Chapter 16. 스트림과 병렬 처리

16.1 스트리 소개

: 스트림(Stream)은 컬렉션(배열 포함)의 저장 요소를 하나씩 참조해서 람다식(함수적 스타일)으로 처리할 수 있도록 해주는 반복자이다.

16.1.1 반복자 스트림

- **Iterator 반복자**

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

- **Iterator -> Stream**

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

> forEach() 메소드
>
> : Consumer 함수적 인터페이스 타입의 매개값을 가지므로 컬렉션의 요소를 소비할 코드를 람다식으로 기술 가능.

```
void forEach(Consumer<T> action)
```

- **예제**

```
package iterator_stream;

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", "신용권", "감자바");

        // 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) );
    }
}
```

실행 결과

```
홍길동
신용권
감자바

홍길동
신용권
감자바
```

16.1.2 스트림의 특징

1. Iterator와 비슷한 역할을 하는 반복자이다.
2. 람다식으로 요소 처리 코드를 제공한다.
3. 내부 반복자를 사용하므로 병렬 처리가 쉽다.
4. 중간 처리와 최종 처리 작업을 수행한다.

람다식으로 요소 처리 코드를 제공한다.

: Stream이 제공하는 대부분의 요소 처리 메소드는 함수적 인터페이스 매개 타입을 가지기 때문.

- **예제**

  Student.java(**학생 클래스**)

```
package characteristic_of_stream;

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; }
}
```

LambdaExpressionsExample.java(**요소 처리를 위한 람다식**)

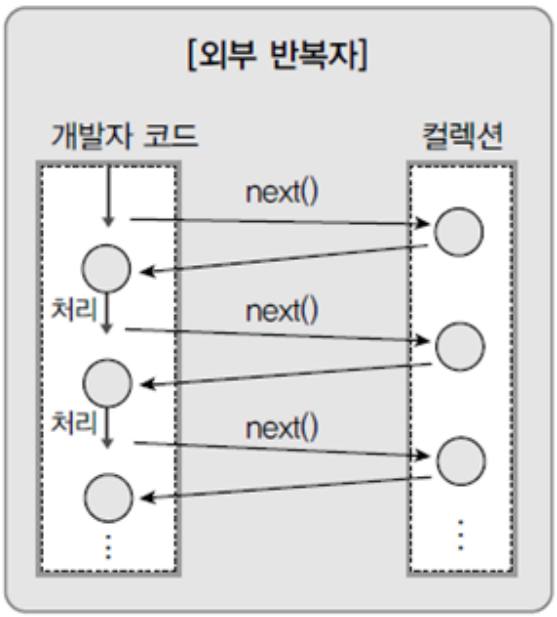
```
package characteristic_of_stream;

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

public class LambdaExpressionsExample {
    public static void main(String[] args) {
        List<Student> list = Arrays.asList(
                new Student("홍길동", 90),
                new Student("신용권", 92)
        );

        // 스트림 얻기
        Stream<Student> stream = list.stream();

        // List 컬렉션에서 Student 를 가져와
        // 람다식의 매개값으로 제공
        stream.forEach( s -> {
            String name = s.getName();
            int score = s.getScore();
            System.out.println(name + "-" + score);
        });
    }
}
```

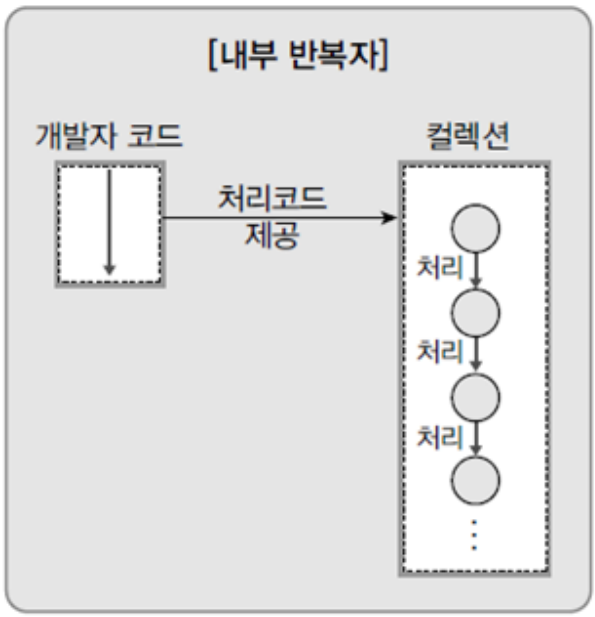
실행 결과

```
홍길동-90
신용권-92
```

내부 반복자를 사용하므로 병렬 처리가 쉽다.

- **외부 반복자(external iterator)** : 개발자가 코드로 직접 컬렉션의 요소를 반복해서 가져오는 코드 패턴(**ex) for, while**)
- **내부 반복자(internal iterator)** : 컬렉션 내부에서 요소들을 반복시키고, 개발자는 요소당 처리해야 할 코드만 제공하는 코드 패턴





> 내부 반복자를 사용해서 컬렉션 내부에서 어떻게 요소를 반복시킬 것인가는 컬렉션에 맡겨두고, 개발자는 요소 처리 코드에만 집중할 수 있다.

- **스트림(stream)** : Iterator 대신 스트림을 이용하면 코드도 간결해지고, 요소의 병렬 처리가 컬렉션 내부에서 처리된다.
  - **병렬(parallel) 처리** : 한 가지 작업을 서브 작업으로 나누고, 서브 작업들을 분리된 스레드에서 병렬적으로 처리하는 것.
  - **병렬 처리 스트림** : 런타임 시 하나의 작업을 서브 작업으로 자동으로 나누고, 서브 작업의 결과를 자동으로 결합해서 최종 결과물을 생성.
- **예제**

```
package characteristic_of_stream;

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

public class ParallelExample {
    public static void main(String[] args) {
        List<String> list = Arrays.asList(
                "홍길동", "신용권", "감자바",
                "람다식", "박병렬"
        );

        // 순차 처리
        Stream<String> stream = list.stream();
        stream.forEach(ParallelExample :: print);   // 메소드 참조
        System.out.println();

        // 병렬 처리
        Stream<String> parallelStream = list.parallelStream();
        parallelStream.forEach(ParallelExample::print); // 메소드 참조
    }
```

```
    public static void print(String str) {
        System.out.println(str + " : " + Thread.currentThread().getName());
    }
}
```

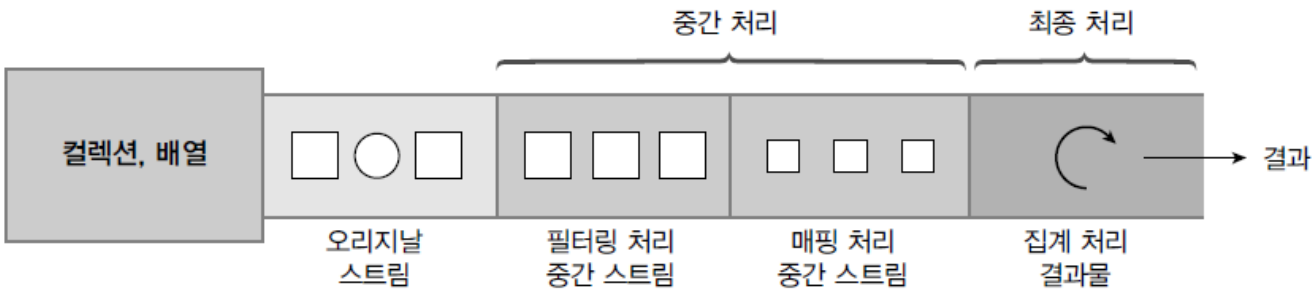
실행 결과

```
홍길동 : main          // 순차처리
신용권 : main
감자바 : main
람다식 : main
박병렬 : main

감자바 : main          // 병렬처리
신용권 : ForkJoinPool.commonPool-worker-9
박병렬 : ForkJoinPool.commonPool-worker-2
람다식 : ForkJoinPool.commonPool-worker-9
홍길동 : ForkJoinPool.commonPool-worker-11
```

> 병렬 처리 스트림은 main 스레드를 포함해서 ForkJoinPool(스레드풀)의 작업 스레드들이 병렬적으로 요소를 처리하는 것을 볼 수 있다.

스트림은 중간 처리와 최종 처리를 할 수 있다.

: 스트림은 컬렉션의 요소에 대해 중간 처리와 최종 처리를 수행할 수 있는데, 중간 처리에서는 매핑, 필터링, 정렬을 수행하고 최종 처리에서는 반복, 카운팅, 평균, 총합 등의 집계 처리를 수행한다.



- **예제**

```
package characteristic_of_stream;

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

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

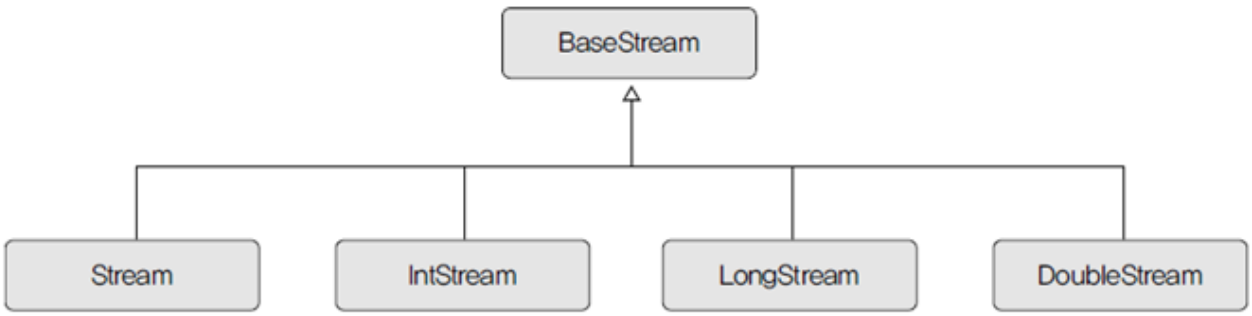
```
        double avg = studentList.stream()
                // 중간 처리(학생 객체를 점수로 매핑)
                .mapToInt(Student::getScore)
                // 최정 처리(평균 점수)
                .average()
                .getAsDouble();

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

실행 결과

```
평균 점수: 20.0
```

16.2 스트림의 종류

: java.util.stream 패키지에는 스트림(stream) API들이 있다. 패키지 내용을 보면 BaseStream 인터페이스를 부모로 해서 자식 인터페이스들이 상속 관계를 이루고 있다.



> Stream은 객체 요소를 처리하는 스트림이고, IntStream, LongStream, DoubleStream은 각각 기본 타입인 int, long, double 요소를 처리하는 스트림이다.

- **스트림 구현 객체**

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

- **예제(컬렉션으로부터 스트림 얻기)**

  Student.java

```java
package from_stream.to_collection;

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;
    }
}
```

FromCollectionExample.java

```
package from_stream.to_collection;

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()));
    }
}
```

실행 결과

```
홍길동
신용권
유미선
```

16.2.2 배열로부터 스트림 얻기

- **예제**

```
package get_stream.from_array;

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();
```

```
    }
}
```

실행 결과

```
홍길동,신용권,김미나,
1,2,3,4,5,
```

16.2.3 숫자 범위로부터 스트림 얻기

- **예제**

```
package get_stream.from_int_range;

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);
    }
}
```

실행 결과

```
총합: 5050
```

> **rangeClosed()** : 첫 번째 매개값에서부터 두 번째 매개값까지 순차적으로 제공하는 IntStream을 리턴.
>
> **range()** : 기능은 위와 같지만, 두 번째 매개값은 포함하지 않는다.

16.2.4 파일로부터 스트림 얻기

- **예제**

```
package get_stream.from_file;

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 {

        // 파일의 경로 정보를 가지고 있는 Path 객체 생성
        Path path = Paths.get("src/get_stream/from_file/linedata.txt");
        Stream<String> stream;

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

        // BufferedReader 의 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);
    }
}
```

실행 결과

```
a
bb
ccc
dddd
eeeee
ffffff
ggggggg

a
bb
ccc
dddd
eeeee
ffffff
ggggggg
```

16.2.5 디렉토리로부터 스트림 얻기

- **예제**

```
package get_stream.from_directory;
```

```java
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("src");
        Stream<Path> stream = Files.list(path);
        // p = 서브 디렉토리 또는 파일에 해당하는 Path 객체
        // p.getFileName() = 서브 디렉토리 이름 또는 파일 이름 리턴
        stream.forEach( p -> System.out.println(p.getFileName()));
    }
}
```

실행 결과

```
characteristic_of_stream
get_stream
iterator_stream
```

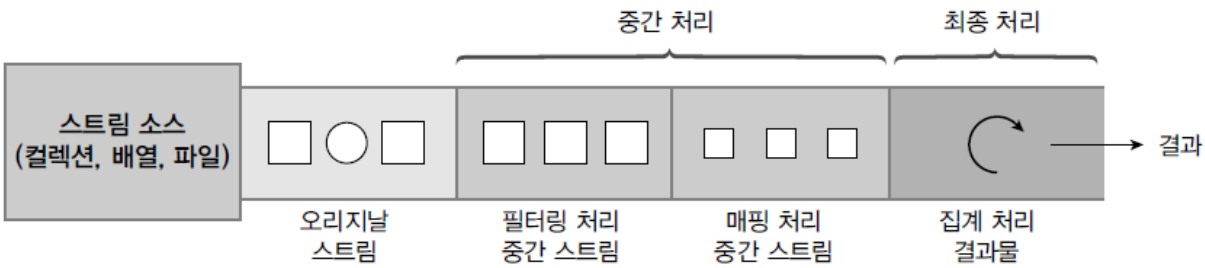
16.3 스트림 파이프라인

- **리덕션(Reduction)** : 대량의 데이터를 가공해서 축소하는 것. 예) 데이터의 합계, 평균값, 카운팅, 최대값 등

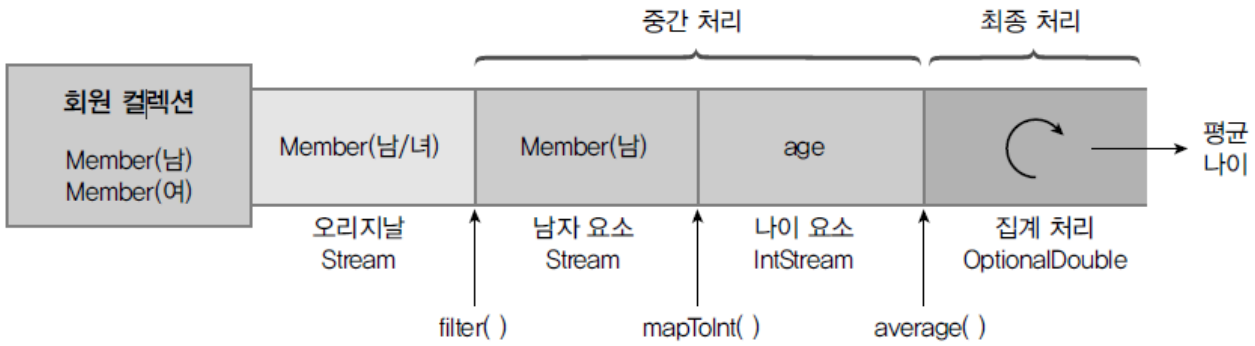
: 컬렉션의 요소를 리덕션의 결과물로 바로 집계할 수 없을 경우에는 집계하기 좋도록 스트림의 **중간 처리(필터링, 매핑, 정렬, 그룹핑)**가 필요하다.

16.3.1 중간 처리와 최종 처리

- **스트림**
  - **중간 처리** : 데이터의 필터링, 매핑, 정렬, 그룹핑 등
  - **최종 처리** : 합계, 평균, 카운팅, 최대값, 최소값 등
    - 위와 같은 처리들을 **파이프라인(pipelines)**으로 해결한다. 파이프라인은 여러 개의 스트림이 연결되어 있는 구조를 말한다.



- 예제



Member.java

```
package stream_pipelines;

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;
    }
}
```

StreamPipelinesExample.java

```
package 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);
    }
}
```

실행 결과

```
남자 평균 나이: 37.5
```

16.3.2 중간 처리 메소드와 최종 처리 메소드

: 리턴 타입이 스트림이라면 중간 처리 메소드이고, 기본 타입이거나 OptionalXXX 라면 최종 처리 메소드이다. 소속된 인터페이스에서 공통의 의미는 Stream, Instream, LongStream, DoubleStream에서 모두 제공된다는 뜻이다.

- **중간 처리 메소드**

종류		리턴 타입	메소드(매개 변수)	소속된 인터페이스
중간 처리	필터링	Stream IntStream LongStream DoubleStream	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

- **최종 처리 메소드**

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

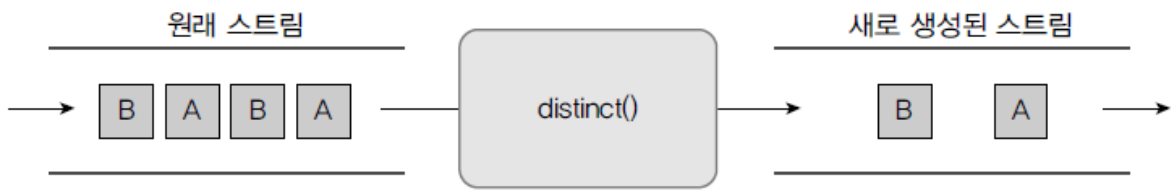
16.4 필터링(distinct(), filter())

: 필터링은 중간 처리 기능으로 요소를 걸러내는 역할을 한다.

- **필터링 메소드**

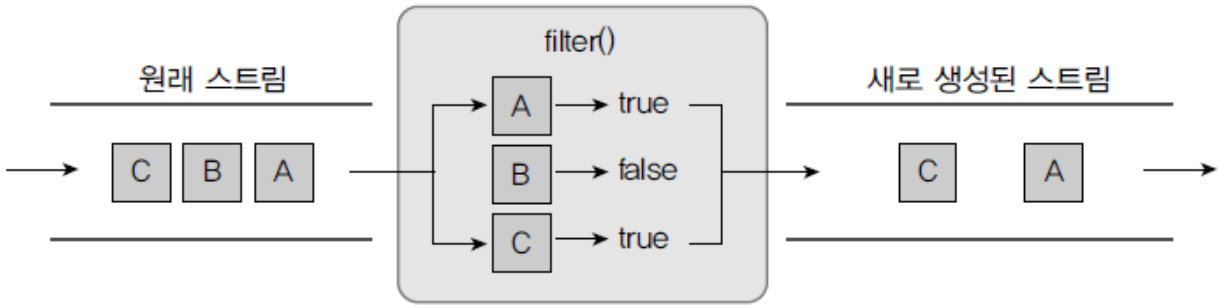
리턴 타입	메소드(매개 변수)	설명
Stream, IntStream LongStream DoubleStream	distinct()	중복 제거
``	filter(Predicate)	조건 필터링
| filter(IntPredicate) |		
| filter(LongPredicate) |		
| filter(DoublePredicate) |		

- **distinct() 메소드**



- **filter() 메소드**

  : 매개값으로 주어진 Predicate가 true를 리턴하는 요소만 필터링한다.



- **예제(성이 "신"인 이름만 필터링)**

```
package filtering;

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() // 중복제거
                .forEach(n -> System.out.println(n));
        System.out.println();

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

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

실행 결과

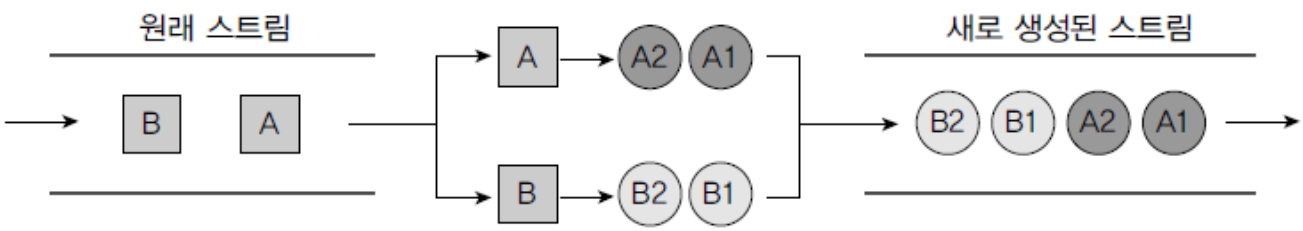
```
홍길동
신용권
감자바
신민철

신용권
신용권
신민철

신용권
신민철
```

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

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

16.5.1 flatMapXXX() 메소드

: 요소를 대체하는 복수 개의 요소들로 구성된 새로운 스트림을 리턴한다.



- **flatMapXXX() 메소드 종류**

리턴 타입	메소드(매개변수)	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

- **예제(컬렉션 요소별로 단어를 뽑아 단어 스트림으로 재생성)**

```
package mapping;
```

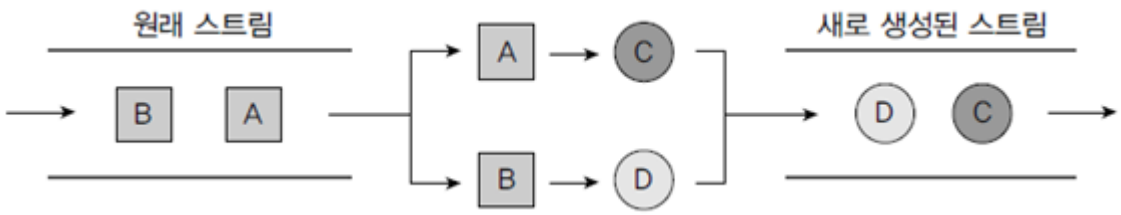
```
import java.util.Arrays;
import java.util.List;

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));
    }
}
```

실행 결과

```
java8
lambda
stream
mapping

10
20
30
40
50
60
```

16.5.2 mapXXX() 메소드

: 요소를 대체하는 요소로 구성된 새로운 스트림을 리턴한다.



- **mapXXX() 메소드 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	mapToInt(LongToIntFunction)	long -> Int
Stream<U>	mapToObj(LongFunction<U>)	long -> U

- **예제(학생의 점수를 요소로 새로운 스트림을 생성 후 순차적 출력)**

  Student.java

```java
package mapping;

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;
    }
}
```

MapExample.java

```
package mapping;

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));
    }
}
```

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

- **asDoubleStream() 메소드** : IntStream의 int 요소 또는 LongStream의 long 요소를 double 요소로 타입 변환해서 DoubleStream을 생성한다.
- **asLongStream() 메소드** : IntStream의 int 요소를 long 요소로 타입 변환해서 LongStream을 생성한다.
- **boxed() 메소드** : int, long, double 요소를 Integer, Long, Double 요소로 박싱해서 Stream 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

- **예제(int 배열로 부터 IntStream을 얻고 int 요소를 double 요소로 변환)**

```
package mapping;

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

```java
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()));
    }
}
```

실행 결과

```
1.0
2.0
3.0
4.0
5.0

1
2
3
4
5
```

16.6 정렬(sorted())

: 스트림은 요소가 최종 처리되기 전에 중간 단계에서 요소를 정렬해서 최종 처리 순서를 벼경할 수 있다.

- **정렬 메소드 종류**

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

> 객체 요소일 경우에는 클래스가 **Comparable을 구현하지 않으면** sorted() 메소드를 호출했을 때 ClassCastException이 발생한다.

- **예제(점수를 기준으로 학생 요소를 오름차순으로 정렬)**

  Student.java(**정렬 가능한 클래스**)

```java
package sorted;

// Comparable 구현 클래스
public class Student implements Comparable<Student> {
    private String name;
    private int score;

    public Student(String name, int score) {
        this.name = name;
        this.score = score;
    }

    public int getScore() {
        return score;
    }

    public String getName() {
        return name;
    }

    @Override
    public int compareTo(Student o) {
        // score < o.score : 음수 리턴
        // score == o.score : 0 리턴
        // score > o.score : 양수 리턴
        return Integer.compare(score, o.score);
    }
}
```

> Comparable을 구현한 상태에서 기본 비교(Comparable) 방법으로 정렬하고 싶다면 다음 세 가지 방법 중 하나를 선택해서 sorted()를 호출하면 된다.

```java
sorted();
sorted( (a,b) -> a.compareTo(b));
sorted( Comparator.naturalOrder() );
```

> 만약 객체 요소가 Comparable을 구현하고 있지만, 기본 비교 방법과 정반대로 정렬하고 싶다면 다음과 같이 sorted()를 호출하면 된다.

```java
sorted( (a,b) -> b.compareTo(a) );
sorted( Comparator.reverseOrder() );
```

> 객체 요소가 Comparable를 구현하지 않았다면 Comparator를 매개값으로 갖는 sorted() 메소드를 사용하면 된다.

```
// 중괄호 안에 a와 b를 비교해서 a가 작으면 음수, 같으면 0, a가 크면 양수를 리턴하는 코드 작성
sorted( (a,b) -> { ... } )
```

SortingExample.java(**점수를 기준으로 오름차순 정렬**)

```
package sorted;

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()   // 숫자를 오름차순으로 정렬
                .forEach(n -> System.out.print(n + ","));
        System.out.println();

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

        studentsList.stream()
                .sorted()   // 정수를 기준으로 오름차순으로 Student 정렬
                .forEach(s -> System.out.print(s.getScore() + ","));
        System.out.println();

        studentsList.stream()   // 정수를 기준으로 내림차순으로 Student 정렬
                .sorted(Comparator.reverseOrder())
                .forEach(s -> System.out.print(s.getScore() + ","));
    }
}
```

실행 결과

```
1,2,3,4,5,
10,20,30,
30,20,10,
```

16.7 루핑(peek(), forEach())

- **루핑(looping)** : 요소 전체를 반복하는 것.
  - **peek() 메소드** : 중간 처리 메소드, 중간 처리 단계에서 전체 요소를 루핑하면서 추가적인 작업을 하기 위해 사용한다.

    예시)

    ```
    intStream
        .filter( a -> a%2==0 )
        .peek( a -> System.out.println(a) )
        .sum()
    ```

    > 이처럼 필터과 요소 확인 후 반드시 최종 처리 메소드가 호출되어야 한다.

  - **forEach() 메소드** : 최종 처리 메소드, 요소를 소비하는 최종 처리 메소드이므로 이후에 sum()과 같은 다른 최종 메소드를 호출하면 안 된다.
- **예제**

```
package looping;

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));  // 동작 X

        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))   // 동작 O
                .sum();                             // 최종 메소드
        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));   // 최종 메소드로 동작
    }
}
```

실행 결과

```
[peek()를 마지막에 호출한 경우]
[최종 처리 메소드를 마지막에 호출한 경우]
2
4
총합: 6
[forEach()를 마지막에 호출한 경우]
2
4
```

16.8 매칭(allMatch(), anyMatch(), noneMatch())

: 스트림 클래스는 최종 처리 단계에서 요소들이 특정 조건에 만족하는지 조사할 수 있도록 세 가지 매칭 메소드를 제공하고 있다.

- **allMatch() 메소드** : 모든 요소들이 매개값으로 주어진 Predicate의 조건을 만족하는지 조사
- **anyMatch() 메소드** : 최소한 한 개의 요소가 매개값으로 주어진 Predicate의 조건을 만족하는지 조사
- **noneMatch() 메소드** : 모든 요소들이 매개값으로 주어진 Predicate의 조건을 만족하지 않는지 조사

리턴 타입	메소드(매개변수)	제공 인터페이스
boolean	allMatch(Predicate<T> predicate) anyMatch(Predicate<T> predicate) noneMatch(Predicate<T> predicate)	Stream
boolean	allMatch(IntPredicate<T> predicate) anyMatch(IntPredicate<T> predicate) noneMatch(IntPredicate<T> predicate)	IntStream
boolean	allMatch(LongPredicate<T> predicate) anyMatch(LongPredicate<T> predicate) noneMatch(LongPredicate<T> predicate)	LongStream
boolean	allMatch(DoublePredicate<T> predicate) anyMatch(DoublePredicate<T> predicate) noneMatch(DoublePredicate<T> predicate)	DoubleStream

- **예제(모든 요소가 2의 배수인지, 하나라도 3의 배수가 존재하는지, 모든 요소가 3의 배수가 아닌지 조사)**

```java
package matching;

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);
    }
}
```

실행 결과

```
모두 2의 배수인가? true
하나라도 3의 배수가 있는가? true
3의 배수가 없는가? false
```

16.9 기본 집계(sum(), count(), average(), max(), min())

- **집계(Aggregate)** : 최종 처리 기능으로 요소들을 처리해서 카운팅, 합계, 평균값, 최대값, 최소값 등과 같이 하나의 값으로 산출하는 것을 말한다. 집계는 대량의 데이터를 가공해서 축소하는 **리덕션(Reduction)**이라고 볼 수 있다.

16.9.1 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

- **OptionalXXX 리턴 타입** : Optional, OptionalDouble, OptionalInt, OptionalLong 클래스 타입을 말한다. 이들은 값을 저장하는 **값 기반 클래스(value-based class)들이다**. 이 객체에서 값을 얻기 위해서는 get(), getAsDouble(), getAsInt(), getAsLong()을 호출하면 된다.ㅋ
- **예제**

```
package basic_aggregation;

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();   // 요소 개수
        System.out.println("2의 배수 개수: " + count);

        long sum = Arrays.stream(new int[] {1, 2, 3, 4, 5})
                .filter(n -> n%2==0)
                .sum();     // 요소 총합
        System.out.println("2의 배수의 합: " + sum);

        double avg = Arrays.stream(new int[] {1, 2, 3, 4, 5})
                .filter(n -> n%2==0)
                .average()  // 요소 평균
                .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()      // 요소 최대값
                .getAsInt();
        System.out.println("최대값: " + max);

        int min = Arrays.stream(new int[] {1, 2, 3, 4, 5})
                .filter(n -> n%2==0)
                .min()      // 요소 최소값
                .getAsInt();
        System.out.println("최소값: " + min);

        int first = Arrays.stream(new int[] {1, 2, 3, 4, 5})
                .filter(n -> n%3==0)
                .findFirst()    // 첫 번재 요소
                .getAsInt();
        System.out.println("첫번째 3의 배수: " + first);
    }
}
```

실행 결과

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

16.9.2 Optional 클래스

: Optional, OptionalDouble, OptionalInt, OptionalLong 클래스들은 저장하는 값의 타입만 다를 뿐 제공하는 기능은 거의 동일하다. Optional 클래스는 집계 값이 존재하지 않을 경우 디폴트 값을 설정할 수도 있고, 집계 값을 처리하는 Consumer도 등록할 수 있다.

- **Optional 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

- **사용 예시**

```java
// 컬렉션의 요소가 추가되지 않아 저장된 요소가 없을 경우
List<Integer> list = new ArrayList<>();
double avg = list.stream()
    .mapToInt(Integer :: intValue)
    .average()
    .getAsDouble();
System.out.println("평균: " + avg);
```

> 요소가 없기 때문에 평균값도 있을 수 없다. 그래서 NoSuchElementException 예외 발생.

요소가 없을 경우 예외를 피하는 방법

1. Optional 객체를 얻어 isPresent() 메소드로 평균값 여부 확인

```
OptionalDouble optional = list.stream()
    .mapToInt(Integer :: intValue)
    .average();
if(optional.isPresent()) {
    System.out.println("평균: " + optional.getAsDouble());
} else {
    System.out.println("평균: 0.0");
}
```

2. orElse() 메소드로 디폴트 값을 정해 놓는다.

```
double avg = list.stream()
    .mapToInt(Integer :: intValue)
    .average()
    .orElse(0.0);
System.out.println("평균: " + avg);
```

3. ifPresent() 메소드로 평균값이 있을 경우에만 값을 이용하는 람다식을 실행

```
list.stream()
    .mapToInt(Integer :: intValue)
    .average()
    .ifPresent(a -> System.out.println("평균: " + a));
```

- **예제**

```
package basic_aggregation;

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

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();
        */

        OptionalDouble optional = list.stream()
                .mapToInt(Integer :: intValue)
                .average();
        if(optional.isPresent()) {
            System.out.println("방법1_평균: " + optional.getAsDouble());
        } else {
            System.out.println("방법1_평균: 0.0");
        }
```

```
        double avg = list.stream()
                .mapToInt(Integer :: intValue)
                .average()
                .orElse(0.0);
        System.out.println("방법2_평균: " + avg);

        list.stream()
                .mapToInt(Integer::intValue)
                .average()
                .ifPresent(a -> System.out.println("방법3_평균: " + a));
    }
}
```

실행 결과

```
방법1_평균: 0.0
방법2_평균: 0.0
```

16.10 커스텀 집계(reduce())

: 기본 집계를 프로그램화해서 다양한 집계 결과물을 만들 수 있도록 reduce() 메소드를 제공한다.

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)

> 스트림에 요소가 전혀 없을 경우 디폴트 값인 identity 매개값이 리턴된다. XXXOperator 매개값은 집계 처리를 위한 람다식을 대입한다.

```
int sum = studentList.stream()
    .map(Student :: getScore)
    .reduce(0, (a, b) -> a+b);
```

- **예제**

  Student.java

```java
package reduce_method;

public class Student {
    private String name;
    private int score;

    public Student(String name, int score) {
        this.name = name;
        this.score = score;
    }

    public int getScore() {
        return score;
    }

    public String getName() {
        return name;
    }
}
```

ReductionExmaple.java(**reduce() 메소드 이용**)

```java
package reduce_method;

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);
    }
}
```

실행 결과

```
sum1: 275
sum2: 275
sum3: 275
```

16.11 수집(collect())

: 스트림은 요소들을 필터링 또는 매핑한 후 요소들을 수집하는 최종 처리 메소드인 collect()를 제공하고 있다. 이 메소드를 이용하면 필요한 요소만 컬렉션으로 담을 수 있고, 요소들을 그룹핑한 후 집계(리덕션)할 수 있다.

16.11.1 필터링한 요소 수집

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

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

매개값인 Collector(수집기)는 어떤 요소를 어떤 컬렉션에 수집할 것인지를 결정한다. Collector의 타입 파라미터 T는 요소이고, A는 누적기(accumulator)이다. 그리고 R은 요소가 저장될 컬렉션이다.

- **Collector 클래스의 정적메소드**

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

> A(누적기)가 ?로 되어 있는데, 이것은 Collector가 R(컬렉션)에 T(요소)를 저장하는 방법을 알고 있어 A(누적기)가 필요 없기 때문이다.

  - **Map과 ConcurrentMap의 차이점**

    : Map은 멀티 스레드에 안전하지 않고, ConcurrentMap은 멀티 스레드에 안전하다.

- **예시**

  학생 중에서 남학생들만 필터링해서 별도의 List로 생성한다.

```
// 1. 전체 학생 List에서 Stream을 얻는다
Stream<Student> totalStream = totalList.stream();
// 2. 남학생만 필터링해서 Stream을 얻는다.
Stream<Student> maleStream = totalStream.filter(s->s.getSex()==Student.Sex.MALE);
// 3. List에 Student를 수집하는 Collector를 얻는다.
Collector<Student, ?, List<Student>> collector = Collectors.toList();
// 4. Stream에서 collect() 메소드로 Student를 수집해서 새로운 List를 얻는다.
List<Student> maleList = maleStream.collect(collector);

// 위의 과정을 간략화
List<Student> maleList = totalList.stream()
    .filters(s->s.getSex()==Student.Sex.MALE)
    .collect(Collectors.toList());
```

- **예제(필터링해서 새로운 컬렉션 생성)**

  Student.java

```
package collect_method;

public class Student {
    public enum Sex { MALE, FEEMALE }
    public enum City { Seoul, Pusan }

    private String name;
    private int score;
    private Sex sex;
    private City city;

    public Student(String name, int score, Sex sex) {
        this.name = name;
        this.score = score;
        this.sex = sex;
    }

    public Student(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 City getCity() {
        return city;
    }

    public Sex getSex() {
        return sex;
    }
}
```

ToListExmaple.java

```
package collect_method;

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.FEEMALE),
                new Student("신용권", 10, Student.Sex.MALE),
                new Student("박수미", 6, Student.Sex.FEEMALE)
        );

        // 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.FEEMALE)
                .collect(Collectors.toCollection(HashSet::new));
        femaleSet.stream()
                .forEach(s -> System.out.println(s.getName()));
    }
}
```

실행 결과

```
홍길동
신용권

김수애
박수미
```

16.11.2 사용자 정의 컨테이너에 수집하기

: 스트림은 요소들을 필터링, 또는 매핑해서 **사용자 정의 컨테이너(List, Set, Map과 다른 컬렉션)** 객체에 수집할 수 있도록 다음과 같이 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>)

- **매개변수**
  - **Supplier** : 요소들이 수질될 컨테이너 객체(R)를 생성하는 역할을 한다.
  - **XXXConsumer** : 컨테이너 객체(R)에 요소(T)를 수집하는 역할을 한다. 스트림에서 요소를 컨테이너에 수집할 때마다 실행.
  - **BiConsumer** : 컨테이너 객체(R)를 결합하는 역할을 한다. 순차 처리 스트림에서는 호출되지 않고, 병렬 처리 스트림에서만 호출되어 스레드별로 생성된 컨테이너 객체를 결합해서 최종 컨테이너 객체를 완성한다.
- **리턴타입**
  - **R** : 요소들이 최종 수집된 컨테이너 객체이다.
- **예제(학생들 중에서 남학생만 수집하는 MaleStudent 컨테이너)**

  MaleStudent.java(**남학생이 저장되는 컨테이너**)

```java
package collect_method;

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

public class MaleStudent {

    // 요소를 저장할 컬렉션
    private List<Student> list;

    public MaleStudent() {
        list = new ArrayList<Student>();
        // 생성자를 호출하는 스레드 이름 출력
        System.out.println("[" + Thread.currentThread().getName()
                + "] MaleStudent()");
    }
```

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

    // 두 MaleStudent 를 결합하는 메소드
    // (병렬 처리 시에만 호출)
    public void combine(MaleStudent other) {
        list.addAll(other.getList());
        // combine() 호출한 스레드 이름 출력
        System.out.println("[" + Thread.currentThread().getName()
        + "] combine()");
    }

    // 요소가 저장된 컬렉션을 리턴
    public List<Student> getList() {
        return list;
    }
}
```

MaleStudentExample.java(**남학생을 MaleStudent에 누적**)

```
package collect_method;

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)
        );

        // 전체 학생 List 에서 Stream 을 얻는다.
        MaleStudent maleStudent = totalList.stream()
                // 남학생만 필터링해서 Stream 을 얻는다.
                .filter(s -> s.getSex() == Student.Sex.MALE)
                // 1. MaleStudent 를 공급하는 Supplier 를 얻는다.
                // 2. MaleStudent 와 Student 를 매개값으로 받아서 MaleStudent 의
accumulate() 메소드로 Student 를 수집하는 BiConsumer 를 얻는다.
                // 3. 두 개의 MaleStudent 를 매개값으로 받아 combine() 메소드로 결합하는
BiConsumer 를 얻는다.
```

```
                // 4. supplier 가 제공하는 MaleStudent 에 accumulator 가 Student 를 수집해
서 최종 처리된 MaleStudent 를 얻는다.
                .collect(MaleStudent::new,
                        MaleStudent::accumulate,
                        MaleStudent::combine);

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

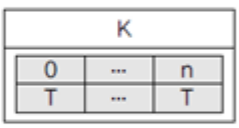
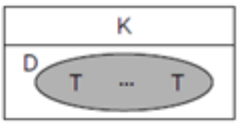
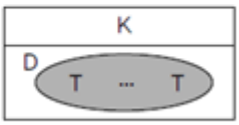
실행 결과

```
[main] MaleStudent()
[main] accumulate()
[main] accumulate()
홍길동
신용권
```

16.11.3 요소를 그룹핑해서 수집

: collect() 메소드는 단순히 요소를 수집하는 기능 이외에 컬렉션의 요소들을 그룹핑해서 Map 객체를 생성하는 기능도 제공한다.

- **컬렉션의 요소들을 그룹핑해서 Map 객체 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)	

- **예시**

1. 학생들을 성별로 그룹핑하고 나서, 같은 그룹에 속하는 학생 List를 생성한 후, 성별을 키로, 학생 List를 값으로 갖는 Map을 생성한다.

```
// 전체 학생 List에서 Stream을 얻는다.
Stream<Student> totalStream = totalList.stream();

// Student를 Student.Sex로 매핑하는 Function을 얻는다.
Function<Student, Student.Sex> classifier = Student :: getSex;

// Student.Sex가 키가 되고, 그룹핑된 List<Stuent>가 값인 Map을 생성하는 Collector를 얻는다.
Collector<Student, ?, Map<Student.Sex, List<Student>>> collector =
    Collectors.groupingBy(classifier);

// Stream의 collect() 메소드로 Student를 Student.Sex별로 그룹핑해서 Map을 얻는다.
Map<Student.Sex, List<Student>> mapBySex = totalStream.collect(collector);

// 간략화
Map<Student.Sex, List<Student>> mapBySex = totalList.stream()
    .collect(Collectors.groupingBy(Student :: getSex));
```

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

```
// 전체 학생 List에서 Stream을 얻는다.
Stream<Student> totalStream = totalList. stream();

// Student를 Student.City로 매핑하는 Function을 얻는다.
Function<Student, Student.City> classifier = Student::getCity;

// Student의 이름을 List에 수집하는 Collector를 얻는다.
// Student를 이름으로 매핑하는 Function을 얻는다.
Function<Student, String> mapper = Student::getName;
// 이름을 List에 수집하는 Collector를 얻는다.
Collector<String, ?, List<String>> collector1 = Collectors.toList();
// Collectors의 mapping() 메소드로 Student를 이름으로 매핑하고 이름을 List에 수집하는 Collector를 얻는다.
Collector<Student, ?, List<String>> collector2 =
    Collectors.mapping(mapper, collector1);

// Student.City가 키이고, 그룹핑된 이름 List가 값인 Map을 생성하는 Collector를 얻는다.
Collectors<Student, ?, Map<Student.City, List<String>>> collector3 =
    Collectors.groupingBy(classfier, collector2);

// Stream의 collect() 메소드로 Student를 Student.City별로 그룹핑해서 Map을 얻는다.
Map<Student.City, List<String>> mapByCity =
    totalStream.collect(collector3);

// 위의 과정 간략화
Map<Student.City, List<String>> mapByCity = totalList.stream()
    .collect(
```

```
        Collectors.groupingBy(
            Student::getCity,
            Collectors.mapping(Student::getName, Collectors.toList())
        )
);

// 위의 과정에 TreeMap 객체 생성 추가
Map<Student.City, List<String>> mapByCity = totalList.stream()
    .collect(
        Collectors.groupingBy(
            Student::getCity,
            TreeMap::new,   // 객체 생성
            Collectors.mapping(Student::getName, Collectors.toList())
        )
);
```

- **예제**

```
package collect_method;

import java.util.Arrays;
import java.util.List;
import java.util.Map;
import java.util.stream.Collector;
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 + " "));
    }
}
```

실행 결과

```
[남학생] 홍길동 신용권
[여학생] 김수애 박수미

[서울] 홍길동 박수미
[부산] 김수애 신용권
```

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 Collector를 가질 수 있다.

- **Collectors.groupingBy() 메소드**

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(ToIntFuncton) summingLong(ToLongFuncton) summingDouble(ToDoubleFuncton)	Int, Long, Double 타입의 합계 산출

- **예시**

  학생들을 성별로 그룹핑한 다음 같은 그룹에 속하는 학생들의 평균 점수를 구하고, 성별을 키로, 평균 점수를 값으로 갖는 Map을 생성한다.

```
// 전체 학생 List에서 Stream을 얻는다.
Stream<Student> totalStream = totalList.stream();

// Student를 Student.Sex로 매핑하는 Function을 얻는다.
Function<Student, Student.Sex> classifier = Student :: getSex;

// Student를 점수로 매핑하는 ToDoubleFunction을 얻는다.
ToDoubleFunction<Student> mapper = Student :: getScore;

// 학생 점수의 평균을 산출하는 Collector를 얻는다.
Collector<Student, ?, Double> collector1 =
    Collectors.averageingDouble(mapper);

// Student.Sex가 키이고, 평균 점수 Double이 값인 Map을 생성하는 Collector를 얻는다.
Collector<Student, ?, Map<Student.Sex, Double>> collector2 =
    Collectors.groupingBy(classifier, collector1);

// Stream의 collect() 메소드로 Student를 Student.Sex별로 그룹핑해서 Map을 얻는다.
Map<Student.Sex, Double> mapBySex = totalStream.collect(collector2);

// 위의 과정 간략화
Map<Student.Sex, Double> mapBySex = totalList.stream()
    .collect(
        Collectors.groupingBy(
            Student :: getSex,
```

```
            Collectors.averagingDouble(Student :: getScore)
        )
    );

// 학생들을 성별로 그룹핑한 다음 같은 그룹에 속하는 학생 이름을 쉼표로 구분해서 문자열로 만들고, 성별을 키로, 이름 문자열을 값으로 갖는 Map을 생성한다.
Map<Student.Sex, String> mapByName = totalList.stream()
    .collect(
        Collectors.groupingBy(
            Student :: getSex,
            Collectors.mapping(
                Student :: getName,
                Collectors.joining(",")
            )
        )
    );
```

- **예제(그룹핑 후 리덕션)**

```
package collect_method;

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

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

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

```
        // 성별을 쉼표로 구분한 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));
    }
}
```

실행 결과

```
남학생 평균 정수: 10.0
여학생 평균 점수: 12.0
남학생 전체 이름: 홍길동,신용권
여학생 전체 이름: 김수애,박수미
```

16.12 병렬 처리

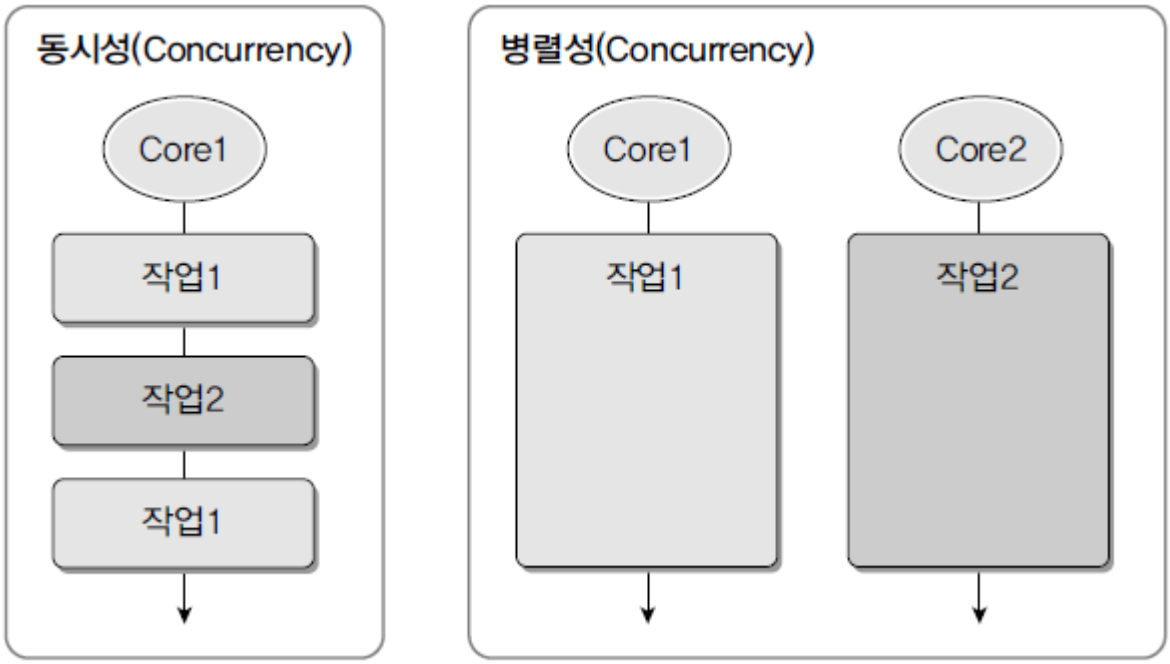
: 멀티 코어 CPU 환경에서 하나의 작업을 분할해서 각각의 코어가 병렬적으로 처리하는 것을 말하는데, 병렬 처리의 목적은 작업 처리 시간을 줄이기 위한 것이다.

16.12.1 동시성(Concurrency)과 병렬성(Parallelism)

- **동시성** : 멀티 작업을 위해 멀티 스레드가 번갈아가며 실행하는 성질
- **병렬성** : 멀티 작업을 위해 멀티 코어를 이용해서 동시에 실행하는 성질을 말한다.
  - **데이터 병렬성(Data parallelism)**

    : 전체 데이터를 쪼개어 서브 데이터들로 만들고 이 서브 데이터들을 병렬 처리해서 작업을 빨리 끝내는 것
  - **작업 병렬성(Task parallelism)**

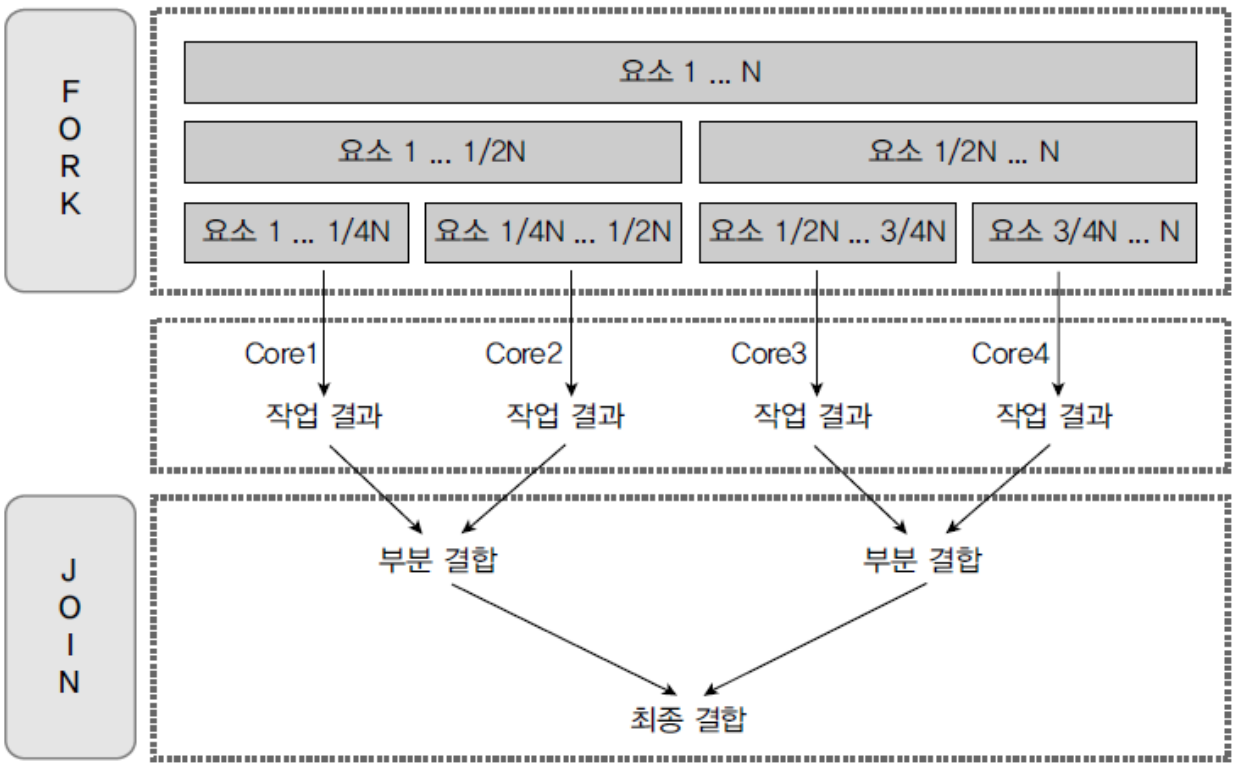
    : 서로 다른 작업을 병렬 처리(요청한 내용을 개별 스레드에서 병렬로 처리)하는 것.



16.12.2 포크조인(ForkJoin) 프레임워크

: 병렬 스트림이 요소들을 병렬 처리하기 위한 프레임 워크이다.

- 병렬 스트림을 이용하면 런타임 시에 포크조인 프레임워크 동작
- **포크 단계**에서는 전체 데이터를 서브 데이터로 분리한다. 그러고 나서 서브 데이터를 멀티 코어에서 병렬로 처리한다.
- **조인 단계**에서는 서브 결과를 결합해서 최종 결과를 만들어 낸다.



- 포크와 조인 기능 이외에 스레드풀인 ForkJoinPool을 제공한다.

- ExecutorSevice의 구현 객체인 ForkJoinPool을 사용해서 작업 스레드를 관리 한다.

16.12.3 병렬 스트림 생성

: 병렬 스트림을 이용할 경우에는 백그라운드에서 포크조인 프레임워크가 사용되기 때문에 개발자는 매우 쉽게 병렬 처리를 할 수 있다.

- **병렬 스트림 메소드**

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

> **parallelStream() 메소드** : 컬렉션으로부터 병렬 스트림을 바로 리턴
>
> **parallel() 메소드** : 순차 처리 스트림을 병렬 처리 스트림으로 변환해서 리턴

- **내부적인 동작 확인을 위한 사용자 정의 컨테이너**

> 순차 처리 스트림

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

> 병렬 처리 스트림으로 변경하면 다음과 같다.

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

- **예제**

```
package parallel_processing;

import collect_method.MaleStudent;
import collect_method.Student;
```

```
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", 10,
                        Student.Sex.FEMALE),
                new Student("신용권", 10,
                        Student.Sex.MALE),
                new Student("박수미", 10,
                        Student.Sex.FEMALE)
        );

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

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

실행 결과

```
[ForkJoinPool.commonPool-worker-3] MaleStudent()
[ForkJoinPool.commonPool-worker-5] MaleStudent()
[main] MaleStudent()
[ForkJoinPool.commonPool-worker-7] MaleStudent()
[ForkJoinPool.commonPool-worker-7] accumulate()
[main] accumulate()
[ForkJoinPool.commonPool-worker-7] combine()
[main] combine()
[main] combine()
홍길동
신용권
```

> main 스레드와 ForkJoinPool에서 3개의 스레드가 사용되어 총 4개의 스레드가 동작한다.

16.12.4 병렬 처리 성능

: 병렬 처리에 영향을 미치는 다음 3가지 요인을 잘 살펴보아야 한다.

요소의 수와 요소당 처리 시간

: 컬렉션에 요소의 수가 적고 처리 시간이 짧으면 순차 처리가 오히려 병렬 처리보다 빠를 수 있다.

스트림 소스의 종류

: HashSet, TreeSet은 요소 분리가 쉽지 않고, LinkedList 역시 링크를 따라가야 하므로 요소 분리가 쉽지 않다. 따라서 이 소스들은 ArrayList, 배열보다는 상대적으로 병렬 처리가 늦다.

코어(Core)의 수

: 코어의 수가 많으면 많을수록 병렬 작업 처리 속도는 빨라진다.

- **예제(순차 처리와 병렬 처리 성능 비교)**

```
package parallel_processing;

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

public class SequentialVsParallelExample {
    public static void work(int value) {
        try {
            // 값이 작을수록 순차 처리가 빠름
            Thread.sleep(100);
        } catch (InterruptedException e) { }
    }

    public static long testSequential(List<Integer> list) {
        long start = System.nanoTime();
        list.stream().forEach(a -> work(a));
        long end = System.nanoTime();
        long runTime = end - start;
        return runTime;
    }

    public static long testParallel(List<Integer> list) {
        long start = System.nanoTime();
        list.stream().parallel().forEach(a -> work(a));
        long end = System.nanoTime();
        long runTime = end - start;
        return runTime;
    }

    public static void main(String[] args) {
        List<Integer> list = Arrays.asList(0, 1, 2, 3, 4, 5, 6, 7, 8, 9);

        long timeSequential = testSequential(list);

        long timeParallel = testParallel(list);

        if(timeSequential < timeParallel) {
            System.out.println("성능 테스트 결과: 순차 처리가 더 빠름");
        } else {
            System.out.println("성능 테스트 결과: 병렬 처리가 더 빠름");
        }
```

```
    }
}
```

실행 결과

```
성능 테스트 결과: 병렬 처리가 더 빠름
```

예제(스트림 소스와 병렬 처리 성능)

```
package parallel_processing;

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

public class ArrayListVsLinkedListExample {
    public static void work(int value){
    }

    public static long testParallel(List<Integer> list) {
        long start = System.nanoTime();
        list.stream().parallel().forEach(a -> work(a));
        long end = System.nanoTime();
        long runTime = end - start;
        return runTime;
    }

    public static void main(String[] args) {
        List<Integer> arrayList = new ArrayList<Integer>();
        List<Integer> linkedList = new LinkedList<Integer>();
        for(int i=0; i<1000000; i++) {
            arrayList.add(i);
            linkedList.add(i);
        }

        long arrayListListParallel = testParallel(arrayList);
        long linkedListParallel = testParallel(linkedList);

        arrayListListParallel = testParallel(arrayList);
        linkedListParallel = testParallel(linkedList);

        if(arrayListListParallel < linkedListParallel) {
            System.out.println("성능 테스트 결과: ArrayList 처리가 더 빠름");
        } else {
            System.out.println("성능 테스트 결과: LinkedList 처리가 더 빠름");
        }
    }
}
```

실행 결과

```
성능 테스트 결과: ArrayList 처리가 더 빠름
```