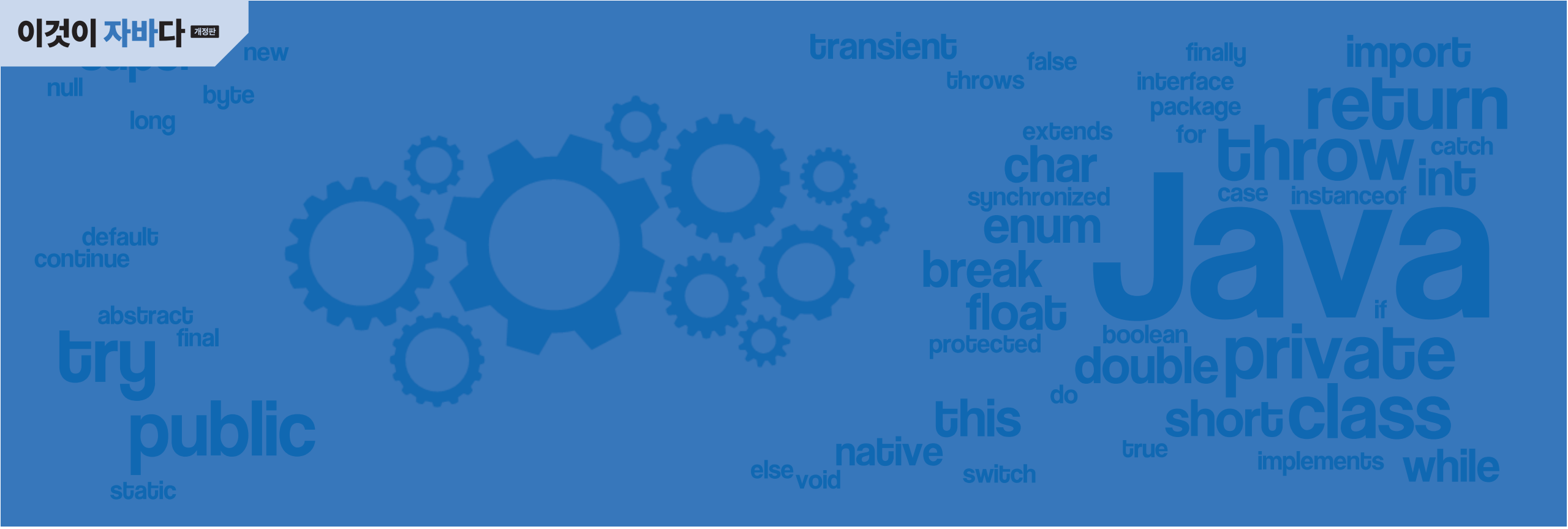


Chapter 17 스트림 요소 처리

▶ Contents

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

17.1 참고 코드 – Iterator vs. Stream

```
import java.util.ArrayList;
import java.util.Iterator;
import java.util.List;

public class IteratorExample {
    public static void main(String[] args) {
        List<Integer> numbers = new ArrayList<>();
        numbers.add(1);
        numbers.add(2);
        numbers.add(3);
        numbers.add(4);
        numbers.add(5);
        // Iterator를 사용하여 반복하면서 요소를 처리
         Iterator<Integer> iterator = numbers.iterator();
        int sum = 0;
        while (iterator.hasNext()) {
            int n = iterator.next();
            if (n % 2 == 0) { // 짝수인 경우에만 처리
                sum += n * 2;
            }
        }
        System.out.println("짝수들의 합: " + sum);
    }
}
```

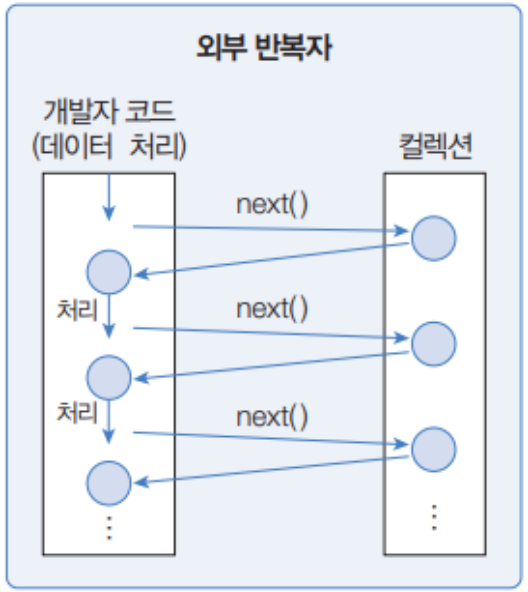
```
import java.util.Arrays;
import java.util.List;

public class StreamExample {
    public static void main(String[] args) {
        List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);

        // Stream을 사용하여 중간 처리와 최종 처리를 수행하는 파이프 라인 형성
         int sum = numbers.stream()        // 스트림 생성
                          .filter(n -> n % 2 == 0)  // 중간 처리: 짝수만 필터링
                          .mapToInt(n -> n * 2)     // 중간 처리: 각 요소를 2배로 변환
                          .sum();                   // 최종 처리: 요소들의 합을 계산

         System.out.println("짝수들의 합: " + sum);
    }
}
```

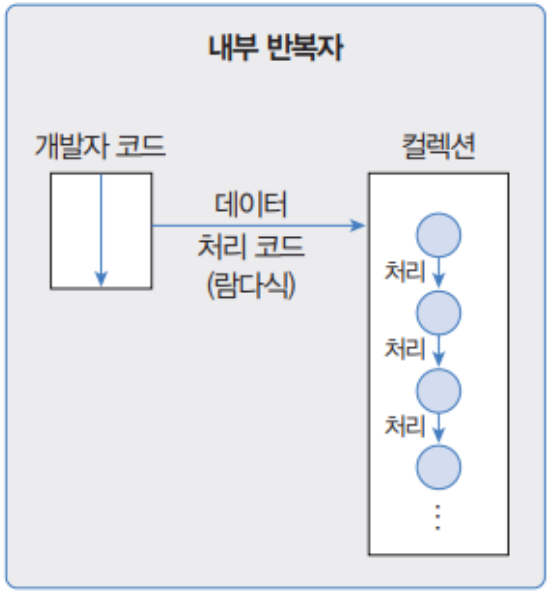
위 코드에서 Stream 예제는 Stream API를 사용하여 중간 처리(filter, mapToInt)와 최종 처리(sum)를 수행하는 파이프 라인을 형성하고 있습니다.

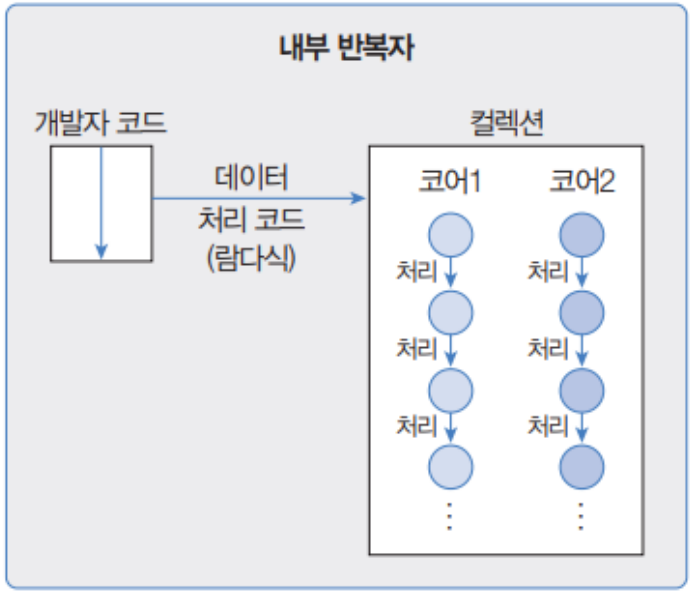
반면에 Iterator 예제는 반복문을 사용하여 직접 요소를 처리하고 있습니다. Stream을 사용한 코드가 간결하고 가독성이 높으며, 내부적으로 최적화된 처리를 수행할 수 있습니다.

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







내부 반복자

- 요소 처리 방법을 컬렉션 내부로 주입시켜서 요소를 반복 처리

```
List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);
numbers.forEach(System.out::println);
```

numbers.forEach(System.out::println)은 리스트 numbers의 각 요소를 출력하는 코드입니다. 이를 자세히 설명하면 다음과 같습니다:

1. Arrays.asList(1, 2, 3, 4, 5)를 통해 정수형 요소를 가진 리스트 numbers를 생성합니다.

2. numbers.forEach(System.out::println)에서 **forEach** 메서드는 리스트의 각 요소를 처리하는 역할을 합니다.

3. System.out::println은 메서드 참조입니다.
   System.out은 자바에서 표준 출력을 나타내는 객체를 참조하고 있으며,
   println은 해당 객체의 println 메서드를 참조합니다.

따라서 numbers.forEach(System.out::println)은 리스트 numbers의 각 요소를 표준 출력으로 출력합니다.

내부 반복자

- 개발자 코드에서 제공한 데이터 처리 코드(람다식)를 가지고 컬렉션 내부에서 요소를 반복 처리

```
import java.util.Arrays;
import java.util.List;
public class Main {
  public static void main(String[] args) {
    List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);
    numbers.forEach(number -> {
    int squared = number * number; // 제곱을 계산
   System.out.println(squared); // 결과를 출력
    });
  }
}
```

위의 코드에서 numbers.forEach(number -> {...}); 부분이 람다식을 통해 제공한 데이터 처리 코드입니다.
이 코드는 리스트의 각 요소를 받아서 제곱을 계산하고, 그 결과를 출력합니다.
forEach 메서드를 통해 각 요소를 반복적으로 처리하며,
람다식을 통해 처리 방법을 정의합니다.

내부 반복자

- 내부 반복자는 멀티 코어 CPU를 최대한 활용하기 위해 요소들을 분배시켜 병렬 작업 가능

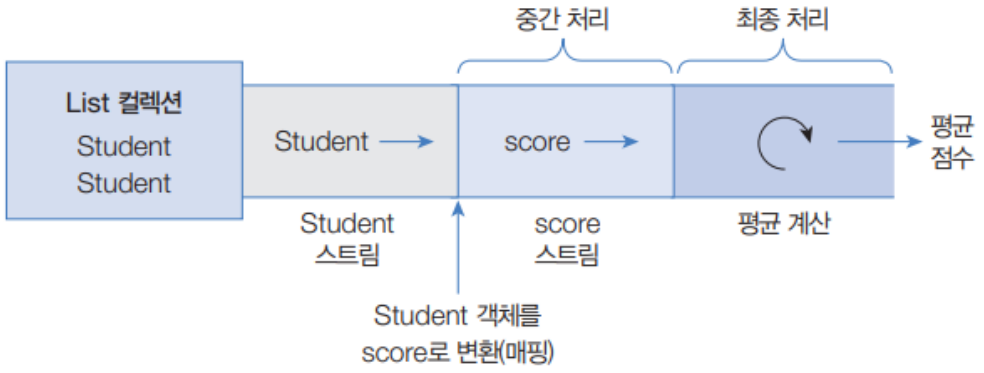
1부터 10까지의 숫자를 포함하는 리스트를 생성하고, 이를 병렬로 처리하여 각 요소를 제곱한 후 총합을 구하는 과정

```
import java.util.ArrayList; import java.util.List;
public class ParallelStreamExample {
public static void main(String[] args) { // 1부터 10까지의 숫자를 포함하는 리스트 생성
  List<Integer> numbers = new ArrayList<>();
  for (int i = 1; i <= 10; i++) {
    numbers.add(i);
  }
  // 리스트를 병렬 스트림으로 변환하여 요소들을 제곱하고 총합을 계산
  int sum = numbers.parallelStream()
                   .mapToInt(num -> num * num)
                   .sum();
  System.out.println("총합: " + sum);
  }
}
```

스트림 파이프라인

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

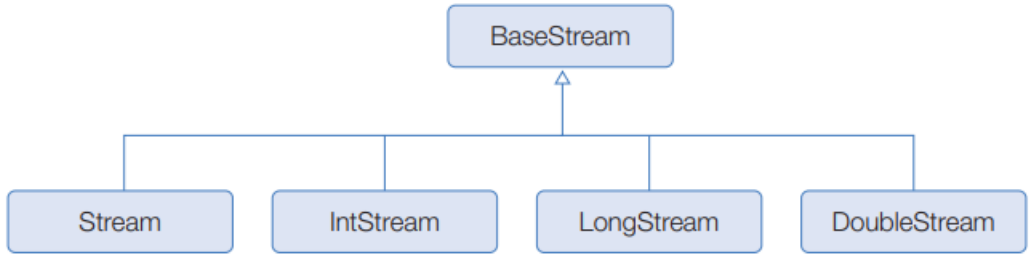
- 오리지널 스트림과 집계 처리 사이의 중간 스트림들은 최종 처리를 위해 요소를 걸러내거나(필터링), 요소를 변환시키거나(매핑), 정렬하는 작업을 수행
- 최종 처리는 중간 처리에서 정제된 요소들을 반복하거나, 집계(카운팅, 총합, 평균) 작업을 수행




```java
import java.util.Arrays; import java.util.List;
public class Main {
  public static void main(String[] args) {
    List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);
   // 소스: 리스트 numbers
   // 중간 연산: map()과 filter() 메서드를 사용하여 스트림을 변환하고 필터링
   // 최종 연산: forEach() 메서드를 사용하여 각 요소를 출력
   numbers.stream()
              .map(x -> x * x) // 제곱
              .filter(x -> x > 10) // 10보다 큰 값만 필터링
              .forEach(System.out::println); // 출력
  }
}
```

스트림 인터페이스

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



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

컬렉션으로부터 스트림 얻기

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

배열로부터 스트림 얻기

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

숫자 범위로부터 스트림 얻기

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

파일로부터 스트림 얻기

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

17.4 리소스로부터 스트림 얻기-예제

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

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

public class Main {
    public static void main(String[] args) {
        List<String> names = new ArrayList<>();
        names.add("Alice");
        names.add("Bob");
        names.add("Charlie");

        // 컬렉션으로부터 스트림 얻기
        Stream<String> stream = names.stream();

        // 스트림을 이용한 처리
        stream.forEach(System.out::println);
    }
}
```

배열로부터 스트림 얻기

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

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

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

        // 배열로부터 스트림 얻기
        IntStream stream = Arrays.stream(numbers);

        // 스트림을 이용한 처리
        stream.forEach(System.out::println);
    }
}
```

숫자 범위로부터 스트림 얻기

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

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

public class Main {
    public static void main(String[] args) {
        // 1부터 10까지의 정수 스트림 얻기
        IntStream stream1 = IntStream.range(1, 11);
        // 1부터 10까지의 정수 스트림 얻기 (10 포함)
        IntStream stream2 = IntStream.rangeClosed(1, 10);

        // 스트림을 이용한 처리
        stream1.forEach(System.out::println);
        stream2.forEach(System.out::println);
    }
}
```

17.4 리소스로부터 스트림 얻기-예제

파일로부터 스트림 얻기

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

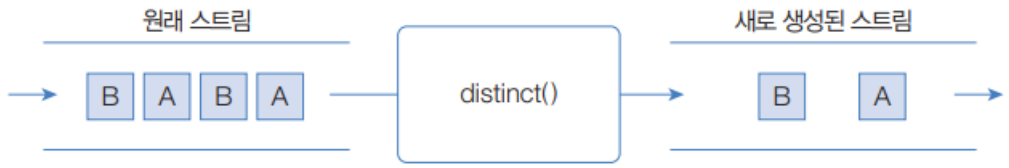
```
import java.io.IOException;
import java.nio.file.Files;
import java.nio.file.Paths;
import java.util.stream.Stream;
public class Main {
    public static void main(String[] args) {
        // 파일로부터 스트림 얻기
        try (Stream<String> lines = Files.lines(Paths.get("example.txt"))) {
            // 파일의 각 행을 출력
            lines.forEach(System.out::println);
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

필터링

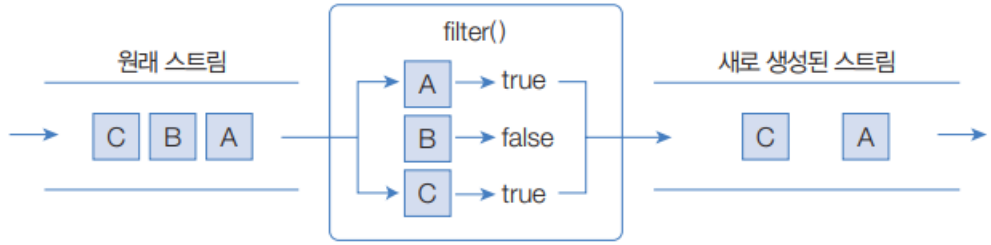
- 필터링은 요소를 걸러내는 중간 처리 기능

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

- distinct() 메소드: 요소의 중복을 제거



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



필터링

- 필터링은 요소를 걸러내는 중간 처리 기능

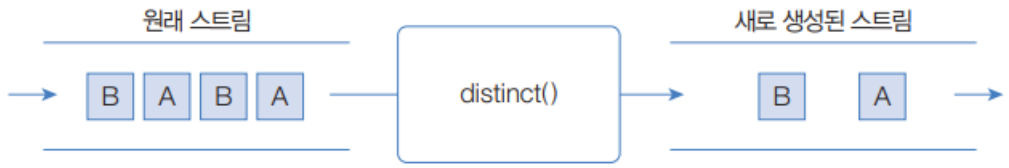
리턴 타입	메소드(매개변수)	설명
Stream IntStream LongStream DoubleStream	distinct( )	– 중복 제거
	filter(Predicate<T>) filter(IntPredicate) filter(LongPredicate) filter(DoublePredicate)	– 조건 필터링 – 매개 타입은 요소 타입에 따른 함수형 인터페이스이므로 람다식으로 작성 가능

- 예제코드

```
import java.util.Arrays; import java.util.List;
public class Main {
  public static void main(String[] args) {
    List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5); // 필터링을 통해 짝수만 걸러내기
    numbers.stream()
                .filter(num -> num % 2 == 0)
                .forEach(System.out::println);
  }
}
```

필터링

- distinct() 메소드: 요소의 중복을 제거

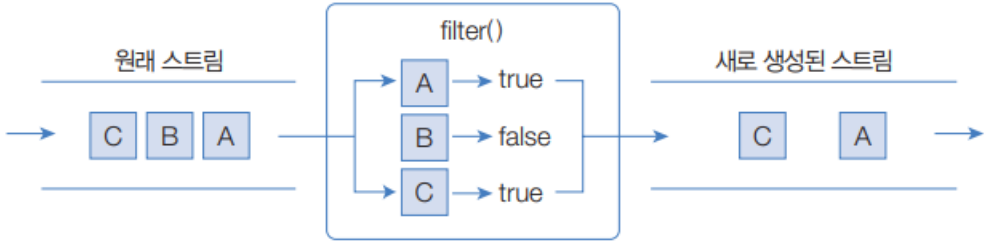


- 예제 코드

```
import java.util.Arrays; import java.util.List;
public class Main {
  public static void main(String[] args) {
    List<Integer> numbers = Arrays.asList(1, 2, 2, 3, 3, 4, 5, 5); // 중복을 제거한 요소 출력
    numbers.stream()
            .distinct()
            .forEach(System.out::println);
  }
}
```

필터링

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



- 예제코드

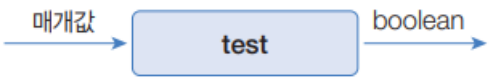
```
import java.util.Arrays; import java.util.List;
public class Main {
  public static void main(String[] args) {
    // 이름의 길이가 4 이상인 요소만 필터링하여 출력
    List<String> names = Arrays.asList("Alice", "Bob", "Charlie", "David", "Eva");

    names.stream()
            .filter(name -> name.length() >= 4)
            .forEach(System.out::println);
  }
}
```

17.5 요소 걸러내기(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.



Predicate 인터페이스의 모든 구현체는 **test() 메서드를 구현해야 한다**는 것을 의미합니다.
이 test() 메서드는 주어진 조건을 만족하는지 여부를 판단하고, 그 결과로 **boolean 값을 반환**합니다.

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

17.5 요소 걸러내기(필터링) - 예제코드

- Predicate: 함수형 인터페이스
- 예제코드

아래의 코드는 주어진 숫자가 양수인지를 판별하는 Predicate를 정의하고, 이를 사용하여 양수인지 여부를 확인하는 예제입니다.

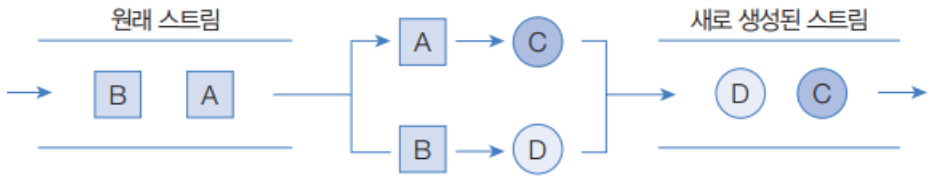
```
import java.util.function.Predicate;
public class Main {
  public static void main(String[] args) {                    // 숫자가 양수인지를 판별하는 Predicate 정의
    Predicate<Integer> isPositive = num -> num > 0;     // 테스트를 위한 숫자
    int number1 = 5; int number2 = -3;                        // Predicate를 이용하여 양수 판별
    boolean result1 = isPositive.test(number1);
    boolean result2 = isPositive.test(number2);           // 결과 출력
    System.out.println(number1 + "는 양수인가? " + result1);
    System.out.println(number2 + "는 양수인가? " + result2);
  }
}
```

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



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

17.6 요소 변환(매핑)-예제코드

매핑 메소드: mapXxx(), asDoubleStream(), asLongStream(), boxed(), flatMapXxx() 등

다음 예제코드는 리스트 안에 있는 문자열을 대문자로 변환하고,

변환된 문자열의 길이를 계산하여 새로운 리스트를 생성하는 과정

```
import java.util.Arrays; import java.util.List; import java.util.stream.Collectors;
public class MappingMethodsExample {
    public static void main(String[] args) {
        List<String> words = Arrays.asList("apple", "banana", "orange", "grape", "watermelon");
       // map(): 문자열을 대문자로 변환
        List<String> uppercaseWords = words.stream()
                                           .map(String::toUpperCase)
                                           .collect(Collectors.toList());
        System.out.println("Uppercase words: " + uppercaseWords);
        // mapToInt(), asIntStream(): 문자열의 길이를 계산
        int totalLength = words.stream()
                               .mapToInt(String::length)
                               .sum();
        System.out.println("Total length of words: " + totalLength);
        // flatMap(), flatMapToInt(): 문자열을 문자로 분리한 후 길이를 계산
         int totalCharCount = words.stream()
                                   .flatMapToInt(word -> word.chars())
                                   .mapToObj(ch -> Character.toString((char) ch))
                                   .collect(Collectors.toList())
                                   .size();
         System.out.println("Total character count of all words: " + totalCharCount);
    }
}
```

17.6 요소 변환(매핑)

- 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

- 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
```

17.6 요소 변환(매핑)-예제코드

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

아래의 예제 코드는 Function 인터페이스를 사용하여 정수를 받아서 그 값을 제곱하여 반환하는 기능을 구현한 것입니다.

```
import java.util.function.Function;
public class FunctionExample {
  public static void main(String[] args) {                          // Function 인터페이스를 구현한 제곱 함수
    Function<Integer, Integer> squareFunction = x -> x * x;  // 제곱 함수를 이용하여 값을 제곱 (람다식 return 생략)
    int result = squareFunction.apply(5);                           // 5를 제곱
    System.out.println("결과: " + result);
  }
}
```

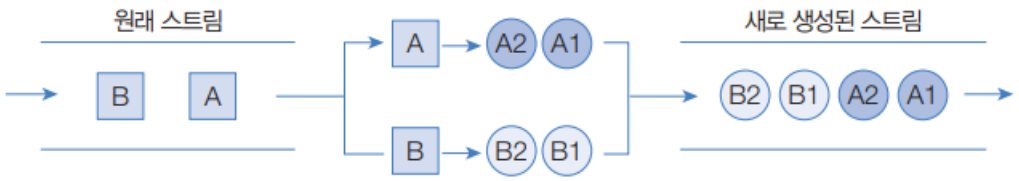
17.6 요소 변환(매핑)

- 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 long –› double
Stream‹Integer› Stream‹Long› Stream‹Double›	boxed( )	int –›Integer long –› Long double –› Double

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

17.6 요소 변환(매핑)-예제코드

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

Java에서는 기본 타입 간의 변환 또는 기본 타입 요소를 래퍼(Wrapper) 객체 요소로 변환할 때, 간편화된 메서드를 제공합니다.
이를 이용하면 명시적인 박싱(boxing)과 언박싱(unboxing) 과정을 줄일 수 있습니다.
대표적으로 mapToXxx() 메서드들이 이러한 기능을 제공합니다. 여기에 간단한 예제 코드를 제시하겠습니다

```
import java.util.Arrays; import java.util.List; import java.util.stream.Collectors;
public class MappingExample {
  public static void main(String[] args) { // 기본 타입 int로 이루어진 리스트
            List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5); // 기본 타입 int를 Integer 객체로 변환하여 새로운 리스트 생성
            List<Integer> wrappedNumbers = numbers.stream()
                                                  .mapToObj(Integer::valueOf)
                                  // boxing: int -> Integer .collect(Collectors.toList());
             System.out.println("Wrapped Numbers: " + wrappedNumbers);
  }
}
```

위 코드에서 mapToObj(Integer::valueOf)는 기본 타입 int를 Integer 객체로 박싱하여 새로운 스트림을 생성합니다. 이렇게 함으로써 기본 타입 요소를 래퍼 객체 요소로 변환할 수 있습니다.

여기서 사용된 mapToObj()는 IntStream에서 Stream<Integer>로 변환하는 데 사용되며, Integer::valueOf는 int를 Integer 객체로 박싱하는 메서드 참조입니다.

17.6 요소 변환(매핑)-예제코드

요소를 복수 개의 요소로 변환

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

```
import java.util.Arrays; import java.util.List; import java.util.stream.Collectors;
public class FlatMapExample {
  public static void main(String[] args) {
    List<List<Integer>> numbers = Arrays.asList(
            Arrays.asList(1, 2, 3),
            Arrays.asList(4, 5, 6),
            Arrays.asList(7, 8, 9) ); // 2차원 리스트를 1차원 리스트로 평면화
    List<Integer> flattenedNumbers = numbers.stream()
                                            .flatMap(innerList -> innerList.stream()) // 2차원 리스트를 1차원으로 평면화
                                            .collect(Collectors.toList());
    System.out.println("Flattened Numbers: " + flattenedNumbers);
  }
}
```

위 코드에서 flatMap() 메서드는 2차원 리스트인 numbers를 1차원 리스트로 만듭니다.
flatMap() 메서드는 각 내부 리스트를 개별적으로 처리하여 하나의 스트림으로 평면화합니다.
그러면 모든 요소가 하나의 리스트로 담기게 됩니다.

```
[[1, 2, 3], [4, 5, 6], [7, 8, 9]]
```

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();
```

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

루핑

- 스트림에서 요소를 하나씩 반복해서 가져와 처리하는 것

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

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


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

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) allMatch(IntPredicate predicate) allMatch(LongPredicate predicate) allMatch(DoublePredicate predicate)	모든 요소가 만족하는지 여부
boolean	anyMatch(Predicate<T> predicate) anyMatch(IntPredicate predicate) anyMatch(LongPredicate predicate) anyMatch(DoublePredicate predicate)	최소한 하나의 요소가 만족하는지 여부
booean	noneMatch(Predicate<T> predicate) noneMatch(IntPredicate predicate) noneMatch(LongPredicate predicate) noneMatch(DoublePredicate predicate)	모든 요소가 만족하지 않는지 여부

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> 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

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 double int long	orElse(T) orElse(double) orElse(int) orElse(long)	집계값이 없을 경우 디폴트 값 설정
void	ifPresent(Consumer) ifPresent(DoubleConsumer) ifPresent(IntConsumer) ifPresent(LongConsumer)	집계값이 있을 경우 Consumer에서 처리

최종 처리에서 average 사용 시 요소 없는 경우를 대비하는 방법

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

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)

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

Student :: getScore
(s) -> s.getScore()
** 메소드 참조, getScore()는 instance method이다.

필터링한 요소 수집

- Stream의 collect(Collector<T,A,R> collector) 메소드는 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

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

* T 요소를 A 누적기를 통해서 R 컬렉션에 저장

17.12 요소 수집 - 부연설명

toMap(Function<T,K> keyMapper,Function<T,U> keyMapper)

Collectors.toMap() 메서드는 스트림의 요소를 매핑하여 맵에 수집하는 데 사용됩니다. 이 메서드는 주어진 두 개의 함수를 사용하여 각 요소를 맵의 키와 값으로 매핑합니다. 여기서 첫 번째 함수는 키 매핑 함수이고, 두 번째 함수는 값 매핑 함수입니다.

이 메서드의 시그니처는 다음과 같습니다.

```
public static <T, K, U> Collector<T, ?, Map<K,U>> toMap(
            Function<? super T, ? extends K> keyMapper, Function<? super T, ? extends U> valueMapper)
```

T: 스트림의 요소 타입입니다.
K: 맵의 키 타입입니다.
U: 맵의 값 타입입니다.

이 메서드는 Collector를 반환하며, T 타입의 요소를 Map<K, U>로 수집합니다.
toMap() 메서드의 두 번째 오버로드는 세 번째 인수로 충돌 해결 방법을 지정할 수 있습니다.
이것은 충돌이 발생했을 때 어떻게 해결할지를 나타내는 방법을 제공합니다.

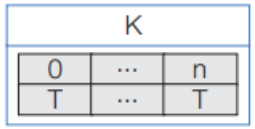
예를 들어, 다음은 toMap() 메서드를 사용하여 리스트의 문자열을 길이를 키로,
문자열을 값으로 매핑하여 맵에 수집하는 예제입니다.

```
import java.util.List; import java.util.Map; import java.util.stream.Collectors;
public class StreamExample {
        public static void main(String[] args) {
                List<String> words = List.of("apple", "banana", "orange777", "grape888"); // 문자열을 길이를 키로, 문자열을 매핑하여 맵에 수집
                Map<Integer, String> wordMap = words.stream()
                                                .collect(Collectors.toMap(
                                                        String::length, // 키 매핑 함수
                                                        word -> word // 값 매핑 함수 ));
                System.out.println("Word Map: " + wordMap);
        }
}
```

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 컬렉션을 생성

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

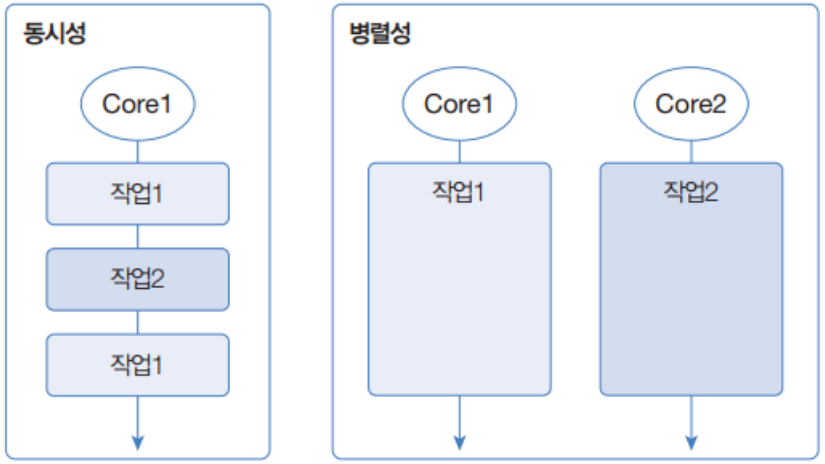


- Collectors.groupingBy() 메소드는 그룹핑 후 매핑 및 집계(평균, 카운팅, 연결, 최대, 최소, 합계)를 수행할 수 있도록 두 번째 매개값인 Collector를 가질 수 있음

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>) reducing(T identity, BinaryOperator<T>)	커스텀 집계 값

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



병렬성 유형	설명	예시
데이터 병렬성	각 요소의 처리를 병렬로 처리하여 속도를 높임	여러 명이 서로 다른 과일을 동시에 세는 작업
작업 병렬성	작업을 여러 단계로 나누어 각 단계를 병렬로 처리함	과일을 세는 작업을 과일 종류를 구별하고, 각각 세는 작업으로 나누어 진행하는 경우

- 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

17.13 요소 병렬 처리 - 부연설명

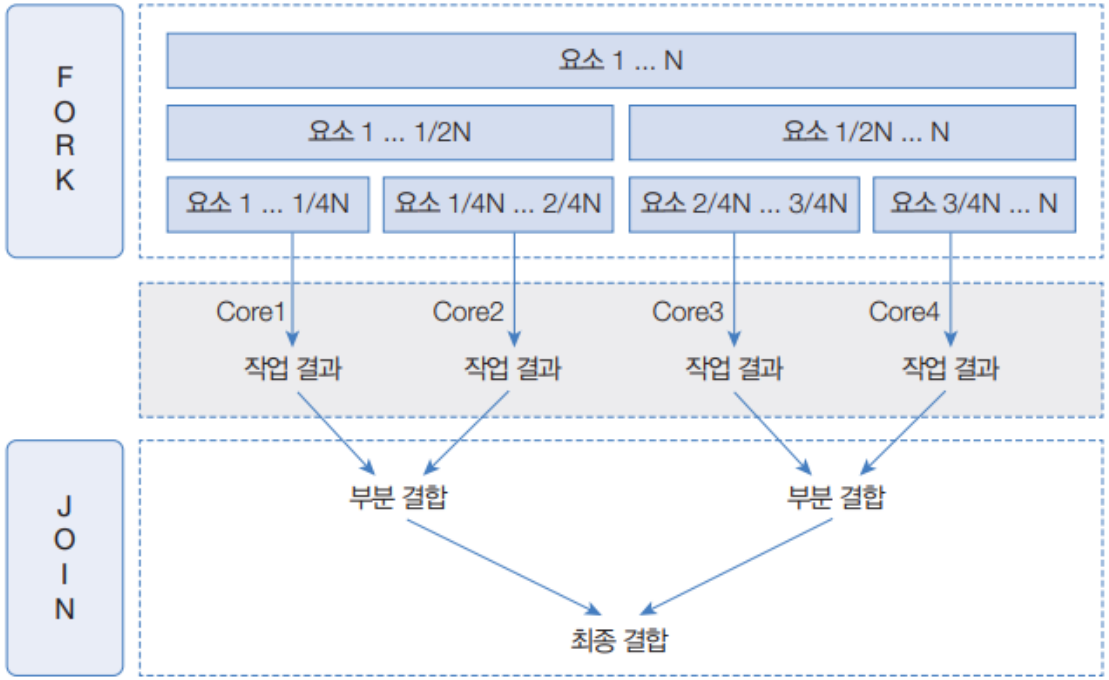
병렬성 유형	설명	예시
데이터 병렬성	각 요소의 처리를 병렬로 처리하여 속도를 높임	여러 명이 서로 다른 과일을 동시에 세는 작업
작업 병렬성	작업을 여러 단계로 나누어 각 단계를 병렬로 처리함	과일을 세는 작업을 과일 종류를 구별하고, 각각 세는 작업으로 나누어 진행하는 경우

데이터 병렬성
: 상자 안에는 여러 종류의 과일이 있습니다.
여러 명의 친구들이 각자 상자 안의 과일을 세는 작업을 할 때,
각 친구는 서로 다른 종류의 과일을 세고 있습니다.
그들은 동시에 작업을 수행하며 서로 영향을 주지 않습니다.
이것이 데이터 병렬성입니다.
각 친구는 독립적으로 작업을 수행하고, 작업이 동시에 진행되므로 전체 작업이 빠르게 처리됩니다.

작업 병렬성
: 상자 안에 있는 과일을 세는 작업을 한 명의 친구에게 맡겼습니다.
그러나 이 작업을 세 단계로 나누었습니다.
첫 번째 단계에서는 과일을 종류별로 구별합니다.
두 번째 단계에서는 각 과일 종류별로 개수를 세고,
세 번째 단계에서는 전체 과일의 개수를 계산합니다.
이때 작업 병렬성을 사용하면 세 단계를 각각의 친구들에게 나눠줄 수 있습니다.
각 친구는 단계를 독립적으로 처리하며, 단계가 완료될 때마다 다음 단계를 진행합니다.
이렇게 함으로써 전체 작업이 빠르게 처리됩니다.
데이터 병렬성은 각 요소를 독립적으로 처리하고,
작업 병렬성은 전체 작업을 단계별로 분할하여 각 단계를 병렬로 처리하는 것입니다.

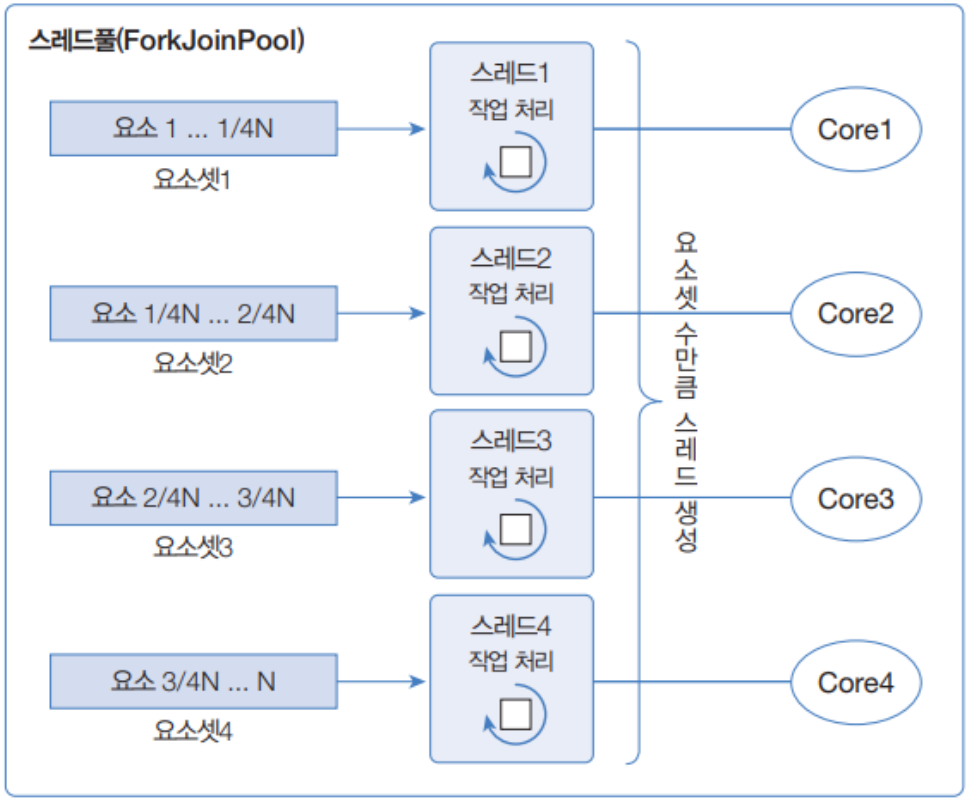
포크조인 프레임워크

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



17.13 요소 병렬 처리

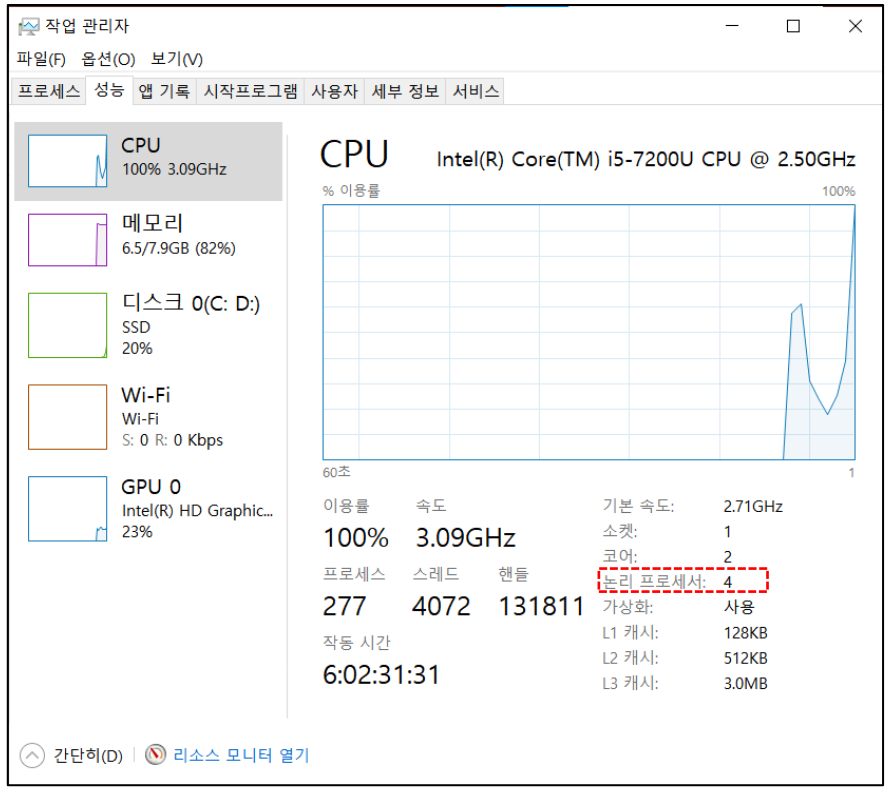
- 포크조인 프레임워크는 ExecutorService의 구현 객체인 ForkJoinPool을 사용해서 작업 스레드를 관리



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



병렬 처리 성능에 영향을 미치는 요인

- 요소의 수와 요소당 처리 시간
- 스트림 소스의 종류
- 코어(Core)의 수


transient
false
finally
import
new
byte
null
long
throws
interface
return
package
extends
for
throw
catch
char
int
synchronized
case
instanceof
enum
default
continue
abstract
final
try
boolean
double
private
public
do
this
short
class
native
true
implements
while
else
void
switch
static

Thank you!