

16장 스트림과 병렬 처리

16.1 스트림 소개

- 스트림은 자바 8부터 추가된 컬렉션(배열 포함)의 저장 요소를 하나씩 참조해서 람다식으로 처리할 수 있도록 해주는 반복자이다.

16.1.1 반복자 스트림

- `List<String>` 컬렉션에서 요소를 순차적으로 처리하기 위해 `Iterator` 반복자를 사용한다.

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

- 이 코드를 `Stream`을 사용해서 변경하면 다음과 같다.

```
List<String> list = Arrays.asList("홍길동", "신용권", "김자바");
Stream<String> stream = list.stream();
stream.forEach( name -> System.out.println(name) );
//void forEach(Consumer<T> action);
```

16.1.2 스트림의 특징

(1) 람다식으로 요소 처리 코드를 제공한다.

- `Stream`이 제공하는 대부분의 요소 처리 메소드는 함수적 인터페이스 매개 타입을 가지기 때문에 람다식 또는 메소드 참조를 이용해서 요소 처리 내용을 매개값으로 전달할 수 있다.

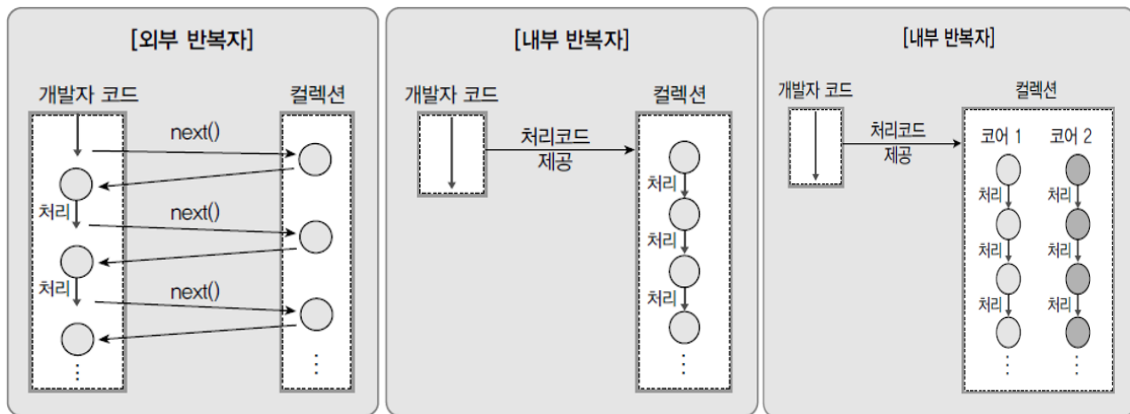
[LambdaExpressionsExample.java] 요소 처리를 위한 람다식

```
01 package sec01.stream_introduction;
02
03 import java.util.Arrays;
04 import java.util.List;
05 import java.util.stream.Stream;
06
07 public class LambdaExpressionsExample {
08     public static void main(String[] args) {
09         List<Student> list = Arrays.asList(
10             new Student("홍길동", 90),
11             new Student("신용권", 92)
12         );
13
14         Stream<Student> stream = list.stream(); // 스트림 얻기
15         stream.forEach(s -> { // List 컬렉션에서 Student를 가져와 람다식의 매개값으로 제공
16             String name = s.getName();
17             int score = s.getScore();
18             System.out.println(name + "-" + score);
19         });
```

```
20     }
21 }
```

(2) 내부 반복자를 사용하므로 병렬 처리가 쉽다.

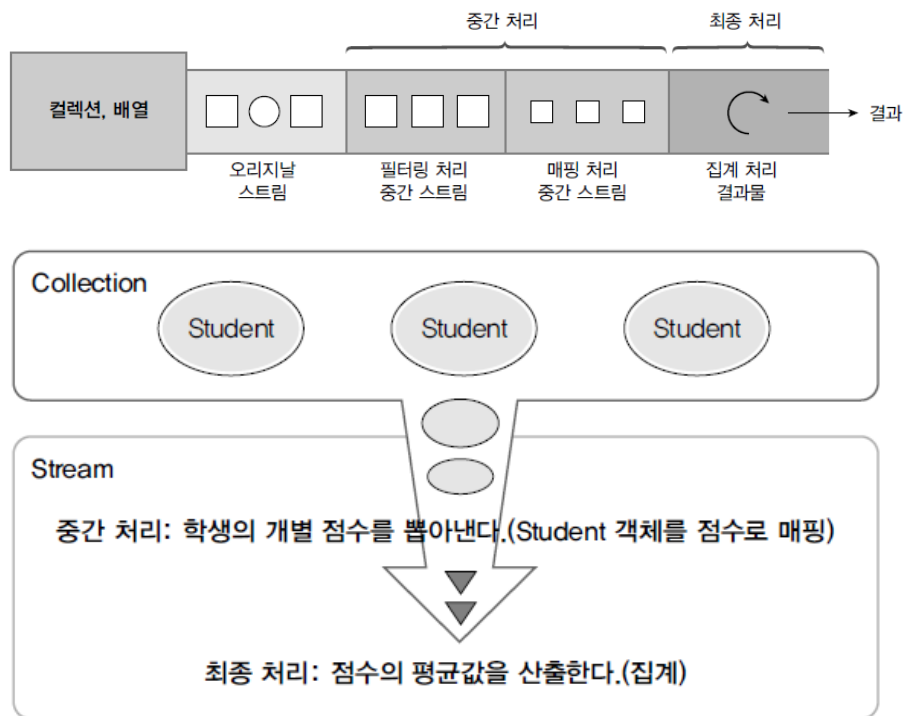
- 컬렉션 내부에서 요소들 반복 시킴 -> 내부 반복자(internal iterator)
- 내부 반복자의 이점
 - 개발자는 요소 처리 코드에만 집중할 수 있다.
 - 내부 반복자는 요소들의 반복 순서를 변경하거나, 멀티 코어 CPU를 최대한 활용하기 위해 요소들을 분배시켜 병렬 작업을 할 수 있게 한다.



[ParallelExample.java] 병렬 처리

```
01 package sec01.stream_introduction;
02
03 import java.util.Arrays;
04 import java.util.List;
05 import java.util.stream.Stream;
06
07 public class ParallelExample {
08     public static void main(String[] args) {
09         List<String> list = Arrays.asList("홍길동", "신용권", "감자바", "람다식", "박병렬");
10
11         //순차 처리
12         Stream<String> stream = list.stream();
13         stream.forEach(ParallelExample :: print);
14
15         System.out.println();
16
17         //병렬 처리
18         Stream<String> parallelStream = list.parallelStream();
19         parallelStream.forEach(ParallelExample :: print);
20     }
21
22     public static void print(String str) {
23         System.out.println(str+ " : " + Thread.currentThread().getName());
24     }
25 }
26
```

(3) 스트림은 중간 처리와 최종 처리를 할 수 있다.



[MapAndReduceExample.java] 중간 처리와 최종 처리

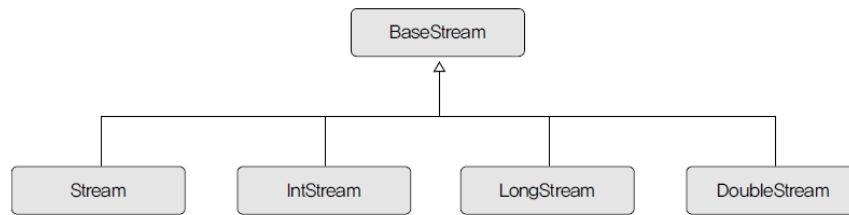
```

01 package sec01.stream_introduction;
02
03 import java.util.Arrays;
04 import java.util.List;
05
06 public class MapAndReduceExample {
07     public static void main(String[] args) {
08         List<Student> studentList = Arrays.asList(
09             new Student("홍길동", 10),
10             new Student("신용권", 20),
11             new Student("유미선", 30)
12         );
13
14         double avg = studentList.stream()
15             //중간처리(학생 객체를 점수로 매핑)
16             .mapToInt(Student :: getScore)
17             //최종 처리(평균 점수)
18             .average()
19             .getAsDouble();
20
21         System.out.println("평균 점수: " + avg);
22     }
23 }

```

16.2 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 : 각각 기본 타입인 int, long, double 요소 처리
- 스트림 인터페이스의 구현 객체는 주로 컬렉션과 배열에서 얻는다.

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

16.2.1 컬렉션으로부터 스트림 얻기

- 다음 예제는 List<Student> 컬렉션에서 Stream<Student>를 얻어내고 요소를 콘솔에 출력한다.

[FromCollectionExample.java] 컬렉션으로부터 스트림 얻기

```

package sec02.stream_kind;

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("신용권", 20),
            new Student("유미선", 30)
        );

        Stream<Student> stream = studentList.stream();
        stream.forEach(s -> System.out.println(s.getName()));
    }
}
    
```

```
}
}
```

[Student.java] 학생 클래스

```
package sec02.stream_kind;

public 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;
    }
}
```

16.2.2 배열로부터 스트림 얻기

- 다음 예제는 String[]과 int[] 배열로부터 스트림을 얻어내고 콘솔에 출력한다.

[FromArrayExample.java] 배열로부터 스트림 얻기

```
package sec02.stream_kind;

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", "신용권", "김미나" };
        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();
    }
}
```

16.2.3 숫자 범위로부터 스트림 얻기

- IntStream의 rangeClosed() 메소드는 첫 번째 매개값에서부터 두 번째 매개값까지 순차적으로 제공하는 IntStream을 리턴한다.

[FromIntRangeExample.java] 정수 범위를 소스로 하는 스트림

```
package sec02.stream_kind;

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);
    }
}
```

16.2.4 파일로부터 스트림 얻기

- 다음 예제는 Files의 정적 메소드인 lines()와 BufferedReader의 lines() 메소드를 이용하여 문자 파일의 내용을 스트림을 통해 행 단위로 읽고 콘솔에 출력한다.

[FromFileContentExample.java] 파일 내용을 소스로 하는 스트림

```
package sec02.stream_kind;

import java.io.BufferedReader;
import java.io.File;
import java.io.FileReader;
import java.io.IOException;
import java.nio.charset.Charset;
import java.nio.file.Files;
import java.nio.file.Path;
import java.nio.file.Paths;
import java.util.stream.Stream;

public class FromFileContentExample {
    public static void main(String[] args) throws IOException {
        Path path = Paths.get("src/sec02/stream_kind/linedata.txt");
        Stream<String> stream;

        // Files.lines() 메소드 이용
        stream = Files.lines(path, Charset.defaultCharset());
        stream.forEach(System.out::println);
        stream.close();
        System.out.println();

        // BufferedReader의 lines() 메소드 이용
        File file = path.toFile();
        FileReader fileReader = new FileReader(file);
        BufferedReader br = new BufferedReader(fileReader);
        stream = br.lines();
        stream.forEach(System.out::println);
        stream.close();
    }
}
```

16.2.5 디렉토리로부터 스트림 얻기

- 다음 예제는 Files의 정적 메소드인 list()를 이용해서 디렉토리의 내용을 스트림을 통해 읽고 콘솔에 출력한다.

[FromDirectoryExample.java] 디렉토리 내용을 소스로 하는 스트림

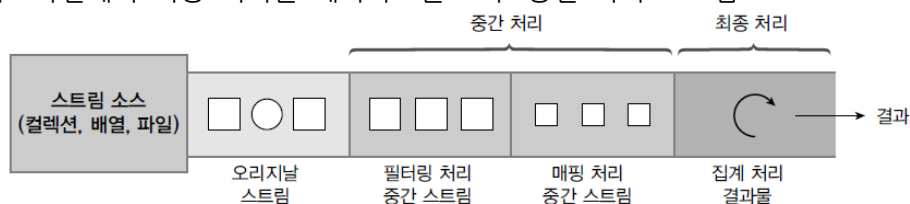
```
package sec02.stream_kind;

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 path = Paths.get("C:/temp/");
        Stream<Path> stream = Files.list(path);
        stream.forEach(p -> System.out.println(p.getFileName()));
    }
}
```

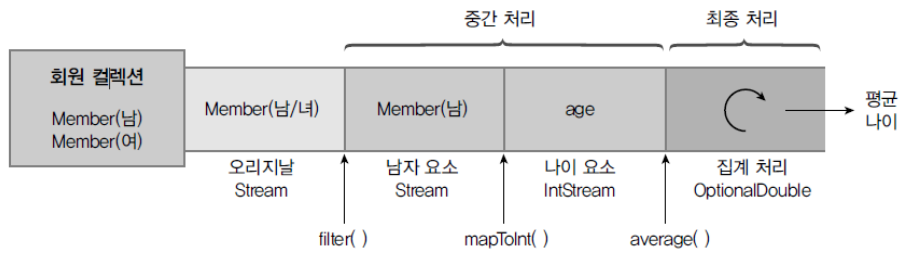
16.3 스트림 파이프라인

- 리덕션(Reduction)
 - 대량의 데이터를 가공해 축소하는 것
 - 리덕션의 결과물 : 데이터의 합계, 평균값, 카운팅, 최대값, 최소값 등
 - 컬렉션의 요소를 리덕션의 결과물로 바로 집계할 수 없을 경우에는?
 - 집계하기 좋도록 필터링, 매핑, 정렬, 그룹핑 등의 중간 처리 필요 → 스트림 파이프 라인의 필요성
- 파이프라인(Pipelines)
 - 여러 개의 스트림이 연결되어 있는 구조
 - 파이프라인에서 최종 처리를 제외하고는 모두 중간 처리 스트림



16.3.1 중간 처리와 최종 처리

- 최종 처리가 시작되기 전까지 중간 처리는 지연된다. 최종 처리가 시작되면 비로소 컬렉션의 요소가 하나씩 중간 스트림에서 처리되고 최종 처리까지 오게된다.
- Stream 인터페이스에는 필터링, 매핑, 정렬 등의 많은 중간 처리 메소드가 있는데, 이 메소드들은 중간 처리된 스트림을 리턴한다. 그리고 이 스트림에서 다시 중간 처리 메소드를 호출해서 파이프라인을 형성하게 된다.
- 예) 회원들 중 남자 회원들의 나이 평균 구하기



```
Stream<Member> maleFemaleStream = list.stream();
Stream<Member> maleStream = maleFemaleStream.filter(m -> m.getSex()==Member.MALE);
IntStream ageStream = maleStream.mapToInt(Member::getAge);
OptionalDouble optionalDouble = ageStream.average();
double ageAvg = optionalDouble.getAsDouble();

// 로컬 변수를 생략하고 연결하면 다음과 같은 형태의 파이프라인 코드만 남는다.
double ageAvg = list.stream()
    .filter(m -> m.getSex()==Member.MALE)
    .mapToInt(Member::getAge)
    .average()
    .getAsDouble();
```

[StreamPipelinesExample.java] 스트림 파이프라인

```
package sec03.stream_pipelines;

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

public class StreamPipelinesExample {
    public static void main(String[] args) {
        List<Member> list = Arrays.asList(
            new Member("홍길동", Member.MALE, 30),
            new Member("김나리", Member.FEMALE, 20),
            new Member("신용권", Member.MALE, 45),
            new Member("박수미", Member.FEMALE, 27)
        );

        double ageAvg = list.stream()
            .filter(m -> m.getSex()==Member.MALE)
            .mapToInt(Member::getAge)
            .average()
            .getAsDouble();

        System.out.println("남자 평균 나이: " + ageAvg);
    }
}
```

16.3.2 중간 처리 메소드와 최종 처리 메소드

- 리턴 타입이 Stream이라면 중간 처리 메소드이다.

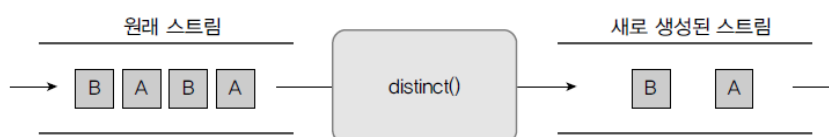
종류		리턴 타입	메소드(매개 변수)	소속된 인터페이스
중간 처리	필터링	Stream IntStream LongStream DoubleStream	distinct()	공통
	매핑		filter(...)	공통
			flatMap(...)	공통
			flatMapToDouble(...)	Stream
			flatMapToInt(...)	Stream
			flatMapToLong(...)	Stream
			map(...)	공통
			mapToDouble(...)	Stream, IntStream, LongStream
			mapToInt(...)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

- 리턴 타입이 기본 타입이거나 OptionalXXX라면 최종 처리 메소드이다.

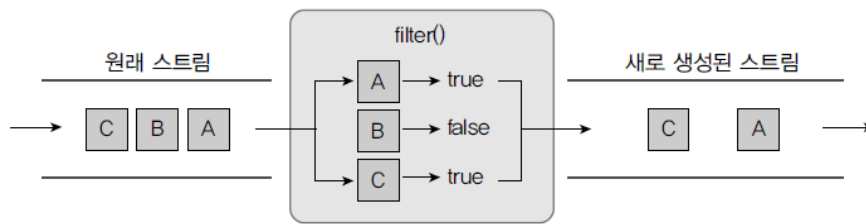
종류		리턴 타입	메소드(매개 변수)	소속된 인터페이스
최종 처리	매칭	boolean	allMatch(...)	공통
		boolean	anyMatch(...)	공통
		boolean	noneMatch(...)	공통
	집계	long	count()	공통
		OptionalXXX	findFirst()	공통
		OptionalXXX	max(...)	공통
		OptionalXXX	min(...)	공통
		OptionalDouble	average()	IntStream, LongStream, DoubleStream
		OptionalXXX	reduce(...)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

16.4 필터링(distinct(), filter())

- 필터링은 중간 처리 기능으로 요소 걸러내는 역할을 한다.
- 필터링 메소드인 distinct()와 filter()메소드는 모든 스트림이 가지고 있는 공통 메소드이다.
- distinct() 메소드: 중복을 제거하는 기능



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



[FilteringExample.java] 필터링

```

01 package sec04.stream_filtering;
02
03 import java.util.Arrays;
04 import java.util.List;
05
06 public class FilteringExample {
07     public static void main(String[] args) {
08         List<String> names = Arrays.asList("홍길동", "신용권", "감자바", "신용권", "신민철");
09
10         names.stream()
11             .distinct()
12             .forEach(n -> System.out.println(n));
13         System.out.println();
14
15         names.stream()
16             .filter(n -> n.startsWith("신"))
17             .forEach(n -> System.out.println(n));
18         System.out.println();
19
20         names.stream()
21             .distinct()
22             .filter(n -> n.startsWith("신"))
23             .forEach(n -> System.out.println(n));
24     }
25 }
26

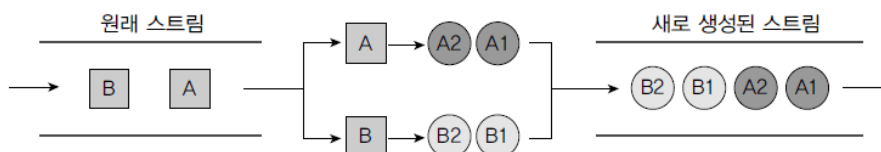
```

16.5 매핑(flatMapXXX(), mapXXX(), asXXXStream(), boxed())

- 매핑(mapping)은 중간 처리 기능으로 스트림의 요소를 다른 요소로 대체하는 작업을 말한다.

16.5.1 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

[FlatMapExample.java] 필터링

```
package sec05.stream_mapping;

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

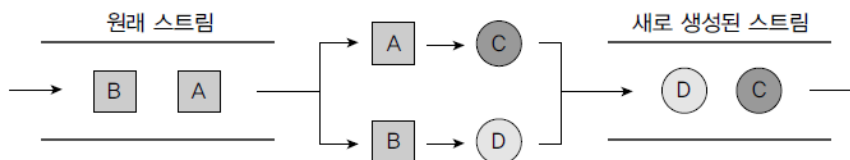
public class FlatMapExample {
    public static void main(String[] args) {
        List<String> inputList1 = Arrays.asList("java8 lambda", "stream mapping");
        inputList1.stream()
            .flatMap(data -> Arrays.stream(data.split(" ")))
            .forEach(word -> System.out.println(word));

        System.out.println();

        List<String> inputList2 = Arrays.asList("10, 20, 30", "40, 50, 60");
        inputList2.stream()
            .flatMapToInt(data -> {
                String[] strArr = data.split(",");
                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);
            })
            .forEach(number -> System.out.println(number));
    }
}
```

16.5.2 mapXXX() 메소드

- 요소를 대체하는 요소로 구성된 새로운 스트림 리턴



- mapXXX() 메소드의 종류

리턴 타입	메소드(매개 변수)	요소 → 대체 요소
Stream<R>	map(Function<T, R>)	T → R
DoubleStream	mapToDouble(ToDoubleFunction<T>)	T → double
IntStream	mapToInt(ToIntFunction<T>)	T → int
LongStream	mapToLong(ToLongFunction<T>)	T → long
DoubleStream	map(DoubleUnaryOperator)	double → double
IntStream	mapToInt(DoubleToIntFunction)	double → int
LongStream	mapToLong(DoubleToLongFunction)	double → long
Stream<U>	mapToObj(DoubleFunction<U>)	double → U
IntStream	map(IntUnaryOperator mapper)	int → int
DoubleStream	mapToDouble(IntToDoubleFunction)	int → double
LongStream	mapToLong(IntToLongFunction mapper)	int → long
Stream<U>	mapToObj(IntFunction<U>)	int → U
LongStream	map(LongUnaryOperator)	long → long
DobleStream	mapToDouble(LongToDoubleFunction)	long → double
IntStream	mapToLong(LongToIntFunction)	long → Int
Stream<U>	mapToObj(LongFunction<U>)	long → U

- 다음 예제는 학생 List에서 학생의 점수를 요소로 하는 새로운 스트림을 생성하고, 점수를 순차적으로 콘솔에 출력한다.

[MapExample.java] 다른 요소로 대체

```

01 package sec05.stream_mapping;
02
03 import java.util.Arrays;
04 import java.util.List;
05
06 public class MapExample {
07     public static void main(String[] args) {
08         List<Student> studentList = Arrays.asList(
09             new Student("홍길동", 10),
10             new Student("신용권", 20),
11             new Student("유미선", 30)
12         );
13
14         studentList.stream()
15             .mapToInt(Student :: getScore)
16             .forEach(score -> System.out.println(score));
17     }
18 }

```

[Student.java] 학생 클래스

```

01 package sec05.stream_mapping;
02
03 public class Student {
04     private String name;
05     private int score;
06
07     public Student(String name, int score) {
08         this.name = name;
09         this.score = score;
10     }

```

```

11
12     public String getName() { return name; }
13     public int getScore() { return score; }
14 }

```

16.5.3 asDoubleStream(), asLongStream(), boxed() 메소드

- asDoubleStream() 메소드는 intStream의 int 요소 또는 LongStream의 long 요소를 double 요소로 타입 변환해서 DoubleStream을 생성한다.

리턴 타입	메소드(매개 변수)	설명
DoubleStream	asDoubleStream()	int → double long → double
LongStream	asLongStream()	int → long
Stream<Integer> Stream<Long> Stream<Double>	boxed()	int → Integer long → Long double → Double

[AsDoubleStreamAndBoxedExample.java] 다른 요소로 대체

```

package sec05.stream_mapping;

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

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

        IntStream intStream = Arrays.stream(intArray);
        intStream
            .asDoubleStream() // DoubleStream 생성
            .forEach(d -> System.out.println(d));

        System.out.println();

        intStream = Arrays.stream(intArray);
        intStream
            .boxed() // Stream<Integer> 생성
            .forEach(obj -> System.out.println(obj.intValue()));
    }
}

```

16.6 정렬(sorted())

- 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

16.7 루핑(peek(), forEach())

- 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

16.8 매칭

- 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

16.9 기본 집계

16.9.1 스트림이 제공하는 기본 집계

- 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

16.9.2 Optional 클래스

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

[표] 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

16.10 커스텀 집계(reduce())

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

인터페이스	리턴 타입	메소드(매개 변수)
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)

16.11 수집(collect())

- 요소들을 필터링 또는 매핑 한 후 요소들을 수집하는 최종 처리 메소드인 collect()
- 이 메소드를 이용하면 필요한 요소만 컬렉션으로 담을 수 있고, 요소들을 그룹핑 한 후 집계 (리덕션) 할 수 있다.

16.11.1 필터링한 요소 수집

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

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

- 매개값인 Collector는 어떤 요소를 어떤 컬렉션에 수집할 것인지를 결정한다. 풀어서 해석하면 T(요소)를 A(누적기)가 R(컬렉션)에 저장한다는 의미이다.
 - 리턴 타입을 보면 A가 ?로 되어 있는데, 이것은 R에 T를 저장하는 방법을 알고 있어 A가 필요 없기 때문이다.

■ Collectors 클래스의 다양한 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

```
// 다음 코드는 전체 학생 중에서 남학생들만 필터링해서 별도의 List로 생성한다.
Stream<Student> totalStream = totalList.stream();
Stream<Student> maleStream = totalStream.filter(s->s.getSex()==Student.Sex.MALE);
Collector<Student, ?, List<Student>> collector = Collectors.toList();
List<Student> maleList = maleStream.collect(collector);

// 상기 코드에서 변수를 생략하면 다음과 같이 간단하게 작성할 수 있다.
List<Student> maleList = totalList.stream()
    .filter(s->s.getSex() == Student.Sex.MALE)
    .collect(Collectors.toList());

// 다음 코드는 전체 학생 중에서 여학생들만 필터링해서 별도의 HashSet으로 생성한다.
Set<Student> femaleList = totalList.stream()
    .filter(s->s.getSex() == Student.Sex.FEMALE)
    .collect(Collectors.toCollection(HashSet::new));
```

[ToListExample.java] 필터링해서 새로운 컬렉션 생성

```
package sec11.stream_collect;

import java.util.Arrays;
import java.util.HashSet;
import java.util.List;
import java.util.Set;
import java.util.stream.Collectors;

public class ToListExample {
    public static void main(String[] args) {
        List<Student> totalList = Arrays.asList(
            new Student("홍길동", 10, Student.Sex.MALE),
            new Student("김수애", 6, Student.Sex.FEMALE),
            new Student("신용권", 10, Student.Sex.MALE),
            new Student("박수미", 6, Student.Sex.FEMALE)
        );

        //남학생들만 묶어 List 생성
        List<Student> maleList = totalList.stream()
            .filter(s->s.getSex()==Student.Sex.MALE)
            .collect(Collectors.toList());

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

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

```
//여학생들만 묶어 HashSet 생성
Set<Student> femaleSet = totalList.stream()
    .filter(s -> s.getSex() == Student.Sex.FEMALE)
    .collect(Collectors.toCollection(HashSet :: new));
//.collect( Collectors.toCollection(()->{return new
HashSet<Student>();});) );
//.collect( Collectors.toCollection(()->new HashSet<Student>()) );
femaleSet.stream()
    .forEach(s -> System.out.println(s.getName()));
}
}
```

16.11.2 사용자 정의 컨테이너에 수집하기

- 스트림은 요소들을 필터링, 또는 매핑해서 컬렉션이 아닌 사용자 정의 컨테이너 객체에 수집할 수 있도록 다음과 같은 collect() 메소드를 추가적으로 제공한다.

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

```
Stream<Student> totalStream = totalList.stream();
Stream<Student> maleStream = totalStream.filter(s -> s.getSex() == Student.Sex.MALE);

Supplier<MaleStudent> supplier = () -> new MaleStudent();
BiConsumer<MaleStudent, Student> accumulator = (ms, s) -> ms.accumulate(s);
BiConsumer<MaleStudent, MaleStudent> combiner = (ms1, ms2) -> ms1.combine(ms2);

// supplier가 제공하는 MaleStudent에 accumulator가 Student를 수집해서 최종 처리된 MaleStudent를 얻는다.
// 싱글 스레드에서는 combiner는 사용되지 않는다.
MaleStudent maleStudent = maleStream.collect(supplier, accumulator, combiner);

// 상기 코드에서 변수를 생략하면 다음과 같이 간단하게 작성할 수 있다.
MaleStudent maleStudent = totalList.stream()
    .filter(s -> s.getSex() == Student.Sex.MALE)
    .collect(
        () -> new MaleStudent(),
        (r,t) -> r.accumulate(t),
        (r1,r2) -> r1.combine(r2)
    );

// 람다식을 메소드 참조로 변경하면 다음과 같이 더욱 간단하게 작성할 수 있다.
MaleStudent maleStudent = totalList.stream()
    .filter(s -> s.getSex() == Student.Sex.MALE)
    .collect(MaleStudent :: new, MaleStudent :: accumulate, MaleStudent :: combine);
```

[MaleStudent.java] 남학생이 저장되는 컨테이너

```
package sec11.stream_collect;

import java.util.ArrayList;
import java.util.List;

public class MaleStudent {
```

```
private List<Student> list;

public MaleStudent() {
    list = new ArrayList<Student>();
    System.out.println("[ " + Thread.currentThread().getName() + " ] MaleStudent()");
}

public void accumulate(Student student) { // 요소를 수집하는 메소드
    list.add(student);
    System.out.println("[ " + Thread.currentThread().getName() + " ] accumulate()");
}

public void combine(MaleStudent other) { // 두 MaleStudent를 결합하는 메소드 (병렬처리 시에만 호출)
    list.addAll(other.getList());
    System.out.println("[ " + Thread.currentThread().getName() + " ] combine()");
}

public List<Student> getList() {
    return list;
}
}
```

[MaleStudentExample.java] 남학생을 MaleStudent에 누적

```
package sec11.stream_collect;

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

public class MaleStudentExample {
    public static void main(String[] args) {
        List<Student> totalList = Arrays.asList(
            new Student("홍길동", 10, Student.Sex.MALE),
            new Student("김수애", 6, Student.Sex.FEMALE),
            new Student("신용권", 10, Student.Sex.MALE),
            new Student("박수미", 6, Student.Sex.FEMALE)
        );

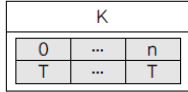
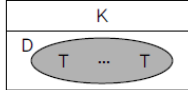
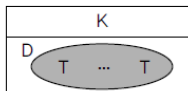
        MaleStudent maleStudent = totalList.stream()
            .filter(s -> s.getSex() == Student.Sex.MALE)
            .collect(MaleStudent :: new, MaleStudent :: accumulate, MaleStudent ::
combine);

        // .collect(()->new MaleStudent(), (r, t)->r.accumulate(t), (r1, r2)->r1.combine(r2));

        maleStudent.getList().stream()
            .forEach(s -> System.out.println(s.getName()));
    }
}
```

16.11.3 요소를 그룹핑해서 수집

- 컬렉션의 요소들을 그룹핑해서 Map객체를 생성한다.
- collect()를 호출시 Collectors의 groupingBy() 또는 groupingByConcurrent()가 리턴하는 Collector를 매개값으로 대입하면 된다.

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)	

- 다음 코드는 학생들을 거주 도시별로 그룹핑하고 나서, 같은 그룹에 속하는 학생들의 이름 List를 생성한 후, 거주 도시를 키로, 이름 List를 값으로 갖는 Map을 생성한다.

```
Stream<Student> totalStream = totalList.stream();

Function<Student, Student.City> classifier = Student::getCity;

Function<Student,String> mapper = Student::getName;
Collector<String, ?, List<String>> collector1 = Collectors.toList();
Collector<Student, ?, List<String>> collector2 = Collectors.mapping(mapper, collector1);

Collector<Student, ?, Map<Student.City, List<String>>> collector3 =
    Collectors.groupingBy(classifier, collector2);

Map<Student.City, List<String>> mapByCity = totalStream.collect( collector3 );

// 상기 코드에서 변수를 생략하면 다음과 같이 간단하게 작성할 수 있다.
Map<Student.City, List<String>> mapByCity = totalList.stream()
    .collect(
        Collectors.groupingBy(
            Student::getCity,
            Collectors.mapping(Student::getName, Collectors.toList())
        )
    );
```

[GroupingByExample.java] 성별로 그룹핑하기

```
package sec11.stream_collect;

import java.util.Arrays;
import java.util.List;
import java.util.Map;
import java.util.stream.Collectors;
```

```

public class GroupingByExample {
    public static void main(String[] args) {
        List<Student> totalList = Arrays.asList(new Student("홍길동", 10, Student.Sex.MALE,
Student.City.Seoul),
                                                new Student("김수애", 6, Student.Sex.FEMALE, Student.City.Pusan),
                                                new Student("신용권", 10, Student.Sex.MALE, Student.City.Pusan),
                                                new Student("박수미", 6, Student.Sex.FEMALE, Student.City.Seoul));

        Map<Student.Sex, List<Student>> mapBySex =
totalList.stream().collect(Collectors.groupingBy(Student::getSex));
        System.out.print("[남학생] ");
        mapBySex.get(Student.Sex.MALE).stream().forEach(s -> System.out.print(s.getName() + " "));
        System.out.print("\n[여학생] ");
        mapBySex.get(Student.Sex.FEMALE).stream().forEach(s -> System.out.print(s.getName() + "
"));

        System.out.println();

        Map<Student.City, List<String>> mapByCity = totalList.stream().collect(
Collectors.groupingBy(Student::getCity,
Collectors.mapping(Student::getName, Collectors.toList())));
        System.out.print("\n[서울] ");
        mapByCity.get(Student.City.Seoul).stream().forEach(s -> System.out.print(s + " "));
        System.out.print("\n[부산] ");
        mapByCity.get(Student.City.Pusan).stream().forEach(s -> System.out.print(s + " "));
    }
}

```

16.11.4 그룹핑 후 매핑 및 집계

- `Collectors.groupingBy()` 메소드는 그룹핑 후, 매핑이나 집계(평균, 카운팅, 연결, 최대, 최소, 합계)를 할 수 있도록 두 번째 매개값으로 `Collector`를 가질 수 있다.
- 다양한 `Collector`를 리턴하는 메소드

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

[GroupingAndReductionExample.java] 그룹핑 후 리덕션

```

package sec11.stream_collect;

import java.util.Arrays;

```

```
import java.util.List;
import java.util.Map;
import java.util.function.Function;
import java.util.function.ToDoubleFunction;
import java.util.stream.Collectors;
import java.util.stream.Collectors;
import java.util.stream.Stream;

public class GroupingAndReductionExample {
    public static void main(String[] args) {
        List<Student> totalList = Arrays.asList(
            new Student("홍길동", 10, Student.Sex.MALE),
            new Student("김수애", 12, Student.Sex.FEMALE),
            new Student("신용권", 10, Student.Sex.MALE),
            new Student("박수미", 12, Student.Sex.FEMALE)
        );

        //성별로 평균 점수를 저장하는 Map 얻기
        Stream<Student> totalStream = totalList.stream();
        Function<Student, Student.Sex> classifier = Student :: getSex;
        ToDoubleFunction<Student> mapper = Student :: getScore;
        Collector<Student, ?, Double> collector1 = Collectors.averagingDouble(mapper);
        Collector<Student, ?, Map<Student.Sex, Double>> collector2 =
Collectors.groupingBy(classifier, collector1);
        Map<Student.Sex, Double> mapBySex = totalStream.collect(collector2);

        /*Map<Student.Sex, Double> mapBySex = totalList.stream()
            .collect(
                Collectors.groupingBy(
                    Student :: getSex,
                    Collectors.averagingDouble(Student :: getScore)
                )
            );*/

        System.out.println("남학생 평균 점수: " + mapBySex.get(Student.Sex.MALE));
        System.out.println("여학생 평균 점수: " + mapBySex.get(Student.Sex.FEMALE));

        //성별로 심표로 구분된 이름을 저장하는 Map 얻기
        Map<Student.Sex, String> mapByName = totalList.stream()
            .collect(
                Collectors.groupingBy(
                    Student :: getSex,
                    Collectors.mapping(
                        Student :: getName,
                        Collectors.joining(",")
                    )
                )
            );
        System.out.println("남학생 전체 이름: " + mapByName.get(Student.Sex.MALE));
        System.out.println("여학생 전체 이름: " + mapByName.get(Student.Sex.FEMALE));
    }
}

package sec11.stream_collect;

import java.util.Arrays;
import java.util.List;
import java.util.Map;
import java.util.function.Function;
import java.util.function.ToDoubleFunction;
import java.util.stream.Collectors;
import java.util.stream.Collectors;
import java.util.stream.Stream;

public class GroupingAndReductionExample {
```

```

public static void main(String[] args) {
    List<Student> totalList = Arrays.asList(new Student("홍길동", 10, Student.Sex.MALE),
                                             new Student("김수애", 12, Student.Sex.FEMALE), new Student("신용권",
10, Student.Sex.MALE),
                                             new Student("박수미", 12, Student.Sex.FEMALE));

    // 성별로 평균 점수를 저장하는 Map 얻기
    Stream<Student> totalStream = totalList.stream();
    Function<Student, Student.Sex> classifier = Student::getSex;
    ToDoubleFunction<Student> mapper = Student::getScore;
    Collector<Student, ?, Double> collector1 = Collectors.averagingDouble(mapper);
    Collector<Student, ?, Map<Student.Sex, Double>> collector2 =
Collectors.groupingBy(classifier, collector1);
    Map<Student.Sex, Double> mapBySex = totalStream.collect(collector2);

    /*
    * Map<Student.Sex, Double> mapBySex = totalList.stream().collect(
    * Collectors.groupingBy( Student :: getSex,
    * Collectors.averagingDouble(Student :: getScore) ));
    */

    System.out.println("남학생 평균 점수: " + mapBySex.get(Student.Sex.MALE));
    System.out.println("여학생 평균 점수: " + mapBySex.get(Student.Sex.FEMALE));

    // 성별로 심표로 구분된 이름을 저장하는 Map 얻기
    Map<Student.Sex, String> mapByName = totalList.stream().collect(
        Collectors.groupingBy(Student::getSex,
Collectors.mapping(Student::getName, Collectors.joining(", ")))));
    System.out.println("남학생 전체 이름: " + mapByName.get(Student.Sex.MALE));
    System.out.println("여학생 전체 이름: " + mapByName.get(Student.Sex.FEMALE));
}
}

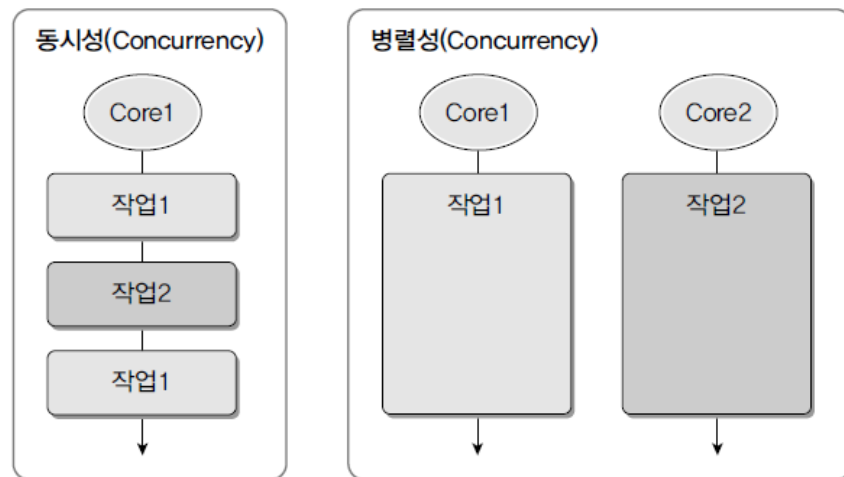
```

16.12 병렬 처리

- 멀티 코어 CPU 환경에서 쓰임
- 하나의 작업을 분할해서 각각의 코어가 병렬적 처리하는 것
- 병렬 처리의 목적은 작업 처리 시간을 줄이기 위한 것
- 자바 8부터 요소를 병렬 처리할 수 있도록 하기 위해 병렬 스트림 제공

16.12.1 동시성(Concurrency)과 병렬성(Parallelism)

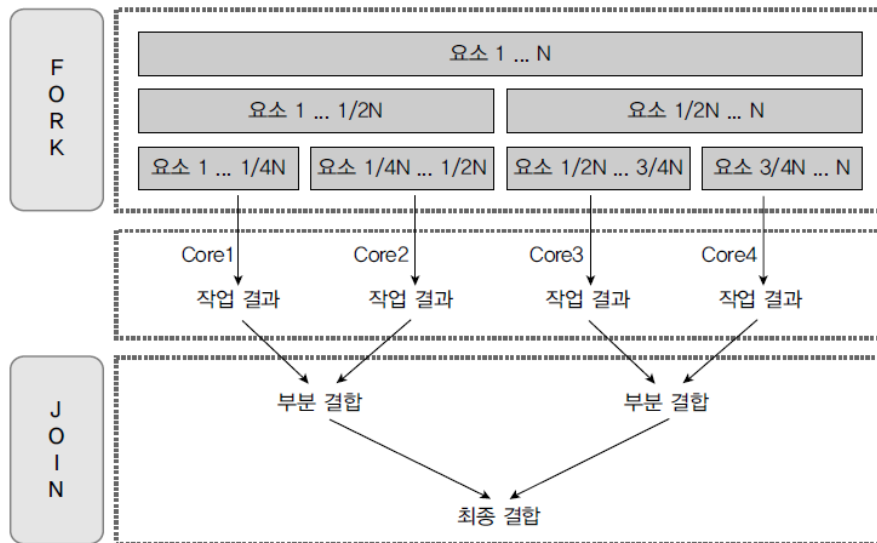
- 동시성 : 멀티 작업을 위해 멀티 스레드가 번갈아 가며 실행하는 성질로 싱글 코어 CPU를 이용한 멀티 작업을 수행한다. 병렬적으로 실행되는 것처럼 보이나 실체는 번갈아 가며 실행하는 동시성 작업이다.
- 병렬성 : 멀티 작업 위해 멀티 코어 이용해서 동시에 실행하는 성질
- 동시성과 병렬성의 비교



- 병렬성의 종류
- 데이터 병렬성
 - 전체 데이터를 쪼개어 서버 데이터들로 만든 뒤 병렬 처리해 작업을 빨리 끝내는 것
 - 자바 8에서 지원하는 병렬 스트림은 데이터 병렬성을 구현한 것
 - 멀티 코어의 수만큼 대용량 요소를 서버 요소들로 나누고
 - 각각의 서버 요소들을 분리된 스레드에서 병렬 처리
 - 예) 쿼드 코어(Quad Core) CPU일 경우 4개의 서버 요소들로 나누고, 4개의 스레드가 각각의 서버 요소들을 병렬 처리
- 작업 병렬성
 - 작업 병렬성은 서로 다른 작업을 병렬 처리하는 것
 - 예) 웹 서버 (Web Server) : 각각의 브라우저에서 요청한 내용을 개별 스레드에서 병렬로 처리

16.12.2 포크조인(ForkJoin) 프레임워크

- 런타임 시 포크조인 프레임워크 동작
- 포크 단계에서는 전체 데이터를 서버 데이터로 분리
- 서버 데이터를 멀티 코어에서 병렬로 처리
- 조인 단계에서는 서버 결과를 결합해서 최종 결과 도출
- 포크 조인 프레임 워크의 원리: ForkJoinPool 사용해 작업 스레드 관리



16.12.3 병렬 스트림 생성

- 코드에서 포크조인 프레임워크 사용해도 병렬처리 가능
- 병렬 스트림 이용할 경우 백그라운드에서 포크조인 프레임 워크 동작
 - 매우 쉽게 구현해 사용 가능
- 병렬 스트림을 얻는 메소드
 - `parallelStream()` 메소드 : 컬렉션으로부터 병렬 스트림을 바로 리턴
 - `parallel()` 메소드 : 순차 처리 스트림을 병렬 처리 스트림으로 변환해서 리턴

인터페이스	리턴 타입	메소드(매개 변수)
<code>java.util.Collection</code>	Stream	<code>parallelStream()</code>
<code>java.util.Stream.Stream</code>	Stream	<code>parallel()</code>
<code>java.util.Stream.IntStream</code>	IntStream	
<code>java.util.Stream.LongStream</code>	LongStream	
<code>java.util.Stream.DoubleStream</code>	DoubleStream	

16.12.4 병렬 처리 성능

- 병렬 처리에 영향을 미치는 3가지 요인
 - 요소의 수와 요소당 처리 시간: 요소 수가 적고 요소당 처리 시간 짧으면 순차 처리가 빠름
 - 스트림 소스의 종류: ArrayList, 배열은 인덱스로 요소 관리 → 병렬처리가 빠름
 - 코어(Core)의 수: 싱글 코어의 경우 순차 처리가 빠름

[과제] 확인문제

1. 스트림에 대한 설명으로 틀린 것은 무엇입니까?

- (1) 스트림은 내부 반복자를 사용하기 때문에 코드가 간결해진다.
- (2) 스트림은 요소를 분리해서 병렬 처리시킬 수 있다.

- (3) 스트림은 람다식을 사용해서 요소 처리 내용을 기술한다.
- (4) 스트림은 요소를 모두 처리하고 나서 처음부터 요소를 다시 반복시킬 수 있다.

2. 스트림을 얻을 수 있는 소스가 아닌 것은 무엇입니까?

- (1) 컬렉션(List)
- (2) int, long, double 범위
- (3) 디렉토리
- (4) 배열

3. 스트림 파이프라인에 대한 설명으로 틀린 것은 무엇입니까?

- (1) 스트림을 연결해서 중간 처리와 최종 처리를 할 수 있다.
- (2) 중간 처리 단계에서는 필터링, 매핑, 정렬, 그룹핑을 한다.
- (3) 최종 처리 단계에서는 합계, 평균, 카운팅, 최대값, 최소값 등을 얻을 수 있다.
- (4) 최종 처리가 시작되기 전에 중간 처리 단계부터 시작시킬 수 있다.

4. 스트림 병렬 처리에 대한 설명으로 틀린 것은 무엇입니까?

- (1) 멀티 코어의 수에 따라 요소를 분배하고 스레드를 생성시킨다.
- (2) 내부적으로 포크조인 프레임워크를 이용한다.
- (3) 병렬 처리는 순차적 처리보다 항상 빠른 처리를 한다.
- (4) 내부적으로 스레드풀을 이용해서 스레드를 관리한다.

5. List에 저장되어 있는 String 요소에서 대소문자와 상관없이 "java"라는 단어가 포함된 문자열만 필터링해서 출력하려고 합니다. 빈칸에 알맞은 코드를 작성해 보세요.

[StreamExample.java]

```
01 package verify.exam05;
02
03 import java.util.Arrays;
04 import java.util.List;
05
06 public class StreamExample {
07     public static void main(String[] args) {
08         List<String> list = Arrays.asList(
09             "This is a java book",
10             "Lambda Expressions",
11             "Java8 supports lambda expressions"
12         );
13         list.stream()
14             _____ #1
15             .forEach(a -> System.out.println(a));
16     }
17 }
18
19 // 실행 결과
20 // This is a java book
21 // Java8 supports lambda expressions
```

6. List에 저장되어 있는 Member의 평균 나이를 출력하려고 합니다. 빈칸에 알맞은 코드를 작성해 보세요.

[StreamExample.java]

```

01 package verify.exam06;
02
03 import java.util.Arrays;
04 import java.util.List;
05
06 public class StreamExample {
07     public static void main(String[] args) {
08         List<Member> list = Arrays.asList(
09             new Member("홍길동", 30),
10             new Member("신용권", 40),
11             new Member("감자바", 26)
12         );
13
14         double avg = list.stream()
15             // #1 작성 위치
16
17
18
19
20         System.out.println("평균 나이: " + avg);
21     }
22
23     static class Member {
24         private String name;
25         private int age;
26
27         public Member(String name, int age) {
28             this.name = name;
29             this.age = age;
30         }
31
32         public String getName() { return name; }
33         public int getAge() { return age; }
34     }
35 }

```

7. List에 저장되어 있는 Member 중에서 직업이 "개발자"인 사람만 별도의 List에 수집하려고 합니다. 빈칸에 알맞은 코드를 작성해 보세요.

[StreamExample.java]

```

01 package verify.exam07;
02
03 import java.util.Arrays;
04 import java.util.List;
05 import java.util.stream.Collectors;
06
07 public class StreamExample {
08     public static void main(String[] args) {
09         List<Member> list = Arrays.asList(
10             new Member("홍길동", "개발자"),
11             new Member("김나리", "디자이너"),
12             new Member("신용권", "개발자")
13         );

```

```

14
15         List<Member> developers = list.stream()
16             // #1 작성 위치
17
18
19         developers.stream().forEach(m -> System.out.println(m.getName()));
20     }
21
22     static class Member {
23         private String name;
24         private String job;
25
26
27         public Member(String name, String job) {
28             this.name = name;
29             this.job = job;
30         }
31
32         public String getName() { return name; }
33         public String getJob() { return job; }
34     }
35 }
36
37 // 실행 결과
38 // 홍길동
39 // 신용권

```

8. List에 저장되어 있는 Member를 직업별로 그룹핑해서 Map<String,List<String>> 객체로 생성하려고 합니다. 키는 Member의 직업이고, 값은 Member의 이름으로 구성된 List<String>입니다. 빈칸에 알맞은 코드를 작성해보세요.

[StreamExample.java]

```

01 package verify.exam08;
02
03 import java.util.Arrays;
04 import java.util.List;
05 import java.util.Map;
06 import java.util.stream.Collectors;
07
08 public class StreamExample {
09     public static void main(String[] args) {
10         List<Member> list = Arrays.asList(
11             new Member("홍길동", "개발자"),
12             new Member("김나리", "디자이너"),
13             new Member("신용권", "개발자")
14         );
15
16         Map<String, List<String>> groupingMap = list.stream()
17             // #1 작성 위치
18
19
20
21
22
23
24
25         System.out.print("[개발자] ");
26         groupingMap.get("개발자").stream().forEach(s -> System.out.print(s + " "));
27         System.out.print("\n[디자이너] ");
28         groupingMap.get("디자이너").stream().forEach(s -> System.out.print(s + " "));

```

```
29         }
30
31         static class Member {
32             private String name;
33             private String job;
34
35
36             public Member(String name, String job) {
37                 this.name = name;
38                 this.job = job;
39             }
40
41             public String getName() { return name; }
42             public String getJob() { return job; }
43         }
44     }
45
46     // 실행 결과
47     // [개발자] 홍길동 신용권
48     // [디자이너] 김나리
```