

람다식

강사 : 강병준

이것이 자

람다식이란

❖ 자바 8부터 함수적 프로그래밍 위해 람다식 지원

■ 람다식(Lambda Expressions)을 언어 차원에서 제공

- 람다 계산법에서 사용된 식을 프로그래밍 언어에 접목
- 익명 함수(anonymous function)을 생성하기 위한 식

■ 자바에서 람다식을 수용한 이유

- 코드가 매우 간결해진다.
- 컬렉션 요소(대용량 데이터)를 필터링 또는 매핑해 쉽게 집계

■ 자바는 람다식을 함수적 인터페이스의 익명 구현 객체로 취급

람다식 → 매개변수를 가진 코드 블록 → 익명 구현 객체

- 어떤 인터페이스를 구현할지는 대입되는 인터페이스에 달려있음

Runnable runnable = `() -> { ... };` ● ----- 람다식

람다식 기본 문법

❖ 함수적 스타일의 람다식 작성법

```
(타입 매개변수, ...) -> { 실행문; ... }
```

```
(int a) -> { System.out.println(a); }
```

- 매개 타입은 런타임시에 대입값 따라 자동 인식 → 생략 가능
- 하나의 매개변수만 있을 경우에는 괄호() 생략 가능
- 하나의 실행문만 있다면 중괄호 { } 생략 가능
- 매개변수 없다면 괄호 () 생략 불가
- 리턴값이 있는 경우, return 문 사용
- 중괄호 { }에 return 문만 있을 경우, 중괄호 생략 가능

타겟 타입과 함수적 인터페이스

❖ 타겟 타입(target type)

- 람다식이 대입되는 인터페이스
- 익명 구현 객체를 만들 때 사용할 인터페이스

인터페이스 변수 = 람다식;

❖ 함수적 인터페이스(functional interface)

- 하나의 추상 메소드만 선언된 인터페이스가 타겟 타입
- @FunctionalInterface 어노테이션
 - 하나의 추상 메소드만을 가지는지 컴파일러가 체크
 - 두 개 이상의 추상 메소드가 선언되어 있으면 컴파일 오류 발생

타겟 타입과 함수적 인터페이스

❖ 매개변수와 리턴값이 없는 람다식

- Method()가 매개 변수를 가지지 않는 경우

```
@FunctionalInterface
public interface MyFunctionalInterface {
    public void method();
}
```

```
MyFunctionalInterface fi = () -> { ... }
```

```
fi.method();
```

❖ 매개변수가 있는 람다식

```
@FunctionalInterface
public interface MyFunctionalInterface {
    public void method(int x);
}
```

```
MyFunctionalInterface fi = (x) -> { ... } 또는 x -> { ... }
```

```
fi.method(5);
```

```
interface MyFunction1 {
    public void method();
}
class MyFun1Class implements MyFunction1 {
    public void method() {
        System.out.println("난 재정의한 클래스");
    }
}
public class Lamda1Ex {
    public static void main(String[] args) {
        MyFunction1 mc1 = new MyFun1Class();
        mc1.method();
        mc1 = new MyFunction1() { // 무명(익명) 클래스
            public void method() {
                System.out.println("난 인터페이스 직접 재정의");
            }
        };
        mc1.method();
        mc1 = ()-> {System.out.println("난 람다로 처리한 메서드");};
        mc1.method();
    }
}
```

```
@FunctionalInterface // 함수적 인터페이스인지 체크
interface MyFun2 {      void method(int x);      }
```

```
public class Lamda2 {
    public static void main(String[] args) {
        MyFun2 mf2;
        mf2 =(x)-> {
            int result = x * 5;
            System.out.println("결과 : "+result);
        };
        mf2.method(4);
        mf2=(x)-> {System.out.println("결과 : "+x * 7);};
        mf2.method(5);
        mf2=x-> {System.out.println("결과 : "+x * 3);}; // 매개변수가
1개인 경우 ()생략
        mf2.method(3);
        mf2=x-> System.out.println("결과 : "+x * 6); // 문장이 하나인
경우 {}생략
        mf2.method(7);
    }
}
```

타겟 타입과 함수적 인터페이스

❖ 리턴값이 있는 람다식

```
@FunctionalInterface
public interface MyFunctionalInterface {
    public int method(int x, int y);
}
```

```
MyFunctionalInterface fi = (x, y) -> { ...; return 값; }
```

```
int result = fi.method(2, 5);
```

```
MyFunctionalInterface fi = (x, y) -> {
    return x + y;
}
```



```
MyFunctionalInterface fi = (x, y) -> x + y;
}
```

```
MyFunctionalInterface fi = (x, y) -> {
    return sum(x, y);
}
```



```
MyFunctionalInterface fi = (x, y) -> sum(x, y);
```




```
@FunctionalInterface
```

```
interface MyInter3 {
```

```
    int methid(int x, int y);
```

```
}
```

```
public class LamReturn1 {
```

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

```
        MyInter3 mi;
```

```
        mi = (x,y)-> {
```

```
            int result = x + y;
```

```
            return result;
```

```
        };
```

```
        System.out.println("결과 : "+mi.methid(12,7));
```

```
        mi = (x,y)-> { return x + y; };
```

```
        System.out.println("결과 : "+mi.methid(12,11));
```

```
        mi = (x,y)-> x + y;
```

```
        System.out.println("결과 : "+mi.methid(21,11));
```

```
    }
```

```
}
```

```
class Student1 {
    private String name;
    private int age;
    public Student1(String name, int age) {
        this.name = name; this.age = age;
    }
    public String getName() { return name; }
    public int getAge() { return age; }
}

public class LamdaStream1Ex {
    public static void main(String[] args) {
        List<Student1> list = Arrays.asList(new Student1("세정", 20),
                                             new Student1("길동", 25), new Student1("강다니엘", 22));
        Stream<Student1> stream = list.stream();
        stream.forEach(s-> {
            System.out.println(s.getName()+" : "+s.getAge());
        });
    }
}
```

스트림과 병렬처리

강사 : 강병준

스트림 소개

❖ 스트림이란?

- 자바 8부터 추가된 컬렉션(배열 포함)의 저장 요소를 하나씩 참조
- 람다식 (함수적-스타일(functional-style))으로 처리할 수 있도록 해주는 반복자

❖ 반복자 스트림

```
List<String> list = Arrays.asList("홍길동", "신용권", "감자바");  
Iterator<String> iterator = list.iterator();  
while(iterator.hasNext()) {  
    String name = iterator.next();  
    System.out.println(name);  
}
```



```
List<String> list = Arrays.asList("홍길동", "신용권", "감자바");  
Stream<String> stream = list.stream();  
stream.forEach( name -> System.out.println(name) );
```

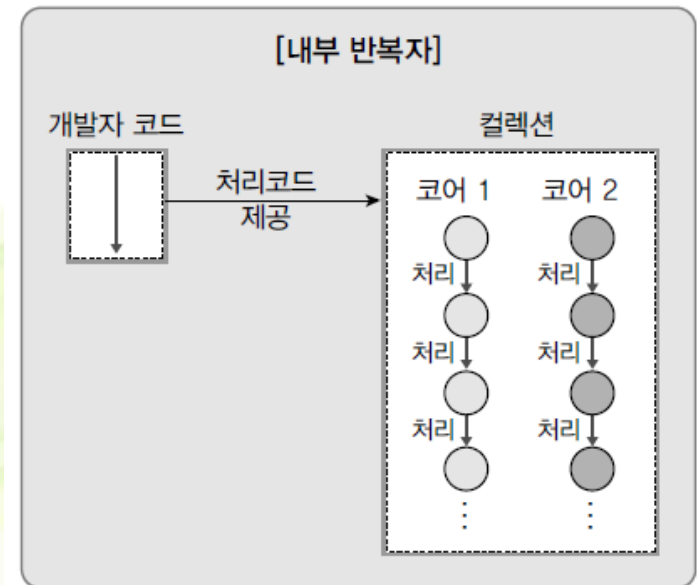
스트림 소개

```
import java.util.Arrays;
import java.util.Iterator;
import java.util.List;
import java.util.stream.Stream;
public class IteratorVsStreamExample {
    public static void main(String[] args) {
        List<String> list = Arrays.asList("홍길동", "설현", "하니", "모하니");
        //Iterator 이용
        Iterator<String> iterator = list.iterator();
        while(iterator.hasNext()) {
            String name = iterator.next();
            System.out.println(name);
        }
        System.out.println();
        //Stream 이용
        Stream<String> stream = list.stream();
        stream.forEach( name -> System.out.println(name) );
    }
}
```

스트림 소개

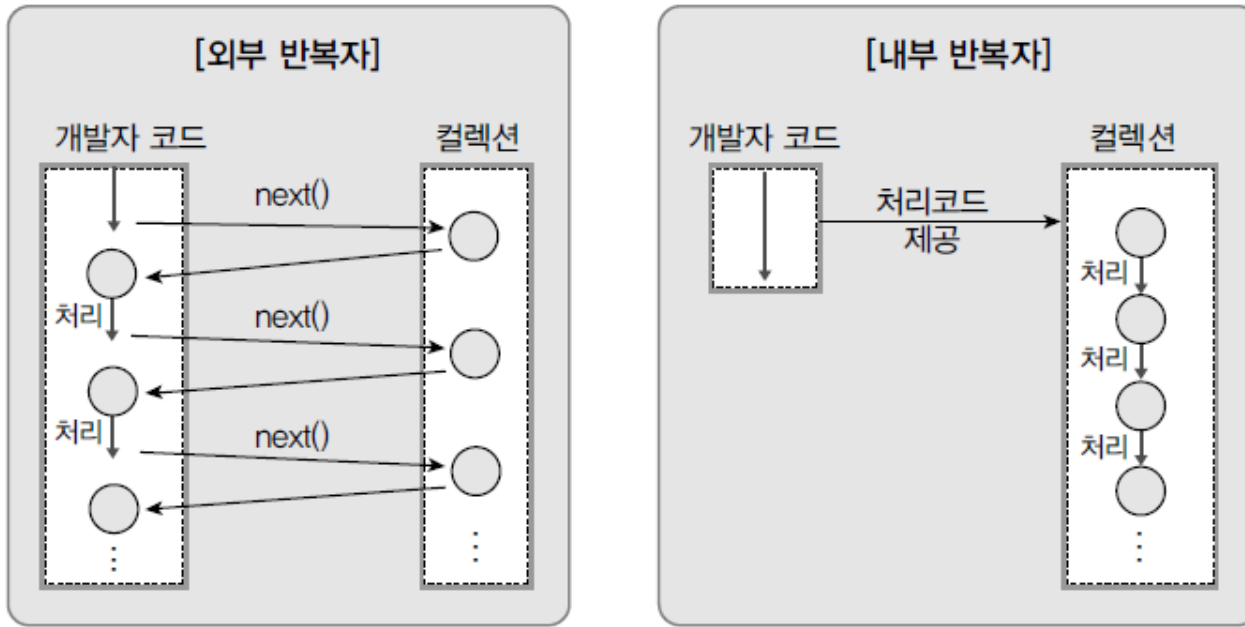
❖ 스트림의 특징

- Iterator와 비슷한 역할 하는 반복자
 - 대부분의 요소처리 메소드는 함수적 인터페이스 매개 타입
- 랴다식으로 요소 처리 코드 제공
 - 대부분의 요소처리 메소드는 함수적 인터페이스 매개 타입
- 내부 반복자 사용하므로 병렬 처리 쉬움
 - 컬렉션 내부에서 요소들 반복 시킴
 - 개발자는 요소당 처리해야 할 코드만 제공



스트림 소개

■ 내부 반복자와 외부 반복자의 비교



외부반복자 : for문 또는 iterator이용

내부반복자 : 컬렉션 내부요소(메소드나 멤버변수 사용)

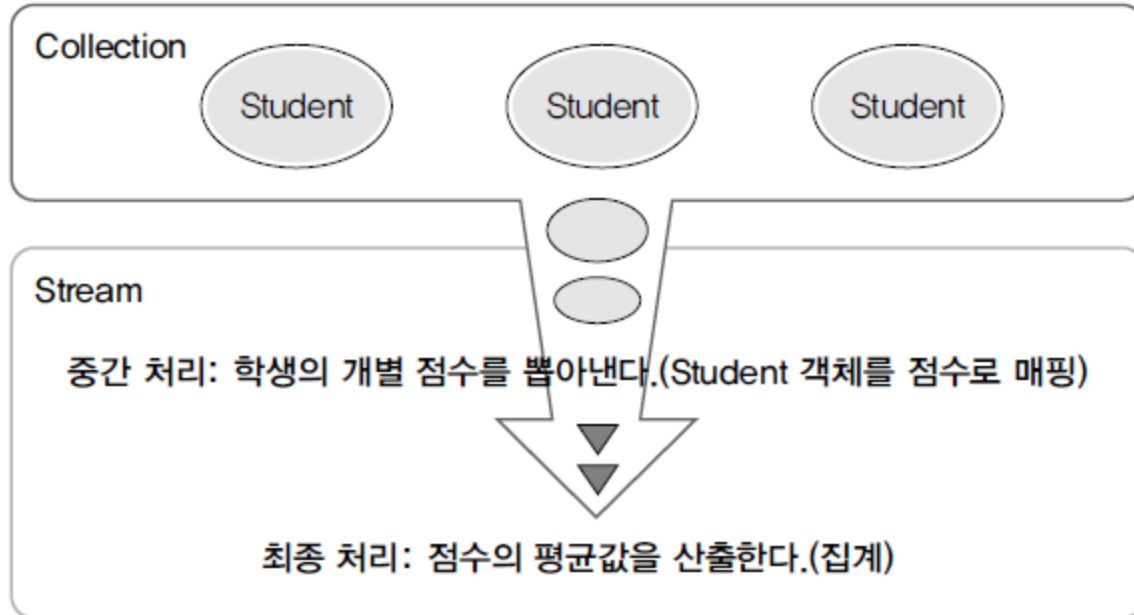
스트림 소개

```
import java.util.Arrays;
import java.util.List;
import java.util.stream.Stream;
class Student {
    private String name;    private int score;
    public Student (String name, int score) {
        this.name = name;    this.score = score;
    }
    public String getName() { return name; }
    public int getScore() { return score; }
}

public class LambdaExpressionsExample {
    public static void main(String[] args) {
        List<Student> list = Arrays.asList(
            new Student("홍길동", 90), new Student("신용권", 92));
        Stream<Student> stream = list.stream();
        stream.forEach(s -> {
            String name = s.getName();
            int score = s.getScore();
            System.out.println(name + "-" + score);
        });
    }
}
```


스트림 소개

- 스트림은 중간 처리와 최종 처리가 가능



- **중간 : 매핑, 필터링, 정렬** **최종 : 반복, 카운팅, 평균, 총합**

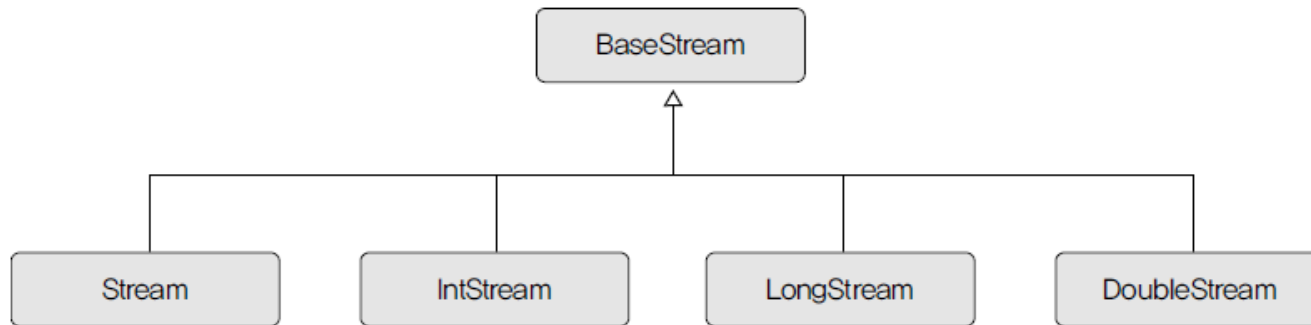
스트림 소개

```
import java.util.Arrays;
import java.util.List;
public class MapAndReduceExample {
    public static void main(String[] args) {
        List<Student> studentList = Arrays.asList(
            new Student("유재석", 10),
            new Student("박명수", 20),
            new Student("정준하", 30)
        );
        double avg = studentList.stream()
            //중간처리(학생 객체를 점수로 매핑)
            .mapToInt(Student :: getScore)
            //최종 처리(평균 점수)
            .average()
            .getAsDouble();
        System.out.println("평균 점수: " + avg);
    }
}
```

스트림의 종류

❖ 스트림의 종류

- 자바 8부터 새로 추가
- `java.util.stream` 패키지에 스트림(stream) API 존재



- **BaseStream**
 - 모든 스트림에서 사용 가능한 공통 메소드 (직접 사용 X)
- **Stream** – 객체 요소 처리
- **IntStream**, **LongStream**, **DoubleStream**은 각각 기본 타입인 `int`, `long`, `double` 요소 처리

스트림의 종류

■ 스트림 인터페이스의 구현 객체

- 주로 컬렉션과 배열에서 얻음
- 소스로부터 스트림 구현 객체 얻는 경우

리턴 타입	메소드(매개 변수)	소스
Stream<T>	java.util.Collection.stream() java.util.Collection.parallelStream()	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.find(Path, int, BiPredicate, FileVisitOption) Files.list(Path)	디렉토리
Stream<String>	Files.lines(Path, Charset) BufferedReader.lines()	파일
DoubleStream IntStream LongStream	Random.doubles(...) Random.ints() Random.longs()	랜덤 수

스트림의 종류

❖ 파일로부터 스트림 얻기

- Files의 정적 메소드인 lines ()와 BufferedReader의 lines () 메소드 이용
- 문자 파일 내용을 스트림 통해 행 단위로 읽고 콘솔에 출력

❖ 디렉토리로부터 스트림 얻기

- Files의 정적 메소드인 list () 이용
- 디렉토리 내용(서브 디렉토리 또는 파일 목록)을 스트림 통해 읽고 콘솔에 출력

스트림의 종류

```
import java.util.Arrays;
import java.util.stream.IntStream;
import java.util.stream.Stream;
```

```
public class FromArrayExample {
    public static void main(String[] args) {
        String[] strArray = { "홍길동", "유재석", "하하" };
        Stream<String> strStream = Arrays.stream(strArray);
        strStream.forEach(a -> System.out.print(a + ","));
        System.out.println();

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

스트림의 종류

```
import java.util.Arrays;
import java.util.List;
import java.util.stream.Stream;

public class FromCollectionExample {
    public static void main(String[] args) {
        List<Student> studentList = Arrays.asList(
            new Student("홍길동", 10),
            new Student("박명수", 20),
            new Student("설현", 30)
        );
        Stream<Student> stream = studentList.stream();
        stream.forEach(s -> System.out.println(s.getName()));
    }
}
```

스트림의 종류

```
import java.io.IOException;
import java.nio.file.Files;
import java.nio.file.Path;
import java.nio.file.Paths;
import java.util.stream.Stream;
```

```
public class FromDirectoryExample {
    public static void main(String[] args) throws IOException {
        // Path java.nio에 추가된 java.io의 File과 비슷한 역할
        Path path = Paths.get("./src/ch20/");
        Stream<Path> stream = Files.list(path);
        stream.forEach( p -> System.out.println(p.getFileName()) );
        list.close();
    }
}
```


스트림의 종류

```
import java.util.stream.IntStream;

public class FromIntRangeExample {
    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);
    }
}
```

스트림 파이프라인

❖ 리덕션(Reduction)

- 대량의 데이터를 가공해 축소하는 것
- 데이터의 합계, 평균값, 카운팅, 최대값, 최소값
- 컬렉션의 요소를 리덕션의 결과물로 바로 집계할 수 없을 경우에는?
 - 집계하기 좋도록 필터링, 매핑, 정렬, 그룹핑 등의 중간 처리 필요
→ 스트림 파이프 라인의 필요성

❖ 파이프라인

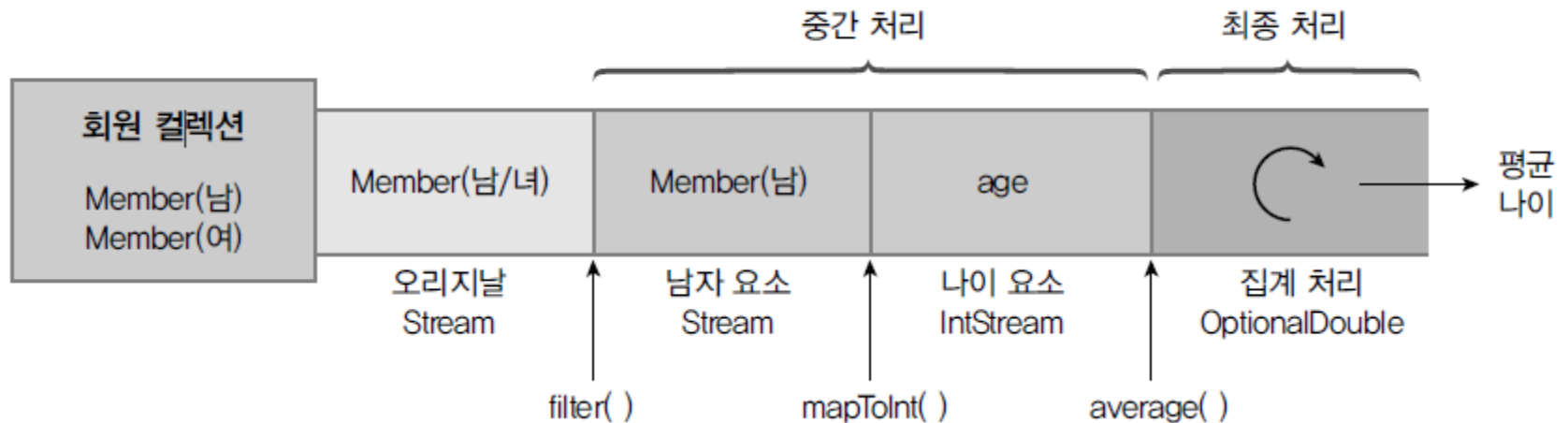
- 여러 개의 스트림이 연결되어 있는 구조
- 파이프라인에서 최종 처리를 제외하고는 모두 중간 처리 스트림



스트림 파이프라인

❖ 중간 처리와 최종 처리

- Stream 인터페이스는 필터링, 매핑, 정렬 등의 많은 중간 처리 메소드 가짐
- 메소드들은 중간 처리된 스트림 리턴
- 스트림에서 다시 중간 처리 메소드 호출해 파이프라인 형성
- Ex) 회원들 중 남자 회원들의 나이 평균 구하기



스트림 파이프라인

❖ 중간 처리 메소드와 최종 처리 메소드

■ 스트림이 제공하는 중간 처리용 메소드 – 리턴 타입이 스트림

종류	리턴 타입	메소드(매개 변수)	소속된 인터페이스
중간 처리	필터링	distinct()	공통
		filter(...)	공통
	매핑	flatMap(...)	공통
		flatMapToDouble(...)	Stream
		flatMapToInt(...)	Stream
		flatMapToLong(...)	Stream
		map(...)	공통
		mapToDouble(...)	Stream, IntStream, LongStream
		mapToInt(...)	Stream, LongStream, DoubleStream
		mapToLong(...)	Stream, IntStream, DoubleStream
		mapToObj(...)	IntStream, LongStream, DoubleStream
		asDoubleStream()	IntStream, LongStream
		asLongStream()	IntStream
		boxed()	IntStream, LongStream, DoubleStream
	정렬	sorted(...)	공통
	루핑	peek(...)	공통

스트림 파이프라인

■ 스트림이 제공하는 최종 처리용 메소드

- 리턴 타입이 기본 타입이거나 OptionalXXX

종류		리턴 타입	메소드(매개 변수)	소속된 인터페이스
최종 처리	매칭	boolean	allMatch(...)	공통
		boolean	anyMatch(...)	공통
		boolean	noneMatch(...)	공통
	집계	long	count()	공통
		OptionalXXX	findFirst()	공통
		OptionalXXX	max(...)	공통
		OptionalXXX	min(...)	공통
		OptionalDouble	average()	IntStream, LongStream, DoubleStream
		OptionalXXX	reduce(...)	공통
		int, long, double	sum()	IntStream, LongStream, DoubleStream
	루핑	void	forEach(...)	공통
	수집	R	collect(...)	공통

필터링

❖ 필터링이란?

- 중간 처리 기능으로 요소 걸러내는 역할
- 필터링 메소드인 `distinct ()`와 `filter ()` 메소드
 - 모든 스트림이 가지고 있는 공통 메소드

리턴 타입	메소드(매개 변수)	설명
Stream IntStream LongStream DoubleStream	<code>distinct()</code>	중복 제거
	<code>filter(Predicate)</code>	조건 필터링
	<code>filter(IntPredicate)</code>	
	<code>filter(LongPredicate)</code>	
	<code>filter(DoublePredicate)</code>	

스트림 파이프라인

```
public class Member {  
    public static int MALE = 0;  
    public static int FEMALE = 1;  
  
    private String name;  
    private int sex;  
    private int age;  
  
    public Member(String name, int sex, int age) {  
        this.name = name;  
        this.sex = sex;  
        this.age = age;  
    }  
  
    public int getSex() { return sex; }  
    public int getAge() { return age; }  
}
```

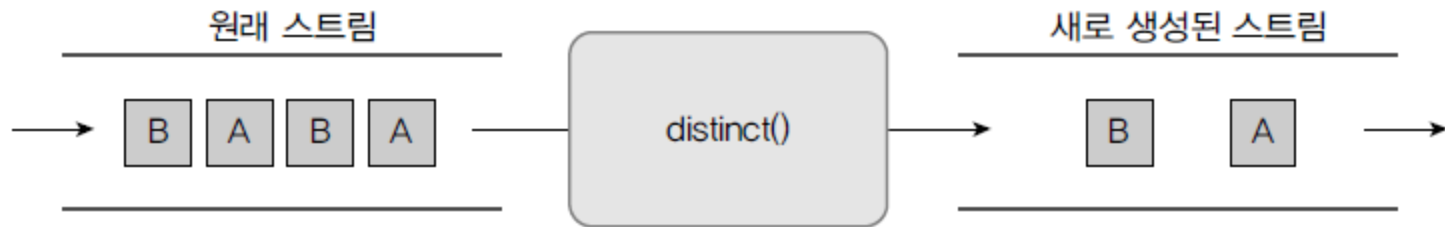
필터링

```
public class StreamPipelinesExample {  
    public static void main(String[] args) {  
        List<Member> list = new ArrayList<>();  
        list.add(new Member("설현",23,Member.FEMALE));  
        list.add(new Member("원빈",33,Member.MALE));  
        list.add(new Member("하니",26,Member.FEMALE));  
        list.add(new Member("공유",38,Member.MALE));  
        list.add(new Member("수지",24,Member.FEMALE));  
  
        double avg = list.stream().mapToInt(Member::getAge)  
            .average().getAsDouble();  
        System.out.println("평균나이 : " + avg);  
        avg = list.stream().filter(m -> m.getGender()==Member.MALE)  
            .mapToInt(Member::getAge).average().getAsDouble();  
        System.out.println("남자평균나이 : " + avg);  
        avg = list.stream().filter(m -> m.getGender()==Member.FEMALE)  
            .mapToInt(Member::getAge).average().getAsDouble();  
        System.out.println("여자평균나이 : " + avg);  
    }  
}
```


필터링

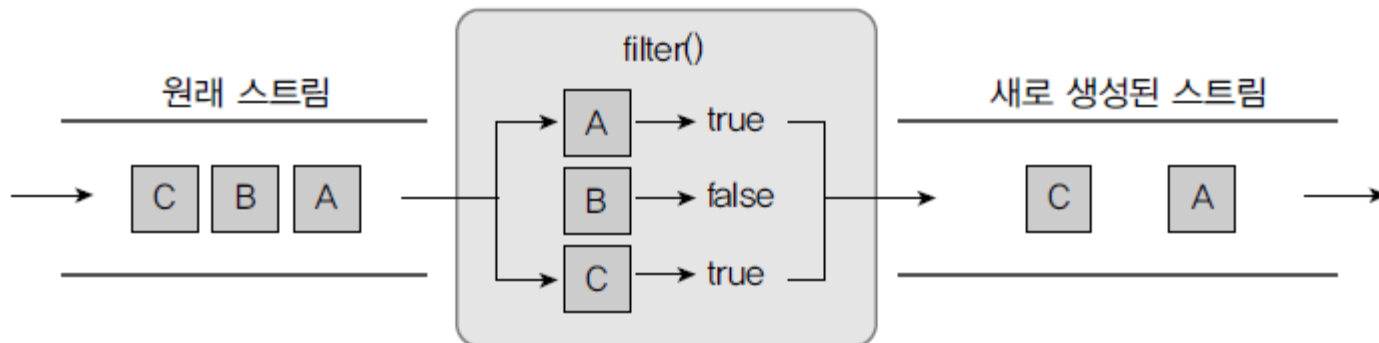
■ distinct () 메소드 – 중복을 제거하는 기능

- Stream의 경우 Object.equals (Object)가 true
 - 동일한 객체로 판단해 중복 제거
- IntStream, LongStream, DoubleStream은 동일값일 경우 중복 제거



■ filter () 메소드

- 매개값으로 주어진 Predicate가 true를 리턴하는 요소만 필터링



필터링

```
import java.util.Arrays;
import java.util.List;
public class FilteringExample {
    public static void main(String[] args) {
        List<String> names = Arrays.asList("홍길동", "신용권", "감자바", "신용권", "신민철");

        names.stream()
            .distinct()
            .forEach(n -> System.out.println(n));
        System.out.println();

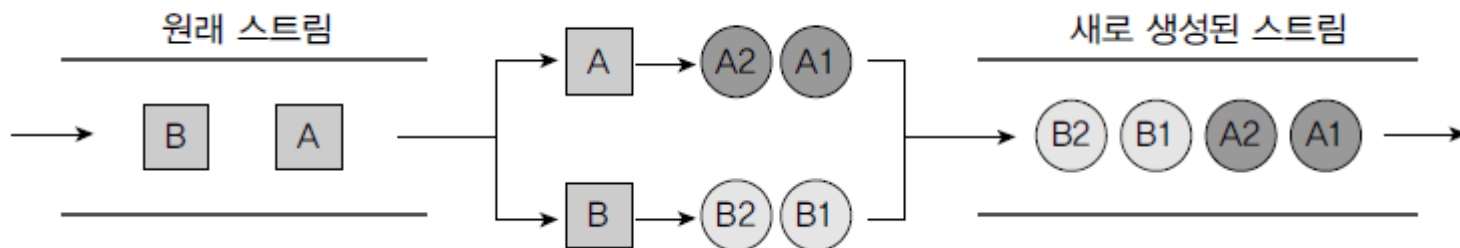
        names.stream()
            .filter(n -> n.startsWith("신"))
            .forEach(n -> System.out.println(n));
        System.out.println();

        names.stream()
            .distinct()
            .filter(n -> n.startsWith("신"))
            .forEach(n -> System.out.println(n));
    }
}
```

매핑

❖ 매핑(mapping)

- 중간 처리 기능으로 스트림의 요소를 다른 요소로 대체하는 작업
- flatMapXXX() 메소드
 - 요소를 대체하는 복수 개의 요소들로 구성된 새로운 스트림 리턴



• flatMapXXX() 메소드의 종류

리턴 타입	메소드(매개 변수)	요소 → 대체 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 → IntStream
LongStream	flatMapToLong(Function<T, LongStream>)	T → LongStream

```
import java.util.Arrays;
import java.util.List;
```

```
public class MapExample {
    public static void main(String[] args) {
        List<Student> studentList = Arrays.asList(
            new Student("홍길동", 10),
            new Student("신용권", 20),
            new Student("유미선", 30)
        );

        studentList.stream()
            .mapToInt(Student :: getScore)
            .forEach(score -> System.out.println(score));
    }
}
```

정렬

❖ 정렬

- 스트림은 요소가 최종 처리되기 전에 중간 단계에서 요소를 정렬
- 최종 처리 순서 변경 가능
- 요소를 정렬하는 메소드

리턴 타입	메소드(매개 변수)	설명
Stream<T>	sorted()	객체를 Comparable 구현 방법에 따라 정렬
Stream<T>	sorted(Comparator<T>)	객체를 주어진 Comparator에 따라 정렬
DoubleStream	sorted()	double 요소를 오름차순으로 정렬
IntStream	sorted()	int 요소를 오름차순으로 정렬
LongStream	sorted()	long 요소를 오름차순으로 정렬

정렬

```
public class Student2 implements Comparable<Student2> {  
    private String name;  
    private int score;  
  
    public Student2(String name, int score) {  
        this.name = name;  
        this.score = score;  
    }  
  
    public String getName() { return name; }  
    public int getScore() { return score; }  
  
    @Override  
    public int compareTo(Student2 o) {  
        return Integer.compare(score, o.score);  
    }  
}
```

정렬

```
import java.util.Arrays;          import java.util.Comparator;
import java.util.List;            import java.util.stream.IntStream;

public class SortingExample {
    public static void main(String[] args) {
        //숫자 요소일 경우
        IntStream intStream = Arrays.stream(new int[] {5, 3, 2, 1, 4});
        intStream.sorted().forEach(n -> System.out.print(n + ", "));
        System.out.println();

        //객체 요소일 경우
        List<Student2> studentList = Arrays.asList( new Student2("홍길동", 30),
            new Student2("신용권", 10), new Student2("유미선", 20));

        studentList.stream()
            .sorted()
            .forEach(s -> System.out.print(s.getScore() + ", "));
        System.out.println();

        studentList.stream()
            .sorted(Comparator.reverseOrder() )
            .forEach(s -> System.out.print(s.getScore() + ", "));
    }
}
```

❖ 루핑(looping)

- 요소 전체를 반복하는 것
- peek()
 - 중간 처리 메소드
 - 중간 처리 단계에서 전체 요소를 루핑하며 추가 작업 하기 위해 사용
 - 최종처리 메소드가 실행되지 않으면 지연
 - 반드시 최종 처리 메소드가 호출되어야 동작
- forEach()
 - 최종 처리 메소드
 - 파이프라인 마지막에 루핑하며 요소를 하나씩 처리
 - 요소를 소비하는 최종 처리 메소드
 - sum()과 같은 다른 최종 메소드 호출 불가

루핑

```
import java.util.Arrays;
public class LoopingExample {
    public static void main(String[] args) {
        int[] intArr = { 1, 2, 3, 4, 5 };
        System.out.println("[peek()를 마지막에 호출한 경우]");
        Arrays.stream(intArr)
            .filter(a -> a%2==0)
            .peek(n -> System.out.println(n)); //동작하지 않음
        System.out.println("[최종 처리 메소드를 마지막에 호출한 경우]");
        int total = Arrays.stream(intArr)
            .filter(a -> a%2==0)
            .peek(n -> System.out.println(n)) //동작함
            .sum();
        System.out.println("총합: " + total);
        System.out.println("[forEach()를 마지막에 호출한 경우]");
        Arrays.stream(intArr)
            .filter(a -> a%2==0)
            .forEach(n -> System.out.println(n)); //동작함
    }
}
```

❖ 매칭이란?

- 최종 처리 단계에서 요소들이 특정 조건에 만족하는지 조사하는 것
- allMatch () 메소드
 - 모든 요소들이 매개값으로 주어진 Predicate의 조건을 만족하는지 조사
- anyMatch() 메소드
 - 최소한 한 개의 요소가 매개값으로 주어진 Predicate 조건을 만족하는지 조사
- noneMatch() 메소드
 - 모든 요소들이 매개값으로 주어진 Predicate의 조건을 만족하지 않는지 조사

매칭

```
import java.util.Arrays;

public class MatchExample {
    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);
    }
}
```

기본 집계

❖ 집계(Aggregate)

- 최종 처리 기능으로 요소들을 처리해 카운팅, 합계, 평균값, 최대값, 최소값 등과 같이 하나의 값으로 산출하는 것
- 집계는 대량의 데이터를 가공해서 축소하는 리덕션 (Reduction)
- 스트림이 제공하는 기본 집계

리턴 타입	메소드(매개 변수)	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

기본 집계

❖ Optional 클래스

- Optional, OptionalDouble, OptionalInt, OptionalLong 클래스
- 저장하는 값의 타입만 다를 뿐 제공하는 기능은 거의 동일
- 단순히 집계 값만 저장하는 것이 아님
- 집계 값이 존재하지 않을 경우 디폴트 값을 설정 가능
- 집계 값을 처리하는 Consumer도 등록 가능
- Optional 클래스들이 제공하는 메소드들

리턴 타입	메소드(매개 변수)	설명
boolean	isPresent()	값이 저장되어 있는지 여부
T double int long	orElse(T) orElse(double) orElse(int) orElse(long)	값이 저장되어 있지 않을 경우 디폴트 값 지정
void	ifPresent(Consumer) ifPresent(DoubleConsumer) ifPresent(IntConsumer) ifPresent(LongConsumer)	값이 저장되어 있을 경우 Consumer에서 처리

커스텀 집계

❖ 커스텀 집계를 위한 메소드

- `sum()`, `average()`, `count()`, `max()`, `min()` 이용
 - 기본 집계 메소드 이용
- `reduce()` 메소드
 - 프로그램화해서 다양한 집계 결과물 만들 수 있도록 제공

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

커스텀 집계

```
import java.util.Arrays;

public class AggregateExample {
    public static void main(String[] args) {
        long count = Arrays.stream(new int[] {1, 2, 3, 4, 5})
            .filter(n -> n%2==0).count();
        System.out.println("2의 배수 개수: " + count);
        long sum = Arrays.stream(new int[] {1, 2, 3, 4, 5})
            .filter(n -> n%2==0).sum();
        System.out.println("2의 배수의 합: " + sum);
        double avg = Arrays.stream(new int[] {1, 2, 3, 4, 5})
            .filter(n -> n%2==0).average().getAsDouble();
        System.out.println("2의 배수의 평균: " + avg);
        int max = Arrays.stream(new int[] {1, 2, 3, 4, 5})
            .filter(n -> n%2==0).max().getAsInt();
        System.out.println("최대값: " + max);
        int min = Arrays.stream(new int[] {1, 2, 3, 4, 5})
            .filter(n -> n%2==0).min().getAsInt();
        System.out.println("최소값: " + min);
        int first = Arrays.stream(new int[] {1, 2, 3, 4, 5})
            .filter(n -> n%3==0).findFirst().getAsInt();
        System.out.println("첫번째 3의 배수: " + first);
    }
}
```

커스텀 집계

```
import java.util.Arrays;
import java.util.List;
public class ReductionExample {
    public static void main(String[] args) {
        List<Student> studentList = Arrays.asList(
            new Student("홍길동", 92),
            new Student("신용권", 95),
            new Student("감자바", 88)
        );
        int sum1 = studentList.stream()
            .mapToInt(Student :: getScore)
            .sum();
        int sum2 = studentList.stream()
            .map(Student :: getScore)
            .reduce((a, b) -> a+b)
            .get();
        int sum3 = studentList.stream()
            .map(Student :: getScore)
            .reduce(0, (a, b) -> a+b);
        System.out.println("sum1: " + sum1);
        System.out.println("sum2: " + sum2);
        System.out.println("sum3: " + sum3);
    }
}
```


❖ 수집 기능

- 요소들을 필터링 또는 매핑 한 후 요소들을 수집하는 최종 처리 메소드 인 `collect ()`
- 필요한 요소만 컬렉션으로 받을 수 있음
- 요소들을 그룹핑 한 후 집계(리덕션)

❖ 필터링한 요소 수집

- `Stream`의 `collect (Collector<T,A,R> collector)` 메소드
 - 필터링 또는 매핑된 요소들을 새로운 컬렉션에 수집하고, 이 컬렉션 리턴

리턴 타입	메소드(매개 변수)	인터페이스
R	<code>collect(Collector<T,A,R> collector)</code>	Stream

수집

❖ 매개값인 Collector collect(Collection<T,A,R> cooker) 반환 R

- 어떤 요소를 어떤 컬렉션에 수집할 것인지 결정
- Collector의 타입 파라미터 T는 요소
- A는 누적기(accumulator)
- R은 요소가 저장될 컬렉션
 - 해석하면 T 요소를 A 누적기가 R에 저장한다는 의미
- Collectors 클래스의 정적 메소드

리턴 타입	Collectors의 정적 메소드	설명
Collector<T, ?, List<T>>	toList()	T를 List에 저장
Collector<T, ?, Set<T>>	toSet()	T를 Set에 저장
Collector<T,?, Collection<T>>	toCollection(Supplier<Collection<T>>)	T를 Supplier가 제공한 Collection에 저장
Collector<T, ?, Map<K,U>>	toMap(Function<T,K> keyMapper, Function< T,U> valueMapper)	T를 K와 U로 매핑해서 K를 키로, U를 값으로 Map에 저장
Collector<T, ?, ConcurrentMap<K,U>>	toConcurrentMap(Function<T,K> keyMapper, Function<T,U> valueMapper)	T를 K와 U로 매핑해서 K를 키로, U를 값으로 ConcurrentMap에 저장

수집

❖ 사용자 정의 컨테이너에 수집하기

- 스트림은 요소들을 필터링, 또는 매핑해 사용자 정의 컨테이너 객체에 수집할 수 있도록
- 이를 위한 추가적인 `collect ()` 메소드

인터페이스	리턴 타입	메소드(매개 변수)
Stream	R	<code>collect(Supplier<R>, BiConsumer<R,? super T>, BiConsumer<R,R>)</code>
IntStream	R	<code>collect(Supplier<R>, ObjIntConsumer<R>, BiConsumer<R,R>)</code>
LongStream	R	<code>collect(Supplier<R>, ObjLongConsumer<R>, BiConsumer<R,R>)</code>
DoubleStream	R	<code>collect(Supplier<R>, ObjDoubleConsumer<R>, BiConsumer<R,R>)</code>

수집

```
public class Student3 {  
    public enum Sex { MALE, FEMALE }  
    public enum City { Seoul, Pusan }  
    private String name;  
    private int score;  
    private Sex sex;  
    private City city;  
  
    public Student3(String name, int score, Sex sex) {  
        this.name = name;    this.score = score;    this.sex = sex;  
    }  
    public Student3(String name, int score, Sex sex, City city) {  
        this.name = name;    this.score = score;  
        this.sex = sex;      this.city = city;  
    }  
  
    public String getName() { return name; }  
    public int getScore() { return score; }  
    public Sex getSex() { return sex; }  
    public City getCity() { return city; }  
}
```

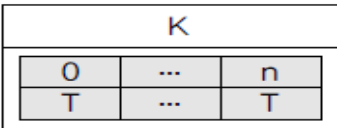
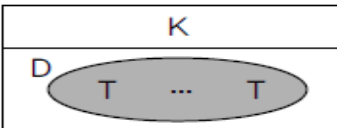
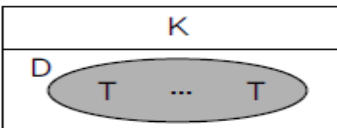
수집

```
public class ToListExample {
    public static void main(String[] args) {
        List<Student3> totalList = Arrays.asList(
            new Student3("홍길동", 10, Student3.Sex.MALE),
            new Student3("김수애", 6, Student3.Sex.FEMALE),
            new Student3("신용권", 10, Student3.Sex.MALE),
            new Student3("박수미", 6, Student3.Sex.FEMALE)
        );
        //남학생들만 묶어 List 생성
        List<Student3> maleList = totalList.stream()
            .filter(s->s.getSex()==Student3.Sex.MALE)
            .collect(Collectors.toList());
        maleList.stream()
            .forEach(s -> System.out.println(s.getName()));
        System.out.println();
        //여학생들만 묶어 HashSet 생성
        Set<Student3> femaleSet = totalList.stream()
            .filter(s -> s.getSex() == Student3.Sex.FEMALE)
            .collect( Collectors.toCollection()->new HashSet<Student3>()) );
        femaleSet.stream()
            .forEach(s -> System.out.println(s.getName()));
    }
}
```

수집

❖ 요소를 그룹핑해서 수집

- collect ()를 호출시 Collectors의 groupingBy() 또는 groupingByConcurrent()가 리턴하는 Collector를 매개값 대입
 - 컬렉션의 요소들을 그룹핑해서 Map객체 생성

리턴 타입	Collectors의 정적 메소드	설명
Collector<T,?,Map<K,List<T>>>	groupingBy(Function<T, K> classifier)	T를 K로 매핑하고 K키에 저장된 List에 T를 저장한 Map 생성
Collector<T,?, ConcurrentMap<K,List<T>>>	groupingByConcurrent(Function<T,K> classifier)	
Collector<T,?,Map<K,D>>	groupingBy(Function<T, K> classifier, Collector<T,A,D> collector)	T를 K로 매핑하고 K키에 저장된 D객체에 T를 누적한 Map 생성
Collector<T,?, ConcurrentMap<K,D>>	groupingByConcurrent(Function<T,K> classifier, Collector<T,A,D> collector)	
Collector<T,?,Map<K,D>>	groupingBy(Function<T,K> classifier, Supplier<Map<K,D>> mapFactory, Collector<T,A,D> collector)	T를 K로 매핑하고 Supplier가 제공하는 Map에서 K키에 저장된 D객체에 T를 누적
Collector<T,?, ConcurrentMap<K,D>>	groupingByConcurrent(Function<T,K> classifier, Supplier<ConcurrentMap<K,D>> mapFactory, Collector<T,A,D> collector)	

❖ 그룹핑 후 매핑 및 집계

■ Collectors.groupingBy() 메소드

- 그룹핑 후, 매핑이나 집계(평균, 카운팅, 연결, 최대, 최소, 합계)를 할 수 있도록 두 번째 매개값으로 Collector를 가질 수 있는 특성

리턴 타입	메소드(매개 변수)	설명
Collector<T,?,R>	mapping(Function<T, U> mapper, Collector<U,A,R> collector)	T를 U로 매핑한 후, U를 R에 수집
Collector<T,?,Double>	averagingDouble(ToDoubleFunction<T> mapper)	T를 Double로 매핑한 후, Double의 평균값을 산출
Collector<T,?,Long>	counting()	T의 카운팅 수를 산출
Collector <CharSequence,?,String>	joining(CharSequence delimiter)	CharSequence를 구분자 (delimiter)로 연결한 String을 산출
Collector<T,?,Optional<T>>	maxBy(Comparator<T> comparator)	Comparator를 이용해서 최대 T를 산출
Collector<T,?,Optional<T>>	minBy(Comparator<T> comparator)	Comparator를 이용해서 최소 T를 산출
Collector<T,?,Integer>	summingInt(ToIntFunction) summingLong(ToLongFunction) summingDouble(ToDoubleFunction)	Int, Long, Double 타입의 합계 산출

수집

```
public class GroupingByExample {
    public static void main(String[] args) {
        List<Student3> totalList = Arrays.asList(
            new Student3("홍길동", 10, Student3.Sex.MALE, Student3.City.Seoul),
            new Student3("김수애", 6, Student3.Sex.FEMALE, Student3.City.Pusan),
            new Student3("신용권", 10, Student3.Sex.MALE, Student3.City.Pusan),
            new Student3("박수미", 6, Student3.Sex.FEMALE, Student3.City.Seoul) );
        Map<Student3.Sex, List<Student3>> mapBySex = totalList.stream()
            .collect(Collectors.groupingBy(Student3::getSex));
        System.out.print("[남학생] ");
        mapBySex.get(Student3.Sex.MALE).stream()
            .forEach(s->System.out.print(s.getName() + " "));
        System.out.print("\n[여학생] ");
        mapBySex.get(Student3.Sex.FEMALE).stream()
            .forEach(s->System.out.print(s.getName() + " "));
        System.out.println();
        Map<Student3.City, List<String>> mapByCity = totalList.stream().collect(
            Collectors.groupingBy(Student3::getCity,
                Collectors.mapping(Student3::getName, Collectors.toList()) ) );
        System.out.print("\n[서울] ");
        mapByCity.get(Student3.City.Seoul).stream().forEach(s->System.out.print(s + " "));
        System.out.print("\n[부산] ");
        mapByCity.get(Student3.City.Pusan).stream().forEach(s->System.out.print(s + " "));
    }
}
```