이제는 인터넷으로 다양한 정보를 얻고 자유롭게 정보를 공유할 수 있는 세상이 됐습니다. 일반 웹 서비스뿐만 아니라 모바일, 사물인터넷(IoT), 클라우드 등 다양한 환경과 플랫폼에서 애플리케이션을 사용하며 많은 데이터를 생성합니다.

이에 따라 최근에는 시스템에서 처리할 이벤트나 데이터가 극단적으로 증가하면서 대용량 데이터의 저장, 업데이트, 실시간 반영을 효율적으로 해결할 방법으로 리액티브 프로그래밍이 주목받고 있습니다.

'리액티브(reactive)'라는 말은 원래 외부 자극에 반응한다는 뜻으로, 2000년대 후반부터 소프트웨어가 갖추어야 하는 속성으로 인식돼 시간 경과에 따라 변경되는 값을 선언적으로 실행하고 값의 변화에 반응해 작동한다는 의미로 많이 사용하게 됐습니다.

리액티브 프로그래밍은 데이터가 변경되는 흐름에 따라 자동으로 전파되는 프로그래밍 방식을 말하며, 주로 GUI 입출력 처리, 시간 경과에 따라 상태가 변화하는 처리, 비동기 통신 처리를 하는 애플리케이션에 사용하기 적합합니다. 최근에는 많은 데이터와 이벤트를 실시간으로 처리하고자 애플리케이션 개발과 대규모 서비스 시스템 구축을 리액티브 프로그래밍 기반으로 하고 있습니다.

2013년에 공개된 RxJava는 마이크로소프트에서 프로그래밍 모델을 비동기 및 이벤트 중심의 데이터 구조로 재정의한 Reactive Extensions를 자바 진영으로 가져온 것으로, 데이터를 작은 데이터 조각의 연속인 스트림으로 보는 방식입니다. 비동기성, 이벤트, 배압(back pressure)이 핵심적인 개념이며, 2019년 4월 현재 2.2.7 버전까지 릴리스됐습니다.

이 책은 상세한 예제를 기반으로 한 입문서로, RxJava의 구조와 기능을 이해하고 현업에 쉽게 적용할 수 있게 도와줍니다. 따라서 리액티브 프로그래밍에 관심 있는 개발자가 어렵지 않게 RxJava를 사용할 수 있게 해주며, RxJava의 레퍼런스만으로는 이해하기 어려웠던 다양한 API 사용법을 예제 중심으로 설명하고 있어 기존에 RxJava를 사용하고 있거나 RxJava 기능을 충분히 활용하려는 개발자에게도 많은 도움이 되는 책입니다. 또한, 이 책에 있는 디버깅과 테스트 방법 등은 기존 RxJava 개발 방식을 효율적으로 한 단계 업그레이드시켜줄 것이라고 믿습니다.

세)이 전달받은 데이터로 무엇을 하는지는 몰라도 됩니다. 또한, 데이터를 생산하는 측은 데이터를 소비하는 측에서 무엇을 하든지 관계가 없으므로 소비하는 측의 처리를 기다릴 필요가 없습니다. 그러므로 데이터를 통지한 후 데이터를 소비하는 측에서 데이터를 처리하는 도중이라도 데이터를 생산하는 측은 바로 다음 데이터를 처리할 수 있습니다. 이처럼 비동기 처리를 쉽게 구현할 수 있습니다.

또한, 시스템 구축 관점에서 볼 때도 리액티브 프로그래밍은 **마이크로서비스**(microservice)와 같이 분산 시스템으로 프로그램을 구현하는 데 적합해 최근 주목받고 있습니다.

다만 리액티브 프로그래밍을 헷갈려서 리액티브 시스템(system)으로 부르지 말아야 합니다. 리액티브 시스템이 리액티브 프로그래밍으로 구현된 시스템을 의미하지 않기 때문입니다. **리액티브 시스템**이란 메시지를 보내 데이터를 처리하고 상황에 따라 **스케일 아웃**(scale out)과 **스케일 인**(scale in)을 자동으로 수행해 장애 내성을 높임으로써 항상 빠르게 응답할 수 있는 시스템을 말합니다. 그러므로 이 리액티브 시스템은 프로그램뿐만 아니라 인프라에 대한 조건도 필요합니다. 그리고 각 서비스를 리액티브 프로그래밍으로 구현하지 않아도 리액티브 시스템을 구축할 수 있습니다.

1.1.3 RxJava의 개요

에릭 마이어가 개발한 .NET 프레임워크의 실험적인 라이브러리인 Reactive Extensions(줄여서 Rx)를 2009년 마이크로소프트(Microsoft)에서 공개하고 2013년 넷플릭스(Netflix)가 자바로 이식한 것이 RxJava의 시작입니다.

- RxJava GitHub https://github.com/ReactiveX/RxJava

현재 Reactive Extensions를 다루는 라이브러리는 ReactiveX라는 오픈 소스 프로젝트로 바뀌어 자바와 .NET뿐만 아니라 자바스크립트(JavaScript)나 스위프트(Swift) 등 여러 프로그래밍 언어를 지원하는 라이브러리를 제공합니다.

- ReactiveX http://reactivex.io

RxJava 1.x 버전 때는 자바에 Reactive Extensions를 이식하는 개발이 진행됐고, 리액티브 프로그래밍 개념이 널리 알려지면서 Reactive Extensions와 별도로 여러 업체와 단체에서 데이터 스트림을 비동기로 다루는 라이브러리와 프레임워크를 출시했습니다. 그러자 같은 처리를 하는데도 불구하고 라이브러리나 프레임워크 차이 때문에 서로 다르게 구현하는 상황이 발생했습니다.

그래서 관련 단체들이 모여 데이터 스트림을 비동기로 다루는 최소한의 API를 정하고 제공했습니다. 그리하여 2015년 4월 자바 기반의 Reactive Streams for JVM 버전 1.0.0을 릴리스하게 됩니다.

RxJava 1.x 버전은 Reactive Streams가 만들어지기 전부터 있어서 Reactive Streams를 지원하지 않았고, Reactive Streams를 지원하는 RxJava 2.0 버전은 2016년 10월에 릴리스됐습니다. 2.0 버전은 기본 구조를 유지하면서도 내부를 완전히 새롭게 구현해 1.x 버전보다 성능이 개선됐습니다. Reactive Streams가 RxJava의 영향을 강하게 받아서 근본적인 구조(데이터 스트림 관리, 생산자와 소비자의 관계 등)는 1.x 버전과 큰 차이가 없으나 나중에 설명할 **배압**(back pressure)[4] 사양에 따라 사용하는 API가 변경됐습니다. 그러므로 1.x 버전에서 2.x 버전으로 전환할 때는 단순하게 패키지나 클래스 이름을 바꾸는 것 외에도 이러한 API 변경 사항도 반영해야 합니다. 또한, RxJava 2.x 버전은 내부 구현을 처음부터 다시 설계해 작업했기 때문에 RxJava 1.x 버전의 구현 코드를 RxJava 2.x 버전으로 간단히 전환할 수 없습니다.

그리고 2.x 버전부터는 Reactive Streams를 구현하므로 다른 라이브러리를 의존할 필요가 없던 1.x 버전과는 다르게 Reactive Streams의 jar 파일이 반드시 필요합니다.

그 외에도 1.x 버전과 2.x 버전은 표 1-1과 같이 루트 패키지가 다릅니다.

표 1-1 루트 패키지

버전	패키지
1.x	rx
2.x	io.reactivex

따라서 RxJava의 두 버전을 같은 프로젝트에 함께 사용할 수는 있으나 프로젝트 내부에서 RxJava에 의존하는 다른 라이브러리를 사용한다면 라이브러리가 어떤 버전을 지원하느냐에 따라 두 버전을 함께 사용하지 못할 수도 있습니다. 또한, 1.x에서 2.x 버전으로 업그레이드하는 도중 같은 일시적인 상황이 아니라면 운영과 유지보수를 생각했을 때 두 버전을 함께 사용하는 것은 바람직하지 않습니다.

RxJava 1.x 버전 개발과 지원은 2018년 3월에 종료됐으므로 이 책에서는 2.x 버전을 다루겠습니다.

4 역주 배압이란 정해진 임계값에 도달하면 이미 버퍼에 있는 요청의 내부 처리가 들어오는 요청을 따라잡을 때까지 버퍼에 넣는 것을 일시적으로 정지하거나 막는 것을 말합니다. 역압이라는 용어로 번역되기도 합니다.

그리고 개발 중이거나 실험적인 RxJava API가 있습니다. 이러한 API는 릴리스 시 변경될 가능성이 크므로 이러한 API에 의존하지 않고 프로그램을 구현하는 것이 안전합니다. 다음과 같은 애너테이션(annotation)이 붙어 있는 것은 개발 중이거나 실험적인 API입니다.

- @Beta
- @Experimental

추가로 io.reactivex.internal로 시작하는 패키지에 있는 클래스는 RxJava 내부에서 기본으로 사용하는 클래스입니다. 따라서 향후에 특정 기능을 지원하려고 변경될 가능성이 크므로 구현 시 이러한 클래스에 의존하지 않는 것이 안전합니다.

1.1.4 RxJava의 특징

RxJava는 디자인 패턴인 **옵저버**(Observer) 패턴을 잘 확장했습니다. 옵저버 패턴은 감시 대상 객체의 상태가 변하면 이를 관찰하는 객체에 알려주는 구조입니다. 이 패턴의 특징을 살리면 데이터를 생성하는 측과 데이터를 소비하는 측으로 나눌 수 있기 때문에 쉽게 데이터 스트림을 처리할 수 있습니다. 게다가 RxJava는 옵저버 패턴에 완료와 에러 통지를 할 수 있어서 모든 데이터 통지가 끝나거나 에러가 발생하는 시점에 별도로 대응할 수도 있습니다. 옵저버 패턴은 '3.1 RxJava와 디자인 패턴'에서 자세히 설명합니다.

RxJava의 또 다른 특징으로 쉬운 비동기 처리를 들 수 있습니다. Reactive Streams 규칙의 근간이 되는 **Observable 규약**이라는 RxJava 개발 가이드라인을 따른 구현이라면 직접 스레드(thread)를 관리하는 번거로움에서 해방될 뿐만 아니라 구현도 간단하게 할 수 있습니다. 또한, 동기 처리나 비동기 처리나 구현 방법에 큰 차이가 없는 것도 RxJava의 특징입니다.

그리고 RxJava는 함수형 프로그래밍의 영향을 받아 함수형 인터페이스를 인자로 전달받는 메서드를 사용해 대부분의 처리를 구현합니다. 이는 입력과 결과만 정해져 있다면 구체적인 처리는 개발자에게 맡길 수 있으므로 더욱 자유로운 구현이 가능함을 의미합니다. 또한, '1.3.2 연산자'에서 자세히 설명하는 함수형 프로그래밍의 원칙을 따르면 처리 작업의 영향 범위를 좁힐 수 있고 동시에 복잡하지 않게 비동기 처리를 할 수 있습니다.

1.2 Reactive Streams

다양한 리액티브 프로그래밍 라이브러리의 표준 사양인 Reactive Streams를 살펴보겠습니다.

1.2.1 Reactive Streams란

RxJava 버전이 1.x에서 2.x로 올라간 배경에는 Reactive Streams가 있습니다. Reactive Streams란 라이브러리나 프레임워크에 상관없이 데이터 스트림을 비동기로 다룰 수 있는 공통 메커니즘으로, 이 메커니즘을 편리하게 사용할 수 있는 인터페이스를 제공합니다.

- Reactive Streams http://www.reactive-streams.org

즉, Reactive Streams는 인터페이스만 제공하고 구현은 각 라이브러리와 프레임워크에서 합니다.

- Reactive Streams Specification for the JVM https://github.com/reactive-streams/reactive-streams-jvm

1.2.2 Reactive Streams의 구성

Reactive Streams는 데이터를 만들어 통지하는 Publisher(생산자)와 통지된 데이터를 받아 처리하는 Subscriber(소비자)로 구성됩니다. Subscriber가 Publisher를 구독(subscribe)하면 Publisher가 통지한 데이터를 Subscriber가 받을 수 있습니다.

- Publisher 데이터를 통지하는 생산자
- Subscriber 데이터를 받아 처리하는 소비자

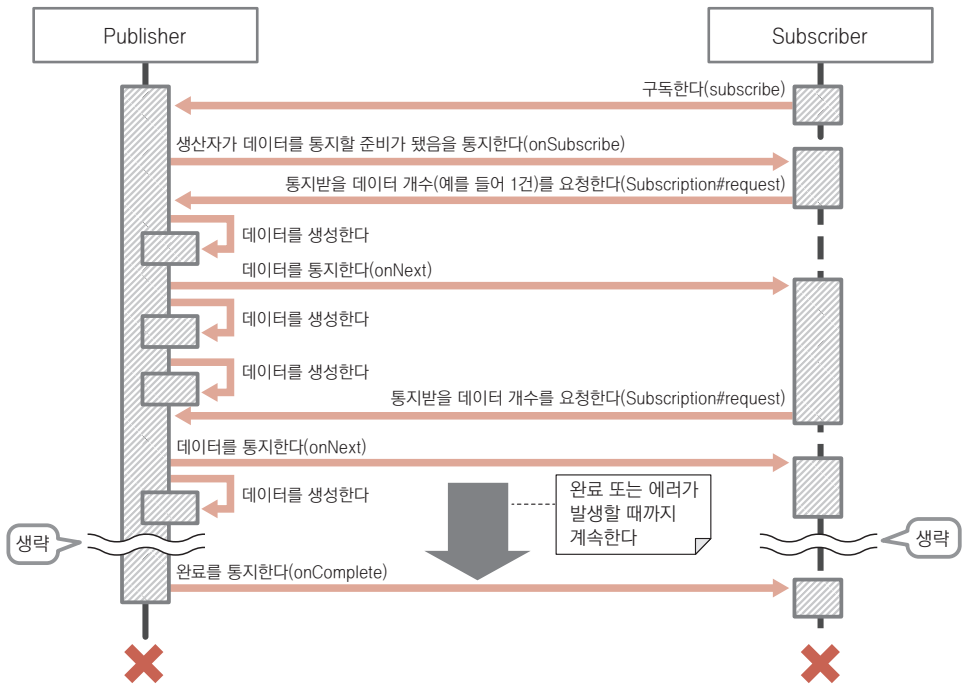
그럼 Publisher가 데이터를 통지한 후 Subscriber가 이 데이터를 받을 때까지의 데이터 흐름을 간단하게 살펴봅시다.

먼저 Publisher는 통지 준비가 끝나면 이를 Subscriber에 통지(onSubscribe)합니다. 해당 통지를 받은 Subscriber는 받고자 하는 데이터 개수를 요청합니다. 이때 Subscriber가 자신이 통지받을 데이터 개수를 요청하지 않으면 Publisher는 통지해야 할 데이터 개수 요청을 기다리게 되

므로 통지를 시작할 수 없습니다.

그다음 Publisher는 데이터를 만들어 Subscriber에 통지(onNext)합니다. 이 데이터를 받은 Subscriber는 받은 데이터를 사용해 처리 작업을 수행합니다. Publisher는 요청받은 만큼의 데이터를 통지한 뒤 Subscriber로부터 다음 요청이 올 때까지 데이터 통지를 중단합니다. 이후 Subscriber가 처리 작업을 완료하면 다음에 받을 데이터 개수를 Publisher에 요청합니다. 이 요청을 보내지 않으면 Publisher는 요청 대기 상태가 돼 Subscriber에 데이터를 통지할 수 없습니다.

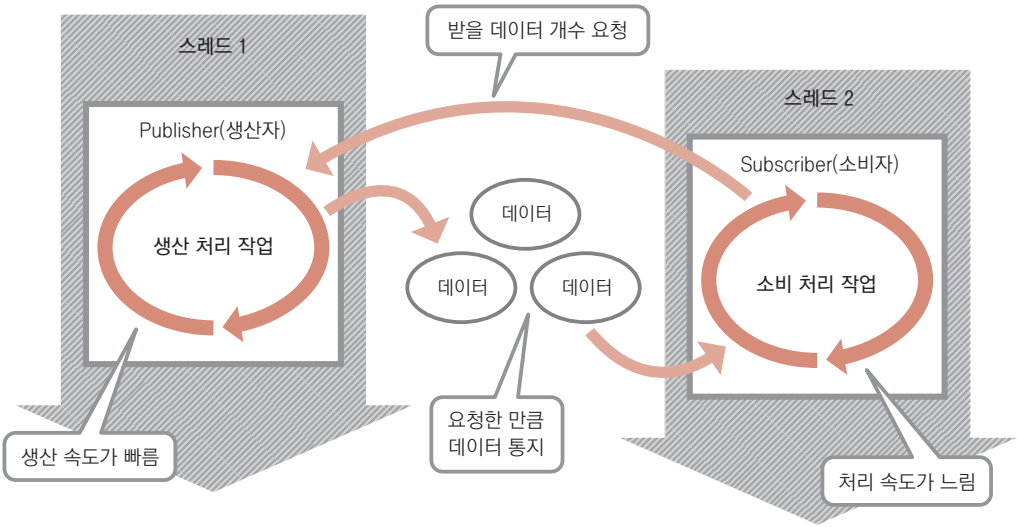
그림 1-5 Publisher와 Subscriber의 시퀀스 다이어그램



Publisher는 Subscriber에 모든 데이터를 통지하고 마지막으로 데이터 전송이 완료돼 정상 종료됐다고 통지(onComplete)합니다. 완료 통지를 하고 나면 Publisher는 이 구독 건에 대해 어떤 통지도 하지 않습니다. 또한, Publisher는 처리 도중에 에러가 발생하면 Subscriber에 발생한 에러 객체와 함께 에러를 통지(onError)합니다.

Subscriber가 Publisher에 통지받을 데이터 개수를 요청하는 것은 Publisher가 통지하는 데이터 개수를 제어하기 위해서입니다. 예를 들어, Publisher와 Subscriber의 처리가 각각 다른 스레드에서 진행되는데 Publisher의 통지 속도가 빠르면 Subscriber가 소화할 수 없을 만큼 많은 처리 대기 데이터가 쌓입니다. 이를 막기 위해 Publisher가 통지할 데이터 개수를 Subscriber가 처리할 수 있을 만큼으로 제어하는 수단이 필요합니다.

그림 1-6 Publisher가 Subscriber에 통지하는 과정



지금까지의 내용을 정리하면 Publisher와 Subscriber는 다음과 같은 4개 프로토콜로 데이터를 통지합니다.

표 1-2 Reactive Streams가 제공하는 프로토콜

프로토콜	설명
onSubscribe	데이터 통지가 준비됐음을 통지
onNext	데이터 통지
onError	에러(이상 종료) 통지
onComplete	완료(정상 종료) 통지

다음으로 Reactive Streams가 제공하는 인터페이스를 살펴보겠습니다. Reactive Streams는 다음 표와 같이 인터페이스 4개를 제공합니다.

표 1-3 Reactive Streams의 인터페이스

인터페이스	설명
Publisher	데이터를 생성하고 통지하는 인터페이스
Subscriber	통지된 데이터를 전달받아 처리하는 인터페이스
Subscription	데이터 개수를 요청하고 구독을 해지하는 인터페이스
Processor	Publisher와 Subscriber의 기능이 모두 있는 인터페이스

Reactive Streams에 선언된 인터페이스는 각각 다음과 같습니다.

예제 1-2 Publisher.java

```
// 데이터를 통지하는 생산자
public interface Publisher<T> {
    // 데이터를 받는 Subscriber 등록
    public void subscribe(Subscriber <? super T> subscriber);
}
```

예제 1-3 Subscriber.java

```
// 데이터를 받아 처리하는 소비자
public interface Subscriber<T> {
    // 구독 시작 시 처리
    public void onSubscribe(Subscription subscription);

    // 데이터 통지 시 처리
    public void onNext(T item);

    // 에러 통지 시 처리
    public void onError(Throwable error);

    // 완료 통지 시 처리
    public void onComplete();
}
```

예제 1-4 Subscription.java

```
// 생산자와 소비자를 연결하는 인터페이스
public interface Subscription {
    // 통지받을 데이터 개수 요청
    public void request(long num);

    // 구독 해지
    public void cancel();
}
```

예제 1-5 Processor.java

```
// Publisher와 Subscriber의 기능이 모두 있는 인터페이스
public abstract interface Processor<T, R> extends Subscriber<T>, Publisher<R> { }
```

Publisher와 Subscriber가 사용하는 Subscription은 통지받을 데이터 개수를 지정해 데이터 통지를 요청하거나 통지받지 않게 구독을 해지할 때 사용하는 인터페이스입니다. Subscription은 Publisher에서 인스턴스가 생성돼 통지 준비가 끝났을 때 호출하는 onSubscribe 메서드의 인자로 Subscriber에 전달됩니다. 이 Subscription을 받은 Subscriber는 Subscription의 메서드를 호출해 데이터 개수를 요청하거나 구독을 해지합니다.

또한, onNext 메서드에서 이 Subscription을 사용하려면 onSubscribe 메서드로 전달받은 Subscription이 Subscriber 내부에 있어야 합니다.

예제 1-6 Subscription 보관

```
publisher.subscribe(new Subscriber<T>() {
    // subscriber 내부에 Subscription 보관하기
    private Subscription subscription;

@Override
    public void onSubscribe(Subscription subscription) {
        // 받은 Subscription을 Subscriber 내부에 보관한다
        this.subscription = subscription;

        // 처음에 통지받을 데이터 개수를 요청한다
        this.subscription.request(num);

        (중략)

    }

    @Override
    public void onNext(T item) {

        (중략)

        // 요청한 데이터를 처리하면 다음 데이터 개수를 요청한다
        subscription.request(num);
    }

    (중략)

});
```

다만 예제 1-6은 Subscription의 처리가 Subscriber 외부에서는 호출되지 않는다는 것을 전제로 구현할 수 있습니다. 하지만 RxJava처럼 외부에서 구독을 해지할 방법을 제공한다면 Subscription이 비동기로 호출돼도 문제가 없게 구현해야 합니다.

추가로 Reactive Streams에 Processor라는 인터페이스가 있습니다. 이 Processor는 Publisher와 Subscriber 모두를 상속받아 데이터 통지와 수신이 가능합니다. 즉, Processor는 다른 Publisher를 구독하거나 다른 Subscriber가 자신을 구독하게 할 수 있습니다.

1.2.3 Reactive Streams의 규칙

Reactive Streams는 앞서 소개한 인터페이스로 데이터를 통지하는 구조를 제공합니다. 하지만 이 구조가 제대로 작동하려면 Reactive Streams의 규칙을 따라야 합니다. 어떤 규칙이 있는지는 Reactive Streams GitHub에 있는 README.md에서 확인할 수 있습니다.

- **reactive-streams-jvm/README.md** https://github.com/reactive-streams/reactive-streams-jvm/blob/master/README.md

Reactive Streams의 기본 규칙은 다음과 같습니다.

- 구독 시작 통지(`onSubscribe`)는 해당 구독에서 한 번만 발생한다.
- 통지는 순차적으로 이루어진다.
- null을 통지하지 않는다.
- Publisher의 처리는 완료(`onComplete`) 또는 에러(`onError`)를 통지해 종료한다.

Reactive Streams에서 구독 시작 통지는 해당 구독에서 한 번만 수행됩니다. 따라서 Subscriber의 `onSubscribe` 메서드는 구독을 시작해 통지 준비가 끝났을 때 한 번만 실행됩니다. 단, 다음 작업은 추천하지 않지만, 처리가 종료된 후에 같은 Publisher와 Subscriber로 `subscribe` 메서드를 호출하면 다시 `onSubscribe` 메서드가 실행됩니다. 이는 처리가 끝난 뒤에 `subscribe` 메서드를 호출하면 새로운 구독을 시작한다고 생각하기 때문입니다. 그러나 같은 인스턴스를 다시 사용해 `subscribe` 메서드를 호출할 때 Publisher나 Subscriber 내부의 관리 상태를 초기화하지 않으면 의도하지 않은 결과가 발생할 수도 있으므로 주의해야 합니다.

또한, Reactive Streams에서는 데이터 통지가 순차적으로 이루어집니다. 즉, 여러 통지를 동시에 할 수 없습니다. 이는 Reactive Streams 사양에 큰 영향을 미친 RxJava의 'Observable 규약

(Observable contract)'[5]이라는 규칙에 따른 것으로, 데이터가 동시에 통지돼 불일치가 발생하는 것을 방지합니다.

그리고 Reactive Streams에서는 null을 통지할 수 없습니다. 만약 null을 통지하면 Reactive Streams에서 `NullPointerException`이 발생합니다. 이는 데이터를 통지할 때만 아니라 에러를 통지할 때도 마찬가지입니다. 이 "null을 통지하지 않는다"라는 사양은 RxJava 1.x 버전과 다른 사양이므로 RxJava 1.x 버전의 프로그램을 RxJava 2.x 버전으로 전환할 때는 주의해야 합니다.

마지막으로 Reactive Streams에서는 완료나 에러를 통지하면 Publisher가 처리를 끝마친 것으로 판단합니다. 이는 완료나 에러 통지를 마친 구독은 더 이상 통지하지 않는다는 의미입니다. 예를 들어, 완료를 통지한 뒤에 에러가 발생했다면 이 에러는 통지하지 않으므로 정상적으로 종료됐다고 생각할 위험성이 있습니다.

추가로 데이터 개수 요청이나 구독 해지를 수행하는 Subscription은 다음과 같은 규칙이 있습니다.

- 데이터 개수 요청에 `Long.MAX_VALUE`를 설정하면 데이터 개수에 의한 통지 제한은 없어진다.
- Subscription의 메서드는 동기화된 상태로 호출해야 한다.

Reactive Streams에서 `Long.MAX_VALUE`를 데이터 개수 요청으로 지정하면 통지할 데이터 개수의 제한이 없어집니다. 그러므로 이 요청을 전송한 후에는 데이터 개수 요청을 보내지 않아도 데이터 통지를 계속해서 받을 수 있습니다.

또한, 요청받은 데이터 개수가 남은 상태에서 추가로 데이터 개수를 요청받으면 새로 요청받은 데이터 개수가 기존 데이터 개수에 추가된다는 점을 주의해야 합니다. 즉, 데이터 개수 요청을 받을 때마다 기존 개수에 더해져 통지 가능한 데이터 개수가 증가합니다. 그래서 이 더해진 결과가 `Long.MAX_VALUE`에 도달하면 통지 가능한 데이터 개수 제한이 없어집니다.

그리고 Subscription의 메서드는 동기화된 상태로 호출해야 합니다. 즉, Subscription의 메서드를 동시에 호출해서는 안 됩니다.

앞의 규칙 이외에도 Reactive Streams에는 세세한 규칙이 있으나 이러한 규칙들은 여기에서 소개한 규칙에 따라 구현하면 자동으로 지켜집니다. RxJava를 사용할 때는 각 통지 메서드와 Subscription의 메서드를 호출할 때 동기화가 이뤄지므로 처리 자체가 스레드 안전(thread safety)한지를 특히 신경 써야 합니다. 이 부분이 제대로 처리되지 않으면 정확한 통지가 안 될 가능성이 큽니다.

5 역주 Observable 규약이란 ReactiveX의 Rx.NET을 설명하는 Rx Design Guidelines에서 비롯됐으며, Observable을 공식적으로 정의한 규칙입니다(참고 URL: http://reactivex.io/documentation/contract.html).

또한 RxJava는 규칙에 따라 구현하지 않아도 문제가 발생하지 않게 내부적으로 구현한 부분도 있습니다. 하지만 이는 문제가 발생하지 않는다고 보장하지 않습니다. 예를 들어, 현재 상태에서 문제가 발생하지 않는다고 해도 향후에 성능 등을 이유로 규칙을 따르지 않는 처리에 대한 구현이 빠질 수도 있습니다. 그러므로 개발자는 현재 상태의 구현을 믿지 말고 Reactive Streams 규칙에 따라 구현해야 합니다.

1.3 RxJava의 기본 구조

RxJava가 어떻게 구성됐는지 살펴보겠습니다.

1.3.1 기본 구조

RxJava는 데이터를 만들고 통지하는 생산자와 통지된 데이터를 받아 처리하는 소비자로 구성됩니다. 이 생산자를 소비자가 구독해 생산자가 통지한 데이터를 소비자가 받게 됩니다. RxJava에서 이 생산자와 소비자의 관계는 크게 두 가지로 나뉩니다. 하나는 Reactive Streams를 지원하는 Flowable과 Subscriber, 다른 하나는 Reactive Streams를 지원하지 않고 배압 기능이 없는 Observable과 Observer입니다

표 1-4 RxJava의 생산자와 소비자 구성

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

Flowable은 Reactive Streams의 생산자인 Publisher를 구현한 클래스고, Subscriber는 Reactive Streams의 클래스입니다. 그래서 기본적인 메커니즘은 Reactive Streams와 같습니다. 생산자인 Flowable로 구독 시작(`onSubscribe`), 데이터 통지(`onNext`), 에러 통지(`onError`), 완료 통지(`onComplete`)를 하고 각 통지를 받은 시점에 소비자인 Subscriber로 처리합니다. 그리고 Subscription으로 데이터 개수 요청과 구독 해지를 합니다.

이에 비해 RxJava 2.x 버전의 Observable과 Observer 구성은 Reactive Streams를 구현하지 않아서 Reactive Streams 인터페이스를 사용하지 않습니다. 하지만 기본적인 메커니즘은 Flowable과 Subscriber 구성과 거의 같습니다. 생산자인 Observable에서 구독 시작(onSubscribe), 데이터 통지(onNext), 에러 통지(onError), 완료 통지(onComplete)를 하면 Observer에서 이 통지를 받습니다.

다만 Observable과 Observer 구성은 통지하는 데이터 개수를 제어하는 배압 기능이 없기 때문에 데이터 개수를 요청하지 않습니다. 그래서 Subscription을 사용하지 않고, Disposable이라는 구독 해지 메서드가 있는 인터페이스를 사용합니다. 이 Disposable은 구독을 시작하는 시점에 onSubscribe 메서드의 인자로 Observer에 전달됩니다. Disposable에는 구독 해지를 위한 다음 2개의 메서드가 있습니다.

표 1-5 Disposable의 메서드

메서드	설명
dispose	구독을 해지한다.
isDisposed	구독을 해지하면 true, 해지하지 않으면 false를 반환한다.

그래서 Observable과 Observer 간에 데이터를 교환할 때 Flowable과 Subscriber처럼 데이터 개수 요청은 하지 않고 데이터가 생성되자마자 Observer에 통지됩니다.

1.3.2 연산자

RxJava에서는 생산자(Flowable/Observable)가 통지한 데이터가 소비자(Subscriber/Observer)에게 도착하기 전에 불필요한 데이터를 삭제하거나 소비자가 사용하기 쉽게 데이터를 변환하는 등 통지하는 데이터를 변경해야 할 때가 있습니다. 이때 사용하는 메서드는 Flowable/Observable의 메서드에서 새로운 Flowable/Observable을 반환하며, 이 메서드를 연결해나감으로써 최종 데이터를 통지하는 Flowable/Observable을 생성합니다. 이 내용은 자바 8의 Stream을 사용한 경험이 있는 사람이라면 쉽게 이해할 수 있습니다. RxJava에서는 이처럼 통지하는 데이터를 생성하거나 필터링 또는 변환하는 메서드를 **연산자**(operator)라고 합니다.