

Java 8, 9

김 순곤

soongon@hucloud.co.kr

목 차

- 00. Java8 과 Modern Java
- 01. 함수형 프로그래밍 개요
- 02. 인터페이스와 람다 표현식
- 03. 스트림의 활용
- 04. 스트림으로 데이터 리द싱
- 05. JodaTime을 개선한 Date API
- 06. 병렬 데이터 처리와 성능
- 07. Java 9 New Features



1-00

Java8 & Modern Java

Modern Java 로의 여정

- 2014년 3월 18일 : Java 8 릴리즈
 - 2년 7개월 만에 새로운 버전
 - 자바 5 이후 대대적인 변화
 - 함수형 프로그래밍 기법 도입

모던 자바 주요 특징

- 람다 지원
- 함수형 프로그래밍 지원
- Stream API 지원 : map, flatMap 등
- 스트링, 컬렉션, 숫자, 수학관련 API 변화
- Optional 지원 : null 세이프 기능
- 동시성 라이브러리 향상
- 인터페이스 대대적 변화
 - Static 메서드 지원
 - Default 메서드 지원
 - 함수형 인터페이스에서 타입 추론 기능 : 람다 지원을 위한 기능
- 새로운 시간과 날짜 관련 API
- JVM 성능 향상 : JRocket 통합
-

Interface 향상

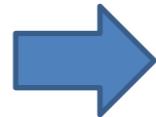
- **default method**
 - 자바8 이전까지 Interface의 abstract 메서드는 반드시 클래스에서 구현해야 하고, Interface에 새로운 메서드가 추가 되면, 구현 클래스를 반드시 수정해야 하므로 바이너리 호환성에 이슈가 생길 수 있었음
 - 자바8 이후에는 Interface에 새롭게 확장할 메서드를 default 메서드로 지정할 경우에도 기존 클래스에 도 영향이 없음.
 - 즉, default 메서드는 구현하는 모든 클래스들이 동일한 기능을 사용할 수 있도록 하는 동시에 , Interface에 변경이 생기더라도 바이러리 호환성을 유지할 수 있는 방법을 제공함.

- **static method**
 - 인터페이스에서 static method 사용 가능
 - Factory 메소드를 인터페이스의 static method로 구현

Lambda Expression

- Lambda는 Java8 기능 중 큰 변화

```
new Thread(new Runnable() {  
    public void run() {  
        System.out.println("Thread run!!");  
    }  
}).run();
```



```
Runnable r = () -> { System.out.println("Thread run!!"); }
```

- Functional Interface

- 하나의 abstract 메서드 만을 가진 인터페이스를 Functional Interface 라고 부른다.
- Lambda expression을 지원하는 java.util.function 패키지가 추가되었다.

Stream API

- Java8은 Stream과 Lambda를 활용하여 이전보다 훨씬 세련되고 향상된 방법으로 Collection을 처리한다.
 - Stream은 순차(sequential), 병렬(parallel) 두 종류가 존재한다.
 - 순차 Stream은 싱글 쓰레드로 순차적으로 처리되며, 병렬 Stream은 Fork/Join 프레임워크로 구현된 메서드에 의해 병렬로 처리된다.

```
int totalCountOfOrder = orderList.stream()
    .filter(b -> b.getProduct() == "iPhone")
    .map(b -> b.getAmount())
    .sum();
```

```
OptionalDouble averageAge = pl
    .parallelStream()
    .filter(search.getCriteria("allPilots"))
    .mapToDouble(p -> p.getAge())
    .average();
```

■ Stream의 장단점

- 장점은 Laziness이다. 즉 Collection의 iteration 처리를 자바에게 맡겨 둠으로써 JVM이 최적화할 수 있는 기회를 제공한다.
- 단점은 재사용이 불가능하다. 한번 사용된 Stream은 재사용이 불가능하며, 필요에 따라 새롭게 만들어야 한다.

Time And Date API

- 기존에 알려진 문제점
 - 1. 년은 1900년 부터 시작한다. 1900년도 이전에 대한 처리는 workaround가 필요
 - 2. 월이 1부터가 아닌 0부터 시작한다. 그러므로 12월은 실제로 숫자 11이다.
 - 3. mutable 하므로 thread safe 하지 않다.
 - 4. performance 문제
- 새로운 Time And Date API
 - java.time 패키지의 LocalDate, LocalTime, LocalDateTime, ZonedDateTime 클래스
 - LocalDate는 년-월-일, LocalTime은 시-분-초-나노초, LocalDateTime은 년-월-일-시-분-초를 내부적으로 저장함

```
LocalDate currentDate = LocalDate.now();
boolean isLeapYear = currentDate.isLeapYear();
LocalDate date = LocalDate.of(2014, Month.FEBRUARY, 12);
LocalTime currentTime = LocalTime.now();
LocalDateTime currentTime = LocalDateTime.now();
currentTime = LocalDateTime.of(2014, 2, 21, 13, 40);
date = currentTime.plusMonths(1).minusDays(3);
```

Project Nashorn(나즈흔)

■ 자바스크립트 엔진

- Javascript를 JVM에서 동작 하려는 시도는 1997년 Netscape가 100% Java 기반의 브라우저를 만들려고 계획할 때 부터 시작되었다. 그 프로젝트 이름은 ‘Rhino’(리노)이다.
- ‘Nashorn’은 Rhino의 독일어라고 하며, Rhino는 ECMAScript 5.1 표준을 완벽히 구현 하고 있다.
- JDK7에 소개되었던 invokedynamic이 있다. Lambda도 내부적으로 invokedynamic을 활용한다.
- invokedynamic은 JVM의 플랫폼화를 하기 위한 필수적인 개념이다.
- Java는 컴파일 타임에 타입을 체크 하지만, Javascript는 ‘duck-type’이라는 타입 시스템을 채용함으로써 컴파일 타임의 type을 강제 하지 않는다.
- Invokedynamic은 duck-typing을 JVM 레벨에서 기본적으로 지원하면서 Java 외에 다른 언어들이 JVM이라는 플랫폼 위에서 최적화된 방식으로 실행될 수 있는 토대를 제공한다.
- Java8에는 jjs 라고 하는 커맨드라인 툴을 이용해서 Nashorn 엔진을 호출 할 수 있다.

```
$ $JAVA_HOME/bin/jjs  
jjs> print('Hello World');
```

Java9 REPL (JShell)

- Java shell

```
G:\>jshell
| Welcome to JShell -- Version 9-ea
| For an introduction type: /help intro

jshell> int a = 10
a ==> 10

jshell> System.out.println("a value = " + a )
a value = 10
```

Java 9 Module System

- 자바9의 가장 큰 변화 중 하나
- Jigsaw (직소) 프로젝트
 - Modular JDK
 - Modular Java Source Code
 - Modular Run-time Images
 - Encapsulate Java Internal APIs
 - Java Platform Module System

```
module com.foo.bar { }
```

개선 사항들

- Immutable 리스트/맵/셋 을 만들 수 있는 팩토리 메서드

```
List immutableList = List.of("one", "two", "three");
```

- 인터페이스에서 private 메서드 사용
- Process API 개선
 - Java.lang.ProcessHandle
 - java.lang.ProcessHandle.Info
- Try with Resources 개선

개선 사항들 - 계속

- CompletableFuture API 개선

```
Executor exe = CompletableFuture.delayedExecutor(50L,  
TimeUnit.SECONDS);
```

- 리액티브 스트림

- **java.util.concurrent.Flow**
- **java.util.concurrent.Flow.Publisher**
- **java.util.concurrent.Flow.Subscriber**
- **java.util.concurrent.Flow.Processor**

개선 사항들 - 계속

- 스트림 API 개선
 - takeWhile 과 dropWhile 메서드 제공

```
jshell> Stream.of(1,2,3,4,5,6,7,8,9,10).takeWhile(i -> i < 5 )  
      .forEach(System.out::println);  
1  
2  
3  
4
```

- Optional 클래스 개선
 - 반환되는 데이터가 있으면 스트림으로 없으면 빈 스트림으로 반환

```
Stream<Optional> emp = getEmployee(id)  
Stream empStream = emp.flatMap(Optional::stream)
```

1-01

함수형 프로그래밍이란?

함수형 사고

잠시 당신이 나무꾼이라고 가정해보자.

“당신은 숲에서 가장 좋은 도끼를 가지고 있고,

그래서 가장 일 잘하는 나무꾼이다.

그런데 어느 날 누가 나타나서 나무를 자르는

새로운 패러다임인 전기톱을 알리고 다닌다.

이 사람이 무척 설득력이 있어서 당신은 사용하는

방법도 모르면서 전기톱을 사게 된다.

당신은 여태껏 했던 방식대로 시동을 걸지도 않고

전기톱으로 나무를 마구 두들겨댄다.

곧 당신은 이 새로운 전기톱은 일시적인 유행일 뿐이라고

단정하고 다시 도끼를 쓰기 시작한다.

그때 누군가 나타나서 전기톱의 시동 거는 법을 가르쳐 준다.”

— “함수형 사고”에서

O'REILLY®

객체지향 개발자에서 함수형 개발자로 거듭나기



한빛미디어

날 포드
김재한

함수형 프로그래밍이란?

- **함수형 프로그래밍(functional programming, FP)이란?**
 - 계산을 수학적 함수의 평가로 취급하고 상태와 가변 데이터를 멀리하는 프로그래밍 패러다임이다.
 - 함수형 프로그래밍은 프로그램을 오직 순수 함수(pure function)들로만 작성되어 진다.
 - 즉, Side Effect(부수효과)가 없는 함수들로만 구축한다는 의미이다.
 - 부수효과들을 제거할 경우에 프로그램의 동작을 이해하고 예측하는 것이 훨씬 쉬워 진다.
- **부수 효과(Side Effect)란?**
 - 변수를 수정하거나, 객체의 필드를 설정한다.
 - 예외(exception)를 던지거나 오류를 내면서 실행을 중단한다.
 - 콘솔에 출력하거나 사용자의 입력을 읽어 들인다.
 - 파일에 기록하거나 파일에서 읽어 들인다.

함수형 프로그래밍 언어들

- 1930년대 알론조 처치(Alonzo Church)와 하스켈 커리(Haskell Curry)가 개발함
 - 알론조 처치는 함수를 정의하고 실행하는 방식을 설명하는 람다 계산법(Lambda Calculus)을 개발했다.
 - 하스켈 커리는 계산에 대한 이론적 근거를 제공하는 조합논리(Combinatory Logic)을 개발했다.
 - 조합 논리는 기본적으로 함수에 해당하는 조합기(combinator)가 계산과정을 나타내기 위해 어떻게 조합을 수행하는지 검사함
- 함수형 프로그래밍의 아이디어를 처음으로 활용한 언어는 리스프(Lisp)이다.
 - 리스프는 1950년대 후반에 개발 되었고, 포트란(Fortran) 다음으로 두번째로 오래된 프로그래밍 언어이다.
- 함수형 언어의 가장 순수한 의미에 근접한 언어는 하스켈(Haskell)이다.
 - 하스켈은 1990년대 초반에 개발 되었다.
- 객체지향 프로그래밍과 함수형 프로그래밍 요소가 결합된 스칼라(Scala)가 있다.
 - 스칼라는 2004년에 개발 되었고, 자바 런타임 환경(JRE)과의 호환성이 좋으며, .NET을 위한 지원도 제공 하고 있다.
- 자바플랫폼과 공존하고 현대적인 리스프에 해당되는 언어는 클로저(Clojure)이다.
 - 클로저는 2007년에 개발되었고, 자바가상머신(JVM)과 공통언어런타임(CLR), 자바스크립트 엔진상에서 동작함
- 얼랭(Erlang)이 보유하는 분산처리, 장애 내구성 등을 공유하는 엘릭서(Elixir)가 있다.
 - 엘릭서는 2012년에 개발되었고, 얼랭(Erlang) 가상머신에서 동작하는 함수형, 동시성 프로그래밍 언어이다.

함수형 프로그래밍의 기본 원리들

- **변경 불가능한 값을 이용**
 - 함수의 계산을 수행하는 동안 변수에 할당된 값들이 절대로 변하지 않는다.
 - 변수에 새로운 값을 설정할 수 있을 뿐이며 일단 값이 설정되면 그 값을 바꿀 수 없다.
- **함수가 1등 시민**
 - 함수가 1등 시민(First-class Citizen)이라는 말은 함수를 변수나 자료 구조 안에 담을 수 있고, 함수를 인자로 전달할 수 있으며, 반환 값으로 사용할 수 있다는 의미이다.
- **람다와 클로저**
 - 람다는 익명함수를 말하며, 인수의 리스트와 함수의 본문만을 가지는 함수이다.
 - 클로저란 자신이 생성될 때의 scope에서 알 수 있었던 변수를 기억하는 함수이다.
- **고계함수(Higher-order Function)**
 - 고계함수는 다른 함수를 인수로 받아 들이거나 함수를 리턴 하는 함수를 가리킨다.
 - 함수가 정수와 동등한 값으로 다뤄지기 때문에 함수를 인자로 넘기거나 결과 값을 함수로 반환 받을 수 있다.

함수형 프로그래밍의 컨셉

- 1. 변경 가능한 상태를 불변상태(Immutable)로 만들어 SideEffect를 없애자
 - 함수 안에서 상태를 관리하고 상태에 따라서 결과값이 달라지면 않 된다는 뜻임.
 - 상태를 사용하지 않음으로 SideEffect를 차단할 수 있다.
- 2. 모든 것은 객체이다.
 - 클래스 외에 함수 또한 객체이기 때문에 함수를 값으로 할당할 수 있고, 파라미터로 전달 및 결과 값으로 반환이 가능하다. 이 3가지 조건을 만족하는 객체를 1급 객체(First-class citizen)라고 한다.
- 3. 코드를 간결하게, 가독성을 높여 로직에 집중시키자.
 - Lambda 및 Stream 과 같은 API를 통해 보일러 플레이트를 제거하고, 내부에 직접적인 함수 호출을 통해 가독성을 높일 수 있다.
- 4. 동시성 작업을 보다 쉽고 안전하게 구현하자.

Java8에서 함수형 프로그래밍을 어떻게 지원하는가?

■ 1.Functional Interface

- 함수형 인터페이스란 하나의 abstract 메서드를 가지는 인터페이스이다. 함수형 인터페이스 지정을 위하여 @FunctionalInterface 어노테이션이 도입되었다.

```
@FunctionalInterface  
public interface Runnable {  
    public abstract void run();  
}
```

■ 2.Lambda

- 함수형 인터페이스는 람다를 이용하여 인스턴스화 될 수 있다. 화살표의 왼편은 입력이고 오른편은 코드이다. 입력타입은 추론 가능하기 때문에 선택이다.

```
(int x, int y) -> { return x + y; }  
(int x, int y) -> x + y  
(x, y) -> x + y  
x -> x * x  
() -> x  
x -> { System.out.println(x); }
```

```
Runnable r = () -> { System.out.println("Running!"); }
```

Java8에서 함수형 프로그래밍을 어떻게 지원하는가?

■ 3.Method Reference

- 메서드 참조는 이름을 가진 메서드 들에 대한 컴팩트 한 람다 표현식이다.

```
String::valueOf  
x -> String.valueOf(x)  
Object::toString  
x -> x.toString()  
x::toString  
() -> x.toString()  
ArrayList::new  
() -> new ArrayList<>()
```

■ 4.Closure

- 람다는 람다 바디의 외부에 정의된 non-static 변수 혹은 객체에 접근 가능하다. 이를 “capturing”이라고 한다.
- 람다 표현식은 오직 로컬 변수와 인자로 던져진 감싸진 블록에서만 접근 가능하다.

```
int x = 5;  
return y -> x + y;
```

Java8에서 함수형 프로그래밍을 어떻게 지원하는가?

■ 4.Stream

- java.util.stream 패키지는 값들의 스트림 위에서 함수형-스타일의 동작을 지원하는 클래스들을 제공한다.

```
int sumOfWeights = blocks.stream()
    .filter(b -> b.getColor() == Red).mapToInt(b -
> b.getWeight()).sum();
```

- 위의 예제는 스트림 위에서 filter-map-reduce를 수행한다.
- 스트림은 먼저 어떤 소스로 부터 스트림을 얻고, 다음으로 하나 이상의 중간적(intermediate) 작업을 수행한 후, 마지막으로 하나의 최종 종료(final terminal) 작업을 수행한다.
- 중간작업은 filter, map, flatMap, peek, distinct, sorted, limit, substream 이다.
- 최종 종료 작업은 forEach, toArray, reduce, collect, min, max, count, anyMatch, allMatch, noneMatch, findFirst, findAny 를 포함한다.

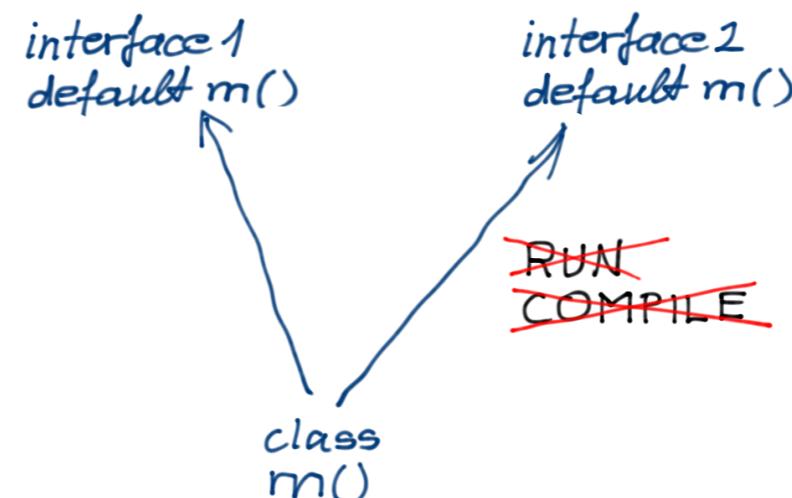
1-02

인터페이스와 람다 표현식

인터페이스의 변화

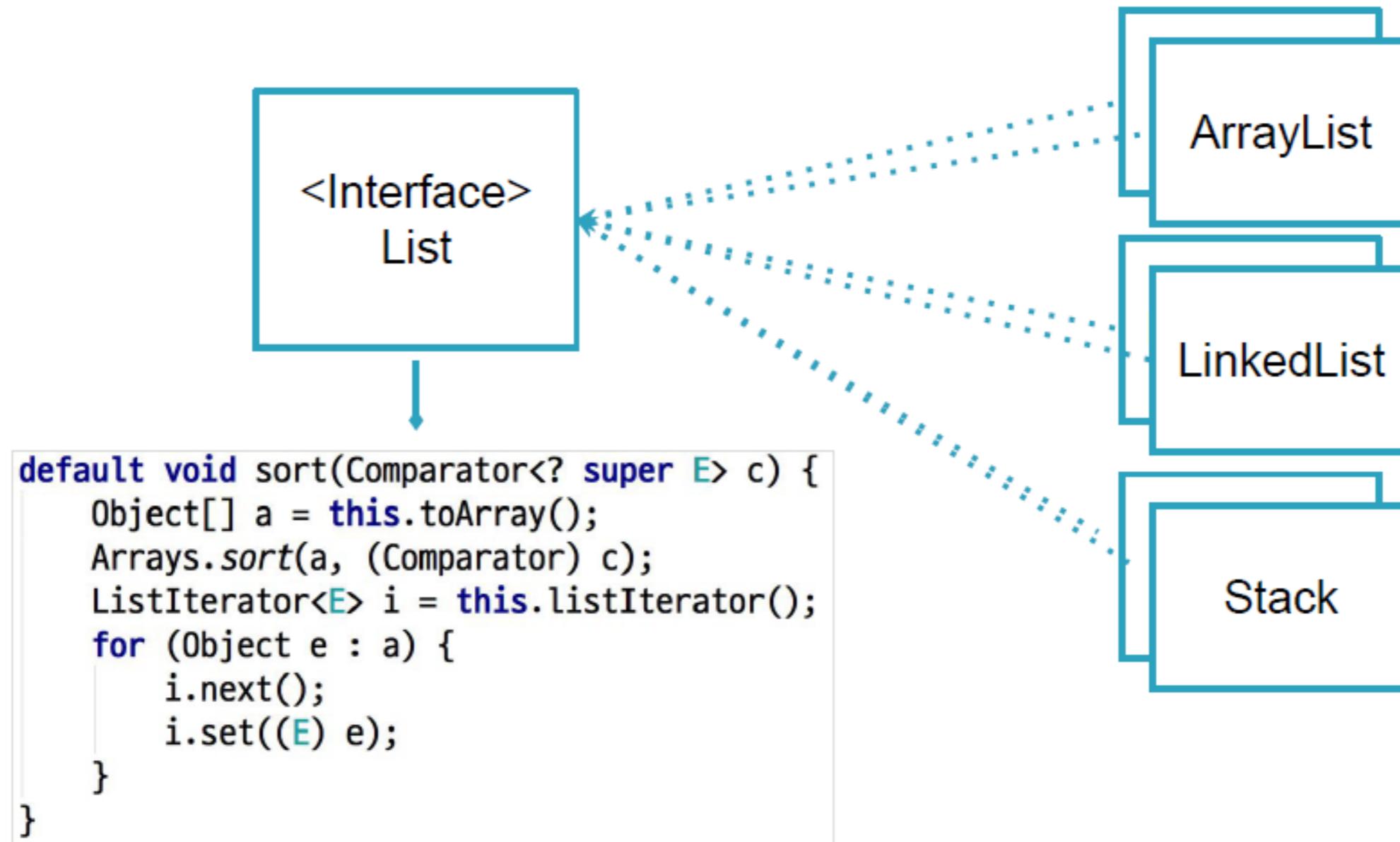
인터페이스의 변화 – default method

- 디폴트 메서드 (default method)
 - 인터페이스 메서드가 구현을 가질 수 있음
 - 디폴트 메소드에는 반드시 default 제어자를 붙여 줘야 함
 - java8 에서 추가 되었으며, 인터페이스 변경을 자유롭게 하기 위해 디폴트 메소드를 적극 사용하고 있음
- 디폴트 메소드 충돌 주의



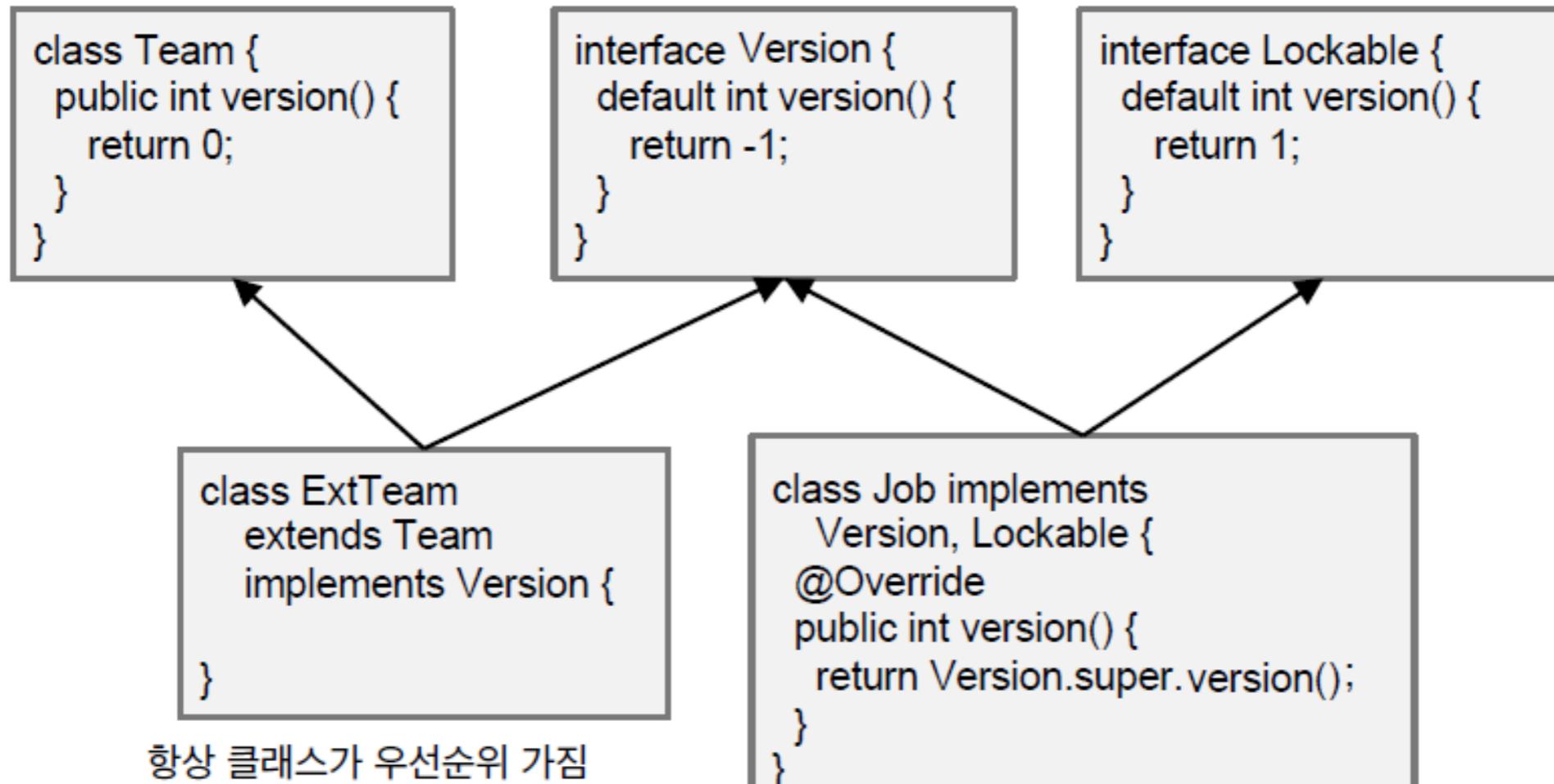
인터페이스의 변화 – default method

- 디폴트 메서드 (default method)
 - default 키워드를 사용하면 인터페이스에 새로운 메서드를 추가 하더라도 기존의 구현체 구조를 변경하지 않고 새로운 기능을 추가 할 수 있다.



인터페이스의 변화 – default method

■ 디폴트 메서드 (default method)의 우선 순위

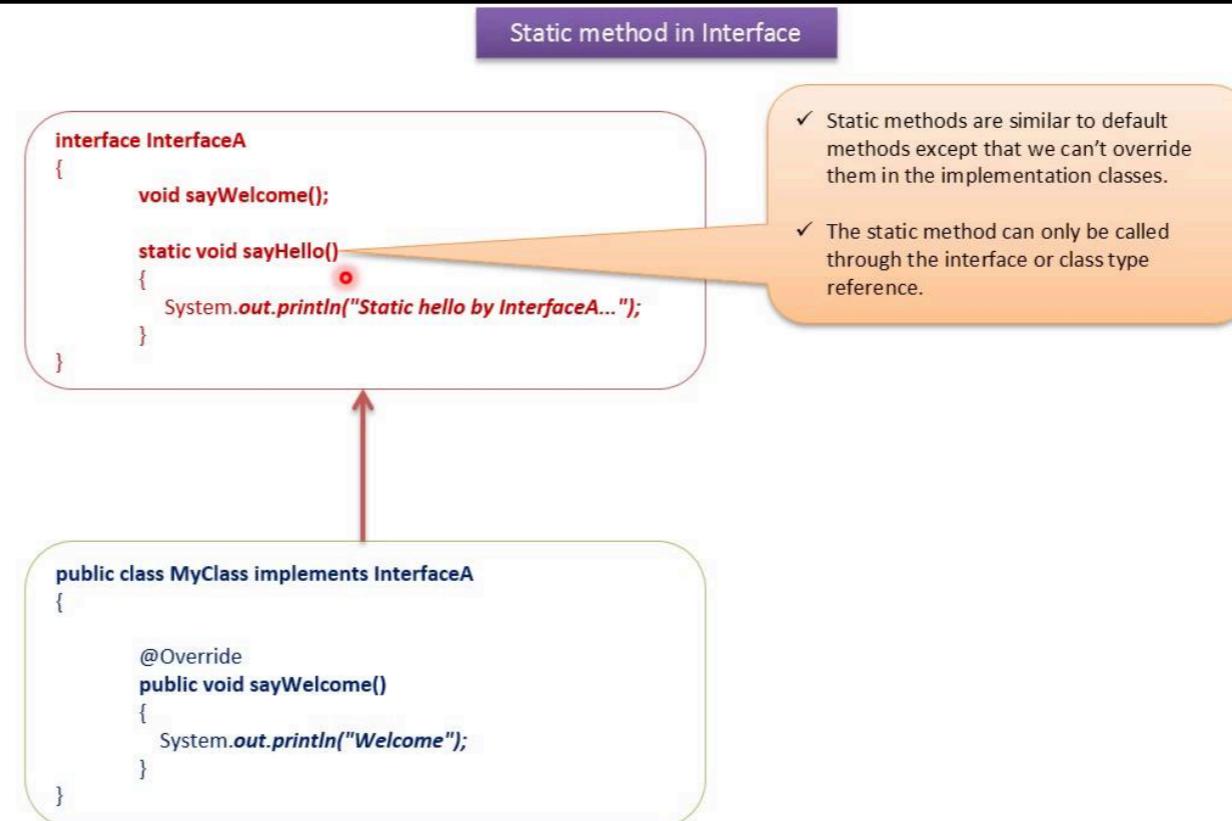


항상 클래스가 우선순위 가짐
- ExtTeam 객체의 getVersion()은
Team 클래스의 version() 사용

상속받은 인터페이스들의 같은 시그너처
를 갖는 메서드 중 한 개라도 디폴트 메서드가 있으면,
하위 타입에서 재정의해 주어야 함

인터페이스의 변화 – static method

- 인터페이스에서 static method 사용 가능
- Factory 메소드를 인터페이스의 static method로 구현
- 예전에는 static 메소드를 포함한 Companion class 를 별도로 두었음
 - Collection/Collections
 - Path/Paths



Lambda

람다

람다(Lambda) 란 무엇인가?

람다 대수

위키백과, 우리 모두의 백과사전.
(람다대수에서 넘어옴)

람다 대수(λ -, lambda-)는 이론 컴퓨터과학 및 수리논리학에서 함수 정의, 함수 적용, 귀납적 함수를 추상화한 형식 체계이다. 1930년대 알론조 처치가 수학기초론을 연구하는 과정에서 람다 대수의 형식을 제안하였다. 최초의 람다 대수 체계는 논리적인 오류가 있음이 증명되었으나, 처치가 1936년에 그 속에서 계산과 관련된 부분만 따로 빼내어 후에 타입 없는 람다 대수 (*untyped lambda calculus*)라고 불리게 된 체계를 발표하였다. 또한 1940년에는 더 약한 형태이지만 논리적 모순이 없는 단순 타입 람다 대수 (*simply typed lambda calculus*)를 도입하였다.

람다 대수는 계산 이론, 언어학 등에 중요한 역할을 하며, 특히 프로그래밍 언어 이론의 발전에 크게 기여했다. 리스프와 같은 함수형 프로그래밍 언어는 람다 대수로부터 직접적인 영향을 받아 탄생했으며, 단순 타입 람다 대수는 현대 프로그래밍 언어의 타입 이론의 기초가 되었다.

<https://ko.wikipedia.org/wiki/람다대수>

람다 문법

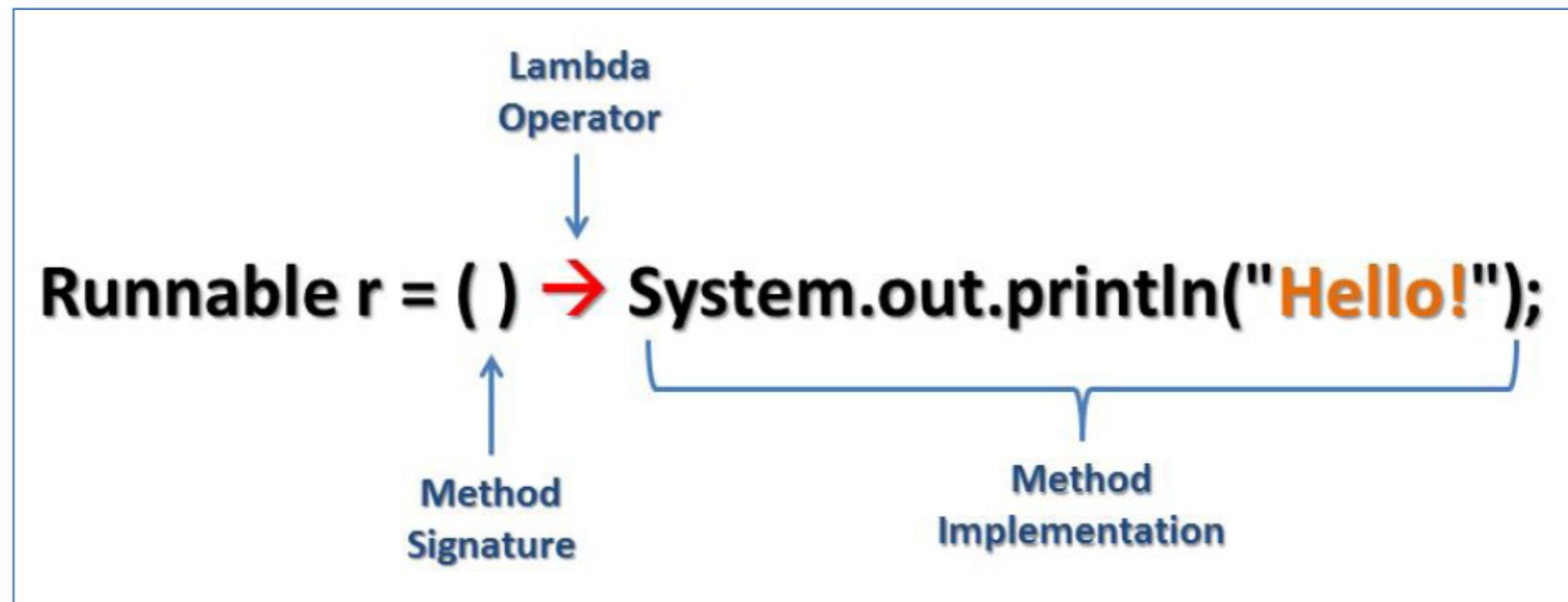
- 수학에서 람다식 문법

$$\lambda x. x+1$$


- 1950년대 LISP

```
(lambda (arg) (+ arg 1))
```

- 자바8



람다 표현식(Lambda expression)

■ 익명 함수 생성 문법

- 람다 표현식은 논리학자인 Alonzo Church가 1930년대에 제안한 람다 대수에서 유래함.
- 람다 표현식은 함수를 간결하게 표현함.
- 프로그래밍 언어의 개념으로는 단순한 익명 함수 생성 문법이라 이해할 만함.
- 이미 많은 언어에서 람다 표현식을 지원 : Ruby, C#, Python, Scala, Javascript
- 람다 표현식이 들어간 Java 8을 ‘모던 Java’ 그 이전을 ‘클래식 Java’
- ‘모던’은 특정 시점에서는 과거와 대비 되는 큰 변화를 설명하기에 유용한 표현임.

■ 람다의 특성

- 익명 : 이름이 없다.
- 함수 : 메서드처럼 특정 클래스에 종속되지 않는다. 파라미터 리스트, 바디, 리턴값을 포함 한다.
- 전달 : 람다 표현식을 메서드의 인자로 전달하거나 변수 값으로 저장할 수 있다.
- 간결성 : 익명 클래스를처럼 부가 코드를 구현할 필요가 없다.

리스트 정렬 사례를 통한 람다식

```
List<String> list = Arrays.asList("a", "b", "c", "d");  
list.sort(/* Comparator 타입을 파라미터로 받음 */);
```

Comparator 인터페이스 – Java 7,6,5

- Comparator 인터페이스

- 객체의 정렬을 위해서 사용

```
public interface Comparator<T> {  
    public int compare(T o1, T o2);  
}
```



```
Comparator<Integer> comparator = new  
Comparator<Integer>() {  
    @Override  
    public int compare(Integer o1, Integer o2) {  
        return o2 - o1;  
    }  
};
```

Comparator 인터페이스 – Java 8

- Comparator 인터페이스

- 객체의 정렬을 위해서 사용

```
public interface Comparator<T> {  
    public int compare(T o1, T o2);  
}
```

compare( , )

```
Comparator<Integer> comparator = (o1, o2) -> o2 - o1;
```

Java 8의 람다식

- 함수형(Functional) 인터페이스의 임의 객체를 람다식으로 표현
- 함수형 인터페이스 : 추상 메서드가 하나인 인터페이스

```
public interface Comparator<T> {  
    public int compare(T o1, T o2);  
}
```



```
Comparator<Long> comparator =  
    (Long o1, Long o2) -> o2 - o1 < 0 ? -1:1;
```

람다식의 구성

- 인자목록과 구문을 ->(화살표)로 연결한다.

(인자목록)

->

{구문}

람다식

=

익명 메서드

```
public void sayHello(PrintStream out) {  
    out.println("Hello World");  
}
```



```
(PrintStream out) -> { out.println("Hello World"); }
```

쓰레드 생성 – Java 7,6,5 익명클래스

■ Runnable 인터페이스

- 메서드를 하나만 가지고 있는 함수형 인터페이스

```
public interface Runnable {  
    public abstract void run();  
}
```

```
public class AsyncHelloWorld {  
    public static void main(String[] args) {  
        new Thread(new Runnable() {  
            @Override  
            public void run() {  
                System.out.println("Hello World");  
            }  
        }).start();  
    } //main  
} //class
```

쓰레드 생성 – Java 8 람다식

- Runnable 인터페이스

- 메서드를 하나만 가지고 있는 함수형 인터페이스

```
public interface Runnable {  
    public abstract void run();  
}
```

```
public class AsyncHelloWorld {  
    public static void main(String[] args) {  
        new Thread(() -> {  
            System.out.println("Hello World");  
        }).start();  
    }  
}
```

람다 표현식의 예

- **파라미터 타입지정**

```
Comparator<Long> longComparator =  
    (Long first, Long second) -> Long.compare(first, second);
```

- **파라미터 타입지정, 문맥에서 유추**

```
Comparator<Long> longComparator =  
    (first, second) -> Long.compare(first, second);
```

- **파라미터가 한 개인 경우, 파라미터 목록에 괄호 생략**

```
Predicate<String> predicate = t -> t.length() > 10;
```

- **파라미터가 없는 경우**

```
Callable<String> callable = () -> "noparam";
```

- **결과 값이 없는 경우**

```
Runnable runnable = () -> System.out.println("no return")
```

- **코드 블록을 사용할 경우, return을 이용해서 결과 값을 리턴**

```
Operator<Integer> plusOp = (op1, op2) -> {  
    int result = op1 + op2;  
    return result * result;  
}
```

```
public interface Operator<T> {  
    public T operate(T op1, T op2);  
}  
  
public interface Callable<V> {  
    V call() throws Exception;  
}  
  
public interface Runnable {  
    public void run();  
}  
  
public interface Predicate<T> {  
    boolean test(T t);  
}
```

람다 표현식의 예

```
() -> {}                      // No parameters; result is void
() -> 42                        // No parameters, expression body
() -> null                       // No parameters, expression body
() -> { return 42; }             // No parameters, block body with return
() -> { System.gc(); }           // No parameters, void block body
() -> {
    if (true)
        return 12;
    else {
        int result = 15;
        for (int i = 1; i < 10; i++)
            result *= i;
        return result;
    }
}                                // Complex block body with returns
```

람다 표현식의 예

```
(int x) -> x+1          // Single declared-type parameter
(int x) -> { return x+1; } // Single declared-type parameter
(x) -> x+1              // Single inferred-type parameter
x -> x+1                // Parens optional for single
                           inferred-type case
(String s) -> s.length() // Single declared-type parameter
(Thread t) -> { t.start(); } // Single declared-type parameter
s -> s.length()           // Single inferred-type parameter
t -> { t.start(); }       // Single inferred-type parameter
(int x, int y) -> x+y    // Multiple declared-type
                           parameters
(x,y) -> x+y             // Multiple inferred-type
                           parameters
(final int x) -> x+1    // Modified declared-type parameter
(x, final y) -> x+y    // Illegal: can't modify inferred-type parameters
(x, int y) -> x+y      // Illegal: can't mix inferred and declared types
```

@FunctionalInterface

- 함수형 인터페이스 강제
 - @FunctionallInterface 를 클래스 상단에 붙여 명시적으로 함수형 인터페이스라는 것을 표기할 수 있다.
 - @FunctionallInterface가 붙은 인터페이스가 추상 메서드를 2개 이상 가지면 컴파일 에러 발생!!

@FunctionalInterface

```
public interface Operator<T> {  
    public T operator(T op1, T op2);  
}
```

함수형 인터페이스

- 람다와 익명 클래스는 다르다.
 - 익명 클래스는 컴파일 시 서브 클래스도 별도의 파일로 컴파일 되고, 람다는 기존 클래스에 포함된다.
 - 람다는 바이트 코드로 변환 시 Java7에서 추가된 InvokeDynamic 명령어를 이용하여 런타임에 코드가 생성된다.
- 함수형 인터페이스를 인자로 받는 메서드에서 람다식을 사용할 수 있다.
- 위와 같은 제약사항으로 인해 람다식의 타입 추론이 가능해졌다.
 - 함수형 인터페이스가 가지고 있는 추상 메서드가 오직 하나 밖에 없으므로, 이를 이용하여 타입을 추론한다.
 - 람다식의 Parameter 추론 : 추상 메서드의 파라미터 정보
 - 람다식의 Return 타입 추론 : 추상 메서드의 리턴 타입 정보
- Java8에서 `java.util.function` 패키지로 다양한 함수형 인터페이스를 제공한다.

미리 정의된 함수형 인터페이스

- **java.util.function 패키지**
 - 다양한 상황에 사용할 수 있는 함수형 인터페이스가 정의됨

| 함수형 인터페이스 | Descriptor | Method명 |
|-------------------|------------------|------------|
| Predicate<T> | T -> boolean | test() |
| BiPredicate<T,U> | (T,U) -> boolean | test() |
| Consumer<T> | T -> void | accept() |
| BiConsumer<T,U> | (T,U) -> void | accept() |
| Supplier<T> | () -> T | get() |
| Function<T,R> | T -> R | apply() |
| BiFunction<T,U,R> | (T,U) -> R | apply() |
| UnaryOperator<T> | T -> T | identity() |
| BinaryOperator<T> | (T,T) -> T | apply() |

동작 파라미터로 코드 개선하기

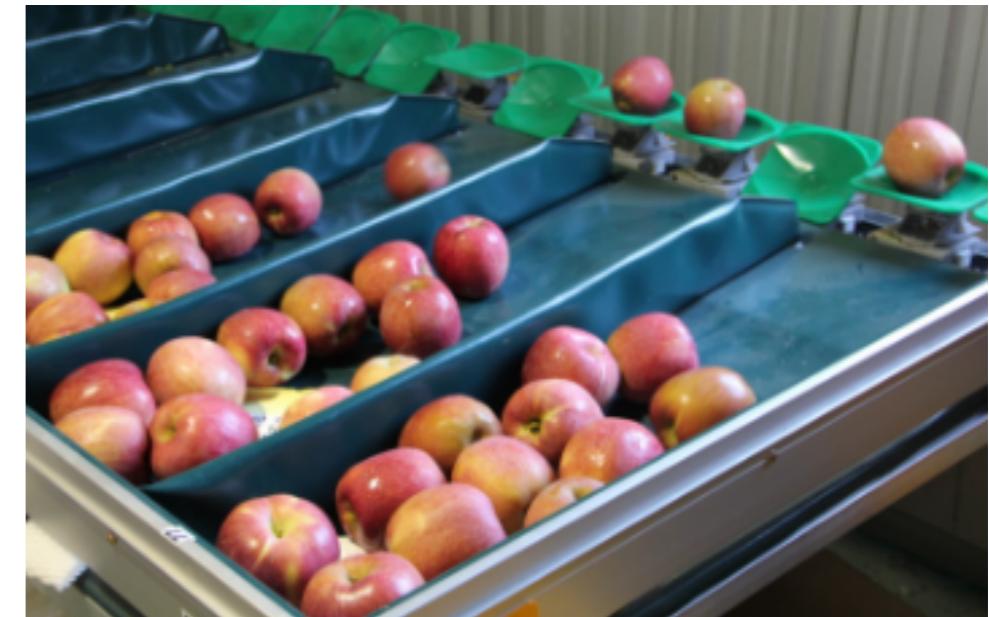
과일 재고 관리 어플리케이션을 개발하려고 할 때
사과의 종류별로 필터링 하여 보여줄 수 있는 기능을 요구함

■ 요구사항

무게가 150g 이상인 사과만 필터링

빨간 사과와 초록 사과를 분리하기

빨간색이며 무게가 150g 이상인 사과를 필터링



동작 파라미터로 코드 개선하기

■ 초록색 사과만 필터링 하기

사과리스트를 파라미터로 받음

```
public List<Apple> filterGreenApples(List<Apple> inventory) {  
    List<Apple> result = new ArrayList<>();  
    for (Apple apple : inventory) {  
        if( "green".equals(apple.getColor()) ){ 조건에 맞는 사과 필터링  
            result.add(apple);  
        }  
    }  
    return result; 필터링된 사과만 반환  
}
```

동작 파라미터로 코드 개선하기

■ 색상을 파라미터로 받아서 필터링 하기

```
public List<Apple> filterApplesByColor(List<Apple> inventory,  
                                         String color) {  
    List<Apple> result = new ArrayList<>();  
    for (Apple apple : inventory) {  
        if( color.equals(apple.getColor()) ){  
            result.add(apple);  
        }  
    }  
    return result;  필터링된 사과만 반환  
}
```

색깔을 파라미터로 받음

파라미터로 받은 색깔과 비교

동작 파라미터로 코드 개선하기

■ 무게를 파라미터로 받아서 필터링 하기

색상을 필터링 하는 경우와 유사한 코드가 나타난다.

```
public List<Apple> filterApplesByWeight(List<Apple> inventory,  
                                         int weight) {  
    List<Apple> result = new ArrayList<>();  
    for (Apple apple : inventory) {  
        if (apple.getWeight() > weight) {    무게를 기준으로 필터링  
            result.add(apple);  
        }  
    }  
    return result;  필터링된 사과만 반환  
}
```

무게를 파라미터로 받음

무게를 기준으로 필터링

동작 파라미터로 코드 개선하기

■ 조건을 파라미터로 직접 넣을 수 있도록 만들기

사용자가 어떤 행동을 할지를 직접 정의할 수 있다. 여기서는 필터 조건이 정의된다.

```
public List<Apple> filterApples(List<Apple> inventory, ApplePredicate p){  
    List<Apple> result = new ArrayList<>();  
    for (Apple apple : inventory) {  
        if (p.test(apple)) {  
            result.add(apple);  
        }  
    }  
    return result;  
}
```

→ Predicate 내부 test 메서드로 조건문 실행

```
public interface ApplePredicate {  
    boolean test(Apple apple);  
}
```

동작 파라미터로 코드 개선하기

- 조건을 파라미터로 직접 넣을 수 있도록 만들기

ApplePredicate를 구현

```
public class AppleColorPredicate implements ApplePredicate{  
    @Override  
    public boolean test(Apple apple) {  
        return "green".equals(apple.getColor());  
    }  
}  
  
filterApples(inventory, new AppleColorPredicate());
```

메서드 실행 시점에 AppleColorPredicate를 사용

동작 파라미터로 코드 개선하기

- ApplePredicate를 익명 클래스로 구현

ApplePredicate를 익명 클래스로 직접 구현

```
filterApples(inventory, new ApplePredicate() {  
    @Override  
    public boolean test(Apple apple) {  
        return "red".equals(apple.getColor());  
    }  
});
```

동작 파라미터로 코드 개선하기

■ 익명 클래스를 람다로 변환하기 - 1

타입은 파라미터에서 이미 정의되어 있으므로 컴파일러 입장에서는 충분히 예측 가능함

```
filterApples(inventory, new ApplePredicate() {  
    @Override  
    public boolean test(Apple apple) {  
        return "red".equals(apple.getColor());  
    }  
});
```

메서드도 하나만 있으므로 컴파일러에서는
이 메서드를 사용한다고 예측 가능

동작 파라미터로 코드 개선하기

■ 익명 클래스를 람다로 변환하기 - 2

```
filterApples(inventory, (Apple apple) ->
    {return "red".equals(apple.getColor());});
```

```
filterApples(inventory, (Apple apple) ->
    "red".equals(apple.getColor()));
```

```
filterApples(inventory, apple -> "red".equals(apple.getColor()));
```

Predicate 함수형 인터페이스

■ Predicate<T> T->boolean

Predicate

boolean

- 추상메서드 test(T)를 람다의 바디에서 정의하고, T 타입을 인자로 받아서 어떤 조건을 충족 하는지 여부를 확인하여 boolean을 리턴 한다.

```
@FunctionalInterface
```

```
public interface Predicate<T> {
```

```
    * Evaluates this predicate on the given argument. □
```

```
    boolean test(T t);
```

■ 아규먼트 타입과 갯수에 따라 분류

| 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 |

Predicate 함수형 인터페이스 예제

- Predicate 함수형 인터페이스

```
import java.util.function.Predicate;  
  
public List<Apple> filterApples2(List<Apple> inventory,Predicate<Apple> p){
```

1. Predicate를 익명 클래스로 직접 구현하여 호출

```
filterApples2(inventory, new Predicate<Apple>() {  
    @Override  
    public boolean test(Apple apple) {  
        return apple.getColor().equals("green");  
    }  
});
```

2. 람다식으로 호출

```
filterApples2(inventory, apple -> apple.getColor().equals("green"));
```

Consumer 함수형 인터페이스

- **Consumer<T>** $T \rightarrow \text{void}$ 아규먼트 값 → **Consumer**
- 추상 메서드 accept(T)를 람다의 바디에서 정의하고, T 타입을 인수로 받아서 리턴 하지 않는다.

```
@FunctionalInterface  
public interface Consumer<T> {  
    * Performs this operation on the given argument.  
    void accept(T t);
```

- **아규먼트의 타입과 갯수에 따라 분류**

| 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 |

Consumer 함수형 인터페이스 예제

■ Consumer 함수형 인터페이스

```
import java.util.function.Consumer;

public void printAppleInfo(List<Apple> inventory, Consumer<Apple> consumer){
    for (Apple apple : inventory) {
        consumer.accept(apple);
    }
}
```

1. Consumer를 익명 클래스로 직접 구현하여 호출

```
printAppleInfo(inventory, new Consumer<Apple>(){
    @Override
    public void accept(Apple apple) {System.out.println(apple);}
});
```

2. 람다식으로 호출

```
Consumer<Apple> consumer = System.out :: println;
printAppleInfo(inventory, consumer);
printAppleInfo(inventory, apple -> System.out.println(apple));
```

Function 함수형 인터페이스

- **Function<T, R>** T->R 아규먼트값 → **Function** → 리턴값
- 추상메서드 apply(T)를 람다의 바디에서 정의하고, T 타입을 인수로 받아서 R 타입을 리턴 한다.

```
@FunctionalInterface  
public interface Function<T, R> {  
  
    * Applies this function to the given argument.  
    R apply(T t);
```

Function 함수형 인터페이스

■ 아규먼트 타입과 리턴 타입에 따라 분류

| 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 |

Function 함수형 인터페이스 예제

```
import java.util.function.Function;
```

```
public List<String> getColorList(List<Apple> inventory, Function<Apple, String> function){  
    List<String> colorList=new ArrayList<String>();  
    for (Apple apple : inventory) {  
        colorList.add(function.apply(apple));  
    }  
    return colorList;  
}
```

1. Function을 익명 클래스로 직접 구현하여 호출

```
getColorList(inventory, new Function<Apple, String>(){  
    @Override  
    public String apply(Apple apple) {  
        return apple.getColor();  
    }  
});
```

2. 람다식으로 호출 Function<Apple, String> function = Apple :: getColor;
 getColorList(inventory, function);
 getColorList(inventory, apple -> apple.getColor());

미리 정의된 Supplier 함수형 인터페이스

- **Supplier<T> () -> T** **Supplier** → 리턴값

- 추상 메서드 get() 을 람다의 바디에서 정의하며, 인수 없이 T 객체를 리턴한다.

```
@FunctionalInterface  
public interface Supplier<T> {  
  
    * Gets a result.  
    T get();  
}
```

- 리턴 타입에 따라 분류

| 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 |

Supplier 함수형 인터페이스 예제

```
class Vehicle{
    public void drive(){
        System.out.println("Driving vehicle ...");
    }
}
class Car extends Vehicle{
    @Override
    public void drive(){
        System.out.println("Driving car...");
    }
}
public class SupplierDemo {
    static void driveVehicle(Supplier<? extends Vehicle> supplier){
        Vehicle vehicle = supplier.get();
        vehicle.drive();
    }
}
public static void main(String[] args) {
    //Using Lambda expression
    driveVehicle(()-> new Vehicle());
    driveVehicle(()-> new Car());
}
```

Operator 함수형 인터페이스

■ Operator 함수형 인터페이스 아규먼트값 → **Operator** → 리턴값

- 아규먼트(argument)와 리턴(return)값이 모두 있는 추상 메서드를 가진다.
- 주로 아규먼트 값을 연산하고 그 결과를 리턴 할 경우에 사용함

■ 아규먼트 타입과 갯수에 따라 분류

| 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 |

Operator 함수형 인터페이스 예제

```
import java.util.function.IntBinaryOperator;

public static int maxOrMin(IntBinaryOperator operator) {
    int result = scores[0];
    for (int score : scores) {
        result = operator.applyAsInt(result, score);
    }
    return result; }
```

1. 람다식으로 호출

```
// 최대값 얻기
int max = maxOrMin((a, b) -> { if (a >= b) return a;
                                    else return b; });
System.out.println("최대값: " + max);
// 최소값 얻기
int min = maxOrMin((a, b) -> { if (a <= b) return a;
                                    else return b; });
System.out.println("최소값: " + min);
```

함수형 인터페이스의 default 메서드

- `negate()`와 `and()` default 메서드
 - Predicate 함수형 인터페이스는 default 형태의 `negate()`과 `and()` 유ти리티 메서드를 가지고 있다.

```
//Predicate 선언하기
Predicate<Apple> redApple
    = a -> a.getColor().equals("red");
//Predicate 뒤집기
Predicate<Apple> notRedApple
    = redApple.negate();
//red & weight > 150
Predicate<Apple> redHeavyApple
    = redApple.and(a -> a.getWeight() > 150);
```

함수형 인터페이스의 default 메서드

- **andThen()과 compose() default 메서드**
 - Consumer, Function, Operator 함수형 인터페이스는 andThen()과 compose() 디폴트 메서드를 가지고 있음
 - 두개의 함수형 인터페이스를 순차적으로 연결하여 실행
 - andThen과 compose()의 차이점은 어떤 함수형 인터페이스 부터 처리 하느냐에 달려 있다.
- **andThen()과 compose() 디폴트 메서드를 제공하는 함수형 인터페이스들**

| 종류 | 함수형 인터페이스 | andThen() | compose() |
|----------|---------------------|-----------|-----------|
| Consumer | Consumer | 0 | |
| | BiConsumer | 0 | |
| | DoubleConsumer | 0 | |
| | IntConsumer | 0 | |
| | LongConsumer | 0 | |
| Function | Function | 0 | 0 |
| | BiFunction | 0 | |
| Operator | BinaryOperator | 0 | |
| | DoubleUnaryOperator | 0 | 0 |
| | IntUnaryOperator | 0 | 0 |
| | LongUnaryOperator | 0 | 0 |

함수형 인터페이스의 default 메서드 예제

```
//Function 조합
Function<Integer, Integer> f = x -> x + 1;
Function<Integer, Integer> g = x -> x * 2;
//f를 부른 다음 g를 호출한다
Function<Integer, Integer> h = f.andThen(g);
```

```
System.out.println(h.apply(2)); //6
```

```
//Function 조합
Function<Integer, Integer> i = x -> x + 1;
Function<Integer, Integer> j = x -> x * 2;
//j를 부른 다음 i를 호출한다
Function<Integer, Integer> k = i.compose(j);
```

```
System.out.println(k.apply(2)); //5
```

기본형 특화 함수형 인터페이스

■ Autoboxing 비용을 줄일 수 있도록 기본형 특화 함수형 인터페이스를 제공

| 함수형 인터페이스 | 함수 디스크립터 | 기본형 특화 |
|----------------------|-----------------|---|
| Predicate<T> | T->boolean | IntPredicate, LongPredicate, DoublePredicate |
| Consumer<T> | T->void | IntConsumer, LongConsumer, DoubleConsumer |
| Function<T,R> | T->R | IntFunction<R>, IntToDoubleFunction, IntToLongFunction, LongFunction<R>, LongToIntFunction, LongToDoubleFunction, LongToIntFunction, DoubleFunction<R>, ToIntFunction<T>, ToDoubleFunction<T>, ToLongFunction<T> |
| Supplier<T> | ()->T | BooleanSupplier, IntSupplier, LongSupplier, DoubleSupplier |
| UnaryOperator<T> | T->T | IntUnaryOperator, LongUnaryOperator, DoubleUnaryOperator |
| BinaryOperator<L, R> | (L, T)->T | IntBinaryOperator, LongBinaryOperator, DoubleBinaryOperator |
| BiPredicate<L, R> | (L, R)->boolean | |
| BiConsumer<T, U> | (T, U)->void | ObjIntConsumer<T>, ObjLongConsumer<T>, ObjDoubleConsumer<T> |
| BiFunction<T, U, R> | (T, U)-> R | ToIntBiFunction<T, U>, ToLongBiFunction<T, U>, ToDoubleBiFunction<T, U> |

메서드 레퍼런스 (Method Reference)

람다를 간결하게 쓸 수 있도록 도와 주는 문법

클래스(객체)명과 메서드명 사이에 구분자(::)를 붙여 사용한다.

```
(apple) -> apple.getWeight();
```

```
Apple::getWeight;
```

메서드 레퍼런스 (Method Reference)의 종류

- 메서드 참조(Method Reference)의 목적
 - 메서드를 참조해서 매개변수의 정보 및 리턴 타입을 알아내어, 람다식에서 불필요한 매개변수를 제거하는 것이 목적

| 종류 | 형식 |
|----------------|--------------|
| 정적 메서드 참조 | 클래스명::정적메서드명 |
| 객체 메서드 참조 | 객체변수::메서드명 |
| 람다인자 객체 메서드 참조 | 클래스명::메서드명 |
| 생성자 참조 | 클래스명::new |

정적 메서드 참조 (Static Method Reference)

- 클래스 내부에 정의된 정적 메서드 호출을 표현할 수 있다.
 - (인자) -> 클래스.정적메서드(인자)
 - 클래스 :: 정적메서드

```
Function<String, Integer> toNumber1 =  
    (str) → Integer.parseInt(str);
```

```
Function<String, Integer> toNumber2 = Integer::parseInt;
```

객체 메서드 참조 (Instance Method Reference)

- 객체의 인스턴스 메서드 호출을 표현할 수 있다.
 - (인자) -> 객체.인스턴스메서드(인자)
 - 객체 :: 인스턴스메서드

```
Consumer<String> out1 = (str) → System.out.println(str);
```

```
Consumer<String> out2 = System.out :: println;
```

인자 객체 메서드 참조 (Argument Instance Method Reference)

- 람다 인자 객체의 인스턴스 메서드 호출을 간결하게 표현할 수 있다.
 - (인자1, 인자2) -> 인자1.인스턴스메서드(인자2)
 - 인자1의 클래스명 :: 메서드명

```
Comparator<Integer> comp1 = (o1,o2) → o1.compareTo(o2);
```

```
Comparator<Integer> comp2 = Integer :: compareTo;
```

생성자 참조 (Constructor Reference)

- 클래스의 생성자를 간결하게 호출할 수 있다.
 - (인자) -> new 클래스(인자)
 - () -> new 클래스()
 - 클래스::new

```
Supplier<String> str1 = () → new String();
```

```
Supplier<String> str2 = String::new;
```

요약

- 람다 표현식은 익명클래스를 간결하게 표현할 수 있다. (약간은 다르다)
- 함수형 인터페이스는 추상 메서드 하나만 정의된 인터페이스이다.
- Java에서는 자주 사용되는 기본적인 형태의 함수형 인터페이스를 제공한다.
- 기본 제공되는 함수형 인터페이스는 Boxing을 피할 수 있도록 IntPredicate, IntToLongFunction과 같은 primitive 타입의 인터페이스를 제공한다.
- 메서드 레퍼런스를 이용하면 기존의 메서드 구현을 재사용 및 전달 가능하다.
- Comparator, Predicate, Function 같은 함수형 인터페이스는 람다 표현식을 조합 할 수 있는 다양한 디폴트 메서드를 제공한다.

1-03

스트림의 활용

스트림(Stream) 이란?

■ 스트림(Stream)의 정의

- 스트림은 Java 8 부터 추가된 컬렉션(배열 포함)의 저장 요소(Element)를 하나씩 참조해서 람다식 (함수적-스타일, functional-style)으로 처리할 수 있도록 해주는 반복자이다.

```
//기존의 For문을 이용한 방식  
int count = 0;  
for (String w : words) {  
    if (w.length() > 12)  
        count++;  
}
```



```
//Stream을 이용한 방식  
long count = words.stream()  
.filter(w -> w.length()>12)  
.count();
```



```
//병렬처리를 수행하는 Stream을 이용한 방식  
long count = words.parallelStream()  
.filter(w -> w.length()>12)  
.count();
```

Stream을 이용하여 코드 개선하기

■ 요구사항

칼로리가 400 이하인 요리를 추출

칼로리 순으로 정렬하고

상위 3개의 이름을 출력하세요.

| | | |
|---|---|--|
|  삼겹살 331kcal (100g) |  돼지갈비 208kcal (100g) |  쇠고기 등심 218kcal (100g) |
|  안심스테이크 897kcal (1인분) |  돼지곱창구이 737kcal (1인분) |  즉발 768kcal (1인분) |
|  프라이드치킨 269kcal (1인분) |  광어 103kcal (100g) |  복어 89kcal (100g) |
|  흰치 132kcal (100g) |  아기침 506kcal (1인분) |  해물탕 289kcal (1인분) |
|  탕수육 481kcal (1인분) |  깐풍기 616kcal (1인분) |  양장찌개 296kcal (1인분) |
|  소주 141kcal (100g) 소주 1병 = 약 360ml |  레드와인 70kcal (100g) |  위스키 277kcal (100g) |

Stream을 이용하여 코드 개선하기

```
public Dish(String name, boolean vegetarian, int calories, Type type) {  
    this.name = name;  
    this.vegetarian = vegetarian;  
    this.calories = calories;  
    this.type = type; }
```

```
Arrays.asList(  
    new Dish("pork", false, 800, Type.MEAT),  
    new Dish("beef", false, 700, Type.MEAT),  
    new Dish("chicken", false, 450, Type.MEAT),  
    new Dish("french fries", true, 530, Type.OTHER),  
    new Dish("rice", true, 300, Type.OTHER),  
    new Dish("spaghetti", true, 400, Type.NOODLE),  
    new Dish("apple", true, 300, Type.FRUIT),  
    new Dish("melon", true, 320, Type.FRUIT),  
    new Dish("salmon", true, 420, Type.FISH),  
    new Dish("prawn", true, 410, Type.FISH) );
```

Stream을 이용하여 코드 개선하기

■ 클래식 자바 스타일, 컬렉션을 활용한 요구사항 구현

```
List<Dish> lowCaloryDishes = new ArrayList<>();
//칼로리가 400 이하인 메뉴만 가지고 온다.
for (Dish dish : menu) {
    if(dish.getCalories() < 400){ lowCaloryDishes.add(dish); }
}
//칼로리 순으로 정렬
Collections.sort(lowCaloryDishes, new Comparator<Dish>() {
    @Override
    public int compare(Dish o1, Dish o2) {
        return Integer.compare(o1.getCalories(), o2.getCalories()); }
});
//음식 이름만 가지고 온다.
List<String> lowCaloryDishesName = new ArrayList<>();
for (Dish dish : lowCaloryDishes) { lowCaloryDishesName.add(dish.getName()); }
//상위 3개의 결과만 반환한다.
List<String> lowCaloryLimit3DishesName = lowCaloryDishesName.subList(0, 3);
System.out.println(lowCaloryLimit3DishesName); //rice, apple, melon]
```

Stream을 이용하여 코드 개선하기

- Stream API를 활용한다면 !!

```
//Stream API 활용
List<String> lowCaloryDishesNames = menu.stream()
    .filter(dish -> dish.getCalories() < 400)
    .sorted(Comparator.comparing(Dish::getCalories))
    .map(Dish::getName)
    .collect(Collectors.toList());

System.out.println(lowCaloryDishesNames); // [rice, apple, melon]
```

Stream을 이용하여 코드 개선하기

■ 또 다른 요구사항

1. 400 칼로리 이하인 메뉴를 다이어트로, 아닐 경우에는 일반으로 나누어라.

```
Map<String, List<Dish>> groupedMenu = menu.stream()
    .collect(Collectors.groupingBy(d -> {
        if (d.getCalories() <= 400)
            return "diet";
        else
            return "normal";
    }));
System.out.println(groupedMenu);
```

2. 가장 칼로리가 높은 메뉴를 찾아라.

```
Dish dish = menu.stream()
    .max(Comparator.comparingInt(Dish::getCalories))
    .get();
System.out.println(dish); //pork
```

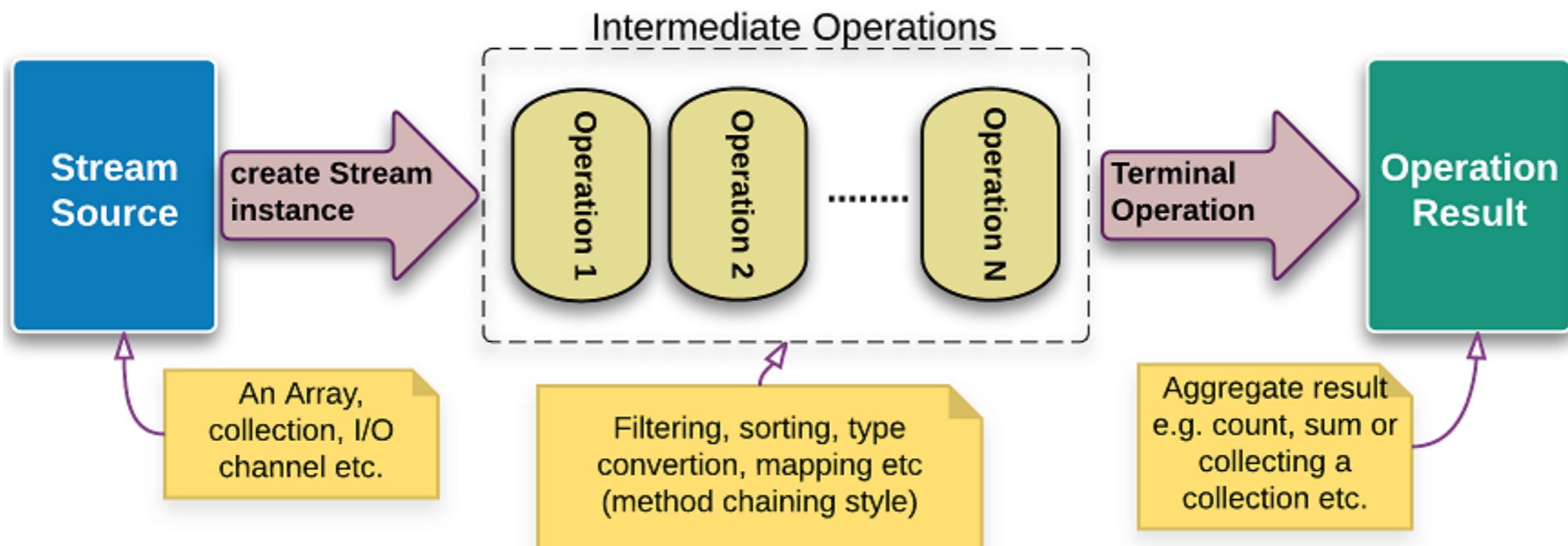
스트림(Stream)과 관련된 용어

- **리덕션(Reduction)**
 - 대량의 데이터를 가공해 축소 하는 것
 - 데이터의 합계, 평균값, 카운팅, 최대값, 최소값
 - 컬렉션의 요소를 리덕션의 결과물로 바로 집계할 수 없을 경우에는?
: 집계하기 좋도록 필터링, 매팅, 정렬, 그룹핑의 중간 처리가 필요함(스트림 파이프라인 필요성)
- **파이프라인(Pipeline)**
 - 여러 개의 스트림이 연결 되어 있는 구조
 - 파이프라인에서 최종 처리를 제외 하고는 모두 중간 처리 스트림



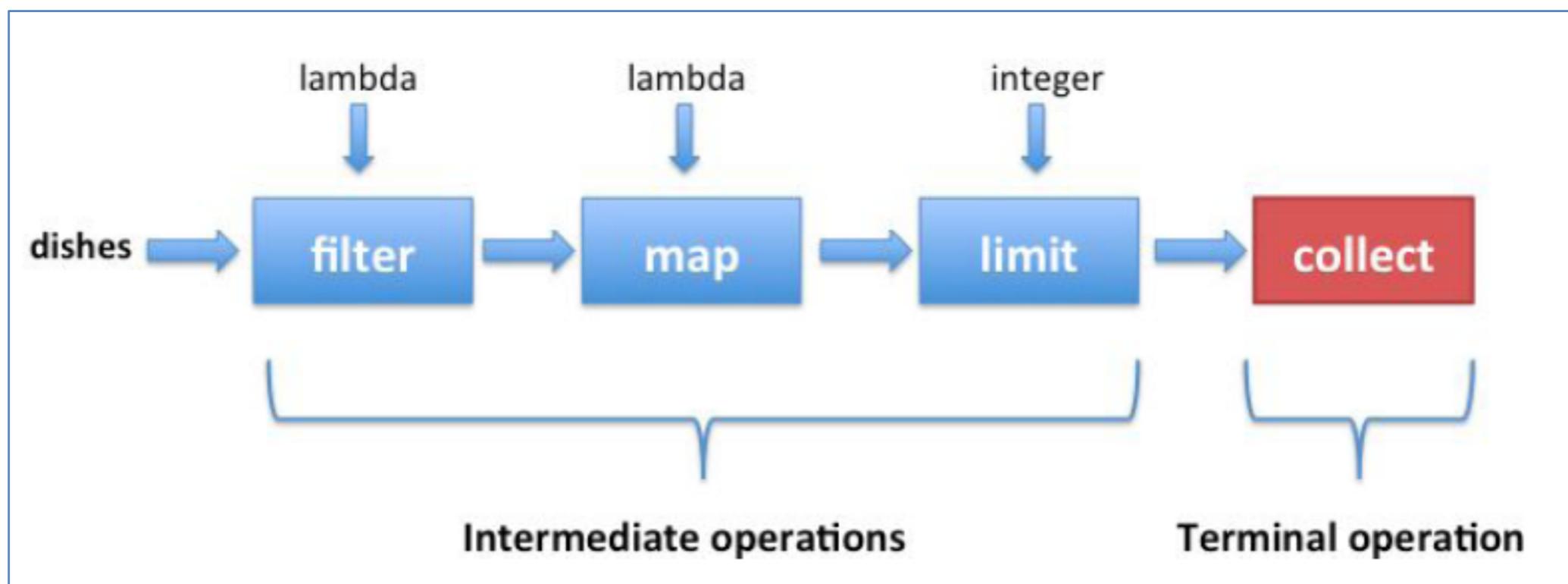
스트림(Stream)의 작업흐름

1. 스트림을 생성한다.
2. 초기 스트림을 다른 스트림으로 변환하는 중간 연산을 지정한다. 여러 단계가 될 수도 있다.
3. 종료 연산을 적용해서 결과를 산출한다. 종료 연산은 앞에서 지연된 중간 연산이 실행되게 한다.



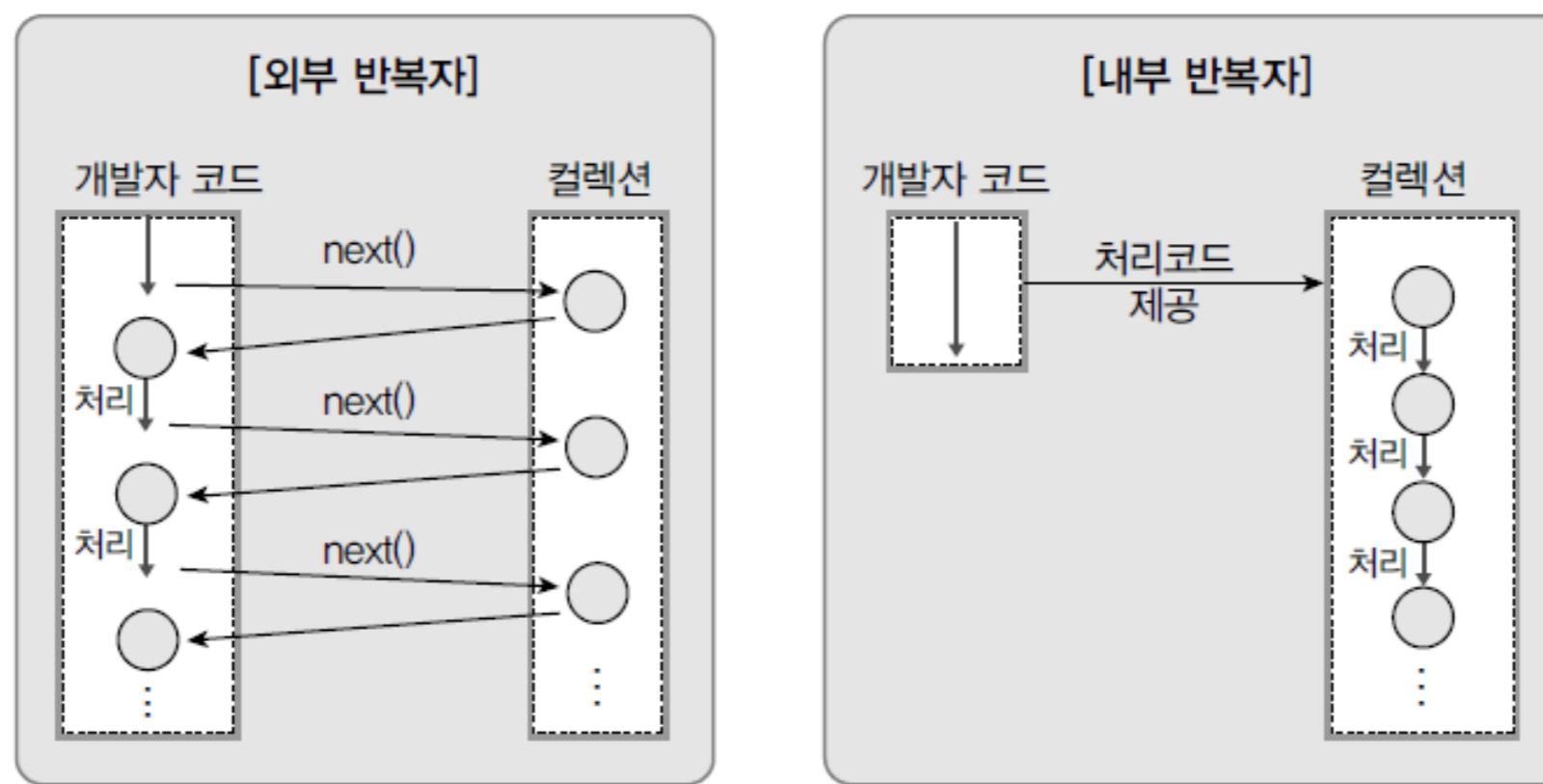
스트림(Stream) 특징

- 1. 스트림(Stream)은 Iterator와 비슷한 역할을 하는 반복자이다.
- 2. 스트림(Stream)이 제공하는 요소 처리 메서드는 함수적 인터페이스 타입으로 람다식 또는 메서드 참조를 이용하여 요소(Element) 처리 내용을 인자로 전달할 수 있다.
- 3. Stream API는 이 메서드들을 연결하여, 복잡한 연산을 처리하는 로직을 쉽고 유연하게 작성해 낼 수 있다.



스트림(Stream) 특징

- 4. 내부 반복자를 사용하므로 병렬 처리가 쉽다.
 - 외부 반복자(external iterator)는 개발자가 직접 컬렉션의 요소를 반복해서 가져오는 코드 패턴이다.
=> index를 이용하는 for문, Iterator를 이용하는 while문은 모두 외부 반복자를 이용 하는 방식이다.
 - 내부 반복자(internal iterator)는 컬렉션 내부에서 요소들을 반복 시키고, 개발자는 요소 별로 처리해야 할 코드만 제공하는 코드 패턴이다.
 - 내부 반복자(internal iterator)를 이점은 컬렉션 내부에서 어떻게 요소를 반복시킬 것인가를 컬렉션에게 맡겨 두고,
개발자는 요소 처리 코드에만 집중할 수 있다.



스트림(Stream)의 특징 - 반복의 내재화

Collection

```
for(int n: numbers) {  
    ...  
}
```

외부반복

External Iteration

명시적 외부 반복

제어 흐름 중복 발생

효율적이고 직접적인 요소 처리

지저분한 코드

유한 데이터 구조 API

Stream

```
numbers.forEach(n -> ...)
```

내부반복

Internal Iteration

반복 구조 캡슐화

제어 흐름 추상화

파이프-필터 기반 API

최적화와 알고리즘 분리

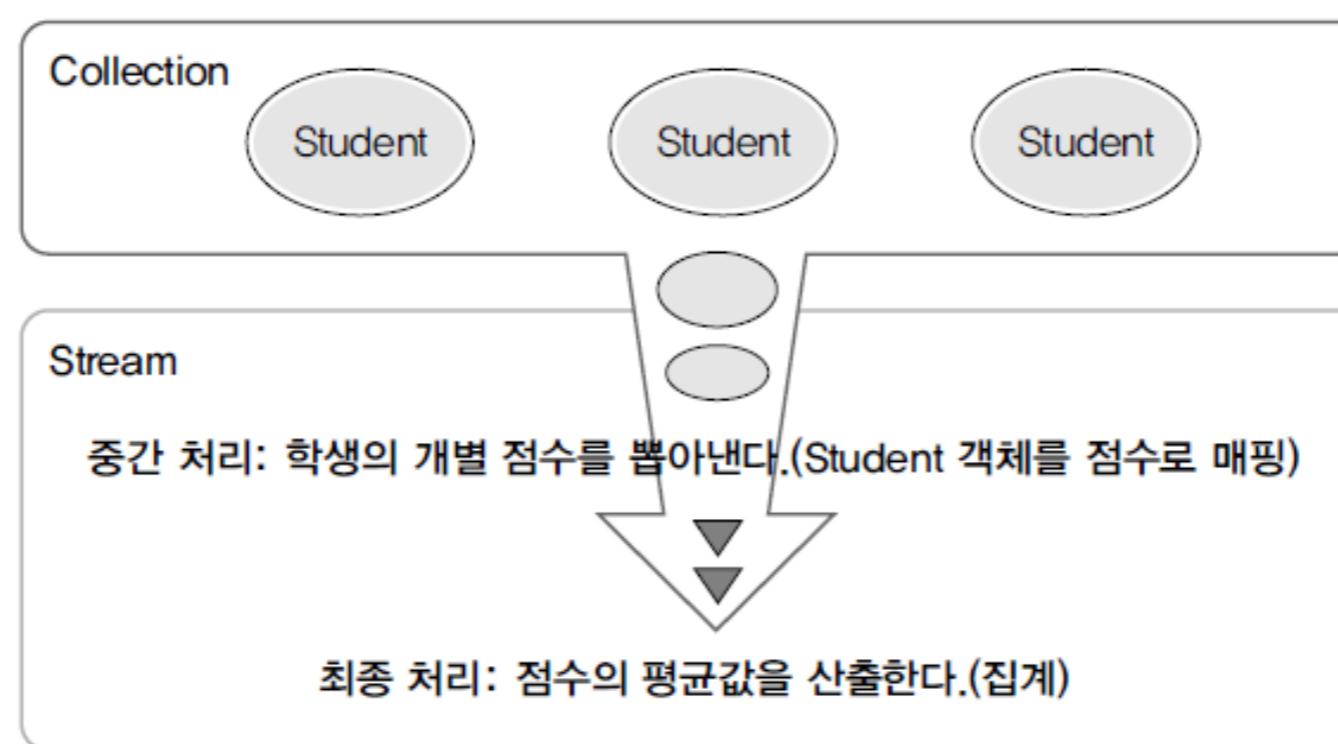
함축적인 표현

무한 연속 데이터 흐름 API

데이터 외 I/O, 값 생성 등 적용

스트림(Stream) 특징

- 5. 스트림은 중간 처리와 최종 처리를 할 수 있다.
 - 중간처리는 맵핑, 필터링, 소팅을 수행하고, 최종처리는 반복, 카운팅, 평균, 총합 등의 집계 처리를 수행한다.



컬렉션 vs 스트림

- 같은 점
 - 컬렉션과 스트림 모두 연속된 요소 형식의 값을 저장하는 자료구조의 인터페이스를 제공한다.
 - 둘 다 순서에 따라 순차적으로 요소에 접근한다.
- 다른 점
 - 컬렉션 : 각 계산식을 만날 때마다 데이터가 계산된다.
 - 스트림 : 최종 연산이 실행될 때 데이터가 계산된다.
- 다른 점 (데이터 접근 측면)
 - 컬렉션 : 자료구조 이므로 데이터에 접근, 읽기, 변경, 저장 같은 연산이 주요 관심사이다.
(직접 데이터 핸들링) 즉 데이터에 접근하는 방법을 직접 작성해야 한다.
 - 스트림 : filter, sorted, map 처럼 계산식(람다)을 표현하는 것이 주요 관심사이다.
(계산식을 JVM에게 던진다) 즉 데이터에 접근하는 방법이 추상화되어 있다.

컬렉션 vs 스트림

- 스트림은 최대한 연산을 지연시킨다.

```
Stream<Integer> filteredStream =  
    Arrays.asList(1, 2, 3, 4, 5)  
        .stream()  
        .filter(x -> x > 2);  
  
Stream<Integer> doubledStream =  
    filteredStream.map(x -> x * 2);  
  
long count = doubledStream.count();
```

count()를 실행할 때 까지 filter와 map을 실행하지 않음

컬렉션 vs 스트림

- 다른 점 (데이터 계산 측면)

- 컬렉션 :

- 작업을 위해서 Iterator로 모든 요소를 순환해야 한다.
 - 메모리에 모든 요소가 올라가 있는 상태에서 요소들을 누적시키며 결과를 계산한다.
 - 메모리 사용량이 늘어난다.

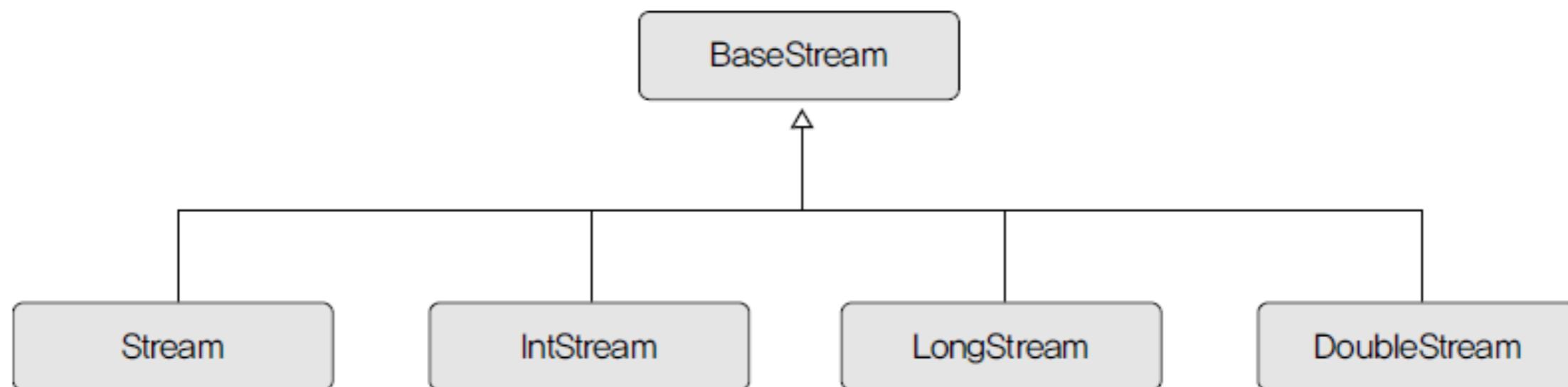
- 스트림 :

- 계산식(알고리즘)을 미리 적어 두고 계산시에 람다식으로 JVM에 넘긴다.
 - 내부에서 요소들을 어떻게 메모리에 올리는지는 관심사가 아니다 (블랙박스)
 - 메모리 사용량이 줄어든다.

스트림(Stream) 종류

■ java.util.stream 패키지의 Stream API

- BaseStream 인터페이스는 모든 스트림에서 사용할 수 있는 공통 메서드 들이 정의되어 있을 뿐 코드에서 직접 사용하지는 않는다.
- Stream은 객체 요소 처리
- IntStream, LongStream, DoubleStream은 각각 기본 타입인 int , long, double 요소 처리



스트림(Stream) 유형

- `java.util.stream`
 - `Stream<T>`: 객체를 요소(element)로 하는 가장 일반적인 스트림
 - `IntStream`: 요소(element)의 타입이 int인 스트림
 - `LongStream`: 요소(element)의 타입이 long인 스트림
 - `DoubleStream`: 요소(element)의 타입이 double인 스트림

스트림(Stream) 종류

■ Stream은 주로 컬렉션과 배열에서 얻음

| 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 |

스트림(Stream) 생성 방법

- 다양한 방식의 스트림 생성 방법 제공
 - Collection: 콜렉션객체.stream(), parallelStream()
 - Files: Stream<String> Files.lines()
 - BufferedReader: Stream<String> lines()
 - Arrays: Arrays.stream(*)
 - Random: Random.doubles(*), ints(*), longs(*)
 - BitSet : IntStream()
 - Stream:
 - Stream.of(*)
 - range(start, end), rangeClosed(start, end)
 - IntStream, LongStream에서 제공
 - Stream.generate(Supplier<T> s)
 - Stream.iterate(T seed, UnaryOperator<T> f)

스트림(Stream) 생성 방법 예제

```
//Stream의 static 메서드 of 사용 - Stream.of(data)
```

```
Stream<String> stream = Stream.of("Java8","Lambdas","Stream","Nashorn");
```

```
//Stream static 메서드 iterate 사용 - Stream.iterate(seed,operator)
```

```
Stream.iterate(0, n -> n + 2)
    .limit(10)
    .forEach(System.out::println);
```

```
//Stream static 메서드 generate 사용 - Stream.generate(()->T)
```

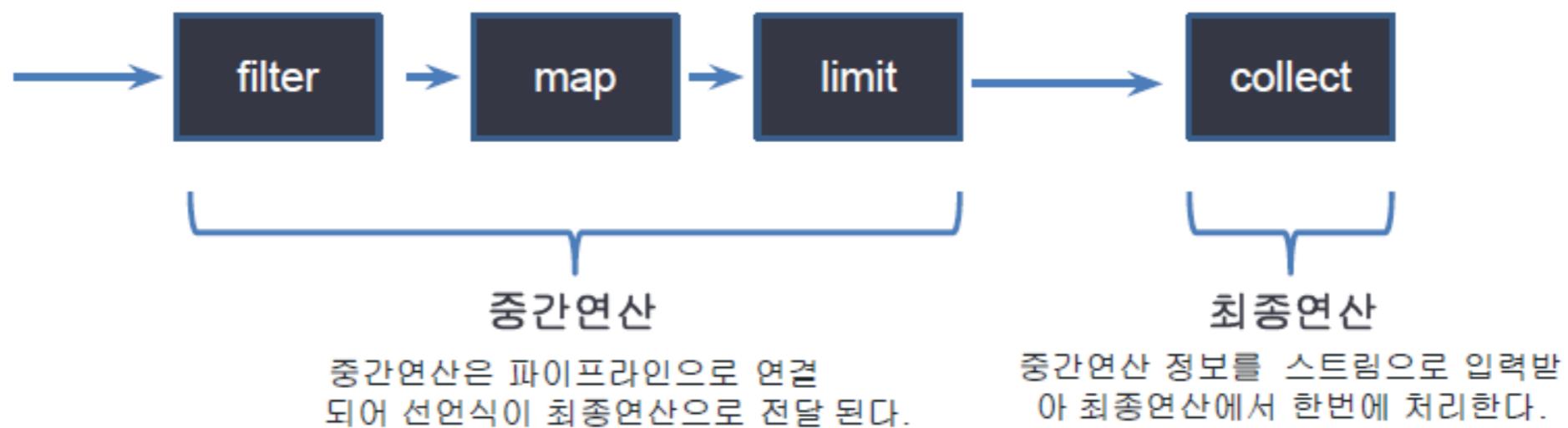
```
Stream.generate(Math::random)
    .limit(10)
    .forEach(System.out::println);
```

```
//Arrays static 메서드 stream 사용 - Arrays.stream(Array[])
```

```
int[] numbers = {2,3,5,7,11};
int sum = Arrays.stream(numbers).sum();
```

스트림(Stream)의 2가지 연산

```
menu.stream()  
.filter(d -> d.getCalories() > 300)  
.map(Dish::getName)  
.limit(3)  
.collect(Collectors.toList()); // [pork, beef, chicken]
```



- 단말 연산을 스트림 파이프 라인에 실행하기 전까지는 아무 연산도 수행하지 않는다.(Lazy)
- 모든 중간 연산을 합친 다음 최종연산에서 한번에 처리한다.
- `filter`, `map`, `limit`는 서로 연결되어 파이프 라인을 형성한다.
- `collect`로 마지막 파이프 라인을 수행 후 완료한다.

스트림의 2가지 연산 : 중간연산

■ Stream이 제공하는 중간 처리용 메서드 – 리턴 타입이 스트림

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

스트림의 2가지 연산 : 중간연산

■ 주요 중개 연산: 상태 없음

- Stream<R> map(Function<? super T, ? extends R> mapper)
 - 입력 T 타입 요소를 1:1로 R 타입 요소로 변환한 스트림 생성
- Stream<T> filter(Predicate<? super T> predicate)
 - T타입의 요소를 확인해서 조건을 충족하는 요소만으로 제공하는 새로운 스트림 생성
- Stream<R> flatMap(Function<? super T, ? extends Stream<? extends R>> mapper)
 - T 타입 요소를 1:N의 R 타입 요소로 변환한 스트림 생성
- Stream<T> peek(Consumer<? super T> action)
 - T 타입 요소를 사용만 하고, 기존 스트림을 그대로 제공하는 스트림 생성
- Stream<T> skip(long n)
 - 처음 n개의 요소를 제외한 나머지 요소로 새 스트림 생성
- Stream<T> limit(long maxSize)
 - maxSize 까지의 요소만 제공하는 스트림 생성
- mapToInt(), mapToLong(), mapToDouble()

```
IntStream.range(1, 100).filter(n -> n % 2 == 0).map(n -> n*n).skip(10).limit(10)
```

스트림의 2가지 연산 : 중간연산

- 주요 중개 연산: 내부 상태 있음
 - Stream<T> sorted()
 - 정렬된 스트림 생성
 - T 타입은 Comparable을 구현한 타입
 - Stream<T> sorted(Comparator<? super T> comparator)
 - 정렬된 스트림을 생성
 - 전체 스트림의 요소를 정렬하기 때문에, 무한 스트림에 적용할 수 없음
 - Stream<T> distinct()
 - 같은 값을 갖는 요소를 제거한 스트림 생성

스트림의 2가지 연산 : 최종연산

- Stream이 제공하는 최종 처리용 메서드 – 리턴 타입이 기본타입이거나 Optional

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

스트림의 2가지 연산 : 최종연산

- Stream 타입의 주요 최종(종단) 연산자
 - void forEach(Consumer<? super T> consumer)
 - T 타입의 요소(Element)를 하나씩 처리
 - R collect(Collector<? Super T,R> collector)
 - T 타입의 요소(Element)를 모두 모아 하나의 자료구조나 값으로 변환
 - Iterator<T> iterator()
 - Iterator 객체반환
 - Optional<T> max(Comparator<? super T> comparator)
 - Optional<T> min(Comparator<? super T> comparator)
 - boolean allMatch(Predicate<? super T> predicate)
 - boolean anyMatch(Predicate<? super T> predicate)
 - boolean noneMatch(Predicate<? super T> predicate)

스트림의 2가지 연산 : 최종연산

- IntStream, LongStream, DoubleStream
 - sum(), min(), max()
 - OptionalDouble average()
- reduce 연산
 - Optional<T> reduce(BinaryOperator<T> accumulator)
 - T 타입의 요소(Element) 둘 씩 reducer로 계산해 최종적으로 하나의 값을 계산
 - T reduce(T identity, BinaryOperator<T> accumulator)
 - <U> U reduce(U identity, BiFunction<U, ? super T, U> accumulator, BinaryOperator<U> combiner)

```
IntStream.range(1, 100).filter(n -> n % 2 == 0).map(n -> n*n)
    .skip(10)
    .limit(10)
    .reduce(0, Integer::sum)
```

스트림의 2가지 연산 : 최종연산

■ collect 연산: 스트림 요소 수집

- ○ <R, A> R collect(Collector<? super T, A, R> collector)
 - T: 입력 타입, A: 결과 축적용 타입, R: 최종 타입

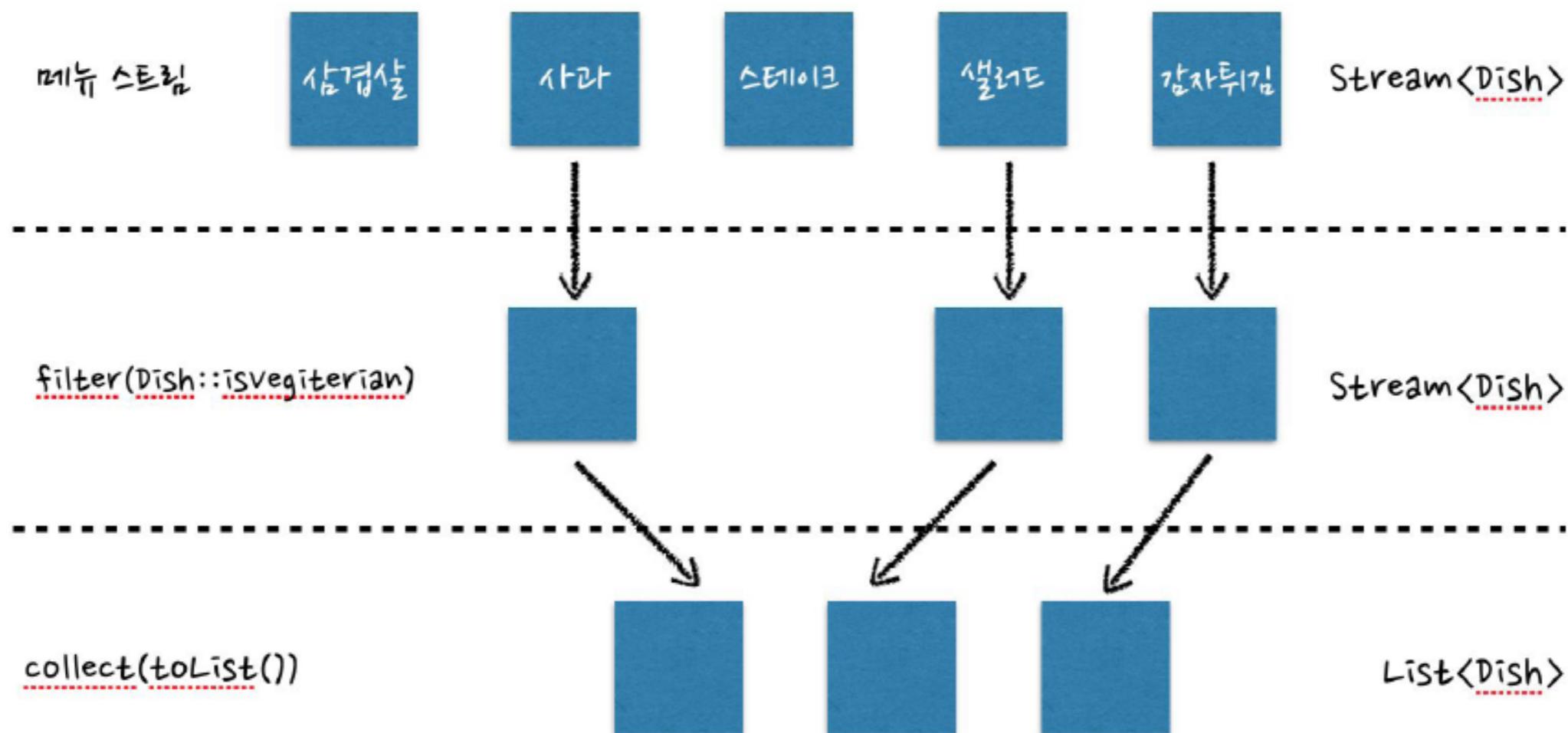
■ Collector 인터페이스 메서드

- Supplier<A> supplier(): A 객체 생성
- BiConsumer<A, T> accumulator(): 결과 축적
- BinaryOperator<A> combiner(): 부분 결과들을 합칠 때 사용
- Function<A, R> finisher(): A를 최종 타입 R로 변환
- Set<Characteristics> characteristics(): 힌트
 - CONCURRENT: 다중 쓰레드에서 실행 가능
 - UNORDERED: 순서에 상관 없음
 - IDENTITY_FINISH: A와 R이 같은 타입

스트림 API의 활용 – 필터링/슬라이싱

- filter (중간연산) : Predicate를 인자로 받아서 true인 요소만을 반환

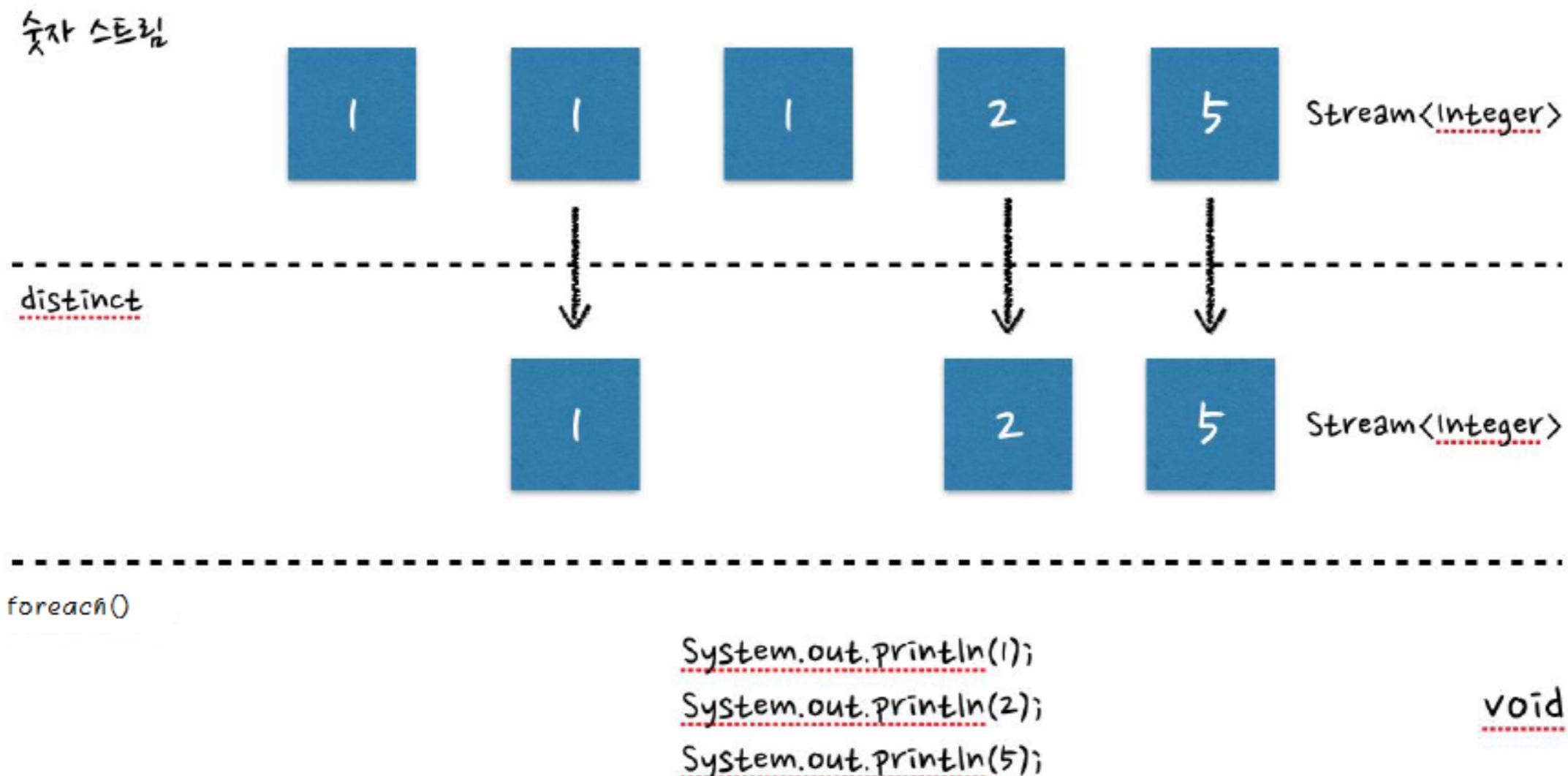
```
menu.stream()  
    .filter(Dish::isVegetarian)  
    .collect(Collectors.toList());
```



스트림 API의 활용 – 필터링/슬라이싱

- `distinct(중간연산)` : 유일한 값을 반환한다.

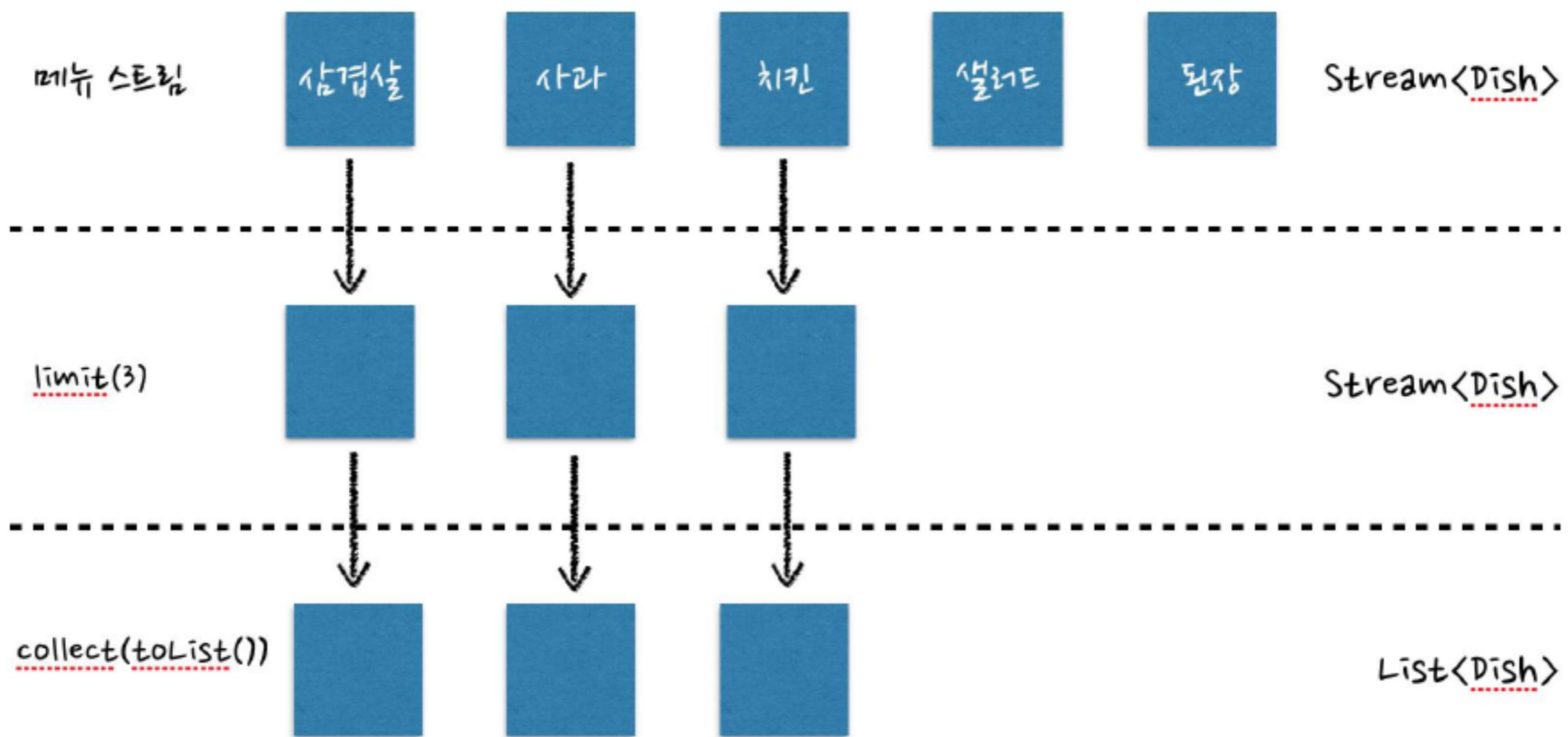
```
List<Integer> numbers = Arrays.asList(1,1,1,2,5);
numbers.stream()
    .distinct()
    .forEach(System.out::println);
```



스트림 API의 활용 – 필터링/슬라이싱

- limit (중간연산) : 지정된 숫자 만큼 반환한다.

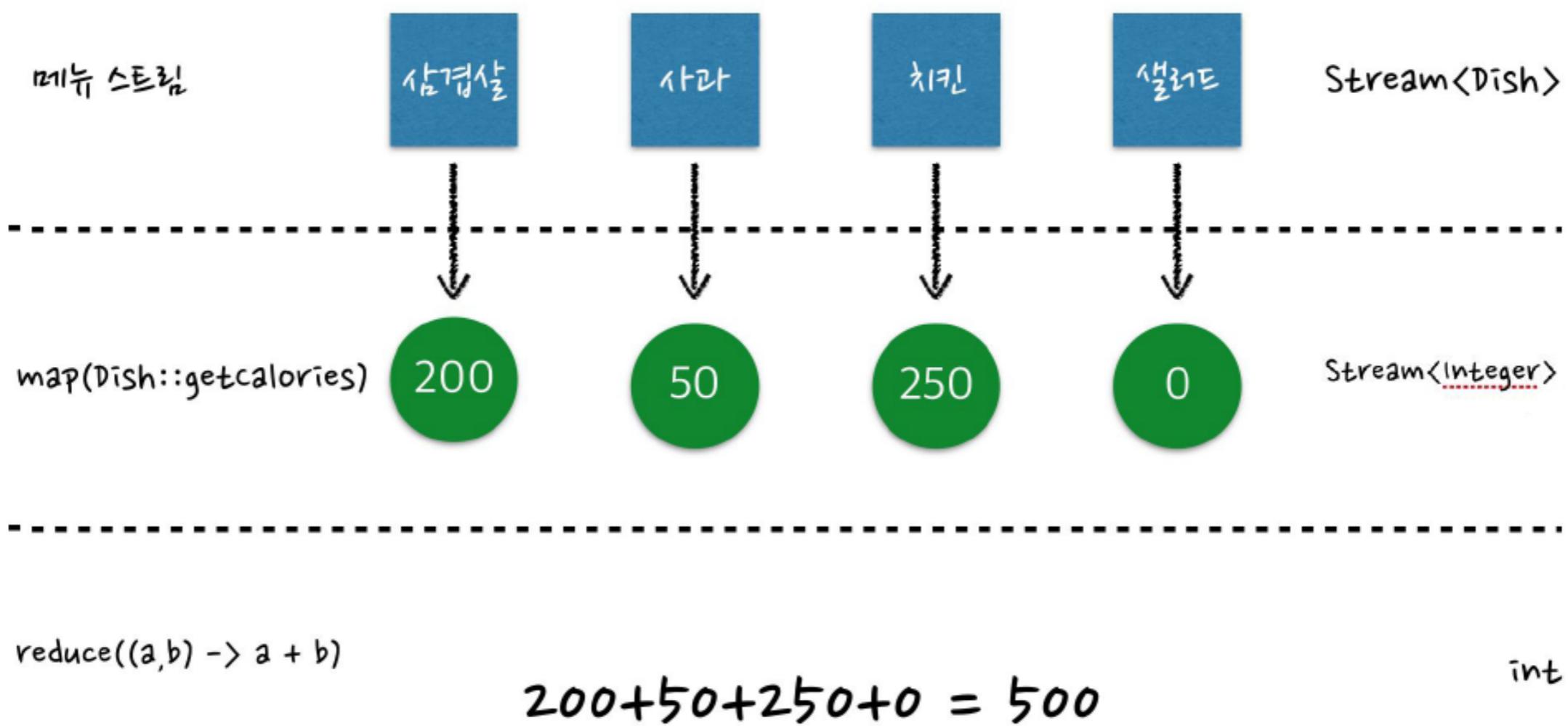
```
menu.stream()  
    .limit(3)  
    .collect(Collectors.toList());
```



스트림 API의 활용 – 맵핑

- map(중간연산) : 스트림의 T 객체를 U로 변환. 파라미터로 Function<T,U>를 사용.

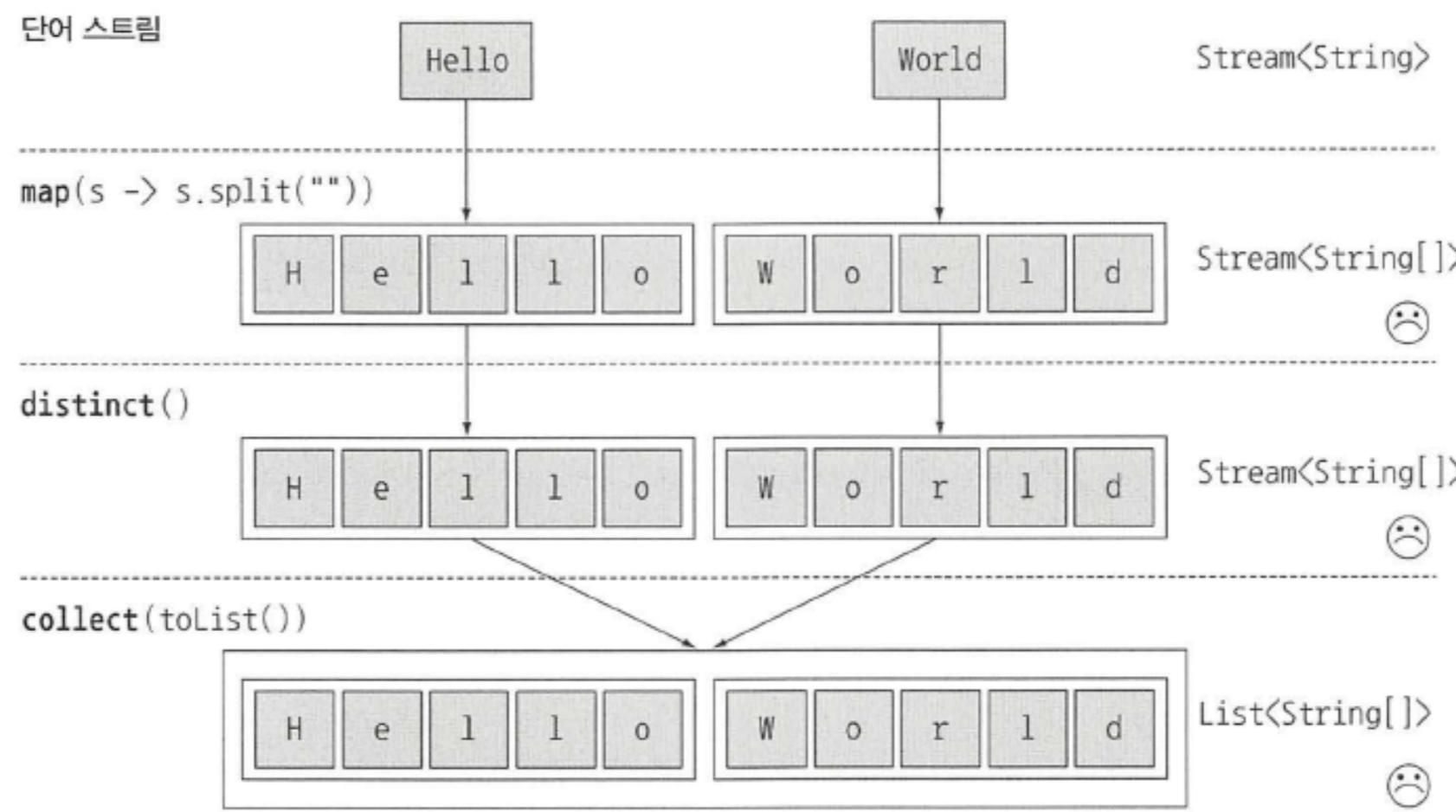
```
menu.stream()
    .map(Dish::getCalories)
    .reduce((prev,curr) -> prev + curr);
```



스트림 API의 활용 – 매핑

- map (중간연산) 을 이용하여 고유한 문자열 찾기 (실패)

```
List<String> words = Arrays.asList( "Hello", "World" );
words.stream()
    .map(word -> word.split(""))
    .distinct()
    .collect(Collectors.toList());
```

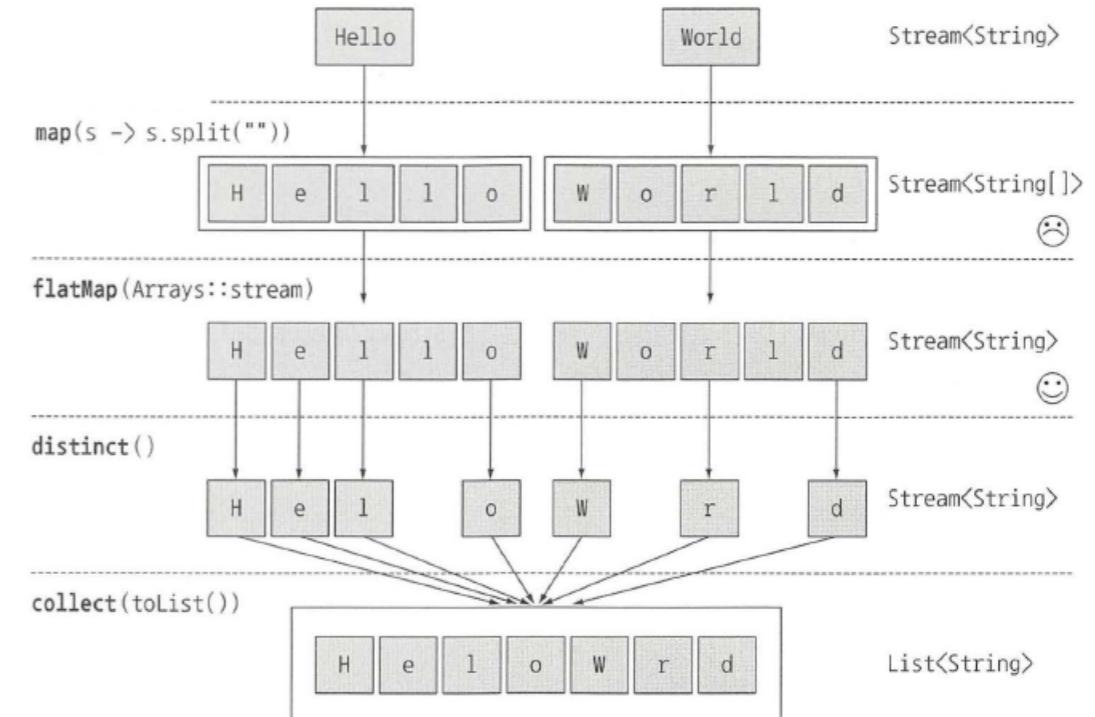


스트림 API의 활용 – 매핑

■ flatMap (중간연산) 을 이용하여 고유한 문자열 찾기 (성공)

```
List<String> words = Arrays.asList( "Hello", "World" );
words.stream()
    .map(word -> word.split( regex: ""))
    .flatMap((String[] strs) -> Arrays.stream(strs))
    .distinct()
    .forEach(System.out::println);

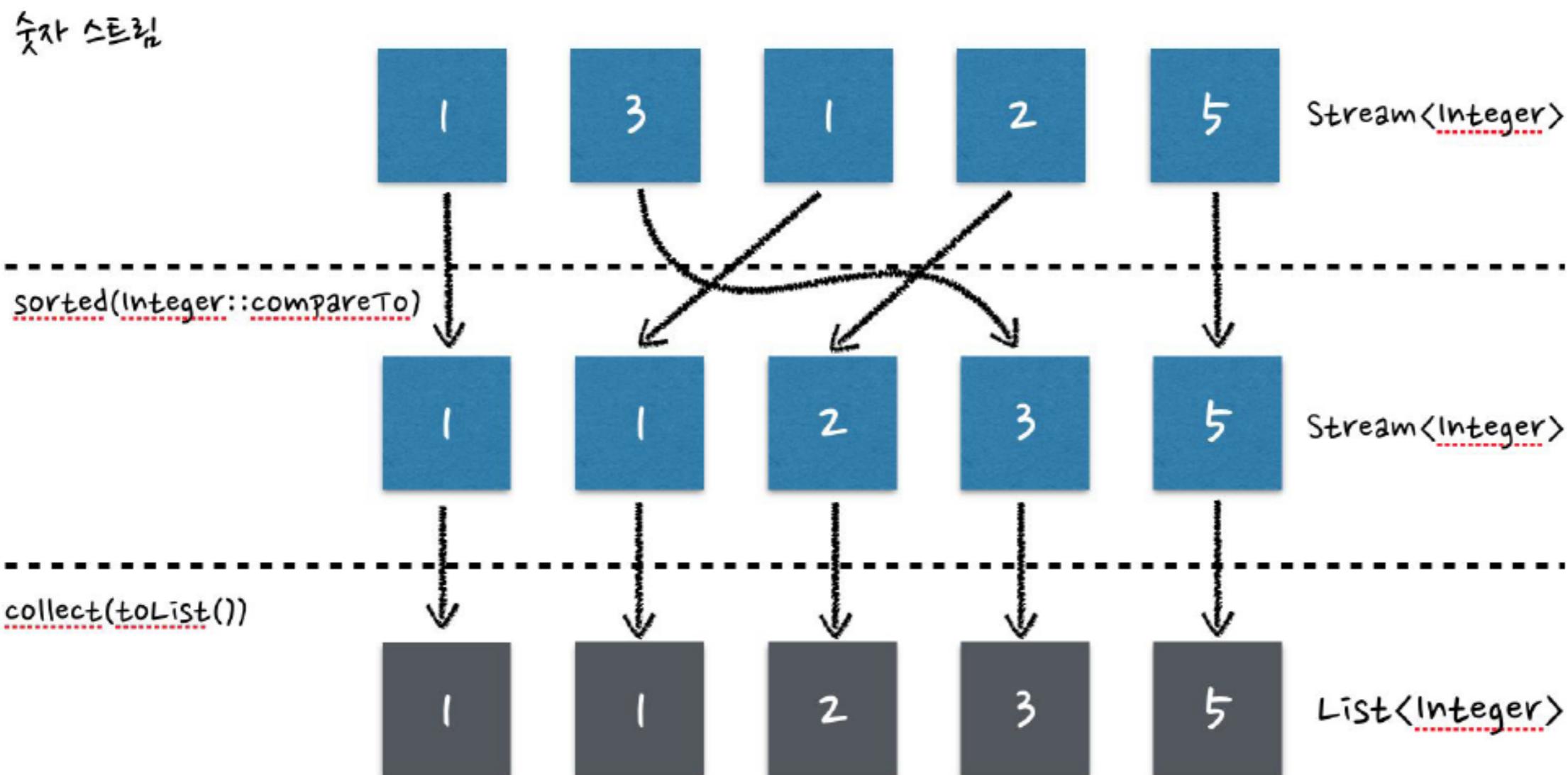
words.stream()
    .flatMap((String line) ->
        Arrays.stream(line.split( regex: "")))
    .distinct()
    .forEach(System.out::println);
```



스트림 API의 활용 – 검색과 매칭

- sorted(중간연산) : Comparator를 사용, 스트림의 요소를 정렬

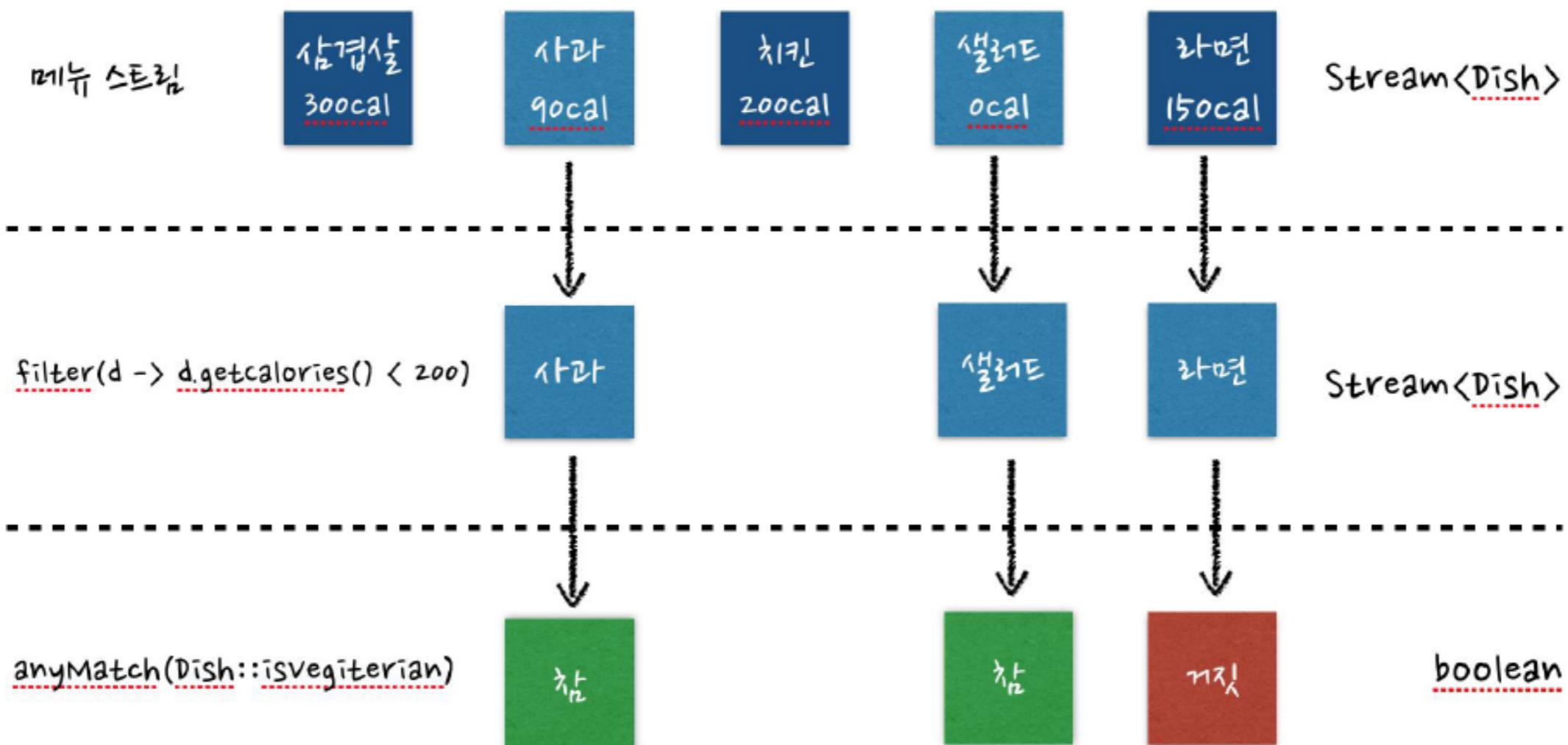
```
Arrays.asList(1,3,1,2,5)
    .stream().sorted(Integer :: compareTo);
```



스트림 API의 활용 – 검색과 매칭

- anyMatch(최종연산) : Predicate가 적어도 한 요소와 일치 하는지 확인한다.

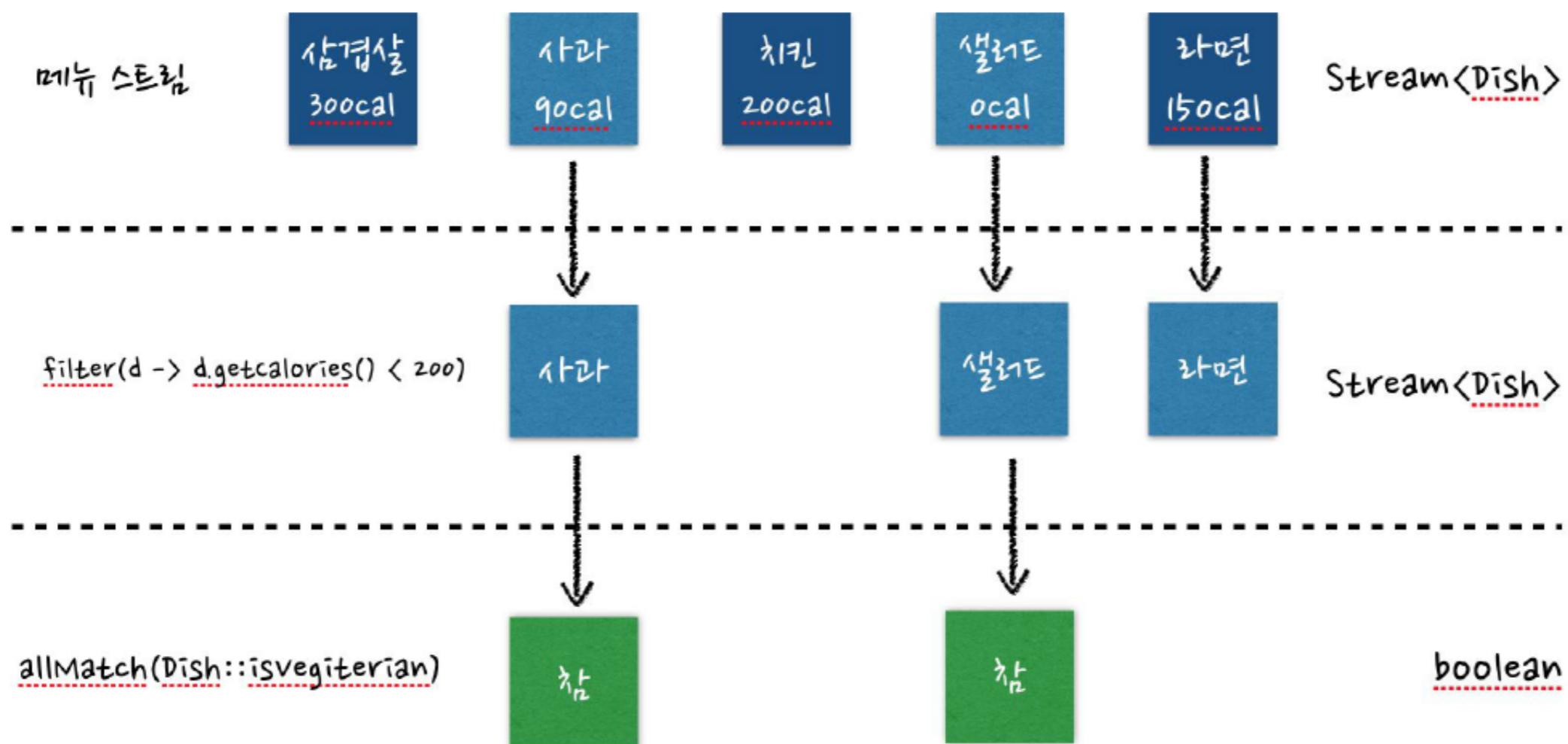
```
menu.stream()
    .filter(d -> d.getCalories() < 200 )
    .anyMatch(Dish::isVegetarian);
```



스트림 API의 활용 – 검색과 매칭

- allMatch (최종연산) : Predicate가 모든 요소와 일치 하는지 확인한다.

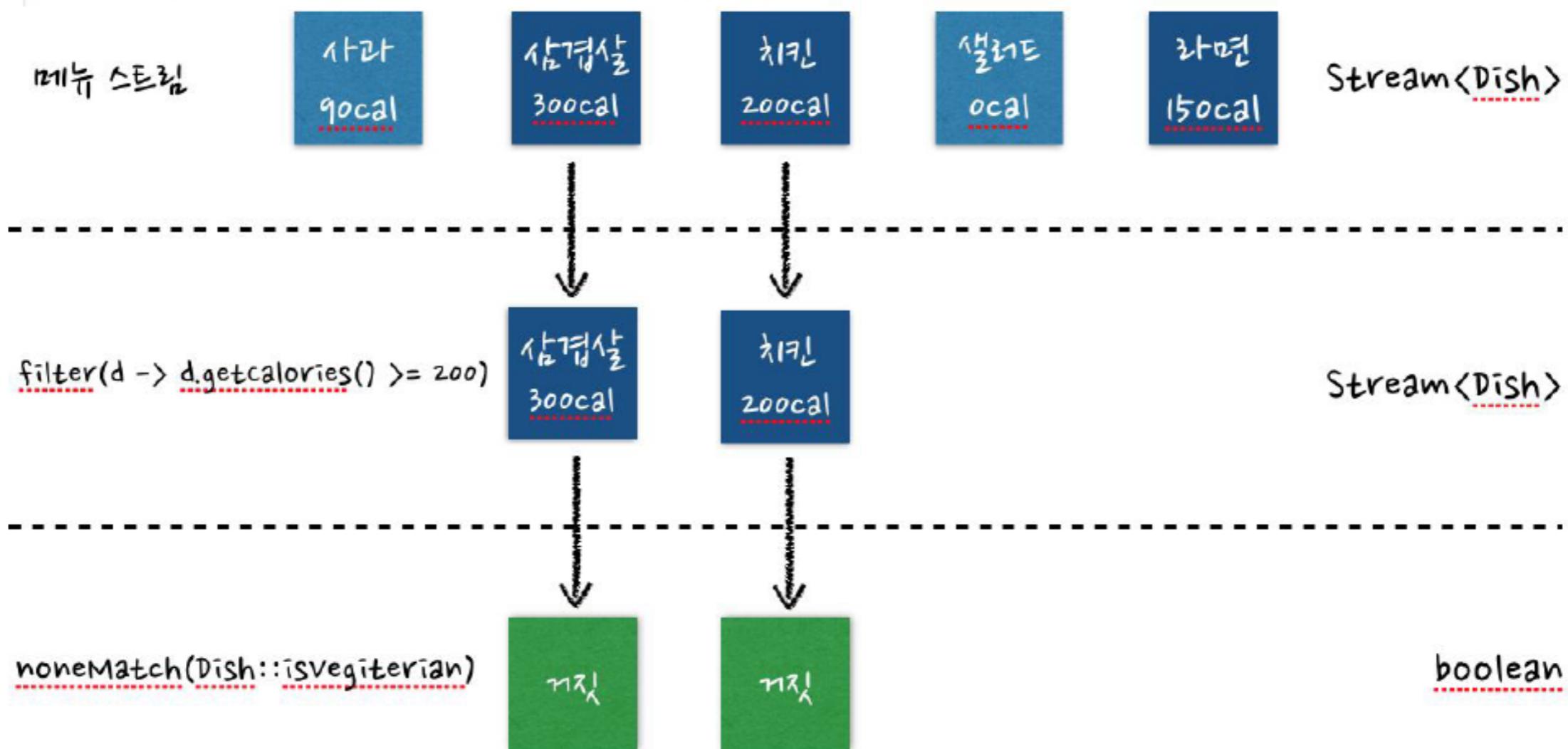
```
menu.stream()
    .filter(d -> d.getCalories() < 200 )
    .allMatch(Dish::isVegetarian);
```



스트림 API의 활용 – 검색과 매칭

- noneMatch (최종연산) : Predicate가 모든 요소와 불일치 하는지 확인한다.

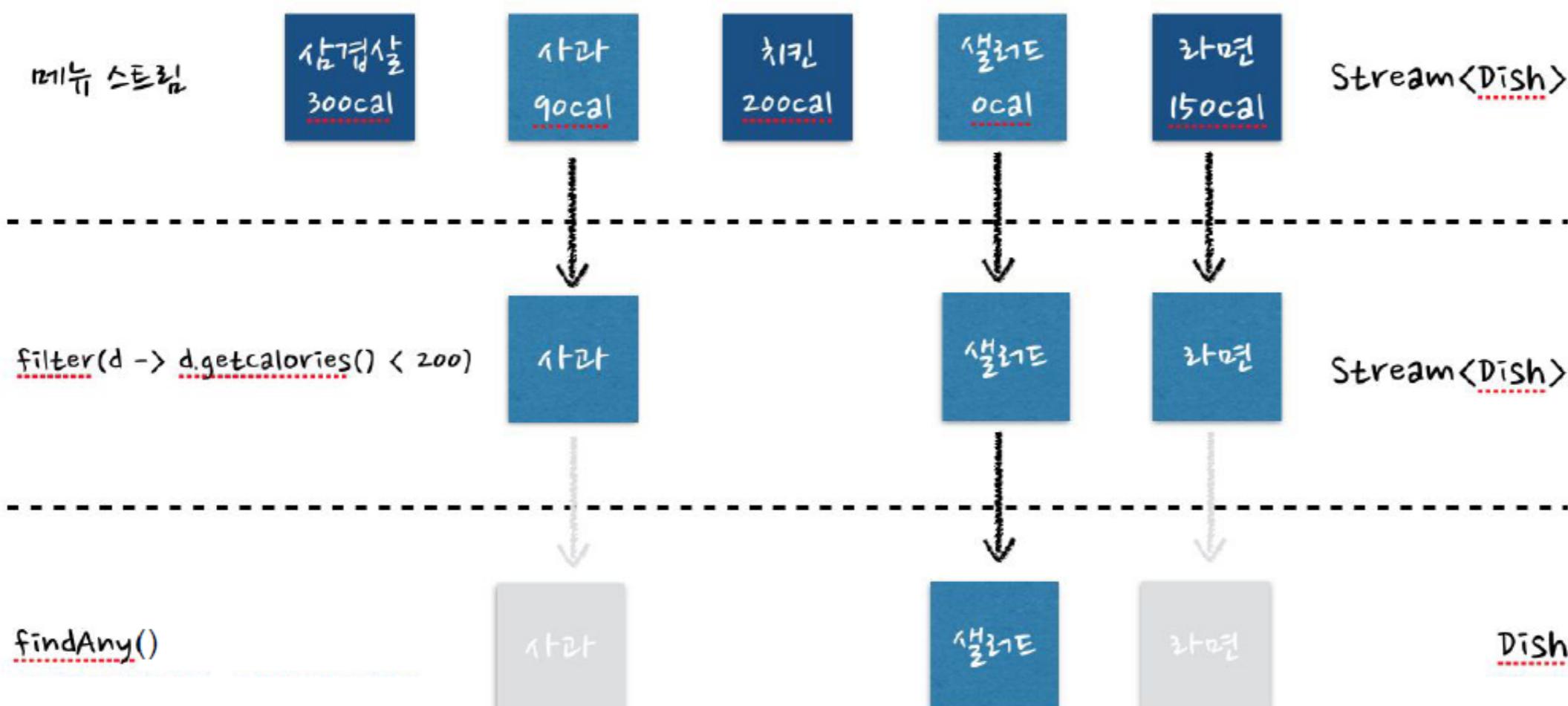
```
menu.stream()
    .filter(d -> d.getCalories() >= 200)
    .noneMatch(Dish::isVegetarian);
```



스트림 API의 활용 – 검색과 매칭

- `findAny(최종연산)` : 현재 스트림에서 임의의 요소를 반환한다.

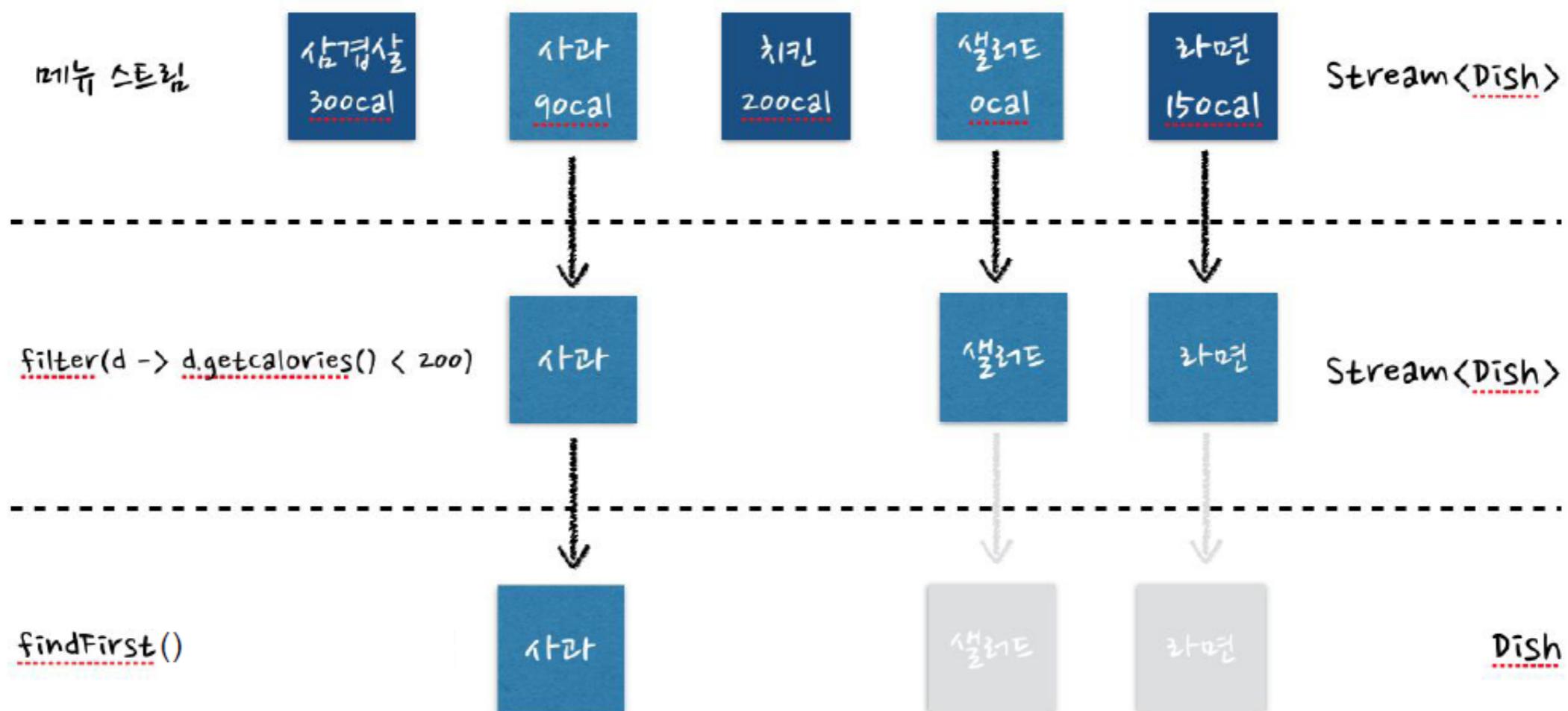
```
menu.stream()  
    .filter(d -> d.getCalories() < 200)  
    .findAny();
```



스트림 API의 활용 – 검색과 매칭

- `findFirst(최종연산)` : 현재 스트림에서 첫번째 요소를 반환한다.

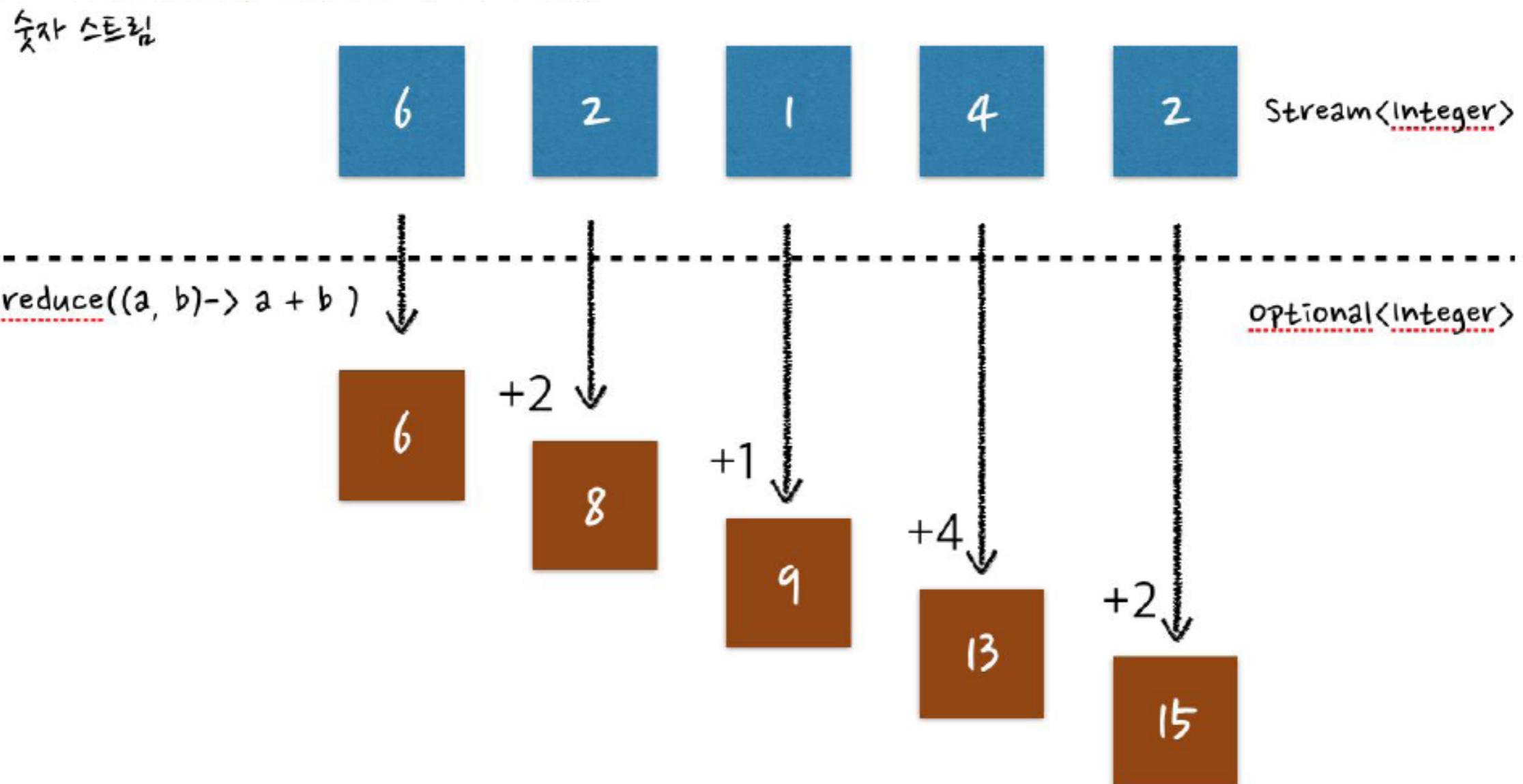
```
menu.stream()  
.filter(d -> d.getCalories() < 200)  
.findFirst();
```



스트림 API의 활용 – 검색과 매칭

- reduce [최종연산] : reduce(init,operator) 또는 reduce(operator) 형태로 사용

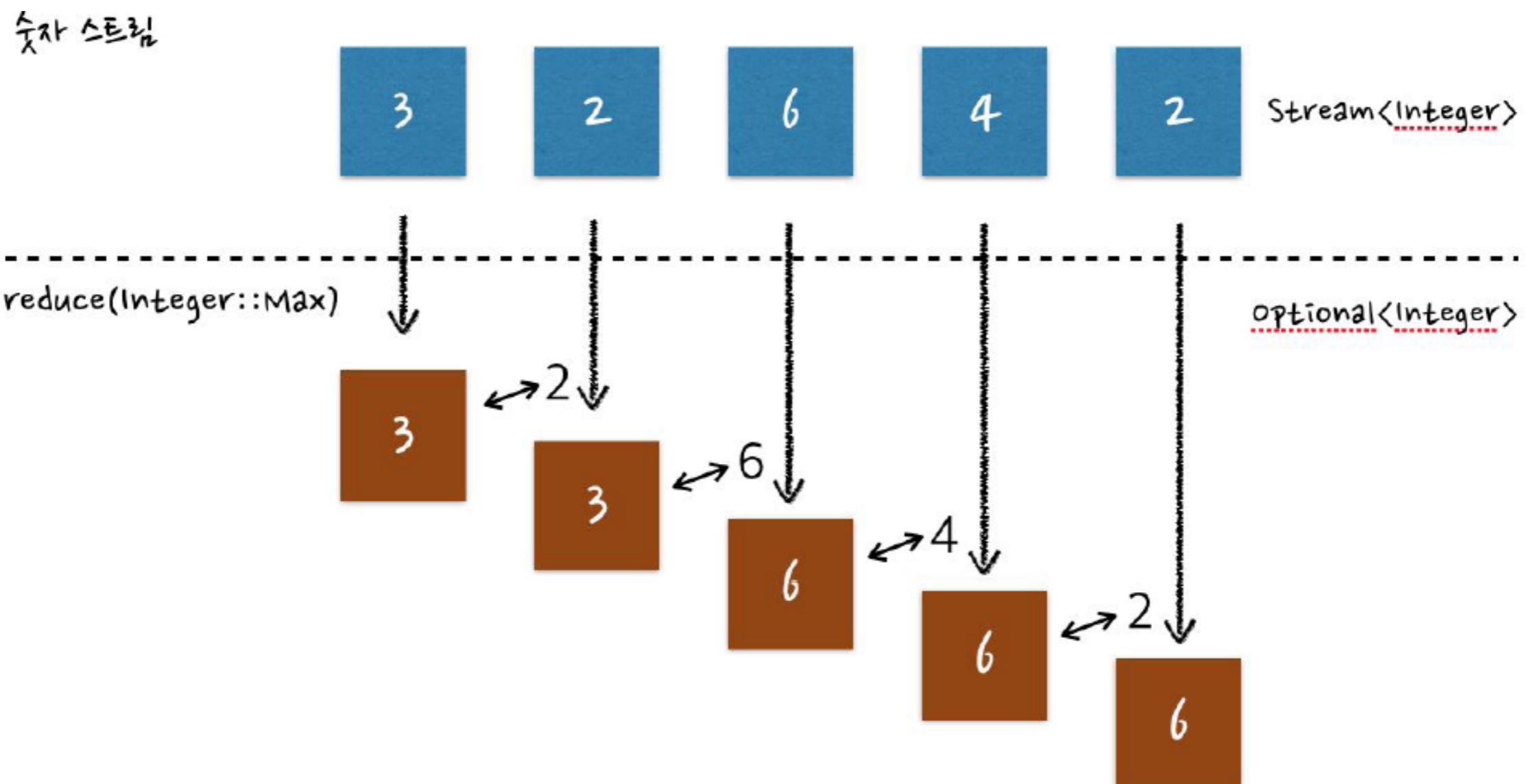
```
Arrays.asList(6,2,1,4,2)
    .stream().reduce(0,(a,b) -> a + b);
```



스트림 API의 활용 – 검색과 매칭

■ reduce [최대값, 최소값]

```
Arrays.asList(3,2,6,4,2)
    .stream().reduce(0, Integer::max);
```



스트림 API의 활용 – reduce 사용 예

```
// reduce(BinaryOperator<T> accumulator)
Optional<Integer> result = numbers
    .stream()
    .reduce((x, y) -> x > y ? x
: y);

// reduce(T identity, BinaryOperator<T>
accu)
Integer multi = numbers
    .stream()
    .reduce(1, (x, y) -> x * y);

// reduce(T identity, BiFunction<U, T, U> biFun,
BinaryOperator<U> accu) Double reduce = numbers
    .parallelStream()
    .reduce(0.0, (val1, val2) -> Double.valueOf(val1 + val2 / 10)
, (val1, val2) -> val1 + val2 );
```

Map, filter, reduce 요약

map, filter, and reduce
explained with emoji 😂

```
map([🐮, 🥔, 🐔, 🌽], cook)  
=> [🍔, 🍟, 🍗, 🍿]
```

```
filter([🍔, 🍟, 🍗, 🍿], isVegetarian)  
=> [🍟, 🍿]
```

```
reduce([🍔, 🍟, 🍗, 🍿], eat)  
=> 💩
```

기본형(Primitive Type) 특화 스트림

- 일반 스트림의 연산에서는 sum()과 같은 연산에 대한 메서드를 제공해 주지 않는다.

```
//The method sum() is undefined for the type Stream<Integer>
int caloriesSum = menu.stream()
    .map(Dish::getCalories)
    .sum();
```

- reduce를 사용하여 데이터를 연산할 수 있지만 직관적이지는 않음

```
int caloriesSum2 = menu.stream()
    .map(Dish::getCalories)
    .reduce(0, Integer::sum);
```

- 그리고 Autoboxing에 따른 성능 저하

```
Double Arr: 0.4030769547000005sec.
double arr: 0.0090505493sec.
Double Arr: 0.4325913248sec.
double arr: 0.0104117909sec.
```

기본형(Primitive Type) 특화 스트림

- 세가지 형태의 기본형 특화 스트림을 제공한다.
 - IntStream
 - LongStream
 - DoubleStream

```
int caloriesSum3 = menu.stream()
    .mapToInt(Dish::getCalories)
    .sum();

//숫자형일 경우 사용 가능한 메서드를 제공함
IntSummaryStatistics statics = IntStream.rangeClosed(1, 100)
    .summaryStatistics();
//{count=100, sum=5050, min=1, average=50.50000, max=100}
System.out.println(statics);

//필요에 따라 Wrapper 타입으로 변환가능
IntStream.rangeClosed(1, 100)
    .boxed()
    .reduce(0, Integer::sum);
```

Optional

OPTIONAL

NullPointerException에서 벗어나기

- NullPointerException이 발생
 - 예를 들어 아래와 같은 클래스들이 있을 때 ScreenResoultion 클래스의 getWidth()을 호출하고 싶다면?

```
public class Mobile {  
    private DisplayFeatures displayFeatures;  
    public DisplayFeatures getDisplayFeatures() {  
        return displayFeatures;  
    }  
}  
  
public class DisplayFeatures {  
    private ScreenResolution resolution;  
    public ScreenResolution getResolution() {  
        return resolution;  
    }  
}  
  
public class ScreenResolution {  
    private int width;  
    public int getWidth() {  
        return width;  
    }  
}
```

- `int width = mobile.getDisplayFeatures().getResolution().getWidth();`
=> Mobile나 DisplayFeatures 가 null 이면 곧바로 NullPointerException 이 발생합니다.

NullPointerException에서 벗어나기

- NullPointerException을 예방하기 위해서

```
public class MobileService {  
    public int getMobileScreenWidth(Mobile mobile){  
        if(mobile != null){  
            DisplayFeatures dfeatures = mobile.getDisplayFeatures();  
            if(dfeatures != null){  
                ScreenResolution resolution = dfeatures.getResolution();  
                if(resolution != null){  
                    return resolution.getWidth();  
                }  
            }  
        }  
        return 0;  
    }  
}
```

- NullPointerException은 방어가 되지만 코드가 길어지고 보기 좋지 않음

Optional 사용하기

- Optional은 값이 있거나 또는 없는 경우를 표현하기 위한 클래스

Java8의 Optional<T>는 하스켈과 스칼라에서 영감을 받아 만들어짐

```
public class Mobile {  
    private Optional<DisplayFeatures> displayFeatures;  
    public Optional<DisplayFeatures> getDisplayFeatures() {  
        return displayFeatures;  
    }  
}  
public class DisplayFeatures {  
    private Optional<ScreenResolution> resolution;  
    public Optional<ScreenResolution> getResolution() {  
        return resolution;  
    }  
}  
public class ScreenResolution {  
    private int width;  
    public int getWidth() {  
        return width;  
    }  
}
```

Optional 사용하기

- Optional은 값이 있거나 또는 없는 경우를 표현하기 위한 클래스

Java8의 Optional을 사용한 Null 체크 패턴

```
public class MobileService {  
  
    public Integer getMobileScreenWidth(Optional<Mobile> mobile){  
        return mobile.flatMap(Mobile::getDisplayFeatures)  
            .flatMap(DisplayFeatures::getResolution)  
            .map(ScreenResolution::getWidth)  
            .orElse(0);  
    }  
}
```

Optional 사용하기

■ Optional을 생성하는 방법

```
MobileService mService = new MobileService();

ScreenResolution resolution = new ScreenResolution(750,1334);
//Null을 허용하지 않는 Optional을 생성하는 방법 - Optional.of()
DisplayFeatures dfeatures = new DisplayFeatures("4.7", Optional.of(resolution));
Mobile mobile = new Mobile(2015001, "Apple", "iPhone 6s", Optional.of(dfeatures));

int width1 = mService.getMobileScreenWidth(Optional.of(mobile));
System.out.println("Apple iPhone 6s Screen Width = " + width1);

//ifPresent() 메서드
Optional<ScreenResolution> screen = dfeatures.getResolution();
screen.ifPresent(System.out :: println);

//비어 있는 Optional을 생성하는 방법 - Optional.empty()
Mobile mobile2 = new Mobile(2015001, "Apple", "iPhone 6s", Optional.empty());
int width2 = mService.getMobileScreenWidth(Optional.of(mobile2));
System.out.println("Apple iPhone 16s Screen Width = " + width2);
```

Optional 사용하기

- 비어있는 Optional을 생성하는 방법

```
Mobile mobile2 = new Mobile(2015001, "Apple", "iPhone 6s", Optional.empty());
```

- Null을 허용하지 않는 Optional을 생성하는 방법

```
DisplayFeatures dfeatures = new DisplayFeatures("4.7", Optional.of(resolution));
Mobile mobile = new Mobile(2015001, "Apple", "iPhone 6s", Optional.of(dfeatures));
```

- 객체가 null 아닐 때 특정 동작을 수행하고 싶으면?

- Optional의 ifPresent() 메서드 이용

```
Optional<ScreenResolution> screen = dfeatures.getResolution();
screen.ifPresent(System.out :: println);
```

- 객체가 null 일때 Default Value를 설정하고 싶으면?

- Optional의 orElse() 메서드 이용

```
mobile.flatMap(Mobile::getDisplayFeatures)
    .flatMap(DisplayFeatures::getResolution)
    .map(ScreenResolution::getWidth)
    .orElse(0);
```

요약

- filter, distinct, skip, limit 메서드로 스트림을 필터링하거나 자를 수 있다.
- map, flatMap 메서드로 스트림의 요소를 추출하거나 변환 할 수 있다.
- findFirst, findAny 메서드로 스트림의 요소를 검색할 수 있다.
- allMatch, noneMatch, anyMatch 메서드를 이용해서 주어진 Predicate와 일치하는 요소를 스트림에서 검색할 수 있다.
- reduce 메서드로 스트림의 모든 요소를 반복 조합하며 값을 도출할 수 있다.
- filter, map 등은 상태를 저장하지 않는 상태 없는 연산(stateless operation)이다.
- reduce, sorted, distinct 같은 연산은 값을 계산하는 데 필요한 상태를 저장하므로 상태 있는 연산(stateful operation)이라고 부른다.
- IntStream, DoubleStream, LongStream은 기본형 특화 스트림이다. 이들 연산은 각각 기본형에 맞게 특화 되어 있다.

1-04

스트림으로 데이터 리듀싱

Stream을 이용하여 코드 개선하기

- 요구사항
 - 각 트랜잭션을 통화별로 그룹화 하여 반환 하여라
(Map<Currency, List<Transaction> 반환)
 - 각 트랜잭션을 통화별로 그룹화 한 다음에 해당 통화의 모든 트랜잭션 합계를 계산하시오.
(Map<Currency, Double> 반환)
 - 각 트랜잭션을 통화별로 그룹화 한 뒤 각 트랜잭션이 5000 이상일 경우를 구분하여 리스트로 반환하시오.
(Map<Currency, Map<Boolean, List<Transaction>>> 반환)

Stream을 이용하여 코드 개선하기

```
public Transaction(Currency currency, double value) {  
    this.currency = currency;  
    this.value = value;  
}
```

```
Arrays.asList(new Transaction(Currency.EUR, 1500.0),  
             new Transaction(Currency.USD, 2300.0),  
             new Transaction(Currency.GBP, 9900.0),  
             new Transaction(Currency.EUR, 1100.0),  
             new Transaction(Currency.JPY, 7800.0),  
             new Transaction(Currency.CHF, 6700.0),  
             new Transaction(Currency.EUR, 5600.0),  
             new Transaction(Currency.USD, 4500.0),  
             new Transaction(Currency.CHF, 3400.0),  
             new Transaction(Currency.GBP, 3200.0),  
             new Transaction(Currency.USD, 4600.0),  
             new Transaction(Currency.JPY, 5700.0),  
             new Transaction(Currency.EUR, 6800.0));
```

Stream을 이용하여 코드 개선하기

- 클래식 자바 스타일, 컬렉션을 활용한 요구사항 구현
 - 각 트랜잭션을 Currency로 분류하여 반환하여라(Map<Currency, List<Transaction> 반환)

```
//그룹화할 트랜잭션을 저장할 맵을 생성한다.  
Map<Currency, List<Transaction>> transactionsByCurrencies = new HashMap<>();  
//트랜잭션 리스트를 반복한다  
for (Transaction transaction : transactions) {  
    //트랜잭션의 통화를 추출한다  
    Currency currency = transaction.getCurrency();  
    List<Transaction> transactionsForCurrency =  
        transactionsByCurrencies.get(currency);  
    //현재 통화를 그룹화하는 맵에 항목이 없으면 항목을 만든다.  
    if (transactionsForCurrency == null) {  
        transactionsForCurrency = new ArrayList<>();  
        transactionsByCurrencies.put(currency, transactionsForCurrency);  
    }  
    //같은 통화를 가진 트랜잭션 리스트에 현재 탐색중인 트랜잭션을 추가한다.  
    transactionsForCurrency.add(transaction);  
}
```

Stream을 이용하여 코드 개선하기

- Stream API를 활용하여 구현
 - 각 트랜잭션을 Currency로 분류하여 반환하여라(Map<Currency, List<Transaction> 반환)

```
Map<Currency, List<Transaction>> transactionsByCurrencies =  
    transactions.stream().collect(groupingBy(Transaction::getCurrency));
```

- 각 트랜잭션을 통화별로 그룹화 한 다음에 해당 통화의 모든 트랜잭션 합계를 계산하시오.
(Map<Currency, Double> 반환)

```
Map<Currency, Double> transactionByCurrencySumming =  
    transactions.stream()  
    .collect(groupingBy(Transaction::getCurrency,  
                        summingDouble(Transaction::getValue)));
```

Stream을 이용하여 코드 개선하기

- Stream API를 활용하여 구현

- 각 트랜잭션을 통화별로 그룹화 한 뒤 각 트랜잭션이 5000 이상일 경우를 구분하여 리스트로 반환하시오.
(Map<Currency, Map<Boolean, List<Transaction>>> 반환)

```
Map<Currency, Map<Boolean, List<Transaction>>>
transactionByCurrencyGreaterthen5000Value =
transactions.stream()
.collect(groupingBy(Transaction::getCurrency,
partitioningBy(tx -> tx.getValue() > 5000)));
```

데이터 수집하기 – collect() 메서드

transactions.stream()

.collect(Collectors.groupingBy(Transaction :: getCurrency));

java.util.stream

Interface Stream<T>

<R,A> R

collect(Collector<? super T,A,R> collector)

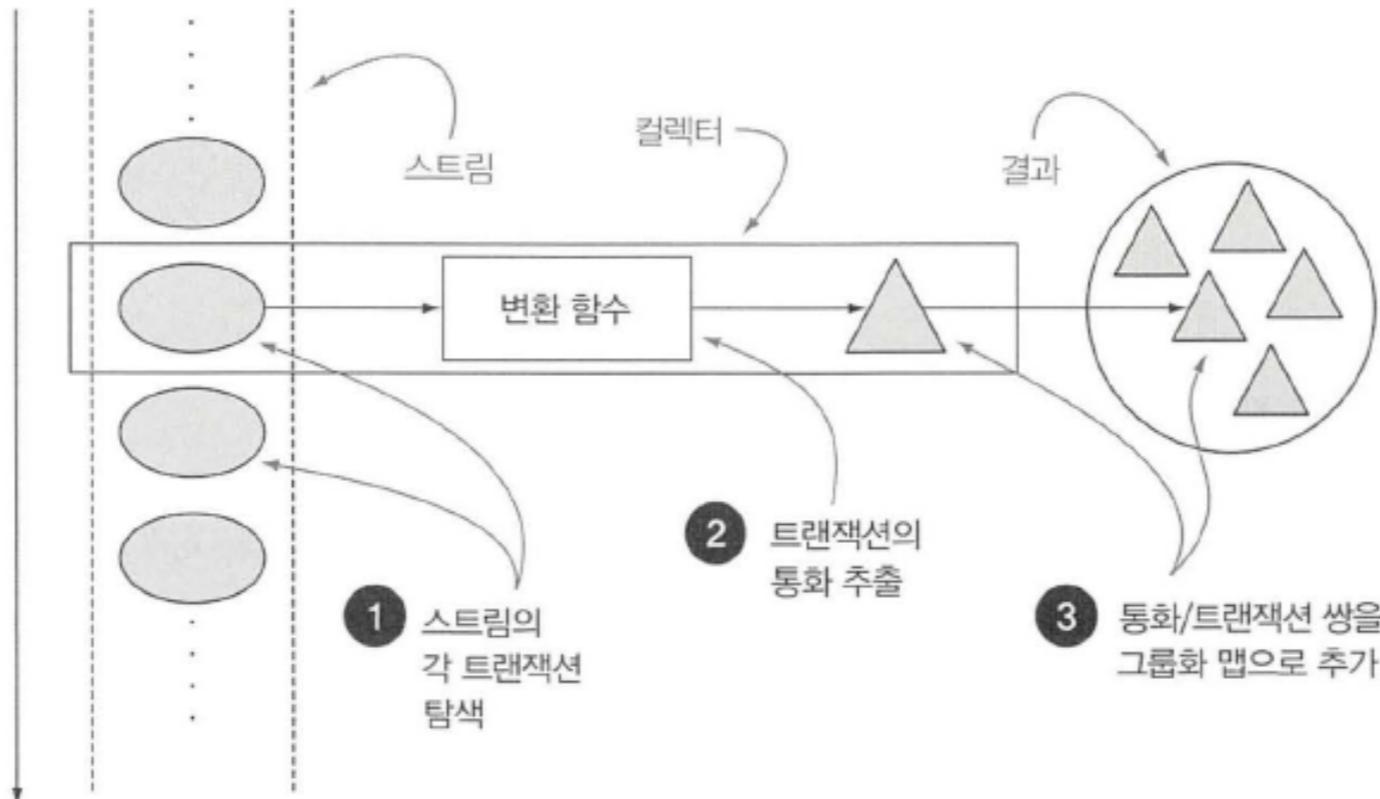
Performs a **mutable reduction** operation on the elements of this stream using a Collector.

- collect는 Stream의 최종 연산 메서드이며, 스트림의 결과를 수집한다.
- 인자로 Collector<T,A,R>의 구현체를 받는다.
- Collector에서 결과값에 대한 연산작업이 정의되어 있음.

데이터 수집하기 – collect() 메서드

- 통화별로 트랜잭션을 그룹화하는 리듀싱(Reducing) 연산

```
transactions.stream()  
    .collect(Collectors.groupingBy(Transaction::getCurrency));
```



- collect() 메서드를 호출하면 스트림의 요소에 리듀싱 연산이 수행된다.
- 컬렉션을 사용했던 코드에서 개발자가 직접 구현했던 작업이 스트림을 사용하면 내부적으로 자동으로 수행되어 진다.
- collect에서는 리듀싱 연산을 이용해서 스트림의 각 요소를 방문하면 컬렉터가 작업을 처리한다.

데이터 수집하기 – Collectors 클래스

■ 미리 정의된 컬렉터 Collectors 클래스

java.util.stream

Class Collectors

java.lang.Object

 java.util.stream.Collectors

```
public final class Collectors
extends Object
```

Implementations of Collector that implement various useful reduction operations, such as accumulating elements into collections, summarizing elements according to various criteria, etc.

■ Collector 인터페이스의 구현체를 제공하는 팩토리 메서드들의 집합 클래스이다.

- 스트림 요소를 하나의 값으로 리듀스 또는 요약하는 : reducing() / summarizingInt()
- 스트림의 요소 그룹화 : groupingBy()
- 스트림의 요소 분할 : partitioningBy()

데이터 수집하기 - Collectors 클래스의 활용

- 요구사항
 - 최소 칼로리, 최대 칼로리, 칼로리 합계, 칼로리 평균 구하기
 - 음식의 종류(Type)별로 그룹핑 하기

데이터 수집하기 – Collectors 클래스의 활용

- 요소의 최대값과 최소값 구하기

maxBy, minBy 메서드를 사용하여 최대값과 최소값을 찾을 수 있다.

```
Comparator<Dish> dishCaloriesComparator =  
    Comparator.comparingInt(Dish::getCalories);
```

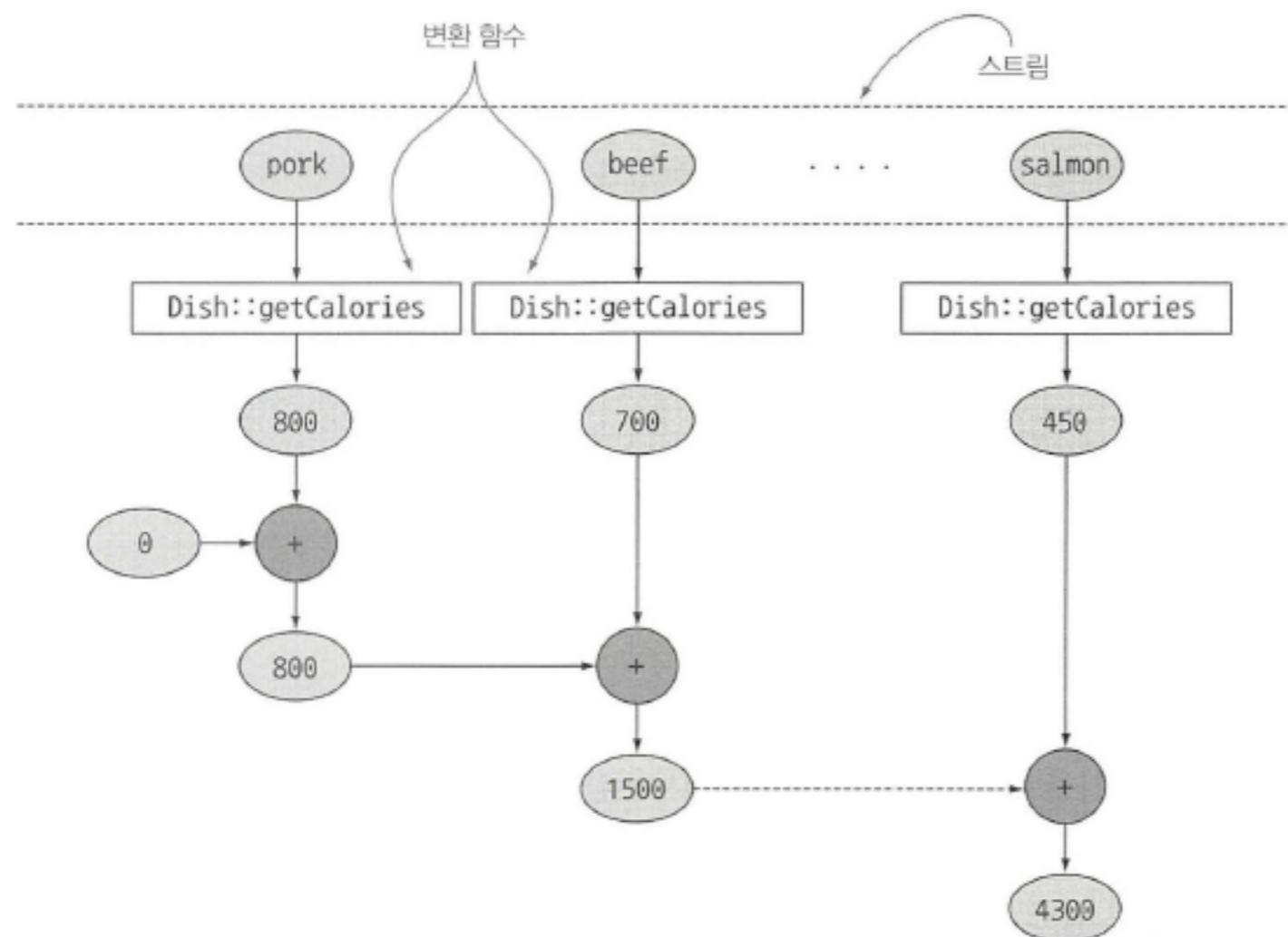
```
Dish mostCalorieDish = menu.stream()  
    .collect(Collectors.maxBy(dishCaloriesComparator))  
    .get();
```

```
Dish lowestCalorieDish = menu.stream()  
    .collect(Collectors.minBy(dishCaloriesComparator))  
    .get();
```

데이터 수집하기 – Collectors 클래스의 활용

- 요소의 합계 구하기
 - 요소의 합 : summingInt(), summingLong(), summingDouble()

```
int totalCalories = Dish.menu.stream()  
    .collect(Collectors.summingInt(Dish::getCalories));
```



데이터 수집하기 – Collectors 클래스의 활용

- 요소의 평균 구하기
 - 요소의 평균 : averagingInt(), averagingLong(), averagingDouble()

```
Double collect = Dish.menu.stream()  
    .collect(Collectors.averagingInt(Dish::getCalories));
```

데이터 수집하기 – Collectors 클래스의 활용

- **요약 연산**

`summarizingInt()`로 요소의 합계, 평균, 최소값, 최대값의 정보를 한꺼번에 가져올 수 있다.

```
// count=9, sum=4300, min=120, average=477.777778, max=800
IntSummaryStatistics menuStatistics = Dish.menu.stream().
collect(
    Collectors.summarizingInt(Dish::getCalories)
);
```

데이터 수집하기 – Collectors 클래스의 활용

- 문자열 연결

joining 메서드를 사용하여 각 문자열 요소를 합쳐 결과로 반환 할 수 있다.

```
//pork, beef, chicken ... pizza, prawns, salmon  
String shortMenu = Dish.menu.stream()  
    .map(Dish::getName)  
    .collect(Collectors.joining(/*구분자*/, ""));
```

데이터 수집하기 – Collectors 클래스의 활용

- 범용 리듀싱 요약 연산
 - reducing 팩토리 메서드를 사용하면 연산 내부 로직을 직접 정의할 수 있다.
=>reducing 메서드를 사용하여도 칼로리의 합계를 구할 수 있다.

```
int totalCalories = Dish.menu.stream().collect(  
    Collectors.reducing(  
        0/*연산의 시작값*/,  
        Dish::getCalories/*연산의 대상이 되는 값을 추출*/,  
        (i, j) -> i + j/*BinaryOperator로 두 값을 합침*/  
    );
```

Collectors.reducing()은 세 개의 인수를 받는다.

- 첫번째 인수는 리듀싱 연산의 시작 값 이거나 스트림에 인수가 없을 때의 반환 값이다.
- 두번째 인수는 연산의 대상이 되는 Dish의 칼로리 값이다.
- 세번째 인수는 같은 종류의 두 항목을 하나의 값으로 더하는 BinaryOperator이다.

데이터 수집하기 – Collectors 클래스의 활용

- 범용 리듀싱 요약 연산

reducing 팩토리 메서드를 사용하면 연산 내부 로직을 직접 정의할 수 있다.

=>reducing 메서드를 사용 하여도 칼로리의 최대값을 구할 수 있다.

```
Optional<Dish> maxCalorieDish = Dish.menu.stream().collect(  
    Collectors.reducing(  
        /*BinaryOperator 연산을 진행*/  
        (d1, d2) -> d1.getCalories() > d2.getCalories() ? d1 : d2  
    )  
);
```

데이터 수집하기 – Collectors 클래스의 활용

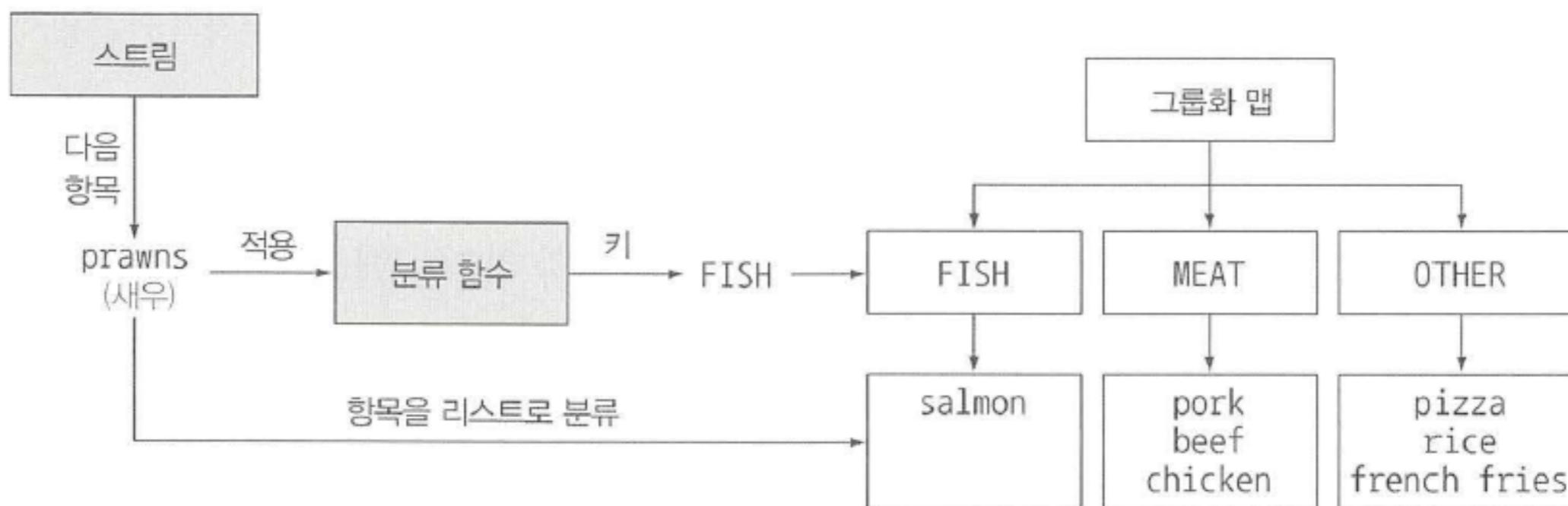
■ 그룹핑 하기 1

groupingBy 메서드는 각 요소들을 조건에 따라 그룹핑 한 결과값을 만들어 낼 수 있다.

=> 1. 요리의 종류별로 그룹핑 하기

```
Map<Dish.Type, List<Dish>> dishesByType = Dish.menu.stream()  
    .collect(groupingBy(Dish::getType));
```

```
{FISH=[prawns, salmon], OTHER=[french fries, rice, season fruit, pizza],  
MEAT=[pork, beef, chicken]}
```



데이터 수집하기 – Collectors 클래스의 활용

■ 그룹핑 하기 1

groupingBy 메서드는 각 요소들을 조건에 따라 그룹핑 한 결과값을 만들어 낼 수 있다.

=>2. 요리의 칼로별로 그룹핑 하기

```
enum CaloricLevel { DIET, NORMAL, FAT };

private static Map<CaloricLevel, List<Dish>> groupDishesByCaloricLevel() {
    return menu.stream().collect(
        groupingBy(dish -> {
            if (dish.getCalories() <= 400)
                return CaloricLevel.DIET;
            else if (dish.getCalories() <= 700)
                return CaloricLevel.NORMAL;
            else
                return CaloricLevel.FAT;
        }));
}
```

{ DIET=[chicken, rice, season fruit, prawns],
FAT=[pork],
NORMAL=[beef, french fries, pizza, salmon] }

데이터 수집하기 – Collectors 클래스의 활용

■ 그룹핑 하기 2 (다수준 그룹화)

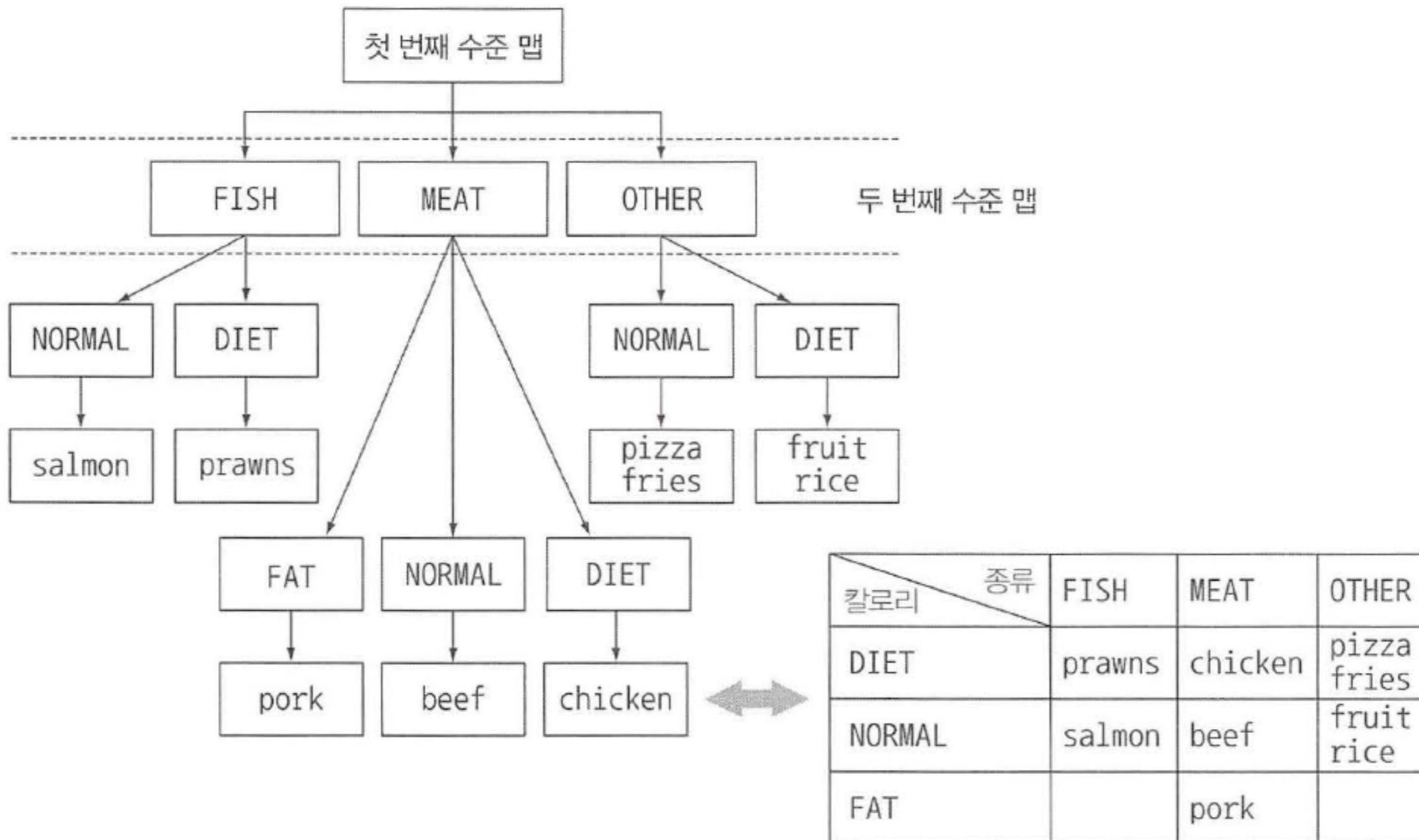
두 개의 인자를 받는 `Collectors.groupingBy()` 를 이용하여 다수준으로 그룹화 할 수 있다.

```
Map<Dish.Type, Map<CaloricLevel, List<Dish>>> groupDishedByTypeAndCaloricLevel() {  
    return menu.stream().collect(  
        groupingBy(Dish::getType,  
                   groupingBy((Dish dish) -> {  
                       if (dish.getCalories() <= 400)  
                           return CaloricLevel.DIET;  
                       else if (dish.getCalories() <= 700)  
                           return CaloricLevel.NORMAL;  
                       else  
                           return CaloricLevel.FAT;  
                   } )  
    );  
}
```

```
{MEAT={DIET=[chicken], NORMAL=[beef], FAT=[pork]},  
 FISH={DIET=[prawns], NORMAL=[salmon]},  
 OTHER={DIET=[rice, seasonal fruit], NORMAL=[french fries, pizza]}}
```

데이터 수집하기 – Collectors 클래스의 활용

■ 그룹핑 하기 2 (다수준 그룹화)



데이터 수집하기 – Collectors 클래스의 활용

■ 그룹핑 하기 3 (서브 그룹으로 데이터 수집)

groupingBy 메서드의 두번째 인자로 Collector를 사용할 수 있다.

해당 인자를 사용하면 추가적인 그룹화나 개수를 세는 등의 작업이 가능하다.

```
// {OTHER=4, MEAT=3, FISH=2}
Dish.menu.stream()
    .collect(groupingBy(
        Dish::getType, Collectors.counting()
    ));

// OTHER={true=[french fries, rice, season fruit, pizza]}
// MEAT={false=[pork, beef, chicken]}
// FISH={false=[prawns, salmon]}
Dish.menu.stream()
    .collect(groupingBy(
        Dish::getType,
        groupingBy(Dish::isVegetarian)
    ));
```

데이터 수집하기 – Collectors 클래스의 활용

■ 그룹핑 하기 4

collectingAndThen 함수를 사용할 경우 결과값을 재 가공 할 수 있다.

=>각 서브그룹에서 가장 칼로리가 높은 요리 찾기

```
Map<Dish.Type, Dish> maxDishes2 = Dish.menu.stream().collect(groupingBy(  
    Dish::getType,  
    Collectors.collectingAndThen(  
        Collectors.maxBy(Comparator.comparingInt(Dish::getCalories)),  
        Optional::get  
    )  
));  
                                {FISH=salmon, OTHER=pizza, MEAT=pork}
```

- groupingBy는 가장 바깥쪽에 위치하면서 요리의 종류에 따라 메뉴 스트림을 세 개의 서브스트림으로 그룹화한다.
- groupingBy 컬렉터는 collectingAndThen 컬렉터를 감싼다. 두 번째 컬렉터는 그룹화된 세 개의 서브스트림에 적용된다.
- collectingAndThen 컬렉터는 세번째 컬렉터 maxBy를 감싼다.
- 리듀싱 컬렉터가 서브스트림에 연산을 수행할 결과에 collectingAndThen의 Optional::get 변환 함수가 적용된다.
- groupingBy 컬렉터가 반환하는 맵의 분류키에 대응하는 세 값이 각각의 요리 형식에서 가장 높은 칼로리이다.

데이터 수집하기 – Collectors 클래스의 활용

■ 분할 1

분할 함수(partitioning function)라 불리는 Predicate를 분류함수로 사용하는 특수한 그룹화 기능이다. 그룹화 맵은 참 아니면 거짓 값을 갖는 두 개의 그룹으로 분류된다.
=>모든 요리를 채식요리와 채식요리가 아닌 요리로 분류하기

```
// {false=[pork, beef, chicken, prawns, salmon],  
// true=[french fries, rice, season fruit, pizza]}  
Map<Boolean, List<Dish>> partitionedMenu = Dish.menu.stream()  
    .collect( Collectors.partitioningBy(Dish::isVegetarian) );
```

데이터 수집하기 – Collectors 클래스의 활용

■ 분할 2

두 번째 인수로 전달 할 수 있는 오버로드된 버전의 `partitioningBy` 메서드

=>모든 요리를 채식요리와 채식요리가 아닌 요리로 그룹화 해서 다시 요리 종류별로 분류하기

```
// {false=[FISH=[prawns, salmon], MEAT=[pork, beef, chicken]],  
// true=[OTHER=[french fries, rice, season fruit, pizza]]}  
Map<Boolean, Map<Type, List<Dish>>> partAndGroupDishes = menu.  
    stream().  
    collect(Collectors.partitioningBy(  
        Dish::isVegetarian,  
        Collectors.groupingBy(Dish::getType)  
    ));
```

데이터 수집하기 – Collectors 클래스의 정적 팩토리 메서드 정리

| 팩토리 메서드 | 반환 형식 | 사용 예제 |
|----------------|----------------------|---|
| toList | List<T> | 스트림의 모든 항목을 리스트로 수집. 활용 예: List<Dish> dishes = menuStream.collect(toList()); |
| toSet | Set<T> | 스트림의 모든 항목을 중복이 없는 집합으로 수집. 활용 예: Set<Dish> dishes = menuStream.collect(toSet()); |
| toCollection | Collection<T> | 스트림의 모든 항목을 공급자가 제공하는 컬렉션으로 수집. 활용 예: Collection<Dish> dishes = menuStream.collect(toCollection(), ArrayList::new); |
| counting | Long | 스트림의 항목 수 계산. 활용 예: long howManyDishes = menuStream.collect(counting()); |
| summingInt | Integer | 스트림의 항목에서 정수 프로퍼티값을 더함. 활용 예: int totalCalories = menuStream.collect(summingInt(Dish::getCalories)); |
| averagingInt | Double | 스트림 항목의 정수 프로퍼티의 평균값 계산. 활용 예: double avgCalories = menuStream.collect(averagingInt(Dish::getCalories)); |
| summarizingInt | IntSummaryStatistics | 스트림 내의 항목의 최댓값, 최솟값, 합계, 평균 등의 정수 정보 통계를 수집. 활용 예: IntSummaryStatistics menuStatistics = menuStream.collect(summarizingInt(Dish::getCalories)); |
| joining | String | 스트림의 각 항목에 toString 메서드를 호출한 결과 문자열을 연결. 활용 예: String shortMenu = menuStream.map(Dish::getName).collect(joining(", ")); |

데이터 수집하기 – Collectors 클래스의 정적 팩토리 메서드 정리

| 팩토리 메서드 | 반환 형식 | 사용 예제 |
|-------------------|-----------------|---|
| maxBy | Optional<T> | 주어진 비교자를 이용해서 스트림의 최댓값 요소를 Optional로 감싼 값을 반환. 스트림에 요소가 없을 때는 Optional.empty()를 반환. 활용 예: Optional<Dish> fattest = menuStream.collect(maxBy(comparingInt(Dish::getCalories))); |
| minBy | Optional<T> | 주어진 비교자를 이용해서 스트림의 최솟값 요소를 Optional로 감싼 값을 반환. 스트림에 요소가 없을 때는 Optional.empty()를 반환. 활용 예: Optional<Dish> lightest = menuStream.collect(minBy(comparingInt(Dish::getCalories))); |
| reducing | 리듀싱 연산에서 형식을 결정 | 누적자를 초기값으로 설정한 다음에 BinaryOperator로 스트림의 각 요소를 반복적으로 누적자와 합쳐 스트림을 하나의 값으로 리듀싱. 활용 예: int totalCalories = menuStream.collect(reducing(0, Dish::getCalories, Integer::sum)); |
| collectingAndThen | 변환 함수가 형식을 반환 | 다른 컬렉터를 감싸고 그 결과에 변환 함수를 적용. 활용 예: int howManyDishes = menuStream.collect(collectingAndThen(toList(), List::size)); |
| groupingBy | Map<K, List<T>> | 하나의 프로퍼티값을 기준으로 스트림의 항목을 그룹화하며 기준 프로퍼티값을 결과 맵의 키로 사용. 활용 예: Map<Dish.Type, List<Dish>> dishesByType = menuStream.collect(groupingBy(Dish::getType)); |

데이터 수집하기 – Collectors 클래스의 정적 팩토리 메서드 정리

| 팩토리 메서드 | 반환 형식 | 사용 예제 |
|----------------|-----------------------|------------------------------------|
| partitioningBy | Map<Boolean, List<T>> | 프레디케이트를 스트림의 각 항목에 적용한 결과로 항목을 분할. |

활용 예: Map<Boolean, List<Dish>> vegetarianDishes =
menuStream.collect(partitioningBy(Dish::isVegetarian));

요약

- collect는 스트림의 요소를 요약 결과로 누적하는 Collector를 인수로 갖는 최종연산
- 스트림의 요소를 하나의 값으로 reduce 하고 요약하는 Collector 뿐 아니라 최대값, 최소값, 평균값을 계산하는 Collector 등이 미리 정의되어 있다.
- 미리 정의된 groupingBy로 스트림의 요소를 그룹화 할 수 있다.
- 미리 정의된 partitioningBy로 스트림의 요소를 분할 할 수 있다.
- Collector는 다수준의 그룹화, 분할, 리듀싱 연산에 적합하게 설계되어 있다.

1-05

날짜와 시간 API

기존 날짜와 시간 API

- 쓰레드에서 안전하지 않음
- 사용하기 불편 - Date, Calendar
- 일관성이 떨어짐 - 애매한 unix time 추상화
- 불규칙한 성능
- 헬로월드 “Java의 날짜와 시간 API” 참고
 - <http://helloworld.naver.com/helloworld/645609>

오픈 소스 *joda-time*은
사실상 표준으로 사용



Stephen Colebourne

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

새로운 API의 목표

- 일관된 API
- Immutable
 - java.time 패키지의 객체는 모두 불변이다.
- 쓰레드에서 안전, 직관적이고 사용하기 편한 API
 - LocalDate.now().with(TemporalAdjusters.lastDayOfMonth()).minusDays(2)
- 관련 표준 준수:
 - ISO-8601
 - CLDR(Unicode Common Locale Data Repository)
 - TZDB(Time-Zone Database)
- UTC와 연계된 명시적 시간 척도
- joda-time 참조



**java.time
package!**

java.time package

- 날짜와 시간을 나타내는 여러가지 API가 새롭게 추가됨

| 패키지 | 설명 |
|--------------------|--|
| java.time | 날짜와 시간을 나타내는 핵심 API 인 LocalDate, LocalTime, LocalDateTime, ZonedDateTime 을 포함하고 있다. 이들 클래스는 ISO-8601 에 정의된 달력 시스템에 기초한다. |
| java.time.chrono | ISO-8601 에 정의된 달력 시스템 이외에 다른 달력 시스템을 사용코자할때 사용할 수 있는 API 들이 포함되어 있다. |
| java.time.format | 날짜와 시간을 파싱하고 포맷팅하는 API 들이 포함되어 있다. |
| java.time.temporal | 날짜와 시간을 연산하기 위한 보조 API 들이 포함되어 있다. |
| java.time.zone | 타임존을 지원하는 API 들이 포함되어 있다. |

- java.time 패키지에는 날짜와 시간을 표현하는 5개의 클래스가 있다.

| 클래스명 | 설명 |
|---------------|---------------------------------------|
| LocalDate | 로컬 날짜 클래스 |
| LocalTime | 로컬 시간 클래스 |
| LocalDateTime | 로컬 날짜 및 시간 클래스(LocalDate + LocalTime) |
| ZonedDateTime | 특정 타임존(TimeZone)의 날짜와 시간 클래스 |
| Instant | 특정 시점의 Time-Stamp 클래스 |

시간과 날짜 다루기 주요 내용

1. 날짜 / 시간 인스턴스 생성
2. Period로 날짜 간 차이 구하기
3. Duration으로 시간 차이 구하기
4. 에포크 타임 다루기
5. 날짜 / 시간 인스턴스 다루기
6. 날짜 / 시간 비교하기
7. 날짜 / 시간 포매팅 하기

날짜 / 시간 인스턴스 생성

- `java.time.LocalDate`: 특정 지역의 날짜, 시간대 정보 제외
 - `java.time.LocalTime`: 특정일의 시간, 시, 분, 초, 나노(10-9)초
 - `java.time.LocalDateTime`으로 날짜와 시간 결합 가능
 - 다양한 날짜 연산 메서드 제공
-
- 생성
 - 현재 : `LocalDateTime.now()`
 - 특정일 : `LocalDate.of(2014, Month.JULY, 26)`
 - unix 시간 : `LocalDate.ofEpochDay(1406321656)`
 - 시간 : `LocalTime.of(10, 30, 50);`

LocalDate 주요 메서드

- plusDays, plusWeeks, plusMonths, plusYears : LocalDate 에 일, 주, 월, 연도를 더한다.
- minusDays, minusWeeks, minusMonths, minusYears : LocalDate 에 일, 주, 월, 연도를 뺀다.
- plus, minus : Duration 또는 Period 를 더하거나 뺀다.
- getDayOfMonth
- getDayOfYear
- getDayOfWeek
- getMonth
- getMonthValue
- getYear
- until
- isBefore, isAfter
- isLeapYear

| 클래스 | 리턴타입 | 메소드(매개변수) | 설명 |
|-----------|-----------|-----------------|-----------|
| LocalDate | int | getYear() | 년 |
| | Month | getMonth() | Month 열거값 |
| | int | getMonthValue() | 월 |
| | int | getDayOfYear | 일년의 몇번째 일 |
| | int | getDayOfMonth() | 월의 몇번째 일 |
| | DayOfWeek | getDayOfWeek() | 요일 |
| | boolean | isLeapYear() | 윤년 여부 |

LocalTime 주요 메서드

- plusHours, plusMinutes, plusSeconds, plusNanos :
 - LocalTime 에 시, 분, 초, 나노초를 더한다.
- minusHours, minusMinutes, minusSeconds, minusNanos :
 - LocalTime 에 시, 분, 초, 나노초를 뺀다.
- plus, minus : Duration 을 더하거나 뺀다.
- getHour, getMinute, getSecond, getNano
- isBefore, isAfter

| 클래스 | 리턴타입 | 메소드(매개변수) | 설명 |
|-----------|------|-------------|--------|
| LocalTime | int | getHour() | 시간 |
| | int | getMinute() | 분 |
| | int | getSecond() | 초 |
| | int | getNano() | 나노초 리턴 |

LocalDateTime, ZonedDateTime

- `LocalDateTime`의 `toLocalDate()` 메서드를 사용하면 `LocalDate` 객체로 변환할 수 있다.
- `ZonedDateTime`에서 제공하는 추가 메서드

| 클래스 | 리턴타입 | 메소드(매개변수) | 설명 |
|----------------------------|-------------------------|--------------------------|--|
| <code>ZonedDateTime</code> | <code>ZoneId</code> | <code>getZone()</code> | 존아이디를 리턴 (예: <code>Asia/Seoul</code>) |
| | <code>ZoneOffset</code> | <code>getOffset()</code> | 존오프셋(시차)을 리턴 |

Instant : 절대시간

- 정확히 하루는 86,400초 - 윤초가 없다.
- Instant 는 타임라인의 한 점이다.

- 예제 : 알고리즘 실행 시간 측정

```
Instant start = Instant.now();  
runAlgorithm();  
Instant end = Instant.now();  
Duration timeElapsed = Duration.between(start, end);  
long millis = timeElapsed.toMillis();  
System.out.printf("%d milliseconds\n", millis);
```

날짜와 시간을 변경하기

■ 변경하기

- 날짜와 시간을 변경할 수 있는 `with()` 메서드가 제공된다.

| | |
|----------------------------|---|
| <code>LocalDateTime</code> | <code>with(TemporalAdjuster adjuster)</code> Returns an adjusted copy of this date-time. |
|----------------------------|---|

- `TemporalAdjuster` 객체는 아래 표에 있는 `TemporalAdjusters`의 static method를 통해 얻을 수 있음

| 메소드(매개변수) | 설명 |
|--|----------------|
| <code>firstDayOfYear()</code> | 이번 해의 첫일 |
| <code>lastDayOfYear()</code> | 이번 해의 마지막 일 |
| <code>firstDayOfNextYear()</code> | 다음 해의 첫일 |
| <code>firstDayOfMonth()</code> | 이번 달의 첫일 |
| <code>lastDayOfMonth()</code> | 이번 달의 마지막 일 |
| <code>firstDayOfNextMonth()</code> | 다음 달의 첫일 |
| <code>firstInMonth(DayOfWeek dayOfWeek)</code> | 이번 달의 첫 요일 |
| <code>lastInMonth(DayOfWeek dayOfWeek)</code> | 이번 달의 마지막 요일 |
| <code>next(DayOfWeek dayOfWeek)</code> | 돌아오는 요일 |
| <code>nextOrSame(DayOfWeek dayOfWeek)</code> | 돌아오는 요일(오늘 포함) |
| <code>previous(DayOfWeek dayOfWeek)c</code> | 지난 요일 |
| <code>previousOrSame(DayOfWeek dayOfWeek)</code> | 지난 요일(오늘 포함) |

날짜와 시간 비교하기

- 날짜와 시간 클래스들은 다음과 같이 비교하거나 차이를 구하는 메서드 들이 제공됨

| 클래스 | 리턴타입 | 메소드(매개변수) | 설명 |
|---|----------|--|-----------|
| LocalDate LocalDateTime | boolean | isAfter(ChronoLocalDate other) | 이후 날짜인지? |
| | | isBefore(ChronoLocalDate other) | 이전 날짜인지? |
| | | isEqual(ChronoLocalDate other) | 동일 날짜인지? |
| LocalTime LocalDateTime | boolean | isAfter(LocalTime other) | 이후 시간인지? |
| | | isBefore(LocalTime other) | 이전 시간인지? |
| LocalDate | Period | until(ChronoLocalDate endDateExclusive) | 날짜 차이 |
| LocalDate LocalTime LocalDateTime | long | until(Temporal endExclusive, TemporalUnit unit) | 시간 차이 |
| Period | Period | between(LocalDate startDateInclusive, LocalDate endDateExclusive) | 날짜 차이 |
| Duration | Duration | between(Temporal startInclusive, Temporal endExclusive) | 시간 차이 |
| ChronoUnit.YEARS | long | between(Temporal temporal1Inclusive, Temporal temporal2Exclusive) | 전체 년 차이 |
| ChronoUnit.MONTHS | | | 전체 달 차이 |
| ChronoUnit.WEEKS | | | 전체 주 차이 |
| ChronoUnit.DAYS | | | 전체 일 차이 |
| ChronoUnit.HOURS | | | 전체 시간 차이 |
| ChronoUnit.SECONDS | | | 전체 초 차이 |
| ChronoUnit.MILLISECONDS | | | 전체 밀리초 차이 |
| ChronoUnit.NANOS | | | 전체 나노초 차이 |

Period와 Duration의 차이점

■ Period와 Duration

- Period : 년, 달, 일의 양을 나타내는 날짜 기준 클래스
- Duration : 시, 분, 초, 나노초의 양을 나타내는 시간 기준 클래스

| 클래스 | 리턴타입 | 메소드(매개변수) | 설명 |
|----------|------|--------------|---------|
| Period | int | getYears() | 년의 차이 |
| | int | getMonths() | 달의 차이 |
| | int | getDays() | 일의 차이 |
| Duration | int | getSeconds() | 초의 차이 |
| | int | getNano() | 나노초의 차이 |

■ between() 메서드의 차이점

- Period와 Duration의 between()
- ChronoUnit의 between()
- 년, 달, 일, 초의 단순 차이를 리턴
- 전체 시간을 기준으로 차이를 리턴

서식 지정과 파싱 : DateTimeFormatter

Predefined formatters:

- ISO_DATE (2015-11-05)
- ISO_TIME (11:25:47.624)
- RFC_1123_DATE_TIME(Thu, 5 Nov 2015 11:27:22 +0530)
- ISO_ZONED_DATE_TIME (2015-11-05T11:30:33.49+05:30[Asia/Kolkata])

```
LocalTime wakeupTime = LocalTime.of(6, 0, 0);
System.out.println(
    "Wakeup time: " + DateTimeFormatter.ISO_TIME.format(wakeupTime));
DateTimeFormatter customFormat =
    DateTimeFormatter.ofPattern("dd MMM yyyy");
System.out.println(
    customFormat.format(LocalDate.of(2016, Month.JANUARY, 01)));
```

날짜 포매팅

- G (era: BC, AD)
- y (year of era: 2015, 15)
- Y (week-based year: 2015, 15)
- M (month: 11, Nov, November)
- w (week in year: 13)
- W (week in month: 2)
- E (day name in week: Sun, Sunday)
- D (day of year: 256)
- d (day of month: 13)

시간 포맷팅

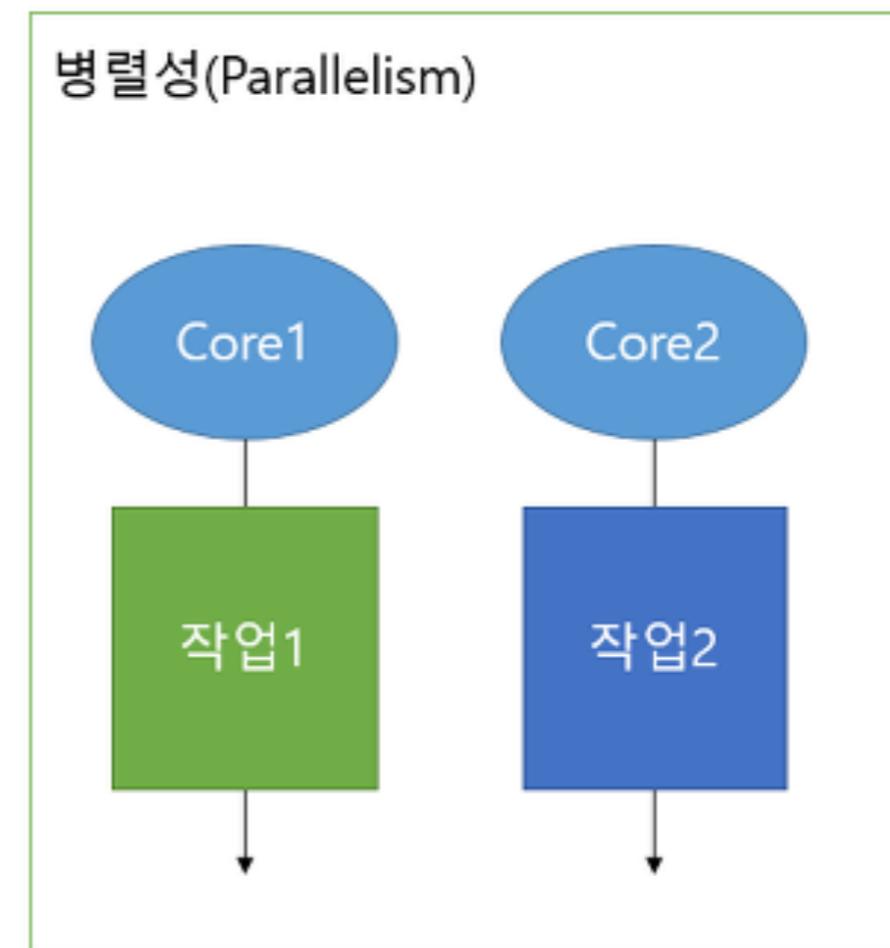
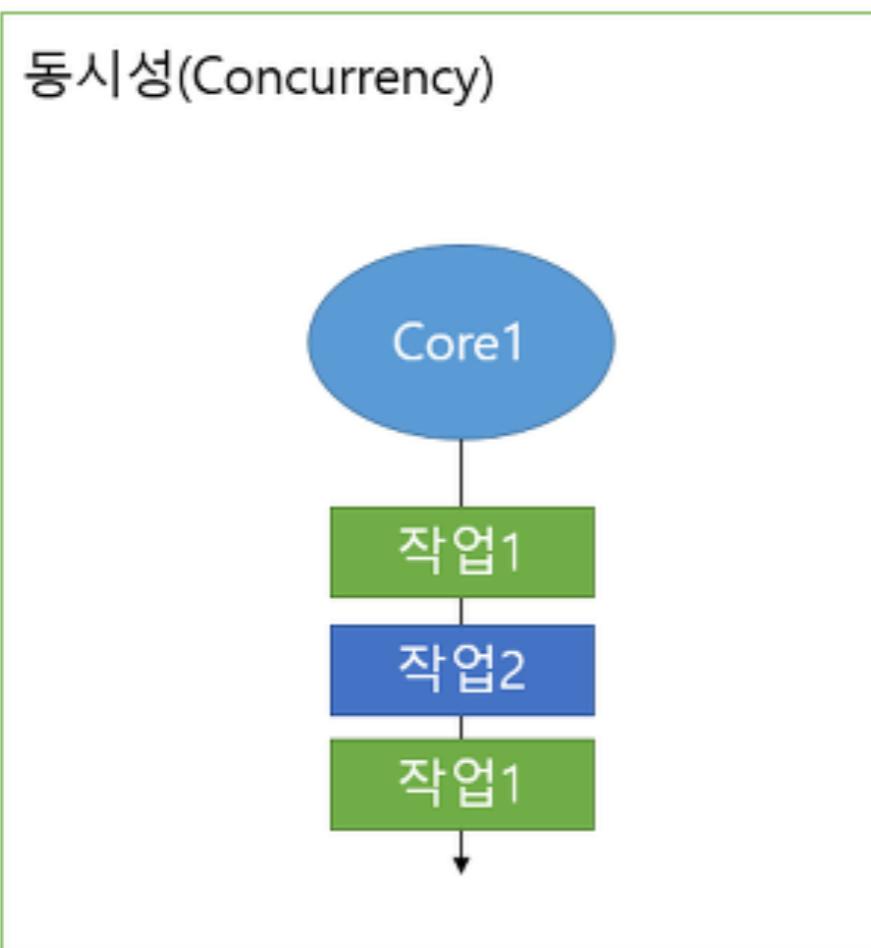
- a (marker for the text a.m./p.m. marker)
- H (hour: value range 0-23)
- k (hour: value range 1-24)
- K (hour in a.m./p.m.: value range 0-11)
- h (hour in a.m./p.m.: value range 1-12)
- m (minute)
- s (second)
- S (fraction of a second)
- z (time zone: general time-zone format)

1-06

병렬 데이터 처리와 성능

■ 동시성(Concurrency)과 병렬성(Parallelism)

- 이 둘은 멀티 스레드의 동작 방식이라는 점에서는 동일 하지만 서로 다른 목적을 가지고 있다.
- 동시성은 멀티 작업을 위해 멀티 스레드가 번갈아 가면서 실행하는 성질을 말하고
- 병렬성은 멀티 작업을 위해 멀티 코어를 이용해서 동시에 실행하는 성질을 말합니다.
- 싱글 코어 CPU를 이용한 멀티 작업은 병렬적으로 실행 되는 것처럼 보이지만, 사실은 번갈아 가면 실행 하는 동시성 작업입니다.



병렬 처리

■ 데이터 병렬성(Data Parallelism)

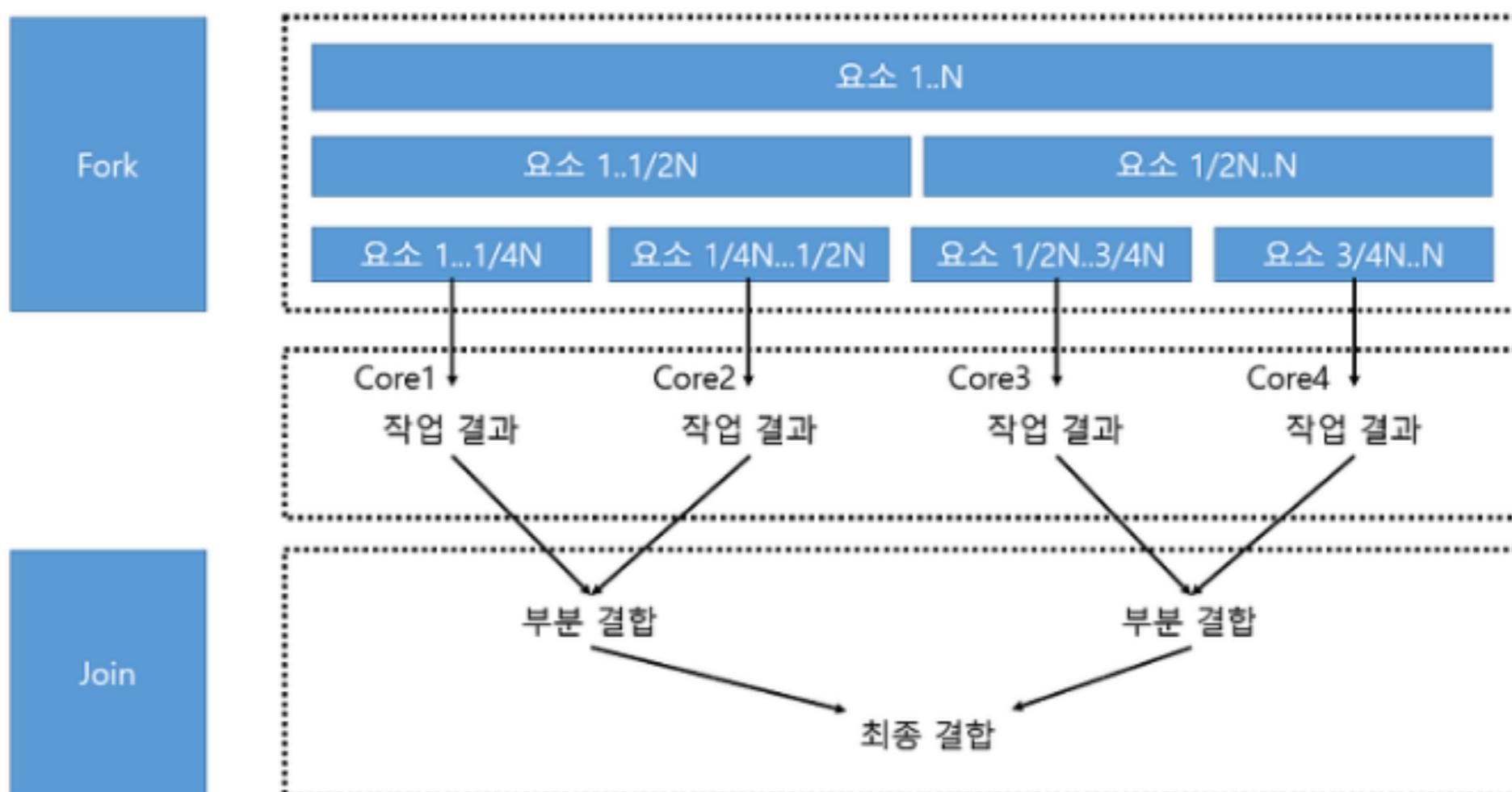
- 데이터 병렬성은 전체 데이터를 쪼개어 서브 데이터들로 만들고, 이 서브 데이터들을 병렬 처리 해서 작업을 빨리 끝내는 것을 말합니다.
- 자바 8에서 지원하는 병렬 스트림은 데이터 병렬성을 구현 한 것입니다.
- 멀티 코어의 수 만큼 대용량 요소를 서브 요소들로 나누고, 각각의 서브 요소들을 분리된 스레드에서 병렬 처리 시킵니다.
- 예를 들어 (Quad Core) CPU일 경우 4개의 서브 요소들로 나누고, 4개의 스레드가 각각의 서브 요소들을 병렬 처리합니다.

■ 작업 병렬성(Task Parallelism)

- 작업 병렬성은 서로 다른 작업을 병렬 처리하는 것을 말합니다. 작업 병렬성의 대표적인 예는 웹 서버입니다.
- 웹서버는 각각의 브라우저에서 요청한 내용을 개별 스레드에서 병렬로 처리합니다.

포크조인(ForkJoin) 프레임워크

- 병렬 스트림은 요소들을 병렬 처리하기 위해 포크조인(ForkJoin) 프레임워크 사용함
 - 병렬 스트림을 이용하면 런타임 시에 포크조인 프레임워크가 동작하는데, **포크 단계에서는 전체 데이터를 서브 데이터로 분리**
 - 조인 단계에서는 서브 결과를 결합해서 최종 결과를 생성
 - 예를 들어 쿼드 코어 CPU에서 병렬 스트림으로 작업을 처리한 경우, 스트림의 요소를 N개라고 보았을 때 포크 단계에서는 전체 요소를 4등분
 - 그리고 1등분씩 개별 코어에서 처리하고 조인 단계에서는 3번의 결합과정을 거쳐 최종결과를 산출



병렬 스트림 생성

■ 병렬 스트림 생성 메서드

- 병렬 처리를 위해 코드에서 포크조인 프레임워크를 직접 사용할 수는 있지만, 병렬 스트림을 이용할 경우에는 백그라운드에서 포크조인 프레임워크가 사용되기 때문에 매우 쉽게 병렬 처리 할 수 있음
- `parallelStream()` 메서드는 컬렉션으로부터 병렬 처리 스트림을 바로 리턴
- `parallel()` 메서드는 순차 처리 스트림을 병렬 스트림으로 변환해서 리턴
- 내부적으로 전체요소를 서브 요소들로 나누고, 이 서브 요소들을 개별 스레드가 처리
- 서브 처리 결과가 나오면 결합해서 마지막 최종 처리 결과를 리턴

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

병렬 스트림 사용하기

- 멀티 쓰레드로 병렬 실행 가능

```
Stream.iterate(1L, i -> i + 1)
    .parallel()
    .limit(n)
    .reduce(0L, Long::sum);
```

parallel을 stream에 추가만 하면
연산을 병렬로 수행할 수 있다.

병렬 스트림 사용하기

■ 병렬 스트림 처리과정

1. 스레드를 만들고 초기화한다.
2. 데이터를 여러 개의 Chunk로 분리한다.
3. 각 스레드에 Chunk를 할당하고 계산한다.
4. 각 스레드의 결과를 하나로 합병한다.

병렬 스트림 사용하기

- 1부터 10,000,000까지 값을 더하는 아래의 로직은 오히려 고전적인 for-loop 방식이 훨씬 빠름

```
// 10ms 소요
public static long iterativeSum(long n) {
    long result = 0;
    for (long i = 0; i <= n; i++) {
        result += i;
    }
    return result;
}
```

```
// 140ms 소요
public static long sequentialSum(long n) {
    return Stream.iterate(1L, i -> i + 1)
        .limit(n)
        .reduce(Long::sum)
        .get();
}
```

병렬 스트림 사용하기

- 1부터 10,000,000까지 값을 더하는 아래의 로직은 오히려 고전적인 for-loop 방식이 훨씬 빠름

```
// 142ms 소요
public static long parallelSum(long n) {
    return Stream.iterate(1L, i -> i + 1).limit(n)
        .parallel()
        .reduce(Long::sum)
        .get();
}
```

병렬 스트림으로 실행했을 때 오히려 낮은 성능

- iterate가 박싱(Boxing)된 객체를 생성하므로 이를 다시 언박싱(Unboxing) 하는 비용이 필요
- iterate는 독립적인 Chunk로 분할 하기는 어려워 병렬로 실행되지 않음

병렬 스트림 사용하기

- 순차 처리 스트림의 성능 개선

```
// 20ms
public static long rangedSum(long n) {
    return LongStream.rangeClosed(1, n)
        .reduce(Long::sum)
        .getAsLong();
}
```

LongStream + 고정 크기로 성능 개선

1. rangeClosed를 사용시 기본형을 직접 사용 하므로 박싱, 언박싱에 발생하는 비용을 줄일 수 있다.
2. 범위가 고정되어 있으므로 쉽게 Chunk를 분할 할 수 있다.

병렬 스트림 사용하기

- 병렬 처리 스트림의 성능 개선

```
// 4ms 드디어!!
public static long parallelRangedSum(long n) {
    return LongStream.rangeClosed(1, n)
        .parallel()
        .reduce(Long::sum)
        .getAsLong();
}
```

비로소 기존의 for-loop문 보다 빠른 코드를 작성하였다.

하지만 for-loop보다 비용이 작지 않다는 것을 기억해야 한다.

병렬 스트림 사용시 주의사항 1

- 측정하라. 병렬스트림은 항상 순차스트림 보다 빠르지 않다.
- 박싱(Boxing)을 주의하라. 오토박싱과 언박싱 과정은 성능을 크게 저하 시킨다.
자바에서 제공하는 기본형 특화 스트림을 사용하는 것이 좋다.
- 순차스트림 보다 병렬스트림에서 성능이 떨어지는 연산이 존재한다. limit나 findFirst 처럼 요소의 순서에 의존하는 연산을 병렬 스트림에서 수행하려면 비싼 비용이 필요하다.
- 소량의 데이터라면 병렬 스트림이 도움이 되지는 않는다.
- 소량의 데이터를 처리하는 과정은 병렬화 과정에서 생기는 부가비용을 상쇄할 수 있을 만큼 이득을 얻지 못하기 때문이다.
- 스트림을 구성하는 자료구조가 적절한지 확인하여라.

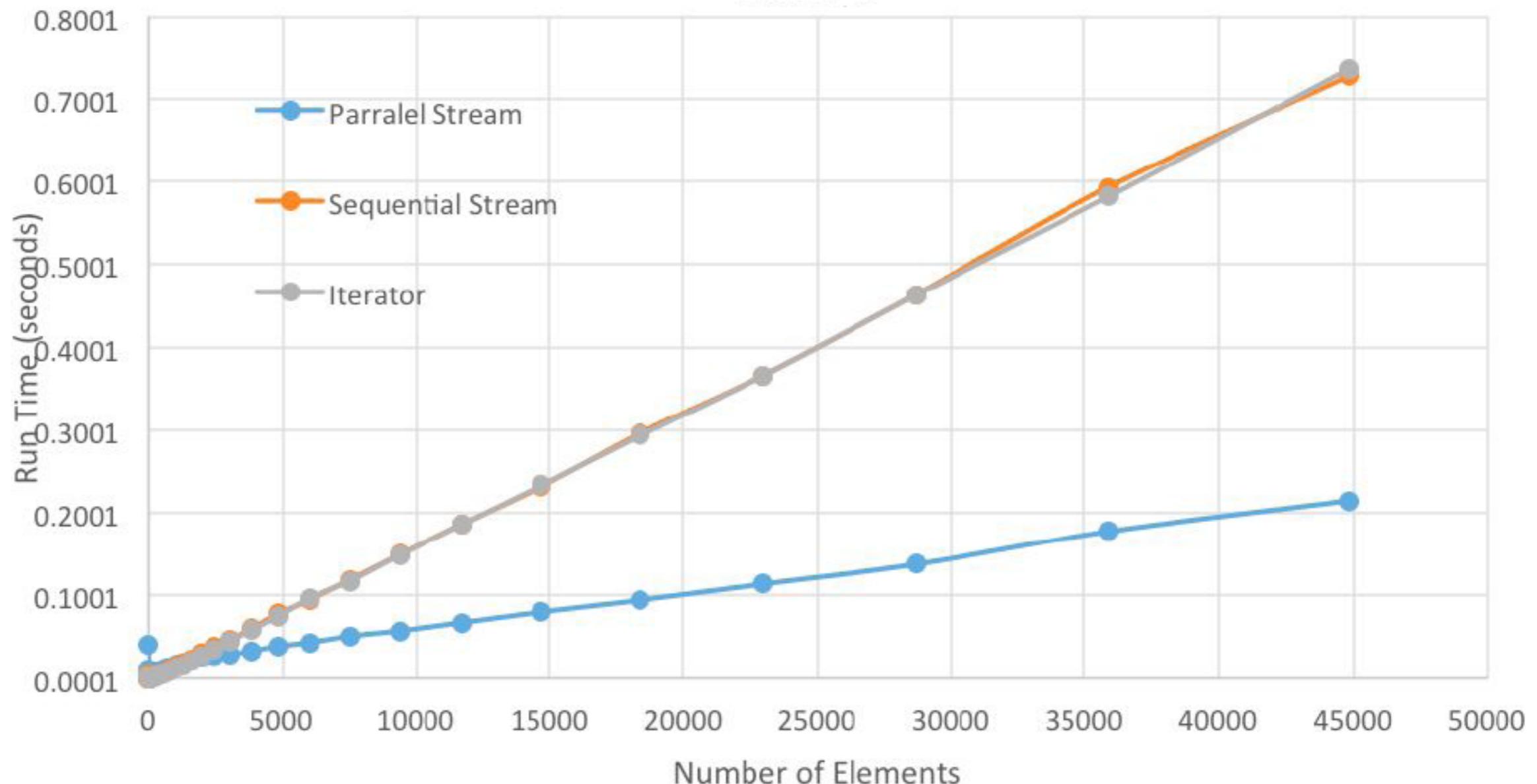
ArrayList는 LinkedList 보다 효율적으로 분리가 가능하다.

병렬 스트림 사용시 주의사항 2

- 스트림의 특성과 파이프라인의 중간 연산이 스트림의 특성을 어떻게 바꾸느냐에 따라서 분해 과정의 성능이 달라질 수 있다.
- 필터 연산이 있을 경우 크기를 예측할 수 없으므로 효과적인 스트림 처리가 되지 않는다.
- 최종 연산 과정에서 병합비용을 살펴보아야 한다.

병합 비용이 비싸다면 병렬 스트림으로 얻은 성능의 이익이 서브스트림의 부분 결과를 합치는 과정에서 상쇄될 수 있다.

병렬 스트림 사용하기



요약

- 스트림은 간단하게 병렬 처리 방식으로 변경할 수 있다.
- 병렬 처리 방식이 항상 빠른 것은 아니므로, 병렬로 데이터를 처리할 경우에는 유의하여 작업을 진행하여야 한다.
- 병렬 처리를 위해 코드에서 포크조인 프레임워크를 직접 사용할 수는 있지만, 병렬 스트림을 이용할 경우에는 백그라운드에서 포크조인 프레임워크가 사용되기 때문에 매우 쉽게 병렬 처리 할 수 있음

1-07

Java 9 New Features

Java 9 향상된 성능

- 메모리 사용 효율 증가
- 성능 증가
 - Locking
 - Secure Apps
 - 그래픽스
- 더 좋아진 하드웨어 사용
- 더 좋아진 문서화
- 그래픽 성능 향상 (윈도우와 리눅스)
- 컴파일 성능 증가

Java 9 New Features

- Store Interned Strings in CDS Archives
- Improve Contended Locking
- Compact Strings
- Improve Secure Application Performance
- Leverage CPU Instructions for GHASH and RSA
- Tiered Attribution for `javac`
- Javadoc Search
- Marlin Graphics Renderer
- HiDPI Graphics on Windows and Linux
- Enable GTK 3 on Linux
- Update JavaFX/Media to Newer Version of GStreamer

Behind the scenes

- Jigsaw – Modularize JDK
- Enhanced Deprecation
- Stack-Walking API
- Convenience Factory Methods for Collections
- Platform Logging API and Service
- `jshell`: The Java Shell (Read-Eval-Print Loop)
- Compile for Older Platform Versions
- Multi-Release JAR Files
- Platform-Specific Desktop Features
- TIFF Image I/O\
- Multi-Resolution Images

New functionality

- Process API Updates
- Variable Handles
- Spin-Wait Hints
- Dynamic Linking of Language-Defined Object Models
- Enhanced Method Handles
- More Concurrency Updates
- Compiler Control

Specialized

- HTTP 2 Client
- Unicode 8.0
- UTF-8 Property Files
- Datagram Transport Layer Security (DTLS)
- OCSP Stapling for TLS
- TLS Application-Layer Protocol Negotiation Extension
- SHA-3 Hash Algorithms
- DRBG-Based `SecureRandom` Implementations
- Create PKCS12 Keystores by Default
- Merge Selected Xerces 2.11.0 Updates into JAXP
- XML Catalogs
- HarfBuzz Font-Layout Engine
- HTML5 Javadoc

New standards

- Parser API for Nashorn
- Prepare JavaFX UI Controls & CSS APIs for Modularization
- Modular Java Application Packaging
- New Version-String Scheme
- Reserved Stack Areas for Critical Sections
- Segmented Code Cache
- Indify String Concatenation
- Unified JVM Logging
- Unified GC Logging
- Make G1 the Default Garbage Collector
- Use CLDR Locale Data by Default
- Validate JVM Command-Line Flag Arguments
- Java-Level JVM Compiler Interface
- Disable SHA-1 Certificates
- Simplified Doclet API
- Deprecate the Applet API
- Process Import Statements Correctly
- Annotations Pipeline 2.0
- Elide Deprecation Warnings on Import Statements
- Milling Project Coin

Housekeeping

- Remove GC Combinations Deprecated in JDK 8
- Remove Launch-Time JRE Version Selection
- Remove the JVM TI `hprof` Agent
- Remove the `jhat` Tool

Gone

Immutable List

■ Immutable List의 특징

- 엘리먼트를 추가, 수정, 삭제 할 수 없다.
- Add/Delete/Update를 수행하면 UnsupportedOperationException이 발생한다.

■ Java SE 8 : Empty Immutable List

```
List<String> emptyList = new ArrayList<>();  
List<String> immutableList = Collections.unmodifiableList(emptyList);
```

■ Java SE 9 : Empty Immutable List

```
static <E> List<E> of()
```

```
List<String> immutableList = List.of();
```

Immutable List

- Java SE 8 : Non-Empty Immutable List

```
List<String> list = new ArrayList<>();  
list.add("one");  
list.add("two");  
list.add("three");  
List<String> immutableList = Collections.unmodifiableList(list);
```

- Java SE 9 : Non-Empty Immutable List

```
static <E> List<E> of(E... elements)
```

```
List<String> immutableList = List.of("one", "two", "three");
```

Private Methods in Interfaces

- Java 7 Interface

- Constant variables
- Abstract methods

```
public interface DBLogging{  
    String MONGO_DB_NAME = "ABC_Mongo_Datastore";  
    String NEO4J_DB_NAME = "ABC_Neo4J_Datastore";  
    String CASSANDRA_DB_NAME = "ABC_Cassandra_Datastore";  
  
    void logInfo(String message);  
    void logWarn(String message);  
    void logError(String message);  
    void logFatal(String message);  
}
```

Private Methods in Interfaces

Java 8 Interface Changes

- Constant variables
- Abstract methods
- Default methods
- Static methods

```
public interface DBLogging{  
    String MONGO_DB_NAME = "ABC_Mongo_Datastore";  
    String NEO4J_DB_NAME = "ABC_Neo4J_Datastore";  
    String CASSANDRA_DB_NAME = "ABC_Cassandra_Datastore";  
    // abstract method example  
    void logInfo(String message);  
    // default method example  
    default void logWarn(String message) {  
        // Step 1: Connect to DataStore  
        // Step 2: Log Warn Message  
        // Step 3: Close the DataStore connection  
    }  
    // static method example  
    static boolean isNull(String str) {  
        System.out.println("Interface Null Check");  
        return str == null ? true : "".equals(str) ? true : false;  
    }  
    // Any other abstract, default, static methods  
}
```

Private Methods in Interfaces

Java 9 Interface Changes

- Constant variables
- Abstract methods
- Default methods
- Static methods
- Private methods
- Private Static methods

```
public interface DBLogging {  
    String MONGO_DB_NAME = "ABC_Mongo_Datastore";  
    String NEO4J_DB_NAME = "ABC_Neo4J_Datastore";  
    String CASSANDRA_DB_NAME = "ABC_Cassandra_Datastore";  
    default void logInfo(String message) {  
        log(message, "INFO");  
    }  
    default void logWarn(String message) {  
        log(message, "WARN");  
    }  
    default void logFatal(String message) {  
        log(message, "FATAL");  
    }  
    private void log(String message, String msgPrefix) {  
        // Step 1: Connect to DataStore  
        // Step 2: Log Message with Prefix and styles etc.  
        // Step 3: Close the DataStore connection  
    }  
    // Any other abstract, static, default methods  
}
```

Optional Class In Java 9

- Java 9에서는 Optional Class에 대해 새로운 메서드들이 추가되었습니다.
 - ifPresent(Consumer action) : 값이 현존한다면 값을 이용하여 액션을 수행 합니다.

```
// Using Java 8
if (OptionalObj.isPresent())
    ...
// Using Java 9
OptionalObj.ifPresent(() -> action());
```

- ifPresentOrElse(Consumer action, Runnable emptyAction) : ifPresent와 유사 하지만 값이 empty일때 emptyAction을 수행합니다.

```
// Using null check:
Student student = getStudentByIdNo(studentIdNo);
if (student != null)
    displayStudentInfo(student);
else
    displayStudentNotFound();

// Using the new Java 9 Optional::ifPresentOrElse:
getStudentByIdNo(studentIdNo).ifPresentOrElse(this::displayStudentInfo, ()-> displayStudentNotFound());
```

Optional Class In Java 9

- Java 9에서는 Optional Class에 대해 새로운 메서드 들이 추가되었습니다.
- stream() : 값의 유무에 따라 구성된 스트림을 반환합니다. empty 체크를 해서 자동으로 해당 element를 삭제합니다.

```
// Using Java 8
List<String> strings = streamOptional()
    .filter(Optional::isPresent)
    .map(Optional::get)
    .collect(Collectors.toList());

// Result: strings[one, two, three]

// Using Java 9
List<String> newStrings = streamOptional()
    .flatMap(Optional::stream)
    .collect(Collectors.toList());

// Result: newStrings[one, two, three]
```

REPL(Read-Eval-Print Loop) : jshell

■ REPL(Read-Eval-Print-Loop)

- Python, Ruby, NodeJS 등의 언어들은 이미 제공하고 있는 REPL(Read-Eval-Print-Loop) 기능이 JAVA9 에는 새로운 REPL command-line tool인 jshell을 포함
- jshell은 컴파일, 수정, 실행을 IDE를 사용하지 않고 쉽게 자바 코드의 개발과 테스트 가능
- 이 도구를 사용하기 쉽게 하기 위해 편집 가능한 기록, 탭 완성, 세미콜론 자동 추가, 미리 구별 가능한 imports 및 definitions와 같은 기능이 포함

```
$ jshell
| Welcome to JShell -- Version 9-ea
| For an introduction type: /help intro

jshell> 3+3
$1 ==> 6

jshell> int x=5
x ==> 5

jshell> x
x ==> 5

jshell> void print() { System.out.println("Hello JShell"); }
| created method print()

jshell> print()
Hello JShell
```

Module System

Jigsaw Project

- Java9의 가장 큰 변화 중 하나는 모듈 시스템
- Jigsaw Project의 특징
 - Modular JDK
 - Encapsulate Java Internal APIs
 - Java Platform Module System
- JDK 모듈을 사용할 수 있고 아래와 같이 자신의 모듈을 만들 수 있음
- Simple Module Example



```
module com.atin.monitor.ui {  
    requires javafx.base;  
    requires javafx.controls;  
    requires javafx.graphics;  
    exports com.atin.monitor.ui.launch;  
    exports com.atin.monitor.ui.show;  
}
```

Process API Improvements

- Process API
 - Java9에서 프로세스 API를 개선함
 - OS 프로세스를 관리 및 컨트롤 할 수 있음
 - java.lang.ProcessHandle
 - java.lang.ProcessHandle.info
 - Process API Example

```
ProcessHandle currentProcess = ProcessHandle.current();
System.out.println("Id: = " + currentProcess.pid());
```

Try with Resources Improvement

- Java 7에서 나온 자원 관리 방법(try with resource)을 개선함

- Java7

```
void testARM_Before_Java9() throws IOException{
    BufferedReader reader1 = new BufferedReader(
        new FileReader("journaldev.txt"));
    try (BufferedReader reader2 = reader1) {
        System.out.println(reader2.readLine());
    }
}
```

- Java9

```
void testARM_Java9() throws IOException{
    BufferedReader reader1 = new BufferedReader(
        new FileReader("journaldev.txt"));
    try (reader1) {
        System.out.println(reader1.readLine());
    }
}
```

Reactive Streams

- **Reactive Programming**
 - Java9은 리액티브 스트림 API를 추가함
 - Java9의 리액티브 스트림 API
 - Java를 이용해서 매우 쉽게 병행성, 확장성, 비동기 어플리케이션을 구현 한 Publish / Subscribe 프레임워크
 - API
 - java.util.concurrent.Flow
 - java.util.concurrent.Flow.Publisher
 - java.util.concurrent.Flow.Subscriber
 - java.util.concurrent.Flow.Processor

Stream API Improvements

- Stream 인터페이스에 추가된 2개의 default method
 - 중요한 2개의 메서드
 - dropWhile 메서드
 - takeWhile 메서드
 - takeWhile()은 인수로서의 predicate를 취하고 주어진 값의 스트림 서브셋을 리턴 한다.
 - 해당 값을 만족시키는 값이 없으면 Empty Stream을 리턴 한다.

```
jshell> Stream.of(1,2,3,4,5,6,7,8,9,10)
          .takeWhile(i -> i < 5 )
          .forEach(System.out::println);
1
2
3
4
```

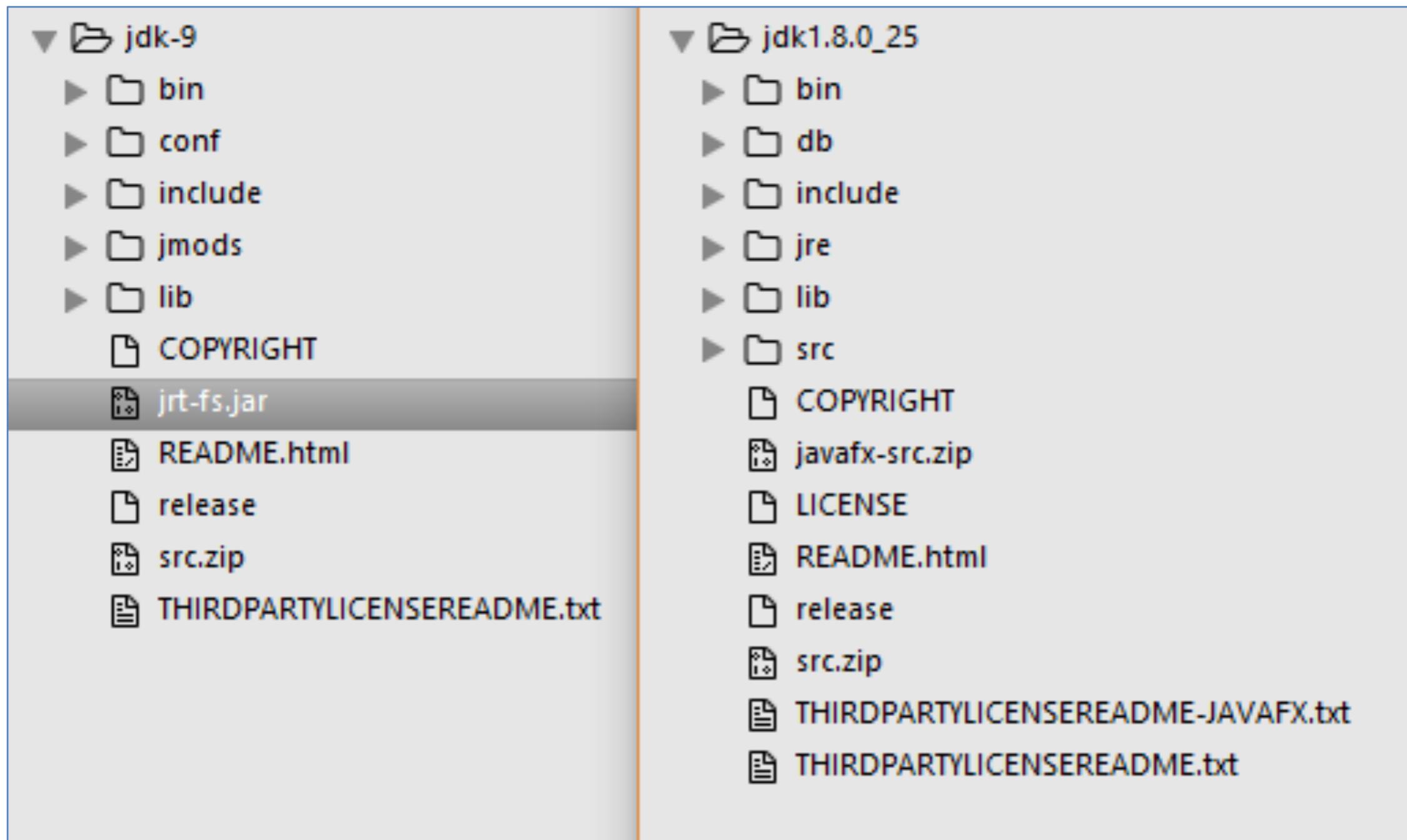
HTTP 2 Client

- **Http API가 추가된 배경**

- HTTP/2 프로토콜과 WebSocket 기능을 지원하기 위함
- java.net.http 패키지에 새로운 HTTP 2 클라이언트 API 도입
- HTTP / 1.1 및 HTTP / 2 프로토콜 지원
- 동기화(블로킹 Mode)와 비동기 Mode를 모두 지원
- WebSocket API를 사용하여 비동기 Mode 지원

JDK 8 vs. JDK 9

- 자바 9에서 Jigsaw 프로젝트에 의해 JDK 가 모듈화 됨
- 설치된 JDK 디렉토리 구조가 변경



Java 9 Modular Programming

JIGSAW 프로젝트

디펜던시 찾기

- 모듈러 프로그래밍의 첫 번째는 디펜던시를 확인하는 것
 - JDK 8에서 소개된 jdeps 툴을 사용하여 정적 분석 가능
 - IDE(인텔리제이 아이디어)에도 포함되어 있음

모듈 어플리케이션 작성 실습

- 어플리케이션을 다음 두개의 모듈로 분리
 - math.util 모듈 : 수학 계산을 수행하는 메서드 포함
 - calculator 모듈 : 계산기 실행
- math.util 모듈
 - MathUtil 클래스
 - public int add(int a, int b);
 - public int sub(int a, int b);
- calculator 모듈
 - Calculator 클래스

모듈 어플리케이션 작성 실습

- 프로젝트 디렉토리 구조

```
application_root_directory
|--module1_root
|---module-info.java
|---com
|----sample
|-----MyClass.java
|--module2_root
|---module-info.java
|---com
|----test
|-----MyAnotherClass.java
```

감사합니다.

soongon@hucloud.co.kr