EJ

이펙티브 자바 CP.6

작성 일시	@2023년 2월 14일 오후 7:57
최종 편집 일시	@2023년 2월 18일 오전 1:36
유형	이펙티브 자바
작성자	
참석자	

6️⃣ 람다와 스트림

▼ 선행 내용

- [JAVA] 람다, 스트림 기본 개념
- [JAVA] Stream 설명

▼ 42. 익명 클래스보다 람다를 사용하라.

- 익명 클래스
  - 이전 자바에서 함수 타입을 표현할 때 추상 메서드를 하나만 담은 인터페이스를 사용했다. 이런 **인터페이스의 인스턴스를 함수 객체**라고 하여, 특정 함수나 동작을 나타나는대 사용했다. JDK 1.1 등장 이후 함수 객체를 만드는 주요 수단은 익명 클래스(아이템 24)가 되었다.
  - ex) 익명 클래스의 인스턴스를 함수 객체로 사용 (문자열 정렬)

```
Collections.sort(word, new Comparator<String>() {
    pulbic int compare(String s1, String s2) {
        return Integer.compare(s1.length(), s2.length());
    }
});
```

        - 전략 패턴 처럼, 함수 객체를 사용하는 과거 객체 지향 디자인 패턴에는 익명 클래스면 충분했다.
            - `Comparator` 인터페이스가 정렬을 담당하는 추상 전략을 뜻하며, 문자열을 정렬하는 구체적인 전략을 익명 클래스로 구현했다.
            - 하지만 익명 클래스 방식은 코드가 너무 길어지기 때문에, 함수형 프로그래밍에 적합하지 않았다.
    - JDK 8 에서는 **추상 메서드 하나를 담는 인터페이스를 함수형 인터페이스**라 부르며, 이 인터페이스들의 인스턴스를 람다식(Lambda) 을 사용해 만들 수 있게 된것이다.
- 람다식
    - ex) 람다식을 함수 객체로 사용 (문자열 정렬)

```
Collections.sort(words,
    (s1, s2) -> Integer.compare(s1.length(), s2.length());
```

        - 람다, 매개변수 (`s1, s2`), 반환 값 타입을 명시하지 않았다.
          (`Comparator<String>`, `String`, `int`)
        - 명시하지 않았지만, 컴파일러가 대신 문맥을 살펴 **타입 추론**을 해준 것이다.
          모든 상황에 추론이 되는 것은 아님으로 명시해야할 수도 있다.
        - **타입을 명시해야 코드가 더 명확할 때만 제외하고는, 람다의 모든 매개변수 타입은 생략하자.**
          생략 후 컴파일러가 “타입을 알 수 없다” 오류를 낼 때, 타입을 명시하면 된다.

> **타입 추론**
> 아이템 26 - 제네릭 raw 타입 쓰지 말라, 아이템 29 - 제네릭을 쓰라, 아이템 30 - 제네릭 메서드를 쓰라 했었다. 이 내용들은 람다와 함께 쓸 때 두 배로 중요해진다.

> **컴파일러가 타입을 추론하는 데 필요한 타입 정보 대부분을 제네릭에서 얻기 때문이다.**
> 우리가 이 정보를 제공하지 않으면 컴파일러는 람다의 타입을 추론할 수 없게 되어, 일일이 명시해야 한다.
> 좋은 예로, 람다식 ex) 에서 `words` 라는 매개변수가 `List<String>` 이 아닌 `List (raw type)`이였다면, 컴파일 오류가 났다.

- 람다식 + 비교자 생성 메서드 (아이템 14, 43)

```
Collections.sort(words, Comparator.comparingInt(String::length));
```

- `List` 인터페이스의 `sort` 메서드

```
words.sort(Comparator.comparingInt(String::length));
```

  - `::` 이중 콜론 연산자는 아이템 43에서 소개한다.

- ex) 이전 상수별 클래스 열거 타입 를 람다로 표현

```
public enum Operation {
    PLUS ("+", (x,y) -> x + y),
    MINUS ("-", (x,y) -> x - y),
    TIMES ("*", (x,y) -> x * y),
    DIVIDE ("/", (x,y) -> x / y);

    private final String symbol;
    private final DoubleBinaryOperator op;

    Operation(String symbol, DoubleBinaryOperator op) {
        this.symbol = symbol;
        this.op = op;
    }

    @Override
    public String toString() {
        return symbol;
    }

    public double apply(double x, double y) {
        return op.applyAsDouble(x, y);
    }
}
```

    - 람다를 `DoubleBinaryOperator` 인터페이스 변수에 할당했다.
      java 가 지원하는 다양한 함수 인터페이스 (아이템 44) 중 하나이다.

- 람다의 단점
    - **메서드나 클래스와 달리 람다는 이름이 없고 문서화도 할 수 없다. 따라서 코드 자체로 동작이 명확히 설명되지 않거나 코드 줄 수가 많아지면 람다를 사용하면 안된다.**
        - 람다 내용이 길거나 (3줄 이상) 읽기 어렵다면 더 줄이거나, 람다를 사용하지 않는 쪽으로 리팩터링해라.
        - 열거 타입 생성자에 넘겨지는 인수들의 타입도 컴파일 타임에 추론된다.
          따라서 열거 타입 생성자 안의 람다는 열거 타입의 인스턴스 멤버에 접근 할 수 없다.
          (인스턴스는 런타임에 만들어지기 때문임)
        - **람다는 함수형 인터페이스에서만** 쓰인다.
          추상 클래스의 인스턴스를 만들 때, 람다를 쓸 수 없으니, 익명 클래스를 써야 한다.
          추상 메서드가 여러 개인 인터페이스의 인스턴스를 만들 때도 익명 클래스를 쓸 수 있다.
        - 람다는 자신을 참조할 수 없다. **람다에서 `this` 는 바깥 인스턴스를 가르킨다.**
          익명 클래스에서 `this` 는 인스턴스 자신을 가르킴으로, 함수 객체가 자신을 참조해야 한다면 반드시 익명클래스를 써야한다.

- 람다의 주의점
    - 람다도 익명 클래스 처럼 직렬화 형태가 구현별로 다를 수있다. 따라서 람다에서 직렬화 하는일은 삼가하자 (익명 클래스의 인스턴스도 마찬가지)
    - 직렬화해야하는 함수 객체가 있다면 (`Comparator` 처럼) `private` 정적 중첩 클래스 (아이템 24)의 인스턴스를 사용하자

- 정리
    - jdk 8 - 함수 객체를 구현하는데 람다가 도입되었다.
      람다는 함수 객체를 아주 쉽게 표현할 수 있어, 코드가 명확하고 간결해 질 수있다.
    - 익명 클래스는 (함수형 인터페이스가 아닌) 타입의 인스턴스를 만들 때만 사용하자.

▼ 43. 람다보다는 메서드 참조를 사용하라.

- 이중 콜론 연산자 (::)
  - java 8 에서 추가된 메서드 참조 연산자이다.
  - 람다식에서 파라미터를 중복해서 사용하고 싶지 않을 때, 사용하고 람다식과 동일한 처리 방법을 갖지만, 이름으로 기존 메소드를 참조함으로써 더욱 간결하다.
  - 사용 방법
    - `[인스턴스]::[메소드명(or new)]`
    - `User::getId`
    - 람다 표현식이 () → {} 에서만 가능하다
    - static 메서드인 경우 인스턴스 대신 클래스 이름으로도 사용할 수 있다
- 메서드 참조 (Method Reference)
  - 함수 객체를 람다보다도 더 간결하게 만들수 있다.
  - ex) 임의의 키와 `Integer` 값의 매핑을 관리하는 프로그램 (`merge` 메서드를 잘 쓴 예)
    이때 값이 키의 인스턴스 개수로 해석된다면, 이 프로그램은 멀티셋을 구현한게 된다.

    ```
    map.merge(key, 1, (count, incr) -> count + incr);
    ```

    - 키가 맵안에 없다면 키와 숫자 1을 매핑하고, 이미 있다면 기존 매핑 값을 증가 시킨다.
    - 자바 8 때 `Map` 에 추가된 `merge` 메서드를 사용했다.
      `merge` 메서드는 키, 값, 함수를 인수로 받으며, 주어진 키가 맵 안에 아직 없다면 주어진 {키, 값} 쌍을 그대로 저장한다.
      반대로 키가 이미 있다면 (세 번째 인수로 받은) 함수를 현재 값과 주어진 값에 적용한 다음, 그 결과로 현재 값을 덮어쓴다.
      즉, 맵에 {키, 함수의 결과} 쌍을 저장한다.
    - 매개변수 count, incr 은 크게 할 일 없어 보인다.
      사실 이 람다는 두 인수의 합을 단순히 반환할 뿐이다.
      자바 8 때 Integer 클래스 (와 모든 기본 타입의 박싱 타입)는 이 람다와 기능이 같은 정적 메서드 sum 을 제공하기 시작했다. 따라서 더 간결하게 작성 가능하다.

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

        - 또한 매개변수 수가 늘어날수록 메서드 참조로 제거할 수 있는 코드양도 늘어난다.
    - 어떤 람다에서는 매개변수의 이름 자체가 프로그래머에게 좋은 가이드가 되기도 한다.
      이런 람다는 길이는 더 길지만, 메서드 참조보다 읽기 쉽고 유지보수가 쉬울 수 있다.
    - 람다로 할 수 없는 일이라면, 메서드 참조로도 할 수 없다.
      그렇더라도 메서드 참조를 사용하는 편이 보통 더 짧고 간결함으로, 람다로 구현했을 때 너무 길거나 복잡하면 메서드 참조가 좋은 대안이 되어준다.
- 람다가 더 좋을 때
    - ex) `GoshThisClassNameIsHumongous` 클래스

```
// 메서드 참조
service.execute(GoshThisClassNameIsHumongous::action);
// 람다
service.execute(() -> action());
```

        - 메서드 참조 쪽은 더 짧지도, 명확하지도 않다. 람다쪽이 보다 좋다.
- **메서드 참조 유형 5가지**
    1. 정적 메서드를 가리키는 메서드 참조
    2. 인스턴스 메서드를 참조하는 유형
        a. 수신 객체 (참조 대상 인스턴스)를 특정하는 한정적 인스턴스 메서드 참조
            - 근복적으로 정적 참조와 비슷하다.
            - 즉, 함수 객체가 받는 인수와 참조되는 메서드가 받는 인수가 똑같다.
        b. 수신 객체 (참조 대상 인스턴스)를 특정하지 않는 비한정적 인스턴스 메서드 참조
            - 함수 객체를 적용하는 시점에 수신 객체를 알려준다.
            - 이를 위해 수신 객체 전달용 매개변수가 매개변수 목록의 첫 번째로 추가되며, 그 뒤로는 참조되는 메서드 선언에 정의된 매개변수들이 뒤따른다.

        - 주로 스트림 파이프라인에서의 매핑과 필터 함수로 쓰인다. (아이템 45)
    3. 클래스 생성자를 가르키는 메서드 참조
    4. 배열 생성자를 가르키는 메서드 참조

메서드 참조 유형	람다	메서드 참조
정적	str -> Integer.parseInt(str)	Integer::parseInt
한정적 (인스턴스)	Instant then = Instant.now(); t -> then.isAfter(t)	Instant.now()::isAtfer
비한정적 (인스턴스)	str -> str.toLowerCase()	String::toLowerCase
클래스 생성자	() -> new TreeMap<K, V> ()	TreeMap<K,V>::new
배열 생성자	len -> new int[len]	int[]::new

- 정리
    - 메서드 참조는 람다의 간단명료한 대안이 될 수 있다.
    - 메서드 참조 쪽이 짧고 명확하면, 메서드 참조, 그 반대이면 람다를 쓰자.
    - 람다를 쓰고 Intellij 에서는 Replace Lambda with Method reference 되기 때문에, 람다를 작성하고 해당 IDE 기능을 사용해서 보다 나은게 어떤건지 확인후 적용하자.

▼ 44. 표준 함수형 인터페이스를 사용하라.

- 람다 지원 후 API 작성 변경
    - 상위 클래스의 기본 메서드를 재정의해 원하는 동작을 구현하는 템플릿 메서드 패턴이 줄었다.
    - 대신, 같은 효과의 함수 객체를 받는 정적 팩터리나 생성자를 제공
    - 일반화 해서 말하자면, 함수 객체를 매개변수로 받는 생성자와 메서드를 더 많이 만들어야 한다. 이때 함수형 매개변수 타입을 올바르게 선택해야 한다.
- `LiskedHash`
    - 이 클래스의 `protected` 메서드인 `removeEldestEntry` 를 재정의하면 캐시로 사용할 수 있다. `put` 메서드는 `removeEldestEntry`를 호출하여 `true`가 반환되면 맵에서 가장 오래된 원소를 제거한다.

```
@Override
protected boolean removeEldestEntry(Map.Entry<K, V> eldest) {
    return size() > 100;
}

// 람다 적용 - 불필요한 함수형 인터페이스
@FunctinalInterface
interface EldestEntryRemovalFunction<K, V> {
    boolean remove(Map<K,V> map, Map.Entry<K,V> eldest);
}
```

    - `removeEldestEntry`는 인스턴스 메서드임으로, 람다로 작성시 (팩터리나 생성자를 호출할 때는, 맵의 인스턴스가 존재하지 않음), 맵은 자기 자신도 함수 객체에 건네줘야 함으로 `Map<K,V> map` 도 인수로 필요하다.
    - 람다로 적용된 메소드를 굳이 사용할 필요는 없다. 자바 표준 라이브러리에 이미 같은 모양의 인터페이스가 준비되어 있다.
    - **필요한 용도에 맞는 게 있다면, 직접 구현하지 말고 표준 함수형 인터페이스를 활용하라**

- **표준 함수형 인터페이스**
    - API가 다루는 개념의 수가 줄어들어 익히기 더 쉽다.
    - 유용한 디폴트 메서드를 많이 제공함으로, 다른 코드와의 상호운용성도 좋아진다.
        - `Predicate` 인터페이스는 `predicate` 들을 조합하는 메서드를 제공한다.
        - 위 예에서 `EldestEntryRemovalFunction` 대신 표준 인터페이스인 `BiPredicate<Map<K,V>, Map.Entry<K,V>` 를 사용 가능하다.
    - `java.util.function` 패키지에는 총 43개의 인터페이스가 있다. 전부 기억할 필요 없고, 기본 인터페이스 6개만 기억하면, 나머지를 충분히 유추해낼 수 있다. 이 기본 인터페이스들은 모두 **참조 타입용**이다.

- **기본 인터페이스**

인터페이스	함수 시그니처	Ex
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

인터페이스	함수 시그니처	Ex
Consumer<T>	void accept(T t)	System.out::println

    - `Operator` - 인수 타입과 반환 타입이 같은 함수
        - `UnaryOperator` - 인수가 1 개인 인터페이스
        - `BinaryOperator` - 인수가 2개인 인터페이스
    - `Predicate` - 인수 하나를 받아 boolean 반환하는 함수
    - `Function` - 인수와 반환 타입이 다른 함수
        - 유일하게 반환 타입만 매개변수화 된다.
        - 인터페이스의 변형은 입력과 결과의 타입이 항상 다르다.
        - 입력과 결과 타입이 모두 기본 타입이면 접두어 `SrcToResult` 를 사용한다.
            - `long` 을 받아 `int` 를 반환하면 `LongToIntFunction` 이 되는 식이다.
    - `Supplier` - 인수를 받지 않고 값을 반환(제공)하는 함수
    - `Consumer` - 인수를 하나 받고 반환 값이 없는 (인수를 소비하는) 함수
- 기본 인터페이스는 기본 타입인 `int`, `long`, `double` 용으로 각 3개씩 변형이 생기는데, 기본 인터페이스 이름 앞에 기본 타입 이름을 붙여 사용된다.
    - `Predicate` → `IntPredicate`, `BinaryOperator` → `LongBinaryOperator`
- 인수 타입을 2개 (혹은 3개)씩 받는 변형된 함수형 인터페이스
    - `Predicate<T>` → `BiPredicate<T,U>`
    - `Function<T,R>` → `BiFunction<T,U,R>`
    - `Consumer<T>` → `BiConsumer<T,U>`
- 기본 인터페이스 변형
    - 기본 타입을 반환하는 `BiFunction` 변형
        - `ToIntBiFunction<T,U>` - `int` 타입 반환
        - `ToLongBiFunction<T,U>` - `long` 타입 반환
        - `ToDoubleBiFunction<T,U>` - `double` 타입 반환
    - 기본 타입을 받는 `BiConsumer` 변형
        - `ObjIntConsumer<T>` - `int` 타입 받음
        - `ObjLongConsumer<T>` - `long` 타입 받음

        - `ObjDoubleConsumer<T>` - `double` 타입 받음
    - `boolean` 타입을 반환하는 `Supplier` 변형
        - `BooleanSupplier`

- 전부 외울 필요 없다.
  표준 함수형 인터페이스 6개를 기억하고, 범용적인 이름으로, 필요할 때 찾아 사용하자.
  기본 타입만 지원한다. 그러니 박싱된 기본 타입을 넣어 사용하지 말자. (아이템 61)
  계산량이 많을때 성능이 처참히 느려질 수 있다.
- 직접 함수형 인터페이스를 작성하는 경우
    - 표준 인터페이스 중 필요한 용도가 맞는 게 없을 때
        - ex) `Predicate` 에서 매개 변수가 3개 이상인 경우, 검사 예외를 던져야 하는 경우
    - 그런데, 구조적으로 똑같은 표준 함수형 인터페이스가 있더라도 작성해야 할 때가 있다.
        - ex) `Comparator<T>` 인터페이스
            - 구조적으로는 `ToIntBiFunction<T,U>` 와 동일하다.
            - 사용해야하는 이유
                1. API 에 자주 사용된다. 이름이 그 용도를 잘 설명한다.
                2. 구현하는 쪽에서 반드시 지켜야 할 규약을 담고 있다.
                3. 비교자들을 변환하고 조합해주는 유용한 디폴트 메서드까지 있다.
    - 고민점 (이 중 하나 이상 만족한다면, 전용 함수형 인터페이스를 구현을 고민해봐야 한다.)
        - 자주 쓰이며, 이름 자체가 용도가 명확함.
        - 반드시 따라야 하는 규약이 있다.
        - 유용한 디폴트 메서드를 제공한다.
- `@FunctionalInterface` 애너테이션
    - 인터페이스가 함수형 인터페이스로 사용됨을 알려주는 애너테이션
    - 목적
        1. 사용자에게 이 인터페이스가 람다용으로 설계된 것임

2. 추상 메서드가 하나만 있어야 함을 컴파일되게 해준다.
3. 유지보수 과정에서 실수로 메서드를 추가하지 못하도록 해준다.

- 직접 만든 함수형 인터페이스에는 항상 `@FunctionalInterface` 애너테이션을 사용하자

- 함수형 인터페이스 API 사용시 주의점
  - 서로 다른 함수형 인터페이스를 같은 위치의 인수로 받는 메서드들을 다중 정의해서는 안된다.
  - 클라이언트에게 불필요한 모호함을 주며, 이 모호함이 에러를 발생하기도한다.
- 정리
  - JDK 8(람다) 이후 - API 설계 시 람다를 염두에 두고 설계해라
  - 입력값과 반환값에 함수형 인터페이스 타입을 활용해라
  - 보통 표준 함수형 인터페이스를 사용하는게 베스트이다.
  - `Comparator` 처럼 직접 새로운 함수형 인터페이스를 만드는 편이 좋을 수 있다.

▼ 45. 스트림은 주의해서 사용하라.

▼ 46. 스트림에서는 부작용 없는 함수를 사용하라.

▼ 47. 반환 타입으로는 스트림보다 컬렉션이 낫다.

▼ 48. 스트림 병렬화는 주의해서 사용하라.