Java

스트림

- 지금까지 컬렉션 및 배열의 저장된 요소를 반복 처리하기 위해서는 for문을 이용하거나 Iterator(반복자)를 사용했다.

```
List<String> list = Arrays.asList("Apple", "Banana", "Cherry");
for (String item : list) {
    // item 처리
}

Set<String> set = Set.of("Apple", "Banana", "Cherry");
Iterator<String> iterator = set.iterator();
while(iterator.hasNext()) {
    String item = iterator.next();
    // item 처리
}
```

- Java 8 부터는 컬렉션 및 배열의 요소를 반복처리하기 위한 또 다른 방법으로 스트림(Stream)을 사용할 수 있다.

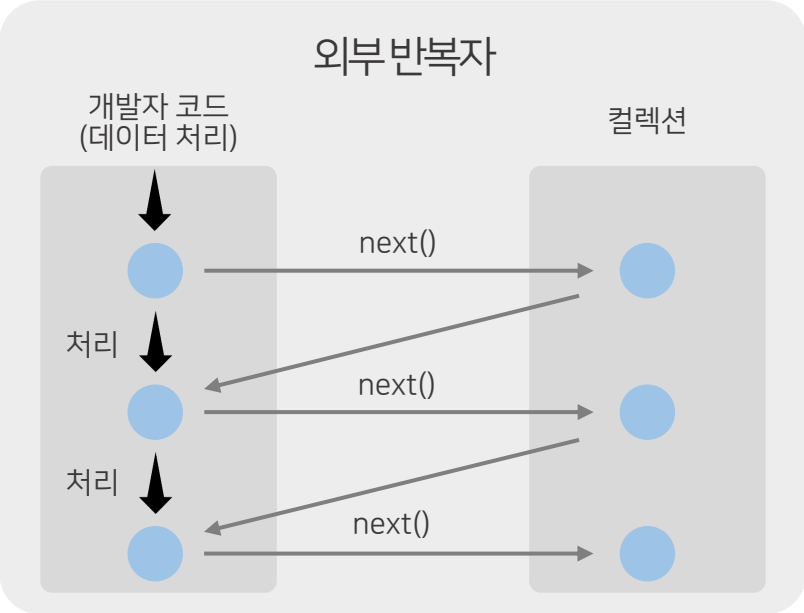
스트림

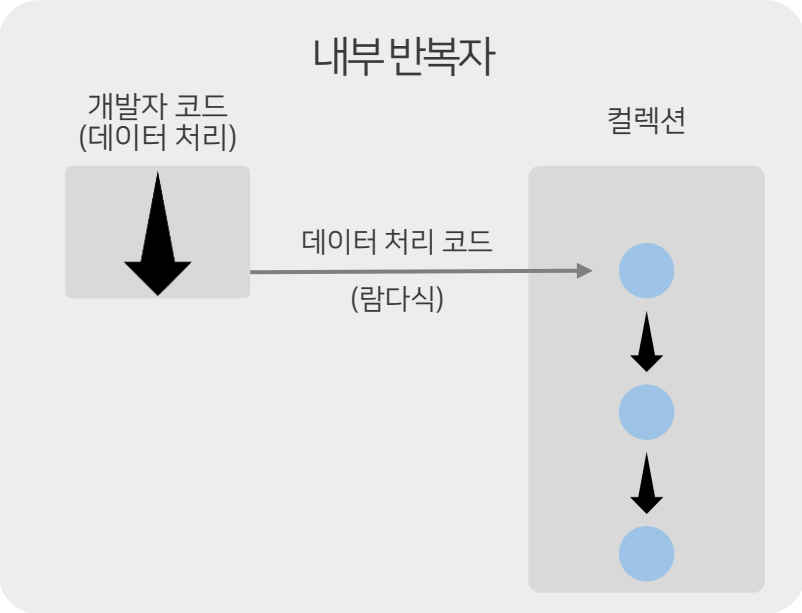
- stream() 메소드로 Stream 객체를 얻고, forEach() 메소드로 요소를 어떻게 처리할 지 람다식으로 제공한다.

- 스트림(Stream)은 Iterator와 비슷한 반복자이지만, 아래와 같은 차이를 가지고 있다.
  1. 내부 반복자이므로 보다 빠른 처리 속도를 가지고 있으며 병렬 처리에 효과적이다.
  2. 람다식으로 다양한 요소 처리를 정의할 수 있다.
  3. 중간 처리와 최종 처리를 수행하도록 파이프 라인을 형성할 수 있다.

스트림 - 내부 반복자

- for문과 Iterator는 컬렉션의 요소를 컬렉션 바깥쪽으로 반복해서 가져와 처리하는 외부 반복자이다.
- 반면에 스트림은 요소 처리 방법을 컬렉션 내부로 주입시켜서 요소를 반복 처리하는 내부 반복자이다.





- 외부 반복자는 컬렉션의 요소를 외부로 가져오는 코드와 처리하는 코드를 모두 개발자 코드가 가지고 있어야 한다.
- 하지만 내부 반복자는 개발자 코드에서 제공한 데이터 처리 코드(람다식)를 가지고 컬렉션 내부에서 요소를 반복 처리한다.

스트림 - 내부 반복자

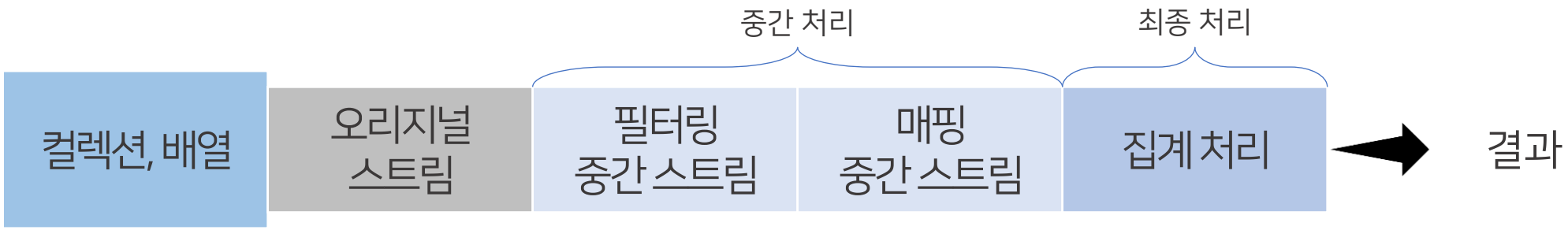
- 내부 반복자를 사용하면 멀티 CPU를 최대한 활용하기 위해 요소들을 분배시켜 병렬 작업을 할 수 있게 된다.
- 이는 하나씩 처리하는 순차적 외부 반복자보다 효율적인 반복이 가능하다.

```
public class ParallelStreamExample {
    public static void main(String[] args) {
        List<String> languageList = new ArrayList<>();
        languageList.add("Java");
        languageList.add("JavaScript");
        languageList.add("Python");
        languageList.add("C");

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

스트림 - 중간 처리와 최종 처리

- 스트림은 하나 이상 연결될 수 있다.
- 아래 그림을 보면 컬렉션의 오리지널 스트림 뒤에 필터링 중간 스트림이 연결될 수 있고, 그 뒤에 매핑 중간 스트림이 연결될 수도 있다.
- 이와 같이 스트림이 연결되어 있는 것을 스트림 파이프라인(Stream Pipeline)이라고 한다.



- 중간 스트림은 최종 처리를 위해 요소를 걸러내거나(필터링), 요소를 변환시키거나(매핑), 정렬하는 작업을 수행한다.
- 최종 처리는 중간 처리에서 정제된 요소들을 반복하거나, 집계 처리 (카운팅, 총합, 평균) 작업을 수행한다.

스트림 - 중간 처리와 최종 처리

- 아래 그림은 Student 객체를 요소로 가지고 있는 컬렉션에서 Student 스트림을 얻고,
  중간 처리를 통해 score 스트림으로 변환한 후 최종 집계 처리로 score 평균을 구하는 과정을 나타낸다.


```
@Data
@AllArgsConstructor
public class Student {
    private String name;
    private int score;
}
```

스트림 - 중간 처리와 최종 처리

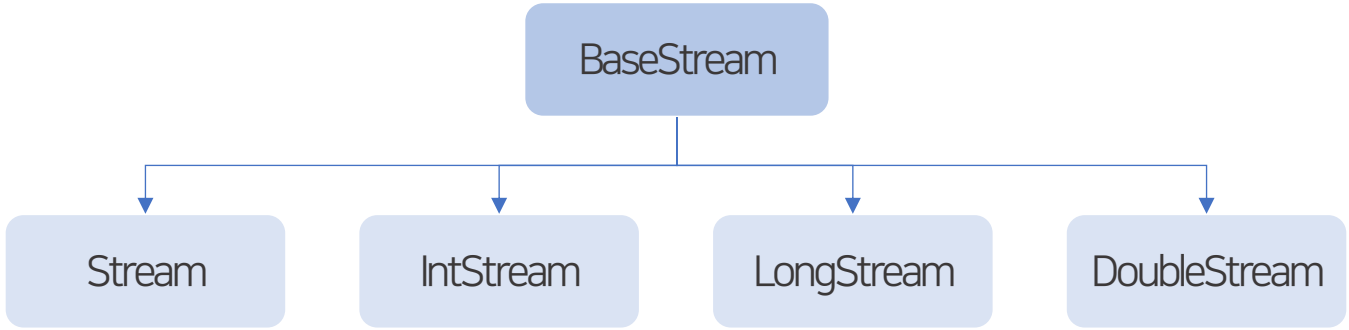
```
public class StudentStreamExample {
    public static void main(String[] args) {
        List<Student> sList = Arrays.asList(
            new Student("Alice", 90),
            new Student("Bob", 80),
            new Student("Carol", 85),
            new Student("David", 95)
        );

        // Stream<Student> studentStream = sList.stream();
        // IntStream scoreStream = studentStream.mapToInt(student -> student.getScore());
        // OptionalDouble optAvg = scoreStream.average();
        // double avg = optAvg.getAsDouble();

        double avg = sList.stream()
            .mapToInt(s -> s.getScore())
            .average()
            .getAsDouble();
        System.out.println("평균 점수 : " + avg);
    }
}
```

스트림 - 리소스로부터 스트림 얻기

- java.util.stream 패키지에는 스트림과 관련한 인터페이스들이 있다.
- BaseStream 인터페이스를 부모로 한 자식 인터페이스는 아래와 같은 상속 관계를 이루고 있다.



- BaseStream에는 모든 스트림에서 사용할 수 있는 공통 메소드들이 정의되어 있으며,
  - Stream은 객체 요소를 처리하는 스트림이고
  - IntStream, LongStream, DoubleStream은 각각 기본 타입인 int, long, double 요소를 처리하는 스트림이다.

스트림 - 리소스로부터 스트림 얻기

- 스트림 인터페이스들은 주로 컬렉션과 배열에서 얻는다. 뿐만 아니라 다양한 리소스로부터 얻을 수 있다.

메소드	리소스
컬렉션.stream() 컬렉션.parellelStream()	List 컬렉션 또는 Set 컬렉션
Arrays.stream(배열) Stream.of(배열) IntStream.of(배열) LongStream.of(배열) DoubleStream.of(배열)	배열
IntStream.range(시작값, 끝값) LongStream.range(시작값, 끝값)	범위
Files.list(Path)	디렉토리
Files.lines(Path, Charset)	텍스트 파일
Random.doubles(…) Random.ints(…) Random.longs(…)	랜덤 수

스트림 - 리소스로부터 스트림 얻기 (컬렉션)

- java.util.Collection 인터페이스는 stream() 메소드와 parallelStream() 메소드를 가지고 있기 때문에 자식 인터페이스인 List와 Set 인터페이스를 구현한 모든 컬렉션에서 객체 스트림을 얻을 수 있다.

```
@Data
public class Product {
    private int pno, price;
    private String name, company;
}

public class ProductExample {
    public static void main(String[] args) {
        List<Product> pList = new ArrayList<>();

        for(int i=1; i<=5; i++) {
            Product p = new Product(i, (int) (10000 * Math.random()), "상품"+i, "회사명");
            pList.add(p);
        }

        Stream<Product> stream = pList.stream();
        stream.forEach(p -> System.out.println(p));
    }
}
```

스트림 - 리소스로부터 스트림 얻기 (배열)

- java.util.Arrays 클래스를 이용하면 다양한 종류의 배열로부터 스트림을 얻을 수 있다.

```
public class ArrayStreamExample {
    public static void main(String[] args) {
        String[] strArr = { "맥북", "아이폰", "에어팟" };
        Stream<String> strStream = Arrays.stream(strArr);
        strStream.forEach(i -> System.out.print(i + ", "));

        System.out.println();

        int[] intArr = { 3, 1, 4, 1, 5, 9, 2 };
        IntStream intStream = Arrays.stream(intArr);
        intStream.forEach(i -> System.out.print(i + ", "));
    }
}
```

스트림 - 리소스로부터 스트림 얻기 (숫자 범위)

- IntStream 또는 LongStream의 정적 메소드인 range() 와 rangeClosed() 메소드를 이용하면, 특정 범위의 정수 스트림을 얻을 수 있다.
- range() 와 rangeClosed() 메소드의 첫 번째 매개값은 시작 수이고, 두 번째 매개값은 끝 수이다.
- range() 메소드는 끝 수를 포함하지 않고, rangeClosed() 메소드는 끝 수를 포함한다.

```
public class RangeStreamExample {
    public static int sum;

    public static void main(String[] args) {
        IntStream stream1 = IntStream.range(1, 10);
        stream1.forEach(i -> sum += i);
        System.out.println(sum);

        sum = 0;

        IntStream stream2 = IntStream.rangeClosed(1, 10);
        stream2.forEach(i -> sum += i);
        System.out.println(sum);
    }
}
```

스트림 - 리소스로부터 스트림 얻기 (파일)

- java.nio.file.Files의 lines() 메소드를 이용하면 텍스트 파일의 행 단위 스트림을 얻을 수 있어서, 텍스트 파일에서 한 행씩 읽고 처리할 때 유용하게 사용할 수 있다.

```
{"pno": 1, "name": "아이폰", "company": "apple", "price": 2000000}
{"pno": 2, "name": "맥북 프로", "company": "apple", "price": 3000000}
{"pno": 3, "name": "갤럭시 워치", "company": "samsung", "price": 400000}
{"pno": 4, "name": "PS5", "company": "sony", "price": 500000}
{"pno": 5, "name": "아이패드", "company": "apple", "price": 800000}
```

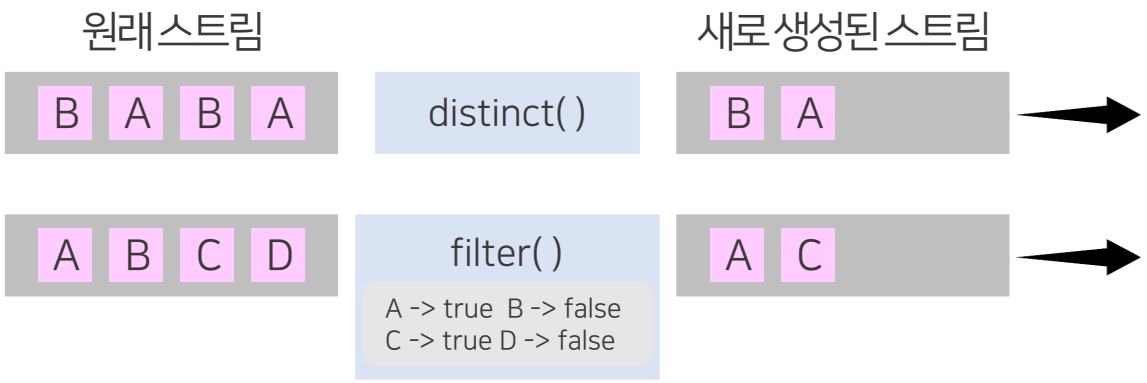
```
public class FileStreamExample {
    public static void main(String[] args) throws Exception {
        String absolutePath = "C:\\Users\\Inkyu\\data.txt";
        Path path = Paths.get(absolutePath);

        Stream<String> stream = Files.lines(path, Charset.defaultCharset());
        stream.forEach(l -> System.out.println(l));
        stream.close();
    }
}
```

스트림 - 요소 걸러내기 (필터링)

- 필터링은 요소를 걸러내는 중간 처리 기능으로, distinct()와 filter() 메소드가 필터링에 쓰인다.

메소드	설명
`distinct( )`	중복 제거
`filter(Predicate 인터페이스)`	조건 필터링 (람다식 작성 가능)



- Predicate 인터페이스는 함수형 인터페이스로 객체를 조사하는 인터페이스로 아래와 같은 종류가 있다.
  - Predicate는 객체 요소를 조사하는 인터페이스이고,
  - IntPredicate, LongPredicate, DoublePredicate는 각각 기본 타입인 int, long, double 요소를 처리하는 스트림이다.
- Predicate 인터페이스에는 매개값을 조사한 후, boolean을 반환하는 test() 메소드가 있다.

스트림 - 요소 걸러내기 (필터링)

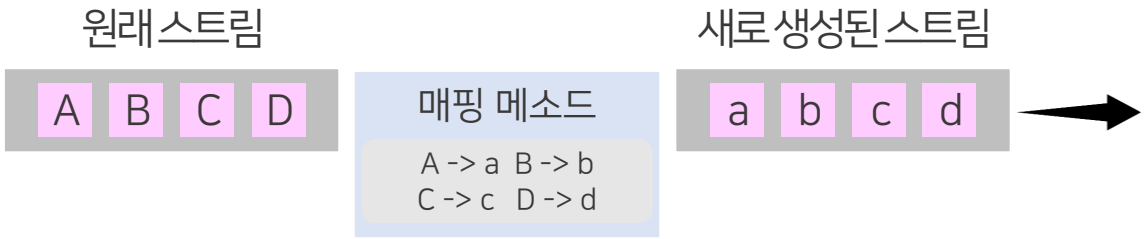
```
public class FilterExample {
    public static void main(String[] args) {
        List<String> nameList = Arrays.asList(
            "우상혁", "황선우", "김우민", "서채현", "신유빈", "우상혁"
        );
        nameList.stream()
            .distinct()
            .forEach(n -> System.out.print(n + ", "));
        System.out.println();

        nameList.stream()
            .filter(n -> n.contains("우"))
            .forEach(n -> System.out.print(n + ", "));
        System.out.println();

        nameList.stream()
            .distinct()
            .filter(n -> n.contains("우"))
            .forEach(n -> System.out.print(n + ", "));
    }
}
```

스트림 - 요소 변환 (매핑)

- 매핑은 스트림의 요소를 다른 요소로 변환하는 중간 처리 기능으로,
  매핑에 사용되는 메소드로는 mapXxx( ), asDoubleStream( ), asLongStream( ), boxed( ), flatMapXxx( ) 등이 있다.



- mapXxx() 메소드의 종류는 상당히 다양하며, 해당 요소를 변환할 때 사용된다.
  - map( ), mapToInt( ), mapToLong( ), mapToDouble( ), mapToObj( )…
- mapXxx() 메소드의 매개변수에는 Function 인터페이스 타입이 들어오는데, Function은 함수형 인터페이스이다.
- 모든 Function 인터페이스에는 매개값을 반환값으로 매핑(변환)하는 applyXxx() 메소드가 있다.

스트림 - 요소 변환 (매핑)

```
public class MapExample1 {
    public static void main(String[] args) {
        List<Student> sList = Arrays.asList(
            new Student("Alice", 90),
            new Student("Bob", 80),
            new Student("Carol", 85),
            new Student("David", 95)
        );

        sList.stream()
            .map(s -> s.getName())
            .forEach(name -> System.out.println(name));
    }
}
```

스트림 - 요소 변환 (매핑)

- asDoubleStream( ), asLongStream( ) 메소드는 기본 타입 간의 변환에 사용되며,
- boxed( ) 메소드는 기본 타입 요소를 래퍼 객체 요소로 변환할 때 사용된다.

메소드	설명
`asLongStream( )`	int -> long
`asDoubleStream( )`	int -> double long -> double
`boxed( )`	int -> Integer long -> Long double -> Double

스트림 - 요소 변환 (매핑)

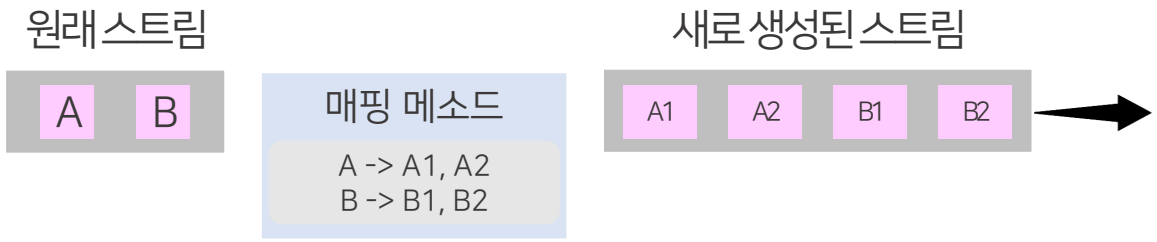
```
public class MapExample2 {
    public static void main(String[] args) {
        IntStream intStream1 = IntStream.range(1, 6);
        intStream1.asDoubleStream()
            .forEach(d -> System.out.println(d));

        System.out.println();

        IntStream intStream2 = IntStream.rangeClosed(1, 5);
        intStream2.boxed()
            .forEach(obj -> System.out.println(obj.intValue()));
    }
}
```

스트림 - 요소 변환 (매핑)

- flatMapXxx( ) 메소드는 하나의 요소를 복수 개의 요소들로 변환해 새로운 스트림을 반환한다.



- flatMapXxx() 메소드의 종류는 상당히 다양하며, 해당 요소를 변환할 때 사용된다.
  - flatMap( ), flatMapToInt( ), flatMapToLong( ), flatMapToDouble( ) …
- flatMapXxx() 메소드의 매개변수에도 MapXxx()과 동일하게 Function 인터페이스 타입이 들어온다.

스트림 - 요소 변환 (매핑)

```java
public class FlatMappingExample {
    public static void main(String[] args) {
        List<String> msgList = new ArrayList<>();
        msgList.add("안녕하세요 최인규입니다.");
        msgList.add("열심히 자바를 공부합시다.");
        msgList.stream().flatMap(msg -> Arrays.stream(msg.split(" ")))
            .forEach(word -> System.out.println(word));

        System.out.println();

        List<String> strNums = Arrays.asList("10, 20, 30", "40, 50");
        strNums.stream()
            .flatMapToInt(item -> {
                String[] strArr = item.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(System.out::println);
    }
}
```

스트림 - 요소 변환 [실습]

```
@Data
@AllArgsConstructor
public class Product {
    private String name;
}

@Data
@AllArgsConstructor
public class Order {
    private List<Product> products;
}

@Data
@AllArgsConstructor
public class User {
    private String name;
    private List<Order> orders;
}
```

스트림 - 요소 변환 [실습]

```
public class FlatMappingPractice {
    public static void main(String[] args) {
        // 샘플 데이터 생성
        List<User> users = Arrays.asList(
            new User("Alice", Arrays.asList(
                new Order(Arrays.asList(new Product("Laptop"), new Product("Mouse"))),
                new Order(Arrays.asList(new Product("Keyboard"), new Product("Monitor")))
            )),
            new User("Bob", Arrays.asList(
                new Order(Arrays.asList(new Product("Tablet"), new Product("Charger"))),
                new Order(Arrays.asList(new Product("Phone"), new Product("Headphones")))
            ))
        );

        users.stream()
            .flatMap(user -> user.getOrders().stream()) // 각 사용자의 모든 주문을 스트림으로
            .flatMap(order -> order.getProducts().stream()) // 각 주문의 모든 상품을 스트림으로
            .forEach(System.out::println); // 각 상품의 이름을 출력
    }
}
```

스트림 - 요소 정렬

- 스트림의 중간 처리 기능으로 정렬이 있고, 이는 sorted() 메소드로 사용 가능하다.
  sorted() 메소드는 요소를 정리한 새로운 스트림을 생성해준다.

- sorted() 메소드는 요소가 Comparable을 구현하고 있어야만 사용 가능하다.
- 만약 Comparable을 구현하지 않은 객체로 구성된 스트림인 경우에는 ClassCastException이 발생한다.

```
@Data
@AllArgsConstructor
public class Student implements Comparable<Student>{
    private String name;
    private int score;

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

스트림 - 요소 정렬

```
public class SortingExample {
    public static void main(String[] args) {
        List<Student> studentList = new ArrayList<>();
        studentList.add(new Student("박명수", 60));
        studentList.add(new Student("유재석", 100));
        studentList.add(new Student("정준하", 40));

        studentList.stream()
            .sorted()
            .forEach(s -> System.out.println(s.getName() + ":" + s.getScore()));
    }
}
```

스트림 - 요소 정렬

- 만약 내림차순으로 정렬하고자 한다면, Comparator.reserverOrder() 메소드가 리턴하는 값을 sorted() 메소드의 매개변수로 제공하면 된다.

```
public class SortingExample {
    public static void main(String[] args) {
        List<Student> studentList = new ArrayList<>();
        studentList.add(new Student("박명수", 60));
        studentList.add(new Student("유재석", 100));
        studentList.add(new Student("정준하", 40));

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

        System.out.println();

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

스트림 - 요소 정렬

- 요소의 객체가 Comparable을 구현하고 있지 않다면, 비교자를 제공해 요소를 정렬시킬 수 있다.
- 비교자는 Comparator 인터페이스를 구현한 객체를 말하는데, 이는 람다식으로 간단하게 작성할 수 있다.

```
@Data
@AllArgsConstructor
public class Student {
    private String name;
    private int score;
}
```

스트림 - 요소 정렬

```
public class SortingExample {
    public static void main(String[] args) {
        List<Student> studentList = new ArrayList<>();
        studentList.add(new Student("박명수", 60));
        studentList.add(new Student("유재석", 100));
        studentList.add(new Student("정준하", 40));

        studentList.stream()
            .sorted((s1,s2) -> Integer.compare(s1.getScore(), s2.getScore()))
            .forEach(s -> System.out.println(s.getName() + ":" + s.getScore()));

        System.out.println();

        studentList.stream()
            .sorted((s1,s2) -> Integer.compare(s2.getScore(), s1.getScore()))
            .forEach(s -> System.out.println(s.getName() + ":" + s.getScore()));
    }
}
```

스트림 - 요소를 하나씩 처리

- 지금까지 예제에서는 forEach() 메소드를 이용해, 요소를 하나씩 반복해서 출력했다.
- 스트림에서 요소를 하나씩 반복해서 가져와 처리하는 것을 루핑(Looping)이라 한다.

- 루핑 메소드에는 peek()와 forEach()가 있다.
  - peek() 메소드는 스트림을 반환하는 메소드로 중간 처리 메소드이고, forEach() 메소드는 void를 반환하는 최종 처리 메소드이다.
  - 즉, peek()는 최종 처리가 뒤에 붙지 않으면 동작하지 않는다.

```
public class LoopExample {
    public static void main(String[] args) {
        int[] intArr = { 1, 2, 3, 4, 5 };
        int total = Arrays.stream(intArr)
            .filter(i -> i%2==0).peek(even -> System.out.println(even)).sum();
        System.out.println("합계: " + total + "\n");

        Arrays.stream(intArr)
            .filter(i -> i%2!=0).forEach(odd -> System.out.println(odd));
    }
}
```

스트림 - 요소 조건 만족 여부 (매칭)

- 요소들이 특정 조건에 만족하는지 여부를 조사하는 최종 처리 기능을 칭이라 한다.
- 매칭과 관련한 메소드는 allMatch(), anyMatch(). noneMatch() 메소드가 있다.

메소드	설명
allMatch( )	모든 요소가 만족하는지 여부
anyMatch( )	최소한 하나의 요소가 만족하는지 여부
noneMatch( )	모든 요소가 만족하지 않는지 여부

스트림 - 요소 조건 만족 여부 (매칭)

```
public class MatchingExample {
    public static void main(String[] args) {
        int[] intArr1 = { 2, 4 };
        boolean result1 = Arrays.stream(intArr1).allMatch(i -> i%2==0);
        System.out.println("{2, 4} 모두 2의 배수인가? : " + result1);

        int[] intArr2 = { 2, 3, 4 };
        boolean result2 = Arrays.stream(intArr2).allMatch(i -> i%2==0);
        System.out.println("{2, 3, 4} 모두 2의 배수인가? : " + result2);

        boolean result3 = Arrays.stream(intArr2).anyMatch(i -> i%3==0);
        System.out.println("{2, 3, 4} 하나라도 3의 배수가 있는가? : " + result3);

        boolean result4 = Arrays.stream(intArr2).noneMatch(i -> i%5==0);
        System.out.println("{2, 3, 4} 5의 배수가 없는가? : " + result4);
    }
}
```

스트림 - 요소 기본 집계

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

- 스트림은 아래와 같은 최종 집계 처리 메소드를 제공한다.

반환 타입	메소드	설명
long	count()	요소 개수
Optional, OptionalInt, OptionalLong, OptionalDouble	findFirtst()	첫 번째 요소
Optional<T> Optional, OptionalInt, OptionalLong, OptionalDouble	max(Comparator<T>) max()	최대 요소
Optional<T> Optional, OptionalInt, OptionalLong, OptionalDouble	min(Comparator<T>) min ()	최소 요소
OptinalDouble	average()	요소 평균
int, long, double	sum()	요소 총합

스트림 - 요소 기본 집계

- 컬렉션의 요소는 동적으로 추가되는 경우가 많다.
- 만약 컬렉션에 요소가 존재하지 않으면 집계 값을 산출할 수 없으므로 NoSuchElementException 예외가 발생한다.

- Optional 클래스는 집계값만을 저장하는 것이 아니라, Optional 클래스가 제공하는 메소드로
  집계값이 존재하지 않을 경우, 디폴트 값을 설정하거나, 집계값을 처리하는 Consumer를 등록할 수 있다.

반환 타입	메소드	설명
Boolean	isPresent()	집계값이 있는지 여부
T, double, int, long	orElse(기본값)	집계값이 없을 경우 디폴트 값 설정
void	ifPresent(i -> {})	집계값이 있을 경우 Consumer 람다식으로 처리

스트림 - 요소 기본 집계

- 만약 평균을 구하는 average()를 최종 처리에서 사용할 경우, 아래와 같이 3가지 방법으로 집계값이 없는 경우를 대비할 수 있다.

1) isPresent() 메소드가 true를 반환할 때만 집계값을 얻는다.

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

2) orElse() 메소드로 집계값이 없을 경우를 대비해서 디폴트 값을 정해놓는다.

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

3) ifPresent() 메소드로 집계값이 있을 경우에만 동작하는 Consumer 람다식을 제공한다.

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

스트림 - 요소 기본 집계

```
public class OptionalExample {
    public static void main(String[] args) {
        List<Integer> list = new ArrayList<>();

        OptionalDouble optionalAverage = list.stream()
            .mapToInt(Integer::intValue)
            .average();

        // optionalAverage.getAsDouble();

        if(optionalAverage.isPresent()) {
            System.out.println("1) 평균: " + optionalAverage.getAsDouble());
        } else {
            System.out.println("1) 평균: " + 0.0);
        }

        System.out.println("2) 평균: " + optionalAverage.orElse(0.0));

        optionalAverage.ifPresent(a -> System.out.println("3) 평균: " + a));
    }
}
```

스트림 - 요소 커스텀 집계

- 스트림은 기본 집계 메소드 외에도 다양한 집계 결과물을 만들 수 있도록 reduce() 메소드도 제공한다.
- reduce() 메소드도 마찬가지로 스트림에 요소가 없을 경우, 예외를 발생시킨다.
- 하지만 매개변수로 identity 매개값을 추가하면, 해당 값을 초기값으로 하기 때문에 예외를 발생시키지 않는다

```
public class ReductionExample1 {
    public static void main(String[] args) {
        int[] intArr = {1,2,3,4};

        // int result = Arrays.stream(intArr).reduce((a, b) -> a * b).getAsInt();

        int result = Arrays.stream(intArr).reduce(1, (a, b) -> a * b);

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

스트림 - 요소 커스텀 집계

```
@Data
@AllArgsConstructor
public class Student {
    private String name;
    private int score;
}

public class ReductionExample2 {
    public static void main(String[] args) {
        List<Student> studentList = new ArrayList<>();
        studentList.add(new Student("박명수", 60));
        studentList.add(new Student("유재석", 100));
        studentList.add(new Student("정준하", 40));

        int sum1 = studentList.stream().mapToInt(Student::getScore).sum();
        System.out.println(sum1);

        int sum2 = studentList.stream().mapToInt(Student::getScore).reduce(0, (a, b) -> a + b);
        System.out.println(sum2);
    }
}
```

스트림 - 요소 수집

- 스트림은 요소들을 중간 처리(필터링, 매핑)한 후 요소들을 수집한느 최종 처리 메소드인 collect() 메소드를 제공한다.
- 이 메소드를 이용하면 필요한 요소만을 컬렉션에 담을 수 있으며, 요소들을 그룹핑하여 집계도 할 수 있다.

스트림 - 요소 수집

```
public class CollectExample1 {
    public static void main(String[] args) {
        List<Student> sList = new ArrayList<>();
        sList.add(new Student("Alice", "여", 90)); sList.add(new Student("Bob", "남", 80));
        sList.add(new Student("Choi", "남", 100)); sList.add(new Student("Diana", "여", 95));

        List<Student> maleList1 = sList.stream().filter(s->s.getGender().equals("남"))
            .collect(Collectors.toList()); // Java16 미만

        List<Student> maleList2 = sList.stream().filter(s->s.getGender().equals("남"))
            .toList(); // Java16 이상

        Set<Student> maleSet1 = sList.stream().filter(s->s.getGender().equals("남"))
            .collect(Collectors.toSet());

        Map<String, Integer> sMap = sList.stream().collect(Collectors.toMap(
                s -> s.getName(),
                s -> s.getScore()
            ));
    }
}
```

스트림 - 요소 그룹핑

- 스트림에서 제공하는 collect() 메소드는 단순히 요소를 수집하는 기능 이외에 컬렉션의 요소들을 그룹핑해서 Map 객체를 생성하는 기능도 있다.
- collect() 메소드의 매개변수에 Collectors.groupingBy() 메소드에서 얻은 Collector를 제공하면 된다.

```
public class CollectExample2 {
    public static void main(String[] args) {
    List<Student> sList = new ArrayList<>();
    sList.add(new Student("Alice", "여", 90));
    sList.add(new Student("Bob", "남", 80));
    sList.add(new Student("Choi", "남", 100));
    sList.add(new Student("Diana", "여", 95));

    Map<String, List<Student>> genderMap = sList.stream().collect(
        Collectors.groupingBy(Student::getGender)
    );

    List<Student> maleList = genderMap.get("남");
    List<Student> femaleList = genderMap.get("여");
    }
}
```

스트림 - 요소 그룹핑

- Collectors.groupingBy() 메소드는 그룹핑 후 매핑 및 집계를 할 수 있도록, 두번째 매개값으로 Collector를 가질 수 있다.
- 두 번째 매개값으로 사용되어 Collector를 얻을 수 있는 Collectors의 정적 메소드들은 아래와 같다.

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

스트림 - 요소 그룹핑

```
public class CollectExample3 {
    public static void main(String[] args) {
        List<Student> sList = new ArrayList<>();
        sList.add(new Student("Alice", "여", 90));
        sList.add(new Student("Bob", "남", 80));
        sList.add(new Student("Choi", "남", 100));
        sList.add(new Student("Diana", "여", 95));

        Map<String, Double> averageMap = sList.stream()
            .collect(
                Collectors.groupingBy(
                    Student::getGender,
                    Collectors.averagingDouble(Student::getScore)
                )
            );

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