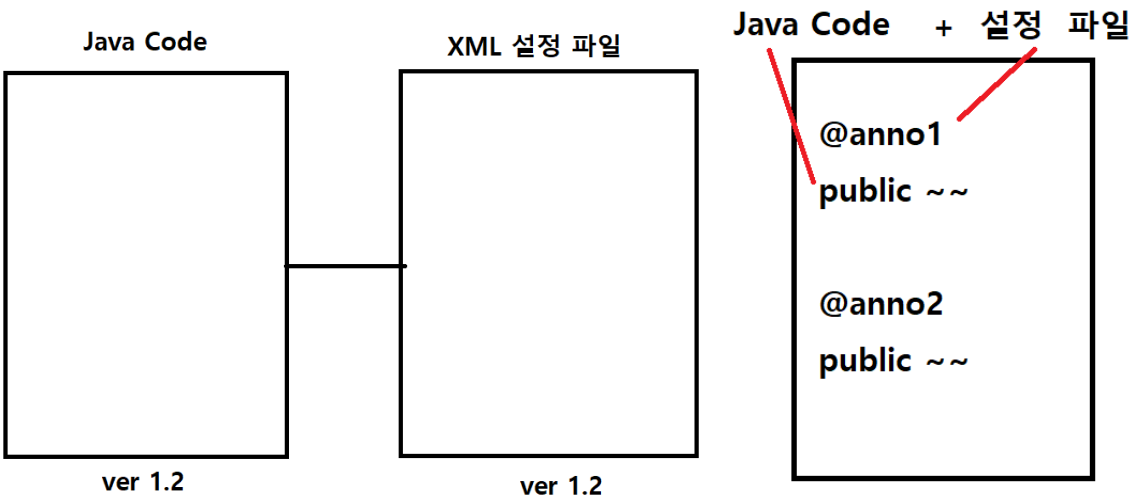
Annotation

Annotation이란?

자바 소스 코드에 추가하여 사용할 수 있는 메타데이터의 일종
보통 @ 기호를 앞에 붙여서 사용
JDK 1.5 버전 이상에서 사용 가능
클래스 파일에 임베디드 되어 컴파일러에 의해 생성된 후 JVM에 포함되어 작동



=> 하나의 파일에서 코드와 설정을 관리할 수 있음

Annotation의 용도

- 컴파일러에게 코드 작성 문법 에러를 체크하도록 정보 제공
- 소프트웨어 개발 툴이 빌드나 배치 시 코드를 자동으로 생성할 수 있도록 정보 제공
- 실행(런타임) 시 특정 기능을 실행하도록 정보 제공

Annotation의 종류

표준 Annotation

자바가 기본적으로 제공해주는 어노테이션

- @Override : 오버라이딩을 올바르게 했는지 컴파일러가 검사
- @Deprecated : 앞으로 사용하지 않을 것을 권장하는 필드나 메소드
- @FunctionalInterface : 함수형 인터페이스에 붙이면, 컴파일러가 올바르게 작성했는지 검사
- @SuppressWarnings : 컴파일러의 경고 메시지가 나타나지 않게 함

메타 Annotation

어노테이션을 위한 어노테이션

- @Target : 어노테이션을 정의할 때, 적용 대상을 지정

```
@Target({TYPE, FIELD, TYPE_USE})
@Retention(RetentionPolicy.SOURCE)
public @interface MyAnnotation{}

@MyAnnotation // 적용 대상이 Type(클래스, 인터페이스)
class MyClass{
      @MyAnnotation //적용 대상이 FIELD인 경우
    int i;

    @MyAnnotation //적용 대상이 TYPE_USE인 경우
    MyClass mc;
}
```

- @Retention : 어노테이션이 유지되는 기간을 지정
  - SOURCE : 소스 파일에만 존재
  - RUNTIME : 클래스 파일에 존재. 실행 시에 사용 가능
- @Documented : javadoc으로 작성한 문서에 포함
- @Inherited : 어노테이션을 자손 클래스에 상속

```
@Inherited
@interface SuperAnno{}

@SuperAnno
class Parent{}

// <- 여기에 @SuperAnno 가 붙은 것으로 인식
class Child extends Parent{}
```

- @Repeatable : 반복해서 붙일 수 있는 어노테이션을 정의
  - 아래의 'ToDocs'와 같이 "컨테이너 어노테이션"도 정의해야 함.

```
@Repeatable(ToDos.class)
@interface ToDo{
      String value();
}

@ToDo("delete test codes.")
@ToDo("override inherited methods")
class MyClass{
      ~~
}

@interface ToDos{
      ToDo[] value();
```

```
}
```

사용자 정의 Annotation

사용자가 직접 정의하는 어노테이션

```
@interface 이름{
      타입 요소 이름(); // 어노테이션의 요소를 선언
          ...
}
```

Annotation 요소 특징

- 적용 시 값을 지정하지 않으면, 사용될 수 있는 기본값을 지정할 수 있다.
- 요소가 하나이고 이름이 value일 때는 요소의 이름이 생략 가능

```
@interface TestInfo{
      String value();
}
@TestInfo("passed") // value="passed"와 동일
class NewClass{...}
```

- 요소의 타입이 배열인 경우, 중괄호({})를 사용해야 한다.

```
@interface TestInfo{
      String[] testTools();
}

@TestInfo(testTools={"JUnit", "AutoTester"})
@TestInfo(testTools="JUnit") // 요소가 1개일 때는 {}를 사용하지 않아도
된다.
@TestInfo(testTool={}) // 요소가 없으면 {}를 써넣어야 한다.
```

Annotation 규칙

- 요소의 타입은 기본형, String, Enum, Annotation, Class만 허용
- 괄호 () 안에 매개변수를 선언할 수 없다.
- 예외를 선언할 수 없다.
- 요소의 타입을 매개변수로 정의할 수 없다.(<T>)
- 잘못된 예시

```
@interface AnnoConfigTest{
    int id = 100; // 상수 ok
    String major(int i, int j) //매개변수 x
    String minor() throws Exception; // 예외 x
    ArrayList<T> list(); // 요소의 타입을 매개변수 x
```

모든 어노테이션의 조상

Annotation은 모든 어노테이션의 조상이지만 상속은 불가능하다.

```
public interface Annotation{
      boolean equals(Object obj);
    int hashCode();
    String toString();

    Class<? extends Annotation> annotationType();
    }
```

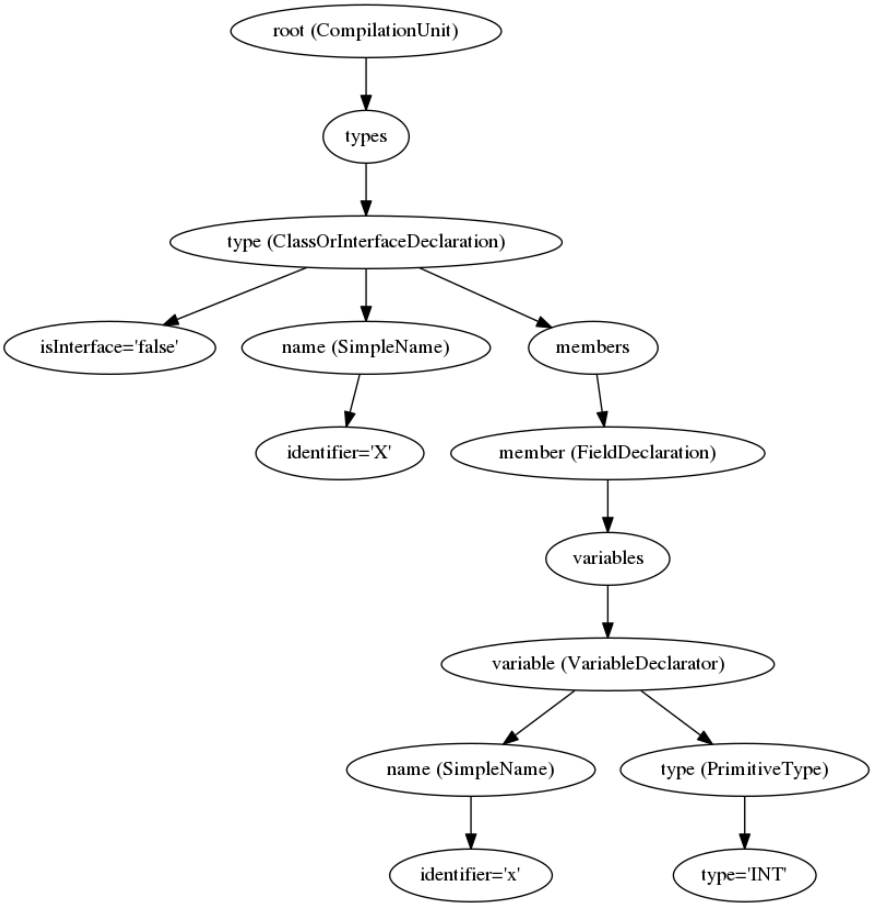
Marker Annotation

요소가 하나도 정의되지 않은 어노테이션
대표적으로 @Test가 있다.
해당 어노테이션은 테스트 프로그램에게 테스트 대상임을 알리는 어노테이션이다.

Annotation Processor

Annotation Processor란?

컴파일 시점에 끼어들어 특정한 어노테이션이 붙어있는 소스 코드를 참조해서 새로운 소스 코드를 만들어 낼 수 있는 기능.
이때 생성되는 소스 코드는 자바일 수도 있고 다른 어떤 코드일 수도 있다.



Annotation Processor 사용 예시

- 롬복(Lombok)
- AutoService : java.util.ServiceLoader 용 파일 생성 유틸리티
- @Override
- Dagger2 : 컴파일 타임 DI 제공
- 안드로이드 라이브러리
    - ButterKnife : @BindView(뷰 아이디와 어노테이션 붙인 빌드 바인딩)
    - DeepLinkDispatch : 특정 URI 링크를 Activity로 연결할 때 사용

Lombok이란?

@Getter, @Setter, @Builder 등의 어노테이션과 어노테이션 프로세서를 제공하여
표준적으로 사용하는 라이브러리
반복적이고 공통적인 코드 작성을 어노테이션을 통해 자동화해주는 라이브러리라고도 할 수 있음

Annotation Processor 장단점

- 장점 : 런타임 비용이 없다
- 단점 : 기존 클래스 코드를 변경할 때는 약간의 hack이 필요(Lombok)

Generic

Generic이란?

자바에서 제네릭(generic)이란 “데이터의 타입을 일반화한다"는 것을 의미
제네릭은 클래스나 메소드에서 사용할 내부 데이터 타입을 컴파일 시에 미리 지정하는 방법

제네릭 사용의 이점

- 클래스나 메소드 내부에서 사용되는 객체의 타입 안전성을 높일 수 있다.
- 반환값에 대한 타입 변환 및 타입 검사에 들어가는 노력을 줄일 수 있다.
- 타입을 국한하기 때문에 요소를 찾아올 때 타입 변환을 할 필요가 없어 프로그램 성능 향상

```
ArrayList list = new ArrayList(); //제네릭을 사용하지 않을경우
list.add("test");
String temp = (String) list.get(0); //타입변환이 필요함

ArrayList<String> list2 = new ArrayList(); //제네릭을 사용할 경우
list2.add("test");
temp = list2.get(0); //타입변환이 필요없음
```

Generic 사용법

```
public class 클래스명<T> {...}
public interface 인터페이스명<T> {...}
```

자주 사용하는 타입인자

<T>	Type
<E>	Element
<K>	Key
<N>	Number
<V>	Value
<R>	Result

제네릭 클래스

```
class ExClassGeneric<T> {
    private T t;
```

```
    public void setT(T t) {
        this.t = t;
    }

    public T getT() {
        return t;
    }
}
```

제네릭 인터페이스

```
interface ExInterfaceGeneric<T> {
    T example();
}

class ExGeneric implements ExInterfaceGeneric<String> {

    @Override
    public String example() {
        return null;
    }
}
```

멀티 타입 파라미터 사용

```
class ExMultiTypeGeneric<K, V> implements Map.Entry<K,V>{

    private K key;
    private V value;

    @Override
    public K getKey() {
        return this.key;
    }

    @Override
    public V getValue() {
        return this.value;
    }

    @Override
    public V setValue(V value) {
        this.value = value;
        return value;
```

```
    }
}
```

Generic 주요 개념

Bounded Type Parameters

매개변수화된 타입에 타입인자를 넣을 때, 타입 제한을 걸고 싶을 때가 있을 수 있다.
예를 들어 Number 타입 또는 Number 타입의 자식 클래스의 객체만 허용하는 숫자 처리를 위한 메소드를 만들 수 있다.
제한된 타입매개변수는 해당 변수 이름 뒤에 extends 키워드를 놓고 제한 시킬 타입을 작성한다.
여기서의 extends는 클래스의 extends 또는 인터페이스의 implements처럼 구현한다는 것은 아니다.

```
public class Box<T> {

    private T t;

    public void set(T t) {
        this.t = t;
    }

    public T get() {
        return t;
    }

    public <U extends Number> void inspect(U u){
        System.out.println("T: " + t.getClass().getName());
        System.out.println("U: " + u.getClass().getName());
    }

    public static void main(String[] args) {
        Box<Integer> integerBox = new Box<Integer>();
        integerBox.set(new Integer(10));
        integerBox.inspect("some text"); // error: this is still String!
    }
}
```

Generic WildCard

와일드카드 타입에는 총 세가지의 형태가 있으며 물음표(?)라는 키워드로 표현된다.

- 제네릭타입<?> : 타입 파라미터를 대치하는 것으로 모든 클래스나 인터페이스 타입이 올 수 있다.

- 제네릭타입<? extends 상위타입> : 와일드카드의 범위를 특정 객체의 하위 클래스로 제한
- 제네릭타입<? super 하위타입> : 와일드카드의 범위를 특정 객체의 상위 클래스로 제한
- 제네릭타입<? extends A & B & C> : 와일드카드의 범위를 명시된 다수 객체의 하위 클래스로 제한
    - 다수의 객체로 제한할 때, 클래스가 앞에 와야하며 클래스가 인터페이스보다 뒤에 위치할 경우 compile-type error가 발생

```
public class Calcu {
    public void printList(List<?> list) {
       for (Object obj : list) {
         System.out.println(obj + " ");
       }
    }

    public int sum(List<? extends Number> list) {
      int sum = 0;
      for (Number i : list) {
        sum += i.doubleValue();
      }
      return sum;
    }

   public List<? super Integer> addList(List<? super Integer> list) {
      for (int i = 1; i < 5; i++) {
       list.add(i);
      }
      return list;
    }
}
```

Generic Method

리턴 타입이 무엇인지는 상관 없이 내가 제네릭 메소드라는 것을 컴파일러에게 알려줘야 한다.
그러기 위해 리턴 타입을 정의하기 전에 제네릭 타입에 대한 정의를 반드시 명시해야 한다.
제네릭 클래스가 아닌 일반 클래스 내우베도 제네릭 메소드를 정의할 수 있다.
즉, 클래스에 지정된 타입 파라미터와 제네릭 메소드에 정의된 타입 파라미터는 상관이 없다.
제네릭 클래스에 <T>를 사용하고, 같은 클래스의 제네릭 메소드에도 <T>로 같은 이름을 가진 타입 파라미터를 사용하더라도 둘은 전혀 상관 없다.

Erasure

공변이란?

자기 자신과 자신 객체로 타입 변환을 허용해주는 것

```
Object[] before = new Long[1];
```

불공변이란?

자기 자신과 타입이 같은 것만 같다고 인식

```
public class Test {
    public static void test(List<Object> list) {

    }
    public static void main(String[] args) {
        List<String> list = new ArrayList<>();
        list.add("Gyunny");
        test(list);   // 컴파일 에러
    }
}
```

구체화(reify)와 비구체화(non-reify)

구체화 타입(reifiable type) : 자신의 타입 정보를 런타임에도 알고 있는 것
비구체화 타입(non-reifiable type) : 런타임에는 소거(erasure)가 되기 때문에 컴파일 타임보다 정보를 적게 가지는 것

Generic Type Erasure란?

erasure(소거)란 원소 타입을 컴파일 타임에만 검사하고 런타임에는 해당 타입 정보를 알 수 없는 것.
즉, 컴파일 타임에만 타입 제약 조건을 정의하고, 런타임에는 타입을 제거한다.

Java Compiler의 타입 소거

제네릭 타입	타입 소거 형식
unbounded Type(<?>, <T>)	Object
bounded Type(<E extends Comparable>)	Comparable

- 제네릭 타입을 사용할 수 있는 일반 클래스, 인터페이스, 메소드에만 소거 규칙을 적용
- 타입 안전성 보존을 위해 필요하다면 type casting 삽입
- 확장된 제네릭 타입에서 다형성을 보존하기 위해 bridge method 생성

unbounded type

```
// 컴파일 할 때 (타입 소거 전)
public class Test<T> {
    public void test(T test) {
        System.out.println(test.toString());
    }
}
// 런타임 때 (타입 소거 후)
public class Test {
    public void test(Object test) {
        System.out.println(test.toString());
    }
}
```

bound type

```
public class Test<T extends Comparable<T>> {
    private T data;

    public T getData() {
        return data;
    }

    public void setData(T data) {
        this.data = data;
    }
}
public class Test {
    private Comparable data;

    public Comparable getData() {
        return data;
    }

    public void setData(Comparable data) {
        this.data = data;
    }
}
```

Bridge Method

```
// 타입 소거 전
public class MyComparator implements Comparator<Integer> {
   public int compare(Integer a, Integer b) {
      //
   }
```

```
}
// 타입 소거 후
public class MyComparator implements Comparator {
   public int compare(Integer a, Integer b) {
      //
   }
}

// bridge method 추가
// 메소드 시그니처 사이에 불일치를 없애기 위해 컴파일러는 런타임에 해당
제네릭 타입의 타입 소거를 위한 bridge method 생성
// 이후 Integer 타입의 compare 메소드 사용 가능
public class MyComparator implements Comparator<Integer> {
   public int compare(Integer a, Integer b) {
      //
   }

   //THIS is a "bridge method"
   public int compare(Object a, Object b) {
      return compare((Integer)a, (Integer)b);
   }
}
```

Lambda

Lambda란?

람다식은 1930년대 알론조 처치(Alonzo Church)라는 수학자가 처음 제시한 함수의 수학적 표기방식인 '람다 대수(lambda calculus)'에 그 뿌리를 두고 있다.
람다식을 이용하면 코드가 간결해지고, 지연 연산 등을 통해서 성능 향상을 도모할 수 있다. 반면 모든 요소를 순회하는 경우에는 성능이 떨어질 수도 있고, 코드를 분석하기 어려워질 수도 있다.

```
// 두 정수를 입력 받아서 합을 구해주는 'sum()' 메소드
public int sum(int a, int b) {
    return a + b;
}

// 람다식 표현
(a, b) -> a + b;

// 람다식 표현의 컴파일러 해석
new Object() {
    int sum(int a, int b) {
          return a + b;
    }
}
```

지연 연산(Lazy Evaluation)이란?

불필요한 연산을 피하기 위해 연산을 지연시키는 것
문제들이 차례로 주어지더라도 마지막 문제를 제공 받을 때까지 기다리다가 마지막 문제를 알게 되면, 그때 연산을 시작
참고 : https://dororongju.tistory.com/137

Lambda 문법

```
(매개변수 목록) -> { 람다식 바디 }
람다식의 파라미터를 추론할 수 있는 경우에는 타입을 생략할 수 있다.
// 매개변수가 하나인 경우 괄호를 생략할 수 있다.
// 바디부분에 하나의 표현식만 오는 경우 중괄호를 생략할 수 있으며 이때
세미콜론은 반드시 생략해야한다.
// 바디의 계산식 결과가 반환된다.
a -> a * a

// return문이 있을 경우 중괄호가 필수되어야한다.
(a, b) -> { return a > b ? a : b }
```

```
(a, b) ->  a > b ? a : b
```

Lambda를 이용한 Runnable 구현

```
Thread thread = new Thread(new Runnable() {

    @Override
    public void run() {
          System.out.println("Start Thread");
          Thread.sleep(1000);
          System.out.println("End Thread");
   }
});

// 람다식 이용
Thread thread = new Thread(() -> {
          System.out.println("Start Thread");
          Thread.sleep(1000);
          System.out.println("End Thread");
});
```

Lambda를 이용한 컬렉션 순회

```
List<String> list = new ArrayList();
list.add("Element1");
list.add("Element2");
list.add("Element3");

list.forEach(x -> System.out.println(x))
// 위 코드는 list.forEach(System.out::println) 으로 축약할 수 있음
```

함수형 인터페이스

람다식을 저장할 수 있는 변수

함수형 인터페이스 정의

함수형 인터페이스를 정의하고 '@functionalInterface' 어노테이션을 붙여주면 자바 컴파일러가 함수형 인터페이스의 정의를 검증해준다.

```
@FunctinalInterface
interface MySum {
    public int sum(int a, int b);
}
```

```
// 람다식
public static void main(String []args) {

    MySum func = (a, b) -> a + b;

    System.out.println(func.sum(10, 11));
}
```

java.util.function 패키지

함수형 인터페이스	메소드
java.lang.Runnable	void run();
Supplier<T>	T get();
Consumer<T>	void accept(T t);
Function<T, R>	R apply(T t);
Predicate<T>	boolean test(T t);
BiConsumer<T, U>	void accept(T t, U u);
BiPredicate<T, U>	boolean test(T t, U u)
UnaryOperator<T>	T apply(T t);
BinaryOperator<T>	T apply(T t1, T t2);
IntFunction<R>, LongFunction<R>, DoubleFunction<R>	R apply(int value), R apply(long value), R apply(double value),
ToIntFunction<R>, ToLongFunction<R>, ToDoubleFunction<R>	int applyAsInt(T t), int applyAsLong(T t), int applyAsDouble(T t)

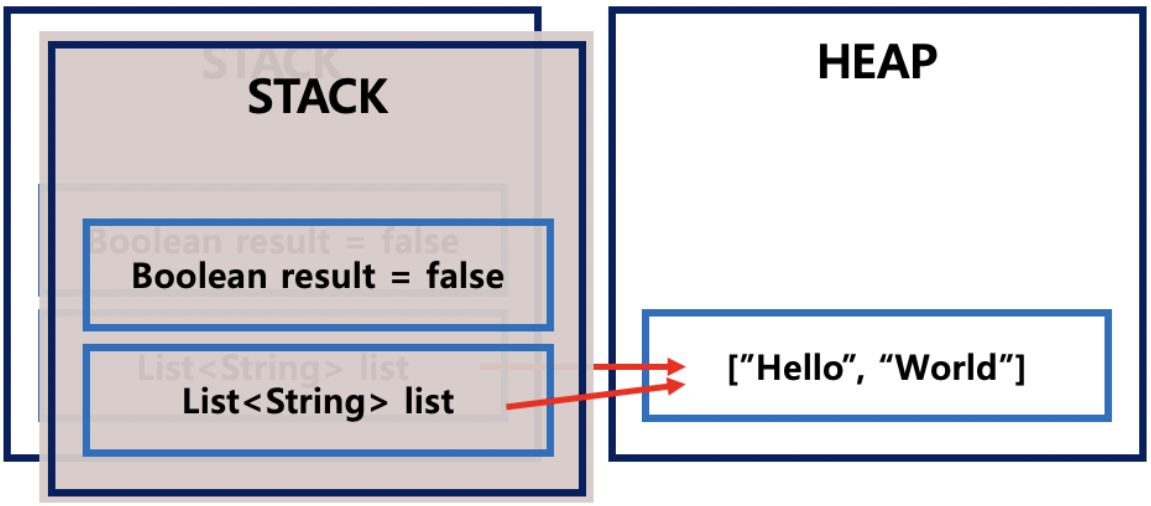
컬렉션과 함께 사용할 수 있는 함수형 인터페이스

인터페이스	메소드	설명
Collection	boolean removeif(Predicate<E> filter	조건에 맞는 요소 삭제
List	void replaceAll(UnaryOperator<E> operator);	모든 요소에 operator 적용 후 대체
Iterable	void forEach(Consumer<T> action);	모든 요소에 action 수행

Map	V compute(K key, BiFunction<K, V, V> f);	지정된 키에 해당하는 값에 f 수행
Map	V computeIfAbsent(K key, Function<K, V, V> f);	지정된 키가 없으면 f 수행 후 추가
Map	V computeIfPresent(K key, Function<K, V> f);	지정된 키가 있을 때, f 수행
Map	V merge(K key, V value, BiFunction<V, V, V> f);	모든 요소에 Merge 수행, 키에 해당하는 값이 있으면 f 수행해서 병합 후 할당
Map	void forEach(BiConsumer<K, V> action);	모든 요소에 action 수행
Map	void replaceAll(BiFunction<K, V, V> f);	모든 요소에 f 수행 후 대체

Variable Capture

람다식을 이용해 함수를 작성할 때, 람다 내부에서 람다 외부 변수를 수정하고자 할 때, "Variable used in lambda expression should be final or effectively final" 에러가 발생하는데, 이는 람다 캡쳐링(Lambda Capturing) 때문이다.



람다 캡쳐링은 Call by Reference가 아닌 Call by Value로 일어난다.
따라서, Call by Value로 캡쳐링이 된 람다 스택 내 변수를 수정하는 것은 컴파일 오류를 발생시킨다.
즉, 람다 외부 변수는 참조만 가능하다.

```
List<Boolean> boolList = new ArrayList<>();
List<String> list = Arrays.asList("Hello", "World");
Optional.ofNullable(list)
        .orElseThrow(Exception::new)
        .stream()
        .forEach(str -> {
            boolList.add(Boolean.TRUE);  // to Heap
        });
```

위와 같이 람다 내부에서 람다 외부 컬렉션을 조작하는 경우는 별다른 에러가 발생하지 않는데, 이는 컬렉션의 경우 데이터가 Stack이 아닌 Heap에 저장되어 있기 때문이다.

메소드, 생성자 레퍼런스

메소드 레퍼런스는 람다 표현식을 더 간단하게 표현하기 위한 방법

```
// 람다식 표현
Consumer<String> func = text -> System.out.println(text);
func.accept("Hello");

// 메소드 레퍼런스 표현
Consumer<String> func = System.out::println;
func.accept("Hello");
```

메소드 레퍼런스는 ClassName::MethodName 형식으로 입력한다. 메소드를 호출하는 것이나 괄호'()'는 써주지 않고 생략한다.
메소드 레퍼런스에는 많은 코드가 생략되어 있기 대문에 사용하려는 메소드의 인자와 리턴 타입을 알고 있어야 한다.

Static Method Reference

```
interface Executable {
    void doSomething(String text);
}

public static class Printer {
    static void printSomething(String text) {
        System.out.println(text);
    }
}

public static void main(String args[]) {
    Executable exe = text -> Printer.printSomething(text);
    Executable exe2 = Printer::printSomething;
    exe.doSomething("do something");
    exe2.doSomething("do something");
}


// Consumer 사용
Consumer<String> consumer = Printer::printSomething;
consumer.accept("do something");
List<String> companies = Arrays.asList("google", "apple", "google",
"apple", "samsung");
// 1. lambda expression
companies.stream().forEach(company -> System.out.println(company));
```

```
// 2. static method reference
companies.stream().forEach(System.out::println);
```

Instance 메소드 레퍼런스

```
public static class Company {
    String name;
    public Company(String name) {
        this.name = name;
    }

    public void printName() {
        System.out.println(name);
    }
}

public static void main(String args[]) {
    List<Company> companies = Arrays.asList(new Company("google"),
        new Company("apple"), new Company("samsung"));
    companies.stream().forEach(company -> company.printName());
    // companies1.stream().forEach(Company::printName);
    // 결과 동일
}
// 실행 결과
// google
// apple
// samsung
List<String> companies = Arrays.asList("google", "apple", "google",
"apple", "samsung");
companies.stream()
         .mapToInt(String::length) // 람다식: company -> company.length()
         .forEach(System.out::println);
// 실행 결과
// 6
// 5
// 6
// 5
// 7
```

Constructor 메소드 레퍼런스

```
public static class Company {
    String name;
    public Company(String name) {
```

```
        this.name = name;
    }

    public void printName() {
        System.out.println(name);
    }
}

public static void main(String args[]) {
    List<String> companies = Arrays.asList("google", "apple", "google",
"apple", "samsung");
    companies.stream()
            .map(name -> new Company(name))
            .forEach(company -> company.printName());
}
// 실행 결과
// google
// apple
// google
// apple
// samsung
```