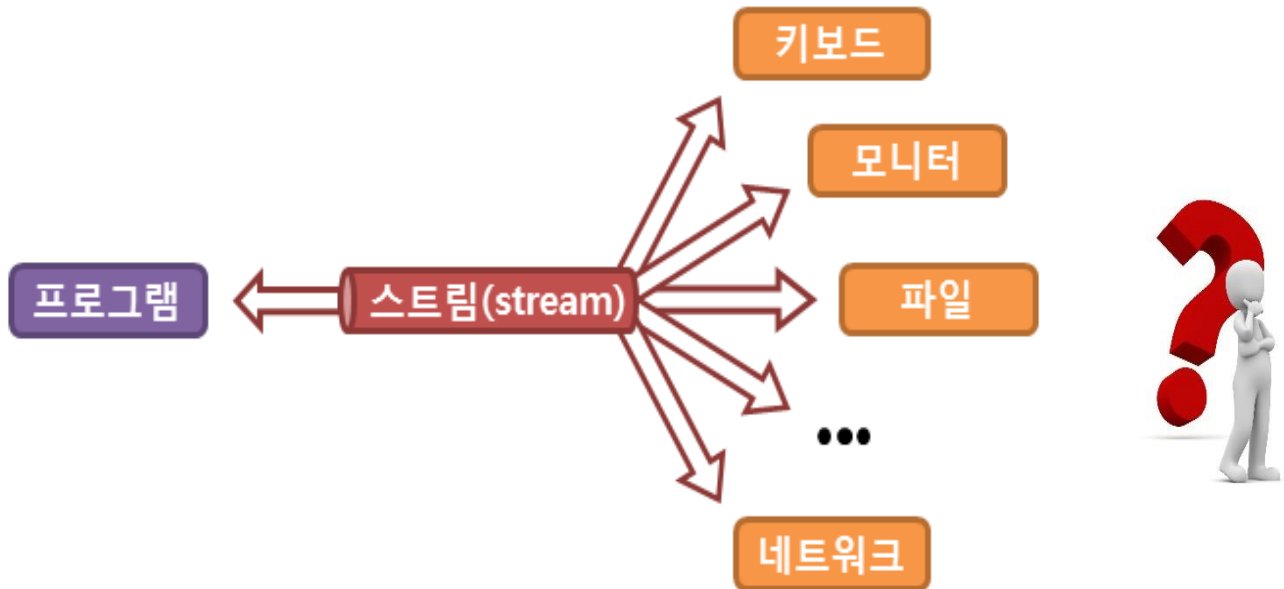


## [ Java 8 Stream API ]

### - 자바의 I/O 스트림


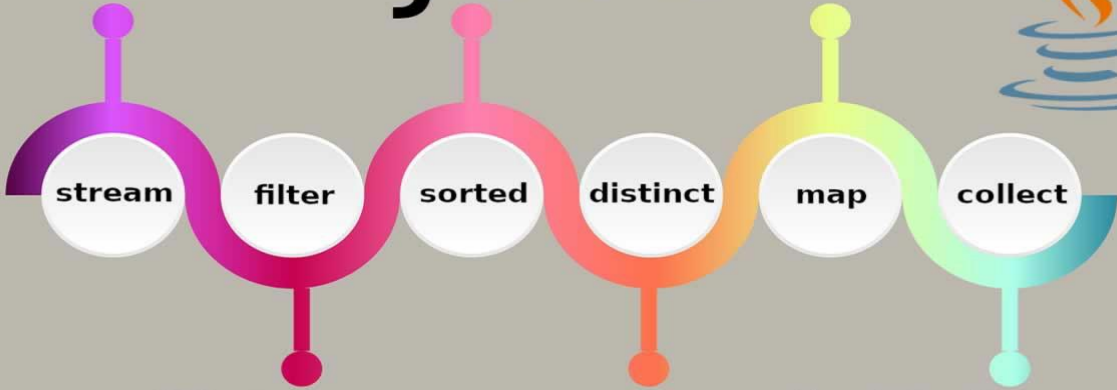


### - 자바8 에서 추가된 스트림 :

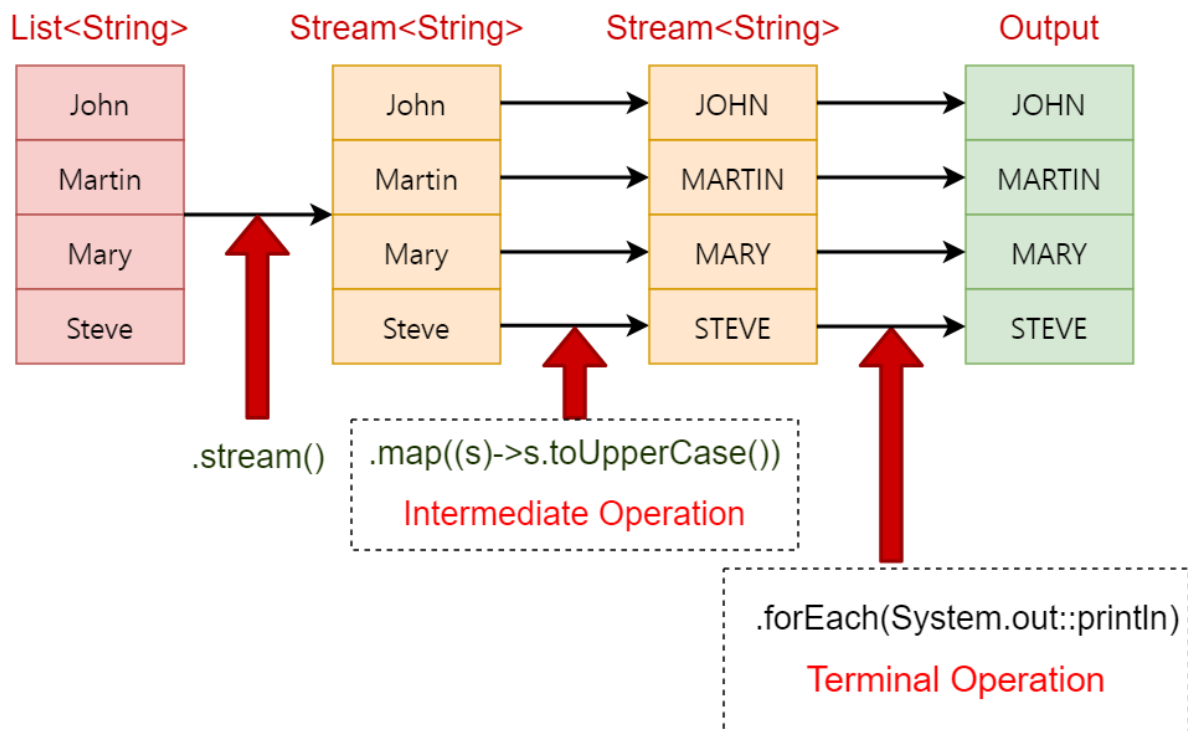
컬렉션, 배열 등에 저장된 요소들을 하나씩 참조하면서 코드를 실행할 수 있는 기능



# Java

# Stream API



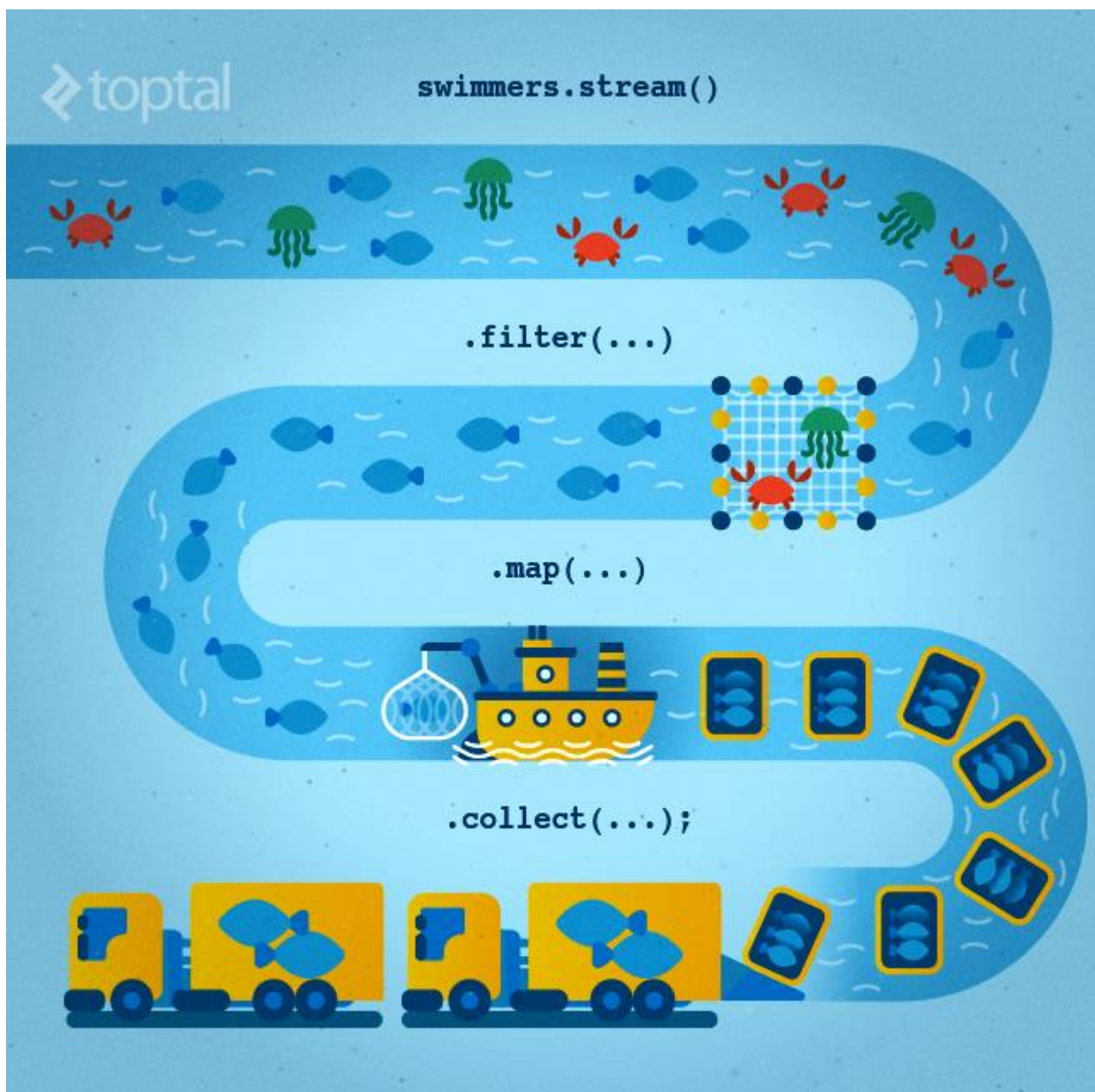
- 자료의 대상과 관계없이 동일한 연산을 수행할 수 있게 하는 기능
- Array, Collection에 동일한 연산이 수행돼서 일관성 있는 처리 기능을 갖게 한다
- 스트림은 1회성이라 한번 사용한 스트림은 재사용 하지 못한다
- 스트림의 연산은 기존의 자료를 변경하지 않고, 새로운 메모리 위에서 동작한다
- 중간연산과 최종 연산이 구분되며, 최종 연산이 수행된 이후 모든 연산이 적용되는 **지연연산**을 지원한다.

Java 8 부터 컬렉션 및 배열의 요소를 반복 처리하기 위해 스트림을 사용한다.  
요소들이 하나씩 흘러가면서 처리된다는 의미이다. (스트림 ---> 데이터 흐름)

```
Stream<String> stream = list.stream();  
stream.forEach( item -> //item 처리 );
```

List 컬렉션의 stream() 메소드로 Stream 객체를 얻고, forEach() 메소드로 요소를 어떻게 처리할지 람다식으로 제공하여 처리한다.

자바 8부터 새로 추가된 java.util.stream 패키지에는 여러 Stream API들이 포함되어 있다.



## - 외부 반복자와 내부 반복자

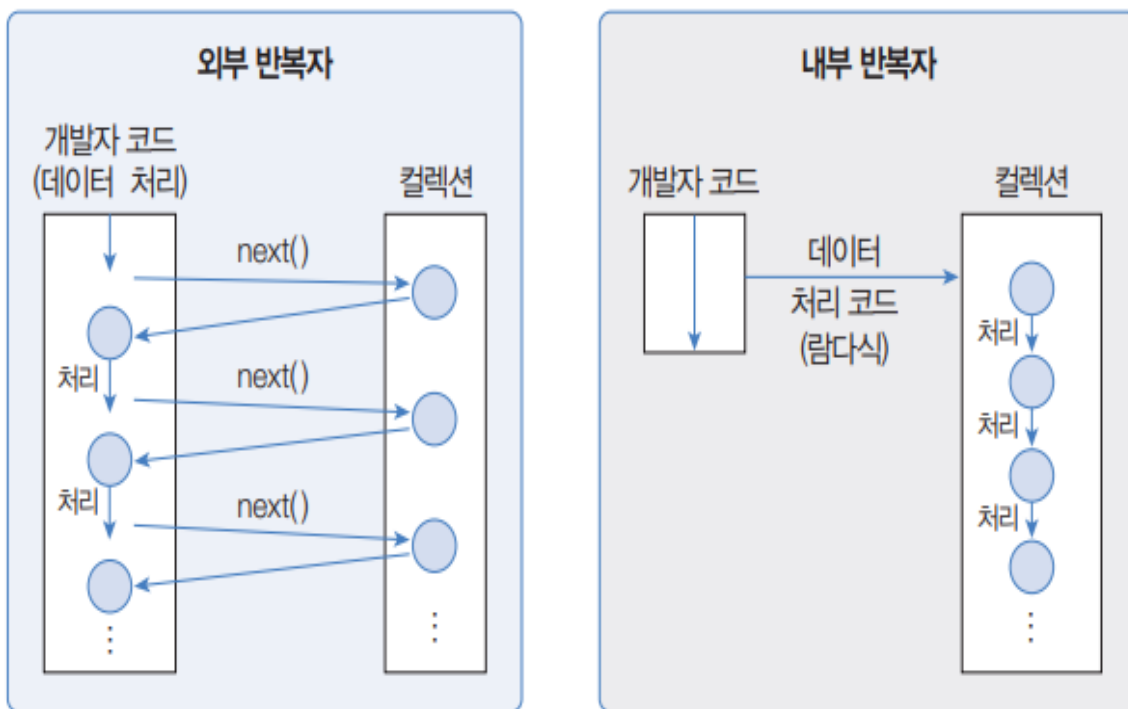
외부반복자는 일반적으로 사용하는 루프처럼 요소를 사용하는 쪽에서 직접 컬렉션 요소를 하나씩 꺼내 와서 반복 처리

내부반복자는 처리할 행동(보통 콜백 함수)을 컬렉션 요소에 넘겨주어 반복 처리

요소 처리 방법을 컬렉션 내부로 주입시켜서 요소를 반복 처리

개발자 코드에서 제공한 데이터 처리 코드(람다식)를 가지고 컬렉션 내부에서 요소를 반복 처리

내부 반복자는 멀티 코어 CPU를 최대한 활용하기 위해 요소들을 분배 시켜 병렬 작업 가능



## - 스트림은 내부 반복자

- 내부 반복자이므로 처리 속도가 빠르고 병렬 처리에 효율적
- 람다식으로 다양한 요소 처리를 정의
- 중간 처리와 최종 처리를 수행하도록 파이프 라인 형성
- 내부 반복자의 이점은 어떻게 요소를 반복시킬 것인가는 컬렉션에 맡겨두고, 개발자는 요소 처리 코드에만 집중
- 요소들의 반복 순서를 변경하거나, 멀티 코어 CPU를 최대한 활용하기 위해 요소들을 분배시켜 병렬 작업을 할 수 있게 도와줌

```
List<String> list = Arrays.asList("A", "B", "C", "D");
```

```
// 스트림 사용✕
```

```
Iterator<String> iter = list.iterator();
```

```
while(iter.hasNext()){
```

```
    String name = iter.next();
```

```
    System.out.println(name);
```

```
}
```

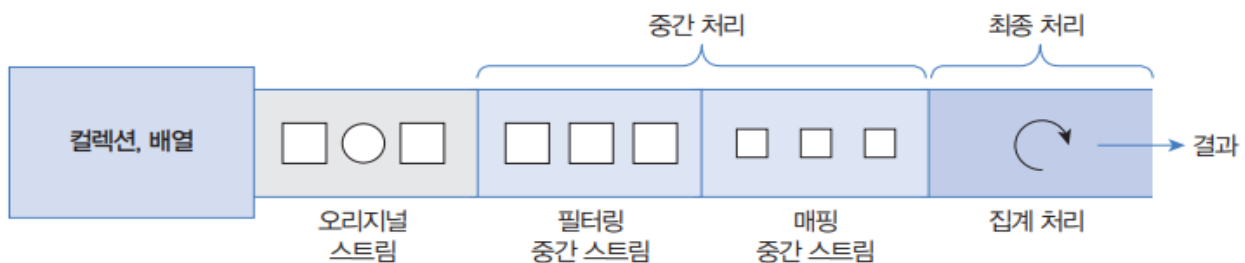
```
// 스트림 사용○
```

```
Stream<String> stream = list.stream();
```

```
list.stream().forEach(name -> System.out.println(name));
```

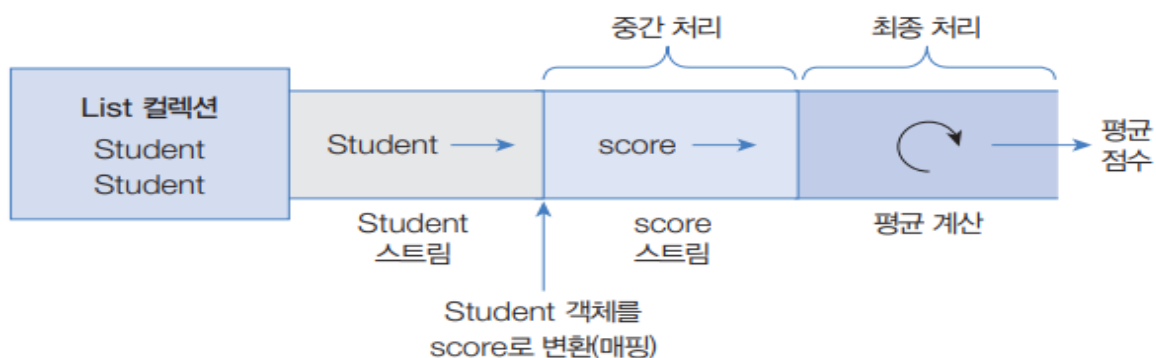
## - 스트림 파이프라인

컬렉션의 오리지널 스트림 뒤에 필터링 중간 스트림이 연결될 수 있고, 그 뒤에 매핑 중간 스트림이 연결될 수 있음



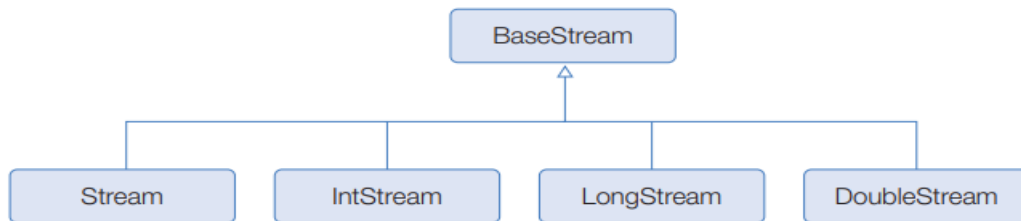
오리지널 스트림과 집계 처리 사이의 중간 스트림들은 최종 처리를 위해 요소를 걸러내거나(필터링), 요소를 변환시키거나(매핑), 정렬하는 작업을 수행

최종 처리는 중간 처리에서 정제된 요소들을 반복하거나, 집계(카운팅, 총합, 평균) 작업을 수행



## - 스트림 인터페이스

java.util.stream 패키지에는 BaseStream 인터페이스를 부모로 한 자식 인터페이스들이 제공되며 BaseStream에는 모든 스트림에서 사용할 수 있는 공통 메소드들이 정의되어 있다.



리턴 타입	메소드(매개변수)	소스
Stream<T>	java.util.Collection.stream() java.util.Collection.parallelStream()	List 컬렉션 Set 컬렉션
Stream<T> IntStream LongStream DoubleStream	Arrays.stream(T[] ), Stream.of(T[] ) Arrays.stream(int[] ), IntStream.of(int[] ) Arrays.stream(long[] ), LongStream.of(long[] ) Arrays.stream(double[] ), DoubleStream.of(double[] )	배열
IntStream	IntStream.range(int, int) IntStream.rangeClosed(int, int)	int 범위
LongStream	LongStream.range(long, long) LongStream.rangeClosed(long, long)	long 범위
Stream<Path>	Files.list(Path)	디렉토리
Stream<String>	Files.lines(Path, Charset)	텍스트 파일
DoubleStream IntStream LongStream	Random.doubles(...) Random.ints() Random.longs()	랜덤 수

### - 컬렉션으로부터 스트림 얻기

java.util.Collection 인터페이스는 **stream()**과 **parallelStream()** 메소드를 가지고 있어 자식 인터페이스인 List와 Set 인터페이스를 구현한 모든 컬렉션에서 객체 스트림을 얻을 수 있음

### - 배열로부터 스트림 얻기

java.util.Arrays 클래스로 다양한 종류의 배열로부터 스트림을 얻을 수 있음

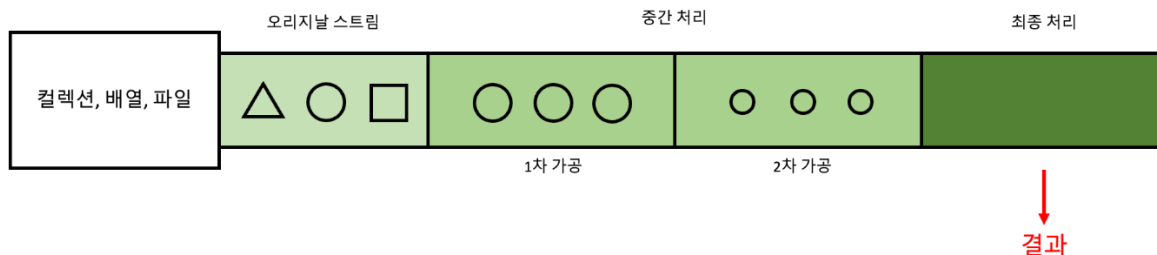
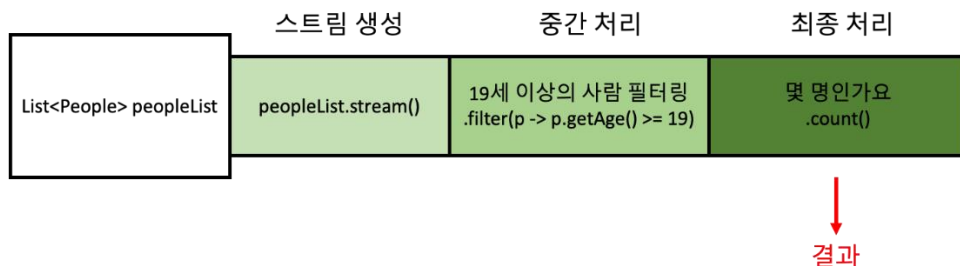
### - 숫자 범위로부터 스트림 얻기

IntStream 또는 LongStream의 정적 메소드인 **range()**와 **rangeClosed()** 메소드로 특정 범위의 정수 스트림을 얻을 수 있음

- 파일로부터 스트림 얻기

java.nio.file.Files의 lines() 메소드로 텍스트 파일의 행 단위 스트림을 얻을 수 있음  
 [ 스트림의 중간 처리 메서드와 최종 처리 메서드 ]

- Stream은 요소에 대해 중간 처리와 최종 처리를 수행할 수 있다.
- 중간 처리(매핑, 필터링, 정렬 등)는 여러 번 사용할 수 있다.
- 최종 처리(반복, 집계처리 등)는 결과 처리이므로 한 번만 사용할 수 있다.



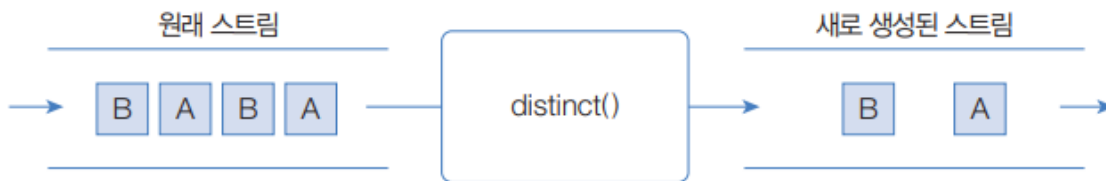
종류		리턴타입	메소드
중간 처리	필터링	Stream	distinct()
	매핑		filter()
			flatMap()
			map()
			boxed()
			정렬
	루핑		peek()
최종 처리	매칭	boolean	allMatch()
			anyMatch()
			noneMatch()
	집계	long	count()
		OptionalXXX	findFirst()
		OptionalXXX	max()
		OptionalXXX	min()
		OptionalDouble	average()
		OptionalXXX	reduce()
		int, long, double	sum()
	루핑	void	forEach()
수집	R	collect()	

## - 필터링

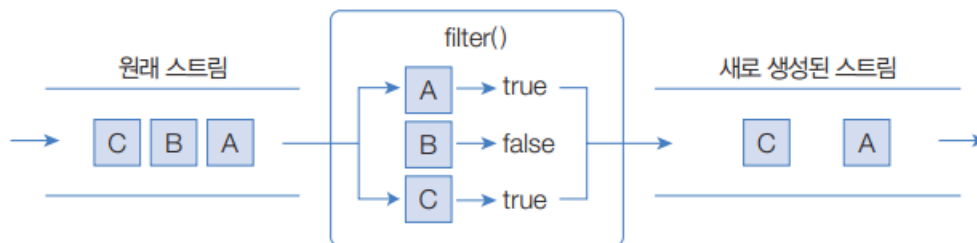
필터링은 요소를 걸러내는 중간 처리 기능

리턴 타입	메소드(매개변수)	설명
Stream IntStream LongStream DoubleStream	distinct()	- 중복 제거
	filter(Predicate<T>)	- 조건 필터링
	filter(IntPredicate)	- 매개 타입은 요소 타입에 따른 함수형 인터페이스이므로 람다식으로 작성 가능
	filter(LongPredicate)	
	filter(DoublePredicate)	

distinct() 메소드: 요소의 중복을 제거



filter() 메소드: 매개값으로 주어진 Predicate가 true를 리턴하는 요소만 필터링



## - 매핑

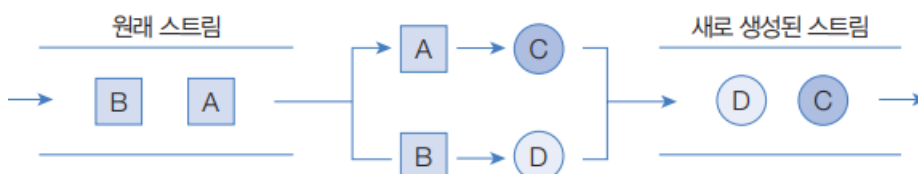
스트림의 요소를 다른 요소로 변환하는 중간 처리 기능

매핑 메소드: `mapXxx()`, `asDoubleStream()`, `asLongStream()`, `boxed()`, `flatMapXxx()` 등

- 요소를 다른 요소로 변환

`mapXxx()` 메소드: 요소를 다른 요소로

변환한 새로운 스트림을 리턴

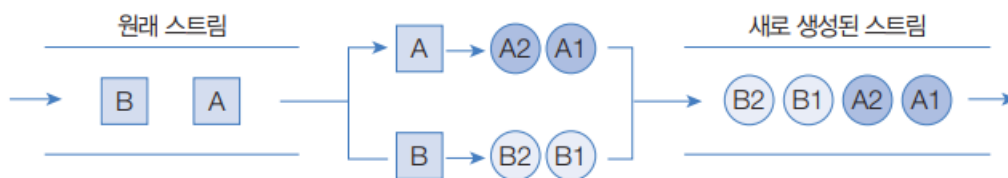




리턴 타입	메소드(매개변수)	요소 → 변환 요소
Stream<R>	map(Function<T, R>)	T → R
IntStream LongStream DoubleStream	mapToInt(ToIntFunction<T>)	T → int
	mapToLong(ToLongFunction<T>)	T → long
	mapToDouble(ToDoubleFunction<T>)	T → double
Stream<U>	mapToObj(IntFunction<U>)	int → U
	mapToObj(LongFunction<U>)	long → U
	mapToObj(DoubleFunction<U>)	double → U
DoubleStream DoubleStream IntStream LongStream	mapToDouble(IntToDoubleFunction)	int → double
	mapToDouble(LongToDoubleFunction)	long → double
	mapToInt(DoubleToIntFunction)	double → int
	mapToLong(DoubleToLongFunction)	double → long

- 요소를 복수 개의 요소로 변환

flatMapXxx() 메소드: 하나의 요소를 복수 개의 요소들로 변환한 새로운 스트림을 리턴



## - 정렬

요소를 오름차순 또는 내림차순으로 정렬하는 중간 처리 기능

리턴 타입	메소드(매개변수)	설명
Stream<T>	sorted()	Comparable 요소를 정렬한 새로운 스트림 생성
Stream<T>	sorted(Comparator<T>)	요소를 Comparator에 따라 정렬한 새 스트림 생성
DoubleStream	sorted()	double 요소를 올림차순으로 정렬
IntStream	sorted()	int 요소를 올림차순으로 정렬
LongStream	sorted()	long 요소를 올림차순으로 정렬

## - Comparable 구현 객체의 정렬

스트림의 요소가 객체일 경우 객체가 Comparable을 구현하고 있어야만 sorted() 메소드를 사용하여 정렬 가능. 그렇지 않다면 ClassCastException 발생

```
public Xxx implements Comparable {
    ...
}
```

```
List<Xxx> list = new ArrayList<>();
Stream<Xxx> stream = list.stream();
Stream<Xxx> orderedStream = stream.sorted();
```

## - Comparator를 이용한 정렬

요소 객체가 Comparable을 구현하고 있지 않다면, 비교자를 제공하면 요소를 정렬시킬 수 있음

```
sorted((o1, o2) -> { ... })
```

괄호 안에는 o1이 o2보다 작으면 음수, 같으면 0, 크면 양수를 리턴하도록 작성

o1과 o2가 정수일 경우에는 Integer.compare(o1, o2)를, 실수일 경우에는 Double.compare(o1, o2)를 호출해서 리턴값을 리턴 가능

## - 루핑

스트림에서 요소를 하나씩 반복해서 가져와 처리하는 것

리턴 타입	메소드(매개변수)	설명
void	forEach(Consumer<? super T> action)	T 반복
	forEach(IntConsumer action)	int 반복
	forEach(DoubleConsumer action)	double 반복

## - 매칭

요소들이 특정 조건에 만족하는지 여부를 조사하는 최종 처리 기능

allMatch(), anyMatch(), noneMatch() 메소드는 매개값으로 주어진 Predicate가 리턴하는 값에 따라 true 또는 false를 리턴

리턴 타입	메소드(매개변수)	조사 내용
boolean	allMatch(Predicate<T> predicate) allMatch(IntPredicate predicate) allMatch(LongPredicate predicate) allMatch(DoublePredicate predicate)	모든 요소가 만족하는지 여부
boolean	anyMatch(Predicate<T> predicate) anyMatch(IntPredicate predicate) anyMatch(LongPredicate predicate) anyMatch(DoublePredicate predicate)	최소한 하나의 요소가 만족하는지 여부
boolean	noneMatch(Predicate<T> predicate) noneMatch(IntPredicate predicate) noneMatch(LongPredicate predicate) noneMatch(DoublePredicate predicate)	모든 요소가 만족하지 않는지 여부

## - 집계

최종 처리 기능으로 요소들을 처리해서 카운팅, 합계, 평균값, 최대값, 최소값 등 하나의 값으로 산출하는 것으로 스트림이 제공하는 기본 집계 메서드를 사용

스트림은 카운팅, 최대, 최소, 평균, 합계 등을 처리하는 다음과 같은 최종 처리 메소드를 제공

리턴 타입	메소드(매개변수)	설명
long	count()	요소 개수
OptionalXXX	findFirst()	첫 번째 요소
Optional<T> OptionalXXX	max(Comparator<T>) max()	최대 요소
Optional<T> OptionalXXX	min(Comparator<T>) min()	최소 요소
OptionalDouble	average()	요소 평균
int, long, double	sum()	요소 총합

Optional<T> 클래스는 Integer나 Double 클래스처럼 'T'타입의 객체를 포장해 주는 래퍼 클래스(Wrapper class) 로서 모든 타입의 참조 변수를 저장할 수 있으며 예상치 못한 NullPointerException 예외를 제공되는 메소드로 간단히 제어할 수 있다. 즉, 복잡한 조건문 없이도 널(null) 값으로 인해 발생하는 예외를 처리할 수 있게 된다.

#### - 필터링한 요소 수집

Stream의 collect(Collector<T,A,R> collector) 메소드는 필터링 또는 매핑된 요소들을 새로운 컬렉션에 수집하고, 이 컬렉션을 리턴

매개값인 Collector는 어떤 요소를 어떤 컬렉션에 수집할 것인지를 결정

타입 파라미터의 T는 요소, A는 누적기accumulator, 그리고 R은 요소가 저장될 컬렉션

리턴 타입	메소드(매개변수)	인터페이스
R	collect(Collector<T,A,R> collector)	Stream

리턴 타입	메소드	설명
Collector<T, ?, List<T>>	toList()	T를 List에 저장
Collector<T, ?, Set<T>>	toSet()	T를 Set에 저장
Collector<T, ?, Map<K,U>>	toMap( Function<T,K> keyMapper, Function<T,U> valueMapper )	T를 K와 U로 매핑하여 K를 키로, U를 값으로 Map에 저장

#### - 스트림은 일회성

스트림은 한 번만 탐색이 가능하고 탐색된 스트림은 소비

다시 탐색하려면 새로운 스트림을 생성해야 함

```
List<String> list = Arrays.asList("A", "B", "C", "D");
```

```
Stream<String> stream = list.stream();
```

```
stream.forEach(name -> System.out.println(name));
```

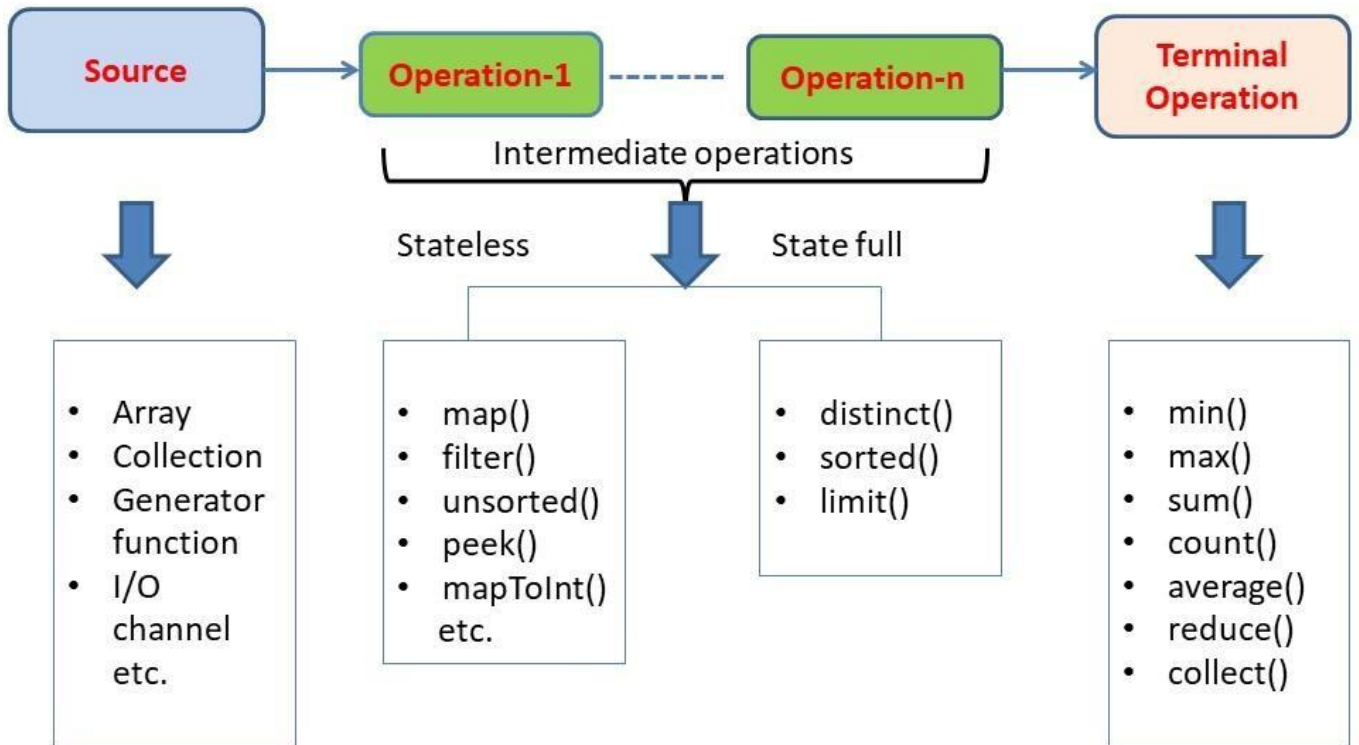
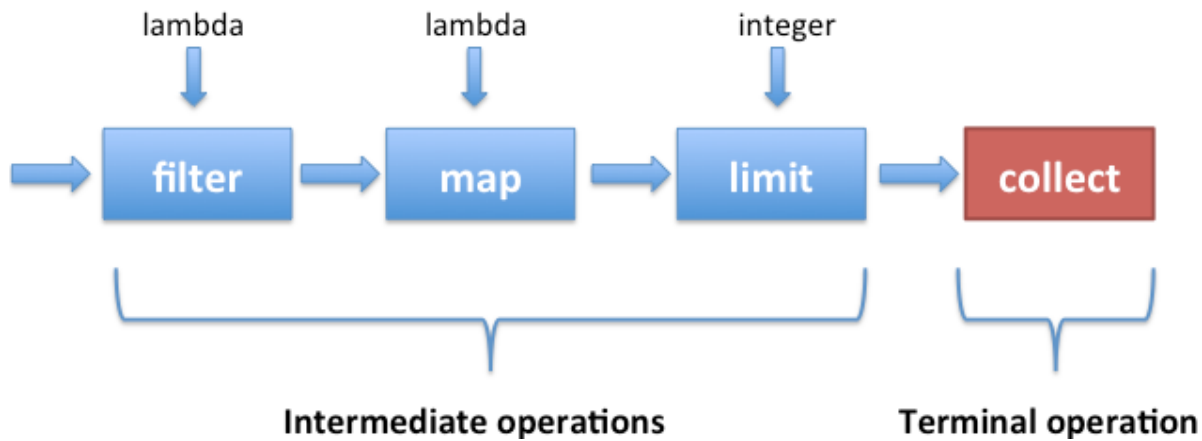
```
stream.forEach(name -> System.out.println(name)); // java.lang.IllegalStateException 발생
```

## [ 정리 ]

스트림은 중간 처리와 최종 처리 단계로 나뉜다.

중간 처리 : 매핑, 필터링, 정렬

최종 처리 : 반복, 카운팅, 평균, 총합 등의 집계 처리



[ 예 1 ]

```
String[] fruits = new String[]{"apple", "oranges",  
                                "banana", "pineapple", "oranges", "apple", "mango"};  
Stream<String> fruitsStream = Arrays.stream(fruits);  
  
List<String> newList1 = fruitsStream.filter( x -> x.endsWith("e")).collect(Collectors.toList());  
List<String> newList2 = Arrays.stream(fruits).map( x ->  
x.toUpperCase()).collect(Collectors.toList());  
  
System.out.println(newList1);  
System.out.println("-----");  
for (String e : newList2)  
    System.out.println(e);
```

[ 예 2 ]

```
List<String> strList = Arrays.asList("3", "1", "4", "2", "5", "5");  
System.out.println(  
    strList.stream()  
        .map(x -> Integer.parseInt(x))  
        .sorted()  
        .distinct()  
        .limit(3)  
        .collect(Collectors.toList()));  
System.out.println("-----");  
  
Stream.of("3", "1", "4", "2", "5", "5")  
    .map(x -> Integer.parseInt(x))  
    .sorted()  
    .distinct()  
    .limit(3)  
    .collect(Collectors.toList())  
    .stream() // 다시 정수형 값을 갖는 스트림으로 변환  
    .filter(x -> x > 1) // 1보다 큰 값만 갖도록 필터링함 {2, 3}  
    .forEach(x -> System.out.println(x));
```

## [ 연산의 순서 ]

스트림은 지연된(lazy) 연산을 한다. 단말 연산이 없으면 연산을 실행하지 않고, 단말연산이 수행되기 전에 중간연산이 실행되지 않는다. 결과가 필요하기 전까지 실행되지 않는다는 뜻이다.

```
Stream.of("3", "1", "4", "2", "5", "5")
    .map(x -> {
        System.out.println("map : " + x);
        return Integer.parseInt(x);
    })
    .filter(x -> {
        System.out.println("filter : " + x);
        return x > 1;
    });
```

이 코드를 실행하면 단말연산이 없기 때문에 아무것도 출력되지 않는다.

```
Stream.of("3", "1", "4", "2", "5", "5")
    .map(x -> {
        System.out.println("map : " + x);
        return Integer.parseInt(x);
    })
    .filter(x -> {
        System.out.println("filter : " + x);
        return x > 1;
    })
    .forEach(x -> {
        System.out.println("forEach : " + x);
    });
```

## [ 스트림의 처리 흐름 ]

스트림이 해당 연산을 처리한 후에 전체 스트림을 넘겨 다른 작업할 것이라 생각할 것이다. map에서 주어진 기능을 모든 엘리먼트들에 대하여 수행한 다음 filter 단계로 넘어가고, filter에서 주어진 기능을 모든 엘리먼트들에 대하여 수행한 다음, forEach 단계로 넘어가 filter에서 걸러진 나머지 값들을 forEach에서 출력할 거라고 생각할 것이다. 그러나 스트림의 요소들이 개별적으로 중간연산들을 통과해서 단말연산에 도달하는 순으로 진행이 된다