

# 씹고 뜯고 맛보고 즐기는 스트림 API

KSUG & 지앤선이 함께하는 테크니컬 세미나 - KSUG에게 듣는 자바 8 이야기

2 RVP100

465 3 RVP100



# 박용권 KSUG 일꾼단

: 한국 스프링 사용자 모임(KSUG)  
: 봄싹(SpringSprout)  
: 라 스칼라 코딩단

: twitter / @arawnkr

내 연락처에 등록된 사람들 중  
플로리다에 사는 남자들의 평균 나이는?

# 명시적 반복을 통해 요구사항 구현...

```
List<Contact> contacts = ContactSource.findAll();

int manCount = 0;
int totalAge = 0;
for(Contact contact : contacts) {
    if("Florida".equals(contact.getState())
        && Gender.Male == contact.getGender()) {
        manCount += 1;
        totalAge += contact.getAge();
    }
}

double averageAge = totalAge / manCount;
```

# 명시적 반복을 통해 요구사항 구현...

```
List<Contact> contacts = ContactSource.findAll();

int manCount = 0;
int totalAge = 0;
for(Contact contact : contacts) {
    if("Florida".equals(contact.getState())
        && Gender.Male == contact.getGender()) {
        manCount += 1;
        totalAge += contact.getAge();
    }
}

double averageAge = totalAge / manCount;
```

**명령형** 프로그래밍



# 명시적 반복을 통해 요구사항 구현...

```
List<Contact> contacts = ContactSource.findAll();

int manCount = 0;
int totalAge = 0;
for(Contact contact : contacts) {
    if("Florida".equals(contact.getState())
        && Gender.Male == contact.getGender()) {
        manCount += 1;
        totalAge += contact.getAge();
    }
}

double averageAge = totalAge / manCount;
```

어떻게가 아닌, 무엇을 계산하는지가 중요



# 스트림 API와 람다 표현식으로 요구사항 구현...

```
List<Contact> contacts = ContactSource.findAll();

contacts.stream()
    .filter(c -> "Florida".equals(c.getState()))
    .filter(c -> Gender.Male == c.getGender())
    .mapToInt(c -> c.getAge())
    .average();
```

# 스트림 API와 람다 표현식으로 요구사항 구현...

```
List<Contact> contacts = ContactSource.findAll();
```

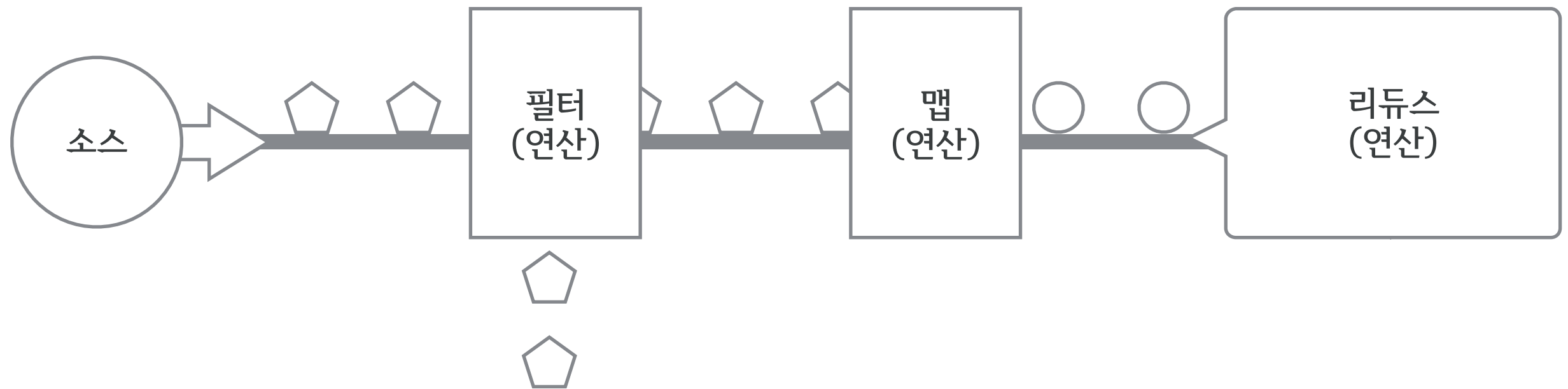
```
contacts.stream()  
    .filter(c -> "Florida".equals(c.getState()))  
    .filter(c -> Gender.Male == c.getGender())  
    .mapToInt(c -> c.getAge())  
    .average();
```



파이프-필터 기반 API



# 스트림 API의 처리 흐름



# 스트림과 콜렉션은 다르다

## Collection

```
for(Contact c : contacts) { ... }
```

- ✓ 외부 반복(External Iteration)
- ✓ 반복을 통해 생성
- ✓ 효율적이고 직접적인 요소 처리
- ✓ 유한 데이터 구조
- ✓ 반복적 재사용 가능

## Stream

```
contacts.forEach(c -> { ... })
```

- ✓ 내부 반복(Internal Iteration)
- ✓ 반복을 통해 연산
- ✓ 파이프-필터 기반 API
- ✓ 무한 연속 흐름 데이터 구조
- ✓ 재사용 불가

# 스트림 API 맛보기

# 스트림 API의 목적

데이터 보관이 아닌  
데이터 처리에 집중

# 스트림 API의 특징

## 반복의 내재화

- ✓ 반복 구조 캡슐화(제어 흐름 추상화, 제어의 역전)
- ✓ 최적화와 알고리즘 분리

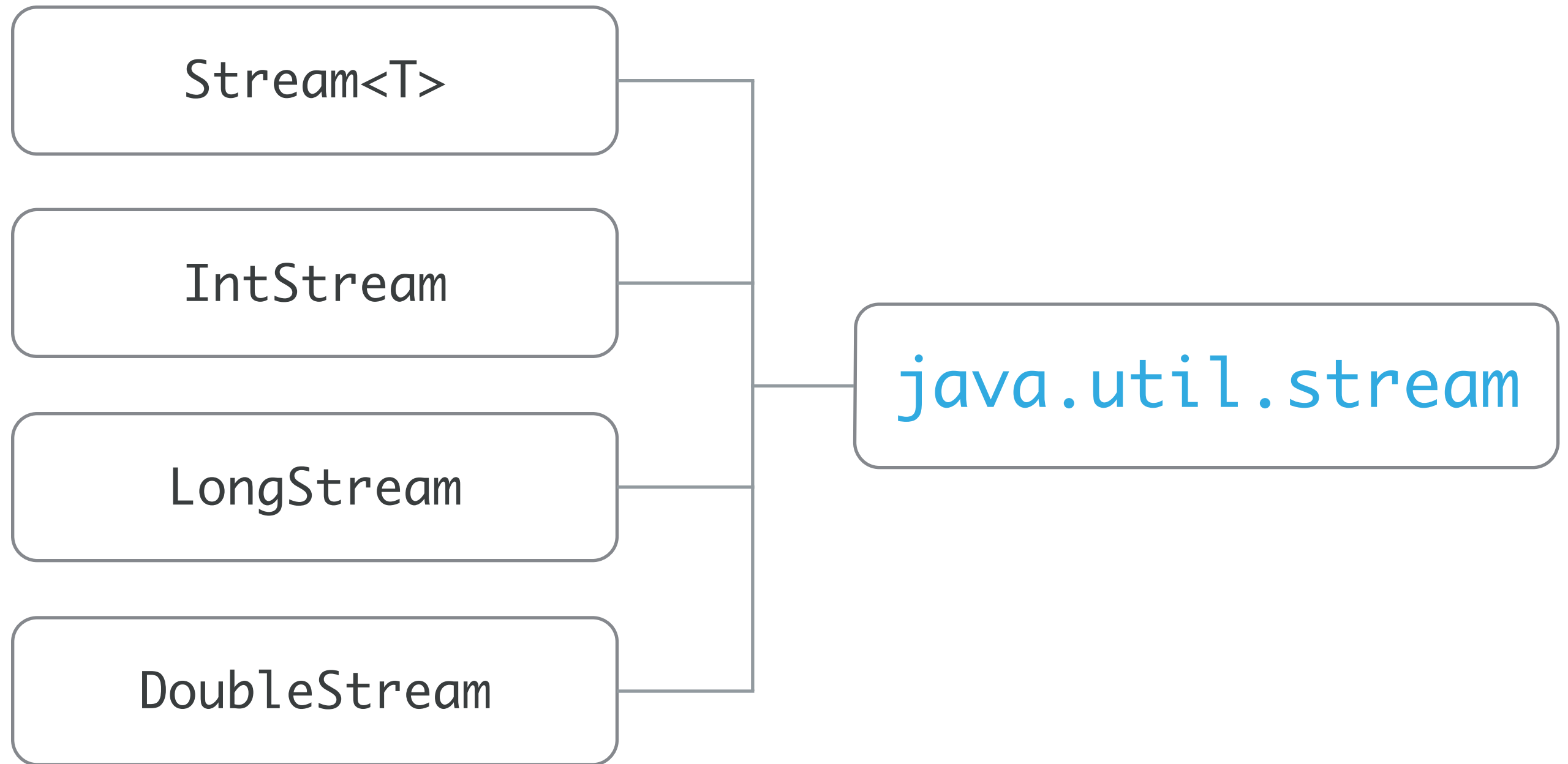
## 지연 연산

- ✓ 스트림을 반환하는 필터-맵 API는 기본적으로 지연(lazy) 연산
- ✓ 지연 연산을 통한 성능 최적화(무상태 중개 연산 및 반복 작업 최소화)

## 병렬 처리

- ✓ 동일한 코드로 순차 또는 병렬 연산을 쉽게 처리
- ✓ 쓰레드에 안전하지 않은 컬렉션도 병렬처리 지원
- ✓ 단, 스트림 연산 중 데이터 원본을 변경하면 안된다

# 스트림 인터페이스: Stream interface



# 스트림 인터페이스: Stream interface

Stream<T>

✓ 객체를 요소로 하는 범용 스트림

IntStream

✓ int를 요소로 하는 스트림

LongStream

✓ long을 요소로 하는 스트림

DoubleStream

✓ double을 요소로하는 스트림

# 스트림 인터페이스: Stream interface

Stream<T>

✓ 객체를 요소로 하는 범용 스트림

IntStream

✓ int를 요소로 하는 스트림

LongStream

✓ long을 요소로 하는 스트림

DoubleStream

✓ double을 요소로 하는 스트림



# 2가지 유형의 스트림 인터페이스



# 함수형 인터페이스: Functional interfaces

## java.util.function.\*

- ✓ 스트림 연산자의 대부분은 인수로 함수형 인터페이스를 받음
- ✓ 함수형 인터페이스는 람다 표현식 또는 메소드 표현식으로 사용

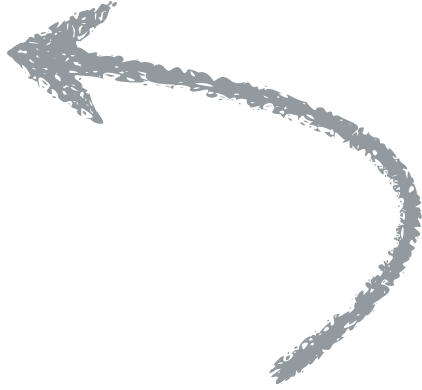
함수형 인터페이스	설명
Supplier<T>	T 타입 값을 공급한다.
Consumer<T>	T 타입 값을 소비한다.
BiConsumer<T, U>	T와 U 타입 값을 소비한다.
Predicate<T>	boolean 값을 반환한다.
ToIntFunction<T> ToLongFunction<T> ToDoubleFunction<T>	T 타입을 인자로 받아 int, long, double 값을 반환한다.
IntFunction<R> LongFunction<R> DoubleFunction<R>	int, long, double 값을 인자로 받아 R 타입 값을 반환한다.
Function<T, R>	T 타입을 인자로 받고 R 타입을 반환한다.
BiFunction<T, U, R>	T와 U 타입을 인자로 받고 R 타입을 반환한다.
UnaryOperator<T>	T 타입에 적용되는 단항 연산자다.
BinaryOperator<T>	T 타입에 적용되는 이항 연산자다.

# 신뢰할 수 없는 반환값

```
List<Contact> contacts = ContactSource.findAll();
```

```
Contact contact = contacts.stream()  
    .filter(c -> "Florida".equals(c.getState()))  
    .findFirst();
```

```
contact.call();
```



스트림에서 필터 연산 후  
연락처(Contact) 객체를 찾아줘-

# 신뢰할 수 없는 반환값

```
List<Contact> contacts = ContactSource.findAll();
```

```
Contact contact = contacts.stream()  
    .filter(c -> "Florida".equals(c.getState()))  
    .findFirst();
```

```
contact.call();
```



**contact == null이면...?**

**NullPointerException이 발생!**

# 신뢰할 수 없는 반환값

```
List<Contact> contacts = ContactSource.findAll();
```

```
Contact contact = contacts.stream()  
    .filter(c -> "Florida".equals(c.getState()))  
    .findFirst();
```

```
contact.call();
```

```
if (contact != null) {  
    contact.call();  
}
```



**방어코드로 회피...**

# 신뢰할 수 있는 반환값

## java.util.Optional<T>

- ✓ Java 8에서 새롭게 추가된 클래스
- ✓ 연산 후 반환값이 있을 수도 없을 수도 있을 때 사용
- ✓ 반환값이 객체 또는 null인 T 보다 안전한 대안

# 신뢰할 수 있는 반환값

## java.util.Optional<T>

```
List<Contact> contacts = ContactSource.findAll();
```

```
Optional<Contact> optional = contacts  
    .stream()  
    .filter(c -> "Florida".equals(c.getState()))  
    .findFirst();
```

```
if (optional.isPresent()) {  
    optional.get().call();  
}
```




값이 있으면 참(true)을  
값이 없으면 거짓(false)을 반환

# 신뢰할 수 있는 반환값

## java.util.Optional<T>

```
List<Contact> contacts = ContactSource.findAll();
```

```
Optional<Contact> optional = contacts  
    .stream()  
    .filter(c -> "Florida".equals(c.getState()))  
    .findFirst();
```



```
if (optional.isPresent()) {  
    optional.get().call();  
}
```

**반환값이 없을 수도 있어,  
그러니 꼭 확인해봐-!**



# 신뢰할 수 있는 반환값

java.util.Optional<T>

```
List<Contact> contacts = ContactSource.findAll();
```

```
contacts.stream()  
    .filter(contact -> "Florida".equals(c.getState()))  
    .findFirst()  
    .ifPresent(contact -> contact.call());
```



값이 있으면 넘겨준 람다를 실행해줘-

# 신뢰할 수 있는 반환값

## java.util.Optional<T>

```
List<Contact> contacts = ContactSource.findAll();
```

```
contacts.stream()  
    .filter(contact -> "Florida".equals(c.getState()))  
    .findFirst()  
    .ifPresent(contact -> contact.call());
```

```
Contact findFirst = contacts.stream()  
    .filter("Florida".equals(c.getState()))  
    .findFirst()  
    .orElseGet(() -> Contact.empty());
```



**값이 있으면 정상 값을 반환하고,  
없으면 기본 값을 반환해줘!**

# 분할 반복자 인터페이스: Splitter interface

## java.util.Spliterator<T>

- ✓ 스트림 API에 사용되는 새로운 클래스
- ✓ 병렬 실행시 컬렉션의 데이터 요소를 분할 처리

```
List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8);
```

```
Spliterator<Integer> firstSplit = numbers.split();  
Spliterator<Integer> secondSplit = firstSplit.trySplit();
```

```
// firstSplit = 5, 6, 7, 8  
// secondSplit = 1, 2, 3, 4
```

# 스트림 API의 3단계

```
orders.stream().map(n->n.price).sum();
```

스트림 생성

중개 연산  
(스트림 변환)

최종 연산  
(스트림 사용)

# 스트림 API의 3단계

```
orders.stream().map(n->n.price).sum();
```

스트림 생성

중개 연산  
(스트림 변환)

최종 연산  
(스트림 사용)

# 스트림 API의 3단계

## 1단계: 스트림 생성

- ✓ Collection `streams()`, `parallelStream()`
- ✓ Arrays `streams(*)`
- ✓ Stream ranges `range(...)`, `rangeClosed(...)`
- ✓ Directly from values `of(*)`
- ✓ Generators `iterate(...)`, `generate(...)`
- ✓ Resources `lines()`

# 스트림 API의 3단계

## 1단계: 스트림 생성

// 컬렉션(List)으로 순차 스트림 생성

```
List<String> collection = new ArrayList<>();  
Stream<String> stream = collection.stream();
```

// 배열을 스트림으로 변환

```
IntStream stream = Arrays.stream(numbers);
```

// of 메소드는 가변인자로 스트림을 생성 가능

```
Stream<String> stream = Stream.of("Using", "Stream", "API", "From", "Java8");
```

// 1부터 10까지 유한 스트림 생성

```
LongStream longStream = LongStream.rangeClosed(1, 10);
```

# 스트림 API의 3단계

## 1단계: 스트림 생성

// 컬렉션(List)으로 순차 스트림 생성

```
List<String> collection = new ArrayList<>();  
Stream<String> stream = collection.stream();
```

**순차 스트림 생성**

// 배열을 스트림으로 변환

```
IntStream stream = Arrays.stream(numbers);
```

// of 메소드는 가변인자로 스트림을 생성 가능

```
Stream<String> stream = Stream.of("Using", "Stream", "API", "From", "Java8");
```

// 1부터 10까지 유한 스트림 생성

```
LongStream longStream = LongStream.rangeClosed(1, 10);
```



# 스트림 API의 3단계

```
orders.stream().map(n->n.price).sum();
```

스트림 생성

중개 연산  
(스트림 변환)

최종 연산  
(스트림 사용)

# 스트림 API의 3단계

## 2단계: 중개 연산자: Intermediate Operator

- ✓ 스트림을 받아서 스트림을 반환
- ✓ 무상태 연산과 내부 상태 유지 연산으로 나누어짐
- ✓ 기본적으로 지연(lazy) 연산 처리(성능 최적화)

# 스트림 API의 3단계

## 2단계: 중개 연산자: Intermediate Operator

- ✓ 스트림을 받아서 스트림을 반환
- ✓ 무상태 연산과 내부 상태 유지 연산으로 나누어짐
- ✓ 기본적으로 지연(lazy) 연산 처리(성능 최적화)

```
contacts.stream()  
    .filter(c -> "Florida".equals(c.getState()))  
    .filter(c -> Gender.Male == c.getGender())  
    .mapToInt(c -> c.getAge());
```

최종 연산자 실행 전에는  
연산 처리하지 않는다

내가 바로 중개 연산자!



# 스트림 API의 3단계

## 2단계: 중개 연산자: Intermediate Operator

- ✓ 스트림을 받아서 스트림을 반환
- ✓ 무상태 연산과 내부 상태 유지 연산으로 나누어짐
- ✓ 기본적으로 **지연(lazy) 연산 처리**(성능 최적화)

```
contacts.stream()  
    .filter(c -> "Florida".equals(c.getState()))  
    .filter(c -> Gender.Male == c.getGender())  
    .mapToInt(c -> c.getAge());
```

```
for(Contact contact : contacts) {  
    filter(...);  
    filter(...);  
    mapToInt(...);  
}
```



연산자 별로 처리를 하지 않고,  
**모든 연산은 한번에 처리**

# 스트림 API의 3단계

```
orders.stream().map(n->n.price).sum();
```

스트림 생성

중개 연산  
(스트림 변환)

최종 연산  
(스트림 사용)

# 스트림 API의 3단계

## 3단계: 최종 연산자: Terminal operator

- ✓ 스트림의 요소들을 연산 후 결과 값을 반환
- ✓ 최종 연산 시 모든 연산 수행 (반복 작업 최소화)
- ✓ 이후 더 이상 스트림을 사용할 수 없음

```
contacts.stream()  
    .filter(c -> "Florida".equals(c.getAddress().getState()))  
    .filter(c -> Gender.Male == c.getGender())  
    .mapToInt(c -> c.getAge())  
    .average();
```



평균 값을 계산 후 반환하는 최종 연산자

지금까지 맛보기였습니다.

# 스트림 API 활용편



# “스트림 API”

## 활용편

1. 뉴욕 주로 등록된 연락처만 출력
2. 연락처에서 이메일만 출력
3. 1부터 100까지 합계 구하기
4. 사람들의 나이 합계, 평균, 최소, 최대 구하기
5. 연락처에서 도시 목록 구하기
6. 주 별로 연락처를 분류하기

# [경고] 코드 여러 번 등장

정신줄 꼭 붙잡으셔야 합니다.

# 뉴욕 주로 등록된 연락처만 출력

```
List<Contact> contacts = ContactSource.findAll();
```

```
// for 구문
```

```
for(Contact contact : contacts) {  
    if(contact.equalToState("New York")) {  
        System.out.print(contact);  
    }  
}
```

# 뉴욕 주로 등록된 연락처만 출력

```
List<Contact> contacts = ContactSource.findAll();
```

```
// for 구문
```

```
for(Contact contact : contacts) {  
    if(contact.equalToState("New York")) {  
        System.out.print(contact);  
    }  
}
```

```
class Contact {
```

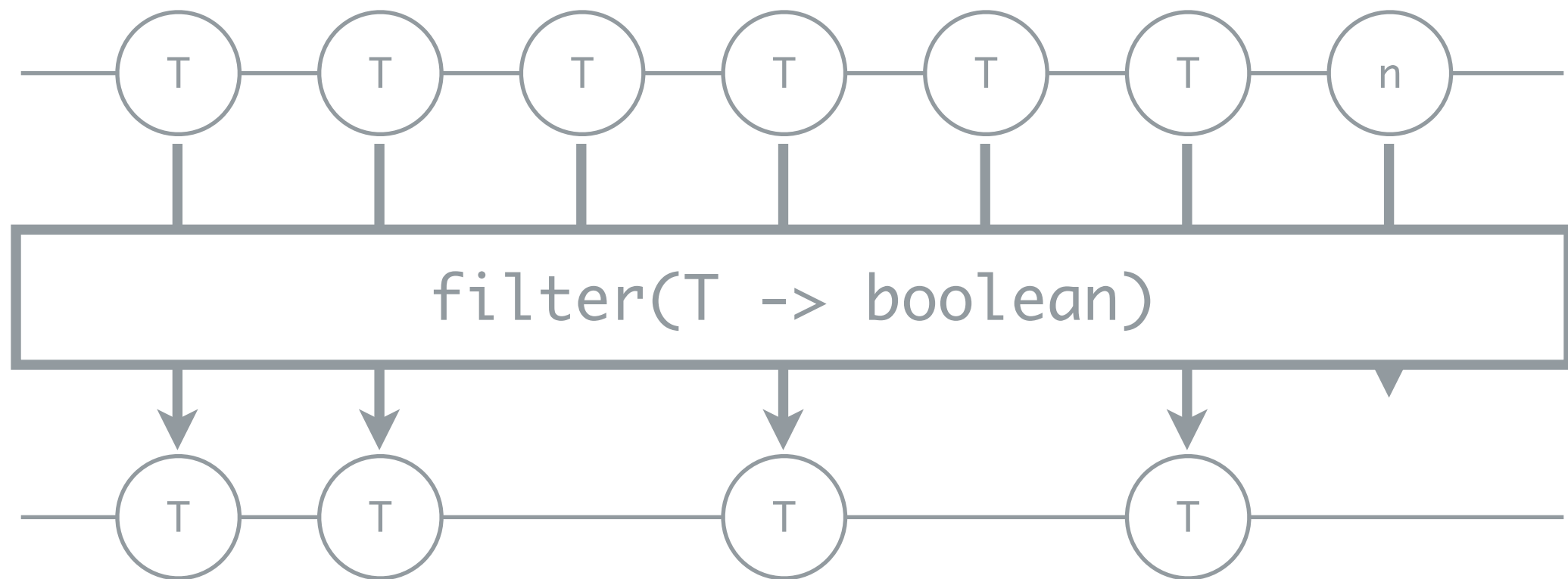
```
    public boolean equalToState(String state) {  
        return Objects.equals(state, getState());  
    }
```

```
}
```

# 조건을 만족하는 요소로 구성된 스트림을 얻을 때

## 중개 연산: `Stream<T> filter(T -> boolean)`

- ✓ 각 요소를 확인해서 조건을 통과한 요소만으로 새 스트림 생성
- ✓ 참/거짓을 반환하는 조건식을 인수로 전달



# 뉴욕 주로 등록된 연락처만 출력

```
List<Contact> contacts = ContactSource.findAll();
```

```
// for 구문
```

```
for(Contact contact : contacts) {  
    if(contact.equalToState("New York")) {  
        System.out.print(contact);  
    }  
}
```

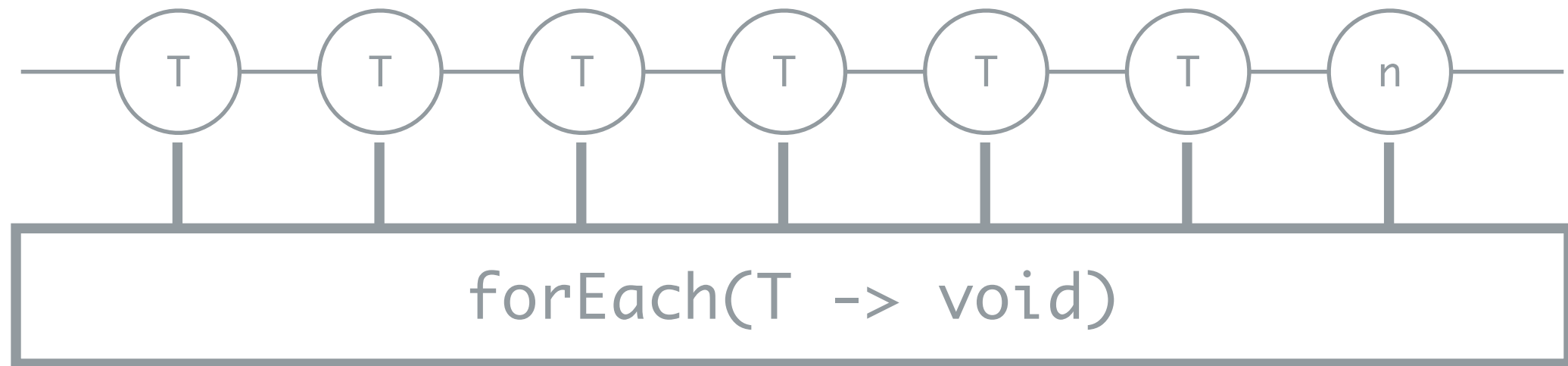
```
// Stream API(filter) + Lambda Expressions
```

```
contacts.stream()  
    .filter(contact -> contact.equalToState("New York"))
```

# 각 요소별로 어떤 처리를 하고 싶을 때

## 최종 연산: `void forEach(T -> void)`

- ✓ 각 요소를 인수로 전달된 함수에 전달해 처리
- ✓ 최종 연산이기 때문이 이후 스트림을 사용할 수 없음



# 뉴욕 주로 등록된 연락처만 출력

```
List<Contact> contacts = ContactSource.findAll();
```

```
// for 구문
```

```
for(Contact contact : contacts) {  
    if(contact.equalToState("New York")) {  
        System.out.print(contact);  
    }  
}
```

```
// Stream API(filter, forEach) + Lambda Expressions
```

```
contacts.stream()  
    .filter(contact -> contact.equalToState("New York"))  
    .forEach(contact -> System.out.println(contact));
```



# 연락처에서 이메일만 출력

```
List<Contact> contacts = ContactsSource.contacts();
```

```
// for 구문
```

```
for(Contact contact : contacts) {  
    System.out.println(contact.getEmail());  
}
```

# 연락처에서 이메일만 출력

```
List<Contact> contacts = ContactsSource.contacts();
```

```
// for 구문
```

```
for(Contact contact : contacts) {  
    System.out.println(contact.getEmail());  
}
```

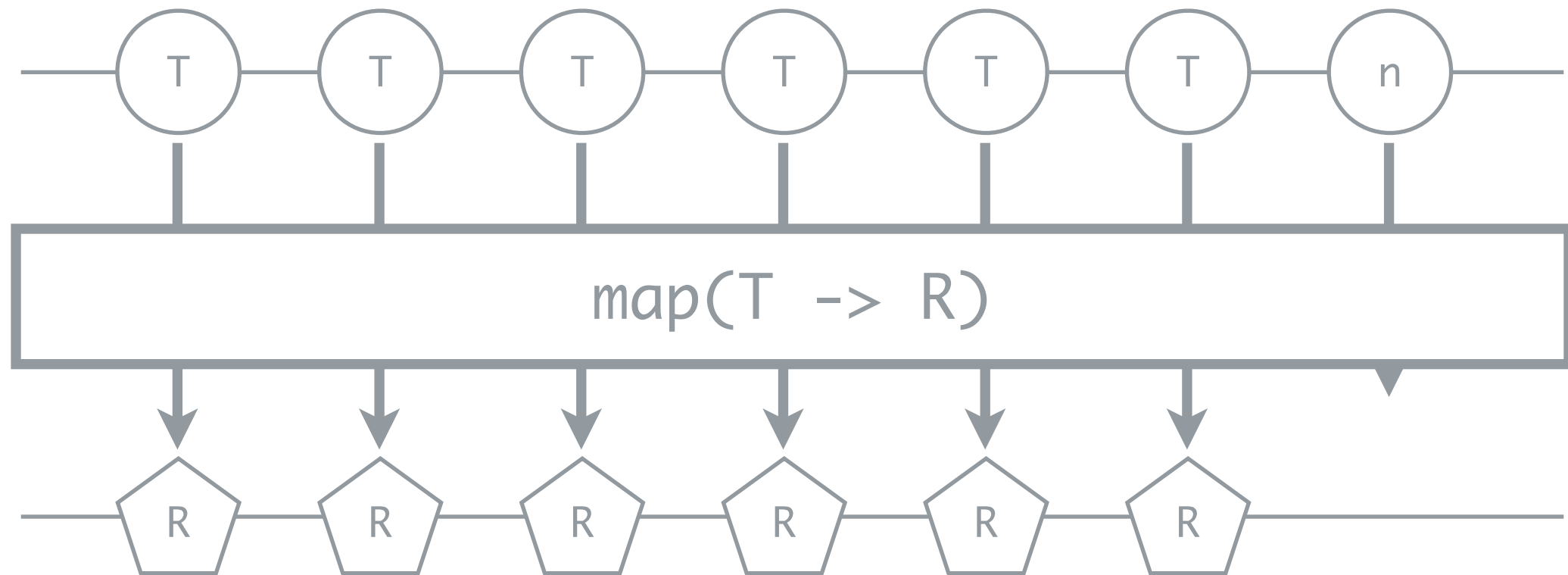
```
// Stream API + Lambda Expressions
```

```
contacts.stream()  
    .forEach(contact -> System.out.println(contact.getEmail()));
```

# 스트림에 있는 값들을 변환하고 싶을 때

## 중개 연산: `Stream<R> map(T -> R)`

- ✓ T 타입의 요소를 1:1로 R 타입의 요소로 변환 후 스트림 생성
- ✓ 변환을 처리하는 함수를 인수로 전달



# 연락처에서 이메일만 출력

```
List<Contact> contacts = ContactsSource.contacts();
```

```
// for 구문
```

```
for(Contact contact : contacts) {  
    System.out.println(contact.getEmail());  
}
```

```
// Stream API + Lambda Expressions
```

```
contacts.stream()  
    .forEach(contact -> System.out.println(contact.getEmail()));
```

```
// Stream API + Lambda Expressions
```

```
contacts.stream()  
    .map(contact -> contact.getEmail())  
    .forEach(email -> System.out.println(email));
```

# 연락처에서 이메일만 출력

```
List<Contact> contacts = ContactsSource.contacts();
```

```
// for 구문
for(Contact contact : contacts) {
    System.out.println(contact.getEmail());
}
```

```
// Stream API + Lambda Expressions
contacts.stream()
    .forEach(contact -> System.out.println(contact.getEmail()));
```

```
// Stream API + Lambda Expressions
contacts.stream()
    .map(contact -> contact.getEmail())
    .forEach(email -> System.out.println(email));
```

```
// Stream API + Method References
contacts.stream()
    .map(Contact::getEmail)
    .forEach(System.out::println);
```

**메소드 참조:**

**람다 표현식의 다른 형태,  
class::method 로 사용**

# 1부터 100까지 합계 구하기

```
// for 구문
int sum = 0;
for(int number = 1; number <= 100; number++) {
    sum += number;
}
```

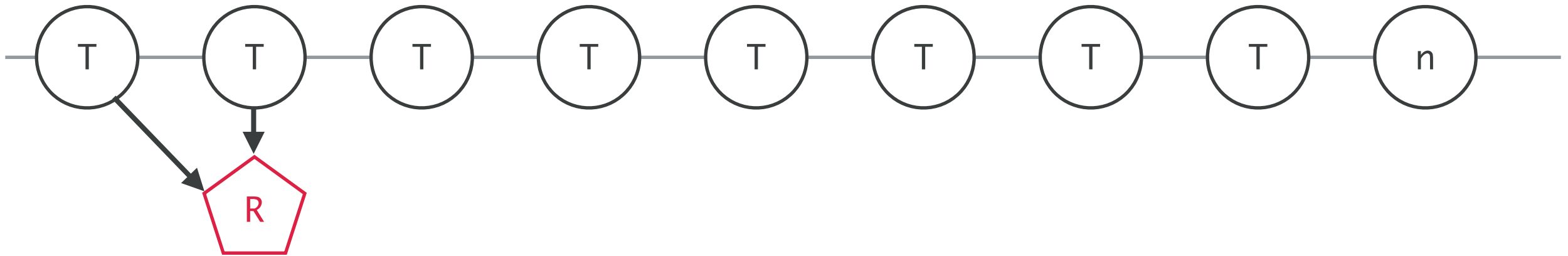
# 스트림의 값들을 결합 또는 계산하고 싶을 때

## 최종 연산: reduce

- ✓ `Optional<T> reduce((T, T) -> T)`
- ✓ `T reduce(T, (T, T) -> T)`
- ✓ T 타입의 요소 둘 씩 reducer로 계산해 최종적으로 하나의 값을 계산

# 스트림의 값들을 결합 또는 계산하고 싶을 때

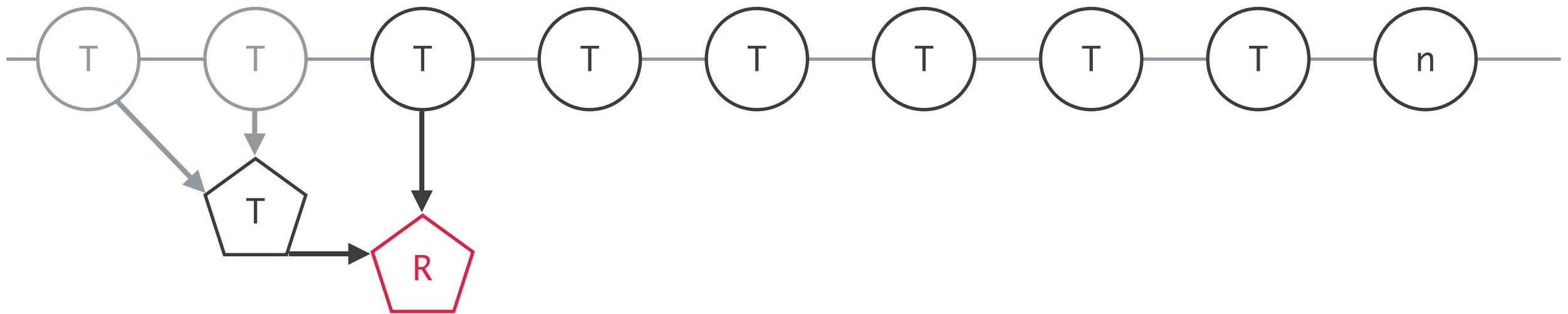
## 최종 연산: reduce 연산 흐름





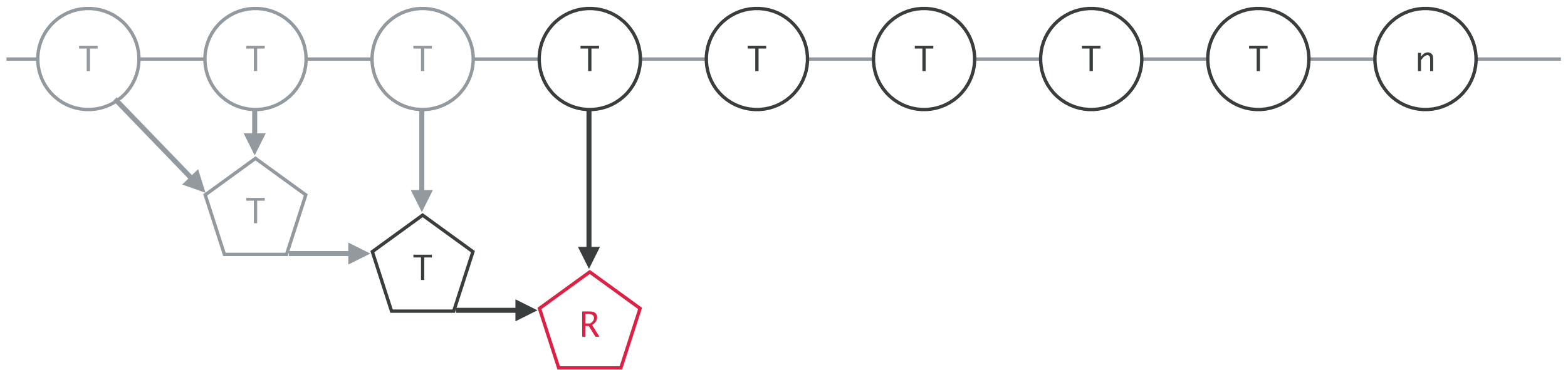
# 스트림의 값들을 결합 또는 계산하고 싶을 때

## 최종 연산: reduce 연산 흐름



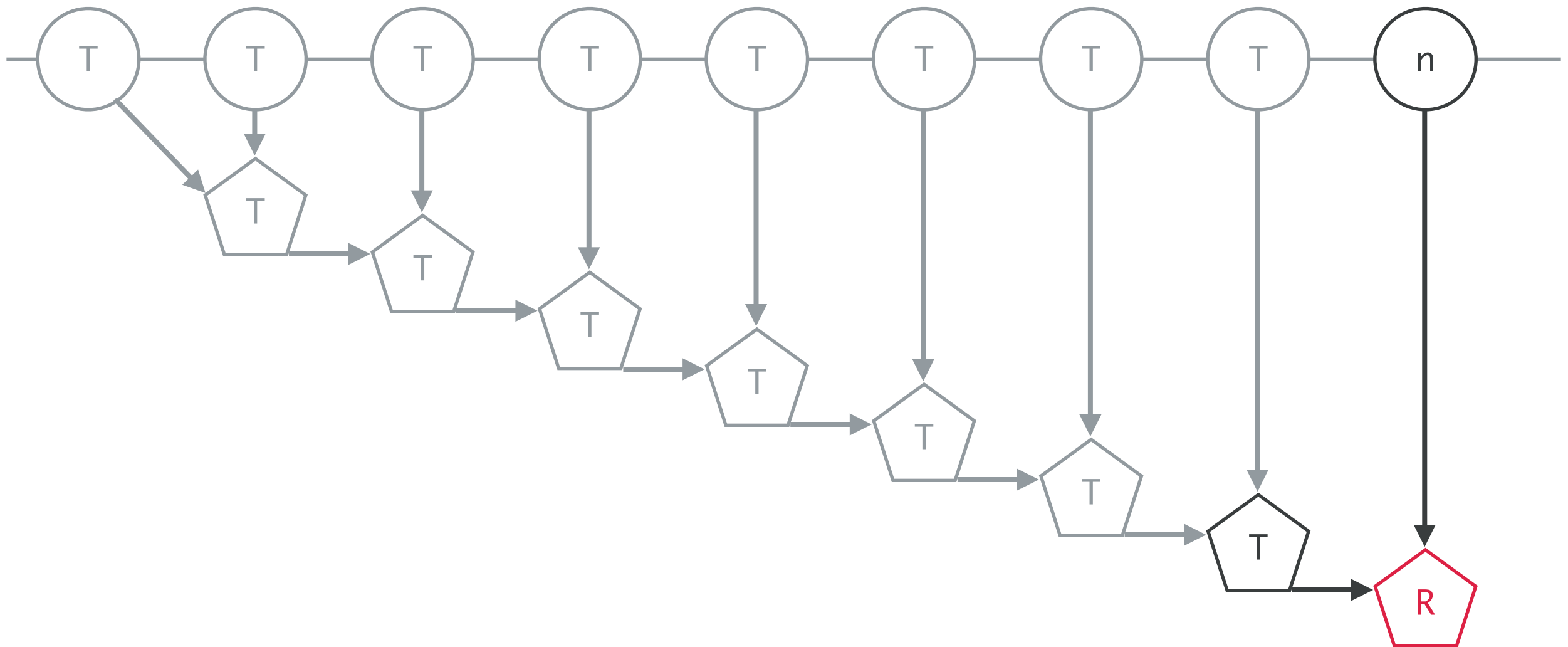
# 스트림의 값들을 결합 또는 계산하고 싶을 때

최종 연산: reduce 연산 흐름



# 스트림의 값들을 결합 또는 계산하고 싶을 때

## 최종 연산: reduce 연산 흐름



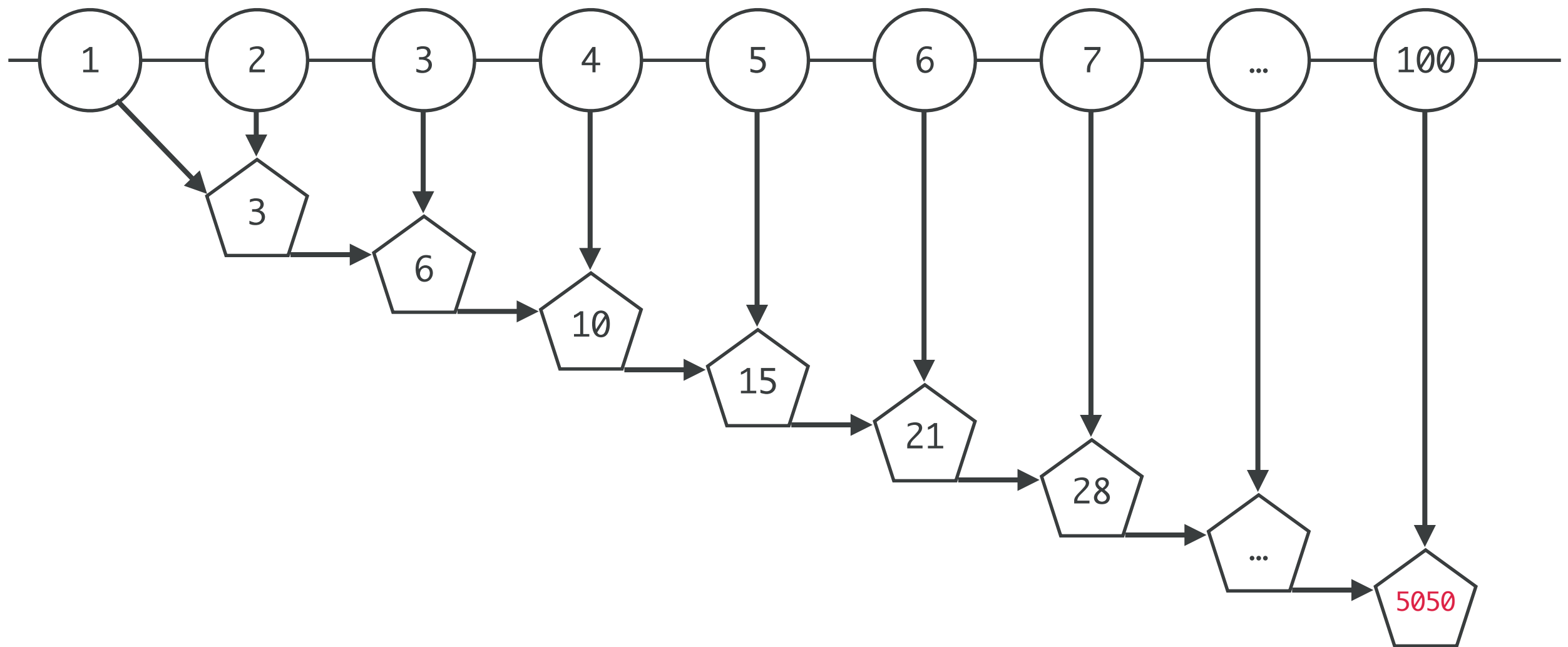
# 1부터 100까지 합계 구하기

```
// for 구문
int sum = 0;
for(int number = 1; number <= 100; number++) {
    sum += number;
}
```

```
// Stream API + Lambda Expressions
int sum = IntStream.rangeClosed(1, 100)
    .reduce(0, (left, right) -> left + right);
```

# 1부터 100까지 합계 구하기

$(1 \text{ to } 100).reduce((l, r) \rightarrow l + r)$



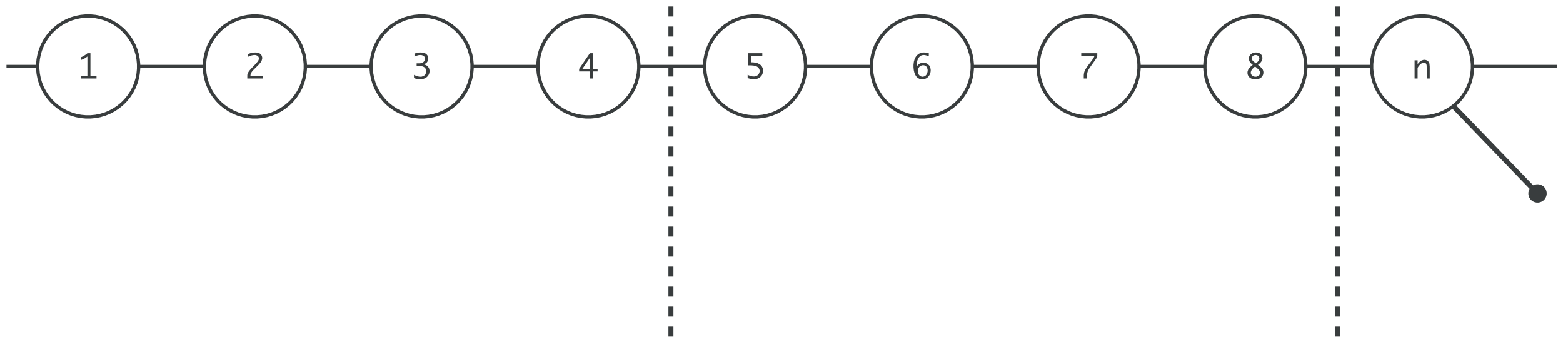
# 1부터 100까지 합계 구하기

```
// for 구문
int sum = 0;
for(int number = 1; number <= 100; number++) {
    sum += number;
}
```

```
// Stream API + Lambda Expressions
int sum = IntStream.rangeClosed(1, 100)
    .parallel()
    .reduce(0, (left, right) -> left + right);
```

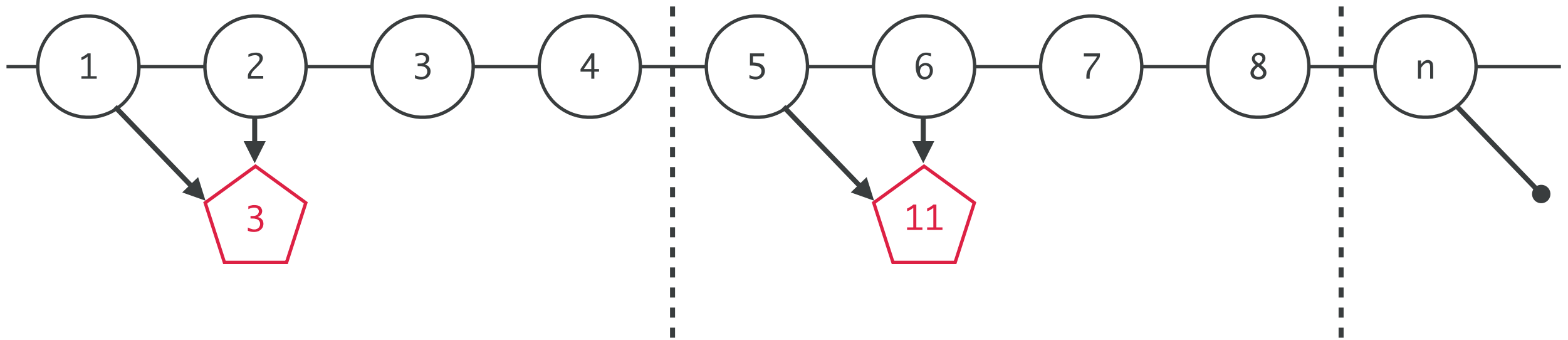
# 1부터 100까지 합계 구하기

병렬 스트림을 통한 reduce 연산 흐름



# 1부터 100까지 합계 구하기

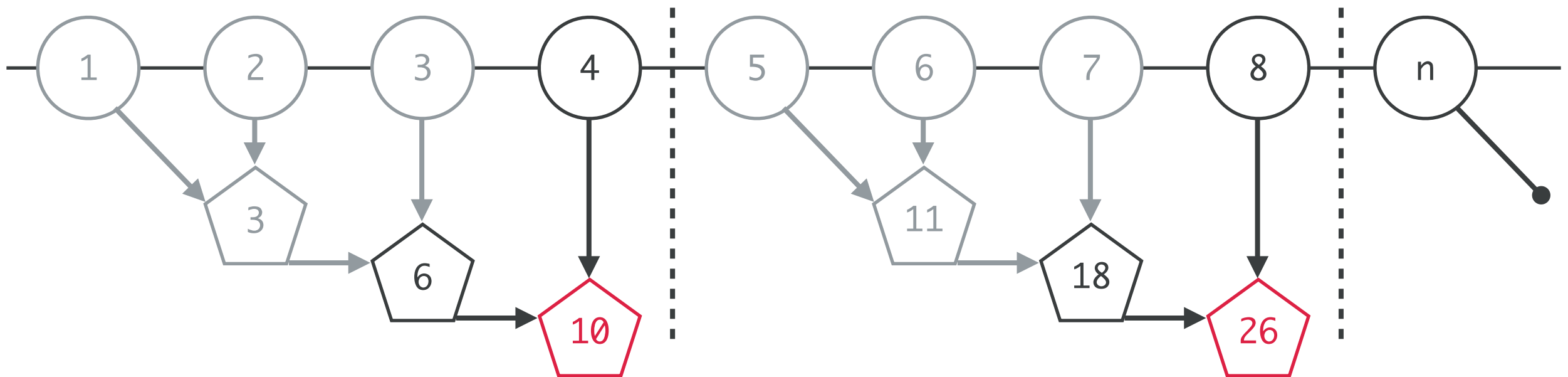
병렬 스트림을 통한 reduce 연산 흐름





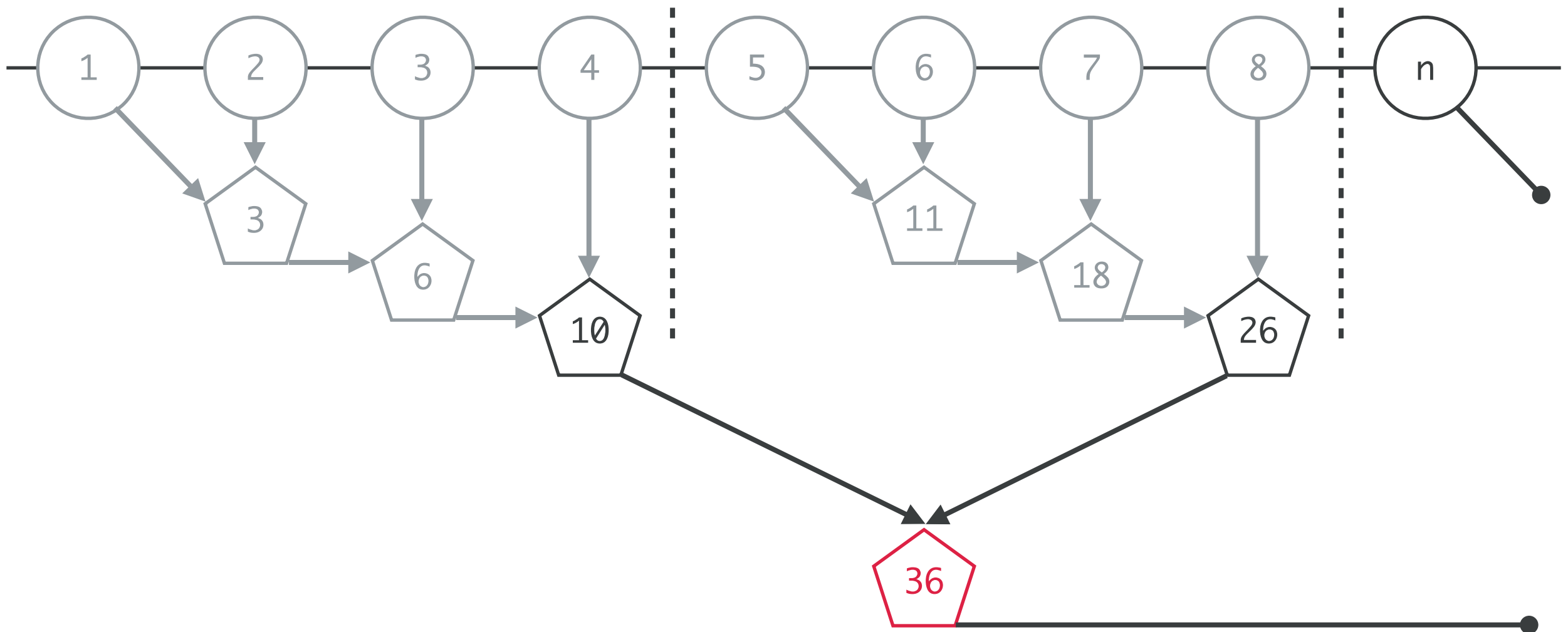
# 1부터 100까지 합계 구하기

병렬 스트림을 통한 reduce 연산 흐름



# 1부터 100까지 합계 구하기

병렬 스트림을 통한 reduce 연산 흐름



병렬 처리를 손 안대고 코풀듯이 할 수 있어요!

# 1부터 100까지 합계 구하기

```
// for 구문
int sum = 0;
for(int number = 1; number <= 100; number++) {
    sum += number;
}
```

```
// Stream API + Lambda Expressions
int sum = IntStream.rangeClosed(1, 100)
    .reduce(0, (left, right) -> left + right);
```

```
// Stream API + Method References
int sum = IntStream.rangeClosed(1, 100)
    .reduce(0, Integer::sum);
```

두 개의 수를 받아 **합계**를  
계산하는 메소드-

JDK에 구현되어 있는  
**다양한 메소드를 활용**하면 좋아요!

# 사람들의 나이 합계, 평균, 최소, 최대 구하기

```
List<Contact> contacts = ContactSource.findAll();
```

```
long sum = 0;
int min = 0, max = 0;
for(Contact contact : contacts) {
    int age = contact.getAge();
    sum += age;
    min = Math.min(min, age);
    max = Math.max(max, age);
}
double average = sum / contacts.size();
```

# 기본 타입 스트림이 제공하는 편리한 계산식

최종 연산: `sum()` | `average()` | `min()` | `max()`

- ✓ 합계 - `[int, long, double] sum()`
- ✓ 평균 - `OptionalDouble average()`
- ✓ 최소값 - `Optional[Int, Long, Double] min()`
- ✓ 최대값 - `Optional[Int, Long, Double] max()`

# 사람들의 나이 합계, 평균, 최소, 최대 구하기

```
List<Contact> contacts = ContactSource.findAll();
```

```
// for 구문
long sum = 0;
int min = 0, max = 0;
for(Contact contact : contacts) {
    int age = contact.getAge();
    sum += age;
    min = Math.min(min, age);
    max = Math.max(max, age);
}
double average = sum / contacts.size();
```

**4번이나 반복하다니!**



// Stream API + Method References

```
int sum = contacts.stream().mapToInt(Contact::getAge).sum();
OptionalDouble average = contacts.stream().mapToInt(Contact::getAge).average();
OptionalInt min = contacts.stream().mapToInt(Contact::getAge).min();
OptionalInt max = contacts.stream().mapToInt(Contact::getAge).max();
```

# 기본 타입 스트림이 제공하는 편리한 계산식

## 최종 연산: `summaryStatistics()`

- ✓ `[Int, Long, Double]SummaryStatistics summaryStatistics()`
- ✓ 합계, 평균, 최소값, 최대값, 개수에 대한 요약 통계

# 사람들의 나이 합계, 평균, 최소, 최대 구하기

```
List<Contact> contacts = ContactSource.findAll();
```

```
// Stream API + Method References  
IntSummaryStatistics summaryStatistics =  
    contacts.stream()  
        .mapToInt(Contact::getAge)  
        .summaryStatistics();
```

```
long sum          = summaryStatistics.getSum();  
double average    = summaryStatistics.getAverage();  
int min           = summaryStatistics.getMin();  
int max           = summaryStatistics.getMax();  
long count        = summaryStatistics.getCount();
```



# 연락처에서 도시 목록 구하기

```
List<Contact> contacts = ContactSource.findAll();
```

```
// for 구문
```

```
List<String> cities = new ArrayList<>();  
for(Contact contact : contacts) {  
    String city = contact.getCity();  
    cities.add(contact.getCity());  
}
```

# 스트림 연산 후 결과를 살펴보고 싶을 때

## 최종 연산: 집계

- ✓ `iterator()`
- ✓ `Object[] toArray()`
- ✓ `A[] toArray(IntFunction<A[]>)`
- ✓ `R collect() -> R, (R, T) -> R, (R, R) -> void`
  - `: () -> R` = 공급자, 대상 객체의 새로운 인스턴스 생성(ex, ArrayList 생성)
  - `: (R, T) -> R` = 누산자, 요소를 대상에 추가(ex, List.add(...))
  - `: (R, R) -> void` = 결합자, 두 객체를 하나로 병합(ex, List.addAll(...))
- ✓ `R collect(Collector<T, ?, R>)`

# 연락처에서 도시 목록 구하기

```
List<Contact> contacts = ContactSource.findAll();
```

```
// for 구문
```

```
List<String> cities = new ArrayList<>();  
for(Contact contact : contacts) {  
    String city = contact.getCity();  
    cities.add(contact.getCity());  
}
```

```
// 반복자로 집계
```

```
Iterator<Contact> contactIterator = contacts.stream().iterator();
```

```
// 배열로 집계
```

```
Object[] objectArray = contacts.stream().toArray();  
Contact[] contactArray = contacts.stream().toArray(Contact[]::new);
```

# 연락처에서 도시 목록 구하기

```
List<Contact> contacts = ContactSource.findAll();
```

```
// Stream API(collect(공급자, 누산자, 결합자)) + Lambda Expressions
```

```
cities = contacts.stream()  
    .map(contact -> contact.getCity())  
    .collect( () -> new ArrayList<>()  
        , (list, city) -> list.add(city)  
        , (left, right) -> left.addAll(right));
```

# 연락처에서 도시 목록 구하기

```
List<Contact> contacts = ContactSource.findAll();
```

```
// Stream API(collect(공급자, 누산자, 결합자)) + Lambda Expressions
cities = contacts.stream()
    .map(contact -> contact.getCity())
    .collect( () -> new ArrayList<>()
        , (list, city) -> list.add(city)
        , (left, right) -> left.addAll(right));
```

```
// Stream API(collect(공급자, 누산자, 결합자)) + Method References
cities = contacts.stream()
    .map(contact -> contact.getCity())
    .collect(ArrayList::new, ArrayList::add, ArrayList::addAll);
```

# 연락처에서 도시 목록 구하기

```
List<Contact> contacts = ContactSource.findAll();
```

```
// Stream API(collect(공급자, 누산자, 결합자)) + Lambda Expressions
cities = contacts.stream()
    .map(contact -> contact.getCity())
    .collect( () -> new ArrayList<>()
        , (list, city) -> list.add(city)
        , (left, right) -> left.addAll(right));
```

```
// Stream API(collect(공급자, 누산자, 결합자)) + Method References
cities = contacts.stream()
    .map(contact -> contact.getCity())
    .collect(ArrayList::new, ArrayList::add, ArrayList::addAll);
```

```
// Stream API(collect(Collector interface)) + Method References
cities = contacts.stream()
    .map(contact -> contact.getCity())
    .collect(Collectors.toList());
```

# 연락처에서 도시 목록 구하기

```
List<Contact> contacts = ContactSource.findAll();
```

```
// Stream API(collect(공급자, 누산자, 결합자)) + Lambda Expressions
cities = contacts.stream()
    .map(contact -> contact.getCity())
    .collect( () -> new ArrayList<>()
        , (list, city) -> list.add(city)
        , (left, right) -> left.addAll(right));
```

```
// Stream API(collect(공급자, 누산자, 결합자)) + Method References
cities = contacts.stream()
    .map(contact -> contact.getCity())
    .collect(ArrayList::new, ArrayList::add, ArrayList::addAll);
```

```
// Stream API(collect(Collector interface)) + Method References
cities = contacts.stream()
    .map(contact -> contact.getCity())
    .distinct()
    .collect(Collectors.toList());
```

중복 제거



# 스트림 연산 후 결과를 살펴보고 싶을 때

## Collectors: 공통 컬렉터용 팩토리 메소드를 제공

// Collection 타입으로 요소를 모을 때

```
contacts.stream().collect(Collectors.toList());  
contacts.stream().collect(Collectors.toSet());  
contacts.stream().collect(Collectors.toCollection(TreeSet::new));
```

// Map 타입으로 요소를 모을 때

```
contacts.stream().collect(Collectors.toMap(Contact::getName  
                                             , Contact::getBirthday));  
contacts.stream().collect(Collectors.toMap(Contact::getName  
                                             , Function.identity()));
```

// 스트림에 있는 모든 문자열을 서로 연결해서 모을 때

```
contacts.stream().map(Contact::getName).collect(Collectors.joining());  
contacts.stream().map(Object::toString).collect(Collectors.joining("|"));
```



# 주 별로 연락처를 분류하기

```
List<Contact> contacts = ContactSource.findAll();

// for 구문
Map<String, List<Contact>> contactsByState = new HashMap<>();
for(Contact contact : contacts) {
    if(!contactsByState.containsKey(contact.getState())) {
        contactsByState.put(contact.getState(), new ArrayList<>());
    }

    contactsByState.get(contact.getState()).add(contact);
}
```

# 주 별로 연락처를 분류하기

```
List<Contact> contacts = ContactSource.findAll();

// for 구문
Map<String, List<Contact>> contactsByState = new HashMap<>();
for(Contact contact : contacts) {
    if(!contactsByState.containsKey(contact.getState())) {
        contactsByState.put(contact.getState(), new ArrayList<>());
    }

    contactsByState.get(contact.getState()).add(contact);
}

// // Stream API(collect(groupingBy)) + Method References
contactsByState = contacts.stream()
    .collect(Collectors.groupingBy(Contact::getState));
```

충분히 씹고 뜯으셨나요?

# 스트림 API 정의

# 스트림 API의 3단계

```
orders.stream().map(n->n.price).sum();
```

스트림 생성

중개 연산  
(스트림 변환)

최종 연산  
(스트림 사용)

# 스트림 생성

- ✓ Collection `streams()`, `parallelStream()`
- ✓ Arrays `streams(*)`
- ✓ Stream ranges `range(...)`, `rangeClosed(...)`
- ✓ Directly from values `of(*)`
- ✓ Generators `iterate(...)`, `generate(...)`
- ✓ Resources `lines()`

# 스트림 생성

## Collection

```
List<String> collection = new ArrayList<>();  
collection.add("김지영");  
collection.add("박성철");  
collection.add("박용권");  
collection.add("정대원");
```

// 순차 스트림

```
Stream<String> stream = collection.stream();
```

// 병렬 스트림

```
Stream<String> parallelStream = collection.parallelStream();
```

# 스트림 생성

## Arrays

```
int[] numbers = new int[]{1, 2, 3, 4, 5, 6, 7, 8, 9, 0};
```

```
// 배열을 스트림으로 변환
```

```
IntStream stream = Arrays.stream(numbers);
```

```
// 순차 스트림을 병렬 스트림으로 변환
```

```
IntStream parallelStream = stream.parallel();
```



# 스트림 생성

## Directly from values

// of 메소드는 가변인자로 스트림을 생성 가능

```
Stream<String> stream = Stream.of("Using", "Stream", "API", "From", "Java8");
```

// 순차 스트림을 병렬 스트림으로 변환

```
Stream<String> parallelStream = stream.parallel();
```

# 스트림 생성

## Stream ranges

// 1부터 9까지 유한 스트림 생성

```
IntStream intStream = IntStream.range(1, 10);
```

// 1부터 10까지 유한 스트림 생성

```
LongStream longStream = LongStream.rangeClosed(1, 10);
```

# 스트림 생성

## Generators: 무한 스트림(Infinite Stream)

// 난수 스트림

```
Stream<Double> random = Stream.generate(Math::random);
```

// 0 1 2 3 ... 무한 수열

```
Stream<Integer> numbers = Stream.iterate(0, n -> n + 1);
```

# 스트림 생성

## Generators: 무한 스트림(Infinite Stream)

// 난수 스트림

```
Stream<Double> random = Stream.generate(Math::random);
```

// 0 1 2 3 ... 무한 수열

```
Stream<Integer> numbers = Stream.iterate(0, n -> n + 1);
```

// 피보나치 수열

```
Stream<int[]> fibonacci = Stream.iterate( new int[]{0,1}
                                           , n -> new int[]{n[1], n[0]+n[1]});
fibonacci.limit(10)
    .map(n -> n[0])
    .forEach(System.out::println);
```

# 스트림 생성

## Resources

// CSV 파일을 읽어 스트림을 생성

```
try(Stream<String> lines = Files.lines(Paths.get("addressBook.csv"))) {  
    long count = lines.map(line -> line.split(","))  
                        .filter(values -> values[6].contains("FL"))  
                        .count();  
} catch (IOException e) {  
    throw new RuntimeException(e);  
}
```

# 스트림 연산자

## 중개 연산자: Intermediate Operator

- ✓ 스트림을 받아서 스트림을 반환
- ✓ 무상태 연산과 내부 상태 유지 연산으로 나누어짐
- ✓ 기본적으로 지연(lazy) 연산 처리(성능 최적화)

## 최종 연산자: Terminal operator

- ✓ 스트림을 받아서 최종 결과 값을 반환
- ✓ 최종 연산 시 모든 연산 수행 (반복 작업 최소화)
- ✓ 이후 더이상 스트림을 사용할 수 없음

# 스트림 중개 연산자

## 무상태 연산

- ✓ `Stream<T> filter(Predicate<? super T> predicate)`  
: T 타입의 요소를 확인해서 기준에 통과한 요소만으로 새 스트림 생성
- ✓ `Stream<R> map(Function<? super T, ? extends R> mapper)`  
: T 타입의 요소를 1:1로 R 타입의 요소로 변환 후 스트림 생성
- ✓ `Stream<R> flatMap(Function<T, Stream<? extends R>> mapper)`  
: T 타입의 요소를 1:n으로 R 타입의 요소로 변환 후 스트림 생성, Monad bind()
- ✓ `Stream<T> skip(long n)`  
: 처음 n개의 요소를 제외한 나머지 요소로 새 스트림 생성
- ✓ `Stream<T> limit(long n)`  
: 처음 n개의 요소로 새 스트림 생성
- ✓ `Stream<T> peek(Consumer<? super T> action)`  
: T 타입의 요소를 엿본 후 스트림 생성

# 스트림 중개 연산자

## 내부 상태 유지 연산

- ✓ `Stream<T> sorted()`

: 정렬된 스트림 생성, T 타입은 Comparable 인터페이스를 구현하고 있어야 함

- ✓ `Stream<T> sorted(Comparator<? super T> comparator)`

: 주어진 Comparator 객체를 사용해 정렬된 스트림 생성

- ✓ `Stream<T> distinct()`

: 중복된 요소를 제거한 스트림 생성



# 스트림 최종 연산자

## 범용 연산 I

- ✓ `Optional<T> reduce(BinaryOperator<T> reducer)*`  
: T 타입의 요소 둘 씩 reducer로 계산해 최종적으로 하나의 값을 계산
- ✓ `Optional<T> min(Comparator<? super T> comparator)`  
: T 타입의 요소 중 최소 값을 찾아 반환
- ✓ `Optional<T> max(Comparator<? super T> comparator)`  
: T 타입의 요소 중 최대 값을 찾아 반환
- ✓ `Optional<T> findFirst()`  
: 비어 있지 않은 스트림에서 첫 번째 값을 반환(filter 연산 결합시 유용)
- ✓ `Optional<T> findAny()`  
: 첫 번째 값은 물론 어떤 요소든지 찾으면 반환(filter 연산 결합시 유용)
- ✓ `long count()`  
: 스트림에서 요소의 개수를 반환

# 스트림 최종 연산자

## 범용 연산 II

- ✓ `boolean anyMatch(Predicate<? super T> predicate)`  
: T 타입의 요소 중 조건을 만족하는 요소가 있는지 검사
- ✓ `boolean allMatch(Predicate<? super T> predicate)`  
: T 타입의 모든 요소가 조건을 만족하는지 검사
- ✓ `boolean noneMatch(Predicate<? super T> predicate)`  
: T 타입의 모든 요소가 조건을 만족하지 않는지 검사

# 스트림 최종 연산자

## 범용 연산 III

- ✓ `R collect(Collector<? super T, R> collector)`  
: T 타입의 요소를 모두 모아 하나의 자료구조나 값으로 변환
- ✓ `void forEach(Consumer<? super T> consumer)`  
: T 타입의 요소를 하나씩 처리

# 스트림 최종 연산자

## 기본 타입 스트림 전용 연산

- ✓ `int|long|double sum()`  
: 요소들의 합계를 반환
- ✓ `OptionalDouble average()`  
: 요소들의 평균을 반환
- ✓ `OptionalInt|Long|Double min()`  
: 최소값을 가진 요소를 반환
- ✓ `OptionalInt|Long|Double max()`  
: 최대값을 가진 요소를 반환
- ✓ `Int|Long|DoubleSummaryStatistics summaryStatistics()`  
: 요소들의 합계, 평균, 개수, 최소값, 최대값을 반환

충분히 즐기셨나요?

# 요약

- ✓ 스트림 라이브러리

  - : 컬렉션 프레임워크보다 한 단계 더 높은 추상화된 API

  - : 내부 반복을 통해 제어 흐름 추상화하고 최적화와 알고리즘을 분리

- ✓ 스트림 연산

  - : 중개 연산 / 파이프와 필터, 맵(Map)

  - : 최종 연산 / 결과 생산(Reduce)

- ✓ 지연 연산을 통한 성능 최적화

- ✓ 병렬 처리 추상화를 통한 손쉬운 사용

- ✓ 컬렉션 외 다양한 연속적 데이터에 사용

궁금하신분!?



<http://www.ksug.org/>

<http://groups.google.com/group/ksug>

<https://www.facebook.com/groups/springkorea/>



# 참고자료

## ✓ 가장 빨리 만나는 자바8

: <http://www.gilbut.co.kr/book/bookView.aspx?bookcode=BN000898&page=1&TF=T>

## ✓ 자바 8 람다의 이해와 의미

: <http://www.slideshare.net/gyumee/java-8-lambda-35352385>

## ✓ Brand new Data Processing – Stream API

: [http://www.slideshare.net/bitter\\_fox/brand-newdataprocessing](http://www.slideshare.net/bitter_fox/brand-newdataprocessing)

## ✓ The Stream API

: <http://blog.hartveld.com/2013/03/jdk-8-33-stream-api.html>

## ✓ java.util.stream (Java Platform SE 8)

: <http://docs.oracle.com/javase/8/docs/api/java/util/stream/package-summary.html>

## ✓ Lambdas and Streams in Java 8 Libraries

: <http://www.drdobbs.com/jvm/lambdas-and-streams-in-java-8-libraries/240166818>