JAVA Stream

프로그램은 사람이 이해하는 코드를 작성.

느려도 꾸준하면 경기에서 이긴다.

작성자 : 홍효상
이메일 : hyomee@naver.com
소스 : https://github.com/hyomee/JAVA_EDU

Content

11. Stream

1. Stream

" 데이터의 흐름 "

사용하는 이유

- 배열 또는 컬렉션 인스턴스에 함수 여러 개를 조합해서 원하는 결과를 필터링하고 가공된 결과를 얻을 수 있다.
  - 배열 또는 컬렉션 인스턴스에 함수 여러 개를 조합해서 원하는 결과를 조합해서 결과를 필터링하고 가공된 결과를 얻을 수 있다.

- 연산은 구현체에 맡기며, 값들의 묶음을 처리하고 원하는 작업을 지정하는 데 필요한 핵심 추상화이다.
  - 즉, 평균을 계산 하는 기능이 있다면 평균을 구하고자 하는 요소의 카운트를 계산 하고 결과를 합치기 위해 다중 스레드를 사용해 연산을 병렬화 하는 일은 스트림 라이브러리에 맡긴다.

- 람다를 이용해서 코드의 양을 줄이고 간결하게 표현할 수 있습니다. 즉, 배열과 컬렉션을 함수형으로 처리할 수 있다.
- 간단하게 병렬처리(multi-threading)가 가능하다

Collection 과 차이점

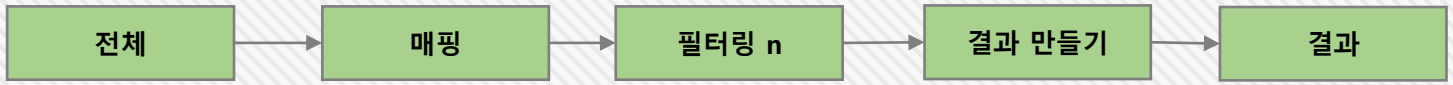
- Stream은 요소들을 보관 하지 않는다. (요소들은 하부 Collection에 보관 되거나 필요할 때 생성)
- Stream 연산은 원본을 변경 하지 않는다. (결과는 새로운 스트림으로 반환 함)
- Stream 연산은 가능하면 지연(Lazy)처리 된다.

이전 Collection 처리 방법

- for, foreach문을 사용 하여 요소를 하나씩 꺼내서 처리.
- 업무(로직)이 복잡 하면 로직이 섞이고 코드의 양이 많음
- 중첩 for문 사용으로 독해 어려움

Stream을 이용해서 작업할 때 연산들의 파이프라인은 세 단계로 설정

- Stream 생성 : Stream instance 생성
- Stream 가공 : 초기 Stream을 다른 Stream으로 변환하는 중간 연산 (Intermediate operations)
  :: Filtering, Mapping -> 하나 이상의 단계로 지정
- Stream 결과 : 최종 연산(terminal operations) 적용 : 이 연산은 지연연산들의 실행을 강제 한다, 이후로는 해당 Stream은 더 사용 할 수 없음



2. Stream 생성

Stream<Object> = Stream.명령어

01. 빈 Stream 생성

- String의 empty와 같다.

```
Stream<Object> emptyStream = Stream.empty();
System.out.println("빈 Stream : " + emptyStream.count());
```

```
빈 Stream : 0
```

02. of 메소드를 사용한 가변 인자 stream 생성

```
System.out.println("* of를 이용한 가변 Stream 생성* ");
Stream<String> subjectStream = Stream.of("국어", "영어", "수학");
subjectStream.forEach(subject -> System.out.println(subject));
```

```
국어
영어
수학
```

03. 빌더를 이용한 Stream 생성

```
Stream<Object> streamObject = Stream.builder()
        .add("한국")
        .add("미국")
        .add("호주")
        .build();
streamObject.forEach(object -> System.out.println(object));
```

```
국어
영어
수학
```

```
Stream<String> streamString = Stream.<String>builder()
        .add("한국")
        .add("미국")
        .add("호주")
        .build();
streamString.forEach(string -> System.out.println(string));
```

```
국어
영어
수학
```

04. generate을 이용한 무한 Stream 생성

- 주의 사이즈를 정해야 한다.

2. Stream 생성

Stream<Object> = Stream.명령어

05. 빌더를 이용한 Stream 생성 – 객체 사용

```java
public class SubjectDTO {

  private int schoolNo;
  private String userId;
  private String subjectId;
  private String subjectName;
  private int score;

  public SubjectDTO() {};

  public SubjectDTO(int schoolNo,
                    String userId,
                    String subjectId,
                    String subjectName,
                    int score) {
    this.schoolNo = schoolNo;
    this.userId = userId;
    this.subjectId = subjectId;
    this.subjectName = subjectName;
    this.score = score;
  }
```

```java
public static Stream<Object> createSubjectDTOStream() {
  return Stream.builder()
      .add(new SubjectDTO("USER_01", "SUBJECT_01", "국어", 80))
      .add(new SubjectDTO("USER_01", "SUBJECT_02", "영어", 90))
      .add(new SubjectDTO("USER_01", "SUBJECT_02", "수학", 85))
      .add(new SubjectDTO("USER_02", "SUBJECT_02", "국어", 70))
      .add(new SubjectDTO("USER_02", "SUBJECT_02", "영어", 80))
      .add(new SubjectDTO("USER_02", "SUBJECT_02", "수학", 75))
      .add(new SubjectDTO("USER_03", "SUBJECT_02", "국어", 95))
      .add(new SubjectDTO("USER_03", "SUBJECT_02", "영어", 65))
      .add(new SubjectDTO("USER_03", "SUBJECT_02", "수학", 75))
      .add(new SubjectDTO("USER_04", "SUBJECT_02", "국어", 95))
      .add(new SubjectDTO("USER_04", "SUBJECT_02", "영어", 85))
      .add(new SubjectDTO("USER_05", "SUBJECT_02", "수학", 75))
      .build();
}
```

```java
Stream<Object> subjectDTOStream = DataCreateService.createSubjectDTOStream();
subjectDTOStream.forEach(subject -> System.out.println(subject.toString()));
```

```
SubjectDTO{, schoolNo='0', userId='USER_01', subjectId='SUBJECT_01', subjectName='국어', score=80}
SubjectDTO{, schoolNo='0', userId='USER_01', subjectId='SUBJECT_02', subjectName='영어', score=90}
SubjectDTO{, schoolNo='0', userId='USER_01', subjectId='SUBJECT_02', subjectName='수학', score=85}
SubjectDTO{, schoolNo='0', userId='USER_02', subjectId='SUBJECT_02', subjectName='국어', score=70}
SubjectDTO{, schoolNo='0', userId='USER_02', subjectId='SUBJECT_02', subjectName='영어', score=80}
SubjectDTO{, schoolNo='0', userId='USER_02', subjectId='SUBJECT_02', subjectName='수학', score=75}
SubjectDTO{, schoolNo='0', userId='USER_03', subjectId='SUBJECT_02', subjectName='국어', score=95}
SubjectDTO{, schoolNo='0', userId='USER_03', subjectId='SUBJECT_02', subjectName='영어', score=65}
SubjectDTO{, schoolNo='0', userId='USER_03', subjectId='SUBJECT_02', subjectName='수학', score=75}
SubjectDTO{, schoolNo='0', userId='USER_04', subjectId='SUBJECT_02', subjectName='국어', score=95}
SubjectDTO{, schoolNo='0', userId='USER_04', subjectId='SUBJECT_02', subjectName='영어', score=85}
SubjectDTO{, schoolNo='0', userId='USER_05', subjectId='SUBJECT_02', subjectName='수학', score=75}
```

2. Stream 생성

Stream<Object> = Stream.명령어

07. Array Stream 생성

- Array : Arrays.stream을 사용

```
System.out.println("===== Stream 변환 =======");
String[] strings = new String[]{"c#", "java", "java script"};

Stream<String> stringStream = Arrays.stream(strings);
stringStream.forEach(s -> System.out.println(s));
// 범위 지정
Stream<String> stringStreamOfElement = Arrays.stream(strings, 1,3) ;
stringStreamOfElement.forEach(s -> System.out.println(s));
```

```
===== Stream 변환 ======= ::
c
java
java script
c++
c#
```

08. Collection을 Stream 생성

- Collection, List, Set : stream 사용
- 병렬 처리 : parallelStream 사용

```
List<String> stringList = new ArrayList<>();
stringList.add("c");
stringList.add("java");
stringList.add("java script");
stringList.add("c++");
stringList.add("c#");
// Stream 변환
System.out.println("===== Stream 변환 ======= :: " );
Stream<String> strStream = stringList.stream();
strStream.forEach(s -> System.out.println(s));
System.out.println("===== parallelStream 변환 =======");
Stream<String> strParallelStream = stringList.parallelStream();
strParallelStream.forEach(s -> System.out.println(s));
```

```
===== parallelStream 변환 =======
java script
c++
c#
java
c
```

1. 기본 타입 Stream

01. IntStream, longStream , doubleStream

- IntStream : short, char, byte, Boolean
- longStream : long
- doubleStream: float

➢ IntStream 생성

- IntStream.of, Arrays,stream 메소드 사용
- IntStream.builder().build() 사용
- Arrays.stream(values. from, to)

- 정적 generate, iterate 사용
- IntStream, longStream :
  크기가 1인 정수 범위를 생성하는 정적 range, rangeClosed
  IntStream zeroToNinetyNine = IntStream.range(0,100); // 100 제외
  IntStream zeroToHundred = IntStream.rangeClosed(0,100); // 100 포함

```
2
34
40
====================
30
40
50
====================
30
40
```

```java
public static void main(String[] args) {
  IntStream intStream = IntStream.builder()
          .add(2)
          .add(34)
          .add(40)
          .build();

  intStream.forEach(i -> System.out.println(i));

  System.out.println("====================");

  intStream = IntStream.of(30, 40, 50);
  intStream.forEach(i -> System.out.println(i));

  System.out.println("====================");
  int[] intArrays = new int[]{20,30,40};
  intStream = Arrays.stream(intArrays, 1,3);
  intStream.forEach(i -> System.out.println(i));

  IntStream zeroToNinetyNine = IntStream.range(0,100);
  IntStream zeroToHundred = IntStream.rangeClosed(0,100);
}
```

1. 기본 타입 Stream

02. mapToInt, mapToLong, mapToDouble

- 객체 Stream을 기본 타입 스트림으로 변환
- 기본 타입 스트림을 객체 스트림으로 변환 : boxed 사용

➢ 기본 타입과 객체 스트림의 차이점

- toArray는 기본 타입 배열을 리턴
- 옵션 결과를 돌려주는 메소드 : OptionalInt, OprionalLong, OptionalDouble
  리턴 : getAsInt, getAsLong, getAsDouble 사용
  - Optional function는 get을 사용함
- 기본 타입 Stream은 평균, 최대값, 최소값을 리턴 하는 sum, average, max, min
- summaryStatistics메소드는 스트림의 합계, 평균, 최대값, 최소값을 동시에 보고 할 수 있는 intSummaryStatistics, LongSummaryStatistics, DoubleSummarySatistics 객체를 돌려줌

```
===== Stream 변환 ======= ::
===== IntStream 변환 ======= ::
1
4
11
3
2
===== Stream 변환 ======= ::
0
1
2
3
4
```

```java
public class ConvertStream {

  public static void main(String[] strings) {
    List<String> stringList = new ArrayList<>();
    stringList.add("c");
    stringList.add("java");
    stringList.add("java script");
    stringList.add("c++");
    stringList.add("c#");

    System.out.println("===== Stream 변환 ======= :: " );
    Stream<String> stringStream = stringList.stream();

    System.out.println("===== IntStream 변환 ======= :: " );
    IntStream lenths = stringStream.mapToInt(String::length);
    lenths.forEach(i -> System.out.println(i));

    System.out.println("===== Stream 변환 ======= :: " );
    Stream<Integer> integerStream = IntStream.range(0,5).boxed();
    integerStream.forEach(i -> System.out.println(i));
  }
}
```

1. 병렬 Stream

01. parallelStream

- Collection.parallelStream을 제외하고는 순차 스트림(Sequential Stream)을 생성
- Stream 대신에 parallelStream 메소드를 사용해서 생성

- `Stream<String> parallelStream = Stream.of(strArrays).parallelStream();`
- `Stream<CustInfo> parallelCustInfoStream = custInfoList.parallelStream();`

- 병렬 여부 확인 : isParallel
- 병렬 모드로 실행이 되면 최종 메소드(terminal method)가 실행 할 때 모든 지연 처리 중 중간 스트림 연산은 병렬화 됨 :: 연산들은 무상태(stateless)고 임의의 순서로 실행
  :: 스레드 안정 보장, race condition등을 고려 해야 함

```
===== Stream 변환 ======= ::
c
java
java script
c++
c#
===== parallelStream 변환 =======
java script
c#
c++
java
c
```

1. 함수형 인터페이스

01. 함수형 인터페이스

- 1개의 추상 메소드를 가지고 있는 인터페이스 :: Single Abstract Method(SAM)
- 사용 이유 : 자바의 람다식은 함수형 인터페이스로만 접근 가능 하기 때문

➢ 익명 클래스와 람다 공통점

- - 익명클래스나 람다가 선언되어 있는 바깥 클래스의 멤버 변수나 메서드에 접근 할 수 있음
  - 하지만 멤버 변수나 메서드의 매개변수에 접근하기 위해서는 해당 변수들이 final의 특성을 가지고 있어야 함.

➢ 익명 클래스와 람다 차이점

- 익명클래스와 람다에서의 this의 의미는 다르다
  : 익명클래스의 this는 익명클래스 자신을 가리키지만 람다에서의 this는 선언된 클래스를 가리킵니다.

- 람다는 은닉 변수(Shadow Variable)을 허용하지 않는다
  익명클래스와 람다에서의 this의 의미는 다르다
  : 익명 클래스는 변수를 선언하여 사용할 수 있지만 람다는 이를 허용하지 않습니다.

- 람다는 인터페이스에 반드시 하나의 메서드만 가지고 있어야 한다!
  : 인터페이스에 @FunctionalInterface 어노테이션을 붙이면 두개 이상의 추상 메서드가 선언되었을 경우 컴파일 에러를 발생시킨다.

함수형 인터페이스	파라미터 타입	리턴 타입	설명
Supplier<T>	없음	T	T 타입 값 리턴
Consumer<T>	T	void	T 타입 값 소비
BiConsumer<T, U>	T, U	void	T, U 타입 값 소비
Predicate<T>	T	boolean	Boolean 값 리턴
ToIntFunction<T> ToLongFunction<T> ToDoubleFunction<T>	T	Int long double	T 타입 값 인자로 받고 Int, long, double 리턴
IntFunction<T> LongFunction<T> DoubleFunction<T>	Int long double	R	받고 Int, long, double 인자로 받고 R 타입 리턴

함수형 인터페이스	파라미터 타입	리턴 타입	설명
Function<T, R>	T	R	T 타입 값 인자로 받고 R 타입 리턴
BiFunction<T, U, R>	T, U	R	T, U 타입 인자로 받고 R 타입 리턴
UnaryOperator<T>	T	T	T 타입에 적용되는 단항 연산자
BinaryOperator<T>	T, T	T	T 타입에 적용되는 이항 연산자

2. 자바에서 기본 제공 하는 함수형 interface

01. 자바에서 기본적으로 제공하는 함수형 인터페이스

- Runnable : 인자를 받지 않고 리턴값도 없는 인터페이스
- Supplier : 인자를 받지 않고 T타입의 객체를 리턴
- Consumer : T타입의 인자를 받고 리턴값은 없음 :
  andThen을 사용하면 두개 이상의 연속적인 Consumer를 실행 할
- Function<T, R> : T 타입의 인자를 받고 R타입의 객체를 리턴
- Predicate : T 타입의 인자를 받고 boolean를 리턴

```java
// Runable
Runnable runnable = () -> System.out.println("Runnable....");
runnable.run();

//Supplier
Supplier supplier = () -> "Supplier .... ";
String str = (String) supplier.get();
System.out.println(str);

//Consumer
Consumer<String> consumer = text -> System.out.println("Consumer...." + text);
Consumer<String> consumerandThen = text -> System.out.println("Consumer andThen...." + text);
consumer.andThen(consumerandThen).accept("String type");

//Function
Function<Integer, Integer> multiply = (value) -> value * 2;
Integer result = multiply.apply(5);
System.out.println("Function...." + result);
```

```java
Function<Integer, Integer> add = (value) -> value + 2;
Function<Integer, Integer> addMultiply = multiply.compose(add);
result = addMultiply.apply(5);
System.out.println("Function....addMultiply :: " + result);

// Predicate
Predicate<Integer> predicate = (num) -> num > 10;
System.out.println("Predicate.... :: " + predicate.test(5));

Predicate<Integer> predicate1 = (num) -> num < 20;

System.out.println("Predicate.... 10 < num < 20 :: " + predicate.and(predicate1).test(25));
System.out.println("Predicate.... 10 < num or num  < 20 :: " + predicate.or(predicate1).test(25));
```

```
Runnable....
Supplier ....
Consumer....String type
Consumer andThen....String type
Function....10
Function....addMultiply :: 14
Predicate.... :: false
Predicate.... 10 < num < 20 :: false
Predicate.... 10 < num or num  < 20 :: true
```

1. Stream API

01. Stream filter, map, flatMap

- filter : 특정 조건과 일치하는 모든 요소를 담는 새로운 스트림을 리턴
- Map : 스트림에 있는 item을 변경 하여 새로운 스트림을 리턴
- flatMap : 여러 개의 스트림을 한 개의 스트림으로 합쳐서 새로운 스트림을 리턴

```
List<String> address = new ArrayList<>();
address.add("서울시 송파구 방이동");
address.add("서울시 송파구 송파동");
address.add("서울시 강남구 개포동");
address.add("서울시 강남구 서초동");
```

```
// Filter : Stream 요소를 하나씩 검색 하여 조건에 만족하는 것을 걸려내는 작업
//          predicate<T>, 즉 T를 인자로 받고 boolean을 리턴하는 함수형 인터페이스로 평가식을 작성
Stream<String> addressStream = address.stream();
List<String> songpa = addressStream.filter(s -> s.contains("송파구")).collect(Collectors.toList());
songpa.stream().forEach(System.out::println);
```

```
서울시 송파구 방이동
서울시 송파구 송파동
```

```
// Map : Stream 요소에 있는 값들을 특정 방식으로 변환 하고 싶을 때 사용
//       변환을 수행 하는 함수를 파라미터로 받는다.
List<String> tmp = address.stream().map(s->s.replaceAll("송파구", "송파")).collect(Collectors.toList());
tmp.stream().forEach(s-> System.out.println(s));
```

```
서울시 송파 방이동
서울시 송파 송파동
서울시 강남구 개포동
서울시 강남구 서초동
```

```
String[][] arrays = new String[][]{ {"a1", "a2"}, {"b1", "b2"}, {"c1", "c2", "c3"} };
Stream<String[]> stream4 = Arrays.stream(arrays);
Stream<String> stream5 = stream4.flatMap(s -> Arrays.stream(s));
stream5.map(String::toUpperCase).forEach(System.out::println);
```

```
A1
A2
B1
B2
C1
C2
C3
```

1. Stream API

02. concat, distinct, limit, skip

- concat : Item을 하나의 스트림으로 합친다
- limit : 일정한 개수 만큼 가져 와서 새로운 스트림 생성
- skip : 일정한 숫자 만큼 item을 건너 띄고 그 이후의 item으로 스트림 생성

```
List<Member> memberList = Arrays.asList(new Member("1", "홍길동"),
                                        new Member("1", "김길자")
                                       );

List<Address> addressList = Arrays.asList(new Address("1", "서울시 송파구"),
                                          new Address("1", "서울시 강동구")
                                    );

Stream<Member> streamMember =  memberList.stream();
Stream<Address> streamAdress = addressList.stream();

Stream<T> streamMemberAddress = (Stream<T>) Stream.concat(streamMember, streamAdress);
streamMemberAddress.forEach(t->{
  if ( t instanceof Address) {
    Address a = (Address) t;
    System.out.println("Address : " + a.getMemberNo() + ", Address : " + a.getAddress());
  } else if (t instanceof Member) {
    Member a = (Member) t;
    System.out.println("MemberNo : " + a.getMemberNo() + ", MemberNm : " + a.getMemberNm());
  }
});
```

1. Stream API

03. sorted

- distinct : 중복되는 item을 모두 제거 하여 새로운 스트림으로 리턴, - equals(), hasCode() 가 재정의 되어 있어야 함
- sorted : Item들을 정렬 하여 새로운 스트림을 생성 - Comparable interface가 구현 되어 있어야 함

```
List<String> asList = Arrays.asList("홍길동", "김길자", "홍길동", "홍상훈", "김길자");

Stream<String> stream1 = asList.stream().distinct();
stream1.forEach(System.out::println);

List<String> langs = Arrays.asList("java", "kotlin", "haskell", "ruby", "smalltalk");
System.out.println("sorted:");
langs.stream().sorted().forEach(System.out::println);

System.out.println("reversed:");
langs.stream().sorted(Comparator.reverseOrder()).forEach(System.out::println);

langs = Arrays.asList("java", "kotlin", "haskell", "ruby", "smalltalk");

System.out.println("sorted:");
langs.stream().sorted(Comparator.comparing(String::length)) .forEach(System.out::println);

System.out.println("reversed:");
langs.stream().sorted(Comparator.comparing(String::length).reversed()).forEach(System.out::println);
```

1. Stream API

04. find

- findfirst : 순서상 가장 첫번째 있는 것을 리턴
- findAny : 순서와 관계 먼저 찾는 객체를 리턴

05. match

- 스트림에서 찾고자 하는 객체가 존재 하는지를 boolean 타입으로 리턴
- anyMatch : 조건에 맞는 객체가 하나라도 있으면 true
- allMatch : 모든 객체가 조건에 맞아야 true
- noneMatch : 조건에 맞는 객체가 없어야 true

06. Collectors

- Collectors.toList() : 작업 결과를 리스트로 반환
- Collectors.joining() : 작업 결과를 하나의 스트링으로 변환
  : delimiter : 각 요소 중간에 들어가 요소를 구분 시켜주는 구분자
  prefix : 결과 맨 앞에 붙는 문자
  suffix : 결과 맨 뒤에 붙는 문자
- Collectors.averageingInt() : 숫자 값(Integer value)의 평균(arithmetic mean)
- Collectors.summingInt() : 숫자값의 합(sum)
- Collectors.summarizingInt() : 합계와 평균
- Collectors.groupingBy() : 특정 조건으로 요소들을 그룹 -> 함수형 인터페이스 Function 을 이용해서 특정 값을 기준으로 스트림 내 요소들을 묶음
- Collectors.partitioningBy() : 함수형 인터페이스 Predicate 를 받습니다. Predicate 는 인자를 받아서 boolean 값을 리턴
- Collectors.collectingAndThen() : 특정 타입으로 결과를 collect 한 이후에 추가 작업이 필요한 경우에 사용

1. Stream API


```
List<ProductInfo> productList =
        Arrays.asList(new ProductInfo(1, "요금상품1", 10000, "P"),
                new ProductInfo(2, "부가상품1", 1000, "R"),
                new ProductInfo(3, "요금상품2", 20000, "P"),
                new ProductInfo(4, "부가상품2", 2000, "R"),
                new ProductInfo(5, "옵션상품1", 3000, "O"));
```

```
List<String> productNmList = productList.stream().map(ProductInfo::getProductNm).collect(Collectors.toList());
productNmList.forEach(System.out::println);
```

```
요금상품1
부가상품1
요금상품2
부가상품2
옵션상품1
```

```
String productNmStr = productList.stream().map(ProductInfo::getProductNm).collect(Collectors.joining());
System.out.println( "productNmStr ::" + productNmStr);
```

```
productNmStr ::요금상품1부가상품1요금상품2부가상품2옵션상품1
```

```
productNmStr = productList.stream().map(ProductInfo::getProductNm).collect(Collectors.joining(",","[","]"));
System.out.println( "productNmStr ::" + productNmStr);
```

```
productNmStr ::[요금상품1,부가상품1,요금상품2,부가상품2,옵션상품1]
```

```
Double priceAvarag = productList.stream().filter(productInfo -> productInfo.getProductType().equals("P"))
        .collect(Collectors.averagingInt(ProductInfo::getPrice));
System.out.println( "priceAvarag ::" + priceAvarag);
```

```
priceAvarag ::15000.0
```

```
Double allPriceAvarag = productList.stream().collect(Collectors.averagingInt(ProductInfo::getPrice));
System.out.println( "allPriceAvarag ::" + allPriceAvarag);
```

```
allPriceAvarag ::7200.0
```

```
int sumP = productList.stream().filter(productInfo -> productInfo.getProductType().equals("P"))
        .collect(Collectors.summingInt(ProductInfo::getPrice));
System.out.println( "sumP ::" + sumP);
```

```
sumP ::30000
```

```
int sum = productList.stream().collect(Collectors.summingInt(ProductInfo::getPrice));
System.out.println( "sum ::" + sum);
```

```
sum ::36000
```

1. Stream API

```
List<ProductInfo> productList =
        Arrays.asList(new ProductInfo(1, "요금상품1", 10000, "P"),
                new ProductInfo(2, "부가상품1", 1000, "R"),
                new ProductInfo(3, "요금상품2", 20000, "P"),
                new ProductInfo(4, "부가상품2", 2000, "R"),
                new ProductInfo(5, "옵션상품1", 3000, "O"));
```

```
IntSummaryStatistics sumavgP = productList.stream().filter(productInfo -> productInfo.getProductType().equals("P"))
        .collect(Collectors.summarizingInt(ProductInfo::getPrice));
System.out.println( "sumavgP ::" + sumavgP);
```

```
sumavgP ::IntSummaryStatistics{count=2, sum=30000, min=10000, average=15000.000000, max=20000}
```

```
IntSummaryStatistics sumavg = productList.stream().collect(Collectors.summarizingInt(ProductInfo::getPrice));
System.out.println( "sumavg ::" + sumavg);
```

```
sumavg ::IntSummaryStatistics{count=5, sum=36000, min=1000, average=7200.000000, max=20000}
```

```
HashMap groupByProduct = (HashMap) productList.stream().collect(Collectors.groupingBy(ProductInfo::getProductType));
groupByProduct.forEach((k,v) -> {
  System.out.println( "key ::" + k);
  List<ProductInfo> values = (List<ProductInfo>) v;
  values.forEach(productInfo -> System.out.println("Values:: " + productInfo.getProductNm()));
});
```

```
key ::P
Values:: 요금상품1
Values:: 요금상품2
key ::R
Values:: 부가상품1
Values:: 부가상품2
key ::O
Values:: 옵션상품1
```

1. Optional

01. Optional

- Null처리를 유연하게 하고자 도입된 객체로 null 객체를 포함한 모든 객체를 포함 할 수 있는 wrapping 하는 객체
- Optional<T> 클래스를 이용해서 NullPointerException 을 방지할 수 있음. Optional<T> 클래스는 한 마디로 null 이 올 수 있는 값을 감싸는 래퍼 클래스로 참조하더라도 null 이 일어나지 않도록 해주는 클래스
- isPresent : 내부 객체가 null 인지 알려 준다
- orElse : Optional이 null인 경우 orElse()의 param이 리턴
- orElseGet : Optional이 null인 경우 어떤 함수를 실행하고 그 실행결과를 대입
- orElseThrow : null인 경우 예외를 던지고 싶을 때

```
class CustomException extends RuntimeException{

  public CustomException() {
    super();
    System.out.println("에러 ......");
  }

}
```

```
optional :: Optional.empty
optional.isEmpty ::true
optional.isPresent ::false
optional result ::Other
optional set result ::Testing....
optional member ::번호 없음
optional null String ::
에러 ......
Exception in thread "main" com.hyomee.streamedu.stream.CustomException Create breakpoint
    at java.base/java.util.Optional.orElseThrow(Optional.java:408)
    at com.hyomee.streamedu.stream.OptionalMain.main(OptionalMain.java:32)
```

```
public static void main(String[] arg) {
  Optional optional = Optional.empty();
  System.out.println( "optional :: " + optional);
  System.out.println( "optional.isEmpty ::" + optional.isEmpty());
  System.out.println( "optional.isPresent ::" + optional.isPresent());

  TextClass textClass = new TextClass();
  Optional<String> op = Optional.ofNullable(textClass.getTest());
  String result = op.orElse("Other");
  System.out.println( "optional result ::" + result);
  textClass.setTest("Testing....");
  op = Optional.ofNullable(textClass.getTest());
  result = op.orElse("Other");
  System.out.println( "optional set result ::" + result);

  MemberAddress memberAddress = new MemberAddress();
  Optional<Member> member = Optional.ofNullable(memberAddress.getMember());
  Optional<String> memberNo = member.map(Member::getMemberNo);
  result = memberNo.orElse("번호 없음");

  System.out.println( "optional member ::" + result);
  result = memberNo.orElseGet(()-> new String());
  System.out.println( "optional null String :: " + result);
  result = memberNo.orElseThrow(CustomException::new);
}
```