7장 모든 객체의 공통 메서드

분류	서적 이펙티브 자바
시작일자	@2022년 5월 16일 오전 9:11
종료일자	@2022년 5월 20일 오전 10:08
참고	참고 블로그 [자바봄 기술 블로그] 참고 github.ido
스터디 링크	GitHub Link [이펙티브 자바 스터디]
서적	
Java	

▼ 목차

🚀 아이템 42. 익명 클래스보다는 람다를 사용하라

✅ 익명 클래스

```
private static final List<String> strings = Arrays.asList("book", "apple", "fineapple", "grape", "car", "bobo", "bbibbi", "crong");

@Test
@DisplayName("익명클래스")
void 익명클래스() {
    Collections.sort(strings, new Comparator<String>() {
        @Override
        public int compare(String o1, String o2) {
            return Integer.compare(o1.length(), o2.length());
        }
    });
}
```

이전(아이템24)에서 익명클래스를 이용했었다.

익명클래스를 사용해 위처럼 문자열 정렬을 구현했다. 예전에는 이런 방식으로 정렬을 많이 구현했는데

`Comparator` 처럼 추상메서드가 하나뿐인 인터페이스는 람다식으로 바꿀 수 있다.

✅ 람다식 적용

```
@Test
@DisplayName("익명클래스->람다식")
void 익명에서람다로() {
    Collections.sort(strings, (s1, s2)-> Integer.compare(s1.length(), s2.length()));
}
```

익명 클래스와는 다르게 `s1` `s2`가 무슨 타입인지 따로 명시해주지 않았다.

❓ 타입을 명시하지 않았는데 컴파일에서 오류가 안나는 것인가?

- 컴파일러는 문맥을 살펴 **타입추론**을 해준다.
- 위의 예시에서는 `strings`는 `ArrayList`로 타입에 제네릭으로 `List<String>` 으로 작성해줬기 때문에 타입추론이 가능하다.
- 단 로타입으로 작성했다면 컴파일 오류가 났을 것이다.

하지만 모든 상황에서 추론을 해주는 것은 아니기에 컴파일러가 오류를 낸다면 해당 타입을 명시하자.

✅ 람다식 + Method Reference

```
@Test
@DisplayName("람다식 + Method Reference 사용")
void 람다식레퍼런스타입() {
    Collections.sort(strings, comparing(String::length));
}
```

✅ List의 sort 메서드 사용

```
@Test
@DisplayName("List의 sort 사용")
void ListSort() {
    strings.sort(comparingInt(String::length));
}
```

⛔ 람다식의 단점

람다식은 코드가 간결해진다는 장점과 타입을 추론해준다는 장점이 있었습니다.

하지만 이러한 람다식도 마냥 좋은 것만은 아닙니다.

- **상황에따라 익명클래스를 사용해야 하는 경우가 있다.**
  - 람다는 **이름이 없고 문서화도 불가능**하다.
  - 즉 코드 자체로 동작이 명확하지 않고 모호하다면 람다를 사용할 수 없다.
- **람다식은 간결함을 유지하지 못하면 코드를 작서한 개발자가 아닌 다른 사람이 볼 때 수수께끼의 코드가 된다.**
  - 람다식을 사용할 때 해당 코드가 간결함을 잃거나 명확성을 잃을 것 같다면 리팩토링을 하거나 사용하지 말자.
- **람다식으로 대체할 수 없는 영역도 존재한다.**
  - 추상 메서드가 2개 이상인 인터페이스에서는 람다식을 쓸 수 없다.
  - 람다는 자신을 참조 할 수 없다.

🚀 아이템 43. 람다보다는 메서드 참조를 사용하라

자바는 함수 객체를 람다 보다 더 간결하게 생성하는 방법을 제공한다. → **메서드 참조**

✅ 기본적인 코드 작성

```
public void saveFoodAll(List<FoodSaveForm> forms) {
    for (FoodSaveForm form : forms) {
        Food food = form.toFood();
        foodRepository.save(food);
    }
}
```

✅ 람다식을 적용한 코드

```
public void saveFoodAll(List<FoodSaveForm> forms) {
    forms.stream()
            .map(form-> form.toFood())
            .forEach(food -> formRepository.save(food));
}
```

✅ 람다식에서 메서드 참조를 사용한 코드

```
forms.stream()
        .map(FoodSaveForm::toFood)
        .forEach(FoodRepository::save);
```

자바8에서 추가된 Map 메서드인 merge 메서드는 키, 값, 함수를 인수로 받아 주어진 카기 맵 안에 없다면 키/값 그대로 저장하고 이미 있다면 세번째 인자로 받은 함수에 기존 값과 새로운 값을 전달해 나온 결과로 덮어씌운다.

- Map 타입 객체에 첫 번째 인자인 key를 매핑한다.
- key가 Map에 존재한다면 두 번째 인자인 value와 기존의 key에 매핑되는 value를 합친다.
- 존재하지 않는다면 새로운 (key, value)를 추가한다

```
Map<Integer, Integer> map = new Map<>();
map.put(1,1);
map.put(3,5);
map.put(5,8);

map.merge(1, 2, (count, incr) -> count + incr); // {1=3, 3=5, 5=8}
map.merge(2, 1, (counc, incr) -> count + incr); // {1=3, 2=1, 3=5, 5=8}
```

> 파라미터인 count와 incr이 많은 공간을 차지한다.

위 더하기 함수식은 자바8 이후부터 Int의 박싱 타입인 Integer의 정적 메서드인 sum을 쓰는 것과 동일하다.

```
map.merge(1, 2, Integer::sum);
```

❓ 그럼 무조건 메서드가 참조가 좋은것인가?

> 그건 또 아니다. 어떤 람다에서는 파라미터 이름이 곧 문서이기도 하기 때문이다.
> 즉 람다가 메서드 참조보다 더 길어도 유지보수 측면이나 가독성에서 도움이 될 수 있다.

메서드 레퍼런스 타입	예제	동일한 표현식을 람다로 바꾸면?8
정적(Static)	Integer::parseInt	str -> Integer.parseInt(str)
바운드(Bound)	Instant.now()::isAfter	Instant then = Instant.now(); t -> then.isAfter(t);
언바운드(UnBound)	String::toLowerCase	str -> str.toLowerCase();
클래스 생성자	TreeMap<K,V>::new	() -> new TreeMap<K,V>();
배열 생성자	int[]::new	len -> new int[len];

🚀 아이템 44. 표준 함수형 인터페이스를 사용하라

❓ 함수형 인터페이스란?

> **하나의 추상 메서드**만 갖고 있는 인터페이스를 뜻합니다.

```
public interface FuncitionalInterface{
  void action();
}
```

❓ 왜 사용하는거지?

🌟 **객체지향 프로그래밍 언어인 자바는 Java8 부터 함수형 프로그래밍을 지원하기 시작했다.**

> 함수형 프로그래밍은 모든 것이 함수인 것을 말합니다. 즉 모든 것을 `f(x) = y` 라고 표현이 가능하게 됐습니다.

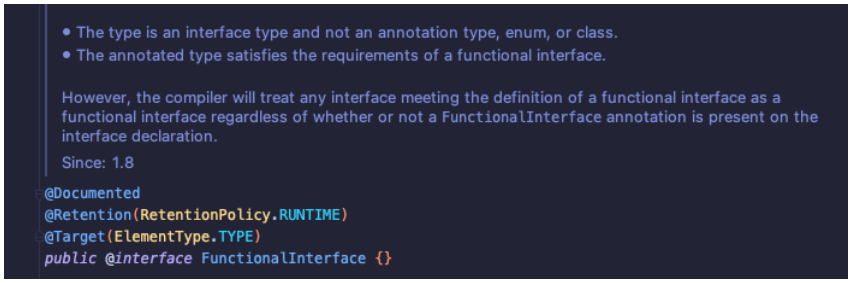
> 이러한 것들의 장점으로 `input` `output` 이 정해져 있어 테스트하기 쉽고 명확해지며 사이드 이펙트를 없애줍니다.
>
> 그렇기에 자바는 이러한 장점을 가진 함수형 프로그래밍을 지원하기 위해서 Java8부터 함수형 인터페이스의 인스턴스를 표현할 수 있는 수단이자 익명 한수인 람다식과 람다식 타입을 표현해줄 수 있는 함수형 인터페이스를 지원하게 됐습니다.

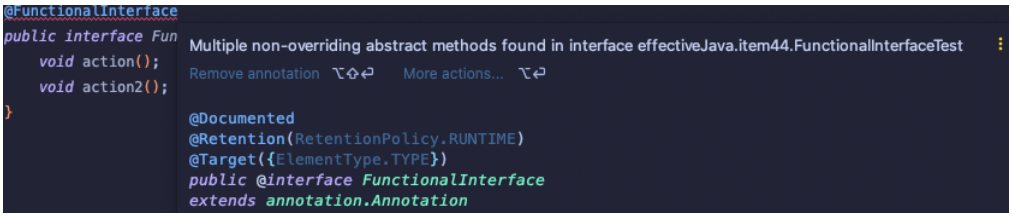
❓ 그럼 왜 추상 메서드가 하나만있는거지?

> 람다식 자체가 익명 함수를 뜻하는데 이름 없는 함수, 즉 함수 그 자체를 의미를 가지기 때문에 하나의 추상메서드만 있어야 합니다.
> 만약 2개 이상의 추상 메서드를 선언할시 컴파일에러가 뜨게 됩니다.

✅ 함수형 인터페이스에서 하나의 추상 메서드만 있다는 것을 보장하는 방법

> `@FunctionalInterface` 어노테이션을 이용해서 보장합니다.





위 사진과 같이 2개 이상의 추상 메서드를 정의할시 컴파일 에러가 뜹니다.

❓ 표준 함수형 인터페이스란?

> 몇 가지 유용한 함수형 인터페이스를 자바 진영에서 미리 만들어 놓은 것이 표준 함수형 인터페이스이다.
>
> `java.util.function` 패키지에 총 43개의 함수형 인터페이스가 있지만 이 것을 모두 외울수는 없지만 기본 인터페이스에서 몇 가지 정해진 규칙으로 파생된 것이 때문에 어떤게 있을지 어느정도 예측이 가능하다.

인터페이스	함수 시그니처	예
UnaryOperator<T>	T apply(T t)	String::toLowerCase
BinaryOperator<T>	T apply(T t1, T t2)	BigInteger::add
Predicate<T>	boolean test(T t)	Collection::isEmpty
Function<T, R>	R apply(T t)	Arrays::asList
Supplier<T>	T get()	Instant::now
Consumer<T>	void accept(T t)	System::out::println

- UnaryOperator는 반환값과 인수(1개)의 타입이 같습니다
  - T로 타입을 줘서 인수와 반환값이 모두 같은 타입이 되도록 했습니다

- BinaryOperator는 반환값과 인수(2개)의 타입이 같습니다
  - T, T로 타입을 줘서 인수와 반환값이 모두 같은 타입이 되도록 했습니다
- Predicate는 인수 하나로 boolean을 반환합니다
- Function은 인수 하나와 반환값이 있는데 서로의 타입이 다릅니다
- Supplier는 인수를 없으며 반환값이 있습니다
- Consumer는 인수 하나를 받고 반환값이 없습니다

이런 기본 인터페이스에서 `int` `long` `double` 각 3개씩 변형이 생기는데 접두로 해당 타입이 붙게됩니다.

- `Predicate` → `IntPredicate` `LongPredicate` `DoublePredicate`

```
// 숫자가 짝수인지 검증하는 함수형 인터페이스

//before
Predicate<Integer> isEvenV1 = value -> value % 2 == 0;
//after
IntPredicate isEvenV2 = value -> value % 2 == 0;
```

🌟 **이러한 표준 함수형 인터페이스를 사용하면 유지보수 할 때도 편하고 표준이기에 다른 개발자와의 소통도 쉽습니다.**

✅ 직접 합수형 인터페이스를 작성해야 하는 경우

> 항상 기본으로만 제공하는 인터페이스로 모든 것을 해결하나갈 수는 없다는 사실을 우리는 이미 앞서 경험해봤습니다.
>
> 예를들면 `Predicate` 를 사용하는데 매개변수가 3개 이상을 넣어야 한다면? 기본적으로 제공된 함수는 인수를 1개만 제공하고 있기에 직접 만들어야한다!

자바에서의 예시 Comparator 함수형 인터페이스

간단하게 로직만 파악하자면 `ToIntBiFUnction<T, U>` 와 같습니다. 그럼에도 `Comparator<T>` 를 사용해야만 하는 이유는 다음과 같습니다.

- 사용빈도가 높으며 이름 자체로 용도를 명확히 설명해준다.
- 구현하는 쪽에서 반드시 지켜야 할 규약을 담고 있다.
- 비교자들을 변환하고 조합해주는 유용한 디폴트 메세지를 갖고 있다.

⛔ 함수형 인터페이스 사용 시 주의사항

> 함수형 인터페이스에서도 오토박싱, 언박싱이 이뤄집니다.
> 때문에 리턴 타입이 기본타입이 아니라 박싱된 객체의 경우 성능 차이가 심각하니 생각하고 사용해야한다!

🚀 아이템 45. 스트림은 주의해서 사용하라

❓ 스트림(stream)이란?

> 자바 8에서 람다와 같이 나온 API로 다량의 데이터 처리 작업을 도와주며 다음과 같은 추상 개념을 가집니다.

- 데이터 **원소의 유한 혹은 무한 시퀀스**를 뜻하는 스트림
- 원소들로 수행하는 연산 단계를 표현하는 **스트림 파이프 라인**

✅ 스트림 파이프 라인

- 소스 스트림 - 중간 연산(스트림을 변환) : 종단 연산
- 지연 평가(Lazy evaluation) :종단 연산이 수행될 때 평가된다.
  → 종단 연산이 수행되지 않으면 아무일도 일어나지 않는다.
- 일단 한 값을 다른 값에 매핑하고 나면 원래의 값은 잃는 구조

❓ 지연 평가란?

> 그때 그때 값을 평가하지 않고, 정말 결과값이 필요한 시점까지 평가를 미루는 것

- 필요할 때만 평가가 되므로 메모리를 효율적으로 사용할 수 있다.
- 무한 자료구조를 만들 수 있음
- 런타임 에러를 방지 할 수 있다. → 컴파일 시 에러를 체크
- 컴파일러 최적화 가능

✅ 스트림 적용 전 후

- 사전 파일에서 단어를 읽어온 다음 사용자가 문자의 길이보다 큰 원소 수를 가진 아나그램 그룹을 출력한다.
  - 아나그램이란 철자를 구성하는 알파벳이 같고 순서만 다른 단어
- 맵의 키는 각 단어를 구성한느 철자들을 알파벳순으로 정렬한 값이다.

스트림 없이 구현한 Anagram

```
public class Anagrams {
    public static void main(String[] args) throws IOException {
        int minGroupSize = 4;
        final Map<String, Set<String>> groups = Anagrams.getGroups("src/main/java/me/catsbi/effectivejavastudy/chapter6/item45/dictionary.txt");

        for (Set<String> value : groups.values()) {
            if (value.size() >= minGroupSize) {
                System.out.println(value.size() + ": "+ value);
            }
        }
    }

    public static Map<String, Set<String>> getGroups(String filePath) throws IOException {
        File dictionay = new File(filePath);
        Map<String, Set<String>> groups = new HashMap<>();

        try(Scanner sc = new Scanner(dictionay)){
            while (sc.hasNext()) {
                String word = sc.next();
                groups.computeIfAbsent(alphabetize(word), (unsued) -> new TreeSet<>()).add(word);
            }
        }
        return groups;
    }

    private static String alphabetize(String word) {
        final char[] chars = word.toCharArray();
        Arrays.sort(chars);
        return new String(chars);
```

```
    }
}
```

적절히 활용한 스트림

```
public class Anagrams {
    public static void main(String[] args) throws IOException {
        Path dictionary = Paths.get(args[0]);
        int minGroupSize = Integer.parseInt(args[1]);

        try (Stream<String> words = Files.lines(dictionary)) {
            words.collect(groupingBy(word -> alphabetize(word)))
                    .values().stream()
                    .filter(group -> group.size() >= minGroupSize)
                    .forEach(group -> System.out.println(group.size() + ": " + group));
        }
    }

    private static String alphabetize(String s) {
        char[] a = s.toCharArray();
        Arrays.sort(a);
        return new String(a);
    }
}
```

`alphabetize` 메서드에서는 스트림을 사용하지 않았다. 그 이유는 char의 스트림은 자바는 지원하지 않고 있기 때문이다.

즉 char는 스트림을 삼가는게 좋다.

❓ 그럼 스트림은 언제 사용하는게 좋을까?

> 기존 코드는 스트림을 사용하도록 리팩터링 하지만 확실하게 가독성이 더 좋아보일때 해야한다.
>
> 반복 코드에서는 코드 블록을 이용하고 스트림 파이프라인에서는 되풀이되는 계산을 함수 객체로 표현을 하는데, 두 방법이 가진 차이점을 고려해 적절히 선택하자

- 코드 블럭에서는 범위 안의 지역 변수를 읽고, 수정할 수 있지만 람다는 `final` 만 읽을 수 있고 지역변수를 수정하지 못한다.
- 코드 블럭은 `return` `break` `continue` 문을 이용해 코드 반복을 제어(종료, 건너뛰기)할 수 있다.

  → 하지만 람다는 불가능하다.

아래의 경우는 스트림을 사용하기 좋은 예시이다.

- 원소들의 시퀀스를 일관되게 변한하는 경우
- 시퀀스를 필터링한다.
- 시퀀스를 하나의 연산을 사용해 결합한다(사칙연산, 최소 최대 구하기)
- 시퀀스를 컬렉션에 모은다.
- 시퀀스에서 특정 조건을 만족하는 원소를 찾는다.

📋 정리

> 반복 코드와 스트림 둘 다 과도하게 하나의 방식만 고집할게 아닌 상황에 맞게 사용하면 된다.
>
> 스트림과 반복 중 어느 쪽이 나은지 확신하기 어렵다면 둘 다 해보고 더 나은쪽을 택하는 방법밖에 없다.

🚪 6장 모든 객체의 공통 메서드