

```
for object to mirror...
mirror_mod.mirror_object = ...

operation == "MIRROR_X":
    mirror_mod.use_x = True
    mirror_mod.use_y = False
    mirror_mod.use_z = False
operation == "MIRROR_Y":
    mirror_mod.use_x = False
    mirror_mod.use_y = True
    mirror_mod.use_z = False
operation == "MIRROR_Z":
    mirror_mod.use_x = False
    mirror_mod.use_y = False
    mirror_mod.use_z = True
```

```
@selection at the end -add
mirror_ob.select= 1
modifier_ob.select=1
context.scene.objects.active = mirror_ob
print("Selected" + str(modifier_ob.name))
mirror_ob.select = 0
= bpy.context.selected_objects[0]
data.objects[one.name].select = 1

print("please select exactly one object")
```

```
-- OPERATOR CLASSES -----

bpy.types.Operator):
    """X mirror to the selected object.mirror_mirror_x"""
    bl_label = "Mirror X"
```

Java 기초

스트림(Stream)

1. 스트림

- Java7 이전까지 List<String> 컬렉션에 순차적으로 처리하기 위해 Iterator를 사용했어야 했다.
- 하지만 Stream이라는 Java8 에서 제공하는 클래스가 나오면서 랴다식으로 요소 처리 코드를 제공할 수 있게 되었다.
- 또한 내부 반복자를 사용하므로 병렬 처리가 쉬어진다는 장점이 있다.
- 스트림은 중간 처리와 최종 처리 또한 가능하다.

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



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

1. 스트림

- 반복자 스트림 예제(1).

```
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; }  
}
```

```
public class LambdaExpressionsExample {  
    public static void main(String[] args) {  
        List<Student> list = Arrays.asList(  
            new Student("홍길동", 90),  
            new Student("신용권", 92)  
        );  
  
        Stream<Student> stream = list.stream();  
        stream.forEach(s -> {  
            String name = s.getName();  
            int score = s.getScore();  
            System.out.println(name + "-" + score);  
        });  
    }  
}
```

1. 스트림

- 반복자 스트림 예제(2).

```
public class ParallelExample {  
    public static void main(String[] args) {  
        List<String> list = Arrays.asList("홍길동", "신용권", "감자바", "람다식", "박병렬");  
  
        //순차 처리  
        Stream<String> stream = list.stream();  
        stream.forEach(ParallelExample :: print);  
  
        System.out.println();  
  
        //병렬 처리  
        Stream<String> parallelStream = list.parallelStream();  
        parallelStream.forEach(ParallelExample :: print);  
    }  
  
    public static void print(String str) {  
        System.out.println(str + " : " + Thread.currentThread().getName());  
    }  
}
```

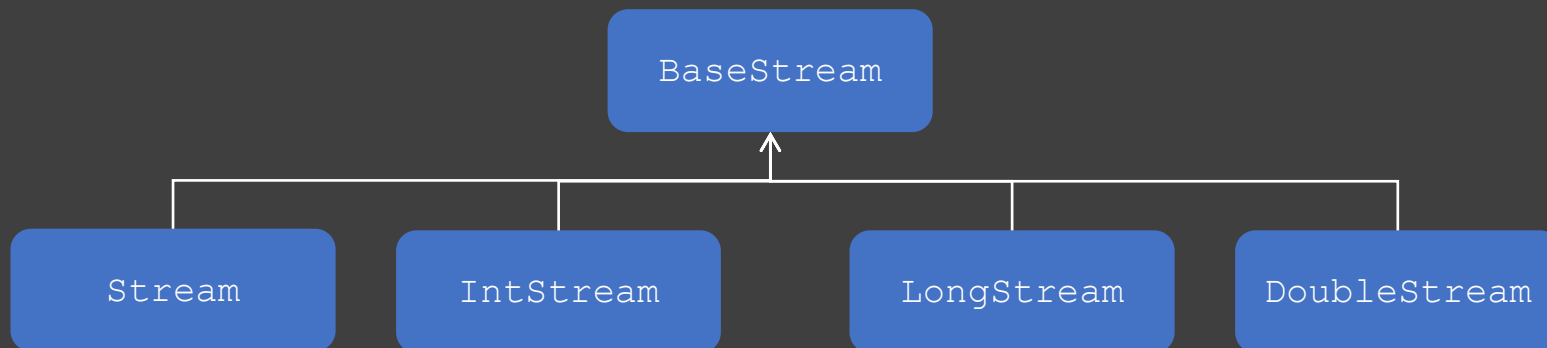
1. 스트림

- 반복자 스트림 예제(3).

```
public class MapAndReduceExample {  
    public static void main(String[] args) {  
        List<Student> studentList = Arrays.asList(  
            new Student("홍길동", 10),  
            new Student("신용권", 20),  
            new Student("유미선", 30)  
        );  
  
        double avg = studentList.stream()  
            //중간처리(학생 객체를 점수로 매핑)  
            .mapToInt(Student::getScore)  
            //최종 처리(평균 점수)  
            .average()  
            .getAsDouble();  
  
        System.out.println("평균 점수: " + avg);  
    }  
}
```

1. 스트림

- `java.util.stream` 패키지에는 스트림(stream) API 들이 포진하고 있다.



- `BaseStream` 인터페이스는 모든 스트림에서 사용할 수 있는 공통 메소드들이 정의되어 있을 뿐 코드에서 직접적으로 사용하지는 않는다.
- 결국 `BaseStream`을 상속하는 하위 메서드를 사용하게 되는데 객체 `Stream`은 객체 요소를 나머지는 `int`, `long`, `double` 요소를 처리하는 스트림이다.
- 이 스트림 인터페이스의 구현 객체는 다양한 소스로부터 얻을 수 있다.

1. 스트림

- 스트림은 주로 컬렉션과 배열에서 얻지만, 다음과 같은 소스로부터 스트림 구현 객체를 얻을 수도 있다.

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

1. 스트림

- 예제(1)

```
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; }  
}
```

```
public class FromCollectionExample {  
    public static void main(String[] args) {  
        List<Student> studentList = Arrays.asList(  
            new Student("홍길동", 10),  
            new Student("신용권", 20),  
            new Student("유미선", 30)  
        );  
  
        Stream<Student> stream = studentList.stream();  
        stream.forEach(s -> System.out.println(s.getName()));  
    }  
}
```


1. 스트림

- 예제(2)

```
public class FromArrayExample {  
    public static void main(String[] args) {  
        String[] strArray = { "홍길동", "신용권", "김미나"};  
        Stream<String> strStream = Arrays.stream(strArray);  
        strStream.forEach(a -> System.out.print(a + ","));  
        System.out.println();  
  
        int[] intArray = { 1, 2, 3, 4, 5 };  
        IntStream intStream = Arrays.stream(intArray);  
        intStream.forEach(a -> System.out.print(a + ","));  
        System.out.println();  
    }  
}
```

1. 스트림

- 예제(3)

```
public class FromIntRangeExample {  
    public static int sum;  
  
    public static void main(String[] args) {  
        IntStream stream = IntStream.rangeClosed(1, 100);  
        stream.forEach(a -> sum += a);  
        System.out.println("총합: " + sum);  
    }  
}
```

1. 스트림

- 예제(4)

```
public class FromFileContentExample {  
    public static void main(String[] args) throws IOException {  
        Path path = Paths.get("src/sec02/stream_kind/linedata.txt");  
        Stream<String> stream;  
  
        //Files.lines() 메소드 이용  
        stream = Files.lines(path, Charset.defaultCharset());  
        stream.forEach( System.out :: println );  
        stream.close();  
        System.out.println();  
  
        //BufferedReader의 lines() 메소드 이용  
        File file = path.toFile();  
        FileReader fileReader = new FileReader(file);  
        BufferedReader br = new BufferedReader(fileReader);  
        stream = br.lines();  
        stream.forEach( System.out :: println );  
        stream.close();  
    }  
}
```

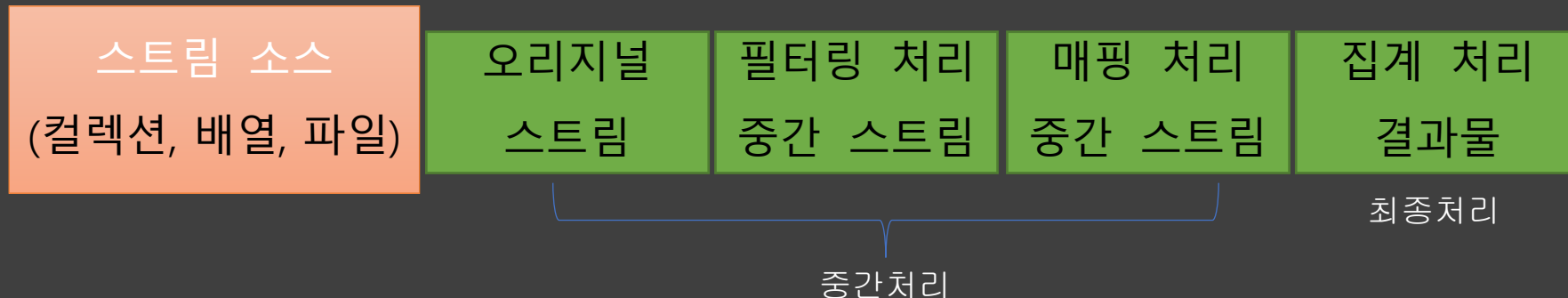
1. 스트림

- 예제(5)

```
public class FromDirectoryExample {  
    public static void main(String[] args) throws IOException {  
        Path path = Paths.get("C:/JavaProgramming/source");  
        Stream<Path> stream = Files.list(path);  
        stream.forEach( p -> System.out.println(p.getFileName()) );  
    }  
}
```

2. 스트림 파이프라인

- 스트림은 데이터의 필터링, 매핑, 정렬, 그룹핑 등의 중간 처리와 합계, 평균, 카운팅, 최대값, 최소값 등의 최종 처리를 파이프라인(Pipelines)으로 해결한다.
- 파이프라인은 여러 개의 스트림이 연결되어 있는 구조를 말한다.



- Stream 인터페이스에는 필터링, 매핑, 정렬 등의 많은 중간 처리 메서드가 있으며, 이 메서드들은 중간 처리된 스트림을 리턴한다.
- 위 스트림에서 다시 중간 처리 메소드를 호출해서 파이프라인을 형성하게 된다.

2. 스트림 파이프라인

- 스트림 파이프라인 예제.

```
public class Member {  
    public static int MALE = 0;  
    public static int FEMALE = 1;  
  
    private String name;  
    private int sex;  
    private int age;  
  
    public Member(String name, int sex, int age) {  
        this.name = name;  
        this.sex = sex;  
        this.age = age;  
    }  
  
    public int getSex() { return sex; }  
    public int getAge() { return age; }  
}
```

2. 스트림 파이프라인

- 스트림 파이프라인 예제.

```
public class StreamPipelinesExample {  
    public static void main(String[] args) {  
        List<Member> list = Arrays.asList(  
            new Member("홍길동", Member.MALE, 30),  
            new Member("김나리", Member.FEMALE, 20),  
            new Member("신용권", Member.MALE, 45),  
            new Member("박수미", Member.FEMALE, 27)  
        );  
  
        double ageAvg = list.stream() 오리지널 스트림  
            .filter(m -> m.getSex() == Member.MALE)  
            .mapToInt(Member::getAge)  
            .average() 최종 처리  
            .getAsDouble(); 중간처리 스트림  
  
        System.out.println("남자 평균 나이: " + ageAvg);  
    }  
}
```

2.1 스트림 파이프라인-중간처리(필터링)

- 필터링은 중간 처리 기능으로 요소를 걸러내는 기능을 하는 메서드이며 아래와 같은 메서드들이 있다.

리턴타입	메소드(매개변수)	설명
Stream IntStream LongStream DoubleStream	distinct()	중복제거
	filter(Predicate)	조건 필터링
	filter(IntPredicate)	
	filter(LongPredicate)	
	filter(DoublePredicate)	

2.1 스트림 파이프라인-중간처리(필터링)

- 필터링 예제

```
public class FilteringExample {  
    public static void main(String[] args) {  
        List<String> names = Arrays.asList("홍길동", "신용권", "감자바", "신용권", "신민철");  
  
        names.stream()  
            .distinct()  
            .forEach(n -> System.out.println(n));  
        System.out.println();  
  
        names.stream()  
            .filter(n -> n.startsWith("신"))  
            .forEach(n -> System.out.println(n));  
        System.out.println();  
  
        names.stream()  
            .distinct()  
            .filter(n -> n.startsWith("신"))  
            .forEach(n -> System.out.println(n));  
    }  
}
```

홍길동
신용권
감자바
신민철

신용권
신용권
신민철

신용권
신민철

2.1 스트림 파이프라인-중간처리(매핑)

- flatMapXXX() 메소드는 요소를 대체하는 복수 개의 요소들로 구성된 새로운 스트림을 리턴한다.
- flatMapXXX() 메소드의 종류는 다음과 같다.

리턴타입	메소드(매개 변수)	요소 -> 대체 요소
Stream<R>	flatMap(Function<T, Stream<R>>)	T -> Stream<R>
DoubleStream	flatMap(DoubleFunction<DoubleStream>)	double -> DoubleStream
IntStream	flatMap(IntFunction<IntStream>)	int -> IntStream
LongStream	flatMap(LongFunction<LongStream>)	long -> LongStream
DoubleStream	flatMapToDouble(Function<T, DoubleStream>)	T -> DoubleStream
IntStream	flatMapToInt(Function<T, IntStream>)	T -> IntStream
LongStream	flatMapToLong(Function<T, LongStream>)	T -> LongStream

2.1 스트림 파이프라인-중간처리(매핑)

- flatMapXXX() 메소드 예제.

```
public class FlatMapExample {  
    public static void main(String[] args) {  
        List<String> inputList1 = Arrays.asList("java8 lambda", "stream mapping");  
        inputList1.stream()  
            .flatMap(data -> Arrays.stream(data.split(" ")))  
            .forEach(word -> System.out.println(word));  
  
        System.out.println();  
  
        List<String> inputList2 = Arrays.asList("10, 20, 30", "40, 50, 60");  
        inputList2.stream()  
            .flatMapToInt(data -> {  
                String[] strArr = data.split(",");  
                int[] intArr = new int[strArr.length];  
                for(int i=0; i<strArr.length; i++) {  
                    intArr[i] = Integer.parseInt(strArr[i].trim());  
                }  
                return Arrays.stream(intArr);  
            })  
            .forEach(number -> System.out.println(number));  
    }  
}
```

2.1 스트림 파이프라인-중간처리(매핑)

- MapXXX() 메소드는 요소를 대체하는 요소로 구성된 새로운 스트림을 리턴한다.
- MapXXX() 메소드의 종류는 다음과 같다.

리턴타입	메소드(매개 변수)	요소 -> 대체 요소
Stream<R>	map(Function<T, R>)	T -> R
DoubleStream	mapToDouble(ToDoubleFunction<T>)	T -> double
IntStream	mapToInt(ToIntFunction<T>)	T -> Int
LongStream	mapToLong(ToLongFunction<T>)	T -> Long
DoubleStream	map(DoubleUnaryOperator)	double -> double
IntStream	mapToInt(DoubleToIntFunction)	double -> Int
LongStream	mapToLong(DoubleToLongFunction)	double -> long
Stream<U>	mapToObj(DoubleFunction<U>)	double -> U

2.1 스트림 파이프라인-중간처리(매핑)

- MapXXX() 메소드의 종류(계속)

리턴타입	메소드(매개 변수)	요소 -> 대체 요소
IntStream	map(IntUnaryOperator)	Int -> Int
DoubleStream	mapToDouble(IntToDoubleFunction)	Int -> double
LongStream	mapToLong(IntToLongFunction)	Int -> long
Stream<U>	mapToObj(IntFunction<U>)	Int -> U
LongStream	map(LongUnaryOperator)	long -> long
DoubleStream	mapToInt(LongToDoubleFunction)	long -> double
IntStream	mapToLong(LongToIntFunction)	long -> Int
Stream<U>	mapToObj(LongFunction<U>)	long -> U

2.1 스트림 파이프라인-중간처리(매핑)

- MapXXX() 메소드 예제

```
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; }  
}
```

```
public class MapExample {  
    public static void main(String[] args) {  
        List<Student> studentList = Arrays.asList(  
            new Student("홍길동", 10),  
            new Student("신용권", 20),  
            new Student("유미선", 30)  
        );  
  
        studentList.stream()  
            .mapToInt(Student :: getScore)  
            .forEach(score -> System.out.println(score));  
    }  
}
```

2.1 스트림 파이프라인-중간처리(매핑)

- `asDoubleStream()` 메소드는 `IntStream`의 `int`요소 또는 `LongStream`의 `long` 요소를 `double` 요소로 타입 변환해서 `DoubleStream`을 생성한다.
- `asLongStream()` 메소드는 `IntStream`의 `int`요소를 `long`요소로 타입 변환해서 `LongStream`을 생성한다.
- `boxed()` 메소드는 `int`, `long`, `double` 요소를 `Integer`, `Long`, `Double` 요소로 박싱해서 `Stream`을 생성한다.

리턴타입	메소드(매개 변수)	요소 -> 대체 요소
<code>DoubleStream</code>	<code>asDoubleStream()</code>	<code>int -> Double</code> or <code>long -> Double</code>
<code>Long</code>	<code>asLongStream()</code>	<code>int -> long</code>
<code>Stream<Integer></code> <code>Stream<Long></code> <code>Stream<Double></code>	<code>boxed()</code>	<code>int -> Integer</code> <code>long -> Long</code> <code>double -> Double</code>

2.1 스트림 파이프라인-중간처리(매핑)

- 예제

```
public class AsDoubleStreamAndBoxedExample {  
    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()));  
    }  
}
```


2.1 스트림 파이프라인-중간처리(정렬)

- 스트림은 요소가 최종 처리되기 전에 중간 단계에서 요소를 정렬해서 최종 처리 순서를 변경할 수 있다.

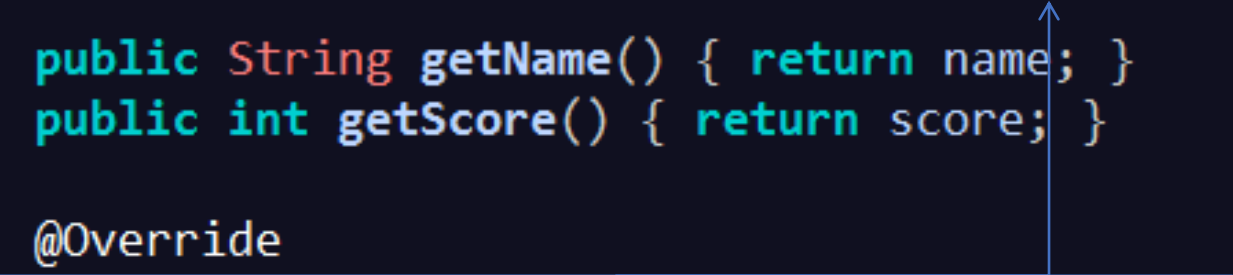
리턴타입	메소드(매개 변수)	요소 -> 대체 요소
Stream<T>	sorted()	객체를 Comparable 구현 방법에 따라 정렬
Stream<T>	sorted(Comparator<T>)	객체를 주어진 Comparator에 따라 정렬
DoubleStream	sorted()	double 요소를 오름차순으로 정렬
IntStream	sorted()	int 요소를 오름차순으로 정렬
LongStream	sorted()	long 요소를 오름차순으로 정렬

2.1 스트림 파이프라인-중간처리(정렬)

- 정렬 예제

```
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);  
    }  
}
```

score < o.score : 음수 리턴
score == o.score : 0리턴
score > o.score : 양수 리턴



2.1 스트림 파이프라인-중간처리(정렬)

- 정렬 예제

```
public class SortingExample {  
    public static void main(String[] args) {  
        //숫자 요소일 경우  
        IntStream intStream = Arrays.stream(new int[] {5, 3, 2, 1, 4});  
        intStream  
            .sorted()  
            .forEach(n -> System.out.print(n + ","));  
        System.out.println();  
  
        //객체 요소일 경우  
        List<Student> studentList = Arrays.asList(  
            new Student("홍길동", 30),  
            new Student("신용권", 10),  
            new Student("유미선", 20)  
        );  
  
        studentList.stream()  
            .sorted()  
            .forEach(s -> System.out.print(s.getScore() + ","));  
        System.out.println();  
  
        studentList.stream()  
            .sorted( Comparator.reverseOrder() )  
            .forEach(s -> System.out.print(s.getScore() + ","));  
    }  
}
```

2.1 스트림 파이프라인-중간처리(반복)

- 반복(looping)은 요소 전체를 반복하는 것을 말한다.
- 반복은 두가지의 메서드를 가진다
- `peek()`는 중간처리 단계에서 전체 요소를 반복하면서 추가적인 작업을 하기 위해 사용한다. 최종 처리 메소드가 실행되지 않으면 지연되기 때문에 반드시 최종 처리 메소드가 호출되어야 동작한다.

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

- 하지만 `forEach()`는 최종 처리 메서드이기 때문에 파이프라인 마지막에 반복하면서 요소를 하나씩 처리한다.
- `forEach()` 메소드는 최종 처리 메소드이므로 후에 `sum()`과 같은 다른 최종 메소드를 호출하면 안된다.

2.1 스트림 파이프라인-중간처리(반복)

- 반복 예제

```
public class LoopingExample {  
    public static void main(String[] args) {  
        int[] intArr = { 1, 2, 3, 4, 5 };  
  
        System.out.println("[peek()를 마지막에 호출한 경우]");  
        Arrays.stream(intArr)  
            .filter(a -> a%2==0)  
            .peek(n -> System.out.println(n));    //동작하지 않음  
  
        System.out.println("[최종 처리 메소드를 마지막에 호출한 경우]");  
        int total = Arrays.stream(intArr)  
            .filter(a -> a%2==0)  
            .peek(n -> System.out.println(n))    //동작함  
            .sum();  
        System.out.println("총합: " + total);  
  
        System.out.println("[forEach()를 마지막에 호출한 경우]");  
        Arrays.stream(intArr)  
            .filter(a -> a%2==0)  
            .forEach(n -> System.out.println(n)); //동작함  
    }  
}
```

2.2 스트림 파이프라인-최종처리(매칭)

- 스트림 클래스는 최종 처리 단계에서 요소들이 특정 조건에 만족하는지 조사할 수 있도록 세 가지 매칭 메소드를 제공하고 있다.
- `allMatch()` 메소드는 모든 요소들이 매개값으로 주어진 `Predicate`의 조건을 만족하는지 조사하며, `anyMatch()` 메소드는 최소한 한 개의 요소가, 그리고 `noneMatch()`는 모든 요소들이 조건을 만족하지 않는지 조사한다.

리턴타입	메소드(매개 변수)	제공 인터페이스
boolean	<code>allMatch(Predicate<T> predicate)</code> <code>anyMatch(Predicate<T> predicate)</code> <code>noneMatch(Predicate<T> predicate)</code>	Stream
boolean	<code>allMatch(IntPredicate predicate)</code> <code>anyMatch(IntPredicate predicate)</code> <code>noneMatch(IntPredicate predicate)</code>	IntStream
boolean	<code>allMatch(LongPredicate predicate)</code> <code>anyMatch(LongPredicate predicate)</code> <code>noneMatch(LongPredicate predicate)</code>	LongStream
boolean	<code>allMatch(DoublePredicate predicate)</code> <code>anyMatch(DoublePredicate predicate)</code> <code>noneMatch(DoublePredicate predicate)</code>	DoubleStream

2.2 스트림 파이프라인-최종처리(매칭)

- 매칭 스트림 예제

```
public class MatchExample {  
    public static void main(String[] args) {  
        int[] intArr = { 2, 4, 6 };  
  
        boolean result = Arrays.stream(intArr)  
            .allMatch(a -> a%2==0);  
        System.out.println("모두 2의 배수인가? " + result);  
  
        result = Arrays.stream(intArr)  
            .anyMatch(a -> a%3==0);  
        System.out.println("하나라도 3의 배수가 있는가? " + result);  
  
        result = Arrays.stream(intArr)  
            .noneMatch(a -> a%3==0);  
        System.out.println("3의 배수가 없는가? " + result);  
    }  
}
```

terminated: MatchExample [java Application] 1.1

모두 2의 배수인가? true

하나라도 3의 배수가 있는가? true

3의 배수가 없는가? false

2.2.1 스트림 파이프라인-최종처리(기본 집계)

- 집계(Aggregate)는 최종 처리 기능으로 요소들을 처리해서 카운팅, 합계, 평균값, 최대값, 최소값 등과 같이 하나의 값으로 산출하는 것을 말한다.
- 스트림에서 제공하는 기본 집계 메소드는 다음과 같다.

리턴타입	메소드(매개 변수)	설명
long	count()	요소 개수
OptionalXXX	findFirst()	첫 번째 요소
Optional<T> OptionalXXX	max(Comparator<T>) max()	최대 요소
Optional<T> OptionalXXX	min(Comparator<T>) min()	최소 요소
OptionalDouble	average()	요소 평균
int, long, double	sum()	요소 총합

2.2.1 스트림 파이프라인-최종처리(기본 집계)

- 집계(Aggregate) 예제.

```
public class AggregateExample {  
    public static void main(String[] args) {  
        long count = Arrays.stream(new int[] {1, 2, 3, 4, 5})  
            .filter(n -> n%2==0)  
            .count();  
        System.out.println("2의 배수 개수: " + count);  
  
        long sum = Arrays.stream(new int[] {1, 2, 3, 4, 5})  
            .filter(n -> n%2==0)  
            .sum();  
        System.out.println("2의 배수의 합: " + sum);  
  
        double avg = Arrays.stream(new int[] {1, 2, 3, 4, 5})  
            .filter(n -> n%2==0)  
            .average()  
            .getAsDouble();  
        System.out.println("2의 배수의 평균: " + avg);  
    }  
}
```

2.2.1 스트림 파이프라인-최종처리(기본 집계)

- 집계(Aggregate) 예제.

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

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

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

2.2.2 스트림 파이프라인-최종처리(Optional 클래스)

- Optional 클래스는 단순히 집계 값만 저장하는 것이 아니라, 집계 값이 존재하지 않을 경우 디폴트 값을 설정할 수도 있고, 집계 값을 처리하는 Consumer도 등록할 수 있다.

리턴타입	메소드(매개 변수)	설명
boolean	isPresent()	값이 저장되어 있는지 여부
T double int long	orElse(T) orElse(double) orElse(int) orElse(long)	값이 저장되어 있지 않을 경우 디폴트 값 지정
void	ifPresent(Consumer) ifPresent(DoubleConsumer) ifPresent(IntConsumer) ifPresent(LongConsumer)	값이 저장되어 있을 경우 Consumer에서 처리

2.2.2 스트림 파이프라인-최종처리(Optional 클래스)

- Optional 클래스 예제

```
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();  
        */  
  
        //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");  
        }  
    }  
}
```

2.2.2 스트림 파이프라인-최종처리(Optional 클래스)

- Optional 클래스 예제

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

//방법3
list.stream()
    .mapToInt(Integer :: intValue)
    .average()
    .ifPresent(a -> System.out.println("방법3_평균: " + a));
}
}
```

2.2.2 스트림 파이프라인-최종처리(커스텀 집계)

- 스트림은 기본 집계 외에 다양한 집계 결과물을 만들 수 있도록 reduce()라는 메소드도 제공한다..

인터페이스	리턴타입	메소드(매개변수)
Stream	Optional<T>	reduce(BinaryOptional<T>)
	T	reduce(T identity, BinaryOptional<T>, accumulator)
IntStream	OptionalInt	reduce(IntBinaryOptional op)
	int	reduce(int identity, IntBinaryOptional op)
LongStream	OptionalLong	reduce(LongBinaryOptional op)
	long	reduce(long identity, LongBinaryOptional op)
DoubleStream	OptionalDouble	reduce(DoubleBinaryOptional op)
	double	reduce(double identity, DoubleBinaryOptional op)

2.2.2 스트림 파이프라인-최종처리(커스텀 집계)

- Reduce 예제

```
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; }  
}
```

2.2.2 스트림 파이프라인-최종처리(커스텀 집계)

- Reduce 예제

```
public class ReductionExample {  
    public static void main(String[] args) {  
        List<Student> studentList = Arrays.asList(  
            new Student("홍길동", 92),  
            new Student("신용권", 95),  
            new Student("감자바", 88)  
        );  
  
        int sum1 = studentList.stream()  
            .mapToInt(Student :: getScore)  
            .sum();  
  
        int sum2 = studentList.stream()  
            .map(Student :: getScore)  
            .reduce((a, b) -> a+b)  
            .get();  
  
        int sum3 = studentList.stream()  
            .map(Student :: getScore)  
            .reduce(0, (a, b) -> a+b);  
  
        System.out.println("sum1: " + sum1);  
        System.out.println("sum2: " + sum2);  
        System.out.println("sum3: " + sum3);  
    }  
}
```