람다와 Stream API

✅명령형 프로그래밍 vs 선언형 프로그래밍

▷ 명령형 프로그래밍의 예시

```
List<Integer> numbers = Arrays.asList(1, 3, 21, 10, 8, 11);
int sum = 0;

for(int number : numbers){
    if(number > 6 && (number % 2 != 0)){
        sum += number;
    }
}
```

▷ 선언형 프로그래밍의 예시

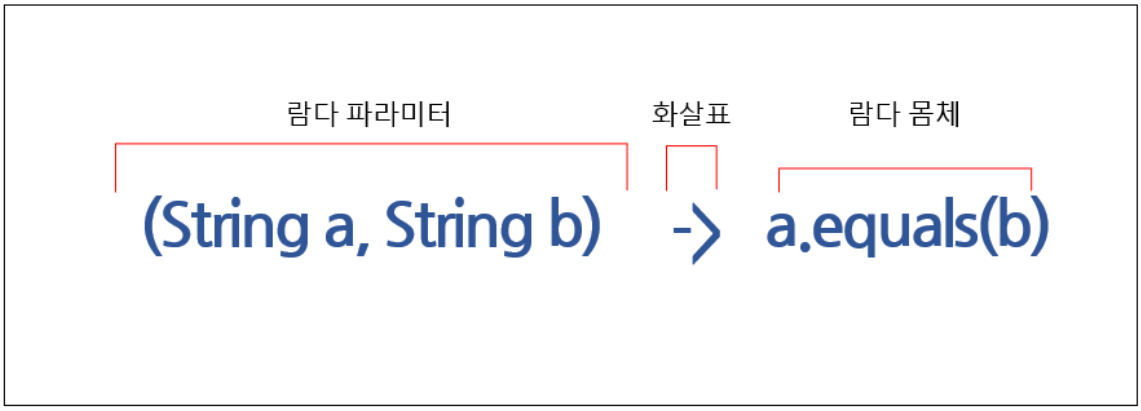
```
List<Integer> numbers = Arrays.asList(1, 3, 21, 10, 8, 11);

int sum = numbers.stream()
        .filter(number -> number > 6 && (number % 2 != 0))
        .mapToInt(number -> number)
        .sum();
```

- 코드 리뷰

✅ 람다 표현식(Lambda Expression)

- 함수형 인터페이스를 구현한 클래스 를 단순화 한 표현식
- 즉, 함수형 인터페이스를 구현한 클래스의 인스턴스



✅ 메서드 레퍼런스(Method Reference)

- **ClassName::static method**
  예) (String s) -> Integer.parseInt(s)
  ↓
  Integer::parseInt
- **ClassName::instance method**
  예) (String s) -> s.toLowerCase( )
  ↓
  String::toLowerCase
- **object::instance method**
  예) (int count) -> obj.getTotal(count)
  ↓
  obj::getTotal
- **ClassName::new**
  예) ( ) -> new CryptoCurrency( )
  ↓
  Car::new

익명 구현 클래스 -> 함수형 인터페이스 -> 람다 표현식 -> 메서드 레퍼런스

- 함수형 인터페이스의 추상 메서드를 설명해놓은 시그니처
- Java 8부터 java.util.function 패키지에서 다양한 함수형 인터페이스를 지원

함수형 인터페이스	함수 디스크립터(Function Descriptor)
Predicate<T>	T -> boolean
Consumer<T>	T -> void
Function<T, R>	T -> R
Supplier<T>	( ) -> T
BiPredicate<L, R>	(L, R) -> boolean
BiConsumer<T, U>	(T, U) -> void
BiFunction<T, U, R>	(T, U) -> R

- 코드 리뷰

✅ Java Stream API

- Stream과 리액티브 프로그래밍은 유사하다

```
Stream
        .of(1, 2, 3, 4)
        .filter(n -> n % 2 != 0)
        .map(n -> n * 2)
        .forEach(n -> System.out.println(n));
```

데이터 소스(Data source)

중간 연산(intermediate operation)

최종 연산(terminal operation)

Stream의 중간 연산과 최종 연산

- 중간 연산(intermediate operation)
  - ✓ Stream 파이프 라인 형성 가능
  - ✓ filter, map, limit, distinct 등

- 최종 연산(terminal operation)
  - ✓ 결과 도출
  - ✓ 최종 연산이 호출 될 때 Stream이 실행된다.
  - ✓ forEach, count, collect 등

Collection과 Stream의 차이점

	Collection	Stream
기본 컨셉	특정 자료구조로 데이터를 저장하는 것이 주 목적이다.	데이터 가공 처리가 주 목적이다.
데이터 수정 여부	데이터 추가/삭제 가능	- 데이터 추가/삭제 불가 - 오로지 데이터 소스를 읽어서 소비하기만 한다.
Iteration 형태	for문 같은 걸로 외부 반복	operation 메서드 내부에서 보이지 않게 반복
탐색 횟수	여러 번 탐색 가능	한번만 탐색 가능
데이터 처리 방식	Eager	Lazy 그리고 Short-circuit

- 코드 리뷰

Stream을 언제 사용하는 것이 좋을까?

- 대용량 데이터의 복잡한 가공처리
- 대용량 데이터가 아니라도 컬렉션 데이터로 복잡한 가공이 필요할 때
- 멀티쓰레딩이 아닌 진짜 병렬 처리

Stream API

- filtering / slicing / mapping / find / match / collect
- reducing/ math / grouping / partitioning
- parallel

너무 많음. 필요할 때 찾아보자