KB금융그룹 | 국민의 평생 금융파트너

2025년 상반기 K-디지털 트레이닝

# 스트림 요소 처리

[KB] IT’s Your Life

KB 국민은행

# 1 스트림이란?

- **스트림**
  - Java 8부터 컬렉션 및 배열의 요소를 반복 처리하기 위해 스트림 사용
  - 요소들이 하나씩 흘러가면서 처리된다는 의미

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

  - List 컬렉션의 stream() 메소드로 Stream 객체를 얻고, forEach() 메소드로 요소를 어떻게 처리할지를 람다식으로 제공

  - 스트림과 Iterator 차이점

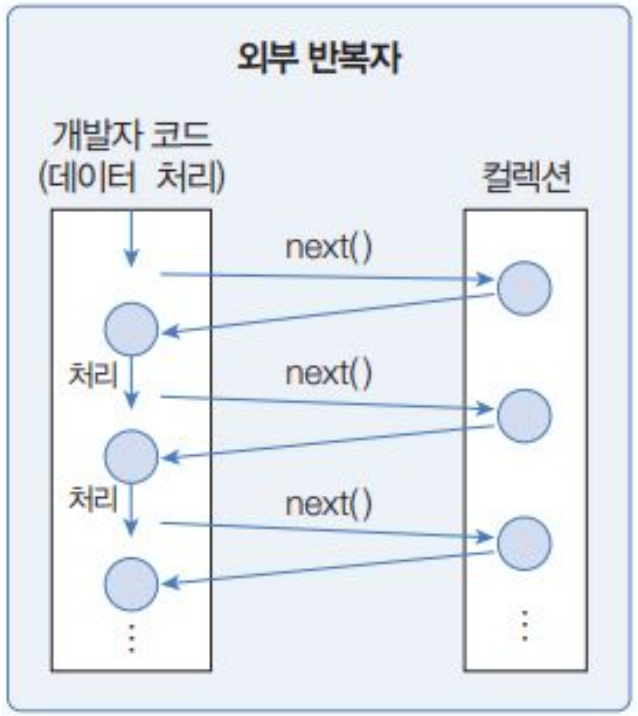
1) 내부 반복자이므로 처리 속도가 빠르고 병렬 처리에 효율적
2) 람다식으로 다양한 요소 처리를 정의
3) 중간 처리와 최종 처리를 수행하도록 파이프 라인을 형성

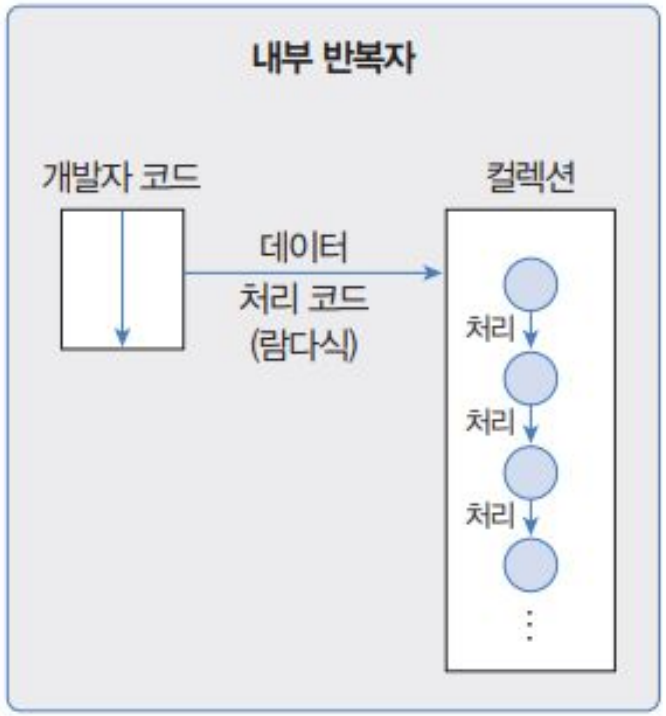
- **StreamExample.java**

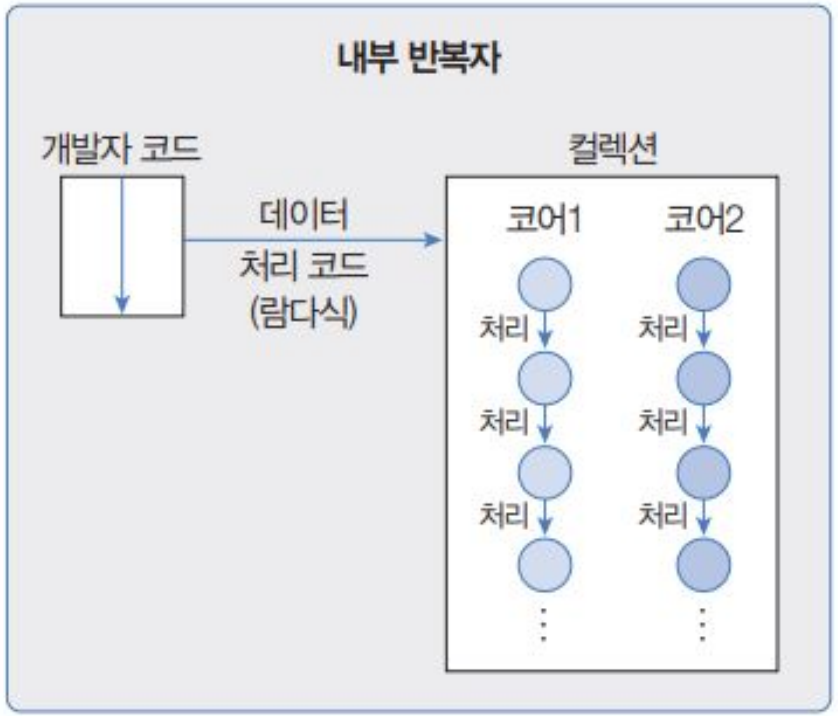
```
package ch17.sec01;

import java.util.HashSet;
import java.util.Iterator;
import java.util.Set;
import java.util.stream.Stream;

public class StreamExample {
	public static void main(String[] args) {
		//Set 컬렉션 생성
		Set<String> set = new HashSet< >();
		set.add("홍길동");
		set.add("신용권");
		set.add("감자바");

		//Stream을 이용한 요소 반복 처리
		Stream<String> stream = set.stream();
		stream.forEach( name -> System.out.println(name) );
	}
}
```

```
홍길동
신용권
감자바
```

- **내부 반복자**
  - 요소 처리 방법을 컬렉션 내부로 주입시켜서 요소를 반복 처리
  - 개발자 코드에서 제공한 데이터 처리 코드(람다식)를 가지고 컬렉션 내부에서 요소를 반복 처리
  - 내부 반복자는 멀티 코어 CPU를 최대한 활용하기 위해 요소들을 분배시켜 병렬 작업 가능



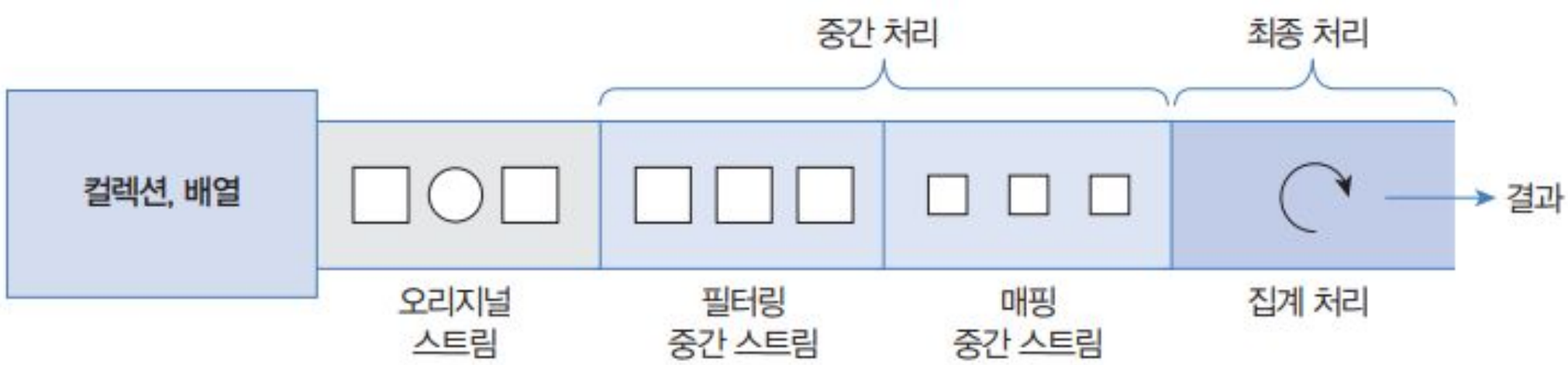



- **ParallelStreamExample.java**

```
package ch17.sec02;

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

public class ParallelStreamExample {
	public static void main(String[] args) {
		//List 컬렉션 생성
		List<String> list = new ArrayList< >();
		list.add("홍길동");
		list.add("신용권");
		list.add("감자바");
		list.add("람다식");
		list.add("박병렬");

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

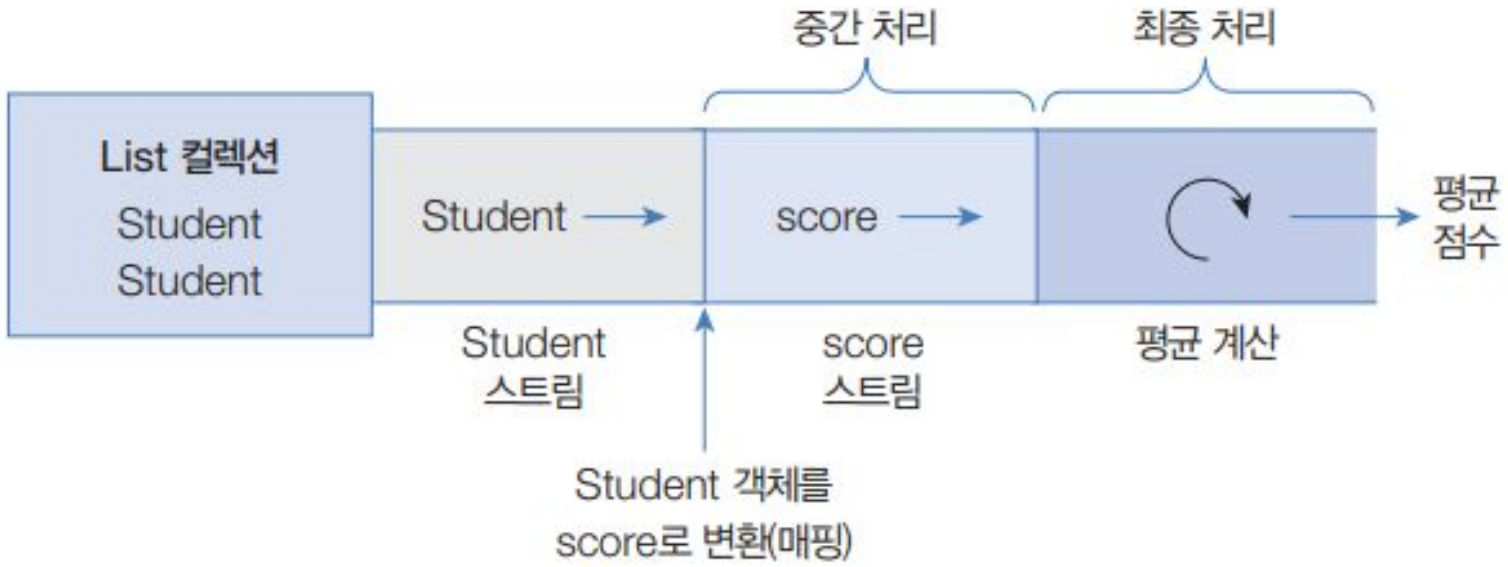
```
박병렬: ForkJoinPool.commonPool-worker-2
신용권: ForkJoinPool.commonPool-worker-1
홍길동: ForkJoinPool.commonPool-worker-3
람다식: ForkJoinPool.commonPool-worker-2
감자바: main
```

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




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

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




```
//Student 스트림
Stream<Student> studentStream = list.stream();
//score 스트림
IntStream scoreStream = studentStream.mapToInt( student -> student.getScore() );
//평균 계산
double avg = scoreStream.average().getAsDouble();
```

Student 객체를 getScore() 메소드의 리턴값으로 매핑

- 스트림 파이프라인
  - 메소드 체이닝 패턴

```
double avg = list.stream()
  .mapToInt(student -> student.getScore())
  .average()
  .getAsDouble();
```

- **Student.java**

```
package ch17.sec03;

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; }
}
```

```
홍길동
신용권
감자바
```

- **StreamPipeLineExample.java**

```
package ch17.sec03;

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

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

		//방법1
		Stream<Student> studentStream = list.stream();
		//중간 처리(학생 객체를 점수로 매핑)
		IntStream scoreStream = studentStream.mapToInt(student -> student.getScore());
		//최종 처리(평균 점수)
		double avg = scoreStream.average().getAsDouble();
		System.out.println("평균 점수: " + avg);
```

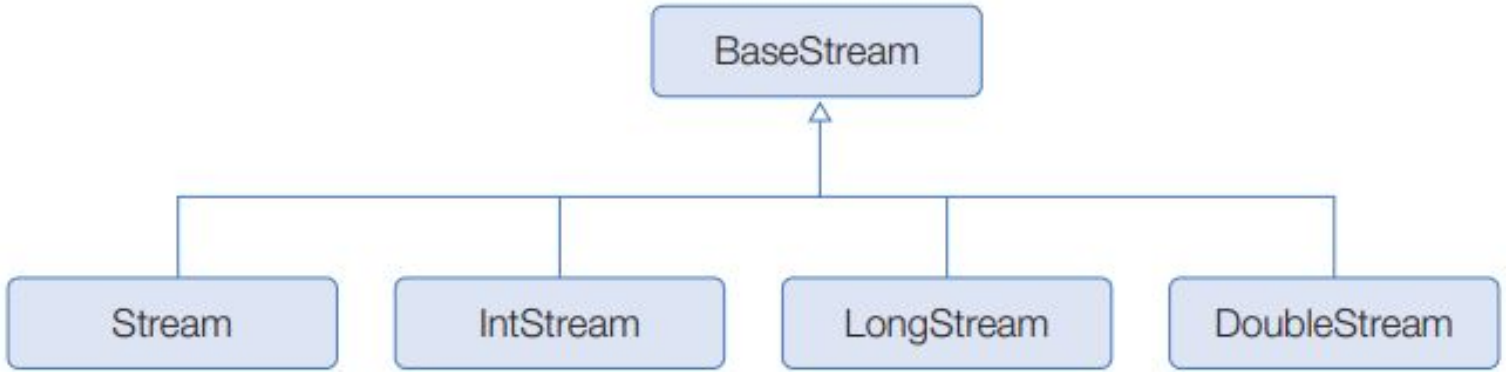
```
평균 점수: 20.0
```

- **StreamPipeLineExample.java**

```
            //방법2
            double avg = list.stream()
                        .mapToInt(student -> student.getScore())
                        .average()
                        .getAsDouble();

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

```
평균 점수: 20.0
```

- 스트림 인터페이스
  - java.util.stream 패키지에는 BaseStream 인터페이스를 부모로 한 자식 인터페이스들은 상속 관계
  - BaseStream에는 모든 스트림에서 사용할 수 있는 공통 메소드들이 정의



- 스트림 인터페이스

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

- **컬렉션으로부터 스트림 얻기**
  - java.util.Collection 인터페이스는 스트림과 parallelStream() 메소드를 가지고 있어 자식 인터페이스인 List와 Set 인터페이스를 구현한 모든 컬렉션에서 객체 스트림을 얻을 수 있음

| 리턴 타입 | 메소드(매개변수) | 소스 |
|---|---|---|
| Stream<T> | java.util.Collection.stream( )<br>java.util.Collection.parallelStream( ) | List 컬렉션<br>Set 컬렉션 |

- **Product.java**

```
package ch17.sec04.exam01;

@AllArgsConstructor
@Data
public class Product {
	private int pno;
	private String name;
	private String company;
	private int price;

}
```

- **StreamExample.java**

```
package ch17.sec04.exam01;

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

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

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

```
{pno:1, name:상품1, company:멋진회사, price:6188}
{pno:2, name:상품2, company:멋진회사, price:2510}
{pno:3, name:상품3, company:멋진회사, price:9932}
{pno:4, name:상품4, company:멋진회사, price:4317}
{pno:5, name:상품5, company:멋진회사, price:170}
```

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

| 리턴 타입 | 메소드(매개변수) | 소스 |
|---|---|---|
| Stream<T> | Arrays.stream(T[ ]), Stream.of(T[ ]) | 배열 |
| IntStream | Arrays.stream(int[ ]), IntStream.of(int[ ]) | |
| LongStream | Arrays.stream(long[ ]), LongStream.of(long[ ]) | |
| DoubleStream | Arrays.stream(double[ ]), DoubleStream.of(double[ ]) | |

- **StreamExample.java**

```
package ch17.sec04.exam02;

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

public class StreamExample {
	public static void main(String[] args) {
		String[] strArray = { "홍길동", "신용권", "김미나"};
		Stream<String> strStream = Arrays.stream(strArray);
		strStream.forEach(item -> System.out.print(item + ","));
		System.out.println();

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

```
홍길동,신용권,김미나,
1,2,3,4,5,
```

- 숫자 범위로부터 스트림 얻기
  - IntStream 또는 LongStream의 정적 메소드인 range()와 rangeClosed() 메소드로 특정 범위의 정수 스트림을 얻을 수 있음

| 리턴 타입 | 메소드(매개변수) | 소스 |
|---|---|---|
| IntStream | IntStream.range(int, int)<br>IntStream.rangeClosed(int, int) | int 범위 |
| LongStream | LongStream.range(long, long)<br>LongStream.rangeClosed(long, long) | long 범위 |

- **StreamExample.java**

```
package ch17.sec04.exam03;

import java.util.stream.IntStream;

public class StreamExample {
	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);
	}
}
```

```
총합: 5050
```

- 파일로부터 스트림 얻기
  - java.nio.file.Files의 lines() 메소드로 텍스트 파일의 행 단위 스트림을 얻을 수 있음

| 리턴 타입 | 메소드(매개변수) | 소스 |
|---|---|---|
| Stream<Path> | Files.list(Path) | 디렉토리 |
| Stream<String> | Files.lines(Path, Charset) | 텍스트 파일 |

- **data.txt**

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

- **StreamExample.java**

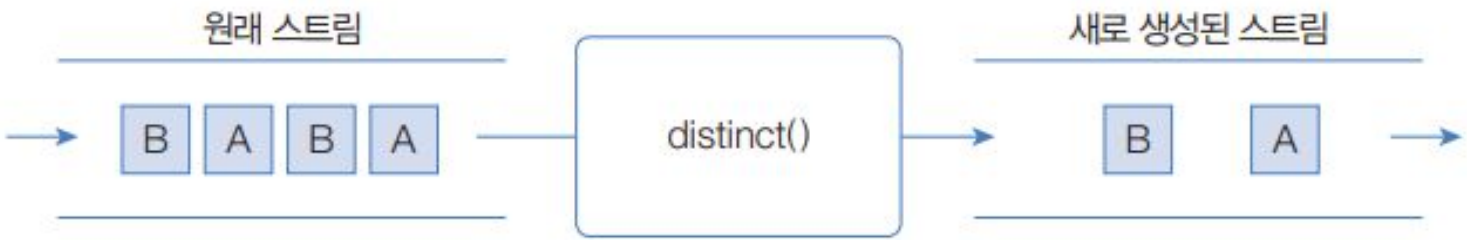
```
package ch17.sec04.exam04;

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

```
{"pno":1, "name":"상품1", "company":"멋진회사",
"price":1558}
{"pno":2, "name":"상품2", "company":"멋진회사",
"price":4671}
{"pno":3, "name":"상품3", "company":"멋진회사", "price":470}
{"pno":4, "name":"상품4", "company":"멋진회사",
"price":9584}
```

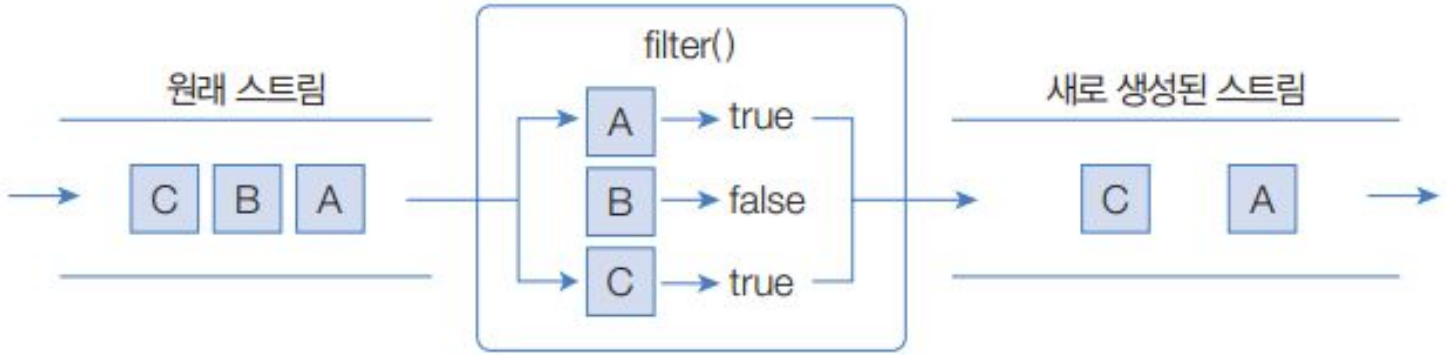
- **필터링**
  - 필터링은 요소를 걸러내는 중간 처리 기능

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

- **필터링**
  - distinct() 메소드: 요소의 중복을 제거



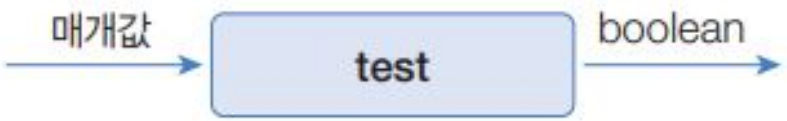

  - filter() 메소드: 매개값으로 주어진 Predicate가 true를 리턴하는 요소만 필터링



- 필터링
  - Predicate: 함수형 인터페이스
  - 

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

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




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

- **FilteringExample.java**

```
package ch17.sec05;

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

public class FilteringExample {
	public static void main(String[] args) {
		//List 컬렉션 생성
		List<String> list = new ArrayList<>();
		list.add("홍길동");
		list.add("신용권");
		list.add("감자바");
		list.add("신용권");
		list.add("신민철");
```

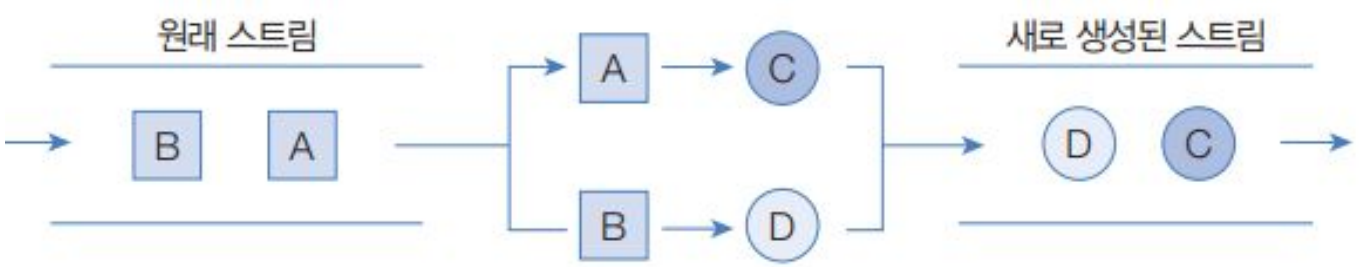
- **FilteringExample.java**

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

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

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

```
홍길동
신용권
감자바
신민철

신용권
신용권
신민철

신용권
신민철
```

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

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



- 매핑
  - 주요 메소드

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

- **맵핑**
  - 매개타입인 Function은 함수형 인터페이스

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

- **매핑**
  - 모든 Function은 매개값을 리턴값으로 매핑(변환)하는 applyXxx() 메소드를 가짐


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

- **Student.java**

```
package ch17.sec06.exam01;

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 ch17.sec06.exam01;

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

public class MapExample {
	public static void main(String[] args) {
		//List 컬렉션 생성
		List<Student> studentList = new ArrayList<>();
		studentList.add(new Student("홍길동", 85));
		studentList.add(new Student("홍길동", 92));
		studentList.add(new Student("홍길동", 87));

		//Student를 score 스트림으로 변환
		studentList.stream()
			.mapToInt(s -> s.getScore())
			.forEach(score -> System.out.println(score));
	}
}
```

```
85
92
87
```

- **매핑**
  - 기본 타입 간의 변환이거나 기본 타입 요소를 래퍼(Wrapper) 객체 요소로 변환하려면 간편화 메소드를 사용할 수 있음

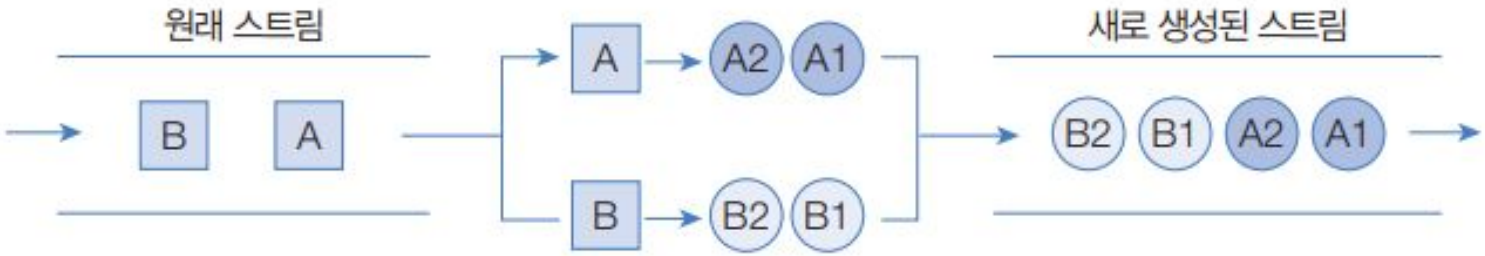
| 리턴 타입 | 메소드(매개변수) | 설명 |
|---|---|---|
| LongStream | asLongStream( ) | int –> long |
| DoubleStream | asDoubleStream( ) | int –> double<br>long –> double |
| Stream<Integer><br>Stream<Long><br>Stream<Double> | boxed( ) | int –>Integer<br>long –> Long<br>double –> Double |

- **MapExample.java**

```
package ch17.sec06.exam02;

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

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

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

		System.out.println();

		intStream = Arrays.stream(intArray);
		intStream
			.boxed()
			.forEach(obj -> System.out.println(obj.intValue()));
	}
}
```

```
1.0
2.0
3.0
4.0
5.0

1
2
3
4
5
```

- **요소를 복수 개의 요소로 변환**
  - flatMapXxx() 메소드: 하나의 요소를 복수 개의 요소들로 변환한 새로운 스트림을 리턴




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

- **FlatMappingExample.java**

```
package ch17.sec06.exam03;

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

public class FlatMappingExample {
	public static void main(String[] args) {
		//문장 스트림을 단어 스트림으로 변환
		List<String> list1 = new ArrayList< >();
		list1.add("this is java");
		list1.add("i am a best developer");
		list1.stream()
			.flatMap(data -> Arrays.stream(data.split(" ")))
			.forEach(word -> System.out.println(word));

		System.out.println();
```

```
this
is
java
i
am
a
best
developer
```

- **FlatMappingExample.java**

```
        //문자열 숫자 목록 스트림을 숫자 스트림으로 변환
        List<String> list2 = Arrays.asList("10, 20, 30", "40, 50");
        list2.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));
    }
}
```

```
10
20
30
40
50
```

- **정렬**
  - 요소를 오름차순 또는 내림차순으로 정렬하는 중간 처리 기능

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

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

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

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

  - 내림차순 정렬
    - Comparator.reverseOrder() 메서드가 리턴하는 Comparator를 제공

```
Stream<Xxx> reverseOrderedStream = stream.sorted(Comparator.reverseOrde());
```

- **Student.java**

```
package ch17.sec07.exam01;

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 String getName() { return name; }
	public int getScore() { return score; }

	@Override
	public int compareTo(Student o) {
		return Integer.compare(score, o.score);
	}
}
```

- **SortingExample.java**

```
package ch17.sec07.exam01;

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

public class SortingExample {
	public static void main(String[] args) {
		//List 컬렉션 생성
		List<Student> studentList = new ArrayList<>();
		studentList.add(new Student("홍길동", 30));
		studentList.add(new Student("신용권", 10));
		studentList.add(new Student("유미선", 20));

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

```
신용권: 10
유미선: 20
홍길동: 30
```

- **FlatMappingExample.java**

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

```
홍길동: 30
유미선: 20
신용권: 10
```

- **Comparator를 이용한 정렬**
  - 요소 객체가 Comparable을 구현하고 있지 않다면, 비교자를 제공하면 요소를 정렬시킬 수 있음

```
sorted((o1, o2) -> { ··· })
```

  - 괄호 안에는 o1이 o2보다 작으면 음수, 같으면 0, 크면 양수를 리턴하도록 작성
  - o1과 o2가 정수일 경우에는 Integer.compare(o1, o2)를, 실수일 경우에는 Double.compare(o1, o2)를 호출해서 리턴값을 리턴 가능

- **Student.java**

```
package ch17.sec07.exam02;

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; }
}
```

- **SortingExample.java**

```
package ch17.sec07.exam02;
…
public class SortingExample {
	public static void main(String[] args) {
		//List 컬렉션 생성
		List<Student> studentList = new ArrayList<>();
		studentList.add(new Student("홍길동", 30));
		studentList.add(new Student("신용권", 10));
		studentList.add(new Student("유미선", 20));

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

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

```
신용권: 10
유미선: 20
홍길동: 30

홍길동: 30
유미선: 20
신용권: 10
```

- **루핑**
  - 스트림에서 요소를 하나씩 반복해서 가져와 처리하는 것

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

- **루핑**
  - 매개타입인 Consumer는 함수형 인터페이스.
  - 모든 Consumer는 매개값을 처리(소비)하는 accept() 메소드를 가지고 있음


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

- **LoopingExample.java**

```
package ch17.sec08;
import java.util.Arrays;

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

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

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

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

```
2
4
총합: 6

2
4
```

- **매칭**
  - 요소들이 특정 조건에 만족하는지 여부를 조사하는 최종 처리 기능
  - allMatch(), anyMatch(), noneMatch() 메소드는 매개값으로 주어진 Predicate가 리턴하는 값에 따라 true 또는 false를 리턴

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

- **MatchingExample.java**

```
package ch17.sec09;

import java.util.Arrays;

public class MatchingExample {
	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);
	}
}
```

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

- **집계**
  - 최종 처리 기능으로 요소들을 처리해서 카운팅, 합계, 평균값, 최대값, 최소값 등 하나의 값으로 산출하는 것

- **스트림이 제공하는 기본 집계**
  - 스트림은 카운팅, 최대, 최소, 평균, 합계 등을 처리하는 다음과 같은 최종 처리 메소드를 제공

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

- **AggregateExample.java**

```
package ch17.sec10;

import java.util.Arrays;

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

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

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

```
2의 배수 개수: 2
2의 배수의 합: 6
```

- **AggregateExample.java**

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

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

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

```
2의 배수의 평균: 3.0
최대값: 4
최소값: 2
```

- **AggregateExample.java**

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

```
첫 번째 3의 배수: 3
```

- **Optional 클래스**
  - Optional, OptionalDouble, OptionalInt, OptionalLong 클래스는 단순히 집계값만 저장하는 것이 아니라, 집계값이 없으면 디폴트 값을 설정하거나 집계값을 처리하는 Consumer를 등록

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

- **Optional 클래스**
  - isPresent() 메소드가 true를 리턴할 때만 집계값 얻기

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

- **Optional 클래스**
  - orElse() 메소드로 집계값이 없을 경우 대비해서 디폴트 값 정해놓기

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

  - ifPresent() 메소드로 집계값이 있을 경우에만 동작하는 Consumer 람다식 제공하기

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

- **OptionalExample.java**

```
package ch17.sec10;

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();
		*/
```

- **OptionalExample.java**

```
//방법1
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");
}

//방법2
double avg = list.stream()
        .mapToInt(Integer :: intValue)
        .average()
        .orElse(0.0);
System.out.println("방법2_평균: " + avg);

//방법3
list.stream()
        .mapToInt(Integer :: intValue)
```

```
방법1_평균: 0.0
방법2_평균: 0.0
```

- 스트림이 제공하는 메소드
  - 스트림은 기본 집계 메소드인 sum(), average(), count(), max(), min()을 제공하지만, 다양한 집계 결과물을 만들 수 있도록 reduce() 메소드도 제공

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

```
(a, b) -> { … return 값; }
또는
(a, b) -> 값   //return 문만 있을 경우 중괄호와 return 키워드 생략 가능
```

- 스트림이 제공하는 메소드
  - reduce()는 스트림에 요소가 없을 경우 예외가 발생하지만,
  - identity 매개값이 주어지면 이 값을 디폴트 값으로 리턴

```
int sum = stream
   .reduce((a, b) -> a+b)
   .getAsInt();
```

```
int sum = stream
   .reduce(0, (a, b) -> a+b);
```

- **Student.java**

```
package ch17.sec11;

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; }
}
```

- **ReductionExample.java**

```
package ch17.sec11;
…

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)
                );
                //방법1
                int sum1 = studentList.stream()
                                .mapToInt(Student :: getScore)
                                .sum();
                //방법2
                int sum2 = studentList.stream()
                                                .map(Student :: getScore)
                                                .reduce(0, (a, b) -> a+b);

                System.out.println("sum1: " + sum1);
                System.out.println("sum2: " + sum2);
        }
}
```

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

- **필터링한 요소 수집**
  - Stream의 collect(Collector<T,A,R> collector) 메소드
    - 필터링 또는 매핑된 요소들을 새로운 컬렉션에 수집하고, 이 컬렉션을 리턴
    - 매개값인 Collector는 어떤 요소를 어떤 컬렉션에 수집할 것인지를 결정
    - 타입 파라미터의 T는 요소, A는 누적기accumulator, 그리고 R은 요소가 저장될 컬렉션

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

- **필터링한 요소 수집**
  - Collector 구현 객체를 얻는 정적 메소드

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

  - Student 스트림에서 남학생만 필터링해서 별도의 List 생성

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

- **필터링한 요소 수집**
  - Collector 구현 객체를 얻는 정적 메소드
    - Student 스트림에서 이름을 키로, 점수를 값으로 갖는 Map 컬렉션 생성

```
Map<String, Integer> map = totalList.stream()
  .collect(
    Collectors.toMap(
      s -> s.getName(),    //Student 객체에서 키가 될 부분 리턴
      s -> s.getScore()    //Student 객체에서 값이 될 부분 리턴
    )
  );
```

    - Java 16의 List 컬렉션 얻기

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

- **Student.java**

```
package ch17.sec12.exam01;

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

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

	public String getName() { return name; }
	public String getSex() { return sex; }
	public int getScore() { return score; }
}
```

- **CollectExample.java**

```
package ch17.sec12.exam01;

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

public class CollectExample {
	public static void main(String[] args) {
		List<Student> totalList = new ArrayList< >();
		totalList.add(new Student("홍길동", "남", 92));
		totalList.add(new Student("김수영", "여", 87));
		totalList.add(new Student("감자바", "남", 95));
		totalList.add(new Student("오해영", "여", 93));

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

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

- **CollectExample.java**

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

        System.out.println();

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

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

```
홍길동
감자바

{오해영=93, 홍길동=92, 감자바=95, 김수영=87}
```

- 요소 그룹핑
  - Collectors.groupingBy () 메소드에서 얻은 Collector를 collect() 메소드를 호출할 때 제공
  - groupingBy()는 Function을 이용해서 T를 K로 매핑하고, K를 키로 해 List<T>를 값으로 갖는 Map

| 리턴 타입 | 메소드 |
|---|---|
| Collector<T,?,Map<K,List<T>>> | groupingBy(Function<T, K> classifier) |


  - "남", "여"를 키로 설정하고 List<Student>를 값으로 갖는 Map 생성

```
Map<String, List<Student>> map = totalList.stream()
  .collect(
    Collectors.groupingBy(s -> s.getSex())    //그룹핑 키 리턴
  );
```

- **Student.java**

```
package ch17.sec12.exam02;

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

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

	public String getName() { return name; }
	public String getSex() { return sex; }
	public int getScore() { return score; }
}
```

- **CollectExample.java**

```
package ch17.sec12.exam02;

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

public class CollectExample {
	public static void main(String[] args) {
		List<Student> totalList = new ArrayList< >();
		totalList.add(new Student("홍길동", "남", 92));
		totalList.add(new Student("김수영", "여", 87));
		totalList.add(new Student("감자바", "남", 95));
		totalList.add(new Student("오해영", "여", 93));

		Map<String, List<Student>> map = totalList.stream()
			.collect(
				Collectors.groupingBy(s -> s.getSex())
			);
```

- **CollectExample.java**

```
            List<Student> maleList = map.get("남");
            maleList.stream().forEach(s -> System.out.println(s.getName()));
            System.out.println();

            List<Student> femaleList = map.get("여");
            femaleList.stream().forEach(s -> System.out.println(s.getName()));
        }
}
```

```
홍길동
감자바

김수영
오해영
```

- **요소 그룹핑**
  - Collectors.groupingBy() 메소드
    - 그룹핑 후 매핑 및 집계(평균, 카운팅, 연결, 최대, 최소, 합계)를 수행할 수 있도록 두 번째 매개값인 Collector를 가질 수 있음
    - Collector를 얻을 수 있는 Collectors의 정적 메소드

| 리턴 타입 | 메소드(매개변수) | 설명 |
|---|---|---|
| Collector | mapping(Function, Collector) | 매핑 |
| Collector | averagingDouble(ToDoubleFunction) | 평균값 |
| Collector | counting( ) | 요소 수 |
| Collector | maxBy(Comparator) | 최대값 |
| Collector | minBy(Comparator) | 최소값 |
| Collector | reducing(BinaryOperator<T>)<br>reducing(T identity, BinaryOperator<T>) | 커스텀 집계 값 |

- **Student.java**

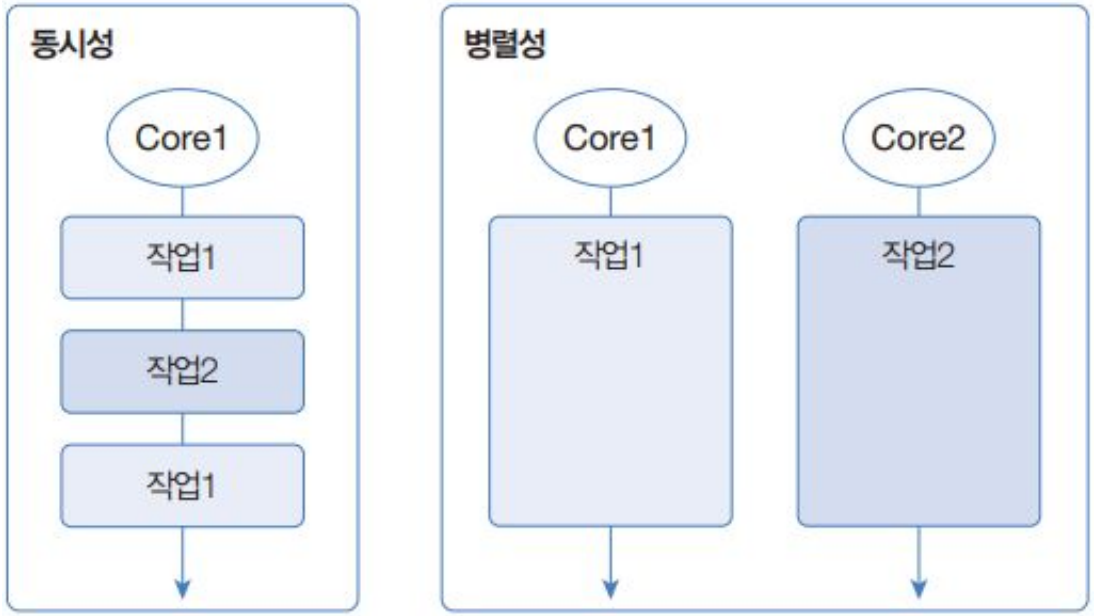
```
package ch17.sec12.exam03;

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

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

	public String getName() { return name; }
	public String getSex() { return sex; }
	public int getScore() { return score; }
}
```

- **CollectExample.java**

```
package ch17.sec12.exam03;
…
import java.util.stream.Collectors;

public class CollectExample {
	public static void main(String[] args) {
		List<Student> totalList = new ArrayList<>();
		totalList.add(new Student("홍길동", "남", 92));
		totalList.add(new Student("김수영", "여", 87));
		totalList.add(new Student("감자바", "남", 95));
		totalList.add(new Student("오해영", "여", 93));

		Map<String, Double> map = totalList.stream()
				.collect(
					Collectors.groupingBy(
						s -> s.getSex(),
						Collectors.averagingDouble(s->s.getScore())
					)
				);

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

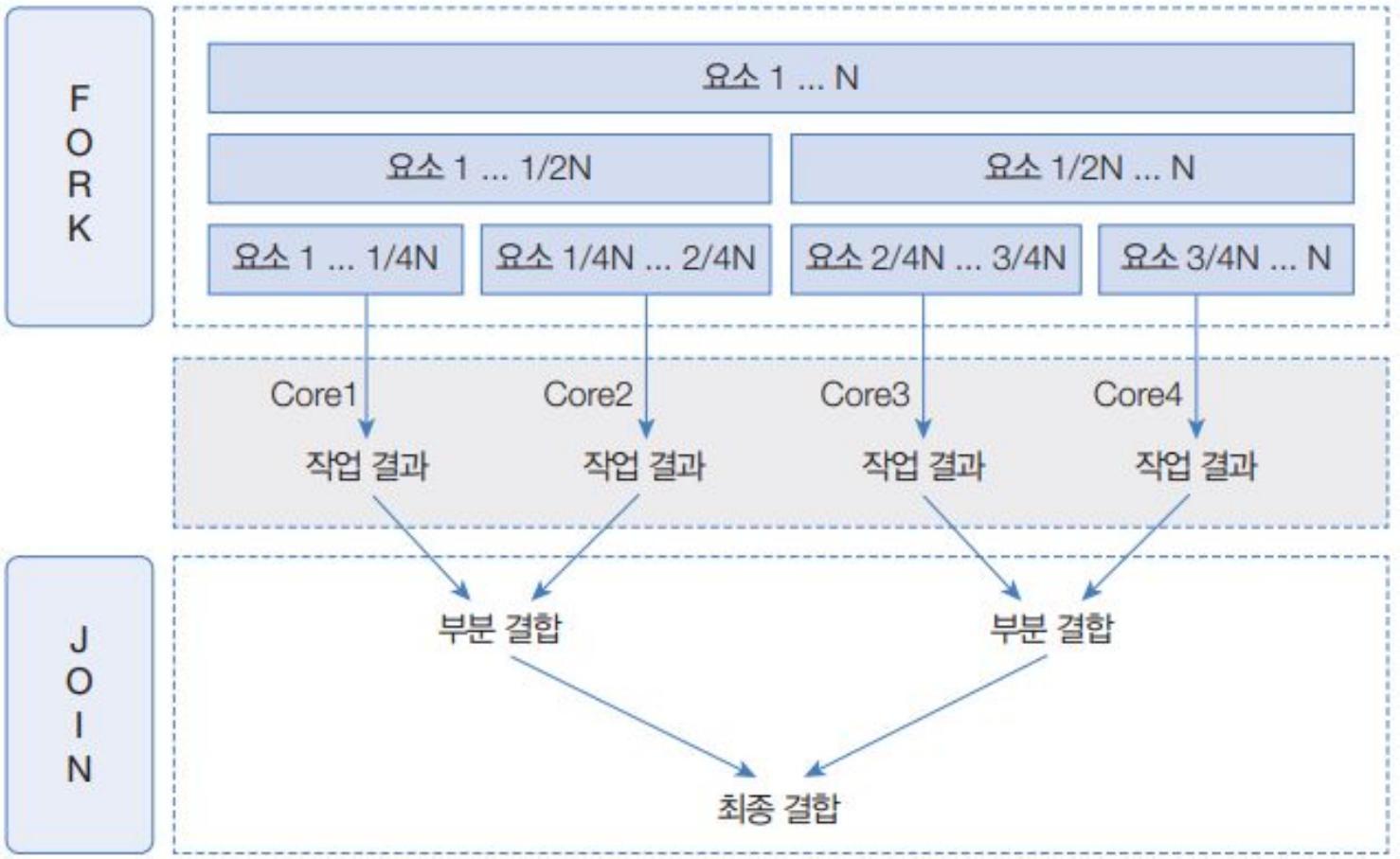
```
{남=93.5, 여=90.0}
```

- **동시성과 병렬성**
  - 동시성: 멀티 작업을 위해 멀티 스레드가 하나의 코어에서 번갈아 가며 실행하는 것
  - 병렬성: 멀티 작업을 위해 멀티 코어를 각각 이용해서 벙렬로 실행하는 것

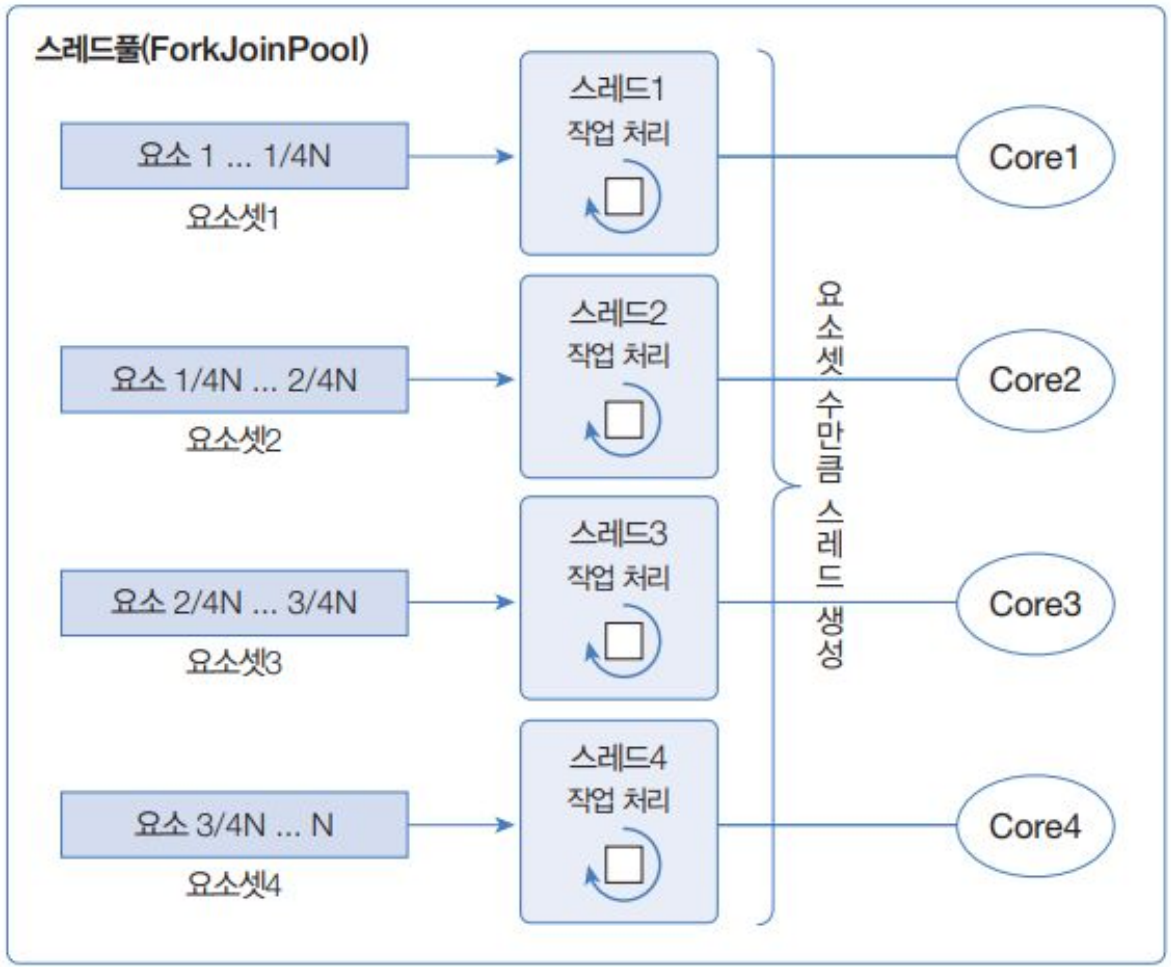



  - 데이터 병렬성: 전체 데이터를 분할해서 서브 데이터셋으로 만들고 이 서브 데이터셋들을 병렬 처리해서 작업을 빨리 끝내는 것
  - 작업 병렬성: 서로 다른 작업을 병렬 처리하는 것

- **포크조인 프레임워크**
  - 포크 단계: 전체 요소들을 서브 요소셋으로 분할하고, 각각의 서브 요소셋을 멀티 코어에서 병렬로 처리
  - 조인 단계: 서브 결과를 결합해서 최종 결과를 만들어냄



- 포크조인 프레임워크
  - ExecutorService의 구현 객체인 ForkJoinPool을 사용해서 작업 스레드를 관리



- **병렬 스트림 사용**
  - 자바 병렬 스트림은 백그라운드에서 포크조인 프레임워크가 사용하므로 병렬 처리 용이
  - parallelStream() 메소드는 컬렉션(List, Set)으로부터 병렬 스트림을 바로 리턴
  - parallel() 메소드는 기존 스트림을 병렬 처리 스트림으로 변환

| 리턴 타입 | 메소드 | 제공 컬렉션 또는 스트림 |
|---|---|---|
| Stream | parallelStream( ) | List 또는 Set 컬렉션 |
| Stream | parallel( ) | java.util.Stream |
| IntStream | | java.util.IntStream |
| LongStream | | java.util.LongStream |
| DoubleStream | | java.util.DoubleStream |

- **ParallelExample.java**

```
package ch17.sec13;

import java.util.ArrayList;
import java.util.List;
import java.util.Random;
import java.util.stream.Stream;

public class ParallelExample {
	public static void main(String[] args) {
		Random random = new Random();

		List<Integer> scores = new ArrayList< >();
		for(int i=0; i<100000000; i++) {		// 1억 개의 Integer 객체 저장
			scores.add(random.nextInt(101));
		}

		double avg = 0.0;
		long startTime = 0;
		long endTime = 0;
		long time = 0;
```

```
{남=93.5, 여=90.0}
```

- **ParallelExample.java**

```
            Stream<Integer> stream = scores.stream();       // 일반 스트림으로 처리
            startTime = System.nanoTime();
            avg = stream
                        .mapToInt(i -> i.intValue())
                        .average()
                        .getAsDouble();
            endTime = System.nanoTime();
            time = endTime - startTime;
            System.out.println("avg: " + avg + ", 일반 스트림 처리 시간: " + time + "ns");

            // 병렬 스트림으로 처리
            Stream<Integer> parallelStream = scores.parallelStream();
            startTime = System.nanoTime();
            avg = parallelStream
                        .mapToInt(i -> i.intValue())
                        .average()
                        .getAsDouble();
            endTime = System.nanoTime();
            time = endTime - startTime;
            System.out.println("avg: " + avg + ", 병렬 스트림 처리 시간: " + time + "ns");
        }
}
```

```
avg: 49.99940116, 일반 스트림 처리 시간: 155627900ns
avg: 49.99940116, 병렬 스트림 처리 시간: 71421700ns
```

# 요소 병렬 처리



- **병렬 처리 성능에 영향을 미치는 요인**
  - 요소의 수와 요소당 처리 시간
  - 스트림 소스의 종류
  - 코어(Core)의 수