

Lambda

1. Lambda

❖ 자바 1.7부터 함수적 프로그래밍을 위해 람다식 지원

- ✓ 메소드의 매개변수로 메소드의 내용을 전달 할 수 있도록 하기 위해서 지원
- ✓ 람다식(Lambda Expressions)을 언어 차원에서 제공
- ✓ 람다 계산법에서 사용된 식을 프로그래밍 언어에 접목
- ✓ 익명 함수(anonymous function)을 생성하기 위한 식

❖ 자바에서 람다식을 수용한 이유

- ✓ 코드가 간결
- ✓ 컬렉션 요소(대용량 데이터)에 필터링 또는 매핑에 도입해서 집계를 쉽게 사용 할 수 있도록 함

Runnable runnable = new Runnable(){
...}; 의 익명 객체 생성을 아래처럼 변경이 가능

Runnable runnable = () -> { ... }; ●----- 람다식

Person

```
public class Person {  
    private String name;  
    private int score;  
  
    public Person() {  
        super();  
    }  
  
    public Person(String name, int score) {  
        super();  
        this.name = name;  
        this.score = score;  
    }  
  
    public String getName() {  
        return name;  
    }  
    public void setName(String name) {  
        this.name = name;  
    }  
}
```

Person

```
    public int getScore() {  
        return score;  
    }  
    public void setScore(int score) {  
        this.score = score;  
    }  
  
    @Override  
    public String toString() {  
        return "Person [name=" + name + ", score=" + score + "];"  
    }  
}
```

PersonLambdaMain

```
package chap15;

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

public class PersonLambdaMain {

    public static void main(String[] args) {
        List<Person> singerList = new ArrayList<>();
        singerList.add(new Person("수지", 100));
        singerList.add(new Person("아이린", 90));
        singerList.add(new Person("제니", 92));
        singerList.add(new Person("쯔위", 88));
        singerList.add(new Person("모모", 98));
```

PersonLambdaMain

개변수로 대입

//1.7이전 버전에서의 List 정렬
//메소드를 매개변수로 사용할 수 없기 때문에 메소드를 구현한 객체를 매

```
Collections.sort(singerList, new Comparator<Person>() {  
    @Override  
    public int compare(Person singer1, Person singer2) {  
        return singer1.getScore() - singer2.getScore();  
    }  
});
```

```
for(Person person : singerList) {  
    System.out.println(person);  
}
```

```
System.out.println("=====");  
//람다를 이용한 정렬  
Collections.sort(singerList,  
    (singer1, singer2) -> singer1.getScore()-
```

singer2.getScore());

```
for(Person person : singerList) {  
    System.out.println(person);  
}
```

```
}
```

```
}
```

1. Lambda

❖ 함수적 스타일의 람다식 작성법

```
(타입 매개변수, ...) -> { 실행문; ... }
```

```
(int a) -> { System.out.println(a); }
```

- ✓ 매개변수 타입은 실행할 때 대입하는 값에 따라 자동 인식하므로 생략 가능
- ✓ 하나의 매개변수만 있을 경우에는 괄호 () 생략 가능
- ✓ 하나의 실행하는 문장만 있다면 중괄호 { } 생략 가능
- ✓ 매개변수 없다면 괄호 () 생략 불가
- ✓ 리턴 값이 있는 경우 return 문 사용
- ✓ 중괄호 { }에 return 문만 있을 경우, 중괄호 생략 가능

1. Lambda

❖ 랴다가 함수적 프로그래밍이지만 자바는 메소드를 별도로 호출할 수 없기 때문에 참조형 변수를 통해서만 호출

인터페이스 변수 = 랴다식;

❖ 타겟 타입(target type)

- ✓ 랴다식이 대입되는 인터페이스
- ✓ 익명 구현 객체를 만들 때 사용할 인터페이스

❖ 함수적 인터페이스(functional interface) - 1.8

- ✓ 하나의 추상 메소드만 선언된 인터페이스
- ✓ @FunctionalInterface 어노테이션을 이용해서 설정
- ✓ 하나의 추상 메소드 만을 가지는지 컴파일러가 체크
- ✓ 두 개 이상의 추상 메소드가 선언되어 있으면 컴파일 오류 발생

매개변수가 없는 람다

//매개변수가 없고 리턴타입이 없는 메소드를 소유한 인터페이스

```
interface NoArgNoReturn{  
    public void method1();  
}
```

```
public class LambdaMain {
```

```
    public static void main(String[] args) {
```

```
        //매개변수가 없고 리턴도 없는 인터페이스 활용
```

```
        NoArgNoReturn ob1 = () -> {System.out.println("매개변수가 없고 리턴  
도 없는 람다");};
```

```
        ob1.method1();
```

```
    }
```

```
}
```

매개변수가 있는 람다

//매개변수가 있고 리턴 타입이 없는 경우

//원본에 작업을 해서 원본을 변환시키거나 출력하는 인터페이스

```
interface ArgNoReturn{
```

```
    public void method2(int x);
```

```
}
```

```
public class LambdaMain {
```

```
    public static void main(String[] args) {
```

//매개변수가 있는 경우 - 매개변수의 자료형은 생략이 가능, 매개변수가
1개인 경우는 ()로 감싸지 않아도 됩니다.

```
        ArgNoReturn ob2 = (int x) ->{System.out.println(x + 10);};
```

```
        ob2.method2(100);
```

```
    }
```

```
}
```

리턴이 있는 람다

//매개변수는 없고 리턴 타입만 있는 경우 : 거의 없는 경우

```
interface NoArgReturn{  
    public double method3();  
}
```

//매개변수가 있고 리턴타입이 있는 경우 - 가장 많은 경우

```
interface ArgReturn{  
    public int method4(String str);  
}
```

```
public class LambdaMain {
```

```
    public static void main(String[] args) {
```

//매개변수는 없고 리턴만 있는 경우 - 거의 없음

```
        NoArgReturn ob3 = () ->{return 10.3;};  
        double d = ob3.method3();  
        System.out.println(d);
```

//매개변수가 있고 리턴이 있는 경우 - 데이터를 가공해서 리턴하는 함수

```
        ArgReturn ob4 = (str) ->{return Integer.parseInt(str);};  
        int i = ob4.method4("123219");  
        System.out.println(i);
```

```
    }
```

```
}
```

람다를 이용한 스레드 구현

- ❖ JDK에서 제공하는 한 개의 추상 메소드를 가진 모든 인터페이스들은 모두 람다식을 이용해서 Anonymous 객체로 구현하는 것이 가능
- ❖ 람다도 Anonymous 객체로 취급되기 때문에 람다를 포함하고 있는 클래스의 인스턴스 변수에 대입이 가능
- ❖ 다른 점은 this를 사용하는 경우로 Anonymous 객체에서는 this가 자기 자신을 나타내지만 람다에서는 포함하고 있는 객체의 참조
- ❖ 람다는 대부분 메소드 안에 구현되므로 메소드의 로컬 변수에는 직접 접근이 안되고 final 특성을 갖는 변수에만 접근이 가능

람다를 이용한 스레드 구현

```
public class ThreadRambda {  
    public static void main(String[] args) {  
        Runnable r = ()->{  
            try{  
                for(int i=0; i<10; i++){  
                    System.out.println(i);  
                    Thread.sleep(1000);  
                }  
            }  
            catch(Exception e){  
                e.printStackTrace();  
            }  
        };  
        Thread th = new Thread(r);  
        th.start();  
    }  
}
```

기존 메소드 넘기기

- ❖ 1.8에서부터는 메소드의 매개변수로 기존 메소드를 전달 할 수 있음
- ❖ 메소드 참조 방법
 - ✓ 인스턴스 메소드: 인스턴스이름::메소드이름
 - ✓ 클래스 내에서 자신의 메소드: this::메소드이름
 - ✓ static 메소드: 클래스이름::메소드이름

기존 메소드 넘기기

```
import java.util.ArrayList;
import java.util.List;
public class MethodArg {
    public static void main(String[] args) {
        List<Person> singerList = new ArrayList<>();
        singerList.add(new Person("수지", 100));
        singerList.add(new Person("아이린", 90));
        singerList.add(new Person("제니", 92));
        singerList.add(new Person("쯔위", 88));
        singerList.add(new Person("모모", 98));

        singerList.forEach(System.out::println);
    }
}
```

2. 표준 API 란다 인터페이스

- ❖ 자주 사용되는 함수적 인터페이스를 `java.util.function` 패키지에서 제공
- ❖ 별도의 인터페이스를 생성하지 않고 란다식을 사용할 수 있도록 하기 위해서 제공
- ❖ 1.8 버전부터 추가되거나 변경된 API에서 이 함수적 인터페이스들을 매개변수 타입으로 사용

종류	추상 메소드
Consumer	매개변수는 있고 리턴 값은 없는 메소드 소유
Supplier	매개변수는 없고 리턴 값만 있는 메소드 소유
Function	매개변수가 있고 리턴 값도 있음
Operator	매개변수가 있고 리턴 값도 있음 매개변수를 연산해서 결과를 리턴
Predicate	매개변수가 있고 리턴 값이 boolean 매개변수를 조사해서 boolean으로 리턴

2.1 Consumer

❖ 매개변수의 타입과 수에 따라서 아래와 같은 Consumer 제공

인터페이스명	추상 메소드	설명
Consumer<T>	void accept(T t)	객체 T를 받아 소비
BiConsumer<T,U>	void accept(T t, U u)	객체 T와 U를 받아 소비
DoubleConsumer	void accept(double value)	double 값을 받아 소비
IntConsumer	void accept(int value)	int 값을 받아 소비
LongConsumer	void accept(long value)	long 값을 받아 소비
ObjDoubleConsumer<T>	void accept(T t, double value)	객체 T와 double 값을 받아 소비
ObjIntConsumer<T>	void accept(T t, int value)	객체 T와 int 값을 받아 소비
ObjLongConsumer<T>	void accept(T t, long value)	객체 T와 long 값을 받아 소비

2.1 Consumer

```
import java.util.function.*;

public class ConsumerMain {

    public static void main(String[] args) {
        Consumer<String> consumer =
            t -> System.out.println(t + "1.8에서 가능");
        consumer.accept("JDK");

        BiConsumer<String, String> bigConsumer =
            (t, u) -> System.out.println(t + u);
        bigConsumer.accept("JDK", "1.8에서 가능");
    }
}
```

2.2 Supplier

❖ 매개변수가 없고 리턴값이 있는 get자료형() 메소드를 소유

인터페이스명	추상 메소드	설명
Supplier<T>	T get()	객체를 리턴
BooleanSupplier	boolean getAsBoolean()	boolean 값을 리턴
DoubleSupplier	double getAsDouble()	double 값을 리턴
IntSupplier	int getAsInt()	int 값을 리턴
LongSupplier	long getAsLong()	long 값을 리턴

2.2 Supplier

```
import java.util.*;
import java.util.function.*;

public class SupplierMain {

    public static void main(String[] args) {
        IntSupplier dice = () -> {
            Random r = new Random();
            int num = r.nextInt(6)+1;
            return num;
        };

        int val = dice.getAsInt();
        System.out.println("주사위: " + val);
    }
}
```

2.3 Function

- ❖ 매개변수가 있고 리턴 값이 있는 get자료형() 메소드를 소유
- ❖ 데이터를 변환을 해서 리턴하는 역할을 수행

인터페이스명	추상 메소드	설명
Function<T,R>	R apply(T t)	객체 T를 객체 R로 매핑
BiFunction<T,U,R>	R apply(T t, U u)	객체 T와 U를 객체 R로 매핑
DoubleFunction<R>	R apply(double value)	double을 객체 R로 매핑
IntFunction<R>	R apply(int value)	int를 객체 R로 매핑
IntToDoubleFunction	double applyAsDouble(int value)	int를 double로 매핑
IntToLongFunction	long applyAsLong(int value)	int를 long으로 매핑
LongToDoubleFunction	double applyAsDouble(long value)	long을 double로 매핑
LongToIntFunction	int applyAsInt(long value)	long을 int로 매핑
ToDoubleBiFunction<T,U>	double applyAsDouble(T t, U u)	객체 T와 U를 double로 매핑
ToDoubleFunction<T>	double applyAsDouble(T value)	객체 T를 double로 매핑
ToIntBiFunction<T,U>	int applyAsInt(T t, U u)	객체 T와 U를 int로 매핑
ToIntFunction<T>	int applyAsInt(T value)	객체 T를 int로 매핑
ToLongBiFunction<T,U>	long applyAsLong(T t, u)	객체 T와 U를 long으로 매핑
ToLongFunction<T>	long applyAsLong(T value)	객체 T를 long으로 매핑

2.3 Function

❖ 매개변수로 리턴 값이 있는 메소드를 대입하면 메소드를 선언할 때 사용한 객체가 메소드의 실행 결과로 대체되는 인터페이스

```
double avg(TolntFunction<Student> function) {  
    int sum = 0;  
    for (Student student : list) {  
        sum += function.applyAsInt(student);  
    }  
    double avg = (double) sum / list.size();  
    return avg;  
}
```

❖ 위와 같은 메소드를 생성하면 `function.applyAsInt(student)`의 값은 `student`는 메소드를 호출할 때 매개변수로 대입된 메소드의 리턴 값으로 대체

2.3 Function

```
import java.util.*;
import java.util.function.*;

public class FunctionMain {

    private static List<Person> list =
        Arrays.asList(
            new Person("김좌진", 90),
            new Person("홍범도", 95));

    public static void strPrint(Function<Person, String> function) {
        for(Person person : list) {
            System.out.println(function.apply(person));
        }
        System.out.println();
    }

    public static void main(String[] args) {
        strPrint( t->t.getName());
    }
}
```

2.4 Operator

- ❖ 매개변수가 있고 리턴값이 있는 apply자료형() 메소드를 소유
- ❖ 연산을 수행한 후 리턴하는 역할을 제공

인터페이스명	추상 메소드	설명
BinaryOperator<T>	BiFunction<T,U,R>의 하위 인터페이스	T 와 U 를 연산한 후 R 리턴
UnaryOperator<T>	Function<T,R>의 하위 인터페이스	T 를 연산한 후 R 리턴
DoubleBinaryOperator	double applyAsDouble(double, double)	두 개의 double 연산
DoubleUnaryOperator	double applyAsDouble(double)	한 개의 double 연산
IntBinaryOperator	int applyAsInt(int, int)	두 개의 int 연산
IntUnaryOperator	int applyAsInt(int)	한 개의 int 연산
LongBinaryOperator	long applyAsLong(long, long)	두 개의 long 연산
LongUnaryOperator	long applyAsLong(long)	한 개의 long 연산

2.4 Operator

```
import java.util.function.*;

public class OperatorMain {
    public static int max(int[] ar, IntBinaryOperator operator) {
        int result = ar[0];
        for (int score : ar) {
            result = operator.applyAsInt(result, score);
        }
        return result;
    }
    public static void main(String[] args) {
        int[] scores = { 92, 95, 87 };
        // 최대값 얻기
        int max = max(scores, (a, b) -> {
            if (a >= b)
                return a;
            else
                return b;
        });
        System.out.println("최대값: " + max);
    }
}
```

2.5 Predicate

❖ 매개변수가 있고 boolean으로 리턴하는 test() 메소드를 소유

인터페이스명	추상 메소드	설명
Predicate<T>	boolean test(T t)	객체 T를 조사
BiPredicate<T,U>	boolean test(T t, U u)	객체 T와 U를 비교 조사
DoublePredicate	boolean test(double value)	double 값을 조사
IntPredicate	boolean test(int value)	int 값을 조사
LongPredicate	boolean test(long value)	long 값을 조사

2.5 Predicate

```
import java.util.*;
import java.util.function.*;

public class PredicateMain {
    public static int avg(List<Person> list, Predicate<Person> predicate) {
        int cnt = 0;
        for (Person person : list) {
            if (predicate.test(person)) {
                cnt++;
            }
        }
        return cnt;
    }

    public static void main(String[] args) {
        List<Person> list = Arrays.asList(new Person("김좌진", 87), new
        Person("윤봉길", 90), new Person("유관순", 95),
            new Person("홍범도", 89));
        int count = avg(list, t -> t.getScore() >= 90);
        System.out.println("90 이상인 인원 수: " + count);
    }
}
```

스트림

1. 스트림

- ❖ 동일한 자료형의 데이터를 여러 개 저장 할 때 Collection이나 배열을 사용
- ❖ 배열이나 Collection의 데이터를 모두 접근하기 위해서는 for 나 Iterator를 이용하는데 이러한 방식으로의 접근은 코드가 길어지게 되고 재 사용성이 떨어짐
- ❖ Collection 이나 배열에서 동일한 작업을 수행하는 메소드가 각각 따로 정의되어 있어서 자료 구조에 따라 메소드를 다르게 호출(get(), sort())
- ❖ 스트림은 JDK 1.8에서 추가된 요소로 컬렉션의 저장 요소를 하나씩 참조하여 람다식으로 처리 할 수 있는 반복자
- ❖ Iterator는 컬렉션의 데이터를 접근할 수 있도록 포인터만 제공하지만 스트림은 데이터를 가지고 작업을 수행 할 수 있도록 해줌

1. 스트림

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

public class StreamMain {
    public static void main(String[] args) {
        List<String> list = Arrays.asList("김좌진", "홍범도", "윤동주");

        // for 문 이용
        System.out.println("일반 for 문 이용");
        for(int i=0; i<list.size(); i++){
            String name = list.get(i);
            System.out.println(name);
        }
        System.out.println("=====");
        // Iterator 이용
        System.out.println("Iterator 이용");
        Iterator<String> iterator = list.iterator();
        while (iterator.hasNext()) {
            String name = iterator.next();
            System.out.println(name);
        }
        System.out.println("=====");
    }
}
```

1. 스트림

```
// 빠른 열거 이용
System.out.println("빠른 열거 이용");
for (String imsi : list) {
    System.out.println(imsi);
}
System.out.println("=====");
// Stream 이용
System.out.println("Stream 이용");
Stream<String> stream = list.stream();
stream.forEach(name -> System.out.println(name));
}
}
```

1. 스트림

❖ 아래와 같은 문자열 배열과 리스트가 있는 경우

```
String [] strAr = {"abc", "def", "ghi"};
```

```
List<String>strList = Arrays.asList(strArr);
```

❖ 2개의 데이터 소스를 기반으로 스트림 생성

```
Stream<String>strStream1 = Arrays.stream(strAr);
```

```
Stream<String>strStream2 = strList.stream();
```

❖ 데이터를 정렬한 후 출력

```
strStream1.sorted().forEach(System.out::println);
```

```
strStream2.sorted().forEach(System.out::println);
```


1. 스트림

❖ 스트림은 데이터 소스를 변경하지 않음

- ✓ 스트림은 데이터를 읽기만 할 뿐 데이터 소스를 변경하지 않음
- ✓ 중간 결과를 배열이나 리스트로 저장할 수 있음

```
List<String> sortedList = strStream2.sorted().collect(Collectors.toList());
```

❖ 스트림은 일회용: 스트림이나 Iterator는 일회용이라서 컬렉션의 요소를 전부 읽고 나면 다시 사용할 수 없어서 재생성 해야 함

❖ 스트림은 작업을 내부적으로 반복해서 처리: 반복문을 메소드 내부에 숨기는 것이 가능

❖ 다양한 연산 제공: 스트림은 중간 처리(매핑, 필터링, 정렬)와 최종 처리(반복, 카운팅, 평균, 합계)를 할 수 있는 연산을 제공

1. 스트림

❖ 스트림의 연산은 지연된 연산

스트림의 중간 연산은 중간에 수행되는 것이 아니고 최종 연산이 수행되기 전까지는 수행되지 않음

❖ Stream<Integer> & IntStream

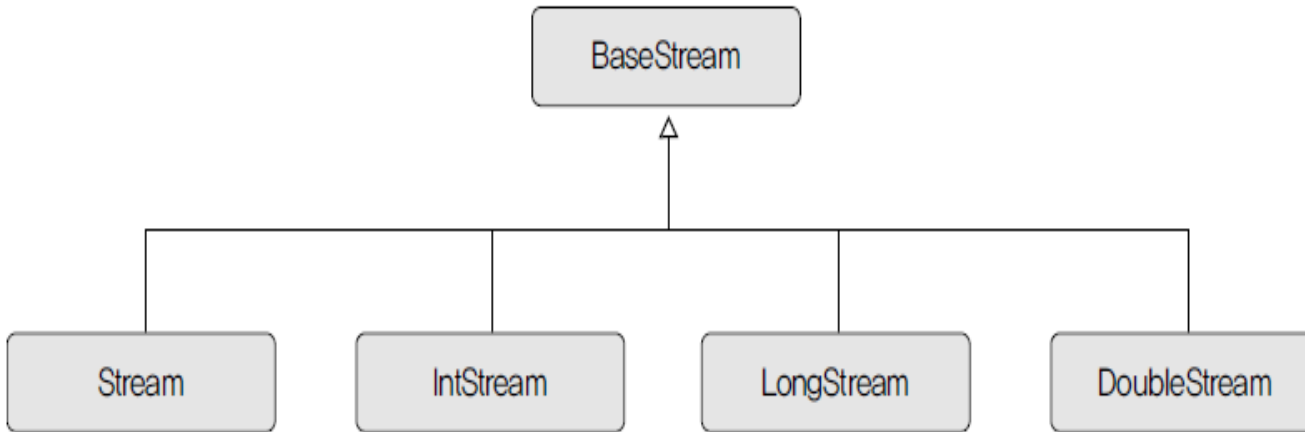
정수 스트림의 경우 Stream<Integer>를 이용해서 생성이 가능한데 오토박싱 & 언박싱으로 인한 불필요한 자원의 소모를 방지하기 위해서 기본 자료형의 스트림을 제공

❖ 병렬 스트림

fork & join 프레임워크를 이용해서 병렬처리를 할 수 있지만 스트림을 이용하면 병렬처리를 훨씬 쉽게 구현

2. 스트림의 종류

❖ java.util.stream 패키지는 스트림 API 들이 소속



2. 스트림의 종류

❖ 스트림 인터페이스의 구현 객체

- ✓ 주로 컬렉션과 배열에서 얻음
- ✓ 소스로부터 스트림 구현 객체 얻는 방법

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

2. 스트림의 종류

❖ 스트림 생성

```
import java.nio.file.Files;
import java.nio.file.Path;
import java.nio.file.Paths;
import java.util.Arrays;
import java.util.List;
import java.util.Random;
import java.util.stream.IntStream;
import java.util.stream.Stream;
```

//컬렉션의 다양한 접근

```
public class StreamCreate {
```

```
    public static void main(String[] args) {
        String[] ar = { "김좌진", "홍범도", "유관순" };
        System.out.println("====배열로부터 스트림 생성====");
        Stream<String> arStream = Arrays.stream(ar);
        arStream.forEach(e -> System.out.println(e));
        System.out.println("=====");

        List<String> list = Arrays.asList("김좌진", "홍범도", "유관순");
        System.out.println("====리스트로부터 스트림 생성====");
        Stream<String> listStream = list.stream();
        listStream.forEach(e -> System.out.println(e));
        System.out.println("=====");
    }
}
```

2. 스트림의 종류

❖ 스트림 생성

```
System.out.println("====정수 범위로부터 스트림 생성====");
IntStream intStream = IntStream.range(100, 104);
intStream.forEach(e -> System.out.println(e));
System.out.println("=====");

// 무한 개수의 랜덤한 정수 스트림을 생성
IntStream randomStream = new Random().ints();
// 3개만 출력 구문 수행
randomStream.limit(3).forEach(n -> System.out.println(n));
System.out.println("=====");
try {
    // 현재 디렉토리를 가지고 스트림을 생성
    Stream<Path> path = Files.list(Paths.get("./"));
    path.forEach(n -> System.out.println(n));
    path.close();
} catch (Exception e) {
    System.out.println(e.getMessage());
}

}
```

3. 스트림 중간 연산

- ❖ 대량의 데이터를 가공해서 축소하는 것을 Reduction
- ❖ Reduction을 한 후 합계, 평균값, 최대값, 최소값, 카운팅 과 같은 집계 가능
- ❖ Reduction에 중간처리를 할 수 있는데 이러한 처리로는 필터링, 매핑, 정렬, 그룹핑 등
- ❖ 스트림에는 이러한 중간처리와 집계를 위한 메소드가 존재



3. 스트림 중간 연산

❖중간 처리 메소드 : 스트림을 리턴

종류	리턴 타입	메소드(매개 변수)	소속된 인터페이스
중간 처리	필터링	distinct()	공통
		filter(...)	공통
	매핑	flatMap(...)	공통
		flatMapToDouble(...)	Stream
		flatMapToInt(...)	Stream
		flatMapToLong(...)	Stream
		map(...)	공통
		mapToDouble(...)	Stream, IntStream, LongStream
		mapToInt(...)	Stream, LongStream, DoubleStream
		mapToLong(...)	Stream, IntStream, DoubleStream
		mapToObj(...)	IntStream, LongStream, DoubleStream
		asDoubleStream()	IntStream, LongStream
		asLongStream()	IntStream
		boxed()	IntStream, LongStream, DoubleStream
	정렬	sorted(...)	공통
	루핑	peek(...)	공통

3.1 distinct()와 filter()

- ❖ `skip(long n)`: n 만큼 넘어감
- ❖ `limit(long n)`: n 만큼 만 추출
- ❖ `distinct()`는 중복을 제거해주는 중간처리 메소드로 `equals` 메소드로 비교
- ❖ `filter(매개변수가 1개이고 boolean을 리턴하는 람다식)`는 원하는 조건에 해당하는 데이터만 골라 내는 메소드로 매개변수는 스트림의 각 요소

3.1 distinct()와 filter()

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

public class StreamFiltering {
    public static void main(String[] args) {
        List<String> list = Arrays.asList(
            "이신", "홍범도", "유관순", "이신", "이순신", "김유
신");

        System.out.println("2개를 넘기고 3개 출력");
        list.stream().skip(2).limit(3).forEach(n -> System.out.println(n));
        System.out.println();
        System.out.println("중복 제거");
        list.stream().distinct().forEach(n -> System.out.println(n));
        System.out.println();
        System.out.println("신으로 끝나는 데이터 추출");
        list.stream().filter(n -> n.endsWith("신")).forEach(n ->
System.out.println(n));
        System.out.println();
        System.out.println("중복을 제거하고 신으로 끝나는 데이터 추출");
        list.stream().distinct().filter(n -> n.endsWith("신")).forEach(n ->
System.out.println(n));

    }
}
```

3.2 flatMap자료형()

❖ 스트림의 요소가 배열이나 컬렉션 일 때 또는 분할 가능한 형태로 되어 있을 때 이를 다시 스트림으로 변환해주는 메소드

리턴 타입	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 → InputSteam
LongStream	flatMapToLong(Function<T, LongStream>)	T → LongStream

3.2 flatMap자료형()

flatMap을 이용한 스트림 생성

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

public class FlatMapMain {
    public static void main(String[] args) {
        List<String> list = Arrays.asList("java8 lambda", "stream mapping");
        list.stream().flatMap(data -> Arrays.stream(data.split("
    )))
        .forEach(word -> System.out.println(word));

        Stream<String[]> ar = Stream.of(new String[] { "베이브루스", "루게릭
        ", "타이콥" },
        new String[] { "사이영", "랜디존슨", "페드로마르
        티네스" });

        Stream<String> stream = ar.flatMap(Arrays::stream);
        stream.forEach(str -> System.out.println(str));
    }
}
```

3.3 map자료형()

❖객체를 다른 자료형 요소로 변환해주는 메소드

❖객체::변환하고자 하는 자료형을 리턴하는 메소드이름 의 형식으로 매개변수를 대입

리턴 타입	메소드(매개 변수)	요소 → 대체 요소
Stream<R>	map(Function<T, R>)	T → R
DoubleStream	mapToDouble(ToDoubleFunction<T>)	T → double
IntStream	mapToInt(ToIntFunction<T>)	T → int
LongStream	mapToLong(ToLongFunction<T>)	T → long
DoubleStream	map(DoubleUnaryOperator)	double → double
IntStream	mapToInt(DoubleToIntFunction)	double → int
LongStream	mapToLong(DoubleToLongFunction)	double → long
Stream<U>	mapToObj(DoubleFunction< U>)	double → U
IntStream	map(IntUnaryOperator mapper)	int → int
DoubleStream	mapToDouble(IntToDoubleFunction)	int → double
LongStream	mapToLong(IntToLongFunction mapper)	int → long
Stream<U>	mapToObj(IntFunction<U>)	int → U
LongStream	map(LongUnaryOperator)	long → long
DobleStream	mapToDouble(LongToDoubleFunction)	long → double
IntStream	mapToInt(LongToIntFunction)	long → Int
Stream<U>	mapToObj(LongFunction<U>)	long → U

3.3 map자료형()

```
public class Student implements Comparable<Student>{
    private int num;
    private String name;
    private String gender;
    private String subject;
    private int score;

    public Student() {
        super();
    }
    public Student(int num, String name, String gender, String subject, int score) {
        super();
        this.num = num;
        this.name = name;
        this.gender = gender;
        this.subject = subject;
        this.score = score;
    }
}
```

3.3 map자료형()

```
public int getNum() {  
    return num;  
}  
public void setNum(int num) {  
    this.num = num;  
}  
public String getName() {  
    return name;  
}  
public void setName(String name) {  
    this.name = name;  
}  
public String getGender() {  
    return gender;  
}  
public void setGender(String gender) {  
    this.gender = gender;  
}  
public String getSubject() {  
    return subject;  
}  
public void setSubject(String subject) {  
    this.subject = subject;  
}
```

3.3 map자료형()

```
public int getScore() {
    return score;
}
public void setScore(int score) {
    this.score = score;
}

@Override
public String toString() {
    return "Student [num=" + num + ", name=" + name + ", gender=" +
gender + ", subject=" + subject + ", score="
+ score + "]\n";
}

@Override
public int compareTo(Student other) {
    //return other.score - this.score;
    return this.name.compareTo(other.name);
}
}
```


3.3 map자료형()

성별이 남자인 데이터의 점수만 추출해서 출력

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

public class StreamMapMain {
    public static void main(String[] args) {
        List<Student> list = Arrays.asList(
            new Student(1, "김좌진", "남자", "컴퓨터공학과", 80 ),
            new Student(2, "홍범도", "남자", "기계공학과", 92),
            new Student(3, "유관순", "여자", "컴퓨터공학과", 87),
            new Student(4, "윤봉길", "남자", "컴퓨터공학과", 85),
            new Student(5, "남자현", "여자", "전자공학과", 78)
        );

        list.stream()
            .filter(student->student.getGender().equals("남자"))
            .mapToInt(Student::getScore)
            .forEach(student -> System.out.println(student));
    }
}
```

3.4 sorted()

❖ 스트림의 데이터를 정렬하기 위한 메소드

리턴 타입	메소드(매개 변수)	설명
Stream<T>	sorted()	객체를 Comparable 구현 방법에 따라 정렬
Stream<T>	sorted(Comparator<T>)	객체를 주어진 Comparator에 따라 정렬
DoubleStream	sorted()	double 요소를 오름차순으로 정렬
IntStream	sorted()	int 요소를 오름차순으로 정렬
LongStream	sorted()	long 요소를 오름차순으로 정렬

❖ 객체가 Comparable 인터페이스를 implements 하고 있으면 compareTo 메소드의 결과를 가지고 오름차순 정렬을 해주고 매개변수로 Comparator.reverseOrder()를 대입해주면 내림차순 정렬을 수행

❖ 객체가 Comparable 인터페이스를 implements 하고 있지 않으면 매개변수로 Comparator를 구현한 객체를 대입하거나 (a,b)->{ } 형태의 람다식을 대입해주면 되는데 이 때 리턴값은 양수, 0, 음수 값 중에 하나여야 하고 양수를 리턴하면 앞의 데이터가 크다고 간주하고 뒤의 데이터와 자리를 교체

3.4 sorted()

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

public class StreamSortMain {
    public static void main(String[] args) {
        //Comparable 인터페이스가 구현된 경우
        int [] ar = {5, 3, 2, 1, 4};
        IntStream intStream = Arrays.stream(ar);
        System.out.println("오름차순 정렬");

        intStream
            .sorted()
            .forEach(n -> System.out.print(n + "Wt"));
        System.out.println();
    }
}
```

3.4 sorted()

```
//Comparator를 직접 구현하는 경우
System.out.println("점수의 내림차순 정렬");
List<Student> list = Arrays.asList(
    new Student(1, "김좌진", "남자", "컴퓨터공학과", 80 ),
    new Student(2, "홍범도", "남자", "기계공학과", 92),
    new Student(3, "유관순", "여자", "컴퓨터공학과", 87),
    new Student(4, "윤봉길", "남자", "컴퓨터공학과", 85),
    new Student(5, "남자현", "여자", "전자공학과", 78)
);

list.stream()
    .sorted((n1, n2)->{
        return n2.getScore() - n1.getScore();
    })
    .forEach(word -> System.out.println(word));
}
```

3.5 루핑

❖ 스트림의 모든 데이터를 순회하면서 작업을 수행하는 메소드는 peek(), forEach() 2개의 메소드가 존재

❖ peek(): 중간 처리 메소드로 중간에 사용

❖ forEach(): 최종 처리 메소드로 마지막에 사용



3.5 루핑

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

public class StreamLoopMain {
    public static void main(String[] args) {
        List<Student> list = Arrays.asList(
            new Student(1, "김좌진", "남자", "컴퓨터공학과", 80 ),
            new Student(2, "홍범도", "남자", "기계공학과", 92),
            new Student(3, "유관순", "여자", "컴퓨터공학과", 87),
            new Student(4, "윤봉길", "남자", "컴퓨터공학과", 85),
            new Student(5, "남자현", "여자", "전자공학과", 78));

        list.stream()
            .peek(word -> System.out.println(word))
            .mapToInt(Student::getScore)
            .sum();
    }
}
```

4. 최종 연산

❖ 최종 처리 메소드: 스트림의 요소를 모두 소모해서 결과를 만들어내는 연산으로 연산을 수행하게 되면 스트림은 종료

종류		리턴 타입	메소드(매개 변수)	소속된 인터페이스
최종 처리	매칭	boolean	allMatch(...)	공통
		boolean	anyMatch(...)	공통
		boolean	noneMatch(...)	공통
	집계	long	count()	공통
		OptionalXXX	findFirst()	공통
		OptionalXXX	max(...)	공통
		OptionalXXX	min(...)	공통
		OptionalDouble	average()	IntStream, LongStream, DoubleStream
		OptionalXXX	reduce(...)	공통
		int, long, double	sum()	IntStream, LongStream, DoubleStream
	루핑	void	forEach(...)	공통
	수집	R	collect(...)	공통

4.1 Optional<T>

- ❖ T 타입의 객체를 감싸는 클래스
- ❖ 스트림은 최종 연산의 결과를 Optional 객체에 담아서 리턴하는 경우가 있음
- ❖ 이유는 반환된 결과가 null 인 경우를 직접 처리하기 보다는 Optional에 정의된 메소드를 이용해서 간단히 처리할 수 있기 때문
- ❖ 값 가져오기
 - ✓ T get(): 값을 가져오고 null 이면 예외 발생
 - ✓ T orElse(T default): 값을 가져오고 null이면 default 값 리턴
 - ✓ Boolean isPresent(): 값이 null이면 false 있으면 true 리턴
- ❖ Optional<T>의 변형
 - ✓ OptionalInt
 - ✓ OptionalDouble
 - ✓ OptionalLong

4.2 forEach

- ❖ 스트림의 요소를 소모하는 최종연산
- ❖ 리턴 타입은 void
- ❖ Consumer 타입의 람다를 대입
- ❖ 매개변수는 스트림의 각 요소
- ❖ 스트림의 내용을 출력할 때 많이 사용



4.3 매핑

- ❖ 최종 처리 단계에서 요소들이 특정 조건에 만족하는지 조사하는 연산
- ❖ `allMatch (Predicate<? super T> predicate)` 메소드: 모든 요소들이 매개변수로 주어진 `Predicate`의 조건을 만족하는지 조사
- ❖ `anyMatch(Predicate<? super T> predicate)` 메소드: 최소한 한 개의 요소가 매개변수로 주어진 `Predicate` 조건을 만족하는지 조사
- ❖ `noneMatch(Predicate<? super T> predicate)` 메소드: 모든 요소들이 매개값으로 주어진 `Predicate`의 조건을 만족하지 않는지 조사
- ❖ `Optional<T> findFirst():` 첫번째 데이터 리턴
- ❖ `Optional<T> findAny():` 병렬 스트림에서 첫번째 데이터 리턴

4.3 매핑

문자열의 길이가 6자 이상인 데이터가 1개 이상 있는지 확인

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

public class StreamMatchMain {
    public static void main(String[] args) {
        Stream<String[]> ar = Stream.of(
            new String[]{"베이브루스", "루게릭", "타이콥"},
            new String[]{"사이영", "랜디존슨", "페드로마르티네스"}
        );
        Stream<String> stream = ar.flatMap(Arrays::stream);
        boolean result =
            stream.anyMatch(name ->
                name.length()>6);
        System.out.println("결과:" + result);
    }
}
```

4.4 집계

- ❖ 최종 처리 기능으로 요소들을 처리해 카운팅, 합계, 평균값, 최대값, 최소값 등과 같이 하나의 값으로 산출하는 것
- ❖ 집계는 대량의 데이터를 가공해서 축소하는 리덕션(Reduction)
- ❖ 기본형 스트림에서는 모든 메소드를 제공하지만 기본형이 아닌 경우는 count 와 min, max만 존재

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

4.4 집계

- ❖ 일반 자료형으로 리턴되는 데이터는 바로 사용이 가능
- ❖ Optional은 일반 자료형의 데이터를 객체로 매핑하는 자료형
- ❖ Optional 타입은 Optional, OptionalDouble, OptionalInt, OptionalLong 타입이 있는데 이 자료형으로 리턴되는 데이터는 get(), getAsDouble(), getAsInt(), getAsLong() 을 호출하면 값을 가져올 수 있음



4.4 집계

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

public class StreamReduceMain {

    public static void main(String[] args) {
        List<Student> list = Arrays.asList(
            new Student(1, "김좌진", "남자", "컴퓨터공학과", 80),
            new Student(2, "홍범도", "남자", "기계공학과", 92),
            new Student(3, "유관순", "여자", "컴퓨터공학과", 87),
            new Student(4, "윤봉길", "남자", "컴퓨터공학과", 85),
            new Student(5, "남자현", "여자", "전자공학과", 78));

        // Map-Reduce Programming
        int sum = list.stream().mapToInt(Student::getScore).sum();
        System.out.println("합계:" + sum);
        double avg =
list.stream().mapToInt(Student::getScore).average().getAsDouble();
        System.out.println("평균:" + avg);
    }
}
```

4.5 reduce

- ❖ 기본 집계 이외의 연산을 수행하고자 하는 경우에 사용하는 메소드로 스트림의 요소를 줄여나가면서 연산을 수행하고 최종 결과를 반환하는 메소드

`Optional<T> reduce(BinaryOperator<T> accumulator)`

`Optional<T> reduce(T identity, BinaryOperator<T> accumulator)`

- ❖ 첫번째 함수는 처음 2개를 가지고 연산을 수행한 후 그 결과를 가지고 다음 데이터와 연산을 수행하여 최종 결과를 리턴
- ❖ `reduce((a,b)->a+b)` 를 하게 되면 두 개 데이터의 합
- ❖ 두번째 함수는 첫번째 매개변수로 초기값을 설정하는 것이 가능
- ❖ 첫번째 함수를 이용하는 경우에 데이터가 없으면 `NoSuchElementException`이 발생하지만 `reduce(0, (a,b)->a+b)`를 입력하게 되면 데이터가 없으면 0을 리턴

4.5 reduce

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

public class StreamReduceSum {
    public static void main(String[] args) {
        List<Student> list = Arrays.asList(
            new Student(1, "김좌진", "남자", "컴퓨터공학과", 80),
            new Student(2, "홍범도", "남자", "기계공학과", 92),
            new Student(3, "유관순", "여자", "컴퓨터공학과", 87),
            new Student(4, "윤봉길", "남자", "컴퓨터공학과", 85),
            new Student(5, "남자현", "여자", "전자공학과", 78));
        int sum = list.stream()
            .filter(student -> student.getGender().equals("남자"))
            .mapToInt(Student::getScore)
            .reduce(0, (n1, n2) -> n1+n2);
        System.out.println("합계:" + sum);
    }
}
```


4.6 collect

- ❖ 필터링 또는 매핑을 한 후 요소들을 수집하는 최종 처리
- ❖ 필요한 요소만 컬렉션에 담을 수 있음
- ❖ Collector의 타입 파라미터 T는 요소
- ❖ A는 누적기(accumulator)
- ❖ R은 요소가 저장될 컬렉션
- ❖ 해석하면 T 요소를 A 누적기가 R에 저장한다는 의미
- ❖ Collectors 클래스의 정적 메소드를 이용

리턴 타입	Collectors의 정적 메소드	설명
Collector<T, ?, List<T>>	toList()	T를 List에 저장
Collector<T, ?, Set<T>>	toSet()	T를 Set에 저장
Collector<T, ?, Collection<T>>	toCollection(Supplier<Collection<T>>)	T를 Supplier가 제공한 Collection에 저장
Collector<T, ?, Map<K,U>>	toMap(Function<T,K> keyMapper, Function<T,U> valueMapper)	T를 K와 U로 매핑해서 K를 키로, U를 값으로 Map에 저장
Collector<T, ?, ConcurrentMap<K,U>>	toConcurrentMap(Function<T,K> keyMapper, Function<T,U> valueMapper)	T를 K와 U로 매핑해서 K를 키로, U를 값으로 ConcurrentMap에 저장

4.6 collect

- ❖ collect 함수에 `Collectors.toList()` 또는 `toSet()`을 대입하면 List 나 Set으로 결과를 반환
- ❖ List 나 Set 인터페이스의 anonymous 객체가 아닌 특정 컬렉션으로 반환하고자 하는 경우에는 `Collectors.toCollection(ArrayList::new)` 처럼 특정 컬렉션의 생성자를 대입
- ❖ 배열로 변환하고자 하는 경우에는 `Collectors.toArray()`를 이용하게 되는데 매개변수를 대입하지 않으면 Object 배열로 리턴하고 특정 클래스의 배열로 만들고자 하는 경우에는 **클래스[]::new** 처럼 생성자를 설정
- ❖ 맵으로 변환할 때는 `toMap` 이라는 메소드를 대입하면 되는데 이 메소드를 사용할 때는 첫 번째 매개변수로 키를 두 번째 매개변수로 값을 설정해 주어야 하는데 하나의 매개변수를 받아서 다른 형태로 변환해주는 람다를 대입

4.6 collect

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

public class StreamCollect1 {
    public static void main(String[] args) {
        List<Student> list = Arrays.asList(
            new Student(1, "김좌진", "남자", "컴퓨터공학과", 80),
            new Student(2, "홍범도", "남자", "기계공학과", 92),
            new Student(3, "유관순", "여자", "컴퓨터공학과", 87),
            new Student(4, "윤봉길", "남자", "컴퓨터공학과", 85),
            new Student(5, "남자현", "여자", "전자공학과", 78));
```

4.6 collect

```
// 남자들만 묶어 List 생성
List<Student> manList = list.stream().filter(s ->
s.getGender().equals("남자")).collect(Collectors.toList());
manList.stream().forEach(s -> System.out.println(s.getName()));
System.out.println("=====");

// 여자들만 묶어 Set 생성
Set<Student> womanSet = list.stream().filter(s ->
s.getGender().equals("여자")).collect(Collectors.toSet());
womanSet.stream().forEach(s -> System.out.println(s.getName()));
System.out.println("=====");

// 모든 데이터를 name을 키로해서 Map 생성
Map<String, Student> map =
list.stream().collect(Collectors.toMap(Student::getName, item -> item));
System.out.println(map);
    }
}
```

4.6 collect

- ❖ counting()을 대입하면 데이터의 개수를 정수로 리턴
- ❖ summingInt(치환함수)를 대입하면 합계
- ❖ averagingInt(치환함수)를 대입하면 평균
- ❖ maxBy(비교함수), minBy(비교함수)를 대입하면 최대나 최소
- ❖ Int 대신에 Double 사용 가능

4.6 collect

```
import java.util.Arrays;
import java.util.Comparator;
import java.util.List;
import java.util.Optional;
import java.util.stream.Collectors;

public class StreamCollect2 {
    public static void main(String[] args) {
        List<Student> list = Arrays.asList(
            new Student(1, "김좌진", "남자", "컴퓨터공학과", 80),
            new Student(2, "홍범도", "남자", "기계공학과", 92),
            new Student(3, "유관순", "여자", "컴퓨터공학과", 87),
            new Student(4, "윤봉길", "남자", "컴퓨터공학과", 85),
            new Student(5, "남자현", "여자", "전자공학과", 78));


        // 데이터 개수
        long cnt = list.stream().count();
        System.out.println("데이터 개수:" + cnt);
        cnt = list.stream().collect(Collectors.counting());
        System.out.println("데이터 개수:" + cnt);
    }
}
```

4.6 collect

```
// 점수 합계
int sum = list.stream().mapToInt(Student::getScore).sum();
System.out.println("점수 합계:" + sum);
sum = list.stream().collect(Collectors.summingInt(Student::getScore));
System.out.println("점수 합계:" + sum);

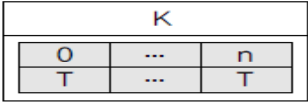

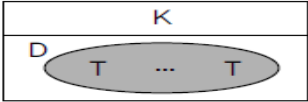
// 최대값
Optional<Student> top =
list.stream().max(Comparator.comparingInt(Student::getScore));
System.out.println(top);
top =
list.stream().collect(Collectors.maxBy(Comparator.comparingInt(Student::getScore)));
System.out.println(top);

    }
}
```



4.6 collect

- ❖ collect는 요소를 그룹핑해서 수집할 수 있는 기능을 제공
- ❖ collect ()를 호출시 Collectors의 groupingBy()를 대입해서 그룹화가 가능
- ❖ 컬렉션의 요소들을 그룹핑해서 Map객체 생성

리턴 타입	Collectors의 정적 메소드	설명
Collector<T,?,Map<K,List<T>>>	groupingBy(Function<T, K> classifier)	T를 K로 매핑하고 K키에 저장된 List에 T를 저장한 Map 생성
Collector<T,?, ConcurrentMap<K,List<T>>>	groupingByConcurrent(Function<T,K> classifier)	
Collector<T,?,Map<K,D>>	groupingBy(Function<T, K> classifier, Collector<T,A,D> collector)	T를 K로 매핑하고 K키에 저장된 D객체에 T를 누적한 Map 생성
Collector<T,?, ConcurrentMap<K,D>>	groupingByConcurrent(Function<T,K> classifier, Collector<T,A,D> collector)	
Collector<T,?,Map<K,D>>	groupingBy(Function<T,K> classifier, Supplier<Map<K,D>> mapFactory, Collector<T,A,D> collector)	T를 K로 매핑하고 Supplier가 제공하는 Map에서 K키에 저장된 D객체에 T를 누적
Collector<T,?, ConcurrentMap<K,D>>	groupingByConcurrent(Function<T,K> classifier, Supplier<ConcurrentMap<K,D>> mapFactory, Collector<T,A,D> collector)	

4.6 collect

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

public class StreamCollect3 {
    public static void main(String[] args) {
        List<Student> list = Arrays.asList(
            new Student(1, "김좌진", "남자", "컴퓨터공학과", 80),
            new Student(2, "홍범도", "남자", "기계공학과", 92),
            new Student(3, "유관순", "여자", "컴퓨터공학과", 87),
            new Student(4, "윤봉길", "남자", "컴퓨터공학과", 85),
            new Student(5, "남자현", "여자", "전자공학과", 78));

        Map<String, List<Student>> groupMap = list.stream()
            .collect(Collectors.groupingBy(Student :: getGender));
        System.out.println(groupMap);
        System.out.print("[남자] ");
        groupMap.get("남자").stream().forEach(s->
            System.out.print(s.getName() + " "));
        System.out.print("\n[여자] ");
        groupMap.get("여자").stream().forEach(s->
            System.out.print(s.getName() + " "));
    }
}
```

4.6 collect

- ❖ 그룹핑 한 후 집계 가능
- ❖ Collect의 매개변수로 `Collectors.groupingBy`(그룹화 할 함수, 구하고자하는 집계함수)를 대입

리턴 타입	메소드(매개 변수)	설명
<code>Collector<T,?,R></code>	<code>mapping(Function<T, U> mapper, Collector<U,A,R> collector)</code>	T를 U로 매핑한 후, U를 R에 수집
<code>Collector<T,?,Double></code>	<code>averagingDouble(ToDoubleFunction<T> mapper)</code>	T를 Double로 매핑한 후, Double의 평균값을 산출
<code>Collector<T,?,Long></code>	<code>counting()</code>	T의 카운팅 수를 산출
<code>Collector <CharSequence,?,String></code>	<code>joining(CharSequence delimiter)</code>	<code>CharSequence</code> 를 구분자(<code>delimiter</code>)로 연결한 String을 산출
<code>Collector<T,?,Optional<T>></code>	<code>maxBy(Comparator<T> comparator)</code>	<code>Comparator</code> 를 이용해서 최대 T를 산출
<code>Collector<T,?,Optional<T>></code>	<code>minBy(Comparator<T> comparator)</code>	<code>Comparator</code> 를 이용해서 최소 T를 산출
<code>Collector<T,?,Integer></code>	<code>summingInt(ToIntFunction) summingLong(ToLongFunction) summingDouble(ToDoubleFunction)</code>	Int, Long, Double 타입의 합계 산출

4.6 collect

```
import java.util.Arrays;
import java.util.Comparator;
import java.util.List;
import java.util.Map;
import java.util.Optional;
import java.util.Set;
import java.util.stream.Collectors;

public class StreamCollect4 {
    public static void main(String[] args) {
        List<Student> list = Arrays.asList(
            new Student(1, "김좌진", "남자", "컴퓨터공학과", 80),
            new Student(2, "홍범도", "남자", "기계공학과", 92),
            new Student(3, "유관순", "여자", "컴퓨터공학과", 87),
            new Student(4, "윤봉길", "남자", "컴퓨터공학과", 85),
            new Student(5, "남자현", "여자", "전자공학과", 78));

        // 성별 평균 점수
        Map<String, Double> groupMap = list.stream()

            .collect(Collectors.groupingBy(Student::getGender,
            Collectors.averagingDouble(Student::getScore)));
        System.out.println("남자 평균 점수: " + groupMap.get("남자"));
        System.out.println("여자 평균 점수: " + groupMap.get("여자"));
```

4.6 collect

```
// 학과별 1등 추출
Map<String, Optional<Student>> topMap =
list.stream().collect(Collectors.groupingBy(Student::getSubject,

Collectors.maxBy(Comparator.comparingInt(Student::getScore))));
Set<String> keys = topMap.keySet();
for (String key : keys)
    System.out.println(key + ":" + topMap.get(key));

}
}
```

5. 병렬처리

❖ 병렬 처리(Parallel Operation)

- ✓ 멀티 코어 CPU 환경에서 쓰임
- ✓ 하나의 작업을 분할해서 각각의 코어가 병렬적 처리하는 것
- ✓ 병렬 처리의 목적은 작업 처리 시간을 줄이기 위한 것
- ✓ 자바 8부터 요소를 병렬 처리할 수 있도록 하기 위해 병렬 스트림 제공

❖ 동시성(Concurrency)

- ✓ 동시성 - 멀티 작업을 위해 멀티 스레드가 번갈아 가며 실행하는 성질
- ✓ 싱글 코어 CPU를 이용한 멀티 작업
- ✓ 병렬적으로 실행되는 것처럼 보임
- ✓ 실체는 번갈아 가며 실행하는 동시성 작업

5. 병렬처리

❖ 병렬성의 종류

✓ 데이터 병렬성

- 전체 데이터를 쪼개어 서브 데이터들로 만든 뒤 병렬 처리해 작업을 빨리 끝내는 것
- 자바 8에서 지원하는 병렬 스트림은 데이터 병렬성을 구현한 것
- 멀티 코어의 수만큼 대용량 요소를 서브 요소들로 나누고 각각의 서브 요소들을 분리된 스레드에서 병렬 처리

ex) 쿼드 코어(Quad Core) CPU일 경우 4개의 서브 요소들로 나누고, 4개의 스레드가 각각의 서브 요소들을 병렬 처리작업 병렬성

✓ 작업 병렬성은 서로 다른 작업을 병렬 처리하는 것

- Ex) 웹 서버 (Web Server):각각의 브라우저에서 요청한 내용을 개별 스레드에서 병렬로 처리

❖ 포크 조인 프레임워크

- ✓ 런타임 시 포크조인 프레임워크 동작
- ✓ 포크 단계에서는 전체 데이터를 서브 데이터로 분리
- ✓ 서브 데이터를 멀티 코어에서 병렬로 처리
- ✓ 조인 단계에서는 서브 결과를 결합해서 최종 결과 도출

5. 병렬처리

❖ 병렬 스트림을 얻는 메소드

- ✓ `parallelStream()` 메소드: 컬렉션으로부터 병렬 스트림을 바로 리턴
- ✓ `parallel()` 메소드: 순차 처리 스트림을 병렬 처리 스트림으로 변환해서 리턴

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

5. 병렬처리

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

public class SteamParallelProcessing {

    // 요소 처리
    public static void work(int value) {
        try {
            Thread.sleep(100);
        } catch (InterruptedException e) {
        }
    }
}
```


5. 병렬처리

// 순차 처리

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

// 병렬 처리

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

5. 병렬처리

```
public static void main(String[] args) {  
    // 소스 컬렉션  
    List<Integer> list = Arrays.asList(0, 1, 2, 3, 4, 5, 6, 7, 8, 9);  
    // 순차 스트림 처리 시간 구하기  
    long sequential = testSequential(list);  
    // 병렬 스트림 처리 시간 구하기  
    long timeParallel = testParallel(list);  
    System.out.println(sequential);  
    System.out.println(timeParallel);  
  
    if (sequential < timeParallel) {  
        System.out.println("성능 테스트 결과: 순차 처리가 더빠름");  
    } else {  
        System.out.println("성능 테스트 결과: 병렬 처리가 더빠름");  
    }  
}
```

```
}
```