

# Reactive Programming

김 순곤

[soongon@hucloud.co.kr](mailto:soongon@hucloud.co.kr)

# 목 차

---

- 1부: 자바 함수형 프로그래밍

- 인터페이스와 람다 표현식
- 스트림의 활용

- 2부: 리액티브 프로그래밍

- Reactive 프로그래밍 소개
- RxJava 3.0 이해와 실습
- Reactor 이해와 실습

1부

# Java Functional Programming

# 자바 8+ 함수형 프로그래밍 지원

---

- 인터페이스 대대적 변화
  - Static 메서드 지원
  - Default 메서드 지원
  - Private 메서드 지원
  - 함수형 인터페이스에서 타입 추론 기능 : 람다 지원을 위한 기능
- 람다 지원
- Stream API 지원 : map, flatMap 등
- Optional 지원 : null 세이프 기능
- 새로운 시간과 날짜 관련 API

# Interface 향상

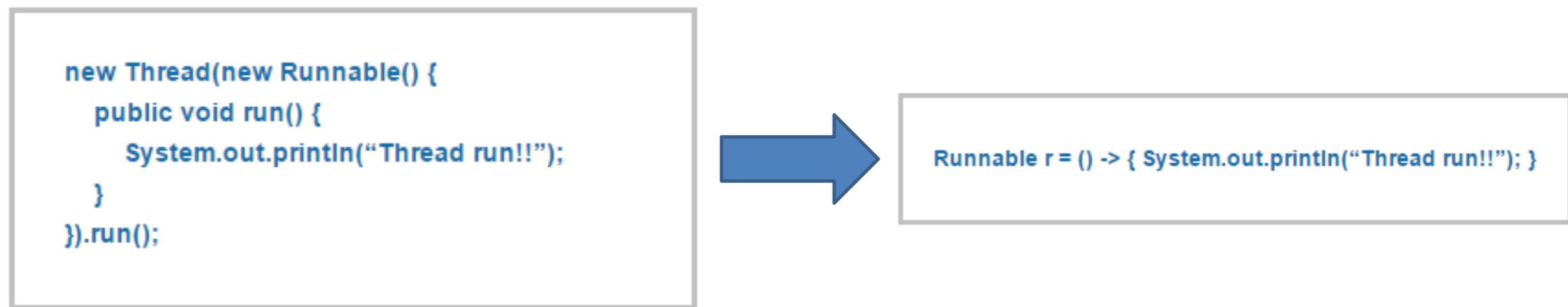
---

## ■ 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 클래스에도 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



- 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.

# **함수형 프로그래밍 이해**

# 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? - IO
  - 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 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 전달 할 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<>()
```

# 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 중간작업(intermediate)을 수행한 후, 마지막으로 하나의 최종 종료(final terminal) 작업을 수행한다.
- 중간작업은 filter, map, flatMap, peek, distinct, sorted, limit, substream 이다.
- 최종 종료작업은 forEach, toArray, reduce, collect, min, max, count, anyMatch, allMatch, noneMatch, findFirst, findAny 를 포함한다.

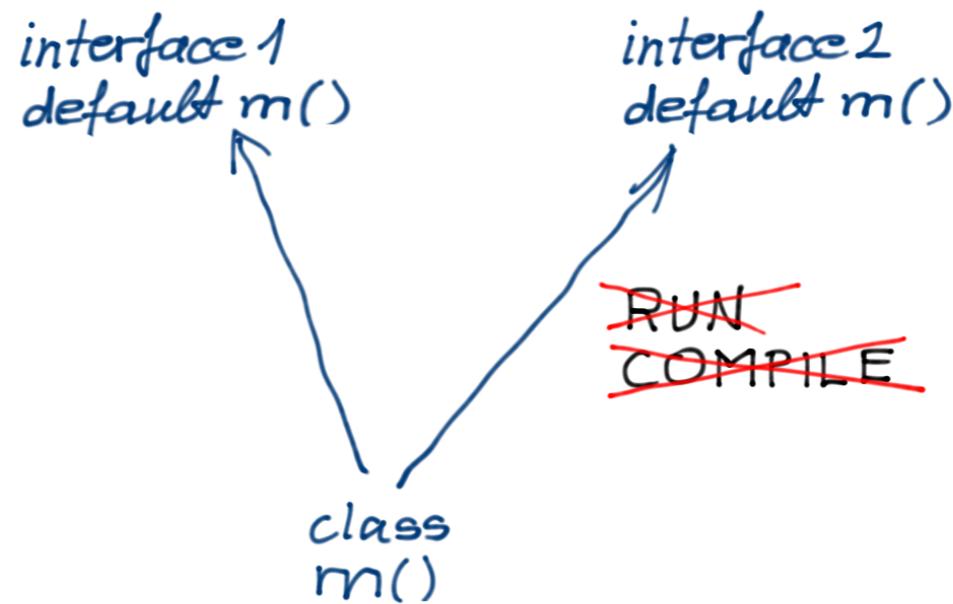
# **인터페이스의 변화**

# 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

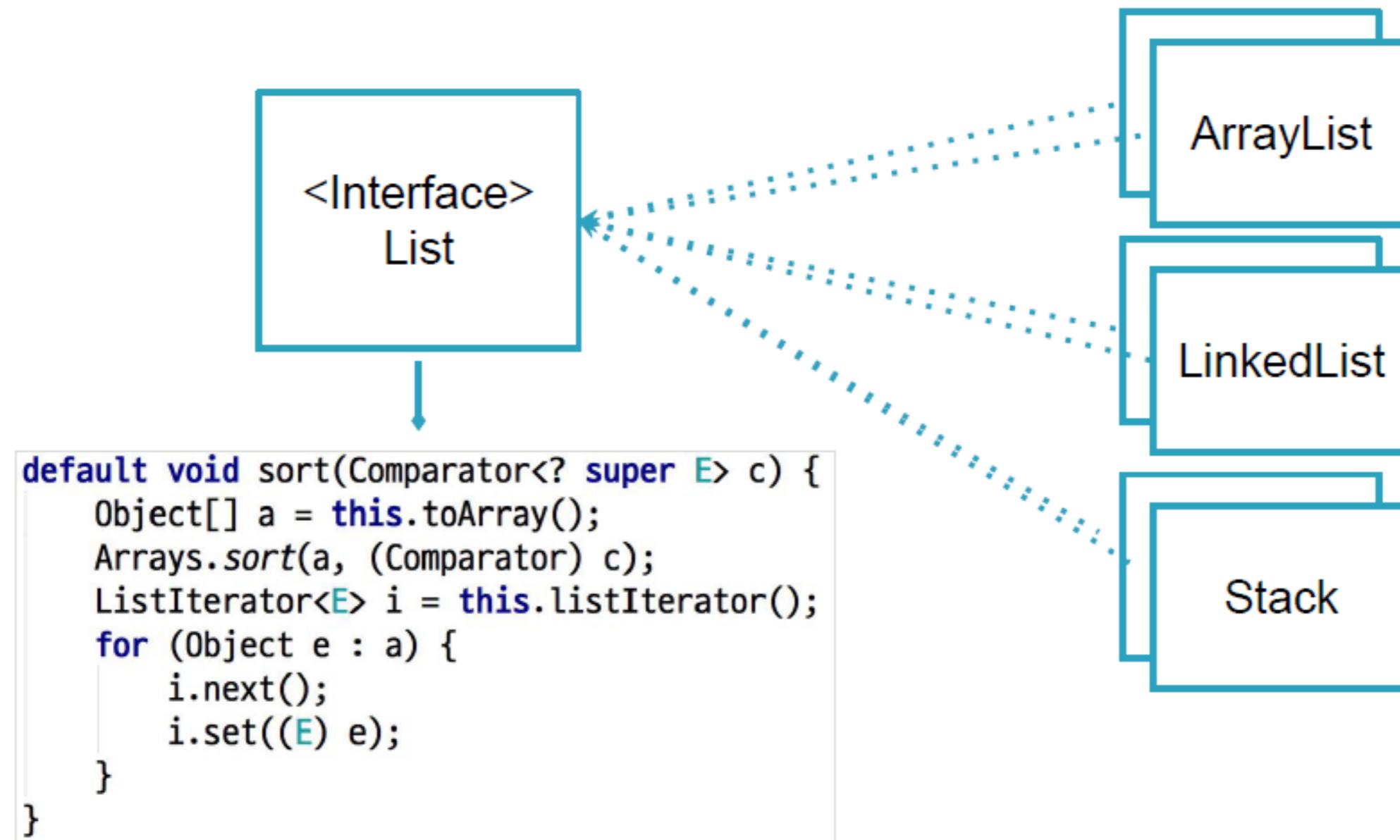
## ■ 디폴트 메소드 충돌 주의



# 인터페이스의 변화 - default method

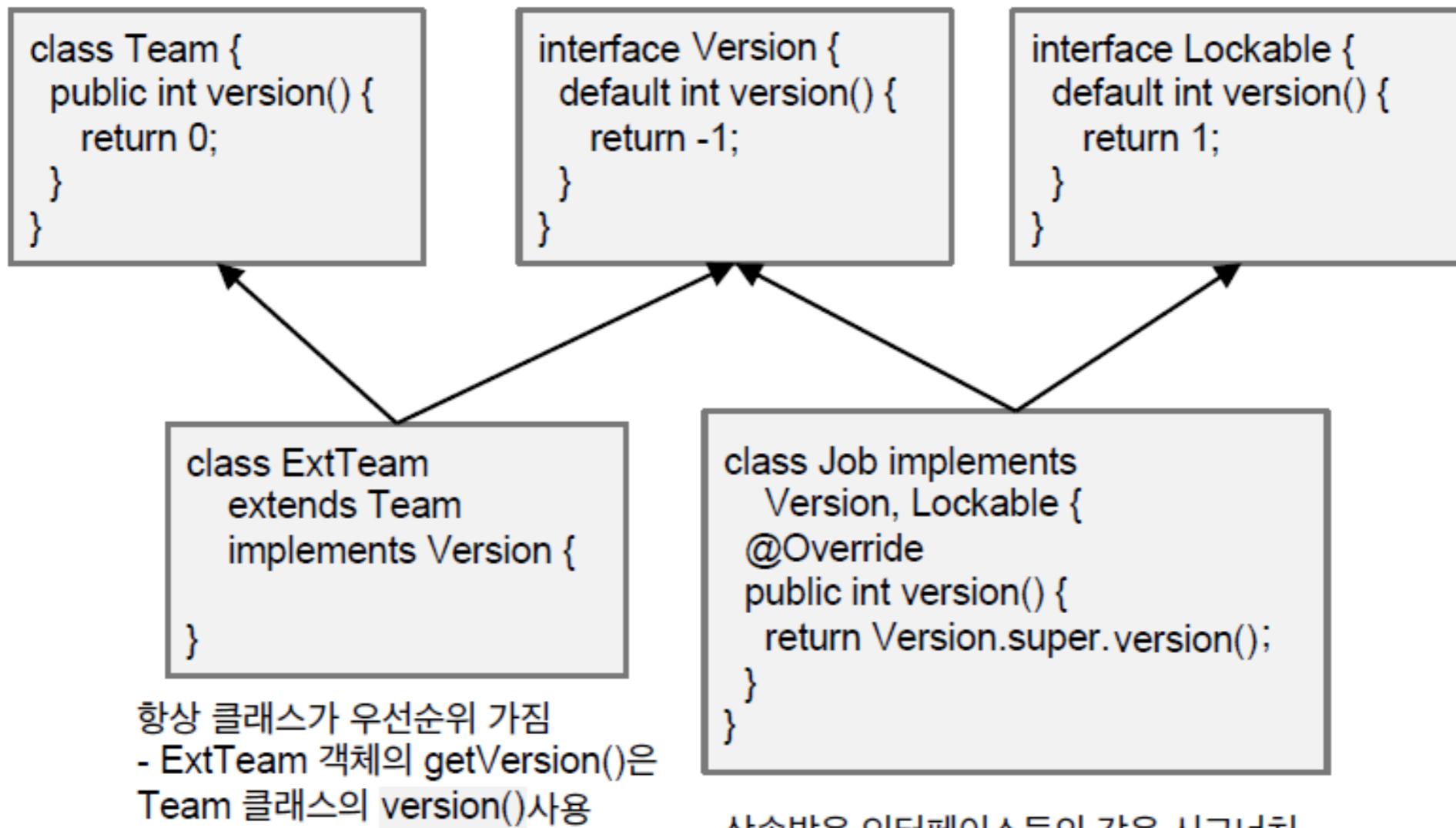
## ■ 디폴트 메서드 (default method)

- default 키워드를 사용하면 인터페이스에 새로운 메서드를 추가 하더라도 기존의 구현체 구조를 변경하지 않고 새로운 기능을 추가 할 수 있다.



# 인터페이스의 변화 - default method

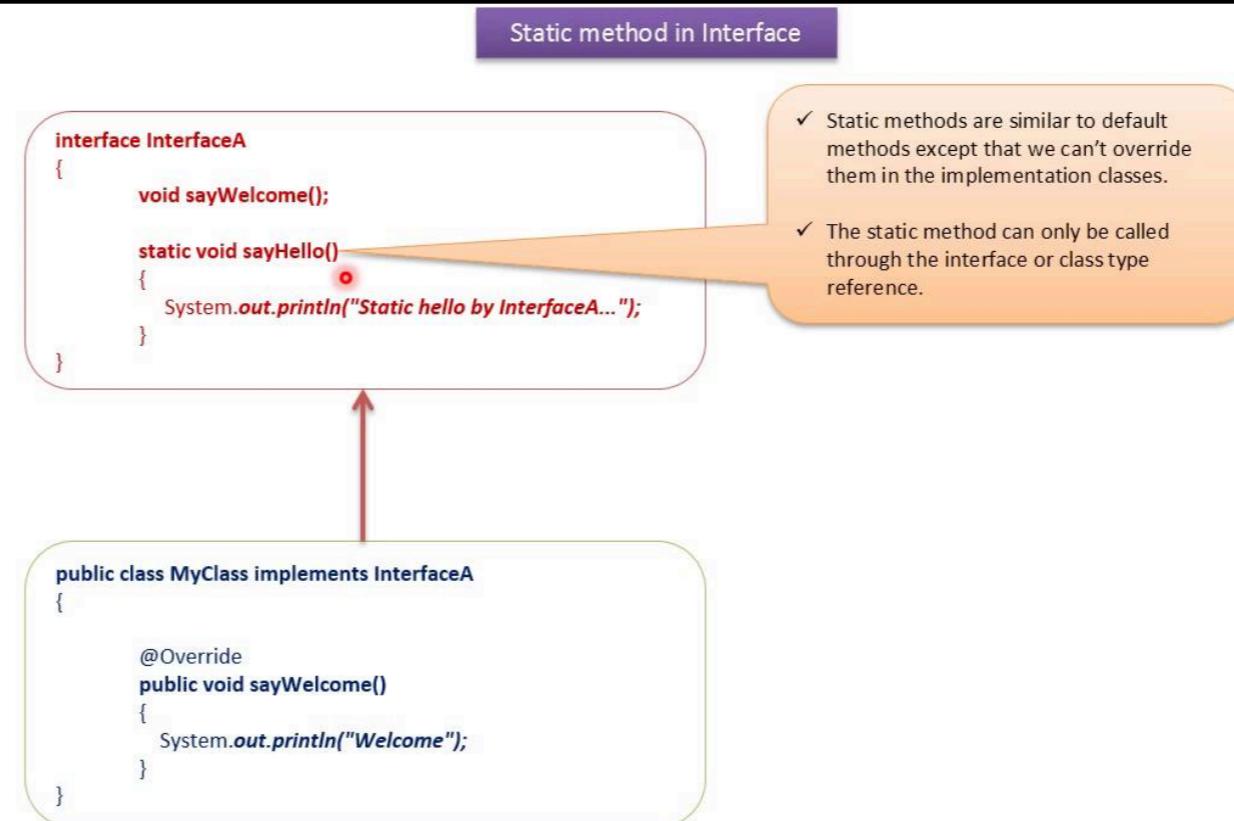
## ■ 디폴트 메서드 (default method)의 우선 순위



# 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( $\lambda$ -, 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

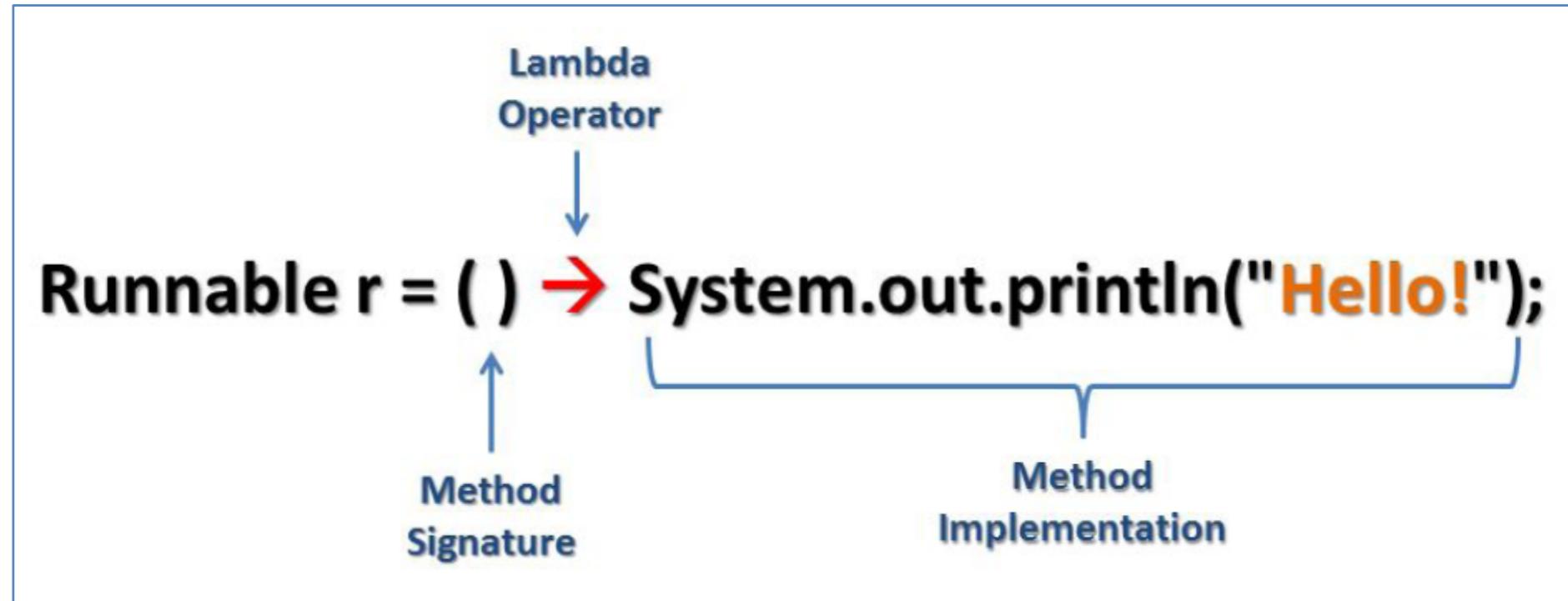
- 수학에서 람다식 문법

$$\lambda x. x+1$$


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

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

## 리스트 정렬 사례를 통한 람다식

# 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 의명클래스

## ■ 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

- @FunctionalInterface 를 클래스 상단에 붙여 명시적으로 함수형 인터페이스라는 것을 표기할 수 있다.
- @FunctionalInterface가 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()

# 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

# Consumer 함수형 인터페이스

## ■ Consumer<T> T -> 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

# Function 함수형 인터페이스

- Function<T, R> T->R      아규먼트값 → **Function** → 리턴값
  - 추상메서드 apply(T)를 람다의 바디에서 정의 하고, 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 T 를 객체 R 로 매핑
BiFunction<T,U,R>	R apply(T t, U u)	객체 T 와 U 를 객체 R 로 매핑
DoubleFunction<R>	R apply(double value)	double 을 객체 R 로 매핑
IntFunction<R>	R apply(int value)	int 를 객체 R 로 매핑
IntToDoubleFunction	double applyAsDouble(int value)	int 를 double 로 매핑
IntToLongFunction	long applyAsLong(int value)	int 를 long 으로 매핑
LongToDoubleFunction	double applyAsDouble(long value)	long 을 double 로 매핑
LongToIntFunction	int applyAsInt(long value)	long 을 int 로 매핑
ToDoubleBiFunction<T,U>	double applyAsDouble(T t, U u)	객체 T 와 U 를 double 로 매핑
ToDoubleFunction<T>	double applyAsDouble(T value)	객체 T 를 double 로 매핑
ToIntBiFunction<T,U>	int applyAsInt(T t, U u)	객체 T 와 U 를 int 로 매핑
ToIntFunction<T>	int applyAsInt(T value)	객체 T 를 int 로 매핑
ToLongBiFunction<T,U>	long applyAsLong(T t, u)	객체 T 와 U 를 long 으로 매핑
ToLongFunction<T>	long applyAsLong(T value)	객체 T 를 long 으로 매핑

# 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

# 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

# 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.

# **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, 맵핑, 정렬, 그룹핑의 중간 처리가 필요함( 스트림 파이프라인 필요성)

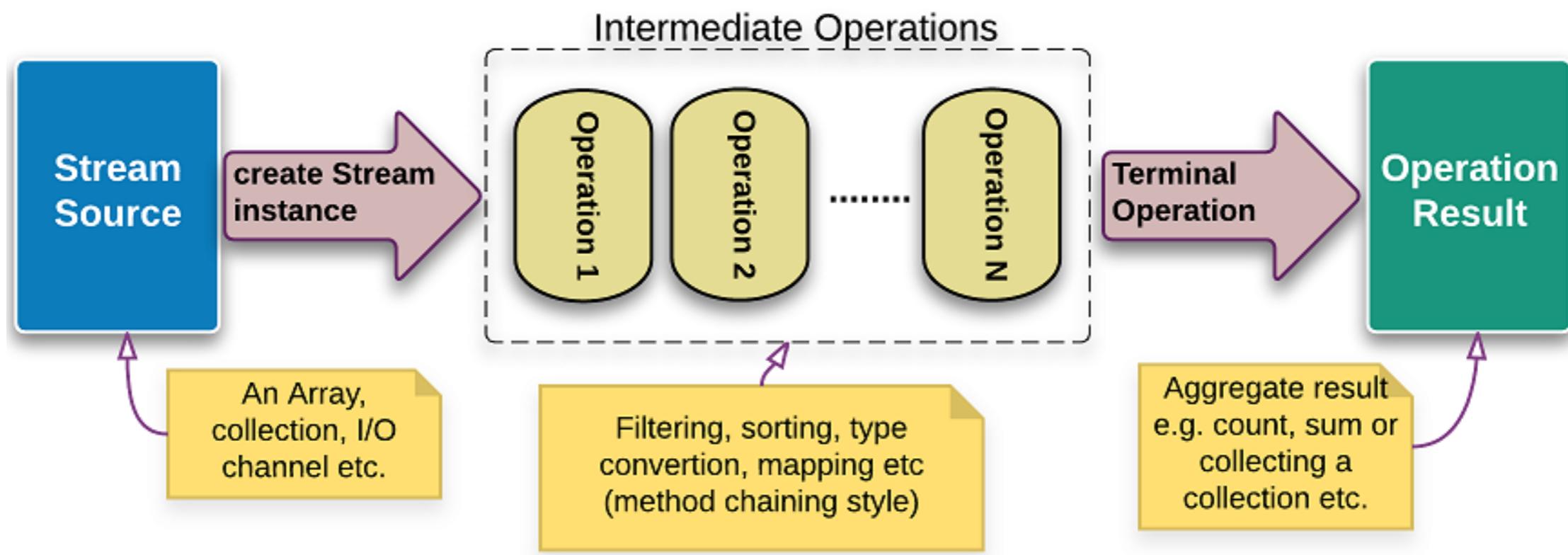
## ■ 파이프라인(Pipeline)

- 여러 개의 스트림이 연결 되어 있는 구조
- 파이프라인에서 최종 처리를 제외 하고는 모두 중간 처리 스트림



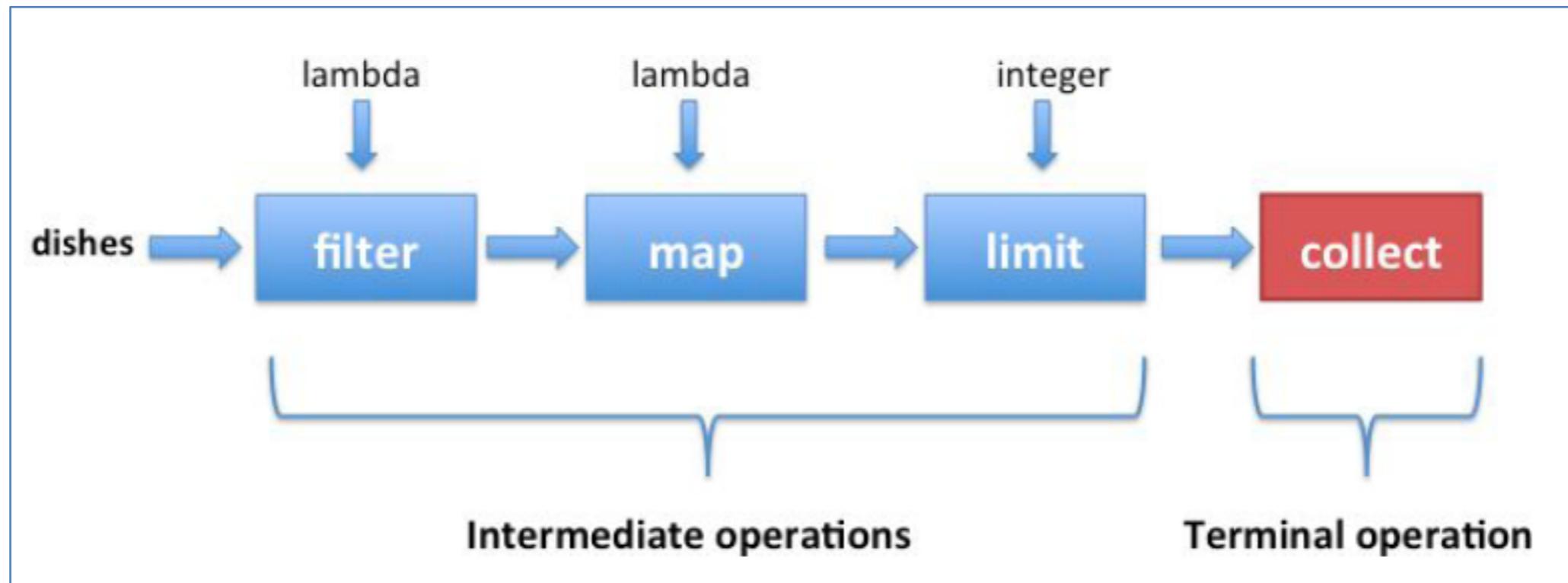
# 스트림(Stream)의 작업흐름

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 지정된 중간 연산이 실행되게 한다.



# 스트림(Stream) 특징

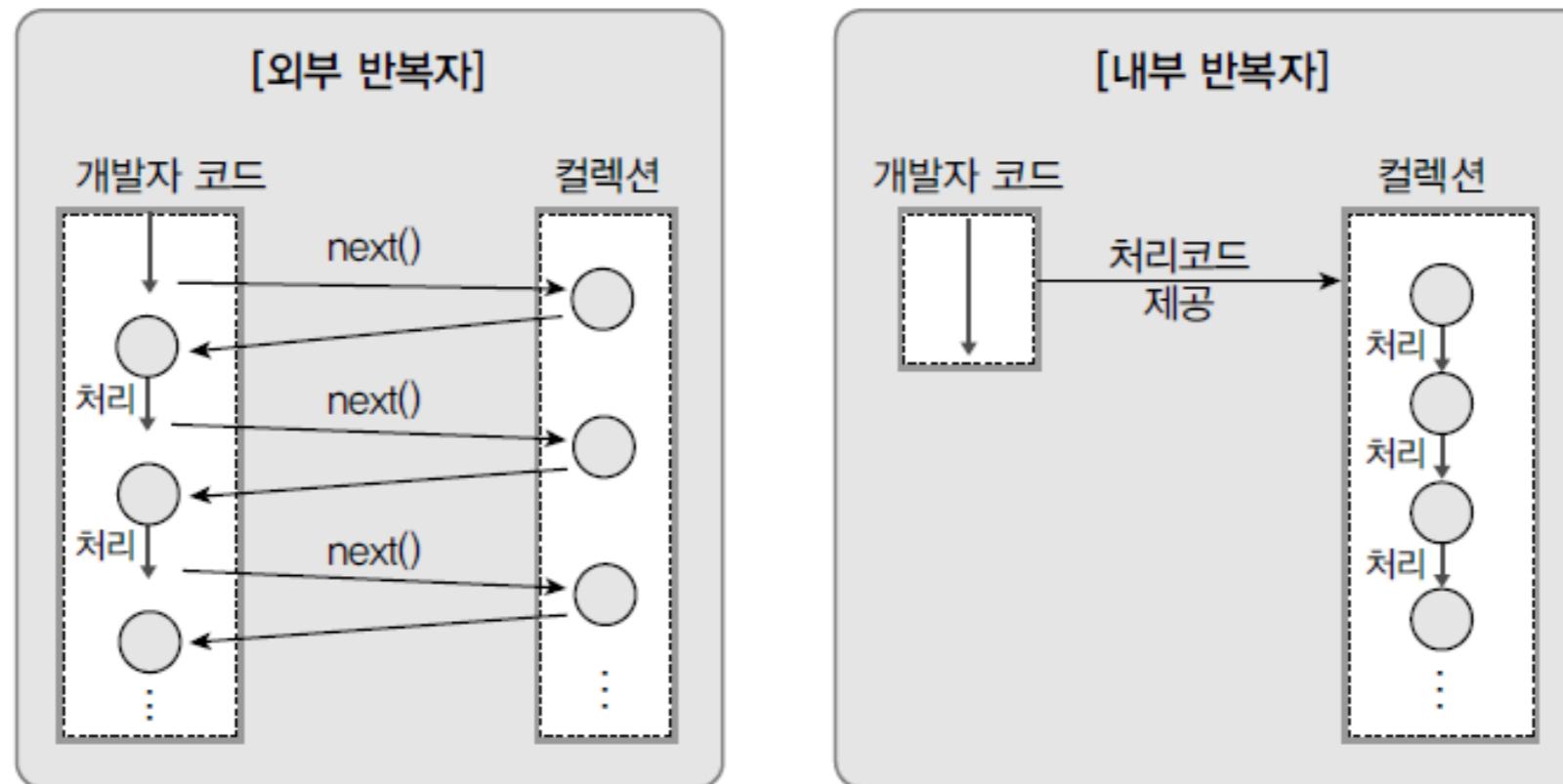
- 1. 스트림(Stream)은 Iterator와 비슷한 역할을 하는 반복자이다.
- 2. 스트림(Stream)이 제공하는 요소 처리 메서드는 함수적 인터페이스 타입이므로 람다식 또는 메서드 참조를 이용하여 요소(Element) 처리 내용을 인자로 전달할 수 있다.
- 3. Stream API는 이 메서드들을 연결하여, 복잡한 연산을 처리하는 로직을 쉽고 유연하게 작성해 낼 수 있다.



# 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

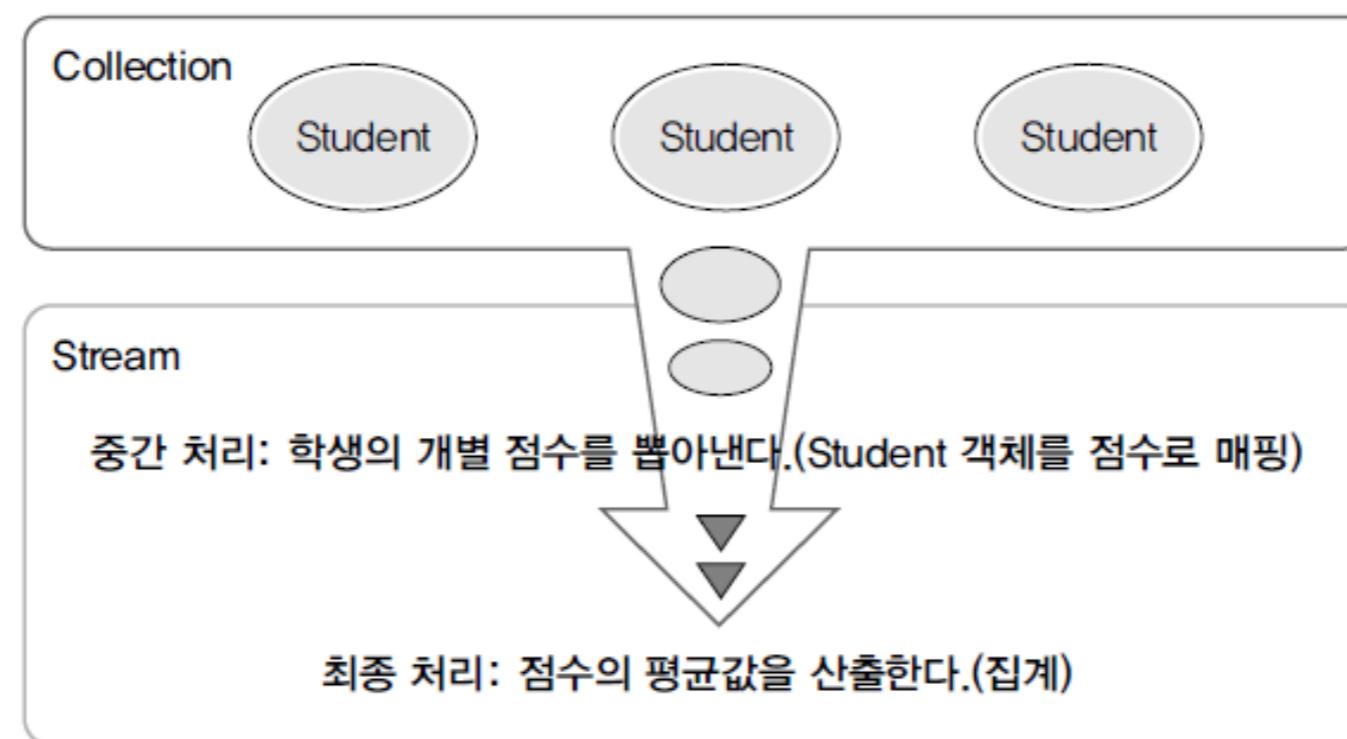
무한 연속 데이터 흐름 API

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

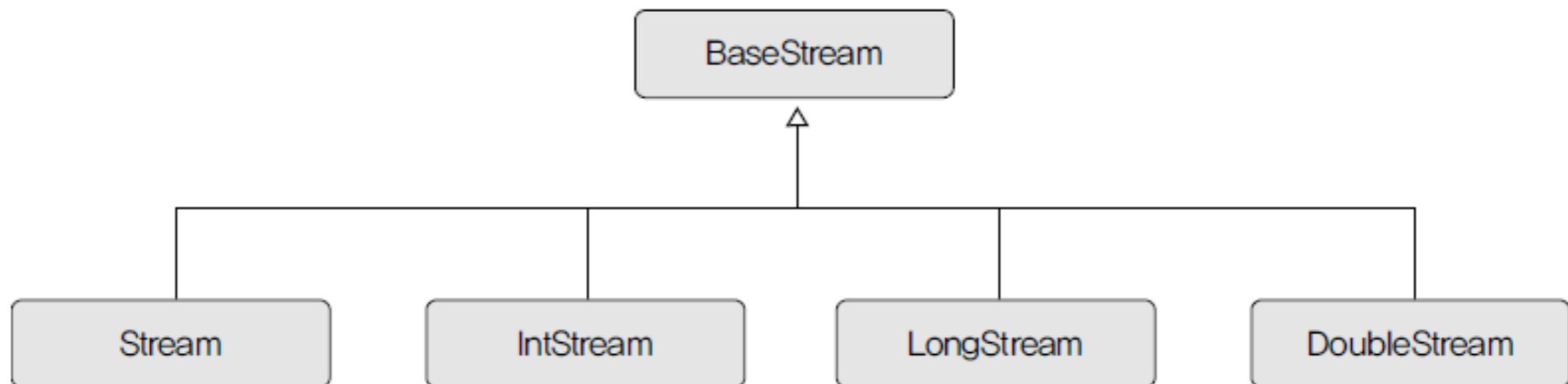
---

- 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");

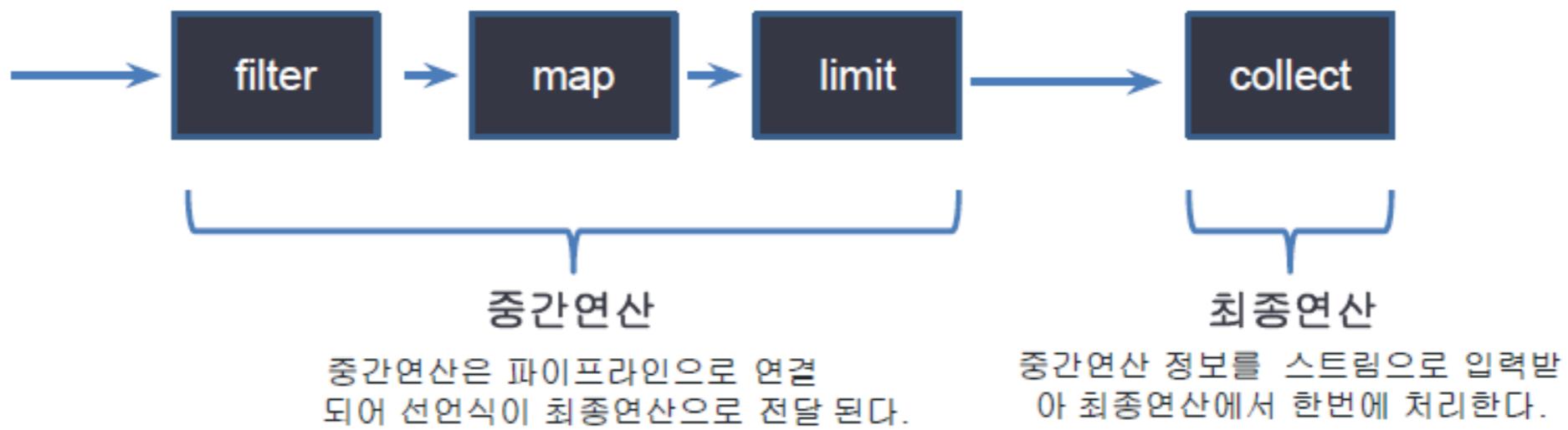
//Stream static 메서드 iterate 사용 - Stream.iterate(seed,operator)
Stream.iterate(0, n -> n + 2)
    .limit(10)
    .forEach(System.out::println);

//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, limit는 서로 연결되어 파이프 라인을 형성한다.
- collect로 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	필터링	distinct()	공통
		filter(...)	공통
		flatMap(...)	공통
		flatMapToDouble(...)	Stream
		flatMapToInt(...)	Stream
	매핑	flatMapToLong(...)	Stream
		map(...)	공통
		mapToDouble(...)	Stream, IntStream, LongStream
		mapToInt(...)	Stream, LongStream, DoubleStream
		mapToLong(...)	Stream, IntStream, DoubleStream
	정렬	mapToObj(...)	IntStream, LongStream, DoubleStream
		asDoubleStream()	IntStream, LongStream
		asLongStream()	IntStream
		boxed()	IntStream, LongStream, DoubleStream
		sorted(...)	공통
	루핑	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

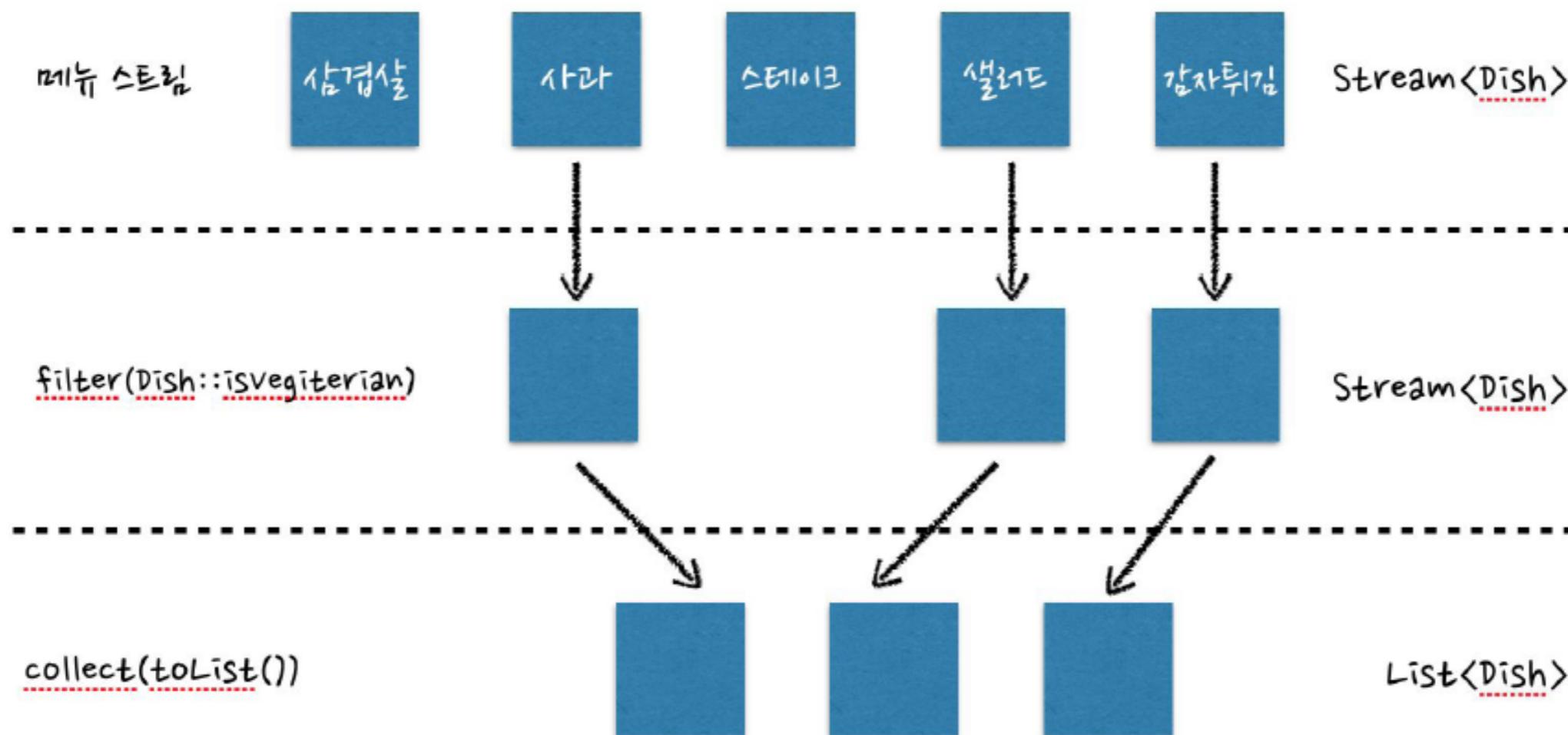
---

- collect 연산: 스트림 요소 수집
  - `O <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

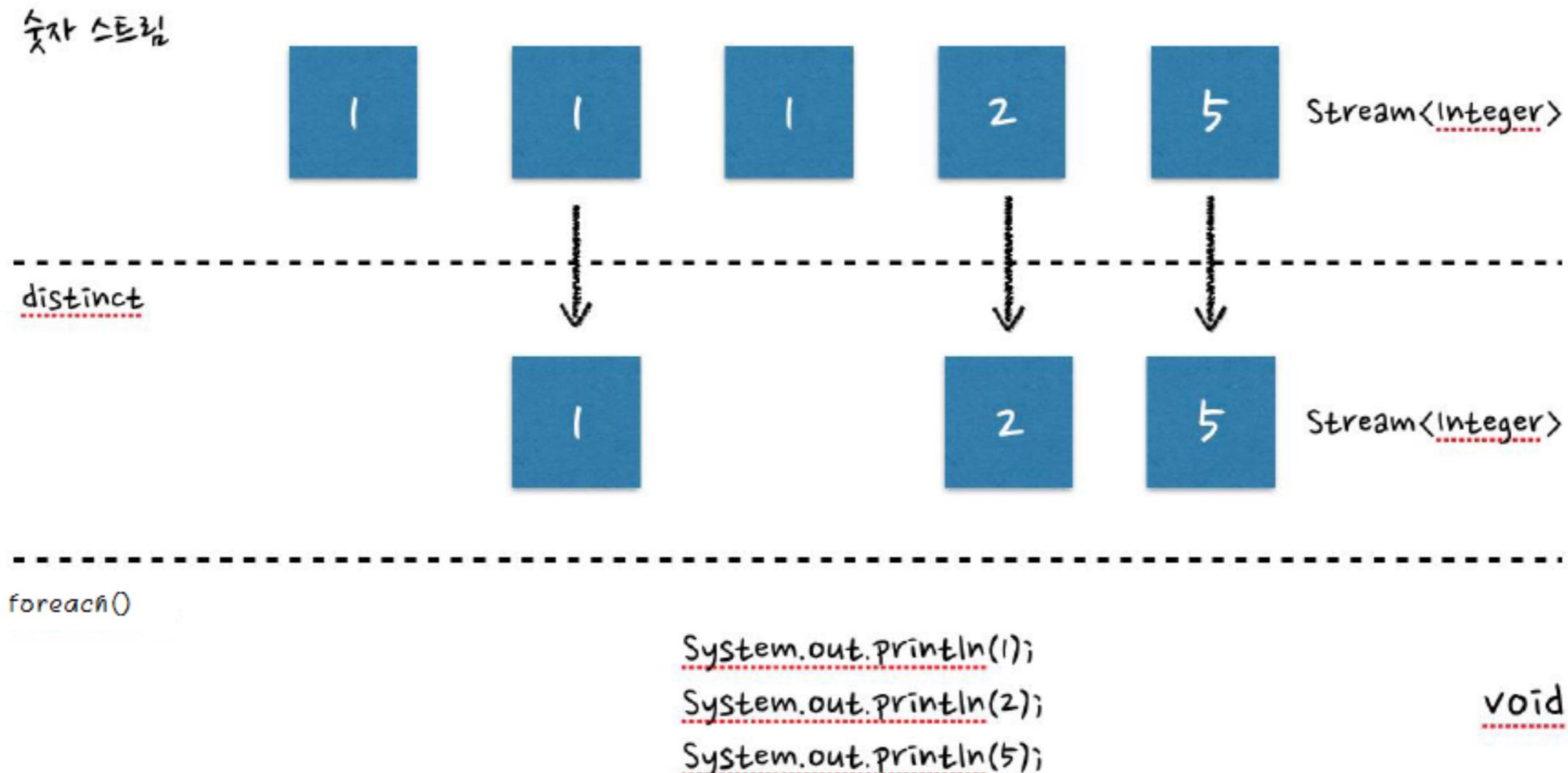
```
menu.stream()  
    .filter(Dish::isVegetarian)  
    .collect(Collectors.toList());
```



# 스트림 API의 활용 – 필터링/슬라이싱

- `distinct(중간연산)` : 유일한 값을 반환한다.

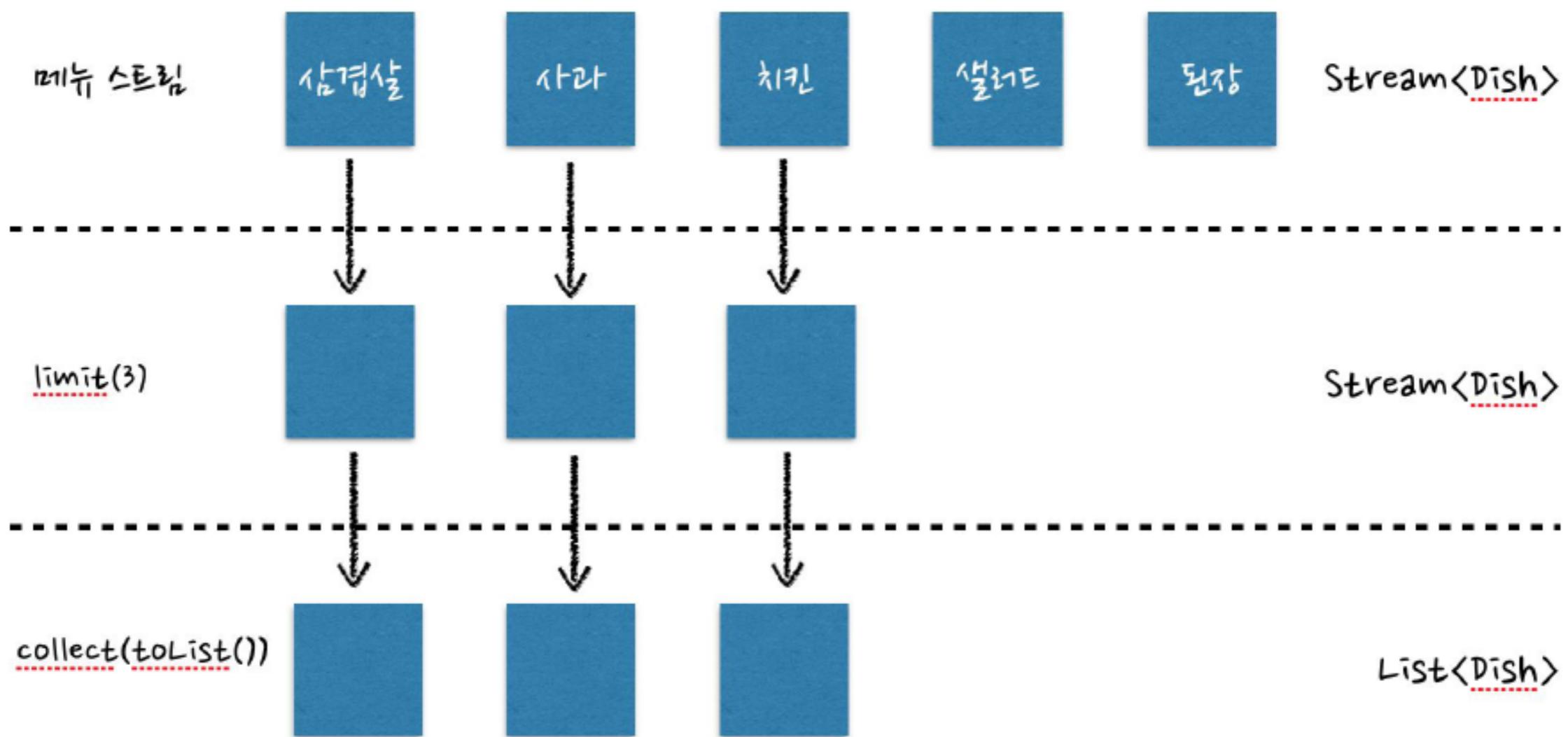
```
List<Integer> numbers = Arrays.asList(1,1,1,2,5);
numbers.stream()
    .distinct()
    .forEach(System.out::println);
```



# 스트림 API의 활용 – 필터링/슬라이싱

- limit (중간연산) : 지정된 숫자 만큼 반환한다.

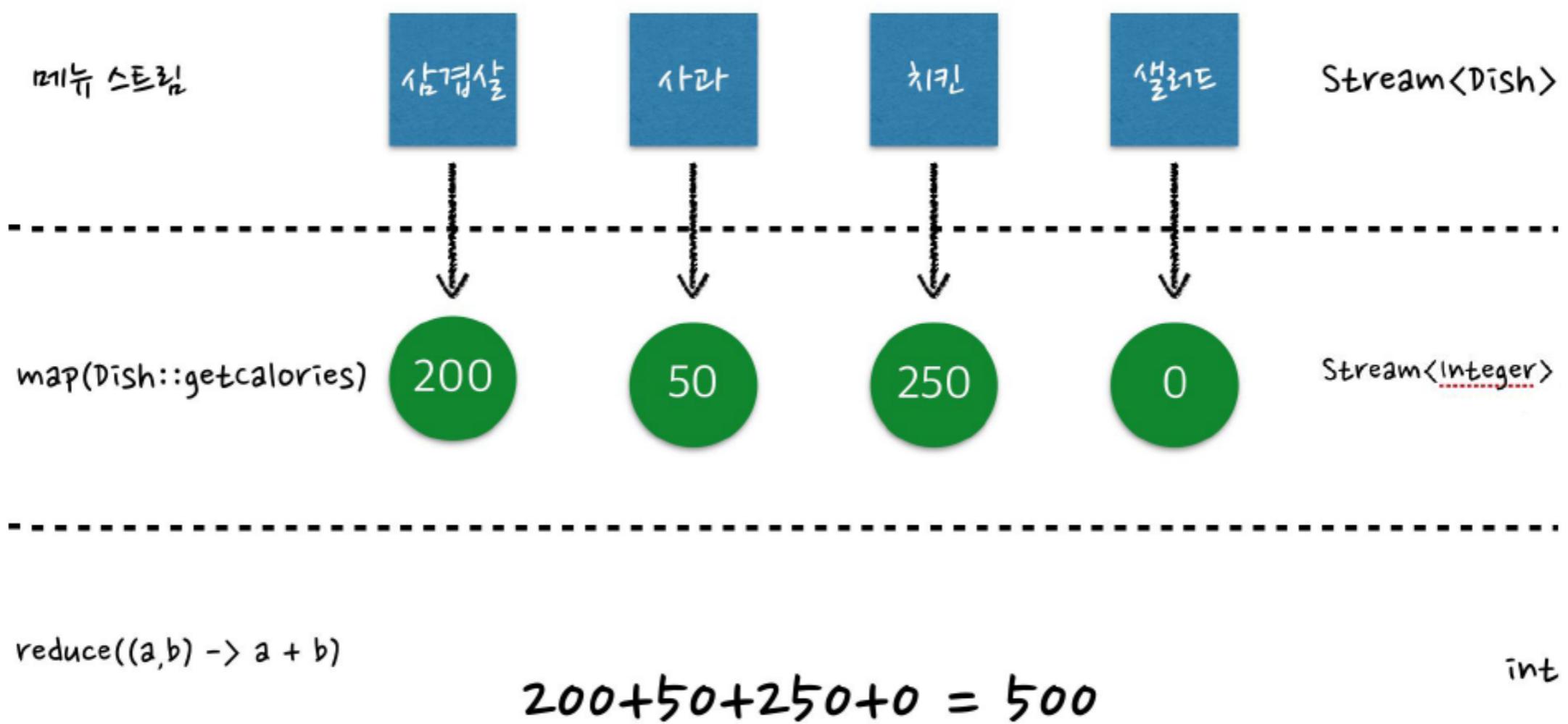
```
menu.stream()  
    .limit(3)  
    .collect(Collectors.toList());
```



# 스트림 API의 활용 – 매핑

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

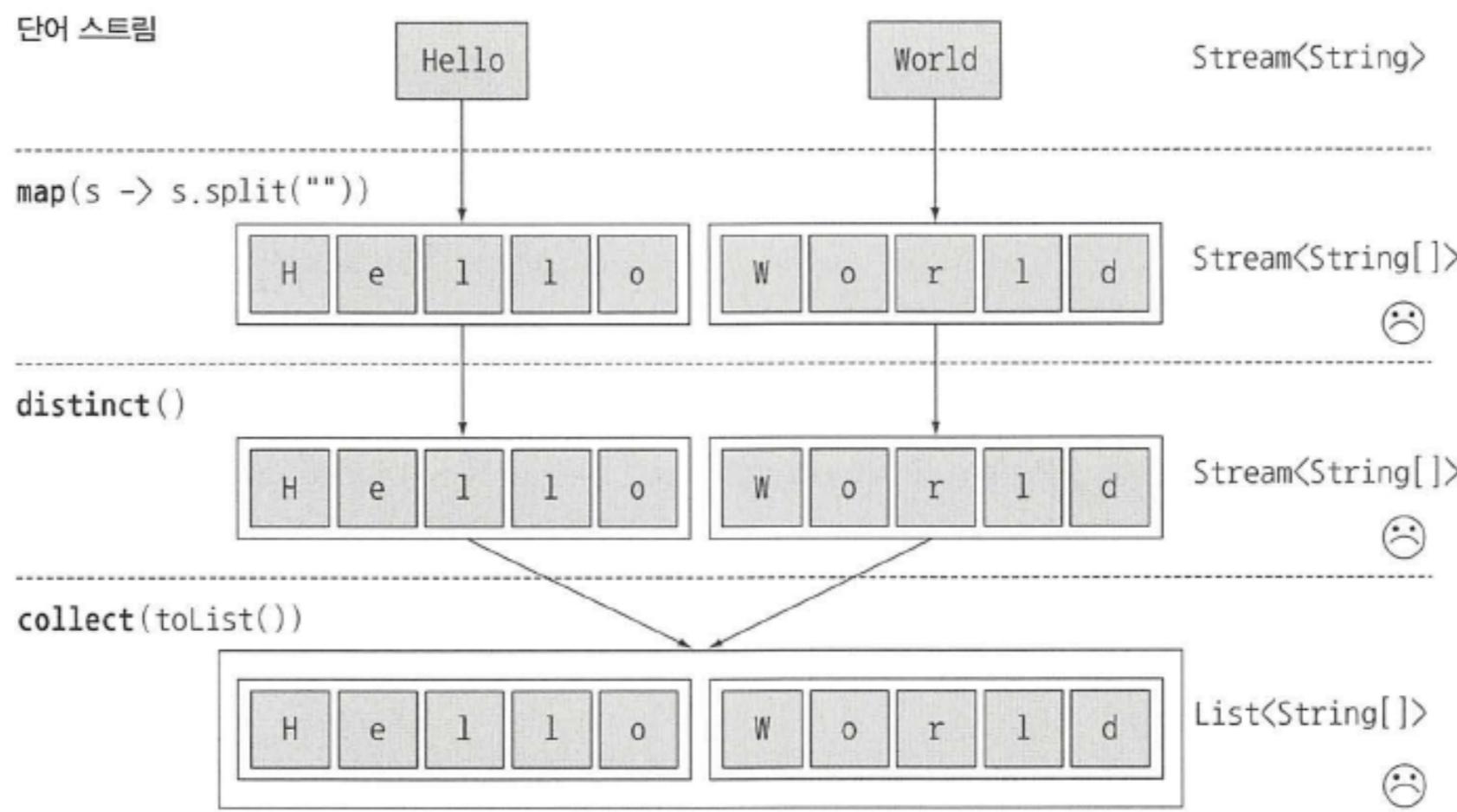
```
menu.stream()
    .map(Dish::getCalories)
    .reduce((prev,curr) -> prev + curr);
```



# 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());
```

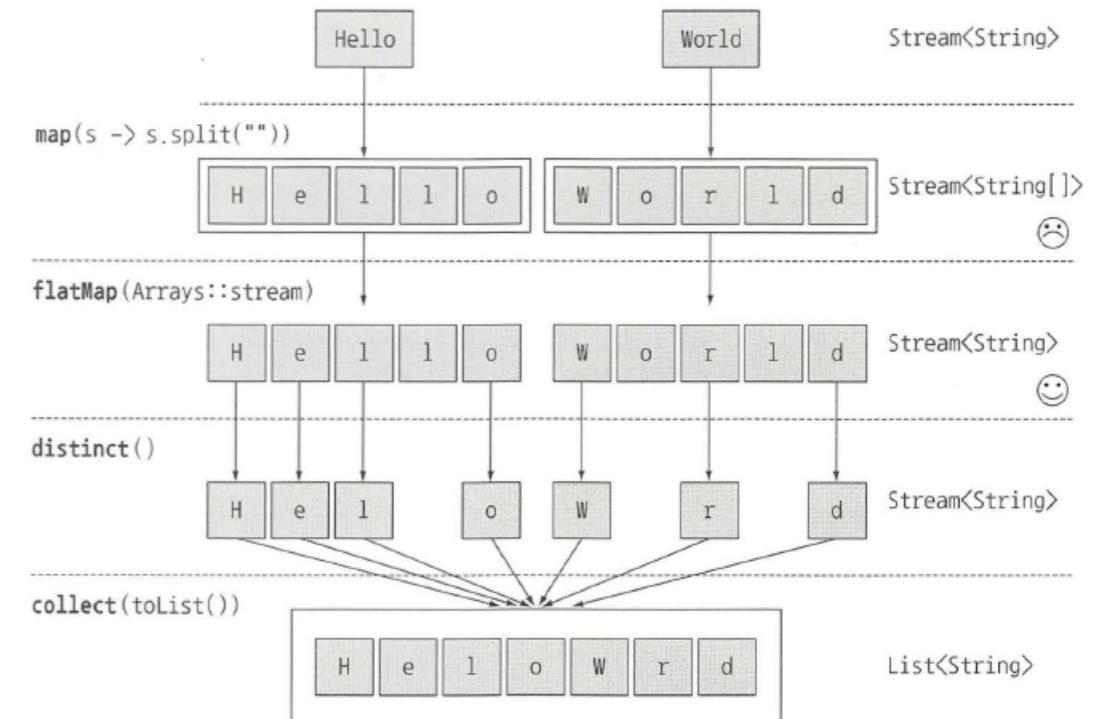


# 스트림 API의 활용 – 매핑

## ■ 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);

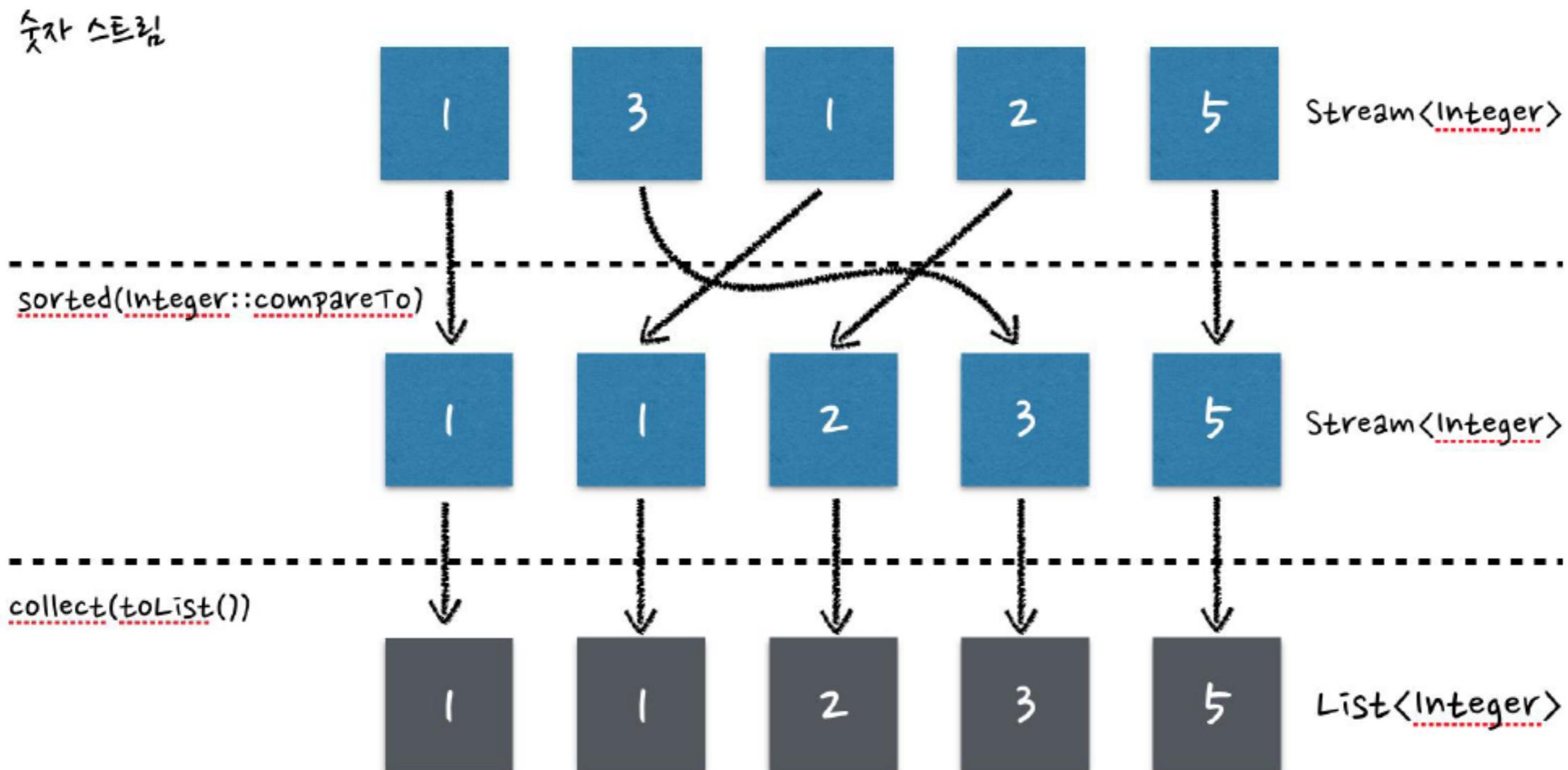
words.stream()
    .flatMap((String line) ->
        Arrays.stream(line.split( regex: "")))
    .distinct()
    .forEach(System.out::println);
```



# 스트림 API의 활용 – 검색과 매칭

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

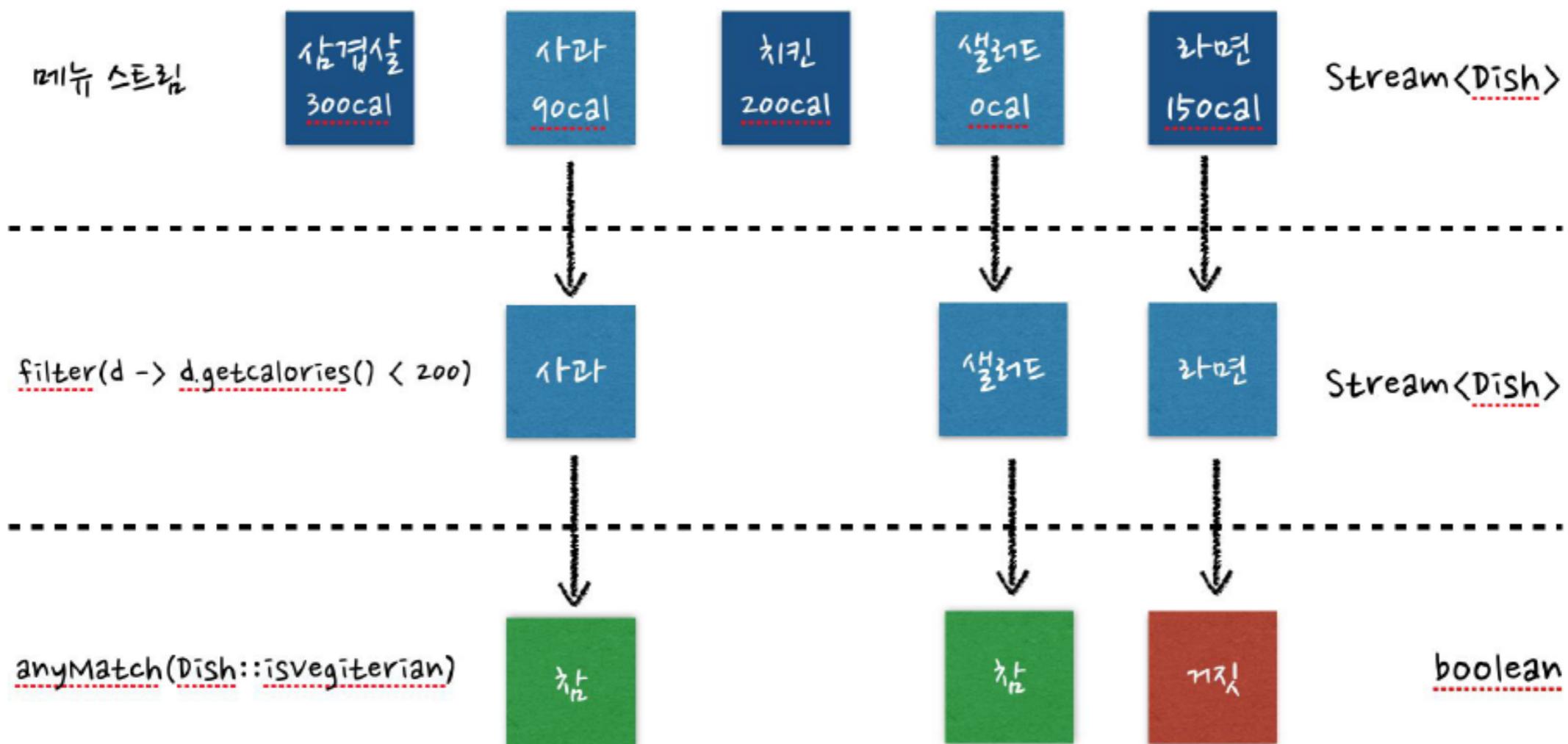
```
Arrays.asList(1,3,1,2,5)
    .stream().sorted(Integer :: compareTo);
```



# 스트림 API의 활용 – 검색과 매칭

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

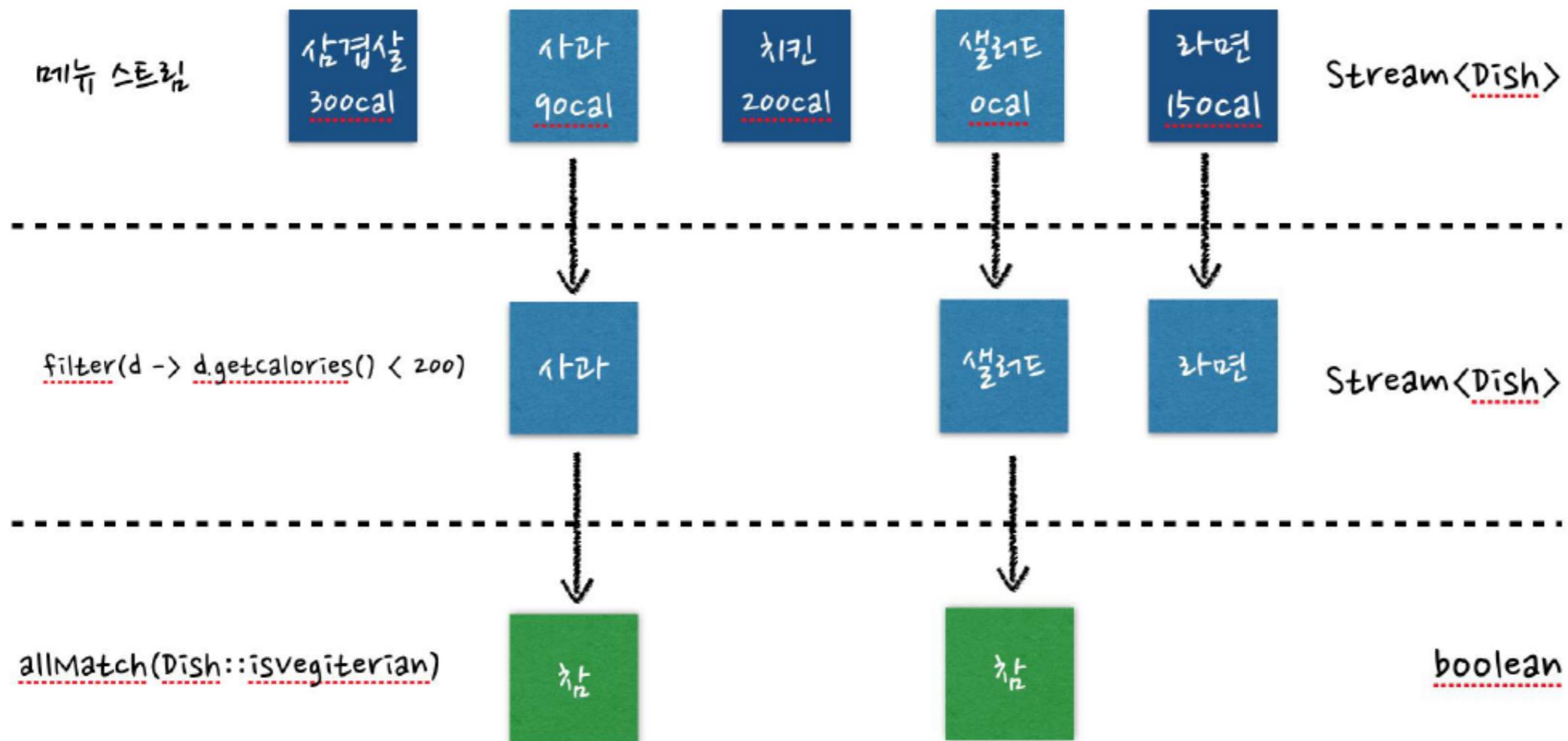
```
menu.stream()
    .filter(d -> d.getCalories() < 200 )
    .anyMatch(Dish::isVegetarian);
```



# 스트림 API의 활용 – 검색과 매칭

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

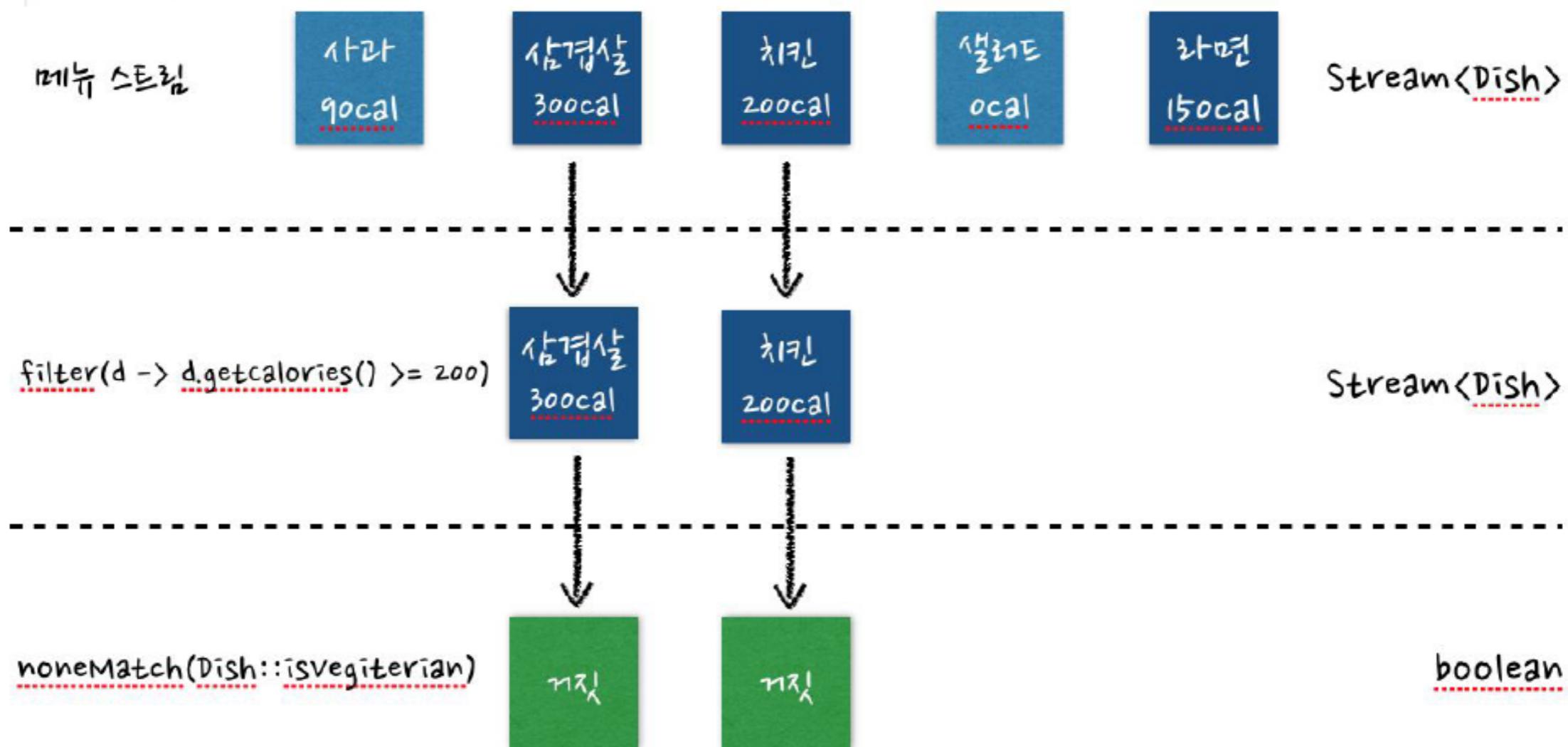
```
menu.stream()
    .filter(d -> d.getCalories() < 200 )
    .allMatch(Dish::isVegetarian);
```



# 스트림 API의 활용 – 검색과 매칭

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

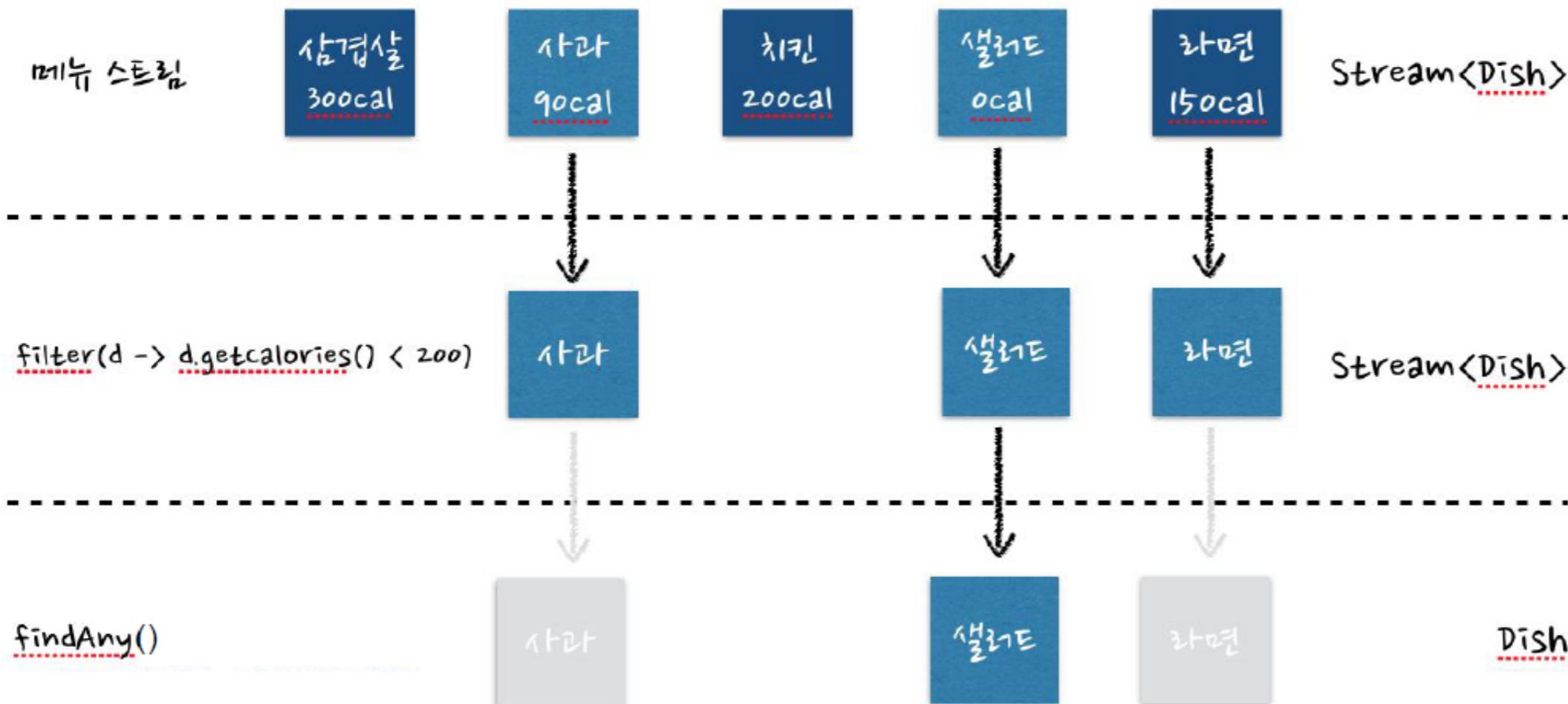
```
menu.stream()
    .filter(d -> d.getCalories() >= 200)
    .noneMatch(Dish::isVegetarian);
```



# 스트림 API의 활용 – 검색과 매칭

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

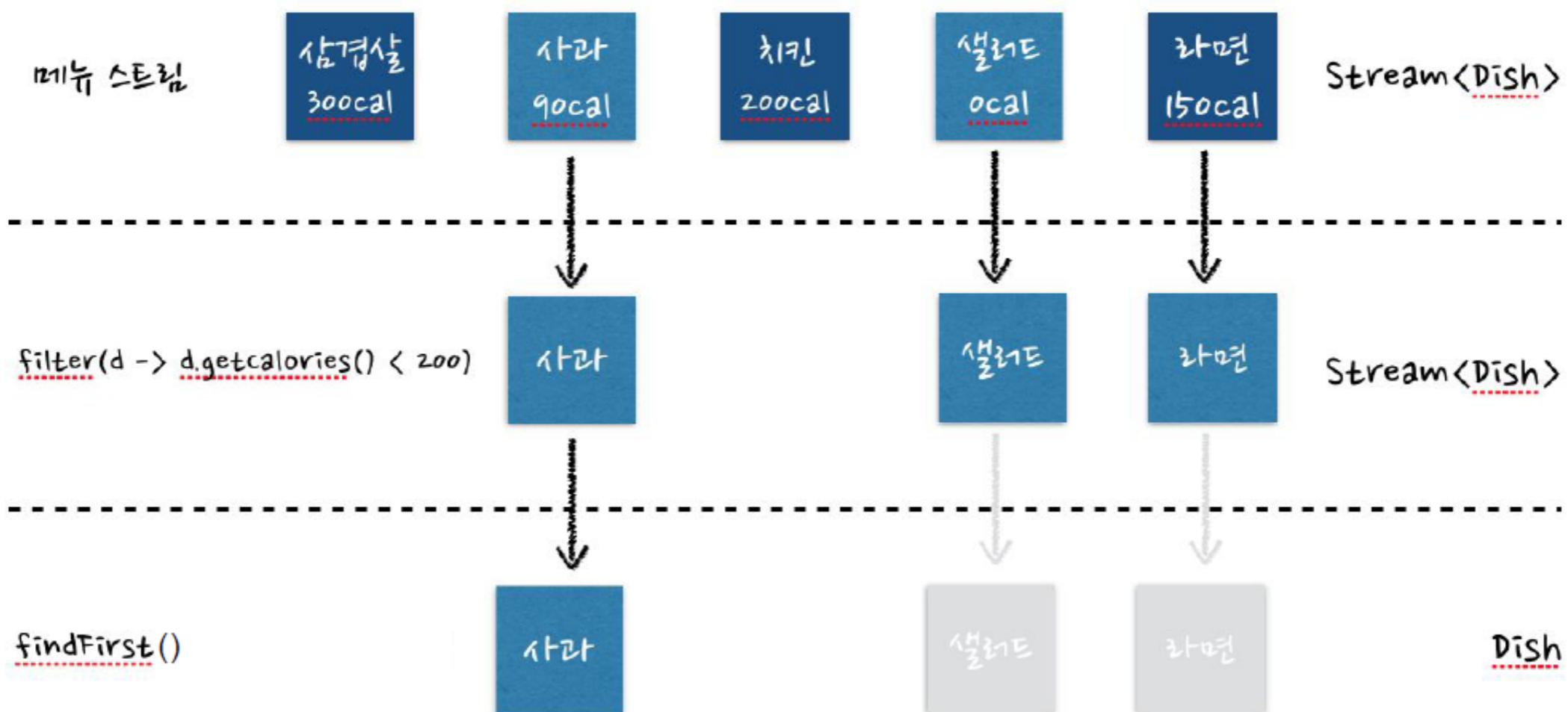
```
menu.stream()
    .filter(d -> d.getCalories() < 200)
    .findAny();
```



# 스트림 API의 활용 – 검색과 매칭

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

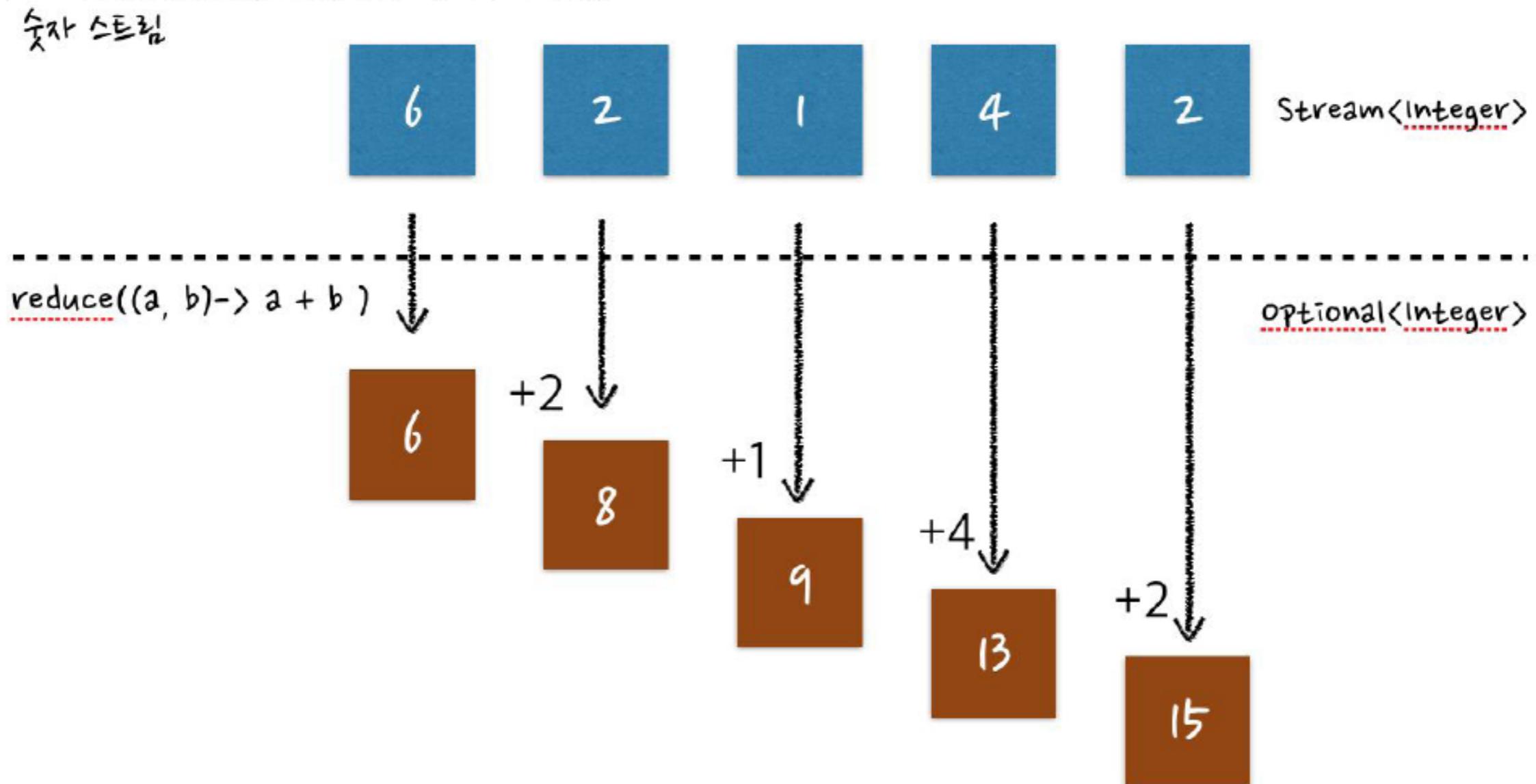
```
menu.stream()  
.filter(d -> d.getCalories() < 200)  
.findFirst();
```



# 스트림 API의 활용 – 검색과 매칭

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

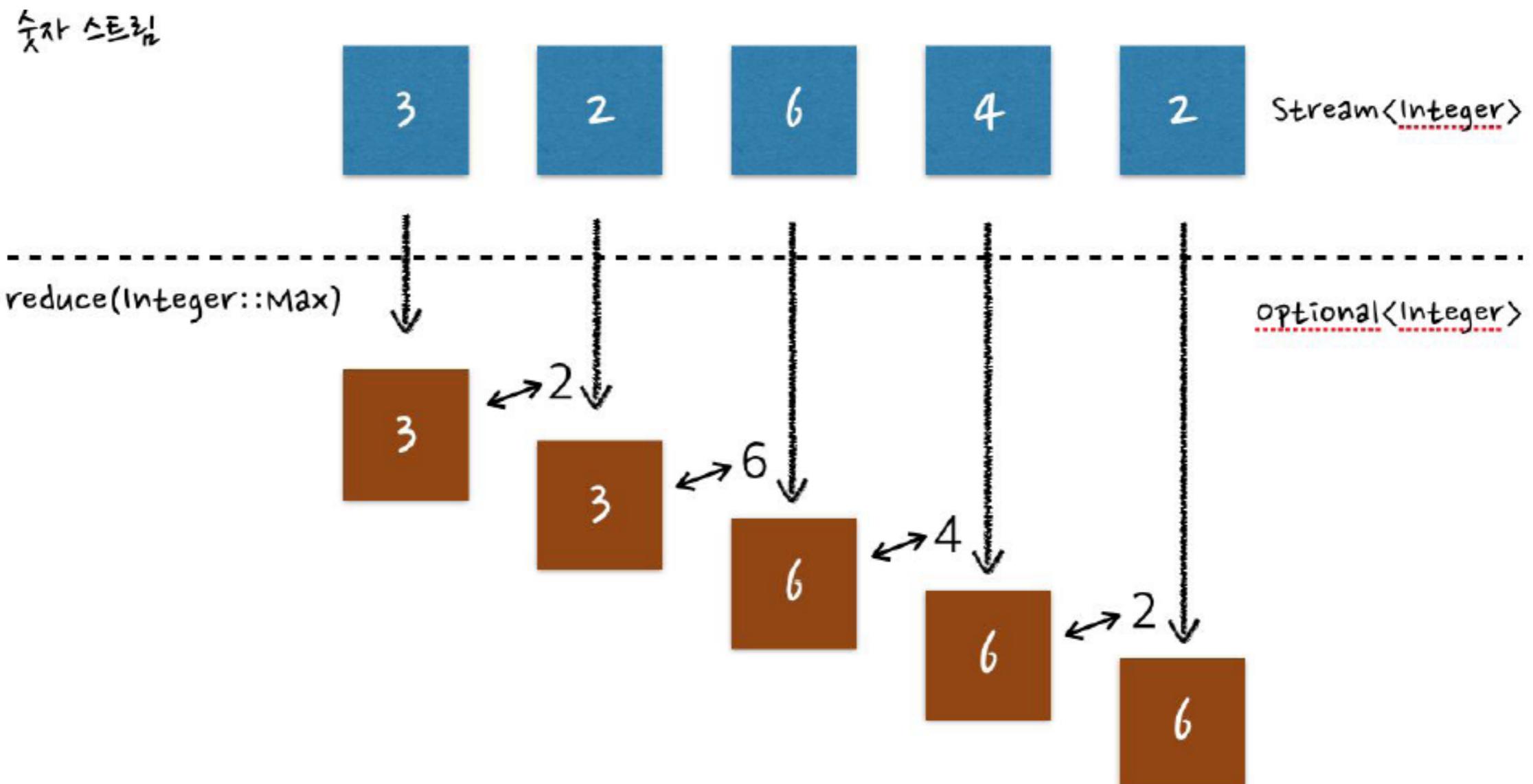
```
Arrays.asList(6,2,1,4,2)
    .stream().reduce(0,(a,b) -> a + b);
```



# 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);
```

# 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.

2부

# Reactive 프로그래밍 이란?

# 리액티브 프로그래밍

---

## ■ 리액티브 프로그래밍 등장 배경

- 빅데이터 :
  - ▶ PB 단위로 구성되고 매일 증가됨
- 다양한 환경 :
  - ▶ 모바일 디바이스 부터 수천개의 멀티 코어 클라우드 클러스터 까지
- 서비스 사용 패턴의 변환 :
  - ▶ 24/365 서비스와 ms 단위의 응답시간 기대

**여러 API를 취합해서 전달해야하는 시스템에서는 SUM([각 API들의 경과시간]) 만큼 필요합니다.**

**반대로 리액티브로 진행할 경우, 여러 API 중 MAX([각 API들의 경과시간]) 이 필요합니다.**

# 리액티브 프로그래밍

---

- 데이터 흐름과 전달에 관한 프로그래밍 패러다임
  - 명령형(imperative) 방식 보다 선언적(declarative) 방식
- 명령형 프로그래밍 : pull 방식
- 리액티브 프로그래밍 : push 방식
- 시스템에 이벤트가 발생했을 때 처리하는 방식
  - 직접 처리하는 것이 아니라..
  - 이벤트 드리븐 프로그래밍 모델이라고 함

# Reactive 시스템

---

- 마이크로 서비스: 다양한 선택
  - 하이브리드 멀티 클라우드 인프라
  - 마이크로 서비스 아키텍처
  - 독립적 팀과 일정을 통한 독립적 개발 / 테스트 / 배포
  
- 리액티브 시스템 적용범위
  - 웹 / 모바일 커머스
  - 데이터 기반 결정

# Reactive 시스템 정의

---

- 2013 Reactive Manifesto
  - 하이브리드 멀티 클라우드 인프라
  - 마이크로 서비스 아키텍처
  - 독립적 팀과 일정을 통한 독립적 개발 / 테스트 / 배포
- 리액티브 시스템 적용범위
  - 웹 / 모바일 커머스
  - 데이터 기반 결정

# Reactive 시스템 정의

## ■ 2013년 리액티브 Manifesto

- <https://www.reactivemanifesto.org/ko>
- 2014. 09. 16 발행됨

## ■ 리액티브 시스템은

- 응답성, Responsive 일정한 응답시간을 보장
- 탄력성, Resilient 장애 발생해도 시스템은 반응
- 유연성, Elastic 부하 과다시 자동으로 자원수가 증가
- 메세지 구동, Message Driven 비동기 메시지로 컴포넌트 통신

# Reactive 프로그래밍

---

- 변화에 대한 반응을 기반으로 코드를 작성하는 방법
- 비동기 처리 파이프라인 구성을 위한 선언적 코드 사용 페러다임
  - Asynchronous, Non-blocking
  - 사용자 경험 향상과 대량의 데이터 처리
- 데이터가 발생하는 부분과 사용하는 부분을 쉽게 분리 가능
  - 쉬운 쓰레드 관리
  - 사이트 이펙트 최소화

위키피디아

리액티브 프로그래밍은 데이터 흐름(data flows)과 변화 전파에 중점을 둔 프로그래밍 패러다임(programming paradigm)이다.

이것은 프로그래밍 언어로 정적 또는 동적인 데이터 흐름을 쉽게 표현할 수 있어야하며, 데이터 흐름을 통해 하부 실행 모델이 자동으로 변화를 전파할 수 있는 것을 의미한다.

# Reactive Extension history

<http://reactivex.io>

- 2007년 MS Volta 프로젝트에서 시작

- Erik Meijer



- 2009. 11. 17

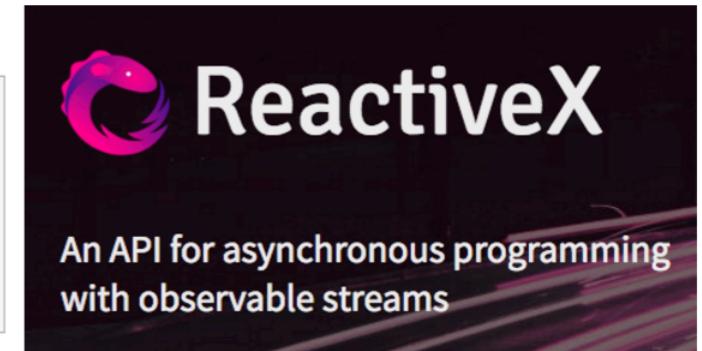
- Reactive Extensions for .NET 릴리즈

- 2010. 03. 17

- RxJS 릴리즈
  - Ajax에서 비동기 행위를 쉽게 처리가 필요함에 따라 개발됨

- 2012. 11. 06

- ReactiveX 오픈소스로



# JVM에서의 Reactive

---

- RxJava 와 Netflix
  - 2012년 북미 인터넷 트래픽 33% 점유(유튜브 15%)
- JVM 환경에서 Reactive Extensions 동작 모델
  - RxJava 1.0 - Jafer Husain in Netflix (2014.11)
- 자바에서의 Reactive 표준
  - Typesafe, Red Hat, Netflix, Pivotal, Oracle, Twitter, [spray.io](http://spray.io) 등
  - Reactive 표준인 reactive streams 정의
- RxJava 2.0 (2016.11)
  - Reactive streams 지원, 2MB 정도로 매우 가벼움



# Reactive Streams

---

<https://www.reactive-streams.org>

- 리액티브 라이브러리의 표준 사양

- 데이터 스트림을 비동기로 다룰 수 있는 공통 메카니즘
- 표준 인터페이스를 제공

- 4개의 인터페이스를 정의

- Subscriber
- Publisher
- Subscription
- Processor(Subscriber, Publisher를 상속)

<https://github.com/reactive-streams/reactive-streams-jvm/>

**동기와 비동기**

# 비동기란?

---

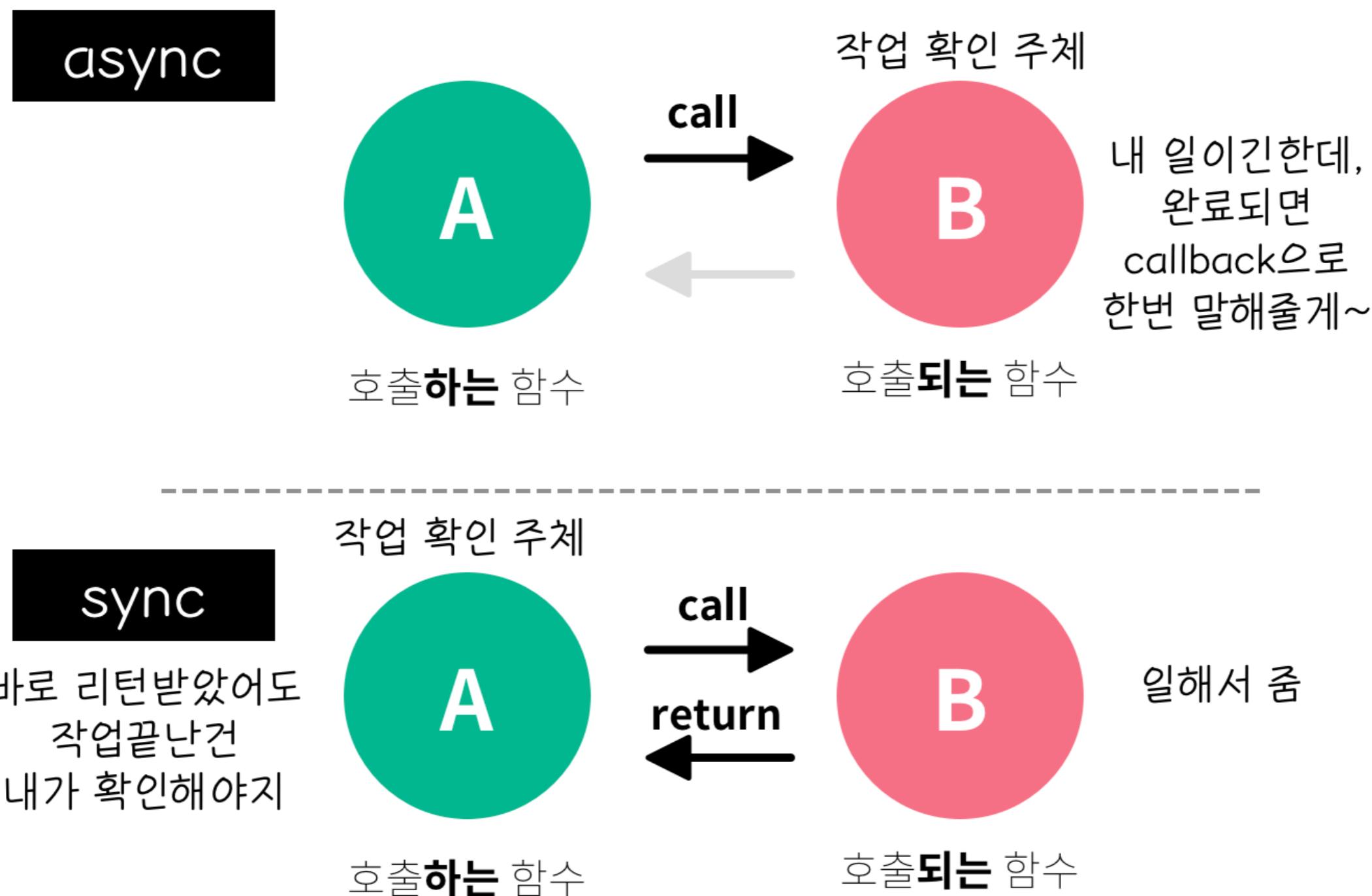
## ■ Synchronous / Asynchronous

- 함수 통신 메카니즘
- 동기는 함수를 호출했을 때 결과가 나오기 까지 어떤 것도 리턴하지 않는다.
- 비동기는 함수가 완료될 때 까지 결과를 기다리지 않는다.
  - ▶ 콜백과 같은 형태로 최종 결과값을 알려줌 (다시 알려준다)

## ■ Blocking / Non-Blocking (IO)

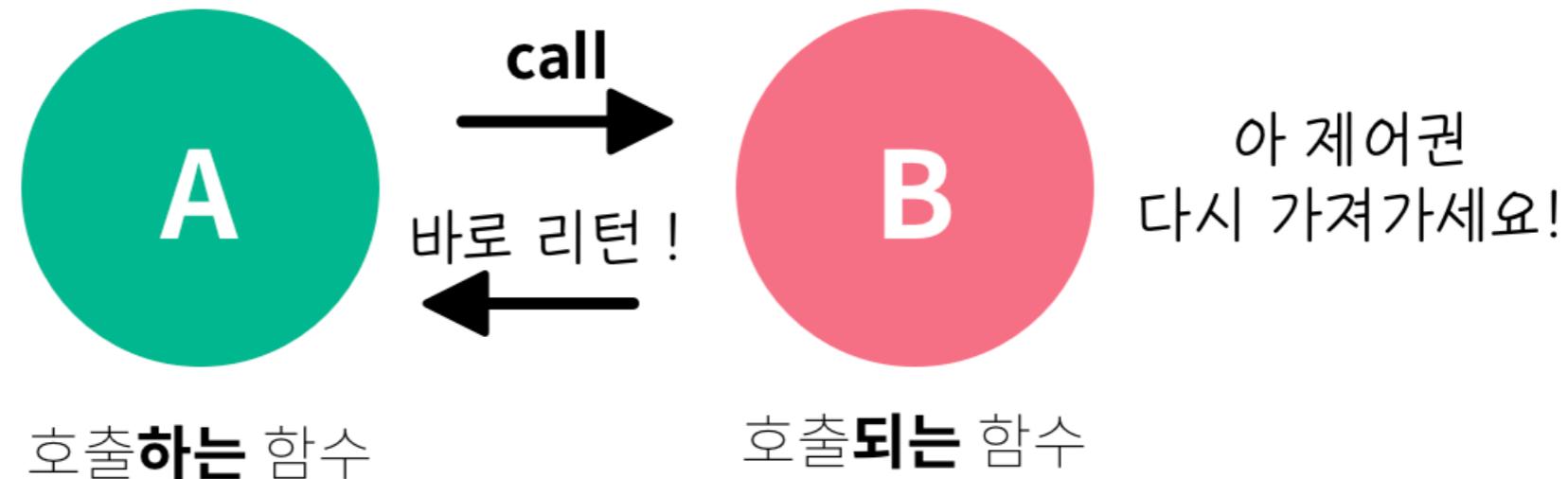
- 함수의 동기/비동기와 상관없이 호출 주체 즉, 프로그램의 상태에 대한 것
- 함수를 호출하고 현재 쓰레드가 결과를 받을 때까지 잠금 상태
  - ▶ 다른 쓰레드를 생성하여 다른 작업을 수행할 수 있음
- 함수를 호출하고 현재 쓰레드에서 다른 작업을 수행 가능
  - ▶ 즉, 쓰레드가 잠금상태가 되지 않음

# sync / async

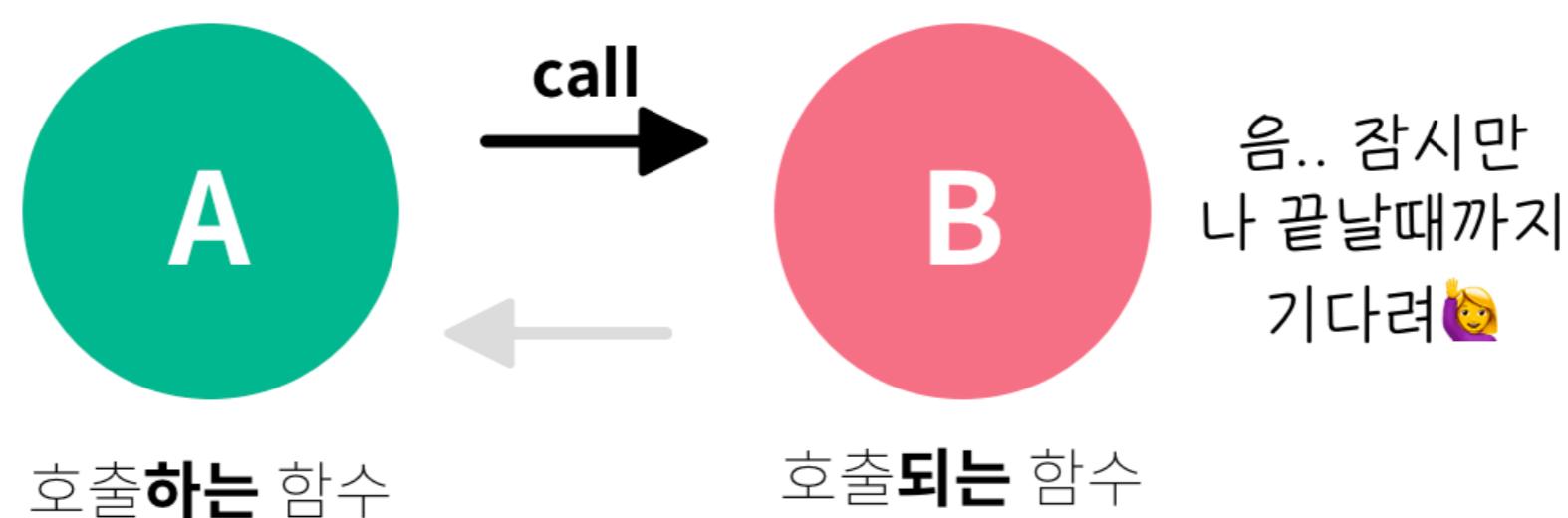


# Non-blocking / Blocking

논블로킹



블로킹



# 유스케이스

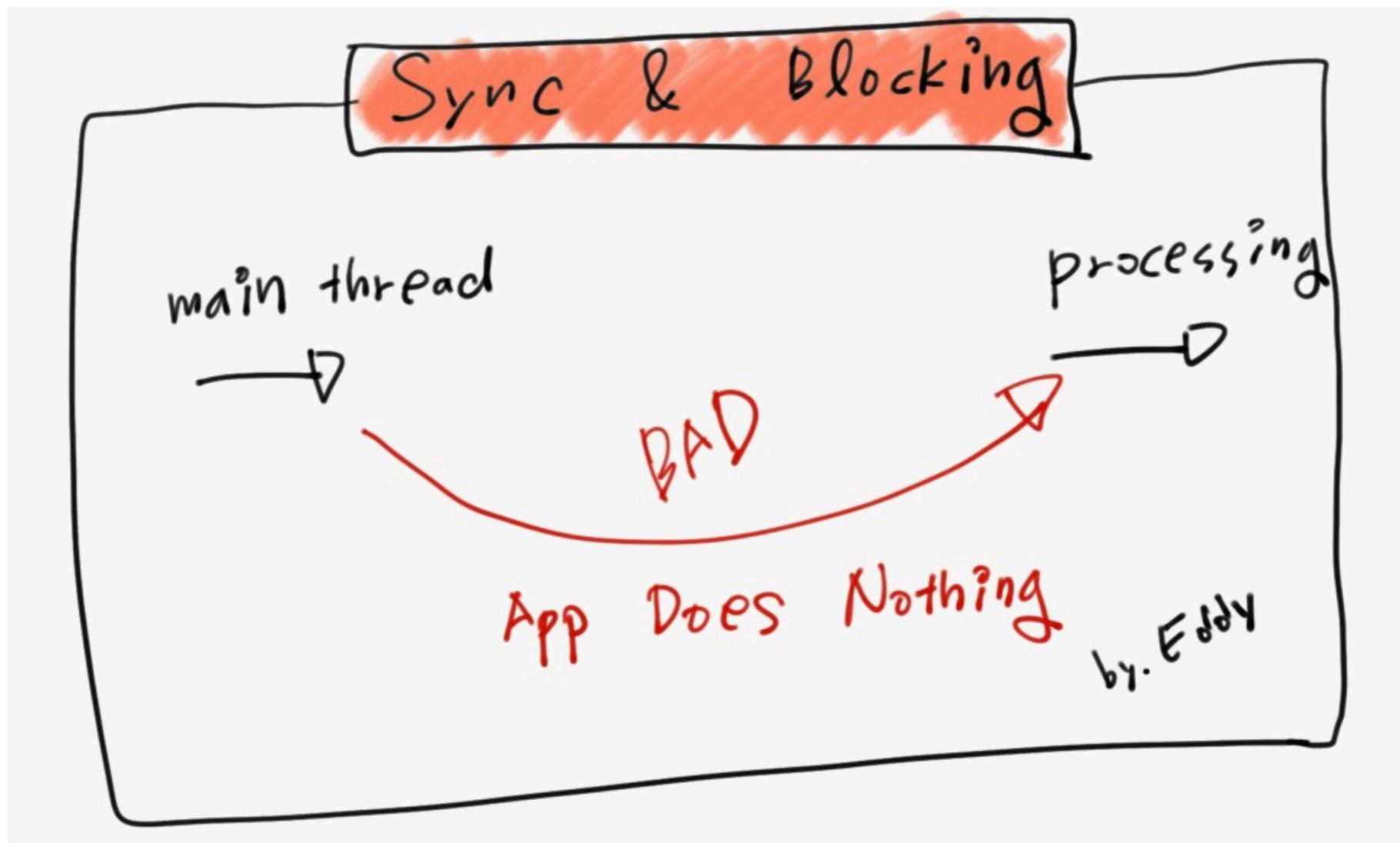
---

- Async 일 때 반드시 Non-Blocking 은 아님
  - Sync & Blocking
  - Sync & Non-Blocking
  - Async & Blocking
  - Async & Non-Blocking

# Sync & Blocking

- 동기 & 블록킹 환경

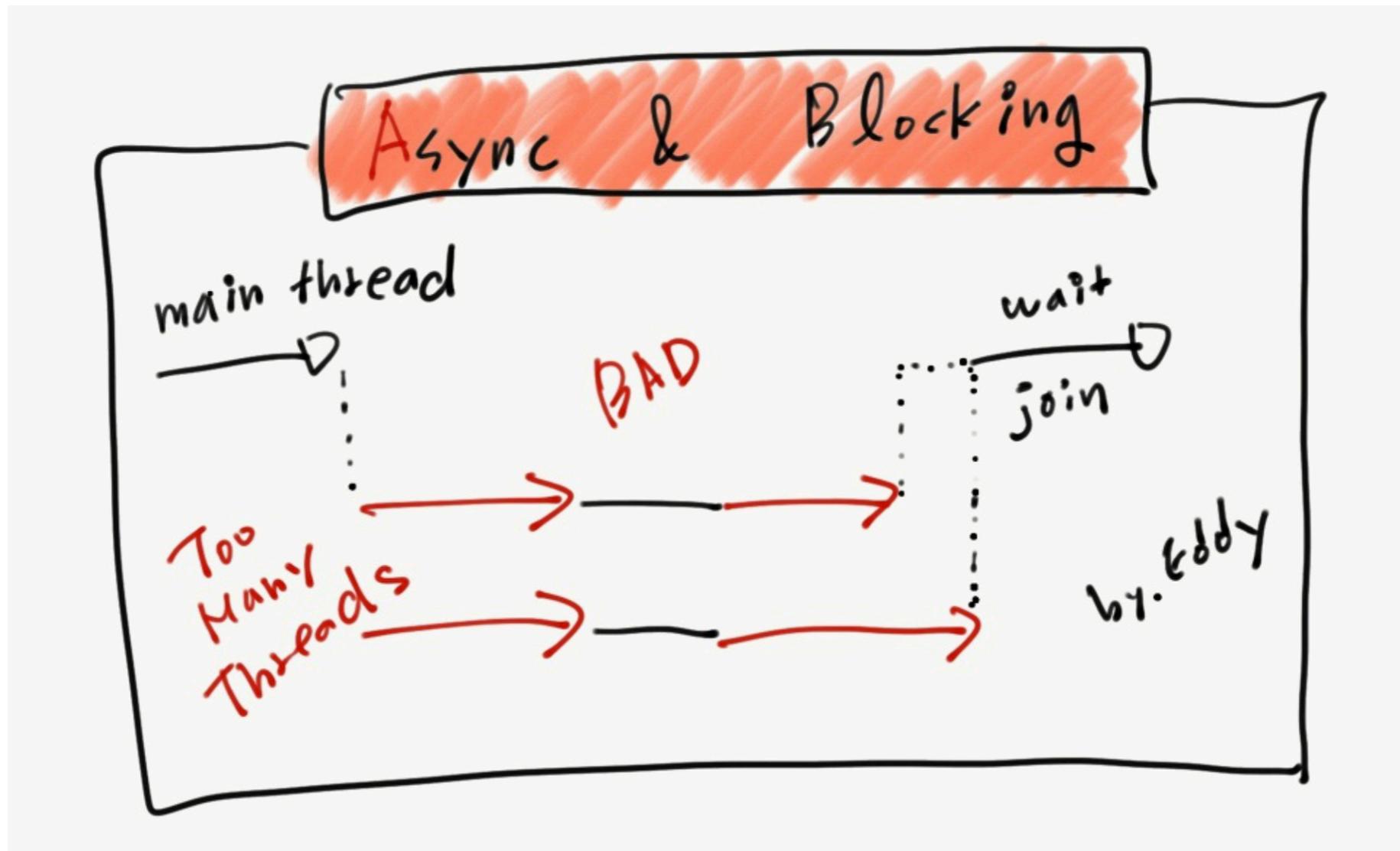
- 작업을 수행 중 일때 다른 작업을 병행할 수 없다.



# Async & Blocking

- 비동기 & 블록킹 환경

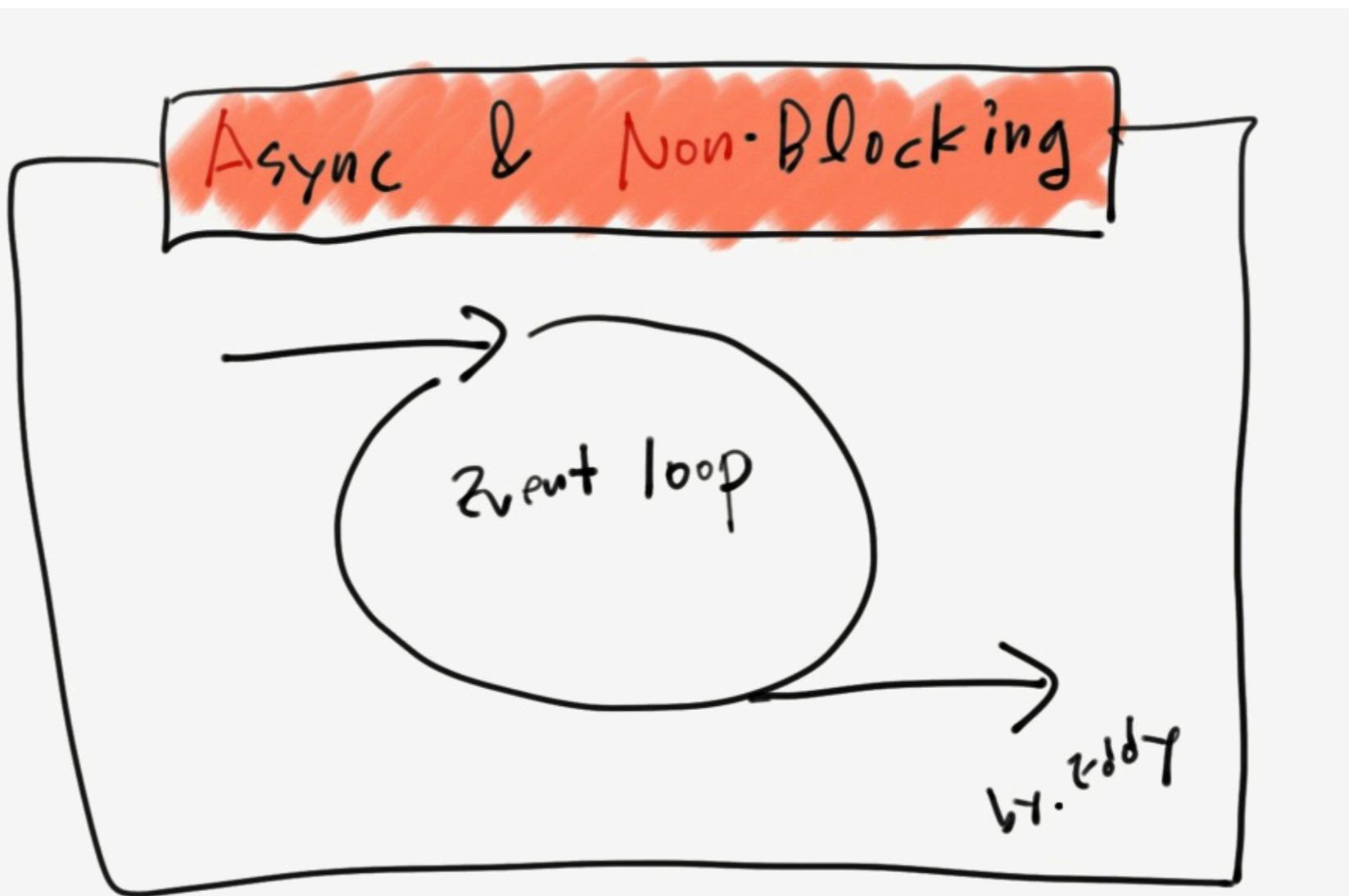
- 쓰레드를 추가로 생성하여 다른 작업 병행 가능



# Async & Non-Blocking

## ■ 비동기 & 논블록킹 환경

- 하나의 쓰레드로 여러 작업 수행 가능
- 현재의 자원에서 더 높은 효율성을 추가하는 방법



# 자바에서의 Async & Non-Blocking

---

- Node.js 가 Async & Non-Blocking 방식으로 서비스 개발 가능
- JVM 환경에서 RxJava, Reactor 등
  - Reactive streams 구현체를 사용하면 가능

# **Reactive Streams**

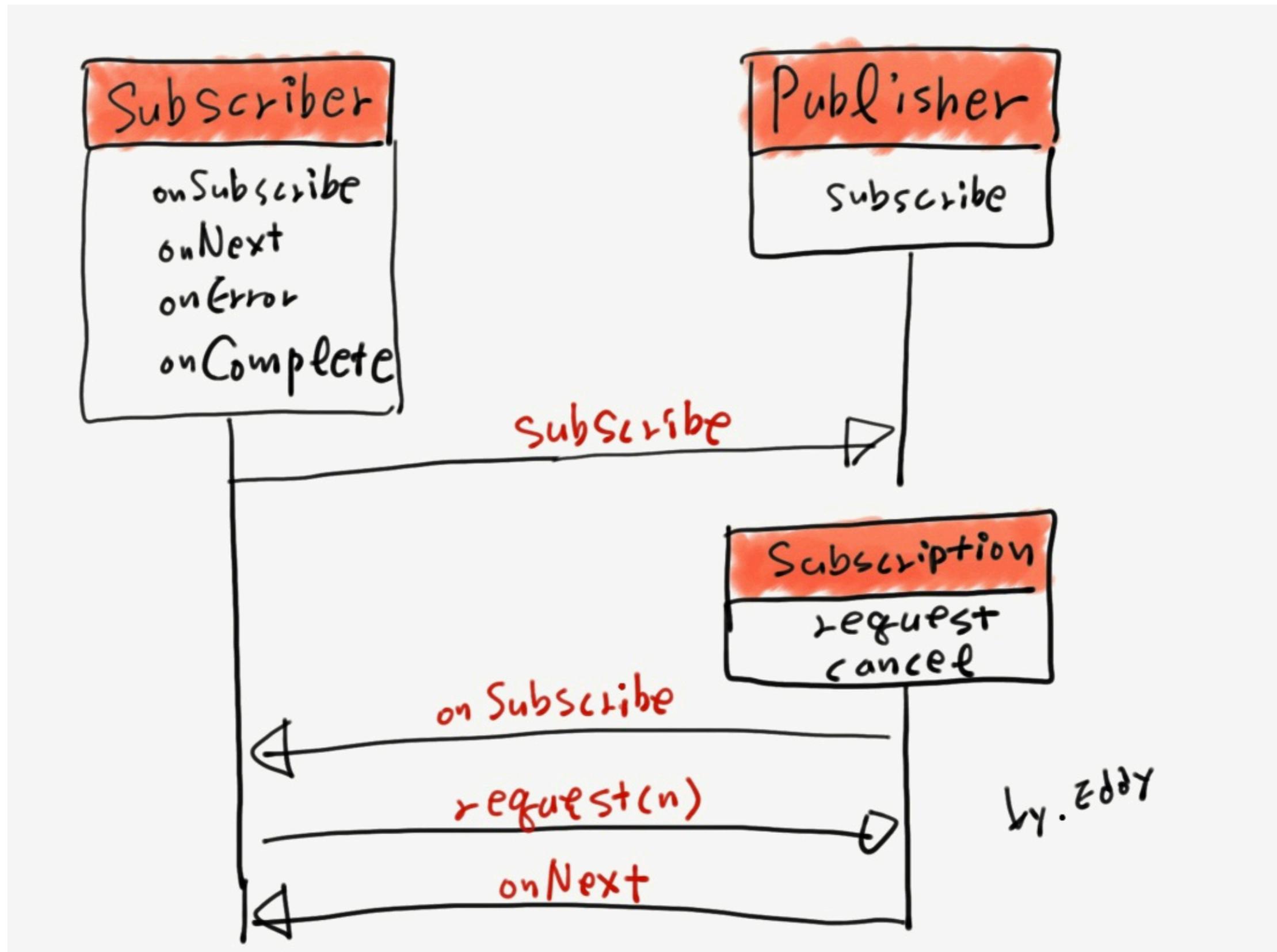
# Reactive Streams

---

- 2013년 Netflix, Pivotal, Typesafe 의 개발자들이 발의해서 만들어짐
  - 리액티브 프로그래밍을 위한 명세(Specification)
  - Reactive Streams 에 정의된 인터페이스를 구현하면 리액티브 프로그래밍을 구현할 수 있다.
- Reactive Streams Interface
  - Processor
  - Publisher
  - Subscriber
  - Subscription

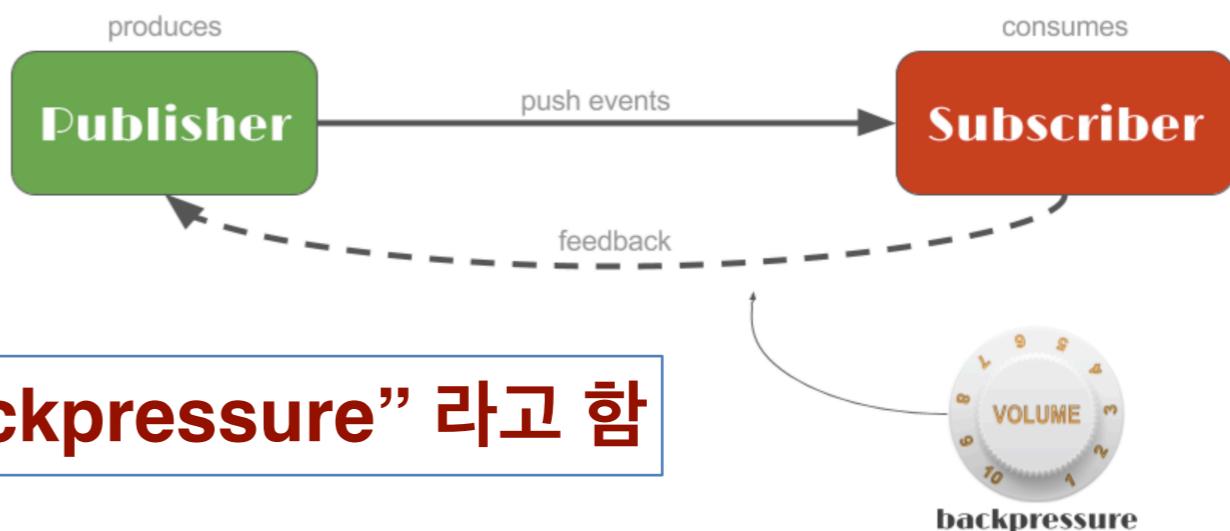
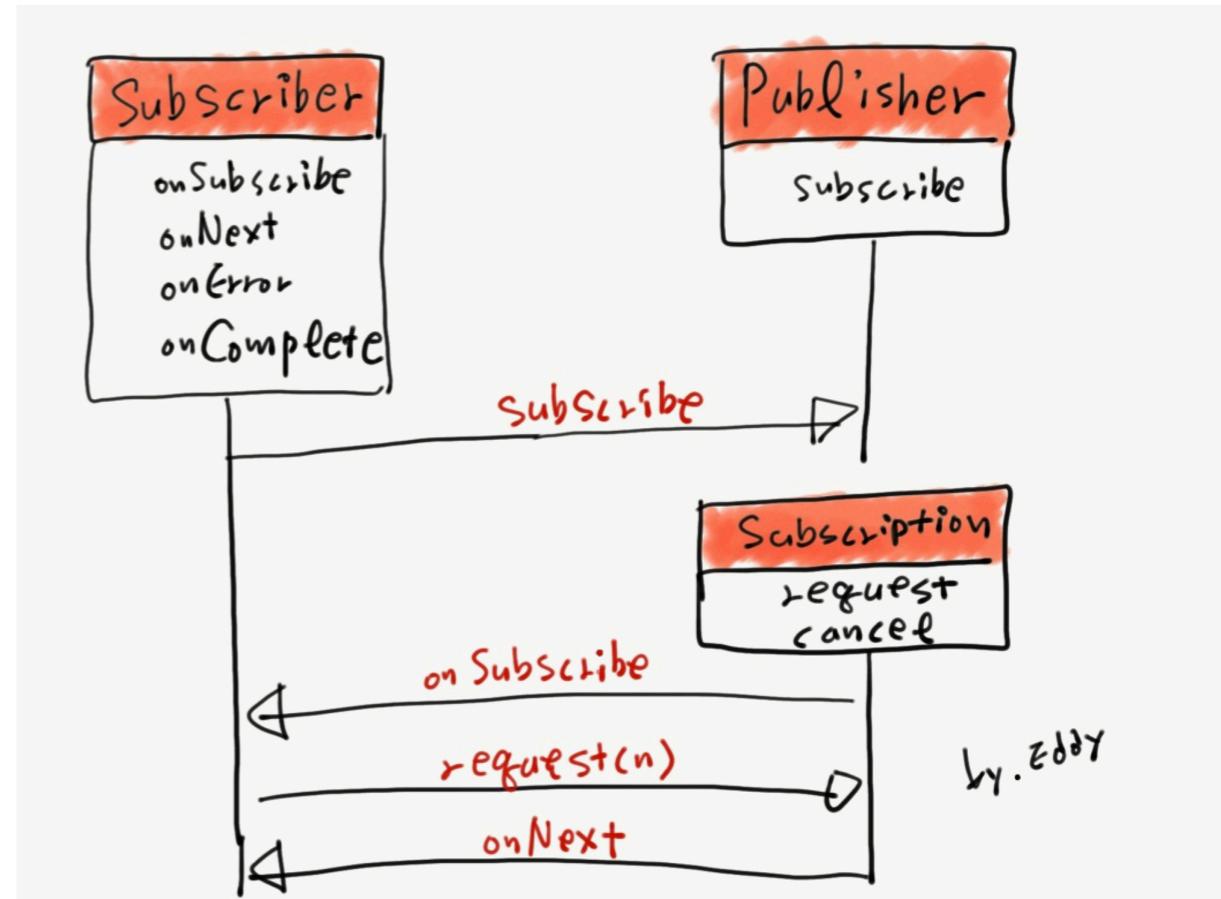
<https://mvnrepository.com/artifact/org.reactivestreams/reactive-streams>

# Reactive Streams



# Backpressure, 역압

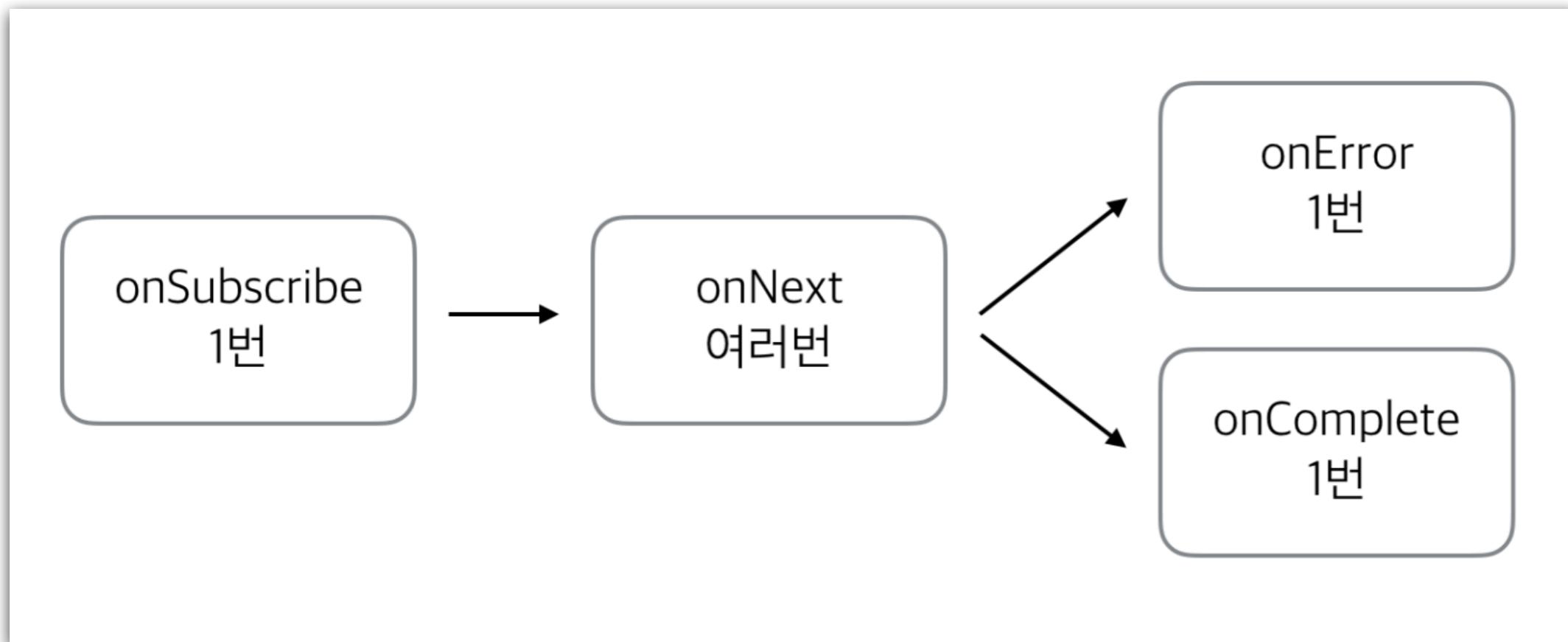
- Subscriber 가 Publisher 에게 데이터 또는 시퀀스 요청
- Publisher 는 Subscriber 에게 onSubscribe() 호출
- Subscriber 는 request(n) 호출
  - Subscription 사용
  - Publisher 는 0..N 개 시퀀스를 Subscriber 에게 전달
- 에러가 발생되면 onError() 호출
- 전달이 완료되면 onComplete() 호출



**request(n) 호출을 “Backpressure” 라고 함**

# Reactive Streams 규칙 (Protocol)

- 구독 시작 통지 (onSubscribe)는 해당 구독에서 한 번만 발생
- 통지 (onNext)는 순차적으로 여러번, 또는 한번도 발생하지 않을 수 있음
  - null을 통지 하지 않는다.
- Publisher 의 처리는 완료 (onComplete) 또는 에러 (onError)를 통지해 완료



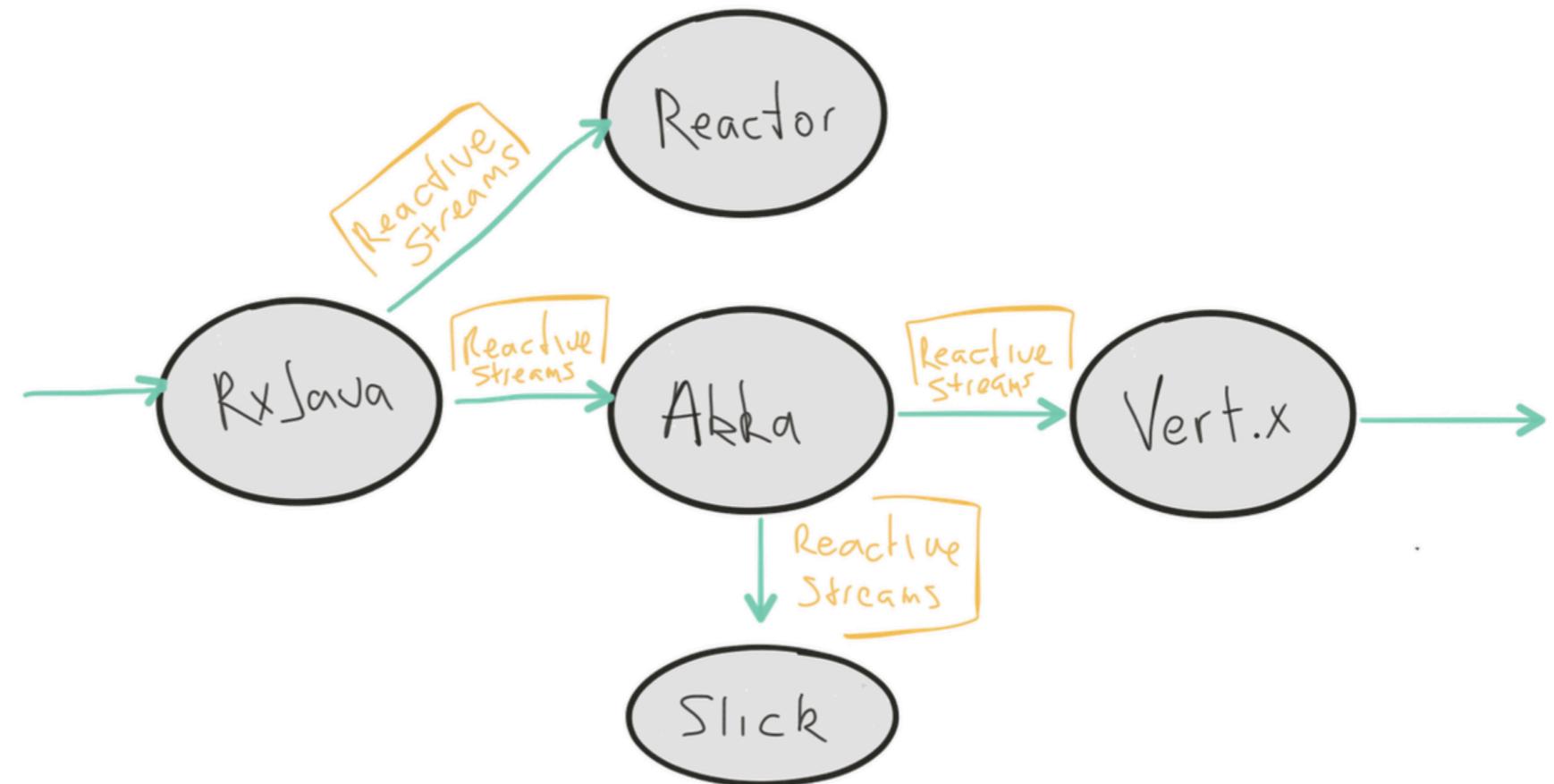
onSubscribe onNext\* (onError | onComplete) ?

# Reactive Streams 구현체

<https://www.reactive-streams.org>

## ■ reactive streams 구현체

- RxJava 2, 3
- Project Reactor (스프링에서 사용)
- Java9 flow
- Akka Streams
- Vert.x
- Ratpack
- Slick



# **개발 환경 설정**

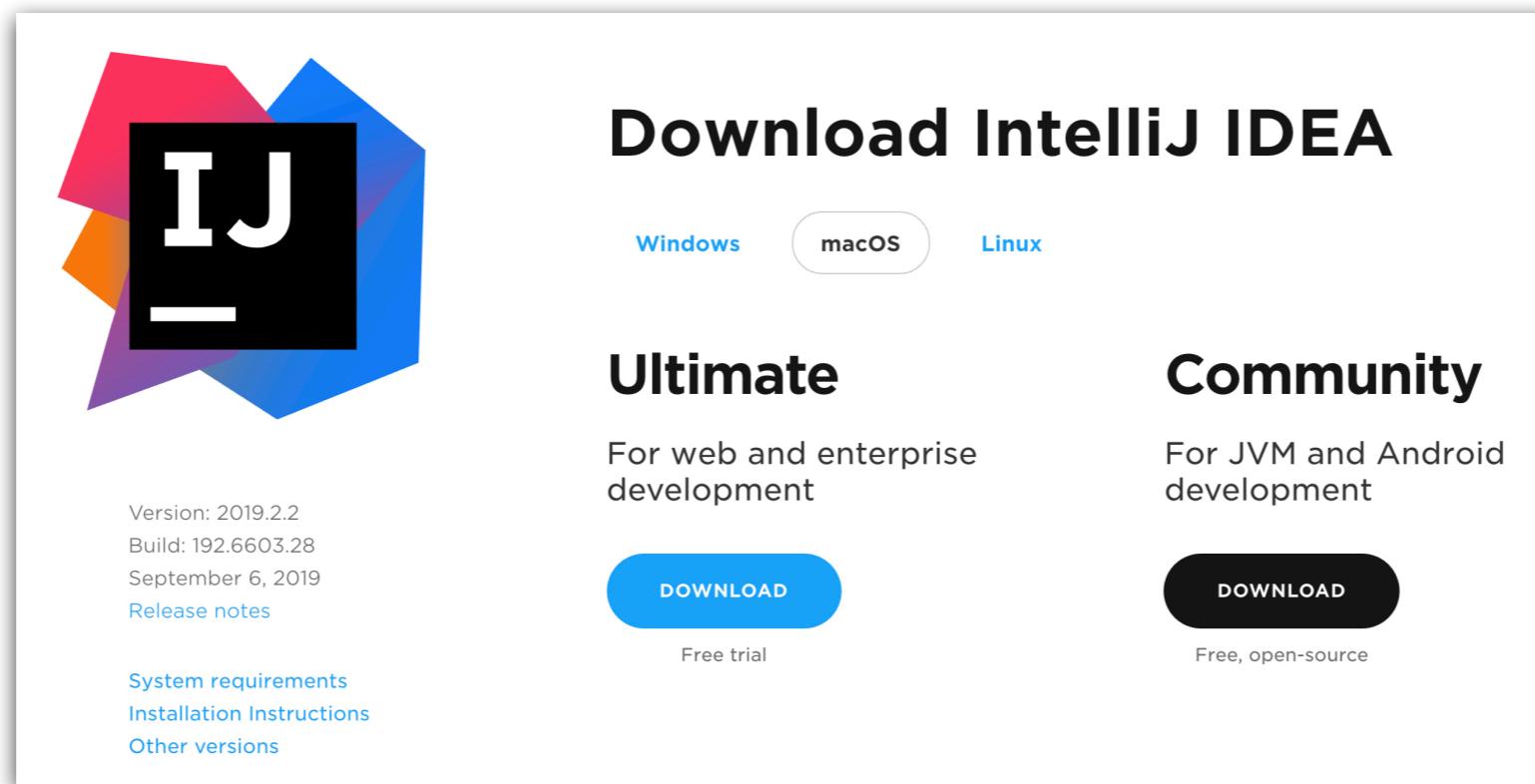
# JDK8 & IDE

## ■ JDK 8 이상 설치 (권장)

- JDK6에서도 동작되나 함수형 프로그래밍을 사용할 수 없음
- <https://www.oracle.com/technetwork/java/javase/downloads/index.html>

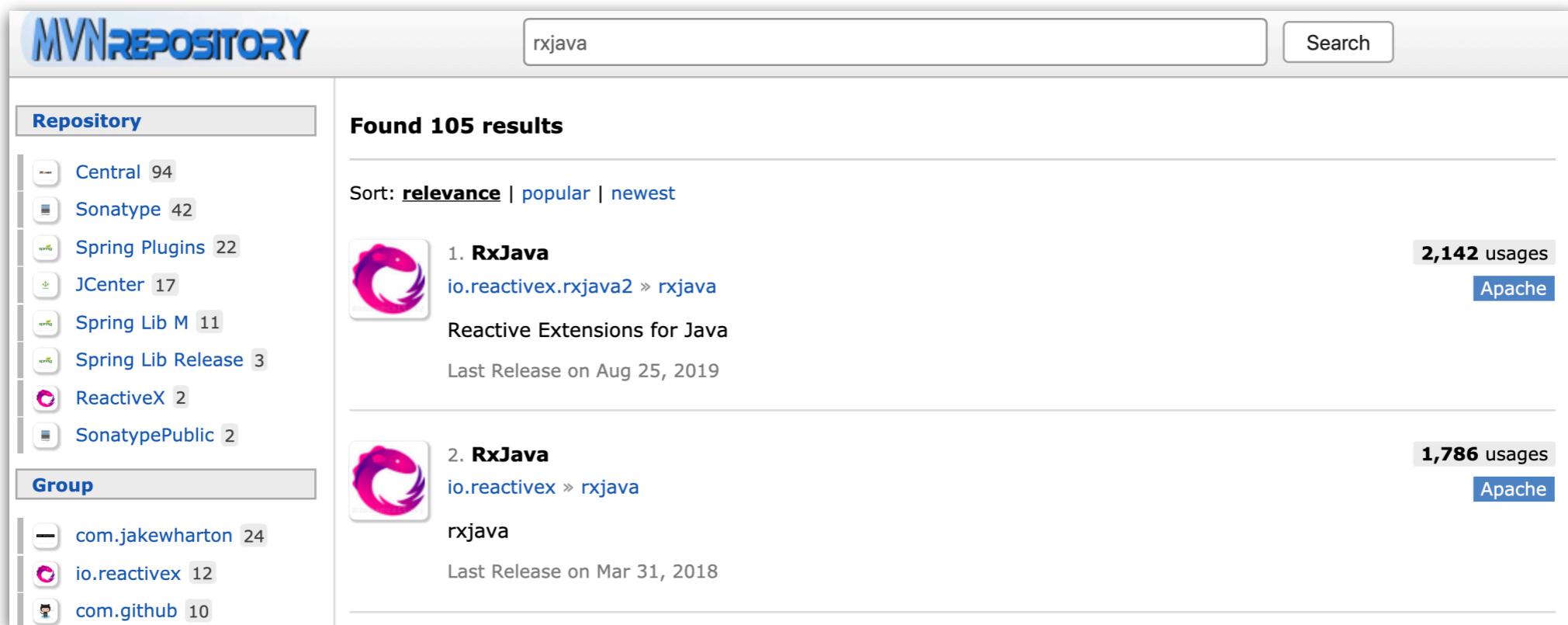
## ■ IntelliJ IDEA CE

- <https://www.jetbrains.com/idea/>



# RxJava 3

- RxJava 3 (RxJava 2 와 유사)
  - 매우 가벼움, 4MB 정도
  - Reactive Streams 하나만 의존성 가짐
  - 메이븐 central repository 로 설치



# RxJava 첫 코드

```
import io.reactivex.Observable;
public class Launcher {
    public static void main(String[] args) {
        Observable<String> myStrings =
            Observable.just("Alpha", "Beta", "Gamma", "Delta", "Epsilon");
    }
}
```

```
import io.reactivex.Observable;

public class Launcher {
    public static void main(String[] args) {
        Observable<String> myStrings =
            Observable.just("Alpha", "Beta", "Gamma", "Delta", "Epsilon");

        myStrings.subscribe(s -> System.out.println(s));
    }
}
```

```
import io.reactivex.Observable;

public class Launcher {
    public static void main(String[] args) {

        Observable<String> myStrings =
            Observable.just("Alpha", "Beta", "Gamma", "Delta", "Epsilon");

        myStrings.map(s -> s.length()).subscribe(s ->
            System.out.println(s));
    }
}
```

1. 리액티브하게 생각하기
2. Observable 과 Observer
3. 기본 연산자

# RxJava 프로그래밍

# RxJava

- Observer 의 약점 혹은 보완 사항
  - 스트림의 끝을 알려주지 않는다.
  - 공급자 측에서 에러 혹은 예외사항이 발생했을 때 알려주지 않는다.

**위 두가지를 보완하여 RxJava 가 만들어 집니다. by 에릭마이어**

# RxJava

- Reactive Extensions에서 JVM으로 확장된 라이브러리
  - Reactive Streams 표준은 아님, 별도로 발전
  - Netflix에서 주도하여 개발
  - RxJava 2.0 이상에서 Reactive streams 지원
    - ▶ RxJava 3.0에서는 Reactive Streams 단 하나의 종속이 있음

구분	생산자	소비자
Reactive Streams 지원	Flowable	Subscriber
Reactive Streams 미지원	Observable	Observer

- 버전
  - 1.x, 2.x, 3.0 세 가지 버전이 공존
  - 처음 프로젝트에는 3.0 사용, 2.x는 2021년 2월 28일까지 버그 수정을 통해서만 유지

# RxJava 설정



## Apache Maven

[maven.apache.org](https://maven.apache.org)



```
<dependency>
    <groupId>io.reactivex.rxjava3</groupId>
    <artifactId>rxjava</artifactId>
    <version>3.0.0</version>
</dependency>
```



## Gradle Groovy DSL

[gradle.org](https://gradle.org)



```
implementation 'io.reactivex.rxjava3:rxjava:3.0.0'
```



## Gradle Kotlin DSL

[github.com/gradle/kotlin-dsl](https://github.com/gradle/kotlin-dsl)



```
implementation("io.reactivex.rxjava3:rxjava:3.0.0")
```

# RxJava 설정

---

## 실습

```
package rxjava.examples;

import io.reactivex.rxjava3.core.*;

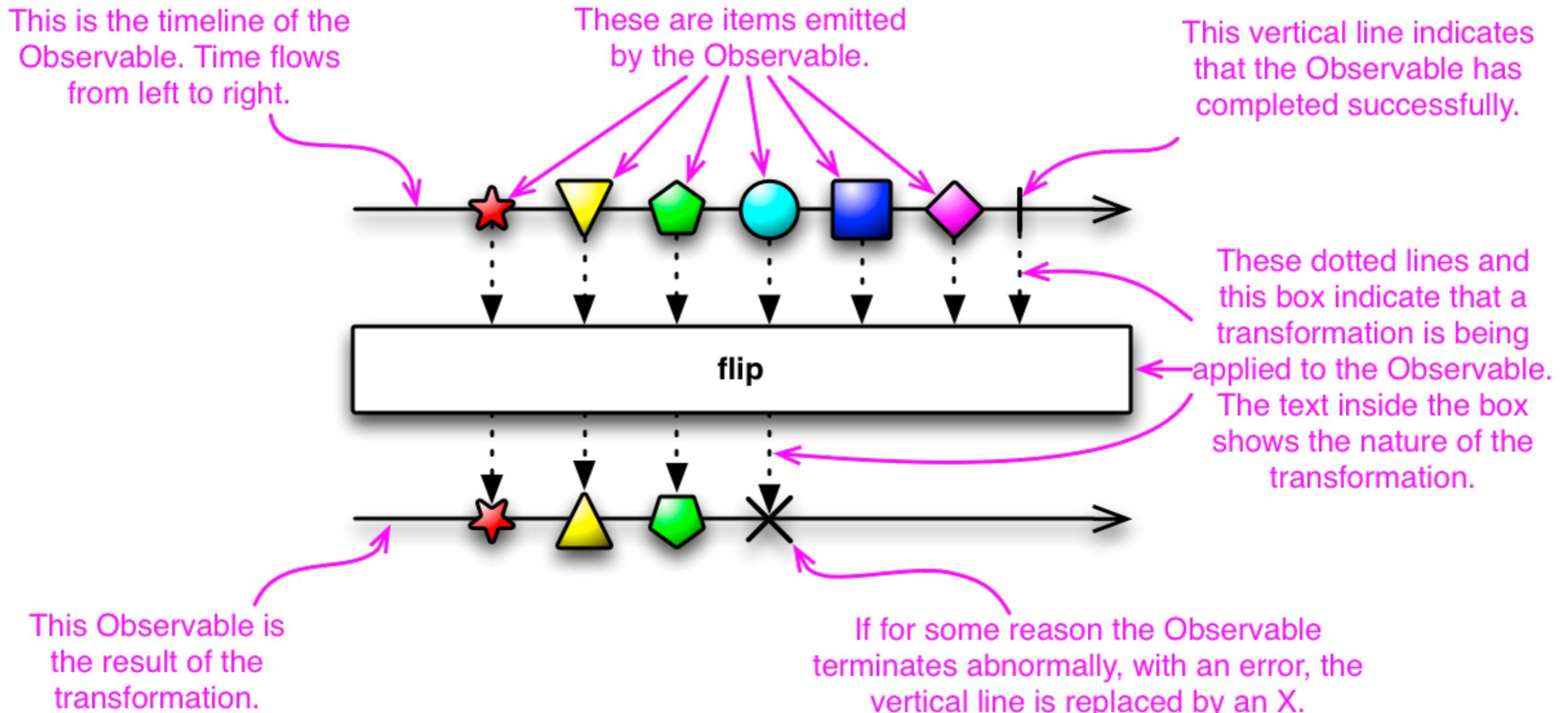
public class HelloWorld {
    public static void main(String[] args) {
        Observable.just("Hello world").subscribe(System.out::println);
    }
}
```

# RxJava 학습 순서

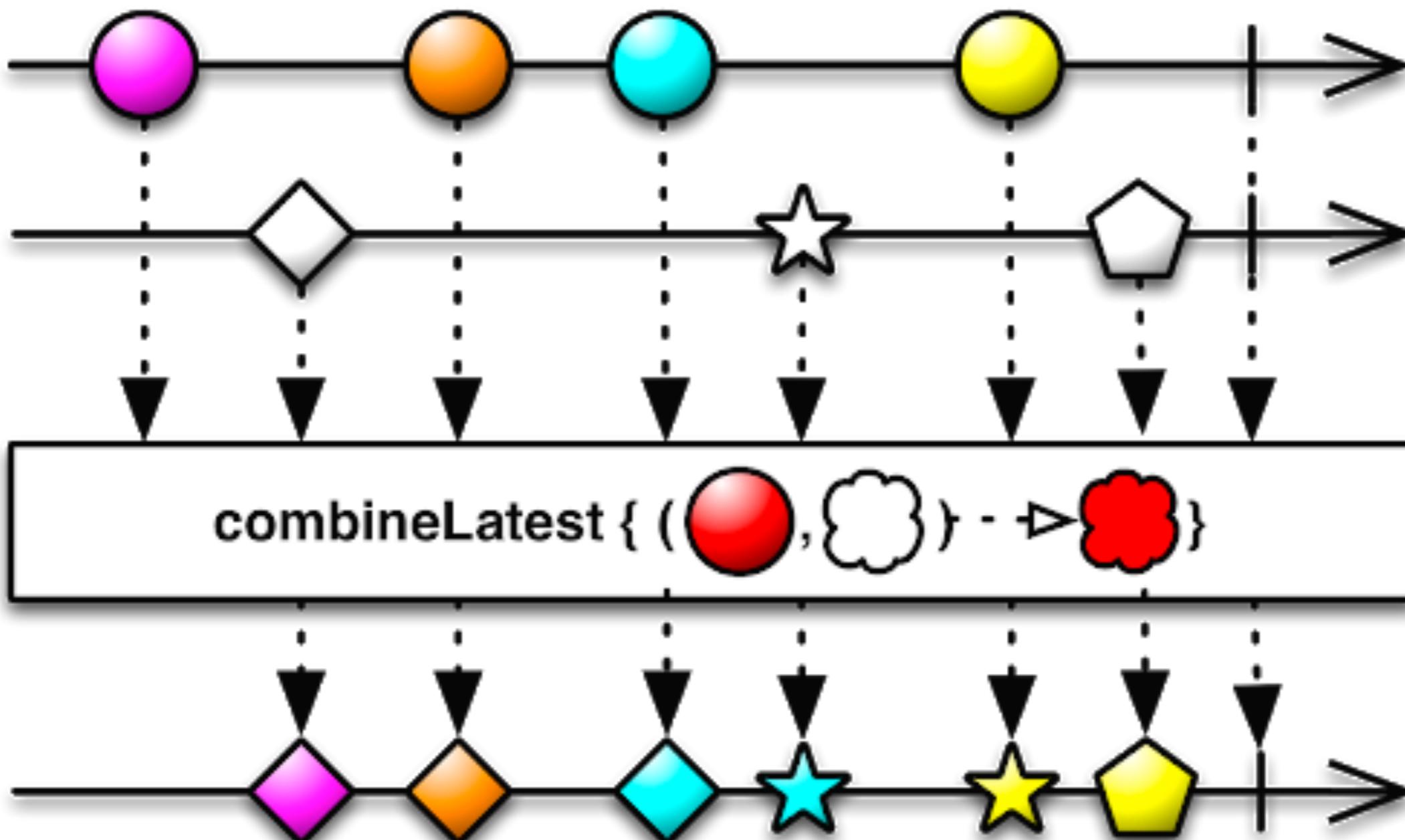
---

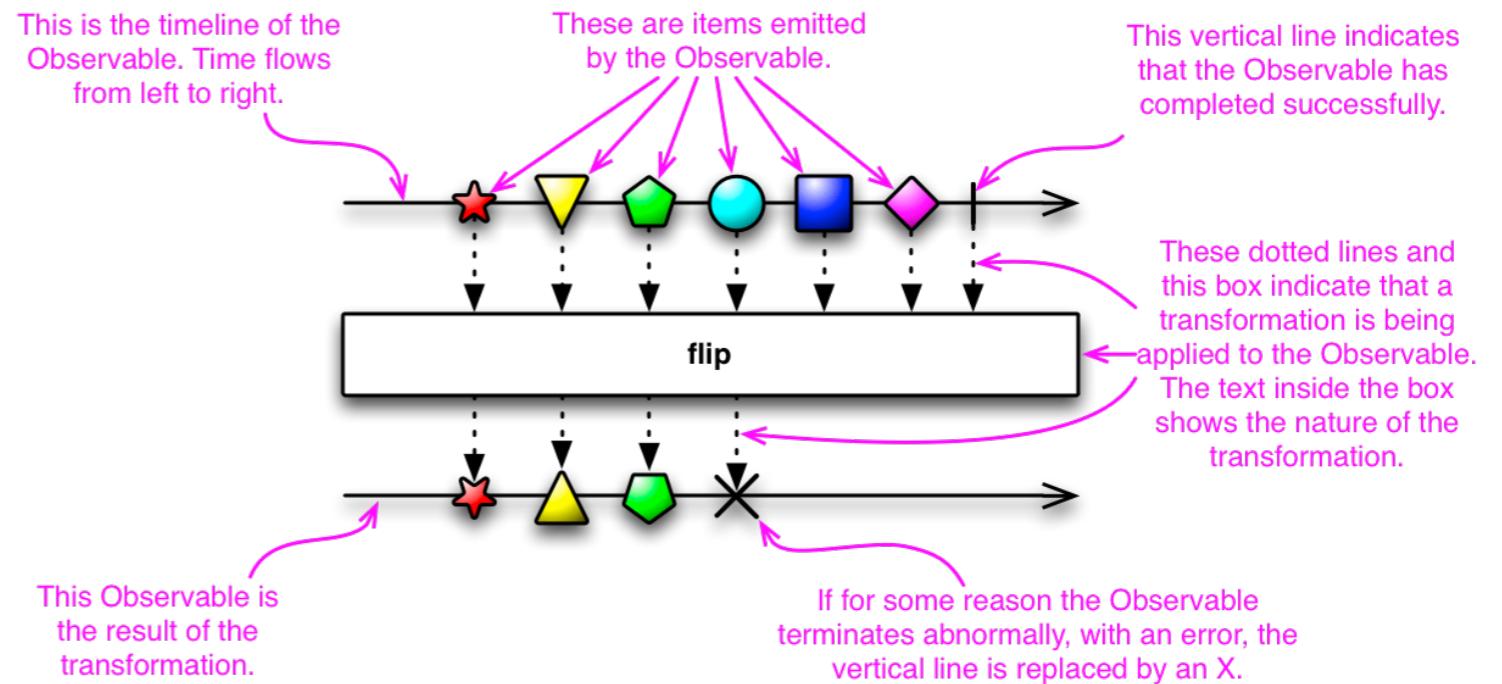
- Observable 클래스를 명확하게 이해한다.
  - 특히 Hot Observable과 Cold Observable의 개념을 꼭 이해해야 한다.
- map(), filter(), reduce(), flatMap() 함수의 사용법을 익힌다.
- 생성 연산자, 결합 연산자, 변환 연산자 등 카테고리별 주요 함수를 공부한다.
- 스케줄러의 의미를 배우고 subscribeOn()과 observeOn() 함수의 차이를 알아둔다.
- 디버깅, 흐름 제어 함수를 익힌다.
  - Low code, No code.(RPA, UiPath) 개발자 <→ 비개발자 (citizen developer)

# 마블 다이어그램



# 마블 다이어그램 예제





2장

# Observable

# Observable

---

- Observable

- 데이터 흐름에 맞게 알림을 보내 구독자가 데이터를 처리할 수 있도록 함
- 데이터를 생산하는 부분
- Push-based

- Observable 동작 방식

- onNext()
- onComplete()
- onError()

```
onSubscribe onNext* (onError | onComplete)?
```

# Observer 인터페이스

```
package io.reactivex;  
  
import io.reactivex.disposables.Disposable;  
  
public interface Observer<T> {  
    void onSubscribe(Disposable d);  
    void onNext(T value);  
    void onError(Throwable e);  
    void onComplete();  
}
```

```
public class Launcher {  
  
    public static void main(String[] args) {  
  
        Observable<String> source =  
            Observable.just("Alpha", "Beta", "Gamma", "Delta",  
                            "Epsilon");  
  
        Observer<Integer> myObserver = new Observer<Integer>() {  
            @Override  
            public void onSubscribe(Disposable d) {  
                //do nothing with Disposable, disregard for now  
            }  
  
            @Override  
            public void onNext(Integer value) {  
                System.out.println("RECEIVED: " + value);  
            }  
  
            @Override  
            public void onError(Throwable e) {  
                e.printStackTrace();  
            }  
  
            @Override  
            public void onComplete() {  
                System.out.println("Done!");  
            }  
        };  
  
        source.map(String::length).filter(i -> i >= 5)  
            .subscribe(myObserver);  
    }  
}
```

# Observable 작동방식

---

- `onNext()`
  - 한번에 하나씩 Observable에서 Observer로 아이템이 전달됨
- `onComplete()`
  - 더 이상 `onNext()` 호출이 발생하지 않음을 나타내는 완료 이벤트가 전달
- `onError()`
  - `retry()`를 사용하지 않는 한 Observable 체인은 종료되고 더 이상 방출 안됨

# Observable 소스 생성 방법

---

- Observable.create()
  - Observable.just()
  - Observable.fromIterable()
- 
- Observable.range()
  - Observable.interval()
  - Observable.future()
  - Observable.empty()
  - Observable.never()
  - Observable.error()
  - Observable.defer()
  - Observable.fromCallable()

# Observable.create()

- Observable 을 사용하여 소스를 생성

```
import io.reactivex.rxjava3.core.Observable;

public class Ch2_01 {
    public static void main(String[] args) {
        Observable<String> source = Observable.create(emitter -> {
            emitter.onNext("Alpha");
            emitter.onNext("Beta");
            emitter.onNext("Gamma");
            emitter.onComplete();
        });
        source.subscribe(s -> System.out.println("RECEIVED: " + s));
    }
}
```

```
source.subscribe(s -> System.out.println("RECEIVED: " + s),
    Throwable::printStackTrace);
```

# Observable.just()

```
import io.reactivex.rxjava3.core.Observable;

public class Ch2_05 {
    public static void main(String[] args) {
        Observable<String> source =
            Observable.just("Alpha", "Beta", "Gamma");
        source.map(String::length).filter(i -> i >= 5)
            .subscribe(s -> System.out.println("RECEIVED: " + s));
    }
}
```

```
import io.reactivex.rxjava3.core.Observable;
import java.util.List;
public class Ch2_06 {
    public static void main(String[] args) {
        List<String> items = List.of("Alpha", "Beta", "Gamma");
        Observable<String> source = Observable.fromIterable(items);
        source.map(String::length).filter(i -> i >= 5)
            .subscribe(s -> System.out.println("RECEIVED: " + s));
    }
}
```

# Observer 인터페이스

```
import io.reactivex.rxjava3.core.Observable;
import io.reactivex.rxjava3.core.Observer;
import io.reactivex.rxjava3.disposables.Disposable;

public class Ch2_07 {
    public static void main(String[] args) {
        Observable<String> source =
            Observable.just("Alpha", "Beta", "Gamma");
        Observer<Integer> myObserver = new Observer<>() {
            @Override
            public void onSubscribe(Disposable d) {
                //do nothing with Disposable, disregard for now
            }
            @Override
            public void onNext(Integer value) {
                System.out.println("RECEIVED: " + value);
            }
            @Override
            public void onError(Throwable e) { e.printStackTrace(); }

            @Override
            public void onComplete() { System.out.println("Done!"); }
        };
        source.map(String::length)
            .filter(i -> i >= 5)
            .subscribe(myObserver);
    }
}
```

# 7|ET Observable 생성 함수

---

## Creating Observables

Operators that originate new Observables.

- **Create** — create an Observable from scratch by calling observer methods programmatically
- **Defer** — do not create the Observable until the observer subscribes, and create a fresh Observable for each observer
- **Empty / Never / Throw** — create Observables that have very precise and limited behavior
- **From** — convert some other object or data structure into an Observable
- **Interval** — create an Observable that emits a sequence of integers spaced by a particular time interval
- **Just** — convert an object or a set of objects into an Observable that emits that or those objects
- **Range** — create an Observable that emits a range of sequential integers
- **Repeat** — create an Observable that emits a particular item or sequence of items repeatedly
- **Start** — create an Observable that emits the return value of a function
- **Timer** — create an Observable that emits a single item after a given delay

# **Cold Observable vs. Hot Observable**

# Cold vs. Hot Observable

---

## ■ Cold Observable

- 각각의 구독자에게 데이터를 방출
- 주로 유한 데이터셋을 방출하는 소스

## ■ Hot Observable

- 라디오 방송국과 유사
- 모든 구독자에게 동일한 데이터를 방출
- 유한 데이터셋 보다 이벤트를 나타냄

# ConnectableObservable

- publish()

- ConnectableObservable 클래스로 변환
- HotObservable을 생성해 줌

```
import io.reactivex.rxjava3.core.Observable;
import io.reactivex.rxjava3observables.ConnectableObservable;

public class Ch2_14 {
    public static void main(String[] args) {
        ConnectableObservable<String> source =
            Observable.just("Alpha", "Beta", "Gamma").publish();
        //Set up observer 1
        source.subscribe(s -> System.out.println("Observer 1: " + s));
        //Setup observer 2
        source.map(String::length)
            .subscribe(i -> System.out.println("Observer 2: " + i));
        //Fire!
        source.connect();
    }
}
```

**Single, Completable, Maybe**

# Single, Completable, Maybe

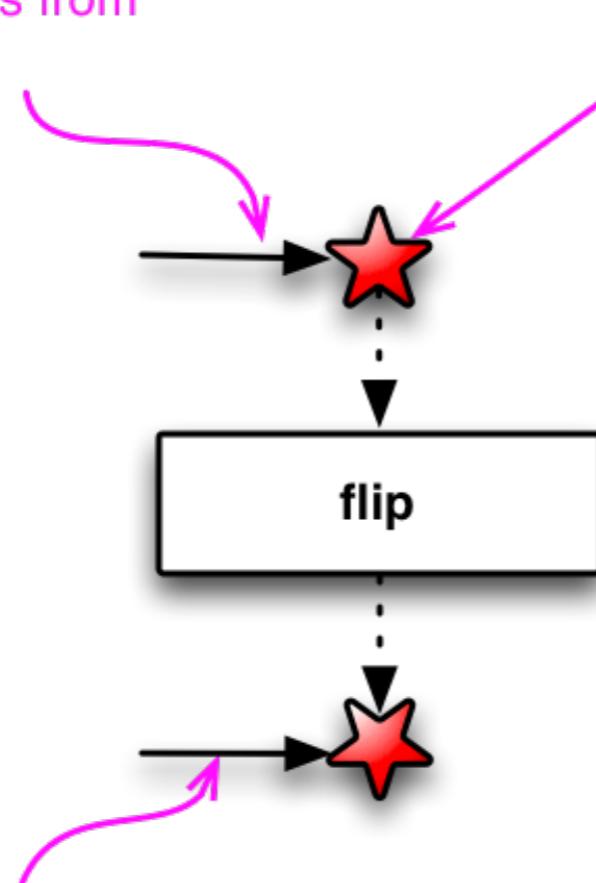
---

## ■ Observable 의 특별한 형태

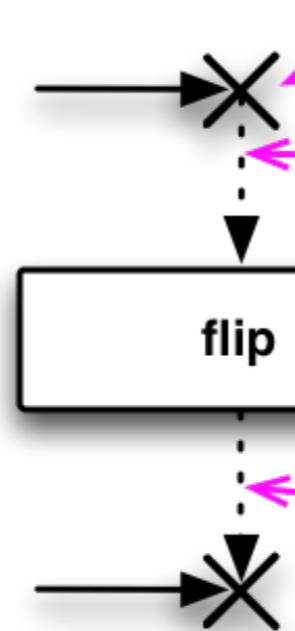
- Single<T>
  - ▶ 단 하나의 아이템만 방출(emit)
  - ▶ onSuccess() 를 통해 onNext() 와 onComplete() 가 통합됨
  - ▶ onError() 로 에러를 알려줌
- Maybe<T>
  - ▶ 아이템이 하나 또는 없을 수도 있음
  - ▶ onSuccess() 로 아이템 방출, onComplete() 는 아이템 방출없이 종료
  - ▶ onError() 로 에러를 알려줌
- Completable<T>
  - ▶ 아이템을 포함하지 않음
  - ▶ onComplete() 와 onError() 만 존재

# Single 클래스

This is the timeline of the Single. Time flows from left to right.



This is the item emitted by the Single.



If for some reason the Single terminates abnormally, with an error, this is indicated with an X.

This Single is the result of the transformation.

# Single<T>

## ■ 하나의 항목만 방출

```
interface SingleObserver<T> {  
    void onSubscribe(@NonNull Disposable d);  
    void onSuccess(T value);  
    void onError(@NonNull Throwable error);  
}
```

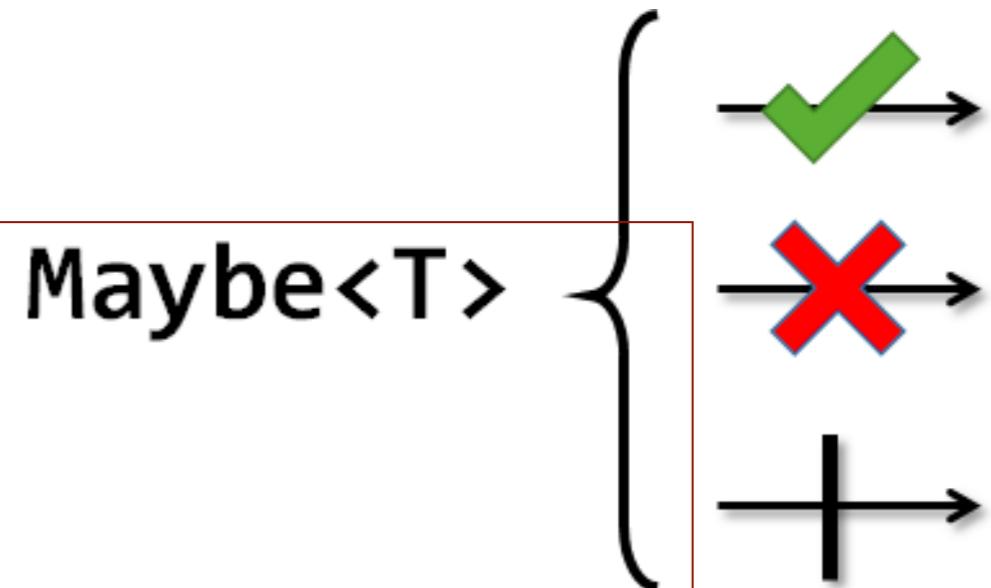
```
import io.reactivex.rxjava3.core.Single;  
  
public class Ch2_28 {  
    public static void main(String[] args) {  
        Single.just("Hello!")  
            .map(String::length)  
            .subscribe(System.out::println,  
                      e -> System.out.println("Error captured: " + e));  
    }  
}
```

```
import io.reactivex.rxjava3.core.Observable;  
  
public class Ch2_29 {  
    public static void main(String[] args) {  
        Observable<String> source = Observable.just("Alpha", "Beta");  
        source.first("Nil") //returns a Single  
            .subscribe(System.out::println);  
    }  
}
```

# Maybe<T>

- 0 또는 1개의 항목을 방출

```
import io.reactivex.rxjava3.core.Maybe;  
  
public class Ch2_30a {  
    public static void main(String[] args) {  
        // has emission  
        Maybe<Integer> source = Maybe.just(100);  
        source.subscribe(s -> System.out.println("Process 1: " + s),  
                        e -> System.out.println("Error captured: " + e),  
                        () -> System.out.println("Process 1 done!"));  
        //no emission  
        Maybe<Integer> empty = Maybe.empty();  
        empty.subscribe(s -> System.out.println("Process 2: " + s),  
                        e -> System.out.println("Error captured: " + e),  
                        () -> System.out.println("Process 2 done!"));  
    }  
}
```



# Completable

---

- 아무 항목도 방출하지 않음

```
import io.reactivex.rxjava3.core.Completable;

public class Ch2_32 {
    public static void main(String[] args) {
        Completable.fromRunnable(() -> runProcess())
            .subscribe(() -> System.out.println("Done!"));
    }

    private static void runProcess() {
        //run process here
    }
}
```

# 구독 중단. Disposable

- Observable에서 onComplete() 가 호출되면 모든 자원들이 정리됨
- 구독 중간에 해지를 하고 싶으면 dispose() 를 호출

```
Observer<Integer> myObserver = new Observer<Integer>() {  
    private Disposable disposable;  
  
    @Override  
    public void onSubscribe(Disposable disposable) {  
        this.disposable = disposable;  
    }  
  
    @Override  
    public void onNext(Integer value) {  
        //has access to Disposable  
    }  
  
    @Override  
    public void onError(Throwable e) {  
        //has access to Disposable  
    }  
  
    @Override  
    public void onComplete() {  
        //has access to Disposable  
    }  
};
```

3장

# 기본 연산자

# 연산자 종류

---

- 필터링(조건부, Conditional) 관련 연산
- 매팅 관련 연산
- 리듀싱 관련 연산 - BiFunction  $((a, b) \rightarrow c)$  -
  - 평균, 최대값, 최소값 25%, 75%, median, mode, var, std
- 컬렉션 연산
- 에러처리 관련 연산
- 액션 연산
- 유틸리티 연산

# 필터링 관련 연산

---

- filter()
  - Predicate<T> 받아서 Observable<T> 생성
- take()
  - take(3) : 3개 데이터를 emit 하고 onComplete() 호출
- skip()
- takeWhile(), skipWhile()
- distinct()
- distinctUntilChanged()
- elementAt()

# 매핑 관련 연산

---

- map( )
  - Function(T, R) 받아서 Observable<T> 반환
- cast( )
- startWith( )
- defaultIfEmpty( )
- switchIfEmpty( )
- sorted( )
- delay( )
- repeat( )
- scan( ) : rolling aggregator

# 리듀싱 관련 연산

---

- count()
- reduce()
- Single<Boolean> 반환
  - all()
  - any()
  - isEmpty()
  - contains()
  - sequenceEqual()

# 컬렉션 연산

---

- `toList()`
- `toSortedList()`
- `toMap, toMultiMap()`
- `collect()`

# 에러처리 관련 연산

---

- `onErrorReturn()`
- `onErrorReturnItem()`
- `onErrorResumeWith()`
- `retry()`

# 액션 연산

---

- doOnNext(), doAfterNext()
  - doOnComplete()
  - doOnError()
  - doOnEach()
- 
- doOnSubscribe()
  - doOnDispose()
- 
- doFinally()

## 유틸리티 연산

---

- delay()
- repeat()
- single()
- timestamp()
- timeInterval()

# Reactive 연산자

Observable 결합

Multicasting, Replaying & Caching

Concurrency & Parallelization

Switching, Throttling, Windowing & Buffering

Flowable 과 역압(Backpressure)

4장

# **Observable 결합**

# 빅데이터 시대(2010~)의 데이터 처리

---

- 데이터 처리(분석)를 RDB에서만 하던것에서 탈피
  - NoSQL 등장
  - 애플리케이션에서도 데이터 처리 수행
    - ▶ 함수형 프로그래밍 -> 모던 랭귀지, 멀티 패러다임 랭귀지
    - ▶ 데이터 처리 라이브러리 -> 리액티브 방식의 프로그래밍 지원 라이브러리 등장
- 리액티브 프로그래밍
  - 데이터를 Flow로 간주(취급)
- 데이터 분석
  - 기술적 분석(descriptive) -> 집계(aggregation) - 인간 Data Analyst(차트 그리기)
  - 예측 분석 -> 머신러닝 - 기계 → Data Scientist

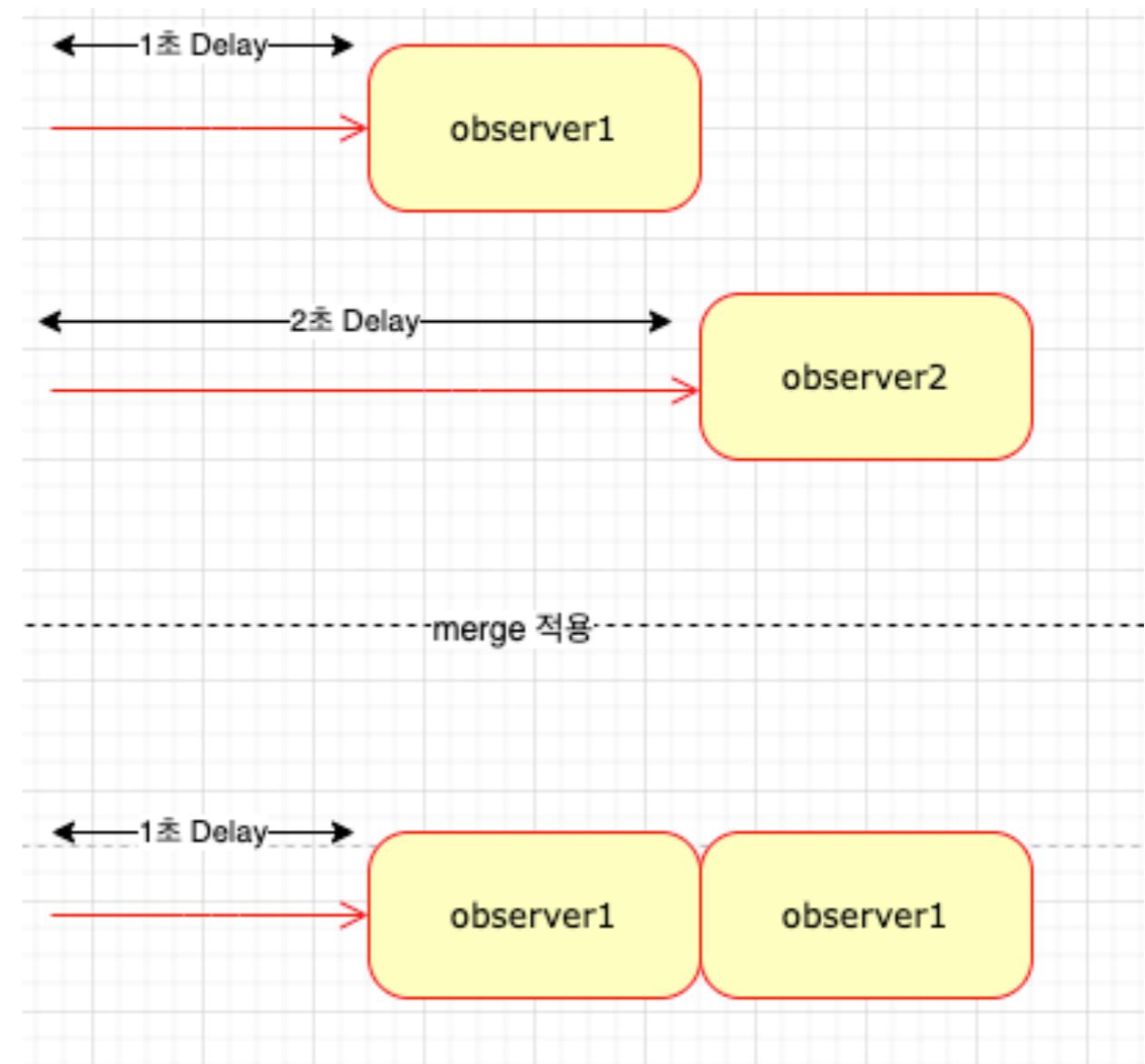
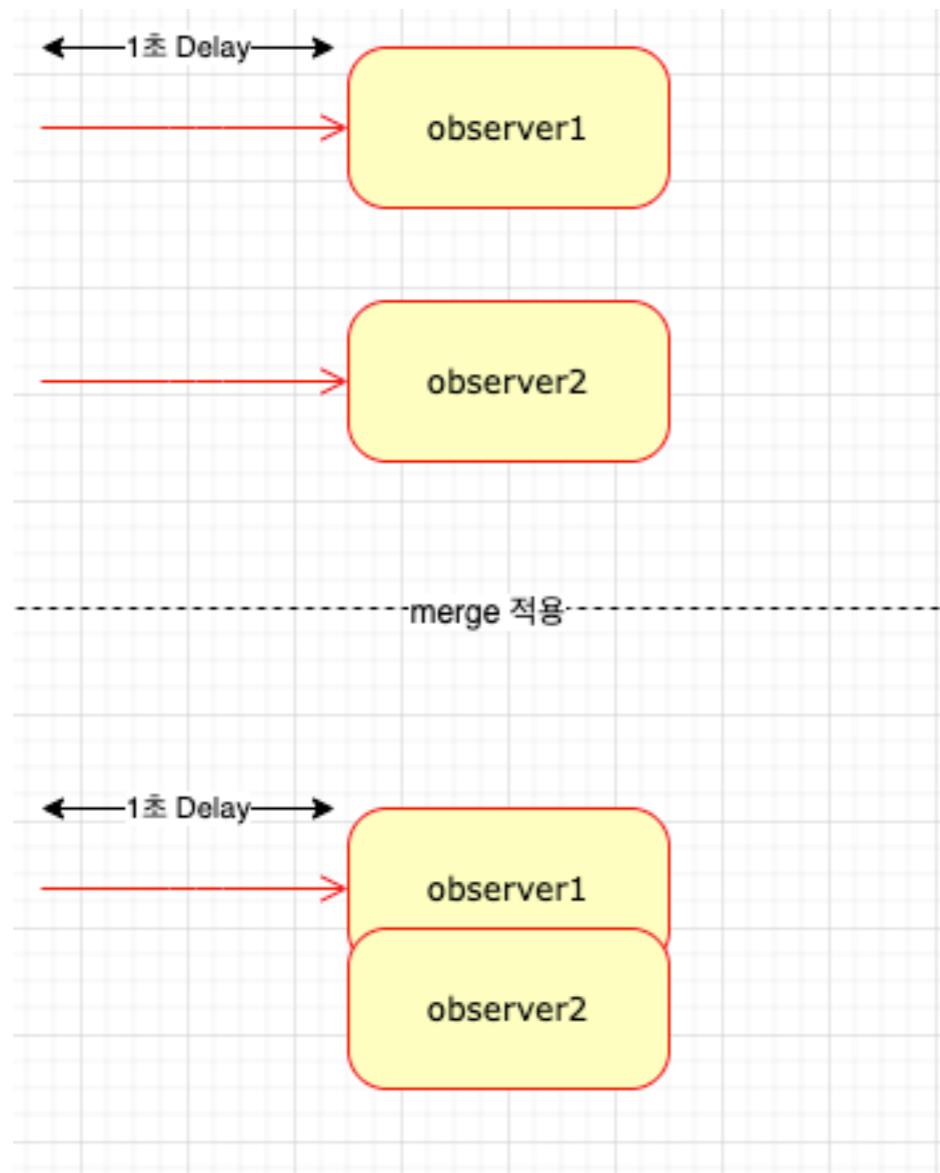
# Combining Observables

---

- Merging
  - `merge()`, `mergeWith()`, `mergeArray()`
  - `flatMap()`, `flatMap` with combiner
- Concatenating
  - `concat()`, `concatWith()`, `concatMap()`
- Ambiguous
  - `amb()`, `ambWith()`
- Zipping
  - `zip()`, `zipArray()`
- Combine latest
  - `combineLatest()`, `withLatestFrom()`
- Grouping
  - `groupBy()`

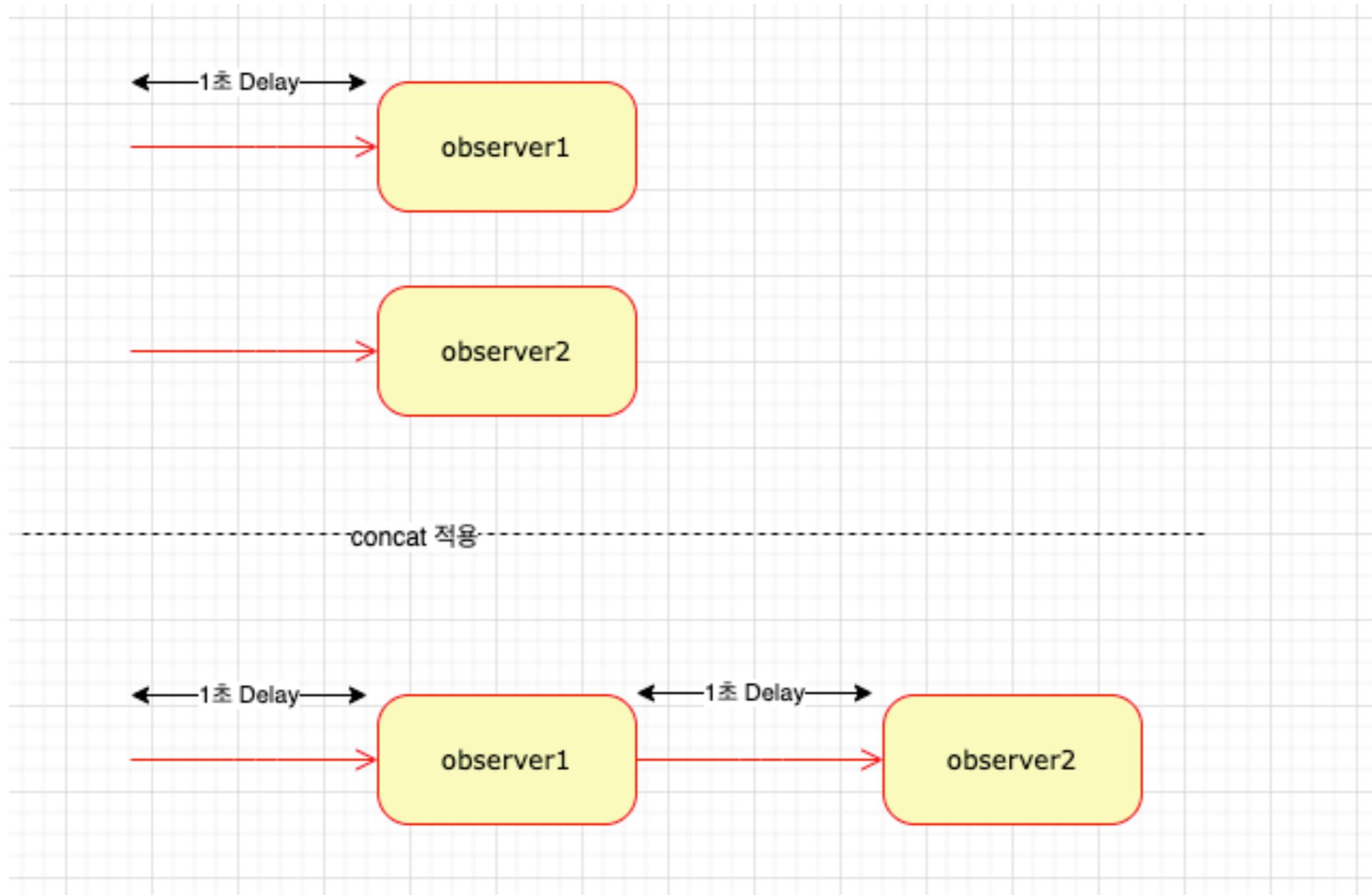
# merge vs. concat

## ■ merge



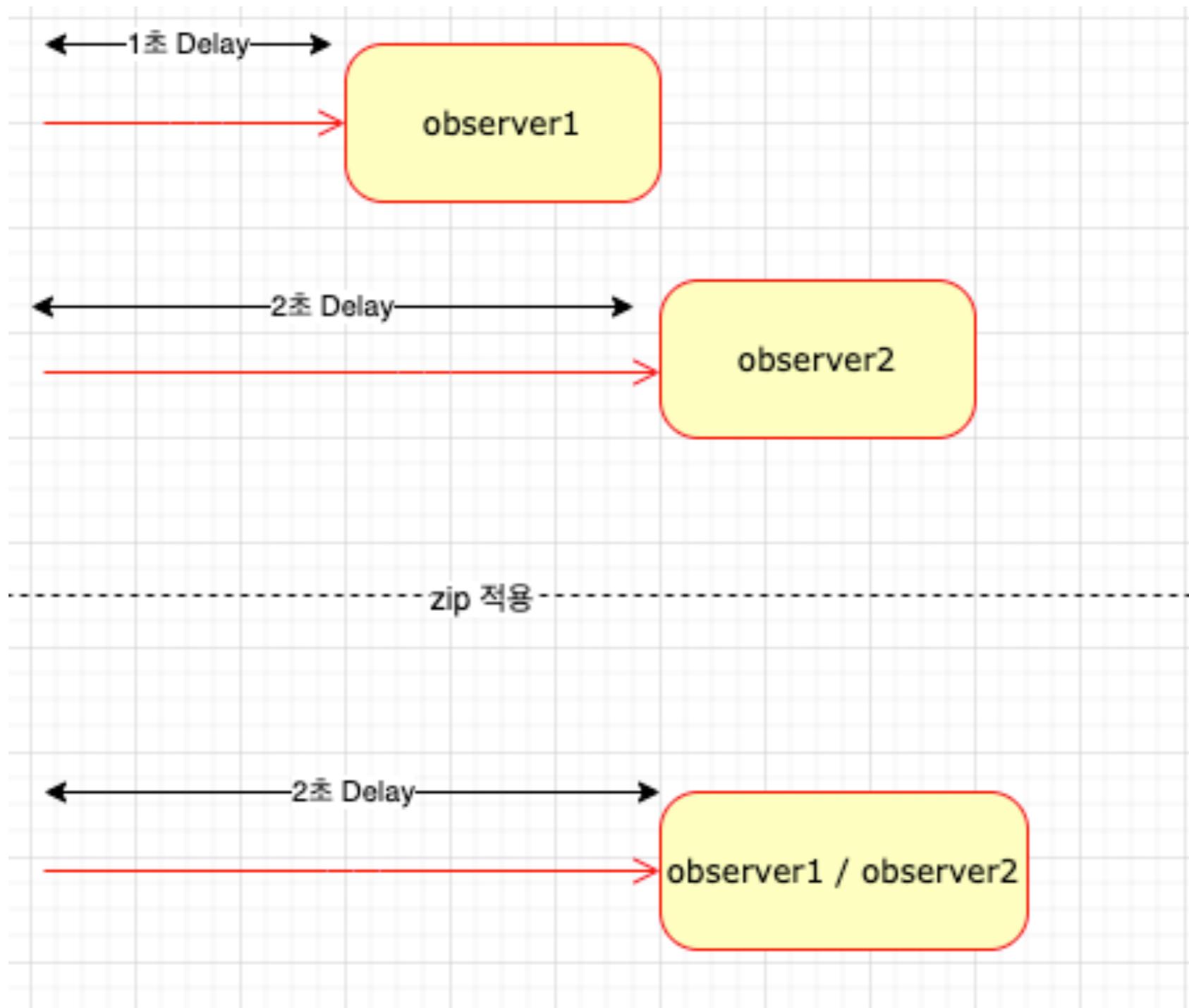
# merge vs. concat

## concat



# zip

- zip



5장

# **멀티캐스팅, 리플레이, 캐싱**

## **Multicasting, Replying, Caching**

# Multicasting, Replying, 그리고 Caching

- 멀티캐스팅 이해
- Automatic connection
- Replying 과 caching

```
import io.reactivex.rxjava3.core.Observable;

public class Ch5_01Multicasting {
    public static void main(String[] args) {
        Observable<Integer> ints = Observable.range(1, 3);
        ints.subscribe(i -> System.out.println("Observer One: " + i));
        ints.subscribe(i -> System.out.println("Observer Two: " + i));
    }
}
```

# Multicasting 옵션

## ■ ConnectableObservable 사용

- “Cold” Observable → “Hot” Observable
- Multicasting 수행

```
import io.reactivex.rxjava3.core.Observable;
import io.reactivex.rxjava3observables.ConnectableObservable;

public class Ch5_02Multicasting {
    public static void main(String[] args) {
        ConnectableObservable<Integer> ints =
            Observable.range(1, 3).publish();

        ints.subscribe(i -> System.out.println("Observer One:" + i));
        ints.subscribe(i -> System.out.println("Observer Two:" + i));
        ints.connect();
    }
}
```

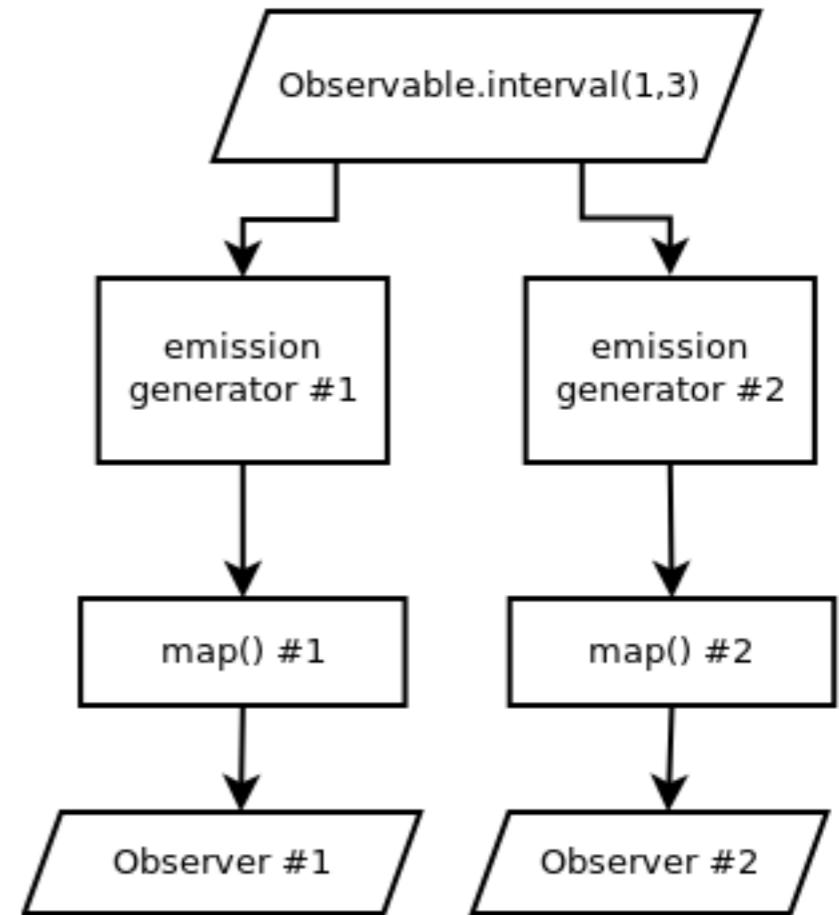
# Multicasting 이해 : 멀티캐스팅과 연산자

```
import io.reactivex.rxjava3.core.Observable;
import java.util.concurrent.ThreadLocalRandom;

public class Ch5_03Multicasting {
    public static void main(String[] args) {
        Observable<Integer> ints = Observable
            .range(1, 3)
            .map(i -> randomInt());

        ints.subscribe(
            i -> System.out.println("Observer 1: " + i));
        ints.subscribe(
            i -> System.out.println("Observer 2: " + i));
    }

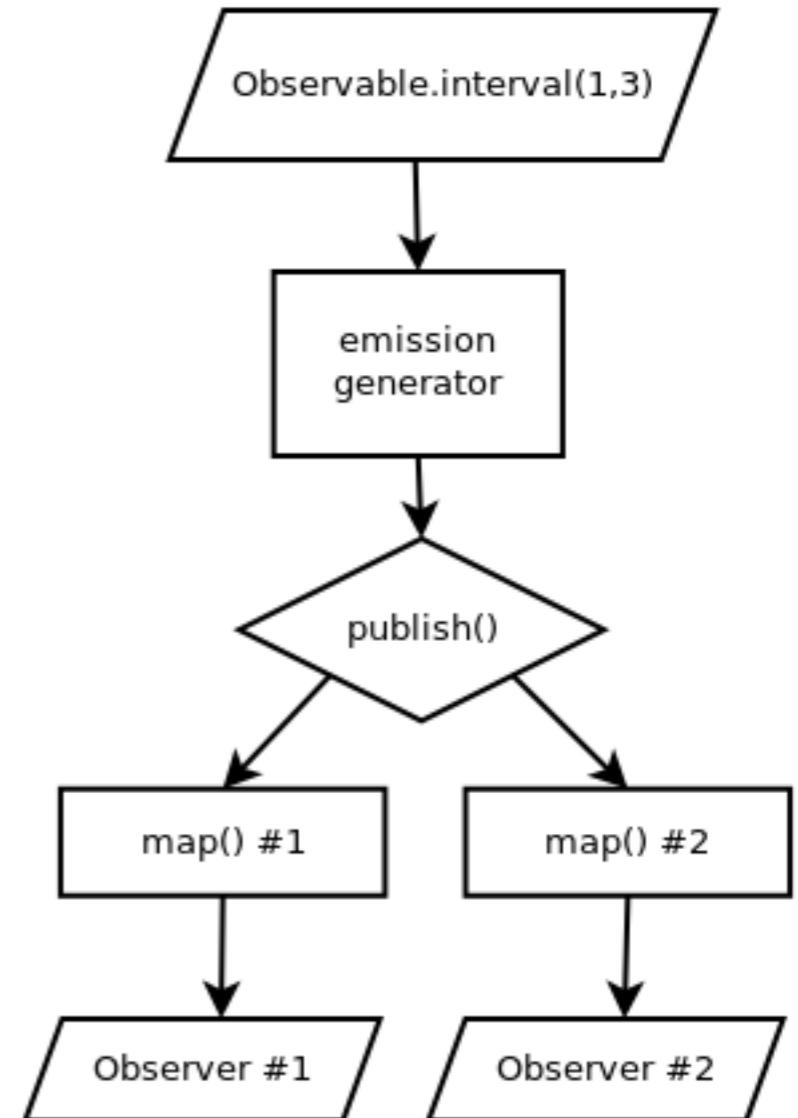
    public static int randomInt() {
        return ThreadLocalRandom.current().nextInt(100000);
    }
}
```



# Multicasting 이해 : 멀티캐스팅과 연산자

```
import io.reactivex.rxjava3.core.Observable;
import io.reactivex.rxjava3observables.ConnectableObservable;
import java.util.concurrent.ThreadLocalRandom;

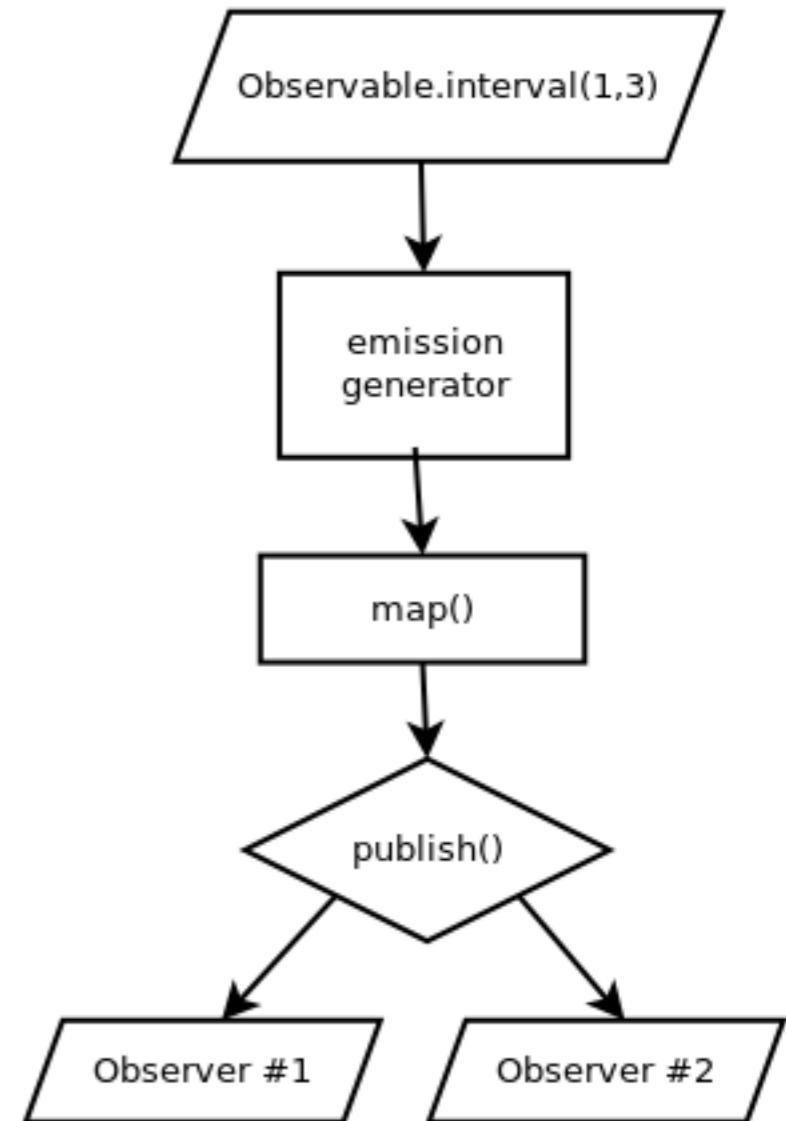
public class Ch5_04Multicasting {
    public static void main(String[] args) {
        ConnectableObservable<Integer> ints =
            Observable.range(1, 3).publish();
        Observable<Integer> rInts = ints.map(i -> randomInt());
        rInts.subscribe(i -> System.out.println("Observer 1: " + i));
        rInts.subscribe(i -> System.out.println("Observer 2: " + i));
        ints.connect();
    }
}
```



# Multicasting 이해 : 멀티캐스팅과 연산자

```
import io.reactivex.rxjava3.core.Observable;
import io.reactivex.rxjava3.observables.ConnectableObservable;
import java.util.concurrent.ThreadLocalRandom;

public class Ch5_05Multicasting {
    public static void main(String[] args) {
        ConnectableObservable<Integer> rInts =
            Observable.range(1, 3).map(i -> randomInt()).publish();
        rInts.subscribe(i -> System.out.println("Observer 1: " + i));
        rInts.subscribe(i -> System.out.println("Observer 2: " + i));
        rInts.connect();
    }
}
```

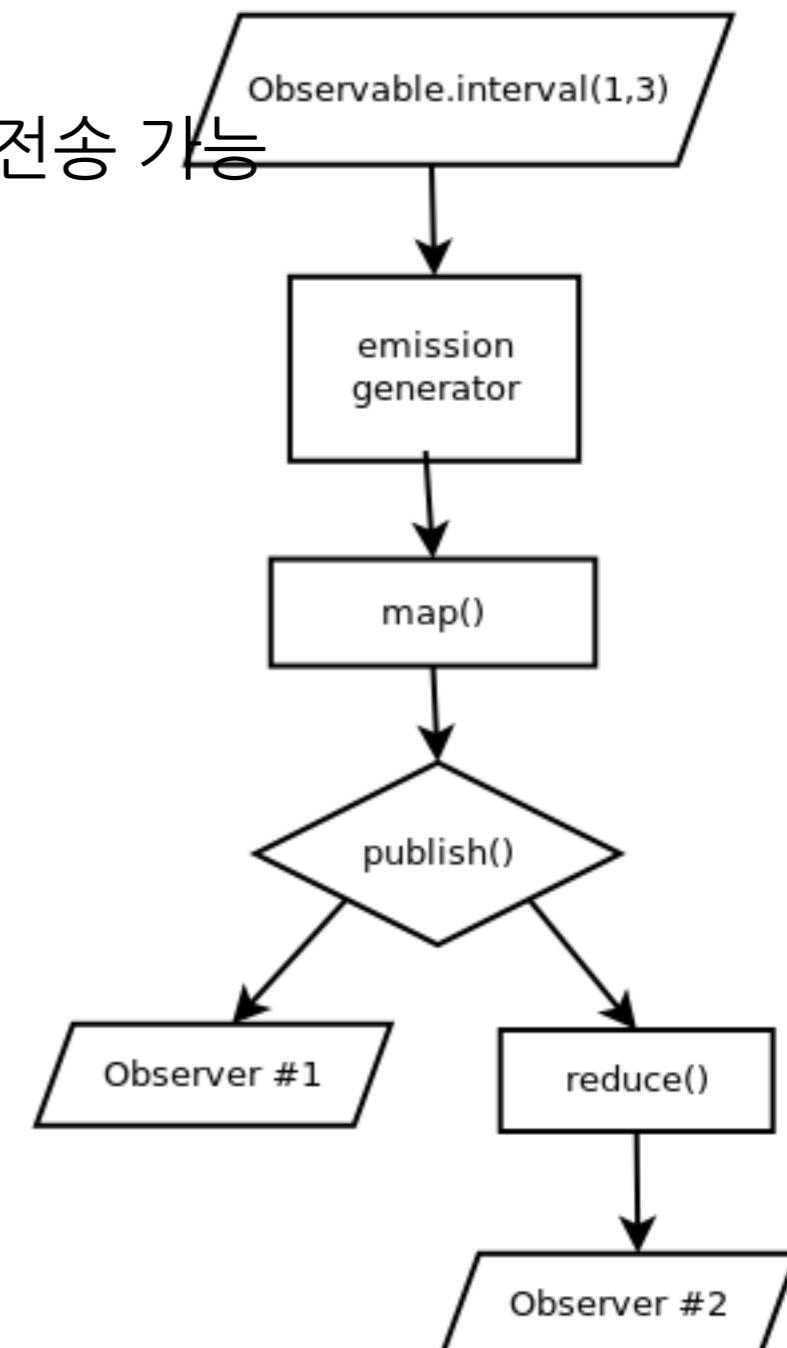


# Multicasting 이해 : 언제 멀티캐스트를 해야 하는가

## 동일한 데이터를 여러 observer에게 보낼 때

- publish() 이전에 데이터를 처리하면 동일한 데이터 전송 가능
- 메모리와 CPU 사용량 줄여 성능 향상 효과

```
public class Ch5_Multicasting06 {  
    public static void main(String[] args) {  
        ConnectableObservable<Integer> rInts =  
            Observable.range(1, 3).map(i -> randomInt()).publish();  
        //Observer 1 - print each random integer  
        rInts.subscribe(i -> System.out.println("Observer 1: " + i));  
  
        //Observer 2 - sum the random integers, then print  
        rInts.reduce(0, (total, next) -> total + next)  
            .subscribe(i -> System.out.println("Observer 2: " + i));  
        rInts.connect();  
    }  
}
```



# Automatic connection

---

- autoConnect(nObserver)
  - 옵저버 수를 지정, 구독이 끝나면 업스트림을 폐기함
- autoConnect()
  - 옵저버 수만큼 기다리지 않고 즉시 emit 수행
- refCount() & share()
  - autoConnect(1) 과 유사
  - 모든 구독이 완료되고 새 구독이 요청되면 다시 emit 수행

# Replaying 과 caching

---

## ■ 리플레이와 캐싱

- 멀티캐스팅은 여러 옵저버에 데이터를 공유하기 위해 캐싱을 사용

## ■ Replaying, replay()

- 새로운 옵저버가 있을 때 데이터를 재발송(re-emit)할 수 있는 기능
- ConnectableObservable 반환
- replay(int) : 파라미터의 숫자는 마지막 발송분만 캐싱한다는 의미

## ■ Caching

- 무기한 데이터를 캐시하고 싶을 때 cache() 연산자 사용

6장

**동시성, 병렬성**

**Concurrency, Parallelization**

# Concurrency 와 Parallelization

---

## ■ 동시성, Concurrency

- 지난 10여년 자바에서 동시성의 필요성 증가
- Concurrency(aka. 멀티쓰레딩) 은 멀티태스킹에 필수
- 컴퓨팅 자원을 최대한 활용하기 위해서 동시성 작업은 필수
- RxJava 에서는 동시성 작업을 쉽고 안전하게 할 수 있도록 지원

동시성의 필요성

RxJava 동시성

스케줄러, Schedulers

subscribeOn()

observeOn()

병렬성, parallelization

unsubscribeOn()

# 동시성, Concurrency 의 필요성

---

- CPU 코어 수 증가와 CPU 성능 향상
  - 여러 코어를 동시에 사용하는 코드는 작성이 어려움
- RxJava 는 동시성(멀티쓰레딩) 작업을 쉽고 안전하게 해줌
  - subscribeOn()
  - observeOn()
  - flatMap()

# RxJava 동시성 소개

---

- `SubscribeOn()`
  - 지정된 스케줄러, Scheduler(별도의 쓰레드)에서 emit하도록 제안
- 데몬같이 무한으로 유지하고 싶을 때  
`sleep(Long.MAX_VALUE);`
- 구독이 완료될 때 까지만 애플리케이션 유지  
`blockingSubscribe()`

# 스케줄러, Scheduler 이해

---

## ■ 스레드 풀, Thread pool

- 스레드의 모음, 스레드 풀의 정책에 따라 스레드 재사용, 지속, 유지
- 스레드 풀은 필요에 따라 동적으로 스레드를 생성하고 파괴

## ■ 스케줄러

- Computation
  - ▶ interval(), delay(), timer(), timeout(), buffer(), take(), skip(), takeWhile(), skipWhile()
- I/O
- New thread
- Single
- Trampoline
- ExecutorService

# subscribeOn() 이해

---

- Emit 을 실행할 스레드를 결정(소스에 제안)
  - 체인의 아무곳에나 넣을 수 있음, 그러나 가능한 소스에 가깝게 배치
- Observable 이 이미 스케줄러를 사용하고 있을 경우
  - subscribeOn() 은 동작하지 않음
  - 예) Observable.interval()
- subscribeOn() 은 단 한번만 적용됨

# observeOn() 이해

---

- ObserveOn() 은 연산 체인 중간에 스레드를 변경 가능
  - 데이터 소스를 만들 때는 I/O 스케줄러 사용
  - 연산을 할 때는 computation 스케줄러 사용

감사합니다.

[soongon@hucloud.co.kr](mailto:soongon@hucloud.co.kr)