7장 람다와 스트림

아이템 42. 익명 클래스보다는 람다를 사용하라.

- **익명클래스 방식**
  - 코드가 너무 길기 때문에 함수형 프로그램이에 적합하지 않다.
- **람다 방식**
  - 함수를 간단한 식으로 표현하는 방법(익명 함수)
  - 컴파일러가 문맥을 살펴 타입 추론, **타입을 명시해야 코드가 더 명확할 때를 제외하고는 람다의 모든 매개변수 타입은 생략하자.**

  익명 클래스 방식의 예

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

람다식의 예

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

컴파일러가 문맥을 살펴 타입을 추론할 수 있기 때문에 코드에 언급이 되어있지 않지만, 컴파일러가 타입을 결정하지 못할 때는 프로그래머가 직접 명시해야한다.

- 비교자 생성 메서드

```
Collections.sort(words, Comparator.comparingInt(String::lengt
```

- List 인터페이서의 sort 메서드

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

words 리스트를 String길이를 기준으로 오름차순 정렬

Collection.sort를 사용할 필요 없이 리스트에 직접 정렬 메서드를 호출할 수 있게 되었다.

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

    private final String symbol;

    Operation(String symbol) {
        this.symbol = symbol;
    }

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

    public abstract double apply(double x, double y);
}
```

- 해당 코드에서 apply 동작이 상수마다 달라야하기 때문에 각 메서드를 재정의하였다.

```
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 기반으로한 열거 타입을 보면 더이상 상수별 클래스는 사용할 필요가 없음을 알 수 있다. 하지만 예외는 존재한다.

- **코드 자체로 동작이 명확히 설명되지 않거나 코드가 길어지는 경우**
    - 람다는 이름도 없고 문서화도 할 수 없기 때문에 동작을 명확히 설명할 수 없는 경우나 코드가 길어지느 경우는 람다를 쓰지 않아야한다.(람다는 한 줄일 때 가장 좋고 길어야 세 줄안에 끝내야 한다.)
- **인스턴스 필드나 메서드를 사용해야만 하는 경우**
    - 열거 타입 생성자에 넘겨지는 인수들의 타입은 컴파일 시점에 추론된다. 반면에, 인스턴스는 런타임에 만들어 진다. 그러므로 열거 타입 생성자 안의 람다는 열거 타입

의 인스턴스 멤버에 접근을 할 수 없다. 인스턴스 필드나 메서드에 접근할려면 인스턴스가 완전히 생성된 후인, 즉 메서드 내에서 람다식을 정의하거나 사용해야한다.

반드시 익명 클래스를 사용해야하는 경우

- **함수형 인터페이스가 아닌 경우**
  - 람다는 함수형 인터페이스에서만 쓰이기 때문에 추상 클래스의 인스턴스를 만들 때는 람다 대신 익명 클래스를 사용해야한다.
- **추상 메서드가 여러개인 인터페이스의 인스턴스를 만드는 경우**
- **함수 객체가 자기 자신을 참조해야하는 경우**
  - 람다에서의 this는 바깥 인스턴스를 가리킨다.
  - 반면, 익명 클래스의 this는 익명클래스의 인스턴스 자신을 가리키기 때문에 함수 객체가 자신을 참조해야한다면 반드시 익명 클래스를 사용해야한다.

주의점

- 람다도 익명 클래스 처럼 직력화 형태가 다를 수 있으므로 람다를 직렬화하는 일은 극히 삼가야한다.

아이템 43. 람다보다는 메서드 참조를 사용하라.

람다가 익명 클래스보다 나은 점 중에서 가장 큰 특징은 **간결함**이다.

자바에서는 함수 객체를 람다보다 더 간결하게 만드는 방법이 존재하는데 그것이 바로 메서드 참조이다.

```
map.merge(key, 1L, (count, incr) -> count + incr);
//해당 람다식은 메서드 참조로 바꾸면 더욱 간결해진다.

map.merge(key, 1L, Long::sum);
//Long::sum은 (a, b) -> a + b와 같은 역할을 한다.

service.execute(() -> action());
//이와 같이 메서드 참조가 람다보다 더 긴 경우가 있는데,
```

```
 //메서드와 람다가 같은 클래스에 있는 경우가 그렇다.
service.execute(UseMethodReference::action);
```

메서드 참조는 람다의 간단명료한 대안이 될 수 있다. 메서드 참조가 짧고 명확하다면 메서드 참조를 사용하고 그렇지 않을 때만 람다를 사용하자.

아이템 44. 표준 함수형 인터페이스를 사용하라.

함수형 인터페이스

- 람다의 지원으로 인해 상위 클래스의 기본 메서드를 재정의해 원하는 동작을 구현하는 템플릿 메서드 패턴의 매력이 줄었다.
    - 템플릿 메서드
        - 기본적인 동작을 제공하는 상위 클래스를 만들고, 하위 클래스에서 특정 메서드를 오버라이딩하여 세부 동작을 구현하는 방식이다.
        - 이 패턴은 효과적이지만, 상속을 필요로 하기 때문에 클래스의 구조가 고정될 수 있으며, 코드 재사용과 동작 변경에 제약이 따른다. 그리고 새로운 동작을 추가하려면 새로운 하위 클래스를 만들어야 하므로 클래스 수가 증가 할 수 있다.
- 이를 대체하는 방법으로 함수 객체를 받는 정적 팩터리나 생성자를 제공하는 방법이 있다.즉, 함수 객체를 매개변수로 받는 생성자와 메서드를 더 많이 사용하는 것이다.

```
// 기존
public class LinkedHashMap<K,V> extends HashMap<K,V> implemen
    ...
    protected boolean removeEldestEntry(Map.Entry<K,V> eldest
        return false;
    }
    ...
}

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

```
//size() > 100을 만족할 때 가장 오래된 엔트리를 제거하게 함으로써 Linke
//최대 크기를 100으로 제한할 수 있다.
```

- 함수형 인터페이스로 나타내기

```
@FunctionalInterface
interface EldestEntryRemovalFunction<K, V> {
    boolean remove(Map<K, V> map, Map.Entry<K, V> eldest);
}
//람다식으로 간결하게 표현이 가능하다.
```

표준 함수형 인터페이스

- 함수형 인터페이스에 필요한 용도에 맞는게 있다면, 직접 구현하지 말고 표준 함수형 인터페이스를 활용하도록 하자.

기본 인터페이스 6개

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**

- UnaryOperator<T>
  - 입력값을 받아 동일한 타입의 결과를 반환하는 함수형 인터페이스
  - 예시) 문자열을 소문자로 변환, 숫자에 특정 연산 적용
- BinaryOperator<T>
  - 동일한 타입의 두 값을 받아 동일한 동일한 타입의 결과를 반환하는 함수형 인터페이스
  - 예시) 두 값을 조합하거나 합산할 때 유용하게 사용
- Predicate<T>

    - 특정 조건을 테스트하고, 조건이 만족되면 true, 아니면 false를 반환하는 함수형 인터페이스이다. 필터링 조건이나 검증에 사용
- Function<T,R>
    - 입력값을 받아 다른 타입의 결과를 반환하는 함수형 인터페이스이다. 데이터를 변환하거나 매핑할 때 자주 사용된다.
    - 예시) Array::aslist - 배열을 리스트로 변환
- Supplier<T>
    - 인수를 받지 않고, 특정 타입의 결과를 반환하는 함수형 인터페이스, 주로 값을 지연 생성하거나 공급할 때 사용
    - 예시) Instant::now - 현재 시간을 반환
- Consumer<T>
    - 입력값을 받아 어떤 동작을 수행하지만 값을 반환하지 않는 함수형 인터페이스
    - 출력이나 상태 변경과 같은 작업을 할 때 사용
        - 예시) sout

표준형 인터페이스는 대부분 기본 타입만 지원한다. 그렇다고 기본 함수형 인터페이스에 박싱된 기본 타입을 넣어 사용하지는 말자. 계산량이 많을 때 성능이 매우 느려질 수 있다.

함수형 인터페이스를 언제 구현해야할까?

- **자주 쓰이며, 이름 자체가 용도를 정확히 설명해 준다.**
- **반드시 따라야 하는 규약이 있다.**
- **유용한 디폴트 메서드를 제공할 수 있다.**

작성법

- 항상 **@FunctionalInterface 애너테이션**을 달도록 하자.
- 이는 @Override처럼 프로그래머의 의도를 명시하는 역할을 한다.
- 해당 클래스의 코드나 설명 문서를 읽을 이에게 그 인터페이스가 람다용으로 설계된 것임을 알려준다.
- 해당 인터페이스가 추상 메서드를 오직 하나만 가지고 있어야 컴파일 되게 해준다.
- 유지보수 과정에서 누군가 실수로 메서드를 추가하지 못하도록 해준다.

함수형 인터페이스 사용시 주의점

- 서로 다른 함수형 인터페이스를 같은 위치의 인수로 받는 메서드들을 다중 정의를 피해야한다.

아이템 45. 스트림은 주의해서 사용하라.

스트림의 특징

- **스트림 안의 데이터 원소들은 객체 참조나 기본타입 값(int,long,double)값이다.**
- **스트림 파이프라인(원소들로 부터 수행하는 연산 단계를 표현하는 개념)은 소스 스트림에서 시작해 종단연산으로 끝나며, 그 사이에 하나 이상의 중간 연산이 있을 수 있다.**
    - 각 중간 연산은 스트림을 어떠한 방식으로 변환한다. 예를 들어, 각 원소에 함수를 적용하거나 특정 조건을 만족 못하는 원소를 걸러낼 수 있다.
- **스트림 파이프라인은 지연 평가 된다.**
    - 평가는 종단 연산이 호출될 때 이뤄지며, 종단 연산에 쓰이지 않는 데이터 원소는 계산에 쓰이지 않는다. 이러한 지연 평가가 무한 스트림을 다룰 수 있게 해주는 열쇠이다.
- **스트림 API는 메서드 연쇄를 지원하는 플루언트API이다.**
    - 파이프라인 하나를 구성하는 모든 호출을 연결하여 단 하나의 표현식으로 완성할 수 있다.
- **기본적으로 스트림 파이프라인은 순차적으로 진행된다.**

하지만 스트림을 잘못 사용하면 읽기 어렵고 유지보수도 힘들어진다.

반복문보다 스트림에 알맞은 상황

시퀀스 : 스트림이 처리하는 데이터의 순차적인 요소들의 흐름

- 원소들의 시퀀스를 일관되게 변환한다.
- 원소들의 시퀀스를 필터링한다.
- 원소들의 시퀀스를 하나의 연산을 사용해 결합한다.

- 원소들의 시퀀스를 컬렉터에 모은다.
- 원소들의 시퀀스에서 특정 조건을 만족하는 원소를 찾는다.

스트림으로 처리하기 어려운 경우

- 한 데이터가 파이프라인의 여러 단계를 통과할 때 이 데이터의 각 단계에서의 값들에 동시에 접근하기 어려운 경우 →이미 지나간 단계에 대한 값에 접근하기 업렵다.
- 해결 방법: 매핑을 거꾸로 수행

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

    primes().map(p -> TWO.pow(p.intValueExact()).subtract(ONE
            .filter(mersenne -> mersenne.isProbablePrime(50))
            .limit(20)
            .forEach(System.out::println);

}

static Stream<BigInteger> primes() {
    return Stream.iterate(TWO, BigInteger::nextProbablePrime)
}
//메르센 소수 출력
//메르센 수는 2^p-1의 형태의 수다. 메루센 소수란 p가 소수 -> 해당 메르센


//만약 지수를 출력할려면?→ 중간 연산에서 수행한 매핑을 거꾸로 수행해 계산
 .forEach(mp -> System.out.println(mp.bitLength() + ": " + mp
```

그럼 스트림과 반복 무엇을 선택해야할까?

```
// 반복문을 사용하는 경우
public static void main(String[] args) {
    private static List<Card> newDeck() {
        List<Card> result = new ArrayList<>();
        for(Suit suit : Suit.values())
```

```
            for(Rank rank : Rank.values())
                result.add(new Card(suit, rank));

        return result;
    }
}

// 스트림을 사용하는 경우
public static void main(String[] args) {
    private static List<Card> newDeck() {
        return Stream.of(Suit.values())
                .flatMap(suit ->
                        Stream.of(Rank.values())
                                .map(rank -> new Card(suit, r
                .collect(toList());
    }
}
```

결국 개인의 취향과 프로그래밍 환경에 문제이다.

만약 확신하기 어렵다면 둘 . 다해보고 더 나은 쪽을 선택해자.

아이템 46. 스트림에서는 부작용 없는 함수를 사용하라.

스트림 패러다임의 핵심

- 핵심은 계산을 일련의 변환으로 재구성하는 부분이다.
- 각 변환 단계는 가능한 한 이전 단계의 결과를 받아 처리하는 순수함수여야한다.

순수 함수란?

- 오직 입력만이 결과에 영향을 주는 함수
- 가변 상태 참조 x
- 함수 스스로도 다른 상태를 변경하지 않음

```
  Map<String, Long> freq = new HashMap<>();
        try (Stream<String> words = new Scanner(System.in).to
```

```
            words.forEach(word -> { //forEach에서 외부 상태를 수정
                freq.merge(word.toLowerCase(), 1L, Long::sum)
            });
        }
        //스트림을 가장한 반복적 코드
        // 스트림 api의 이점을 살리지 못하며, 같은 기능의 반복적 코드보다
        //읽기 어렵고, 유지보수가 어렵다.
```

왜 forEach를 사용하면 안될까?

- 스트림에서 forEach는 요소를 돌면서 실행되는 연산의 최종작업이다.
- 보통 sout 메소드를 넘겨서 결과를 출력할 때 사용한다.
- 종료 조건이 있는 로직을 구현할 때 주의해야한다.
    - 스트림 자체를 강제적으로 종료시키는 방법은 없다. 무언가 강제적인 종료 조건에 대한 로직이 있는 for-loop를 stream.forEach()로 구현한다면 기존 for-loop에 비해 비효율이 발생한다.
- 그러므로 최종 연산으로만 사용하자. 굳이 로직이 안에 들어가지 않더라도 중간연산을 통해 충분히 수행할 수 있다.

Collector

- 스트림은 collect()메서드를 통해 값을 누적한다.
    - tolist : 스트림 요소들의 값을 List에 누적한다.
    - toset : 스트림 요소들의 값을 Set에 누적한다.
    - toMap : 스트림 요소들의 값을 Map에 누적한다.
    - toCollection(collectionFactory) : 스트림의 값을 프로그래머가 지정한 Collection에 누적한다.
- toMap

```
public enum Command {
    START("S"),
    PAUSE("P"),
    RESUME("R"),
    QUIT("Q"),;
```

```
    //각 hotkey를 key로, Command를 value로 하는 map 만들기
    private static final Map<String, Command> COMMAND_MAP =
            .collect(Collectors.toMap(  // 값을 Map에 누적한다
                    command -> command.hotkey,
                    // keyMapper : Function이 들어간다. 여기서
                    command -> command  // valueMapper Fun
            ));

    public static Command of(String hotkey) {
        return COMMAND_MAP.get(hotkey);
    }

    private final String hotkey;

    Command(String hotkey) {
        this.hotkey = hotkey;
    }
}
```

- keyMapper : 스트림 원소를 키에 매핑하는 `Function<T,R>`이 온다.
- valueMapper : 스트림 원소를 값에 매핑하는 `Function<T,R>`이 온다.
- toMap()의 keyMapper와 valueMapper를 이용해서, collect() 메서드가 스트림의 요소들을 `Map`에 누적시킨다

grounpingBy

- 해당 메서드는 입력으로 분류 함수를 받고, 출력으로는 원소들을 카테고리별로 모아 놓은 맵을 담은 Collector()을 반환한다.

minBy & maxBy

- 이는 `Collectors`에 정의되어 있으나, "수집/누적"과는 관련이 없다.
- 스트림에서 가장 작은 값, 혹은 가장 큰 값을 찾아 반환할 뿐이다.

아이템 47.반환 타입으로는 스트림보다 컬렉션이 낫다.

- 자바 7까지는 메서드의 반환 타입으로 Collection, Set,List 같은 컬렉션 인터페이스, 혹은 Iterable이나 배열을 썼다.
- 자바 8에서는 스트림이 도입되면서 선택지가 복잡해졌다
- 스트림은 반복을 지원하지 않는다.
- Stream 인터페이스는 Iterable인터페이스가 정의한 방식대로 종작한다. 그럼에도 for-each로 스트림을 반복할 수 없는 까닭은 바로 Stream이 Iterable을 확장하지 않아서 이다.

어댑터 메서드를 사용하면 스트림을 iterate로 변경할 수 있다.

```
//Stream<E>를 Iterable<E>로 중개해주는 어댑터
public static <E> Iterable<E> iterableOf(Stream<E> stream)
{
    return stream::iterator;
}
```

반대로 API가 Iterable을 받아서 스트림을 반환하는 어댑터도 구현이 가능하다.

```
//Iterabel<E>를 Stream<E>로 중개해주는 어댑터
public static <E> Stream<E> streamOf(Iterable<E> iterable){
   return StreamSupport.stream(iterable.spliterator(),fals
e);
}
```

- 어댑터 메서드 : 서로 호환되지 않는 인터페이스르 가진 두 클래스 간의 연결을 가능하게 해주는 메서드
- 어댑터 패턴
  - 기존 클래스의 인터페이스와 호환되지 않는 인터페이스가 필요한 경우
  - 호환되지 않는 클래스 간의 상호 운용성 제공
- 어댑터 메서드의 장점
  - 재사용성 증가
  - 유연성 제공
  - 유지보수 용이

Collection VS Stream

Collection

- Collection 인터페이스는 Iterable 인터페이스의 하위 타입이고 stream 메서드도 제공하니 일반적으로는 Collection을 반환하는편이 좋다.
- 하지만 Collection의 각 원소는 메모리에 올라가므로, 시퀀스의 크기가 크다면 고민해보는 것이 좋다.

Stream

- 컬렉션의 contains와 size를 시퀀스의 내용을 확정하기 전 까지 구할 수 없는 경우에는 Stream을 반환하는 것이 좋다.

아이템 48. 스트림 병렬화는 주의해서 사용하라.

동시성 프로그래밍을 . 할때는 안전성과 응답기능 상태를 유지하기 위해 애써야하는데, 병렬 스트림 파이프라인에서도 다를바 없다.

```
public static void main(String[] args) {
   primes().map(p -> TWO.pos(p.intValueExact()).subtract(ONE)
      .fileter(mersenne -> mersenne.isProbablePrime(50))
      .limit(20)
      .forEach(System.out::println);
}

static Stream<BigInteger> primes() {
   return Stream.iterate(TWO, BigInteger::nextProbablePrime);
}
```

메르센 소수를 생성하는 프로그램이다. 여기서 만약 속도를 높이고자 스트림 파이프라인의 parallel()주석을 지우고 병렬화 시킨다면 아무것도 출력하지 못하면서 CPU 점유율만 높아지는 상태가 무한히 계속되게 된다.

왜 무한히 계속 될까?

- Stream.iterate로 만든 스트림은 무한한 숫자를 계속 생성한다. 그런데 병렬처리를 하게 되면 스레드가 동시에 이 무한한 스트림을 다루면서 값을 가져오고, 계산을 시작한다. 이 과정에서 값이 무한히 생성되고, 각 스레드는 계속해서 값을 처리하려고 하면서 멈추지 않는다.

- 병렬화로 인해 스레드가 가져오는 순서가 엉키게 되면서 limit()가 제대로 작동하지 않는다.
- 여러 스레드가 무한히 값을 생성하고 계산을 반복하기 때문에, CPU 사용량은 높아지지만 실제로는 아무것도 출력하지 않는 상황이 발생한다.

`스트림 파이프라인을 마구잡이로 병렬화 하면 안 된다. 성능이 오히려 끔찍하게 나빠질 수도 있다.`

- 대체로 스트림의 소스가 ArrayList, HashMap, HashSet, ConcurrentHashMap의 인스턴스거나 배열, int, long 범위일 때 병렬화의 효과가 가장 좋다.
- —>이 자료구조들은 모두 데이터를 원하는 크기로 정확하고 손쉽게 나눌 수 있어서 일을 다수의 스레드에 분배하기에 좋다는 특징이 있다.

스트림 파이프라인의 종단 연산의 동작 방식 역시 병렬 수행 효율에 영향을 준다.

- **종단 연산에서 수행하는 작업량이 파이프라인 전체 작업에서 상당 비중을 차지하면서 순차적인 연산이라면 파이프라인 병렬 수행의 효과는 제한될 수밖에 없다. 종단 연산 중 병렬화에 가장 적합한 것은 축소다. reduce, min, max, count, sum가 속한다. anyMatch, allMatch, noneMatch처럼 조건에 맞으면 바로 반환되는 메서드도 적합하다.** 반면, 가변 축소를 수행하는 collect 메서드는 병렬화에 적합하지 않다. 합치는 부담이 너무 크다.
- collect: collect는 결과를 여러 개의 스레드가 생성한 데이터들을 모아서 하나의 최종 결과로 합치는 과정이 필요하다. 이 합치는 과정에서 스레드 간의 동기화가 필요하거나, 중간 결과를 병합하는 부담이 커진다. 예를 들어, 리스트나 맵에 데이터를 모으는 작업에서는 여러 스레드가 동일한 데이터를 병합하기 때문에 동기화가 필요해져 성능이 저하될 수 있다.

조건이 잘 갖춰지면 parallel 메서드 호출 하나로 거의 프로세서 코어 수에 비례하는 성능 향상을 만끽할 수 있다.

```
static long pi(long n) {//소수 계산 스트림 파이프라인 - 병렬화에 적합
   return LongStream.rangeClosed(2, n)
      .mapToObj(BigInterger::valueOf)
      .filter(i -> i.isProbablePrime(50))
```

```
        .count();
}
static long pi(long n) {//병렬화 버전
   return LongStream.rangeClosed(2, n)
        .parallel()
        .mapToObj(BigInteger::valueOf)
        .filter(i -> i.isProbablePrime(50))
        .count();
}
```

- 무작위 수들로 이뤄진 스트림을 병렬화하려거든 ThreadLocalRandom, SplittableRandom 인스턴스를 이용하자.
- 그냥 Random은 모든 연산을 동기화하기 때문에 병렬 처리하면 최악의 성능을 보일 것이다.

계산도 올바로 수행하고 성능도 빨라질 거라는 확신 없이는 스트림 파이프라인 병렬화는 시도조차 하지 말자. 잘못되면 오동작하게 하거나 성능을 급격히 떨어뜨린다. 테스트 후에, 계산도 정확하고 성능도 좋아졌음이 확실해졌을 때, 오직 그럴 때만 병렬화 버전 코드를 운영 코드에 반영해보자.