8253jang@daum.net
장희정

1. Stream 이란?

자바에서 **컬렉션(List, Set, Map)이나 배열의 데이터를 효율적으로** 처리하기 위해 제공되는 연속적인 데이터 흐름(Stream)이며, 데이터 **변환, 필터링, 정렬 등을 간결(함수형 프로그래밍 방식)**하고 직관적으로 처리할 수 있도록 도와주는 API이다.

2. Stream API가 왜 만들어졌을까?

(1) 기존 방식의 한계

자바에서는 컬렉션(List, Set, Map 등)이나 배열을 다룰 때, 반복문(for, while)을 써서 데이터를 처리해왔지만 이런 방식에는 몇 가지 단점이 있다.

☞ 문제점 1 : 코드가 장황하고 가독성이 떨어짐

```
List<String> names = Arrays.asList("Alice", "Bob", "Charlie", "David");

List<String> filteredNames = new ArrayList<>();
for (String name : names) {
    if (name.startsWith("C")) {
        filteredNames.add(name.toUpperCase());
    }
}
System.out.println(filteredNames);
```

for 문을 사용하니 코드가 길어지고 가독성이 떨어진다.
if 문으로 조건을 확인하고, filteredNames 리스트에 추가하는 과정이 반복돼서 비효율적이다.

☞ 문제점 2 : 병렬 처리(Parallel Processing)가 어렵다.

일반적인 for 문을 쓰면 데이터를 하나씩 순차적으로 처리한다.
CPU의 여러 코어를 활용하는 병렬 처리(Parallel Processing)가 어렵기 때문에 속도가 느릴 수 있다.

(2) Stream API 의 등장

위 문제를 해결하기 위해 Java 8(2014 년)에서 Stream API 가 등장했다.
Stream API 를 사용하면 **가독성이 좋아지고, 병렬 처리가 쉬워지고, 데이터 가공이 간편** 해진다.

결론적으로 Stream API 의 장점은

코드가 간결 해진다. 데이터 가공(Filtering, Mapping, Sorting)이 쉬워진다. 병렬 처리(Parallel Processing)가 가능하다.

Stream API 사용 예제

(1) 기존 방식 vs. Stream API 방식 비교

예) 이름 리스트에서 'C'로 시작하는 이름을 대문자로 변환해 출력하기

기존 방식 (for문 사용)

```
List<String> names = Arrays.asList("Alice", "Bob", "Charlie", "David");
List<String> filteredNames = new ArrayList<>();

for (String name : names) {
    if (name.startsWith("C")) {
        filteredNames.add(name.toUpperCase());
    }
}
System.out.println(filteredNames); //
```

Stream API 방식

```
List<String> names = Arrays.asList("Alice", "Bob", "Charlie", "David");

List<String> result = names.stream()    // Stream 생성
        .filter(name -> name.startsWith("C"))  // 'C'로 시작하는 이름 필터링
        .map(String::toUpperCase)  // 대문자로 변환
        .toList();  // 리스트로 변환

System.out.println(result); //
```

☞ **차이점 (더 짧고 가독성이 좋다)**

- filter() : 원하는 데이터만 걸러냄.
- map() : 데이터를 변환(소문자 → 대문자).
. - toList() : 최종 결과를 리스트로 변환.

(2) 병렬 처리 (Parallel Processing)

대량의 데이터를 처리할 때 parallelStream()을 사용하면 **멀티코어 CPU 를 활용하여 속도를 높일 수 있다.**

기존 방식 (for문)

```
List<String> names = Arrays.asList("Alice", "Bob", "Charlie", "David");
for (String name : names) {
    if (name.startsWith("C")) {
        System.out.println(Thread.currentThread().getName() + " - " + name);
    }
}

단일 스레드에서만 실행된다.
```

Stream API 병렬 처리

```
List<String> names = Arrays.asList("Alice", "Bob", "Charlie", "David");

names.parallelStream()
     .filter(name -> name.startsWith("C"))
     .forEach(name ->
         System.out.println(Thread.currentThread().getName() + " - " + name)
     );

여러 개의 스레드가 병렬로 실행된다.
```

4. Stream API 언제 사용하면 좋을까?

1) 대량의 데이터 처리 (Filtering, Mapping, Sorting)

filter(), map(), sorted() 등을 사용하여 데이터를 쉽게 변환하고 정렬할 수 있다.

예) 학생들의 점수 리스트에서 80점 이상인 학생만 추출하고 정렬하기.

```
List<Integer> scores = Arrays.asList(90, 70, 85, 60, 95, 80);
List<Integer> topScores = scores.stream()
    .filter(score -> score >= 80)
    .sorted()
    .toList();
```

```
System.out.println(topScores); // [80, 85, 90, 95]
```

2) 데이터 그룹핑 (Grouping)

Collectors.groupingBy()를 활용하여 데이터를 쉽게 그룹화할 수 있다.

예) 학생들의 점수를 학점별(A, B, C)로 그룹핑하기.

```
Map<String, List<Integer>> groupedScores =
    scores.stream()
    .collect(Collectors.groupingBy(score -> {
        if (score >= 90) return "A";
        else if (score >= 80) return "B";
        else return "C";
    }));

System.out.println(groupedScores);
// {A=[90, 95], B=[85, 80], C=[70, 60]}
```

3) 병렬 처리 (Parallel Processing)

parallelStream()을 사용하여 CPU 성능을 최대로 활용 가능하다.

데이터가 많을 때 병렬 처리를 하면 속도가 훨씬 빨라진다.

```
long count = names.parallelStream().filter(name -> name.startsWith("C")).count();
System.out.println("C로 시작하는 이름 개수: " + count);
```

기존 방식 (for-loop)	Stream API 방식
코드가 길고 복잡함	코드가 간결하고 가독성이 좋음
데이터 처리 속도가 느림	병렬 처리로 성능 향상 가능
여러 개의 변수를 사용해야 함	함수형 스타일로 직관적인 코드 작성 가능

Stream API

- Java 8부터 컬렉션 및 배열의 요소를 반복 처리하기 위해 스트림 사용한다.
- 요소들이 하나씩 흘러가면서 처리된다는 의미이다.
- List 컬렉션의 stream() 메소드로 Stream 객체를 얻고, forEach() 메소드로 요소를 어떻게 처리할지를 람다식으로 제공한다.

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

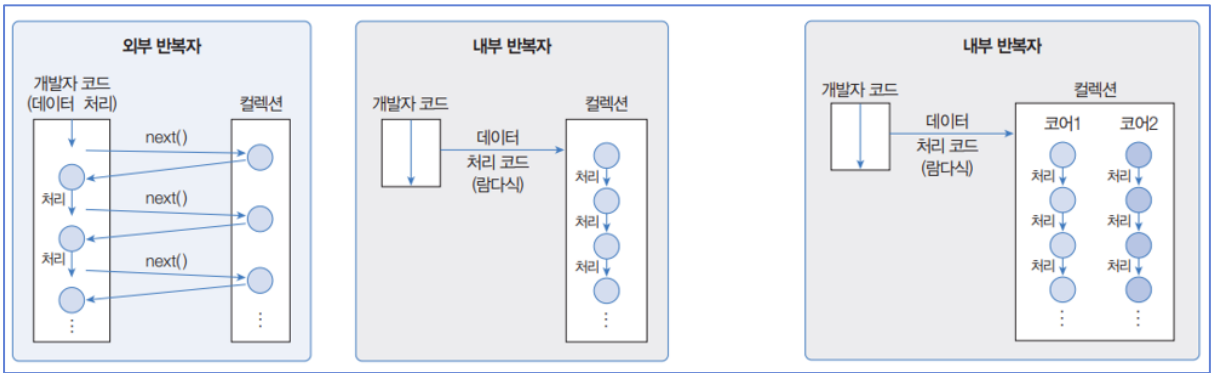
- **스트림과 Iterator 차이점**

1) 내부 반복자이므로 처리 속도가 빠르고 병렬 처리에 효율적
2) 람다식으로 다양한 요소 처리를 정의
3) 중간 처리와 최종 처리를 수행하도록 파이프 라인을 형성

```
7  public class Ex01StreamExample {
8      public static void main(String[] args) {
9          Set<String> set = new HashSet<>();
10
11         set.add("하승현");
12         set.add("장희정");
13         set.add("이가현");
14
15         //Stream api 이용
16         Stream<String> stream = set.stream();
17         stream.forEach(s-> System.out.println(s));
18     }
19 }
```

내부 반복자

- 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 분배시켜 병렬 작업 가능

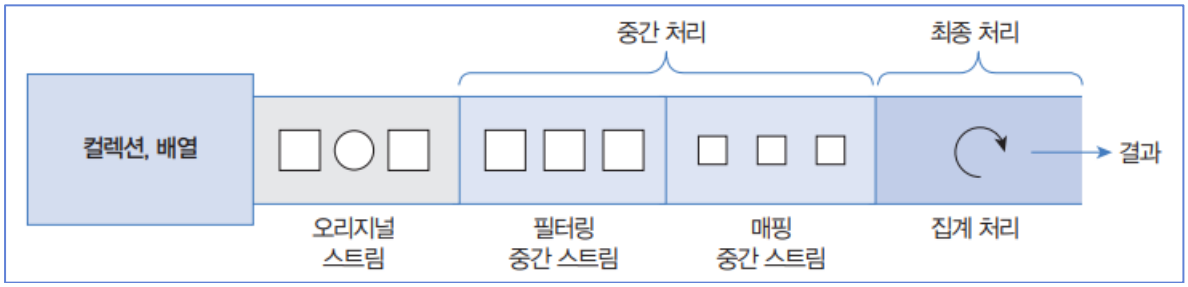


외부반복자 일 경우 컬렉션의 요소를 외부로 가져오는 코드와 처리하는 코드를 모두 개발자 코드가 가지고 있어야 한다. 반면, 내부 반복자 일 경우는 개발자 코드에서 제공한 데이터 처리코드(람다식)를 가지고 컬렉션 내부에서 요소를 반복처리한다.
내부 반복자는 병렬작업이 가능하고 효율적으로 요소를 반복 시킬수 있다는 장점이 있다.

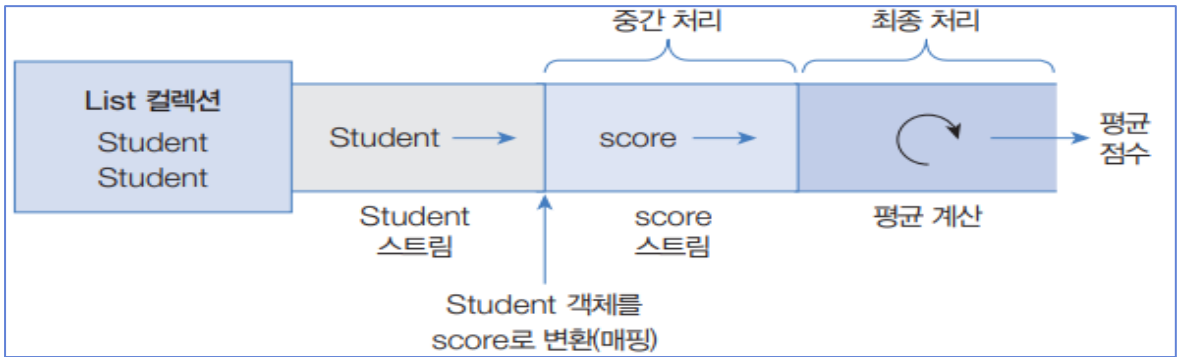
```java
public class Ex02ParallelStreamExample {
    public static void main(String[] args) {
        //List 컬렉션 생성
        List<String> list = new ArrayList< >();
        list.add("삼순이");
        list.add("삼식이");
        list.add("철이");
        list.add("순이");
        list.add("순돌이");
        list.add("순돌이");

        for(String name :list){
            System.out.println(name + ": " + Thread.currentThread().getName());
        }
        System.out.println("----------------------------");
        //병렬 처리
        Stream<String> parallelStream = list.parallelStream();
        parallelStream.forEach( name -> {
            System.out.println(name + ": " + Thread.currentThread().getName());
        } );
    }
}
```

중간처리와 최종처리

- 컬렉션의 오리지널 스트림 뒤에 필터링 중간 스트림이 연결될 수 있고, 그 뒤에 매핑 중간 스트림이 연결될 수 있다.



오리지널 스트림과 집계처리 사이의 중간 스트림들은 최종 처리를 위해 요소를 걸러내거나(필터링) 요소를 변환시키거나(매핑), 정렬하는 작업을 수행한다. 최종 처리는 중간 처리에서 정제된 요소들을 반복하거나 집계(카운팅, 총합,평균) 작업을 수행한다.



스트림 파이프라인으로 구성할 때 주의할점은 파이프라인 맨 끝에는 반드시 최종처리 부분이 있어야 한다. 최종처리가 없다면 오리지널 및 중간처리 스트림은 동작하지 않는다.

Stream 사용법

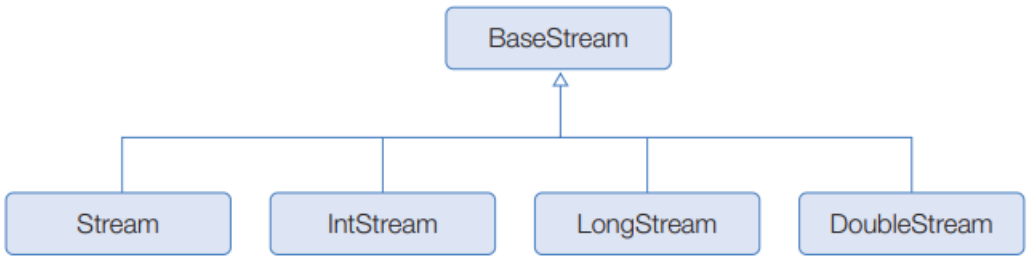
Stream만들기 ----> 중간처리(n번이상가능) ----> 최종처리(1번가능)

Stream은 한번 사용하고 나면 다시 사용 할 수 없다.
Stream을 여러 번 만들어도 Collection, 배열은 변하지 않고 그대로 사용 가능하다.

```java
public class Ex03StreamPipeLineExample {
    new *
    public static void main(String[] args) {
        List<Student> list = Arrays.asList(
                new Student( name: "장희정",  score: 95),
                new Student( name: "이찬범",  score: 80),
                new Student( name: "이가현",  score: 100)
                );

        Stream<Student> studentStream = list.stream();
        //중간 처리(학생 객체를 점수로 매핑) - mapToInt는 IntStream를 반환한다.
        IntStream scoreStream = studentStream.mapToInt(student -> student.getScore());
        //최종 처리(평균 점수)
        double avg = scoreStream.average().getAsDouble();

        System.out.println("--- 메소드 체이닝 패턴이용----------------------------------");
         avg = list.stream() Stream<Student>
                .mapToInt(student -> student.getScore()) IntStream
                .average() OptionalDouble
                .getAsDouble();

        System.out.println("평균 점수: " + avg);
    }
}
```

☞스트림 Interface

- java.util.stream패키지에는 BaseStream인터페이스를 부모로 한 많은 자식 인터페이스를 제공한다.
- BaseStream에는 모든 스트림에서 사용할 수 있는 공통 메소드들이 정의 되어 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.Collection인터페이스는 stream()과 parallelStream() 메소드를 가지고 있어 자식 인터페이스인 List와 Set 인터페이스를 구현한 모든 컬렉션에서 객체 스트림을 얻을 수 있다.

♣배열로부터 스트림 얻기

- java.util.Arrays클래스로 다양한 종류의 배열로부터 스트림을 얻을 수 있다.

♣숫자 범위로부터 스트림 얻기

- IntStream또는 LongStream의 정적 메소드인 range()와 rangeClosed() 메소드로 특정 범위의 정수 스트림을 얻을 수 있다.

♣파일로부터 스트림 얻기

- java.nio.file.Files의 lines() 메소드로 텍스트 파일의 행 단위 스트림을 얻을 수 있다.

☞ Collection으로부터 Stream 사용 예제

```
public class Ex04StreamExample {
    new *
    public static void main(String[] args) {
        //List 컬렉션 생성
        List<Product> list = new ArrayList<>();
        for(int i=1; i<=5; i++) {
            Product product = new Product(i, name: "상품"+i, company: "멋진회사", (int)(10000*Math.random()));
            list.add(product);
        }

        //객체 스트림 얻기
        Stream<Product> stream = list.stream();
        stream.forEach(p -> System.out.println(p));
    }
}
```

☞ Array로부터 스트림 얻기

```
public class Ex05StreamExample {
    public static void main(String[] args) {
        System.out.println("--1. String Array --------------");
        String[] strArray = { "송중기", "유재석", "이가현"};
        Stream<String> strStream = Arrays.stream(strArray);
        strStream.forEach(item -> System.out.print(item + ","));

        System.out.println("\n--2. int Array --------------");
        int[] intArray = { 1, 2, 3, 4, 5 };
        IntStream intStream = Arrays.stream(intArray);
        intStream.forEach(item -> System.out.print(item + ","));
        System.out.println();
    }
}
```

☞숫자 범위로부터 스트림 얻기

: IntStream 또는 LongStream의 정적 메소드인 range() 와 rangeClosed() 메소드를 이용하면 특정 범위의 정수 스트림을 얻을 수 있다.

끝수를 포함하지 않으면 range() , 끝 수를 포함하면 rangeCloed() 사용한다.

static IntStream	**range**(int startInclusive, int endExclusive) Returns a sequential ordered IntStream from startInclusive (inclusive) to endExclusive (exclusive) by an incremental step of 1.
static IntStream	**rangeClosed**(int startInclusive, int endInclusive) Returns a sequential ordered IntStream from startInclusive (inclusive) to endInclusive (inclusive) by an incremental step of 1.

```java
public class Ex06StreamExample {
    2 usages
    public static int sum;
    public static void main(String[] args) {
        IntStream stream = IntStream.rangeClosed(1, 100);
        stream.forEach(a -> sum += a);
        System.out.println("총합: " + sum);
    }
}
```

☞File로부터 스트림 얻기

: java.nio.file.Files의 lines() 메소드를 이용하면 텍스트파일의 행 단위 스트림을 얻을 수 있다. 텍스트파일에서 한 행씩 읽고 처리할 때 유용하다.

data.txt

```
{"pno":1, "name":"상품1", "company":"Grace 주식회사", "price":1558}
{"pno":2, "name":"상품2", "company":"Grace 주식회사", "price":4671}
{"pno":3, "name":"상품3", "company":"Grace 주식회사", "price":470}
{"pno":4, "name":"상품4", "company":"Grace 주식회사", "price":9584}
{"pno":5, "name":"상품5", "company":"Grace 주식회사", "price":6868}
```

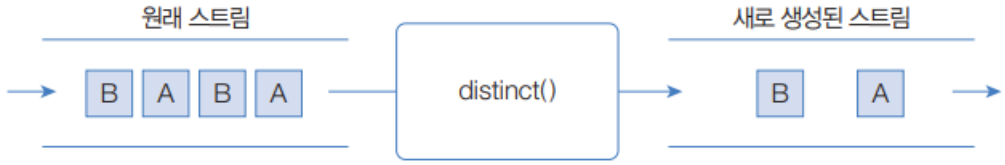
```java
public class Ex07StreamExample {
    new *
    public static void main(String[] args) throws Exception {
        Path path = Paths.get(Ex07StreamExample.class.getResource( name: "data.txt").toURI());
        Stream<String> stream = Files.lines(path, Charset.defaultCharset());
        stream.forEach(line -> System.out.println(line));
        stream.close();
    }
}
```

☞요소 걸러내기 - 필터링

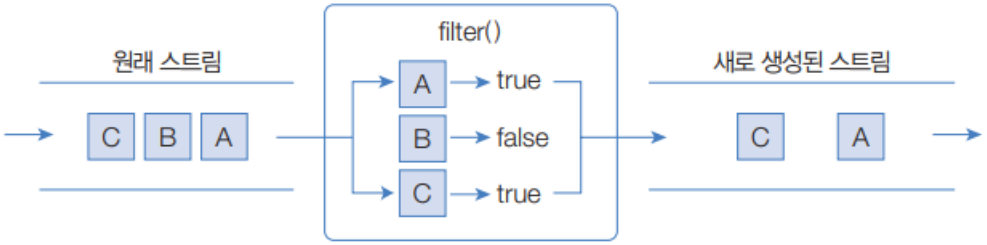
- 필터링은 요소를 걸러내는 중간 처리 기능

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

☞distinct() 메소드: 요소의 중복을 제거



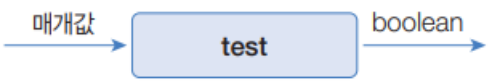
☞filter() 메소드: 매개값으로 주어진 Predicate가 true를 리턴하는 요소만 필터링



Predicate: 함수형 인터페이스

인터페이스	추상 메소드	설명
Predicate<T>	boolean test(T t)	객체 T를 조사
IntPredicate	boolean test(int value)	int 값을 조사
LongPredicate	boolean test(long value)	long 값을 조사
DoublePredicate	boolean test(double value)	double 값을 조사

모든 Predicate는 매개값을 조사한 후 boolean을 리턴하는 test() 메소드를 가지고 있다.



```
T -> { ··· return true }
또는
T -> true;   //return 문만 있을 경우 중괄호와 return 키워드 생략 가능
```

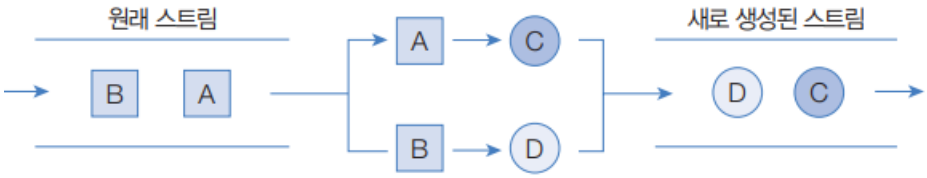
```java
public class Ex08FilteringExample {
    public static void main(String[] args) {
        //List 컬렉션 생성
        List<String> list = new ArrayList<>();
        list.add("이찬범"); list.add("장희정"); list.add("주현솔");  list.add("장희정"); list.add("이가현");

        //중복 요소 제거
        list.stream()
            .distinct()
            .forEach(n -> System.out.println(n));
        System.out.println();

        //"장"으로 시작하는 요소만 필터링
        list.stream()
            .filter(n -> n.startsWith("장"))
            .forEach(n -> System.out.println(n));
        System.out.println();

        //중복 요소를 먼저 제거하고, "장"으로 시작하는 요소만 필터링
        list.stream()
            .distinct()
            .filter(n -> n.startsWith("장"))
            .forEach(n -> System.out.println(n));
    }
}
```

☞요소 변환–매핑

- 스트림의 요소를 다른 요소로 변환하는 중간 처리 기능
- 매핑 메소드: mapXxx(), asDoubleStream(), asLongStream(), boxed(), flatMapXxx() 등이 있다.

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

mapXxx() 메소드: 요소를 다른 요소로 변환한 새로운 스트림을 리턴



인수<T> 와 리턴<R>이 있는Function은 함수형 인터페이스

인터페이스	추상 메소드	매개값 -> 리턴값
Function<T,R>	R apply(T t)	T -> R
IntFunction<R>	R apply(int value)	int -> R
LongFunction<R>	R apply(long value)	long -> R
DoubleFunction<R>	R apply(double value)	double -> R
ToIntFunction<T>	int applyAsInt(T value)	T -> int
ToLongFunction<T>	long applyAsLong(T value)	T -> long
ToDoubleFunction<T>	double applyAsDouble(T value)	T -> double
IntToLongFunction	long applyAsLong(int value)	int -> long
IntToDoubleFunction	double applyAsDouble(int value)	int -> double
LongToIntFunction	int applyAsInt(long value)	long -> int
LongToDoubleFunction	double applyAsDouble(long value)	long -> double
DoubleToIntFunction	int applyAsInt(double value)	double -> int
DoubleToLongFunction	long applyAsLong(double value)	double -> long

모든 Function은 매개값을 리턴값으로매핑(변환)하는 applyXxx() 메소드를 가짐


```
T -> { … return R; }
또는
T -> R;   //return 문만 있을 경우 중괄호와 return 키워드 생략 가능
```

```
6  public class EX09MapExample {
       new *
7      public static void main(String[] args) {
8          //List 컬렉션 생성
9          List<Student> studentList = new ArrayList<>();
10         studentList.add(new Student( name: "홍길동",  score: 85));
11         studentList.add(new Student( name: "홍길동",  score: 92));
12         studentList.add(new Student( name: "홍길동",  score: 87));
13
14         //Student를 score 스트림으로 변환
15         studentList.stream() Stream<Student>
16             .mapToInt(s -> s.getScore()) IntStream
17             .forEach(score -> System.out.println(score));
18     }
19 }
```

기본 타입 간의 변환이거나 기본 타입 요소를 래퍼(Wrapper) 객체 요소로 변환하려면 간편화 메소드를 사용할 수 있다.

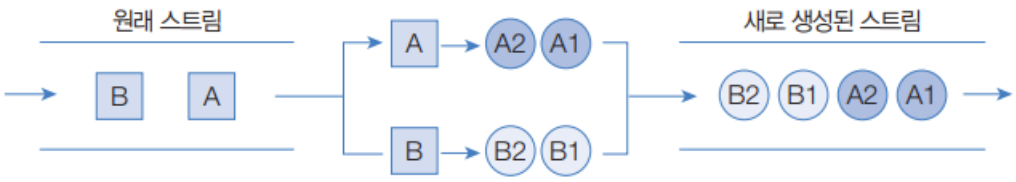
리턴 타입	메소드(매개변수)	설명
LongStream	asLongStream( )	int –› long
DoubleStream	asDoubleStream( )	int –› double long –› double
Stream⟨Integer⟩ Stream⟨Long⟩ Stream⟨Double⟩	boxed( )	int –›Integer long –› Long double –› Double

```java
public class Ex10MapExample {
    public static void main(String[] args) {
        int[] intArray = { 1, 2, 3, 4, 5};

        IntStream intStream = Arrays.stream(intArray);
        intStream
            .asDoubleStream() //int형 Stream을 ---> dobule형 Stream으로 변환
            .forEach(d -> System.out.println(d));

        System.out.println();

        intStream = Arrays.stream(intArray);
        intStream
            .boxed() // 기본타입 Stream을  래퍼 Stream<Integer>로 변환
            .forEach(obj -> System.out.println(obj.intValue()));
    }
}
```

요소를 복수 개의 요소로 변환

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

리턴 타입	메소드(매개변수)	요소 -> 변환 요소
Stream<R>	flatMap(Function<T, Stream<R>>)	T -> Stream<R>
DoubleStream	flatMap(DoubleFunction<DoubleStream>)	double -> DoubleStream
IntStream	flatMap(IntFunction<IntStream>)	int -> IntStream
LongStream	flatMap(LongFunction<LongStream>)	long -> LongStream
DoubleStream	flatMapToDouble(Function<T, DoubleStream>)	T -> DoubleStream
IntStream	flatMapToInt(Function<T, IntStream>)	T -> InputStream
LongStream	flatMapToLong(Function<T, LongStream>)	T -> LongStream



원래 스트림의 A요소를 A1,A2 요소로 변환하고 B요소를 B1,B2로 변환하면 A1,A2,B1,B2요소를 가지는 새로운 스트림이 생성된다.

```java
public class Ex11FlatMappingExample {
    public static void main(String[] args) {
        //문장 스트림을 단어 스트림으로 변환
        List<String> list1 = new ArrayList<>();
        list1.add("My name is Heejung.");
        list1.add("Have a nice day!");
        list1.stream().
        flatMap(data -> Arrays.stream(data.split( regex: " ")))
        .forEach(word -> System.out.println(word));

        System.out.println();

        //문자열 숫자 목록 스트림을 숫자 스트림으로 변환
        List<String> list2 = Arrays.asList("10, 20, 30", "40, 50");
        list2.stream() Stream<String>
        .flatMapToInt(data -> {
            String[] strArr = data.split( regex: ",");
            System.out.println("strArr.lenght = " + strArr.length);
            int[] intArr = new int[strArr.length];
            for (int i = 0; i < strArr.length; i++) {
                intArr[i] = Integer.parseInt(strArr[i].trim());
            }
            return Arrays.stream(intArr);
        }) IntStream
        .forEach(number -> System.out.println(number));
    }
}
```

요소 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 가능하다. 그렇지 않다면 ClassCastException발생한다.

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

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

```
public class Student implements Comparable<Student> {
    2 usages
    private String name;
    4 usages
    private int score;
    3 usages    new *
    public Student(String name, int score) {
        this.name = name;
        this.score = score;
    }
    2 usages    new *
    public String getName() { return name; }
    2 usages    new *
    public int getScore() { return score; }
    new *
    @Override
    public int compareTo(Student o) {
        //the value 0 if x == y; a value less than 0 if x < y; and a value greater than 0 if x > y
        return Integer.compare(score, o.score);
    }
}
```

```java
public class Ex12SortingExample {
    new *
    public static void main(String[] args) {
        //List 컬렉션 생성
        List<Student> studentList = new ArrayList<>();
        studentList.add(new Student( name: "홍길동",  score: 30));
        studentList.add(new Student( name: "김희선",  score: 10));
        studentList.add(new Student( name: "이나영",  score: 20));

        //점수를 기준으로 오름차순으로 정렬한 새 스트림 얻기
        studentList.stream()
            .sorted()
            .forEach(s -> System.out.println(s.getName() + ": " + s.getScore()));
        System.out.println();

        //점수를 기준으로 내림차순으로 정렬한 새 스트림 얻기
        studentList.stream()
        .sorted(Comparator.reverseOrder())
        .forEach(s -> System.out.println(s.getName() + ": " + s.getScore()));
    }
}
```

Comparator를 이용한 정렬

- 요소 객체가 Comparable을 구현하고 있지 않다면, 비교자를 제공하면 요소를 정렬시킬 수 있다.

```
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

```java
public class Student {
    2 usages
    private String name;
    2 usages
    private int score;
    new *
    public Student(String name, int score) {
        this.name = name;
        this.score = score;
    }
    2 usages    new *
    public String getName() { return name; }
    6 usages    new *
    public int getScore() { return score; }
}
```

```java
public class Ex13SortingExample {
    new *
    public static void main(String[] args) {
        //List 컬렉션 생성
        List<Student> studentList = new ArrayList<>();
        studentList.add(new Student( name: "삼순이",  score: 30));
        studentList.add(new Student( name: "삼식이", score: 10));
        studentList.add(new Student( name: "이효리",  score: 20));

        //점수를 기준으로 오름차순으로 정렬한 새 스트림 얻기
        studentList.stream()
            .sorted((s1, s2) -> Integer.compare(s1.getScore(), s2.getScore()))
            .forEach(s -> System.out.println(s.getName() + ": " + s.getScore()));
        System.out.println();

        //점수를 기준으로 내림차순으로 정렬한 새 스트림 얻기
        studentList.stream()
            .sorted((s1, s2) -> Integer.compare(s2.getScore(), s1.getScore()))
            .forEach(s -> System.out.println(s.getName() + ": " + s.getScore()));
    }
}
```

8253jang@daum.net
장희정

☞요소를 하나씩 처리(루핑)

- 루핑(looping)스트림에서 요소를 하나씩 반복해서 가져와 처리하는 것을 말한다.
- 루핑메소드는 peek() - 중간처리메소드 과 forEach() –최종처리메소드 가 있다.

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

- 매개타입인 Consumer는 함수형 인터페이스로 모든 Consumer는 매개값을 처리(소비)하는 accept() 메소드를 가지고 있다.


```
T -> { … }
또는
T -> 실행문;    //하나의 실행문만 있을 경우 중괄호 생략
```

```java
public class Ex14LoopingExample {
    public static void main(String[] args) {
        int[] intArr = { 1, 2, 3, 4, 5 };

        System.out.println("1.잘못 작성한 경우 ------------");
        //잘못 작성한 경우
        Arrays.stream(intArr)
            .filter(a -> a%2==0)
            .peek(n -> System.out.println(n));      //최종 처리가 없으므로 동작하지 않음

        //중간 처리 메소드 peek()을 이용해서 반복 처리
        System.out.println("2. 중간 처리 메소드 peek()을 이용해서 반복 처리------------");
        int total = Arrays.stream(intArr)
            .filter(a -> a%2==0)
            .peek(n -> System.out.println(n))       //동작함
            .sum(); //최종 처리
        System.out.println("총합: " + total + "\n");

        //최종 처리 메소드 forEach()를 이용해서 반복 처리
        System.out.println("3. 최종 처리 메소드 forEach()를 이용해서 반복 처리------------");
        Arrays.stream(intArr)
            .filter(a -> a%2==0)
            .forEach(n -> System.out.println(n));   //최종 처리이므로 동작함
    }
}
```

실행결과

```
1.잘못 작성한 경우 ------------
2. 중간 처리 메소드 peek()을 이용해서 반복 처리------------
2
4
총합: 6

3. 최종 처리 메소드 forEach()를 이용해서 반복 처리------------
2
4


Process finished with exit code 0
```

요소 조건 만족 여부(매칭)

- 요소들이 특정 조건에 만족하는지 여부를 조사하는 최종 처리 기능이다

- 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
booean	noneMatch(Predicate<T> predicate) noneMatch(IntPredicate predicate) noneMatch(LongPredicate predicate) noneMatch(DoublePredicate predicate)	모든 요소가 만족하지 않는지 여부

```
public class Ex15MatchingExample {
    public static void main(String[] args) {
        int[] intArr = { 2, 4 ,6 };

        boolean result = Arrays.stream(intArr)
            .allMatch(a -> a%2==0);
        System.out.println("모두 2의 배수인가? " + result);

        result = Arrays.stream(intArr)
            .anyMatch(a -> a%3==0);
        System.out.println("하나라도 3의 배수가 있는가? " + result);

        result = Arrays.stream(intArr)
            .noneMatch(a -> a%3==0);
        System.out.println("3의 배수가 없는가?  " + result);
    }
}
```

집계(Aggregate)

- Aggregate는 최종 처리 기능으로 요소들을 처리해서 카운팅, 합계, 평균값, 최대값, 최소값 등

과 같이 하나의 값으로 산출하는 것을 말한다. 즉, 대량의 데이터를 가공해서 하나의 값으로 축소하는리덕션(Reduction)이라고 볼 수 있다.

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

AggregateExample.java

```
public class AggregateExample{
public static void main(String[] args) {
//정수 배열
int[] arr= {1, 2, 3, 4, 5};

//카운팅
long count = Arrays.stream(arr)
                .filter(n ->n%2==0)
                .count();
System.out.println("2 의 배수 개수: " + count);

//총합
long sum = Arrays.stream(arr)
                .filter(n ->n%2==0)
                .sum();
System.out.println("2 의 배수의 합: " + sum);

//평균
double avg = Arrays.stream(arr)
                .filter(n ->n%2==0)
                .average()
                .getAsDouble();
System.out.println("2 의 배수의 평균: " + avg);
```

```
//최대값
int max = Arrays.stream(arr)
                .filter(n ->n%2==0)
                .max()
                .getAsInt();
System.out.println("최대값: " + max);

//최소값
int min = Arrays.stream(arr)
   .filter(n ->n%2==0)
                .min()
                .getAsInt();
System.out.println("최소값: " + min);

//첫 번째 요소
int first = Arrays.stream(arr)
                .filter(n ->n%3==0)
                .findFirst()
                .getAsInt();
System.out.println("첫 번째 3 의 배수: " + first);
    }
}
```

실행결과

```
2의 배수 개수: 2
2의 배수의 합: 6
2의 배수의 평균: 3.0
최대값: 4
최소값: 2
첫 번째 3의 배수: 3
```

Optional 클래스

- Optional, OptionalDouble, OptionalInt, OptionalLong클래스는 단순히 집계값만 저장하는 것이 아니라, 집계값이 없으면 디폴트 값을 설정하거나 집계값을 처리하는 Consumer를 등록

리턴 타입	메소드(매개변수)	설명
boolean	isPresent( )	집계값이 있는지 여부
T double int long	orElse(T) orElse(double) orElse(int) orElse(long)	집계값이 없을 경우 디폴트 값 설정
void	ifPresent(Consumer) ifPresent(DoubleConsumer) ifPresent(IntConsumer) ifPresent(LongConsumer)	집계값이 있을 경우 Consumer에서 처리

최종 처리에서 average 사용 시 요소 없는 경우를 대비하는 방법

1) isPresent() 메소드가 true를 리턴할 때만 집계값을 얻는다.
2) orElse() 메소드로집계값이 없을 경우를 대비해서 디폴트 값을 정해놓는다.
3) ifPresent() 메소드로 집계값이 있을 경우에만 동작하는 Consumer 람다식을 제공한다.

```
/**
 *  Optional,  OptionalXxx 클래스들은 단순히 집계값만 저장하는 것이 아니라
*  집계값이 존재하지 않을 경우 디폴트 값을 설정하거나 집계값을 처리하는 Consumer 를 등록할수 있다.
 * */
public class OptionalExample{
public static void main(String[] args) {
List<Integer>list = new ArrayList<>();

/*//예외 발생(java.util.NoSuchElementException)
      double avg = list.stream()
         .mapToInt(Integer :: intValue)
         .average()
         .getAsDouble();
      */

      //방법1
OptionalDouble optional = list.stream()
         .mapToInt(Integer :: intValue)
         .average();
if(optional.isPresent()) {
System.out.println("방법 1_평균: " + optional.getAsDouble());
      } else {
System.out.println("방법 1_평균: 0.0");
      }

//방법2
double avg = list.stream()
         .mapToInt(Integer :: intValue)
         .average()
         .orElse(0.0);
```

```
System.out.println("방법 2_평균: " + avg);

//방법 3
list.stream()
        .mapToInt(Integer :: intValue)
        .average()
        .ifPresent(a ->System.out.println("방법 3_평균: " + a));
    }
}
```

요소 커스텀 집계

- 스트림은 기본 집계 메소드인 sum(), average(), count(), max(), min()을 제공하지만, 다양한 집계 결과물을 만들 수 있도록 reduce() 메소드도 제공한다.
- reduce()는 스트림에 요소가 없을 경우 예외가 발생하지만, identity 매개값이 주어지면 이 값을 디폴트 값으로 리턴한다.

인터페이스	리턴 타입	메소드(매개변수)
Stream	Optional<T>	reduce(BinaryOperator<T> accumulator)
	T	reduce(T identity, BinaryOperator<T> accumulator)
IntStream	OptionalInt	reduce(IntBinaryOperator op)
	int	reduce(int identity, IntBinaryOperator op)
LongStream	OptionalLong	reduce(LongBinaryOperator op)
	long	reduce(long identity, LongBinaryOperator op)
DoubleStream	OptionalDouble	reduce(DoubleBinaryOperator op)
	double	reduce(double identity, DoubleBinaryOperator op)

```
public class Ex17ReductionExample {
public static void main(String[] args) {
List<Student>studentList= Arrays.asList(
new Student("홍길동", 92),
new Student("꽃미남", 95),
new Student("미소왕", 88)
        );
//방법 1
System.out.println("방법 1 ----------");
int sum1 = studentList.stream()
                .mapToInt(Student :: getScore)
                .sum();
```

```
System.out.println("sum1: " + sum1);

System.out.println("방법 2 ----------");
//방법 2
int sum2 = studentList.stream()
                        .map(Student :: getScore)
//.reduce(0, (a, b) ->a+b); //a 는 결과값,  b 는 현재값
.reduce(0, (a, b) -> {
System.out.println("a = "+a +" , b = "   +b);
return a+b;
                    });
System.out.println("sum2: " + sum2);
    }
}
```

실행결과

```
sum1: 275

a = 0 , b = 92
a = 92 , b = 95
a = 187 , b = 88
sum2: 275
```

필터링한 요소 수집

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

```
public class Ex18CollectExample {
public static void main(String[] args) {
List<Student>totalList= new ArrayList<>();
totalList.add(new Student("장희정", "남", 92));
totalList.add(new Student("하승현", "여", 87));
totalList.add(new Student("오문정", "남", 95));
totalList.add(new Student("김은영", "여", 93));

//남학생만 묶어 List 생성
/*List<Student>maleList = totalList.stream()
                .filter(s->s.getSex().equals("남"))
                .collect(Collectors.toList());*/

List<Student>maleList= totalList.stream()
                .filter(s->s.getSex().equals("남"))
                .toList();

maleList.stream()
            .forEach(s ->System.out.println(s.getName()));

System.out.println();

//학생 이름을 키, 학생의 점수를 값으로 갖는 Map 생성
Map<String, Integer>map = totalList.stream()
                .collect(
Collectors.toMap(
s ->s.getName(), //Student 객체에서 키가 될 부분 리턴
s ->s.getScore() //Student 객체에서 값이 될 부분 리턴
)
            );
```

```
System.out.println(map);
	}
}
```