

# 이펙티브 자바 CP.6

🕒 작성 일시	@2023년 2월 14일 오후 7:57
🕒 최종 편집 일시	@2023년 2월 15일 오후 9:47
📄 유형	이펙티브 자바
👤 작성자	
👥 참석자	

## 6 람다와 스트림

### 선행 내용

- 42. 익명 클래스보다 람다를 사용하라.
- 43. 람다보다는 메서드 참조를 사용하라.
- 44. 표준 함수형 인터페이스를 사용하라.
- 45. 스트림은 주의해서 사용하라.
- 46. 스트림에서는 부작용 없는 함수를 사용하라.
- 47. 반환 타입으로는 스트림보다 컬렉션이 낫다.
- 48. 스트림 병렬화는 주의해서 사용하라.

## 6 람다와 스트림

### ▼ 선행 내용

- [JAVA] 람다, 스트림 기본 개념

### ▼ 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 나타내는 데 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>() {
    public 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 8에서는 추상 메서드 하나를 담은 인터페이스를 함수형 인터페이스라 부르며, 이 인터페이스들의 인스턴스를 람다식(Lambda)을 사용해 만들 수 있게 되었다.
- 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)` 이었다면, 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.

- ▼ 44. 표준 함수형 인터페이스를 사용하라.
- ▼ 45. 스트림은 주의해서 사용하라.
- ▼ 46. 스트림에서는 부작용 없는 함수를 사용하라.
- ▼ 47. 반환 타입으로는 스트림보다 컬렉션이 낫다.
- ▼ 48. 스트림 병렬화는 주의해서 사용하라.