

아이템 29

이왕이면 제네릭 타입으로 만들라

```

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 Object push(Object item) {
        ensureCapacity();
        elements[size++] = item;
        return item;
    }

    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);
    }
}

```

이 클래스의 문제점은?

```
public static void main(String[] args) {  
    Stack stack = new Stack();  
    stack.push(1);  
    String item = (String) stack.pop();  
    System.out.println(item);  
}
```

ClassCastException



우선 Object를 Generic으로

```
public class Stack<E> {  
  
    private E[] elements;  
    private int size = 0;  
    private static final int DEFAULT_INITIAL_CAPACITY = 16;  
  
    public Stack() {  
        elements = new E[DEFAULT_INITIAL_CAPACITY];  
    }  
    ...  
}
```



컴파일 에러 발생!

Generic은 실체화 할 수 없다!

우회 방법 1

- 배열을 Object로 생성하고 Generic으로 형 변환

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

우회 방법 2

- elements의 타입을 Object[]로 바꾸고 pop에서 형변환

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

1과 2 방법의 차이는?

- 방법 1

- 배열을 E[]로 선언하여 E타입만 받음을 명시해준다! (타입 안전)
- 하지만 런타임에는 E[]가 아닌 Object[]로 동작
- 힙 오염의 가능성 존재(런타임에 Object[]로 동작하므로)

- 방법 2

- 애초에 Object[] 배열이므로 힙 오염의 가능성 X
- 그렇다면 pop 시 타입 보장은?
 - push로 E 만 들어오므로 Object[]에 저장되는 데이터가 모두 E 타입임이 보장
- 메서드 호출 시마다 타입 캐스팅을 해주어야 한다.

잠깐, 힙 오염(Heap Pollution)이란?

- 매개변수 유형이 다른 서로 다른 타입을 참조할 때 생기는 문제
- 컴파일은 된다. 런타임에 `ClassCastException`을 발생 시킬 뿐.
 - 예를 들어 `List<Integer> a`를 선언해 봤는데, `List<String>`으로 선언된 다른 변수 `b`가 `List<Integer>`를 참조할 경우
 - `a`는 `int` list니까 `int`를 넣고 꺼내야 함.
 - `B`는 `String` list니까 `String`을 넣고 꺼내야 함.
 - 둘의 참조가 같으므로 넣고 꺼내는 타입이 달라질 수 있음.
(`a`에 `int`를 넣고 `b`에서 꺼내면 `String`으로 꺼내야 함)

잠깐, 힙 오염(Heap Pollution)이란?

```
public static void main(String[] args) {  
    List<Integer> intList = new ArrayList<>();  
    intList.add(1);  
  
    Object object = intList;  
  
    List<String> stringList = (ArrayList<String>) object;  
  
    String data = stringList.get(0);  
    System.out.println(data);  
}
```

List<String>에서 String으로 꺼내니까 컴파일 O
하지만 실제 데이터는 int이므로 **ClassCastException**

질문: 방법 1에서 왜 런타임에서 E[]가 Object[]로 바뀌죠?

- 타입 소거(Type Erasure) 때문
 - 제네릭은 컴파일 이후 Object 타입으로 바뀐다.
 - 방법 1에서 E 배열로 형변환 해 주었지만 이는 비검사
 - 컴파일러가 검사를 마치고 Object 배열로 다시 바꿔준다.

컴파일러: 이제 당신의 제네릭입니다.

나: 뭐야 내 제네릭 돌려줘요



방법 1의 경우 확장해서 설명

```
public void invalidPush(Object item) {  
    ensureCapacity();  
    elements[size++] = (E) item;  
}
```

이런 메서드 추가

```
public static void main(String[] args) {  
    Stack<Integer> stack = new Stack<>();  
    stack.invalidPush("1");  
    int invalidItem = stack.pop();  
    System.out.println(invalidItem);  
}
```

아무 이상 없이 컴파일!

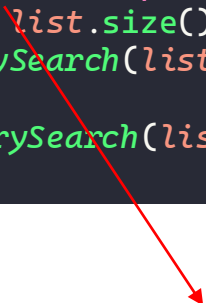
실행 결과

Exception in thread "main" java.lang.ClassCastException:
class java.lang.String cannot be cast to class java.lang.Integer (java.lang.String
and java.lang.Integer are in module java.base of loader 'bootstrap')
at Application.main(Application.java:9)

제네릭 타입 사용 시 주의점

- 타입 매개변수에 제약이 없다. 기본 타입 빼고.
 - 기본 타입 사용 시에는 박싱된 타입으로 우회 가능하다.
- 타입 매개변수에 제약을 둘 수도 있다. (bounded type parameter)

```
public static <T>
int binarySearch(List<? extends Comparable<? super T>> list, T key) {
    if (list instanceof RandomAccess || list.size() < BINARYSEARCH_THRESHOLD)
        return Collections.indexedBinarySearch(list, key);
    else
        return Collections.iteratorBinarySearch(list, key);
}
```



Comparable을 구현한 타입만 올 수 있다.
이 때 형변환 없이 Comparable의 메서드 사용 가능!

타입 매개변수 제약

- $\langle ? \text{ extends } T \rangle$ 형태 - 공변(convariance)
 - T 를 상속받은 타입이 올 수 있다.
 - 런타임에서 잘못된 형변환이 되지 않을까? -> T 로 타입 업캐스팅한 데이터는 **read-only**
- $\langle ? \text{ super } T \rangle$ 형태 - 반공변(contravariance)
 - T 의 부모 타입이 올 수 있다.
 - 런타임에서 잘못된 형변환이 되지 않을까? -> T 로 다운캐스팅한 데이터는 **write-only**