

구현 class 상세

ObjectArrayListLimitedCapacity.class

```
package org.example.ArrayListAssignment;

/**
 * 고정 용량(10)의 배열 기반 ArrayList 구현.
 * 다양한 타입의 Object를 저장할 수 있음.
 */public class ObjectArrayListLimitedCapacity {

    // 내부 저장용 배열
    protected Object[] array;

    // 현재 저장된 요소 수
    protected int size;

    // 기본 배열 크기 (고정)
    private static final int DEFAULT_CAPACITY = 10;

    /**
     * 생성자: 크기 10의 배열을 초기화하고, size는 0으로 설정
     */
    public ObjectArrayListLimitedCapacity() {
        this.array = new Object[DEFAULT_CAPACITY];
        this.size = 0;
    }

    /**
     * 현재 요소 수를 반환
     */
    public int size() {
        return this.size;
    }

    /**
     * 리스트가 비어있는지 여부 확인
     */
    public boolean isEmpty() {
        return this.size == 0;
    }

    /**
     * 주어진 인덱스에 있는 요소를 반환
     */
    public Object get(int index) {
        return this.array[index];
    }

    /**
     * 주어진 인덱스에 요소 삽입.
     * 기존 요소들은 오른쪽으로 한 칸씩 이동.
     */
    public void add(int index, Object o) {
        // 뒤에서부터 한 칸씩 밀기
        for (int i = this.size; i > index; i--) {
            array[i] = array[i - 1];
        }
    }
}
```

```

        array[index] = o;
        this.size++;
    }

    /**
     * 리스트 끝에 요소 추가
     */
    public void add(Object o) {
        array[this.size] = o;
        this.size++;
    }

    /**
     * 주어진 인덱스에 있는 요소를 제거하고 반환
     * 이후 요소들은 왼쪽으로 한 칸씩 이동
     */
    public Object remove(int index) {
        Object removedObject = this.array[index];
        // 요소들을 앞으로 한 칸씩 당김
        for (int i = index; i < this.size - 1; i++) {
            this.array[i] = this.array[i + 1];
        }
        this.array[this.size - 1] = null; // 마지막 위치 null 처리
        this.size--;
        return removedObject;
    }
}

```

◆ `public ObjectArrayListLimitedCapacity()`

- 고정된 크기(10)의 배열을 생성하고, 초기 크기를 0으로 설정하는 생성자이다.
- 내부적으로 `Object[]` 배열을 생성하여 데이터를 저장하도록 한다.
- 프로그램 시작 시 리스트가 비어 있는 상태로 초기화된다.

◆ `public int size()`

- 현재 리스트에 저장된 요소의 개수를 반환한다.
- 외부에서 현재 저장된 데이터의 수를 확인할 수 있도록 한다.

◆ `public boolean isEmpty()`

- 리스트가 비어있는지 여부를 반환한다.
- `size` 가 0일 경우 `true`, 그렇지 않으면 `false` 를 반환한다.

◆ `public Object get(int index)`

- 주어진 인덱스에 해당하는 요소를 반환한다.
- 내부 배열의 해당 위치 값을 그대로 반환하며, 데이터 접근 기능을 제공한다.

◆ `public void add(Object o)`

- 리스트의 끝에 새 요소를 추가한다.
- 현재 `size` 위치에 데이터를 삽입하고, `size` 값을 1 증가시킨다.

◆ `public void add(int index, Object o)`

- 주어진 인덱스 위치에 요소를 삽입한다.
- 삽입 이후의 요소들을 오른쪽으로 한 칸씩 이동시켜 공간을 확보한 뒤 삽입을 수행한다.

◆ `public Object remove(int index)`

- 주어진 인덱스 위치의 요소를 제거하고, 제거된 요소를 반환한다.
- 이후 요소들을 왼쪽으로 한 칸씩 이동시켜 리스트를 정렬하며, 마지막 인덱스는 `null` 로 처리하고 `size` 를 1 감소시킨다.

ObjectArrayList.class

```
package org.example.ArrayListAssignment;

/**
 * ObjectArrayListLimitedCapacity 클래스를 상속하여
 * 배열의 용량이 자동으로 확장되는 ArrayList 클래스
 */
public class ObjectArrayList extends ObjectArrayListLimitedCapacity {
    // 초기 배열 용량
    private static int CAPACITY = 10;

    /**
     * 생성자: 부모 클래스의 생성자를 호출하여 배열 초기화
     */
    public ObjectArrayList() {
        super();
    }

    /**
     * 요소를 리스트 끝에 추가함
     * 현재 크기가 용량과 같으면 배열 크기를 두 배로 확장한 뒤 추가함
     */
    public void add(Object o){
        if (size() == CAPACITY) {
            // 새로운 배열 생성 및 복사
            Object[] newArray = new Object[CAPACITY * 2];
            System.arraycopy(array, 0, newArray, 0, CAPACITY);
            array = newArray;
            CAPACITY *= 2; // 용량 업데이트
        }
        array[size()] = o; // 중복 저장되므로 사실상 필요 없는 코드
        super.add(o);      // 부모의 add를 통해 size 증가
    }

    /**
     * 현재 요소 수를 반환함
     */
    public int size() {
        return super.size();
    }

    /**
     * 주어진 객체가 리스트에 존재하는지 확인함
     * null을 만나면 탐색을 중단함
     */
    public boolean contains(Object o) {
        for (Object object : array) {
            if (object == null) {
                break;
            }
            if (object.equals(o)) {
                return true;
            }
        }
        return false;
    }
}
```

`ObjectArrayList` 는 `ObjectArrayListLimitedCapacity` 클래스를 상속받아 **배열의 자동 확장 기능**을 지원하도록 설계한 클래스이다.

기본 클래스는 고정 크기 배열만 지원했으나, 이 클래스는 **요소 수가 초기 용량(10)을 초과할 경우 배열을 두 배 크기로 복사하여** 저장 용량을 확장한다.

◆ `public ObjectArrayList()`

- 부모 클래스의 생성자를 호출하여 배열을 초기화한다.
- 별도의 동작은 없으며 상속 구조를 기반으로 생성된다.

◆ `public void add(Object o)`

- 요소를 리스트 끝에 추가한다.
- 현재 리스트의 크기가 `CAPACITY` 와 같을 경우, 배열의 크기를 두 배로 늘리고 기존 데이터를 복사한다.
- 이후 부모 클래스의 `add()` 메서드를 호출하여 데이터를 실제로 삽입한다.

◆ `public int size()`

- 현재 저장된 요소의 개수를 반환한다.
- 부모 클래스의 `size()` 메서드를 그대로 호출하여 반환한다.

◆ `public boolean contains(Object o)`

- 전달된 객체가 배열 내에 존재하는지 여부를 검사한다.
- 배열의 각 요소와 `equals()` 메서드로 비교하며, `null` 을 만나면 탐색을 중단한다.
- 리스트에 원하는 객체가 있는지 확인할 수 있는 기능을 제공한다.
-

ParaStack< T >.class

```
package org.example.ArrayListAssignment;

/**
 * 제네릭 스택 구현 클래스
 * 내부적으로 ObjectArrayList를 사용하여 데이터를 저장함
 * push/pop/isEmpty 기능을 제공함
 */
public class ParaStack<T> {

    // 내부 저장소로 사용할 리스트 (ObjectArrayList)
    private ObjectArrayList list;

    /**
     * 생성자: 내부 리스트를 초기화함
     */
    public ParaStack() {
        list = new ObjectArrayList();
    }

    /**
     * 스택에 요소를 추가(push)함
     */
    public void push(T item) {
        list.add(item);
    }
}
```

```

/**
 * 스택의 가장 위 요소를 제거하고 반환(pop)함
 * 현재 사이즈를 출력하고, 마지막 요소를 제거하여 반환함
 */
public T pop() {
    System.out.println(list.size()); // 디버깅 용도 출력
    return (T) list.remove(list.size() - 1);
}

/**
 * 스택이 비어 있는지 여부를 반환함
 */
public boolean isEmpty() {
    return list.isEmpty();
}
}

```

ParaStack<T> 클래스는 제네릭 타입을 지원하는 스택 구조를 구현한 클래스이다.

내부 저장소로는 `ObjectArrayList` 를 사용하며, `push()`, `pop()`, `isEmpty()` 메서드를 통해 스택의 기본 동작을 제공한다. `T` 를 통해 다양한 데이터 타입을 저장할 수 있도록 설계되었다.

◆ `public ParaStack()`

- 내부적으로 사용할 `ObjectArrayList` 를 초기화한다.
- 제네릭 타입과 무관하게 공통 리스트를 생성하여 데이터를 저장할 수 있도록 한다.

◆ `public void push(T item)`

- 전달된 데이터를 스택의 맨 위에 추가한다.
- 내부 리스트의 `add()` 메서드를 호출하여 구현한다.

◆ `public T pop()`

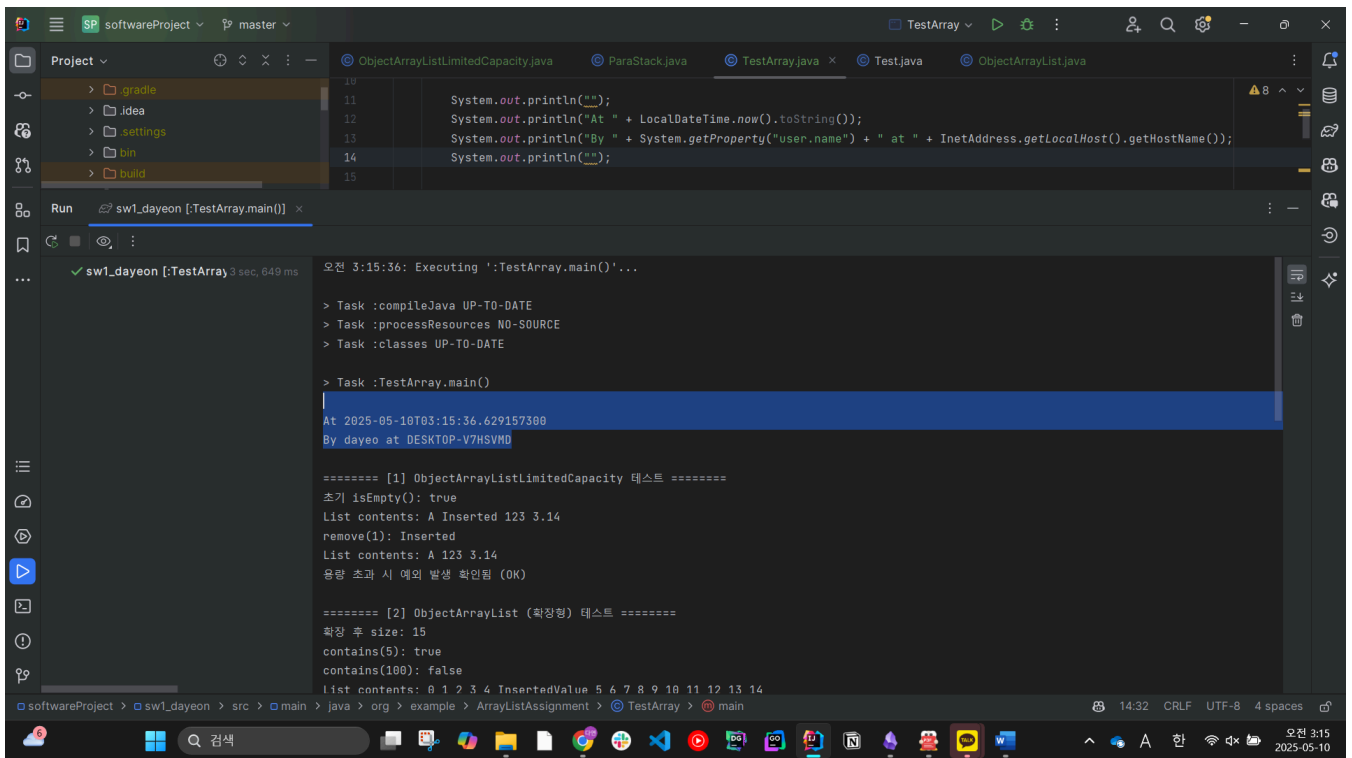
- 스택의 맨 위에 있는 요소를 제거하고 반환한다.
- 내부 리스트의 `size()` 를 활용하여 마지막 요소를 `remove()` 한다.
- 반환 시 제네릭 타입 `T` 로 캐스팅하여 반환한다.

◆ `public boolean isEmpty()`

- 스택이 비어 있는지를 검사한다.
- 내부 리스트의 `isEmpty()` 메서드를 그대로 호출하여 결과를 반환한다.

3. 테스트 항목 및 결과

테스트 전 본인인증



테스트 항목

테스트 번호	항목	설명
1	고정 용량 ArrayList 테스트	값 삽입/삭제/삽입 위치 확인, 용량 초과 동작 확인
2	자동 확장 ArrayList 테스트	10개 초과 삽입 시 배열 확장 확인, contains() 확인
3	다양한 타입 저장 테스트	String, Integer, Boolean 등 다양한 타입 저장
4	ParaStack 테스트	push/pop, isEmpty 확인
5	ParaStack 테스트	타입별 스택 동작 확인
6	ParaStack raw 타입 테스트	제네릭 미사용 시 캐스팅 위험 확인

```
package org.example.ArrayListAssignment;

public class TestArray {

    public static void main(String[] args) {

        System.out.println("===== [1] ObjectArrayListLimitedCapacity 테스트 =====");
        ObjectArrayListLimitedCapacity limitedList = new ObjectArrayListLimitedCapacity();
        System.out.println("초기 isEmpty(): " + limitedList.isEmpty());
        limitedList.add("A");
        limitedList.add(123);
        limitedList.add(3.14);
        limitedList.add(1, "Inserted");
        printList(limitedList);
        System.out.println("remove(1): " + limitedList.remove(1));
        printList(limitedList);

        // 10개 초과 입력 테스트 (초기 용량 확인용)
        try {
            for (int i = 0; i < 11; i++) {
                limitedList.add("Item" + i);
            }
        } catch (ArrayIndexOutOfBoundsException e) {
            System.out.println("용량 초과 시 예외 발생 확인됨 (OK)");
        }
    }
}
```

```

}

System.out.println("\n===== [2] ObjectArrayList (확장형) 테스트 =====");
ObjectArrayList expandableList = new ObjectArrayList();
for (int i = 0; i < 15; i++) {
    expandableList.add(i);
}
System.out.println("확장 후 size: " + expandableList.size());
System.out.println("contains(5): " + expandableList.contains(5));
System.out.println("contains(100): " + expandableList.contains(100));
expandableList.add(5, "InsertedValue");
printList(expandableList);

System.out.println("\n===== [3] 다양한 타입 저장 테스트 =====");
ObjectArrayList typeTestList = new ObjectArrayList();
typeTestList.add("StringType");
typeTestList.add(42); // Integer
typeTestList.add(3.1415); // Double
typeTestList.add(true); // Boolean
printList(typeTestList);

System.out.println("\n===== [4] ParaStack<String> 제네릭 테스트 =====");
ParaStack<String> stringStack = new ParaStack<>();
stringStack.push("One");
stringStack.push("Two");
System.out.println("pop(): " + stringStack.pop()); // Two
System.out.println("isEmpty(): " + stringStack.isEmpty());
System.out.println("pop(): " + stringStack.pop()); // One
System.out.println("isEmpty(): " + stringStack.isEmpty());

System.out.println("\n===== [5] ParaStack<Integer> 제네릭 테스트 =====");
ParaStack<Integer> intStack = new ParaStack<>();
intStack.push(10);
intStack.push(20);
System.out.println("pop(): " + intStack.pop());
System.out.println("pop(): " + intStack.pop());
System.out.println("isEmpty(): " + intStack.isEmpty());

System.out.println("\n===== [6] 타입 지정 없이 ParaStack 테스트 (비타입 안전 실험) =====");
ParaStack rawStack = new ParaStack(); // 타입 미지정 → raw typerawStack.push("raw_string");
rawStack.push(100); // 타입 다르게 넣음

// 꺼내서 잘못된 타입으로 캐스팅 시도
Object value1 = rawStack.pop(); // → 안전함
Object value2 = rawStack.pop(); // → 안전함
System.out.println("정상 출력: " + value1 + ", " + value2);

// 비정상적인 캐스팅 시도 (컴파일 가능하지만 런타임 오류)
String wrongCast = (String) value2; // → Integer를 String으로 캐스팅
System.out.println("캐스팅된 문자열: " + wrongCast);
}

// 출력 헬퍼 함수
private static void printList(ObjectArrayListLimitedCapacity list) {
    System.out.print("List contents: ");
    for (int i = 0; i < list.size(); i++) {
        System.out.print(list.get(i) + " ");
    }
    System.out.println();
}
}

```

◆ [1] ObjectArrayListLimitedCapacity 테스트

목적:

고정 용량(10)의 리스트에서 기본적인 동작(add , get , remove , isEmpty)이 제대로 수행되는지 확인한다.
또한, 고정된 용량을 초과하여 삽입할 경우 예외가 발생하는지도 확인한다.

내용:

- isEmpty() 를 통해 초기 상태 확인
- add() 로 문자열, 정수, 실수 삽입
- 특정 위치(index=1)에 삽입 후, remove() 를 통해 해당 항목 제거
- 반복문을 통해 11개의 요소를 삽입하여 용량 초과 시 ArrayIndexOutOfBoundsException 발생 여부 확인

```
ObjectArrayListLimitedCapacity.java
11      System.out.println("");
12      System.out.println("At " + LocalDateTime.now().toString());
13      System.out.println("By " + System.getProperty("user.name") + " at " + InetAddress.getLocalHost().getHostName());
14      System.out.println("");
15

Run: sw1_dayeon [:TestArray.main()] x
...
✓ sw1_dayeon [:TestArray] 3 sec, 649 ms
> Task :TestArray.main()

At 2025-05-10T03:15:36.629157300
By dayeo at DESKTOP-V7HSVMD

===== [1] ObjectArrayListLimitedCapacity 테스트 =====
초기 isEmpty(): true
List contents: A Inserted 123 3.14
remove(1): Inserted
List contents: A 123 3.14
용량 초과 시 예외 발생 확인됨 (OK)

===== [2] ObjectArrayList (확장형) 테스트 =====
확장 후 size: 15
contains(5): true
contains(100): false
List contents: 0 1 2 3 4 InsertedValue 5 6 7 8 9 10 11 12 13 14

===== [3] 다양한 타입 저장 테스트 =====
List contents: StringType 42 3.1415 true

===== [4] ParaStack<String> 제네릭 테스트 =====
?
```

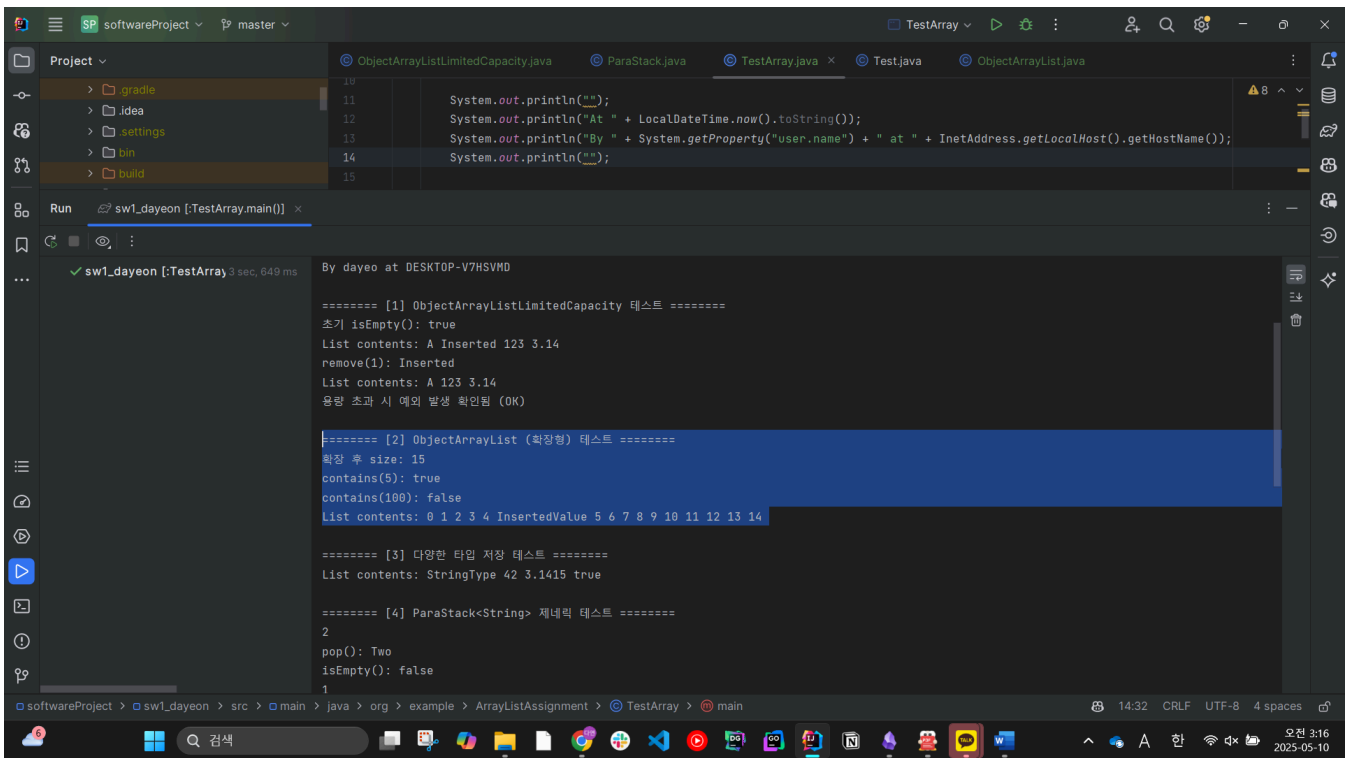
◆ [2] ObjectArrayList (확장형) 테스트

목적:

기본 용량(10)을 초과하는 데이터를 삽입했을 때 내부 배열이 2배 크기로 자동 확장되는지 확인한다.
또한 contains() 메서드를 사용하여 값이 리스트에 존재하는지 검사한다.

내용:

- 15개의 정수 추가 → 내부 배열이 10 → 20으로 확장됨
- size() 로 확장된 결과 확인
- contains(5) → true, contains(100) → false 결과 확인
- 중간 삽입 add(5, "InsertedValue") 수행 후 리스트 출력



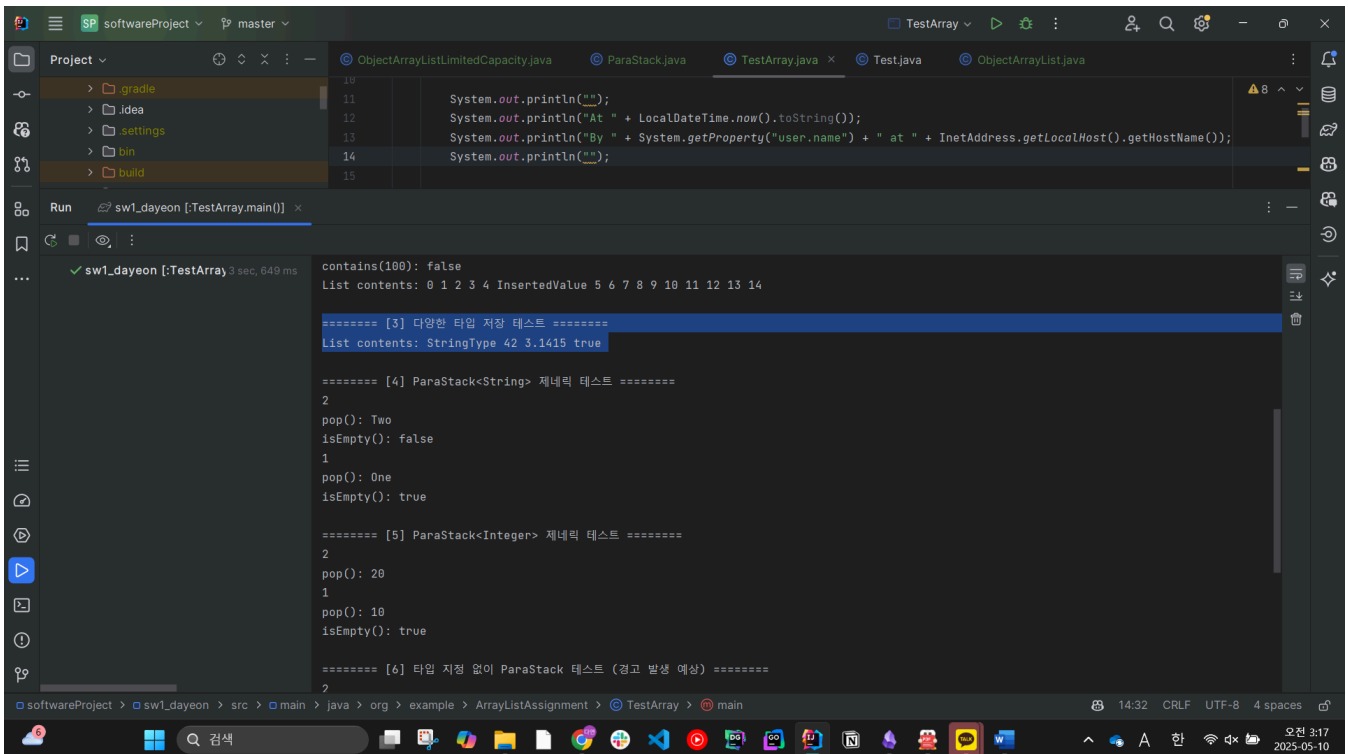
◆ [3] 다양한 타입 저장 테스트

목적:

ObjectArrayList 가 제네릭이 아니므로 다양한 타입(String , Integer , Double , Boolean)을 혼합 저장할 수 있는지 확인한다.

내용:

- 문자열, 정수, 실수, 불리언을 하나의 리스트에 차례대로 삽입
- printList() 를 통해 삽입 결과 출력



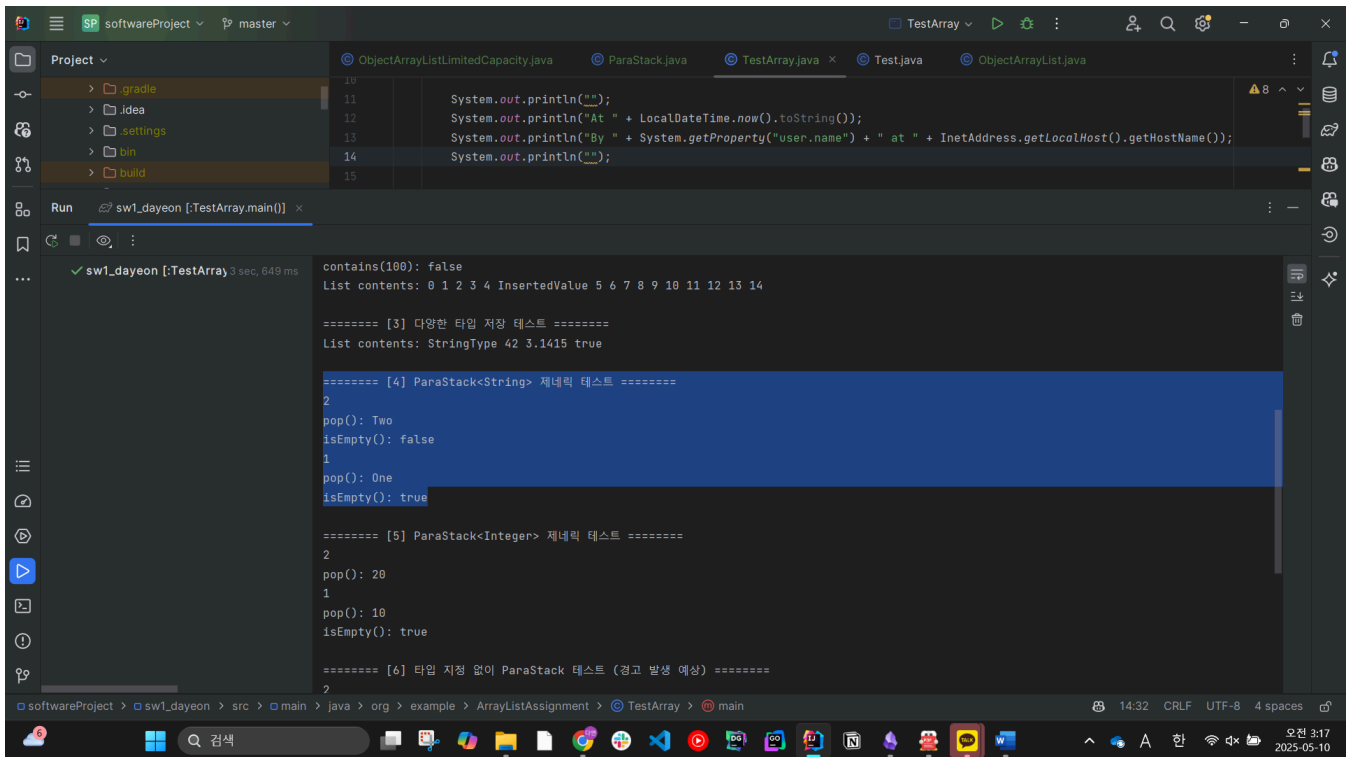
◆ [4] ParaStack<String> 제네릭 테스트

목적:

ParaStack 클래스가 제네릭 타입으로 동작하는지 확인한다.
push(), pop(), isEmpty() 가 예상대로 작동하는지 검증한다.

내용:

- 문자열 "One", "Two" 를 스택에 추가
- 두 번 pop() 하여 LIFO(후입선출) 순서 확인
- isEmpty() 를 통해 스택이 비었는지 확인



```
10
11      System.out.println("");
12      System.out.println("At " + LocalDateTime.now().toString());
13      System.out.println("By " + System.getProperty("user.name") + " at " + InetAddress.getLocalHost().getHostName());
14      System.out.println("");
15
Run sw1_dayeon [:TestArray.main()] x
...
✓ sw1_dayeon [:TestArray] 3 sec, 649 ms contains(100): false
List contents: 0 1 2 3 4 InsertedValue 5 6 7 8 9 10 11 12 13 14

===== [3] 다양한 타입 저장 테스트 =====
List contents: StringType 42 3.1415 true

===== [4] ParaStack<String> 제네릭 테스트 =====
2
pop(): Two
isEmpty(): false
1
pop(): One
isEmpty(): true

===== [5] ParaStack<Integer> 제네릭 테스트 =====
2
pop(): 20
1
pop(): 10
isEmpty(): true

===== [6] 타입 지정 없이 ParaStack 테스트 (경고 발생 예상) =====
2
```

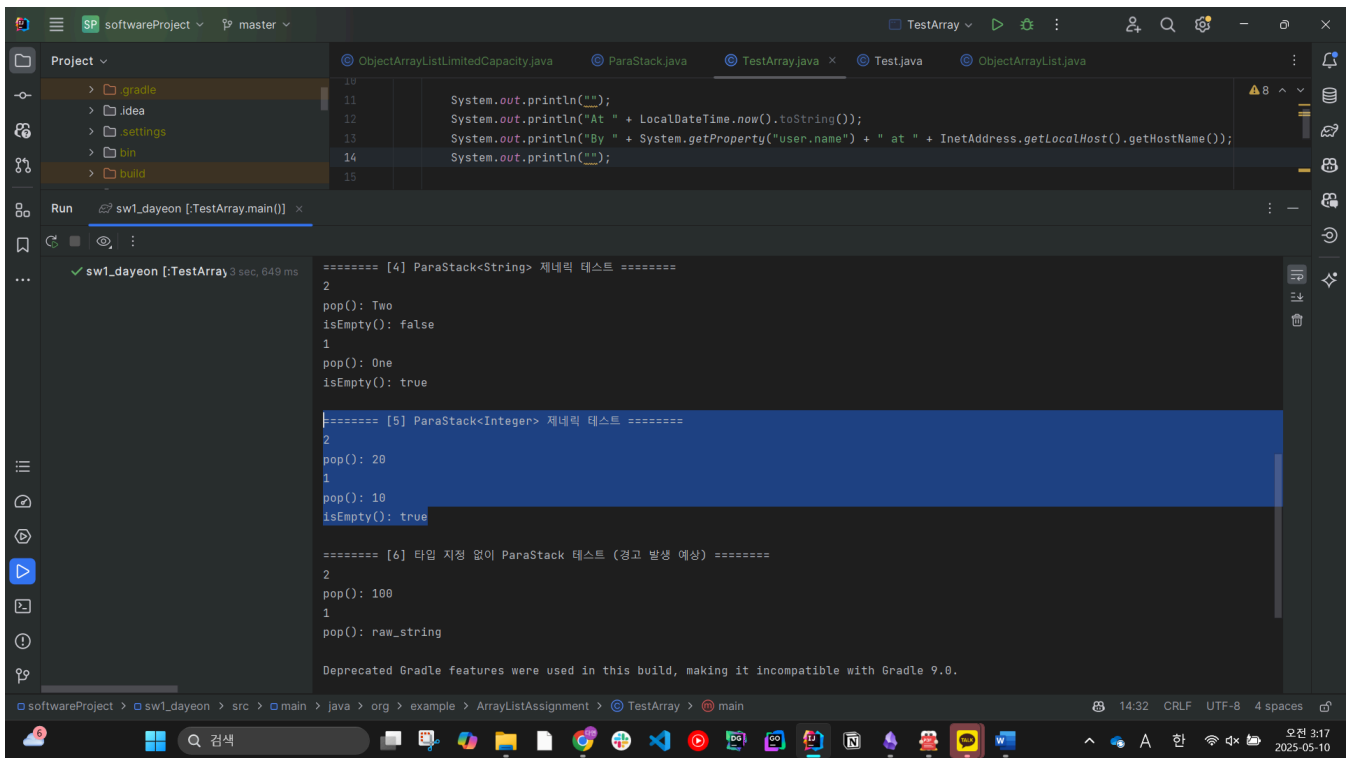
◆ [5] ParaStack<Integer> 제네릭 테스트

목적:

다른 제네릭 타입 (Integer)으로도 ParaStack 이 문제없이 작동하는지 확인한다.

내용:

- 정수 10, 20 을 스택에 push()
- pop() 2회로 정수 반환 확인
- 마지막에 isEmpty() 로 빈 스택 여부 확인



◆ [6] ParaStack 타입 미지정 테스트 (raw 타입)

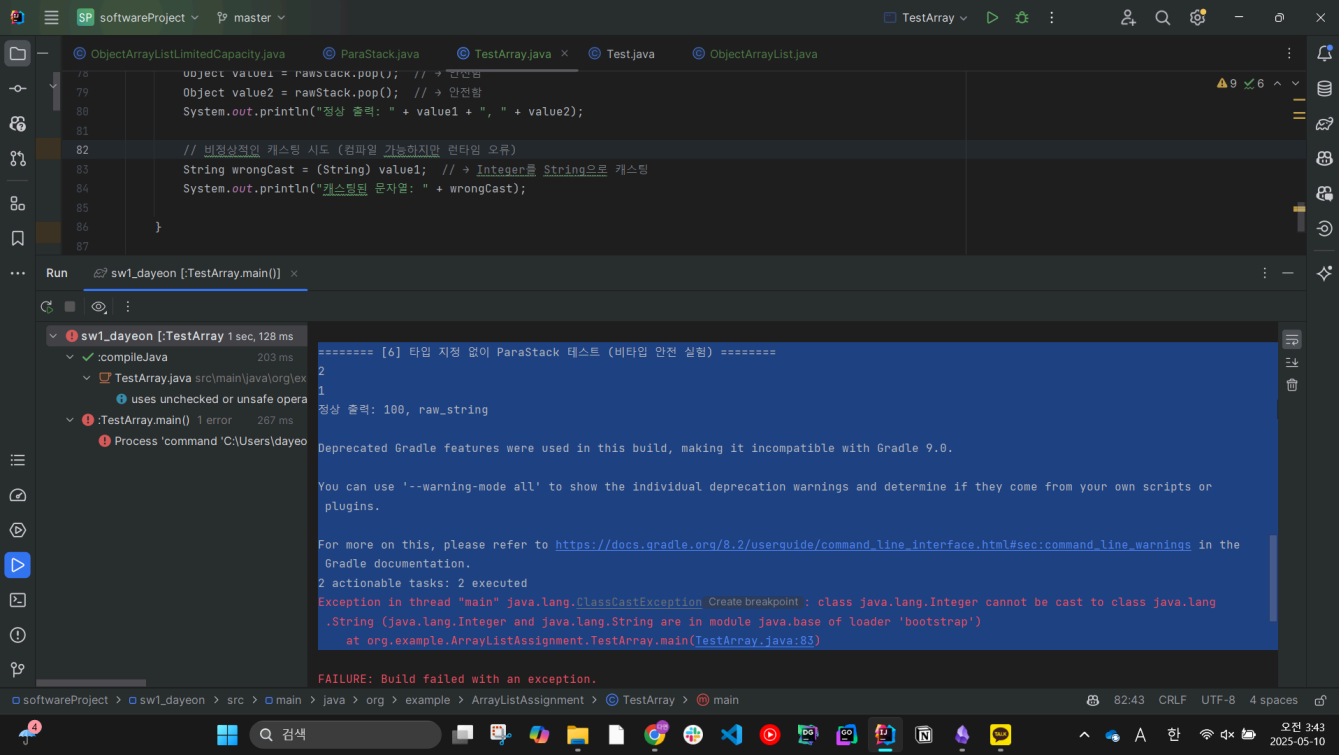
목적:

제네릭 타입을 명시하지 않은 상태에서 다양한 타입의 데이터를 혼합하여 저장하고, 잘못된 타입으로 캐스팅할 경우 **컴파일은 되지만 런타임 오류가 발생함을 실험적으로 확인**한다. 이를 통해 제네릭 타입 지정의 필요성과 type-safety의 중요성을 이해할 수 있다.

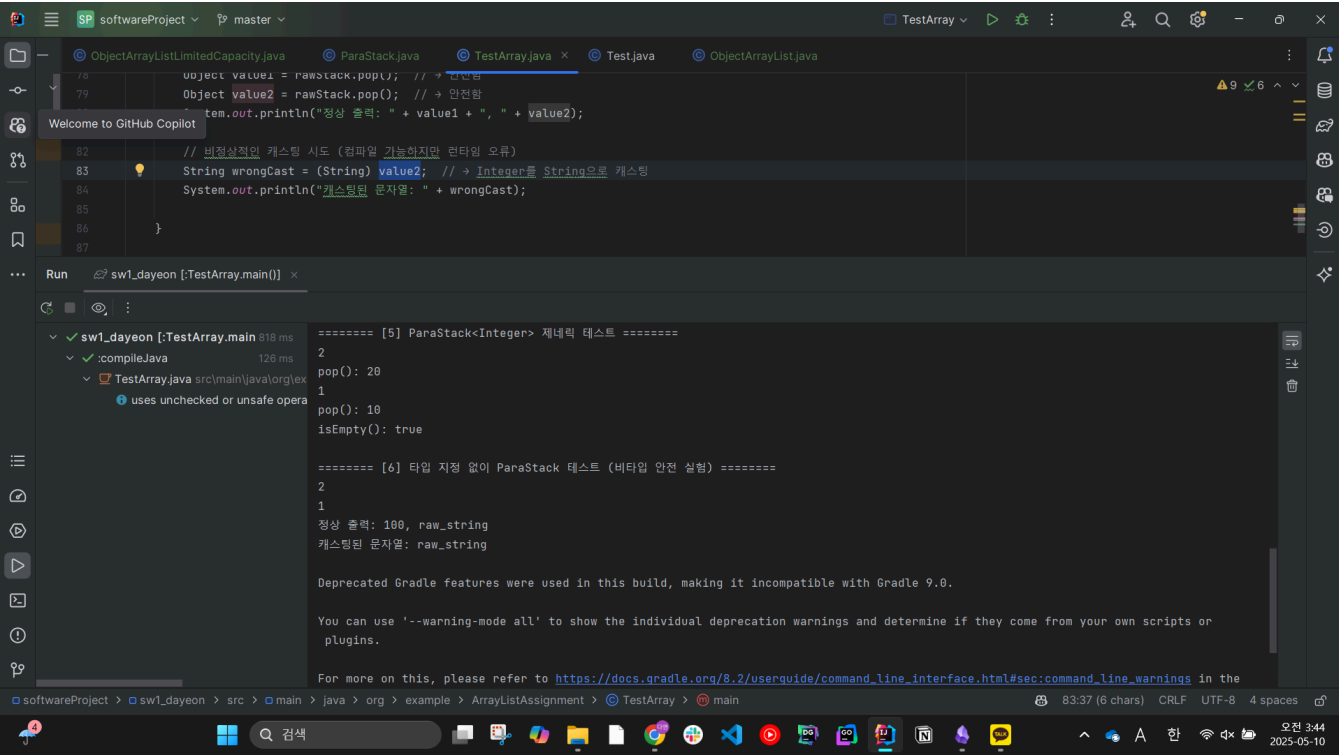
내용:

- 타입을 명시하지 않은 ParaStack 객체를 생성하여 raw type으로 사용함
- 문자열 "raw_string" 과 정수 100 을 스택에 순차적으로 저장
- 이후 pop() 으로 꺼낸 정수 100 을 String 으로 잘못 캐스팅함
- 해당 캐스팅은 **컴파일 시 오류 없이 통과되지만**, 런타임에서 **ClassCastException** 이 발생함
- 이를 통해 제네릭을 사용하지 않으면 **타입 안정성이 무너지고, 예기치 못한 런타임 예외**로 이어질 수 있음을 실험적으로 검증함

value 1을 캐스팅하여 오류 발생한 스크린샷



value 2를 캐스팅하여 정상적으로 작동하는 스크린샷



평가표

평 가 표

평가 항목	학생 자체 평가 (리포트 해당 부분 표시 및 간단한 의견)	평가 (빈칸)	점수 (빈칸)
<ul style="list-style-type: none"> - ObjectArrayList 구현? <ul style="list-style-type: none"> * ObjectArrayListLimitedCapacity 확장 <ul style="list-style-type: none"> . array 용량 제한을 해결 . inheritance 이용 필수 - 실험으로 검증? <ul style="list-style-type: none"> * 검증 사항? generic, 용량 	add시 CAPACITY를 늘린 새 array를 생성하여 용량 제한 해결 inheritance를 활용하여 super.add super.size 등을 사용 expandableList를 활용하여 확장 사이즈 검증, String, Integer 두가지 모두 검증 확인		
<ul style="list-style-type: none"> - ParaStack 구현? <ul style="list-style-type: none"> * ObjectArrayList를 저장공간으로 사용? * parameterized coding - 실험으로 검증? <ul style="list-style-type: none"> * parameterizing * <u>type-safety</u> 동작 확인 	ParaStack의 내부 저장소로 Object ArrayList를 활용하여 스택 구조 구현 T라는 제네릭 타입을 활용하여 parameterized coding을 진행 ParaStack<String>과 <Integer> 두가지 타입으로 선언한 뒤 동작 확인 타입을 명시하지 않은 ParaStack을 생성하여 String과 Integer의 혼합 저장 진행		
기타			
총평/계			

* 학생 자체 평가는 점수에 반영되지 않음.

* 학생 스스로 자신의 보고서를 평가하면서, 체계적으로 프로젝트를 마무리하도록 유도하는 것이 목
적임.