1. 스트림

1. 스트림이란

1. 스트림은 컬렉션이나 배열의 요소를 반복하여 처리하기 위한 방식 중 하나이다.
2. 컬렉션에 .stream() 메소드를 가지고 있어서 컬렉션을 스트림형식으로 변경해서 사용할 수 있다.
3. 최종처리가 끝난 Stream은 자동으로 닫히기 때문에 다시 사용할 수 없다.
4. 스트림 리턴하는 메소드들은 중간처리 메소드이고 나머지 메소드들은 최종처리 메소드이다.

2. 스트림 선언

1. 컬렉션<타입> list => Strema<타입> typeStream = list.stream();

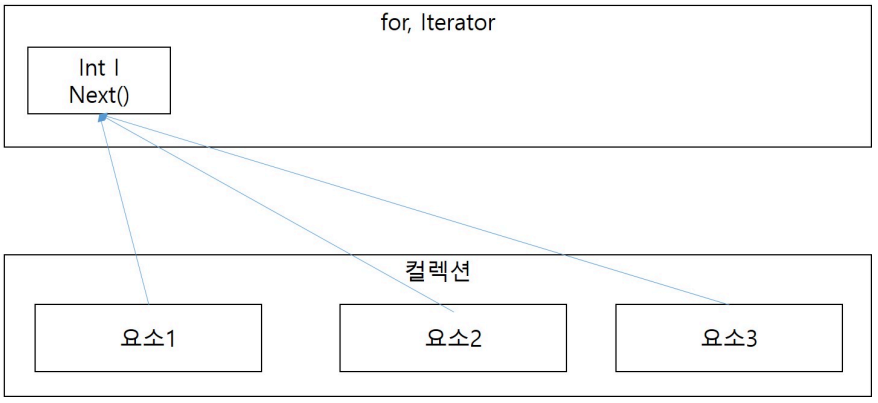
3. 스트림의 forEach

1. 람다식으로 구현해한다.
2. typeStream.forEach(컬렉션의 요소 한개를 담을 변수명 -> {처리할 내용})
3. 내부반복자여서 처리속도가 Iterator보다 빠르다.
4. 람다식으로 다양한 요소를 처리할 수 있다.
5. 중간처리, 최종처리를 수행하도록 파이프 라인(체이닝기법)을 형성할 수 있다.

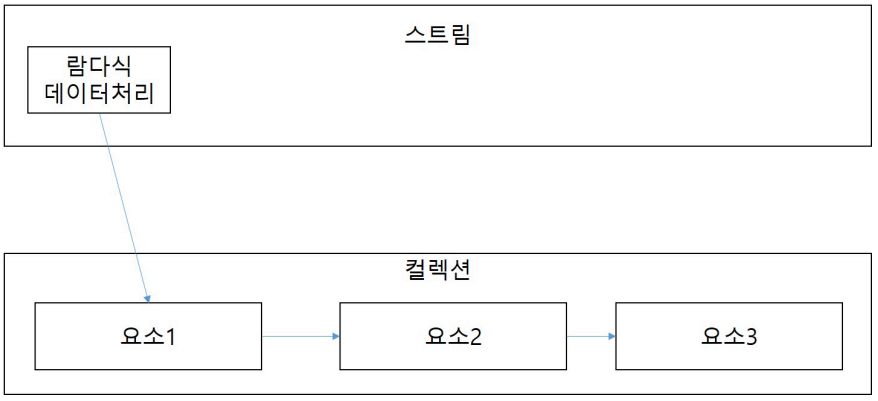
4. 내부반복자

1. for나 Iterator는 컬렉션의 요소를 외부로 꺼내서 사용하는 외부반복자.
2. 스트림은 요소를 컬렉션 내부로 넣어서 반복 처리하는 내부 반복자.

3. 외부반복자 : 컬렉션의 요소를 외부의 변수에 꺼내와서 처리



4. 내부반복자 : 컬렉션 내부에 요소를 주입하여 반복 처리



5. 스트림을 연결하여 파이프라인 만들기

1. 스트림 하나이상 연결할 수 있다. 원본 스트림의 필터링된 중간 스트림이 연결될 수 있고 그 후에 매핑 스트림으로 타입을 변경한 스트림이 연결될 수도 있다. 마지막에 최종처리를 해주는 스트림이 연결돼서 파이프라인을 구성할 수 있다.
2. 현대차 스트림 -> 아반떼 모델만 남은 스트림 -> 가격 -> 총 가격출력
   Stream -> 멤버변수인 model=아반떼 -> Stream
3. 체이닝기법

- int 총 가격 = list.stream() .mapToInt(hCar -> hCar.getPrice()) .sum();
- 메소드의 호출을 연결해서 여러개의 메소드를 호출하는 방식
- 파이프라인을 구성할 때 체이닝기법을 사용하면 더 편리하게 파이프라인을 구성할 수 있다.
- 체이닝기법을 사용해서 파이프라인을 구성할 때 주의할 점은 최종 처리가 없으면 중간 처리까지의 체이닝기법이 동작하지 않는다.

6. 스트림에서 자주 쓰는 메소드

1. Stream stream.(void)**forEach**(type의 값을 주입할 변수명(원하는대로 정한다.) -> 실행코드) : Stream에 담겨있는 의 요소를 순회하면서(하나씩 접근) 실행코드를 요소의 개수만큼 반복실행. 리턴타입이 void라 리턴할 수 없다.
2. Stream stream.(Stream)**map**(type의 값을 주입할 변수명(원하는대로 정한다.) -> 새로운 스트림을 만들어줄 실행코드) : Stream에 담겨있는 의 요소를 순회하면서(하나씩 접근) 실행코드에서 나온 결과값으로 새로운 스트림을 생성해서 리턴
3. Stream stream.(Stream)**filter**(type의 값을 주입할 변수명(원하는대로 정한다.) -> 조건코드) : 변수명 담긴 요소를 조건코드로 검사한다. true가 나오는 요소들만 다시 스트림으로 묶어서 리턴.
4. Stream stream.(Stream)**reduce**((결과값의 타입 변수명, type의 변수명) -> 어떻게 결과 값을 만들지) : 스트림을 하나의 결과값으로 만들어주는 메소드

7. 스트림 얻는 방법

1. 컬렉션에서 스트림 얻기
    - List, Set => stream(), parallelStream() => Stream
2. 배열
    - Arrays.stream(T[]), Stream.of(T[]) => Stream
    - Arrays.stream(int[]), IntStream.of(int[]) => IntStream
    - Arrays.stream(long[]), LongStream.of(long[]) => LongStream
    - Arrays.stream(double[]), DoubleStream.of(double[]) => DoubleStream
3. Files 클래스에서 얻기
    - Files.list(Path) => Stream
    - Files.lines(Path, Charset) => Stream : 텍스트파일의 데이터를 행으로 나눠서 행들의 스트림으로 생성
4. Random 클래스에서 얻기
    - Random.doubles(...) => DoubleStream
    - Random.ints(...) => IntStream
    - Random.longs(...) => LongStream

8. 그외의 메소드

1. dinstinct() : 중복 제거. 객체는 equals가 true로 나오는 중복된 객체를 제거한다. 일반 값들은 같은 값이면 제거
2. mapToInt, Long, Double : IntStream, LongStream, DoubleStream으로 변환해주는 메소드. Wrapper 클래스 형태의 스트림에서 기본자료형 스트림으로 변환.

```
Stream<Integer> => IntStream
Stream<Long> => LongStream
Stream<Double> => DoubleStream
```

3. 형변환

```
IntStream.asLongStream() -> LongStream
IntStream, LongStream.asDoubleStream() -> DoubleStream
IntStream.boxed()              => Stream<Integer>
LongStream.boxed() => Stream<Long>
DoubleStream.boxed()           => Stream<Double>
```

6. Stream flatMap(Function<T, R>, DoubleFunction, IntFunction, LongFunction) : 스트림의 요소를 하나 이상의 새로운 스트림으로 만들어서 모든 스트림을 하나의 스트림으로 연결하는 기능. 스트림의 요소들이 다른 스트림을 가지고 있을 때 하나의 스트림으로 통합해서 사용할 수 있게 해준다.
7. sorted() : 요소들을 오름차순이나 내림차순으로 정렬한 새로운 스트림을 리턴.
8. peek() : 요소에 하나씩 접근해서 처리를 해주는 메소드(forEach와 동일한 기능). 원본 스트림 리턴(새로운 스트림을 만드는 메소드는 map을 사용해야 한다.). forEach가 최종 처리 메소드면 peek 중간 처리 메소드.
9. boolean allMatch(Predicate(true or false) 구현체(람다식)) : 모든 요소가 만족하는지 검사.
10. boolean anyMatch(Predicate(true or false) 구현체(람다식)) : 최소한 하나의 요소라도 만족하는지 검사.
11. noneMatch(Predicate(true or false) 구현체(람다식)) : 모든 요소가 만족하지 않는지 검사.
12. 집계 메소드
    - 모든 스트림에서 사용가능
        - count() : 요소의 개수를 리턴(long)
        - findFirst() : 첫 번째 요소 리턴(Optional객체)
        - max() : 최대 요소 리턴(Optional객체)
        - min() : 최소 요소 리턴(Optional객체)
    - IntStream, LongStream, DoubleStream에서만 사용가능
        - sum() : 합계를 리턴(int, long, double)
        - average() : 평균을 리턴(OptionalDouble)
13. 요소 수집
    - R collect(Collector<T, A, R>) : 인터페이스 Stream 리턴.
        - toList(), toSet() : List, Set로 변환

        - toMap(Fuction key, Function value) : Map<key, value>으로 변환
        - groupingBy(Function<T, K>) : T를 K(key)로 매핑하고 K를 키로 가지는 List를 V(value) 갖는 맵을 생성. 요소를 정의한 키로 그룹핑해준다. Function 인터페이스에서는 Key 지정할 메소드를 구현.
            - mapping(Function, Collector) : 매핑해주는 메소드
            - averagingDouble(ToDoubleFunction) : 평균값을 V로 지정해주는 메소드
            - counting() : 요소 개수를 V로 지정
            - maxBy(Comparator) : 최대 값을 V로 지정
            - minBy(Comparator) : 최소 값을 V로 지정
            - reducing(BinaryOperator) : 커스텀 집계를 V로 지정

9. 스트림의 병렬처리

1. 병렬처리는 멀티 스레드와 마찬가지로 코어 별로 작업을 하나씩 처리해서 동시에 여러개의 작업을 처리해주는 것을 말한다.
2. 동시성 : 1코어가 여러개의 작업을 가지고 한 번에 한 작업을 처리해주는 것을 의미.
3. 병렬성 : 멀티코어가 각각 작업 하나 씩을 가지고 동시에 여러개의 작업을 처리하는 것을 의미.
    - 데이터 병렬성 : 전체 데이터를 분할해서 서브 데이터셋을 만들고 멀티코어가 각각의 서브 데이터셋을 병렬처리.
    - 작업 병렬성 : 멀티코어가 여러개의 작업을 각각 나눠가져서 병렬처리.
4. 포크조인 프레임워크 : 병렬 스트림을 처리하기 위한 프레임워크. 포크단계에서 스트림을 서브 스트림 여러개로 분할을 하고 멀티코어가 각각의 서브 스트림을 처리한 후 처리된 서브 스트림을 최종 병합해서 결과를 만들어준다. ExecutorService를 구현한 ForkJoinPool 클래스를 이용해서 분할된 처리 작업을 스레드로 관리.
5. 포크조인 프레임워크를 사용하려면 병렬 스트림을 생성해서 사용해야한다.
6. 병렬 스트림 얻기
    - parallelStream() : 컬렉션을 병렬 스트림으로 변환
    - Stream, IntStream, LongStream, DoubleStream의 parallel() : 병렬 스트림 생성
7. 병렬 처리 성능
    - 무조건 빠르다는 것은 아니다. 요소의 수가 적어지면 일반 스트림 처리가 빠르다. 병렬 스트림은 스레드 풀을 생성하는 시간과 스레드를 분배하는 시간이 포함되기 때문에 요소의 개수가 적고 요소를 처리하는 시간이 짧을수록 성능이 나빠진다.

- 코어의 수가 많아지고 요소의 개수가 많아질 수록 병렬 스트림의 성능이 증가한다.

8. 조인 포크 프레임워크
    - 병렬 스트림을 생성한 후 처리할 작업을 지정하면 작업을 스레드 단위로 분할해주는 단계를 포크 단계라고 부른다.
    - 작업처리의 결과를 병합해주는 단계를 조인 단계라고 부른다.