
    arrayList2.add(200);

    String str = arrayList.get(0);    // Runtime Exception
```


아이템 29 이왕이면 제네릭 타입으로 만들라

왜 컴파일러는 모를까?

- 타입 캐스팅 체크는 컴파일러가 하지 않는다. 오직 대입되는 참조변수에 저장할 수 있느냐만 검사한다.
- 제네릭의 소거 특성으로 인해 컴파일시 끝난 클래스 파일의 코드에는 타입 파라미터 대신 Object 가 남아있게 된다. ArrayList<Object> 에는 String 이나 Integer 모두 넣는 것이 가능해진다.

```
@Test
@SuppressWarnings("unchecked")
void contextLoads() {
    ArrayList<String> arrayList = new ArrayList<>();

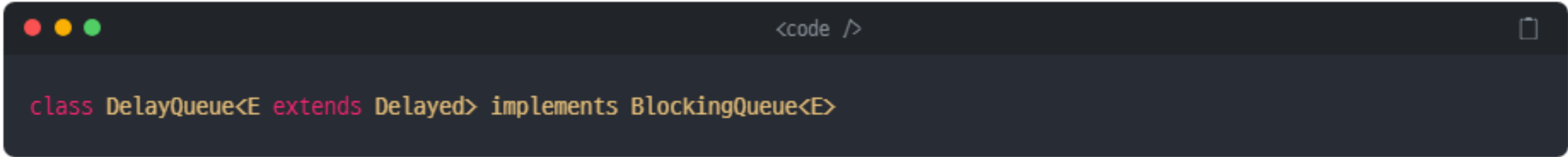
    ArrayList<Integer> arrayList2 = (ArrayList<Integer>) (Object) arrayList;
    arrayList2.add(100);
    arrayList2.add(200);

    String str = arrayList.get(0);    // Runtime Exception
```

아이템 29 이왕이면 제네릭 타입으로 만들라

한정적 타입 매개변수

Stack 처럼 대다수의 제네릭 타입은 타입 매개변수의 아무런 제약을 두지 않지만, 간혹 받을 수 있는 하위 타입에 제약이 있는 제네릭 타입도 존재한다.

A code editor window with a dark background. It has three colored window control buttons (red, yellow, green) in the top left corner. In the top right corner, there is a tab labeled "<code />" and a clipboard icon. The main area contains a single line of Java code: `class DelayQueue<E extends Delayed> implements BlockingQueue<E>`. The code is color-coded: `class` is pink, `DelayQueue` is yellow, `<E extends Delayed>` is pink, `implements` is yellow, and `BlockingQueue` is yellow.

```
class DelayQueue<E extends Delayed> implements BlockingQueue<E>
```

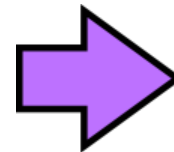
아이템 29 이왕이면 제네릭 타입으로 만들라

NOTE

결론: 기존 타입 중 제네릭이었어야 하는게 있다면 제네릭 타입으로 변경하자. 기존 클라이언트에는 아무 영향을 주지 않으면서, 새로운 사용자를 훨씬 편하게 해준다.

아이템 30 이왕이면 제네릭 메서드로 만들라

```
public static Set union (Set s1, Set s2){  
    Set result = new HashSet<>(s1);  
    result.addAll(s2);  
    return result;  
}
```



```
<code />  
  
public static <E> Set<E> union (Set<E> s1, Set<E> s2){  
    Set<E> result = new HashSet<>(s1);  
    result.addAll(s2);  
    return result;  
}
```

아이템 30 이왕이면 제네릭 메서드로 만들라

제네릭 싱글턴 팩터리

```
<code />  
  
private static UnaryOperator<Object> IDENTITY_FN = (t) -> t;  
  
@SuppressWarnings("unchecked")  
public static <T> UnaryOperator<T> identityFunction(){  
    return (UnaryOperator<T>) IDENTITY_FN;  
}
```

아이템 30 이왕이면 제네릭 메서드로 만들라


재귀적 타입 한정

재귀적 타입 한정은 자신이 들어간 표현식을 사용하여 타입 매개변수의 허용 범위를 한정하는 개념이다. 이런 재귀적 타입 한정은 주로 (거의)같은 타입의 원소와의 순서를 지정해주는 Comparable과 함께 사용된다.

```
<code />  
  
public interface Comparable<T>{  
    int compareTo(T o);  
}
```

아이템 30 이왕이면 제네릭 메서드로 만들라

결론

 **NOTE** 제네릭 타입 메서드는 타입 안전하며 사용하기 쉽다.
메서드도 타입과 마찬가지로 형변환 없이 사용할 수 있는 편이 좋다.
만약 형변환 해야 하는 메서드가 있다면 제네릭하게 만들자.

아이템 31 한정적 와일드카드를 사용해 API 유연성을 높라

왜 와일드 카드 타입을 사용해야 할까?

제네릭은 불공변이다. 즉, Type1과 Type2가 있을 때 List<Type1>은 List<Type2>의 하위 타입도 상위타입도 아니게 된다.

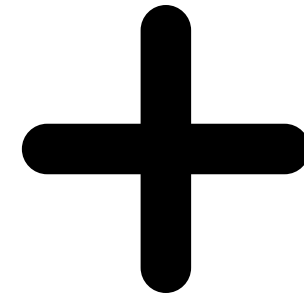
즉, List<String>은 List<Object>의 하위 타입이 아니라는 뜻이다.

List<Object>에는 어떤 객체든 넣은 수 있지만 List<String>에는 문자열만 넣을 수 있다.



아이템 31 한정적 와일드카드를 사용해 API 유연성을 높라

```
public class Stack<E> {  
    public Stack();  
    public void push(E e);  
    public E pop();  
    public boolean isEmpty();  
}
```



```
public void pushAll(Iterable<E> src) {  
    for (E e: src)  
        push(e);  
}
```

아이템 31 한정적 와일드카드를 사용해 API 유연성을 높라



Number타입의 스택에 integer는 하위타입이기 때문에
들어갈 수 있다고 생각하지만,
제네릭은 **불공변**이어서 에러가 발생한다.

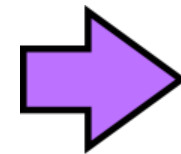
```
<code />

@Test
public void stackTest() {
    Stack<Number> stack = new Stack<>();
    List<Integer> integers = List.of(1, 2, 3);
    stack.pushAll(integers);
    System.out.println("stack = " + stack);
}
```

아이템 31 한정적 와일드카드를 사용해 API 유연성을 높라

생산자

```
public void pushAll(Iterable<E> src) {  
    for (E e: src)  
        push(e);  
}
```

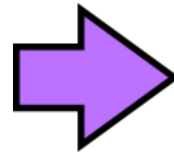


```
public void pushAll(Iterable<? extends E> src) {  
    for (E e : src) {  
        push(e);  
    }  
}
```

아이템 31 한정적 와일드카드를 사용해 API 유연성을 높라

소비자

```
public void popAll(Collection<E> dst) {  
    while(size > 0) {  
        dst.add(pop());  
    }  
}
```



```
public void popAll(Collection<? super E> dst) {  
    while(size > 0) {  
        dst.add(pop());  
    }  
}
```

아이템 31 한정적 와일드카드를 사용해 API 유연성을 높라

PECS란?

생산자와 소비자 패턴에서 제네릭의 형태상위 타입일수록 더 적은 정보를 가지고 있다.
하위 타입일수록 더 구체적인 다른 정보를 많이 가지고 있다.

생산자일 때는 하위 타입을 받아도 객체는 필요한 정보만 쓰면 되니, extends를 쓴다.
소비자일 때는 소비자의 정보를 받아줄 타입이 필요 하니, super를 쓴다.

아이템 31 한정적 와일드카드를 사용해 API 유연성을 높라

메서드 선언에 타입 매개변수가 한번만 나온다면 와일드 카드를 쓰자.

```
<code />

@Test
public void methodSignatureWildcardTest() {
    List<String> strings = new ArrayList<>(List.of("a", "b", "c", "d", "e", "f"));
    swap(strings, 0, 1);
    System.out.println("strings = " + strings);
}

public static void swap(List<?> list, int i, int j) {
    swapHelper(list, i, j);
}

private static <E> void swapHelper(List<E> list, int i, int j) {
    list.set(i, list.set(j, list.get(i)));
}
```

Tip. 그냥 ? 와일드카드 타입만 쓰면, List<?>에는 null이외의 값을 넣을 수 없다는 에러가 발생한다..

아이템 31 한정적 와일드카드를 사용해 API 유연성을 높라

NOTE

조금 복잡하더라도 와일드카드 타입을 적용하면 API가 훨씬 유연해진다. 그러니 널리 쓰일 라이브러리를 작성한다면 반드시 와일드카드 타입을 적절히 사용해줘야 한다. PECS 공식을 기억하자.