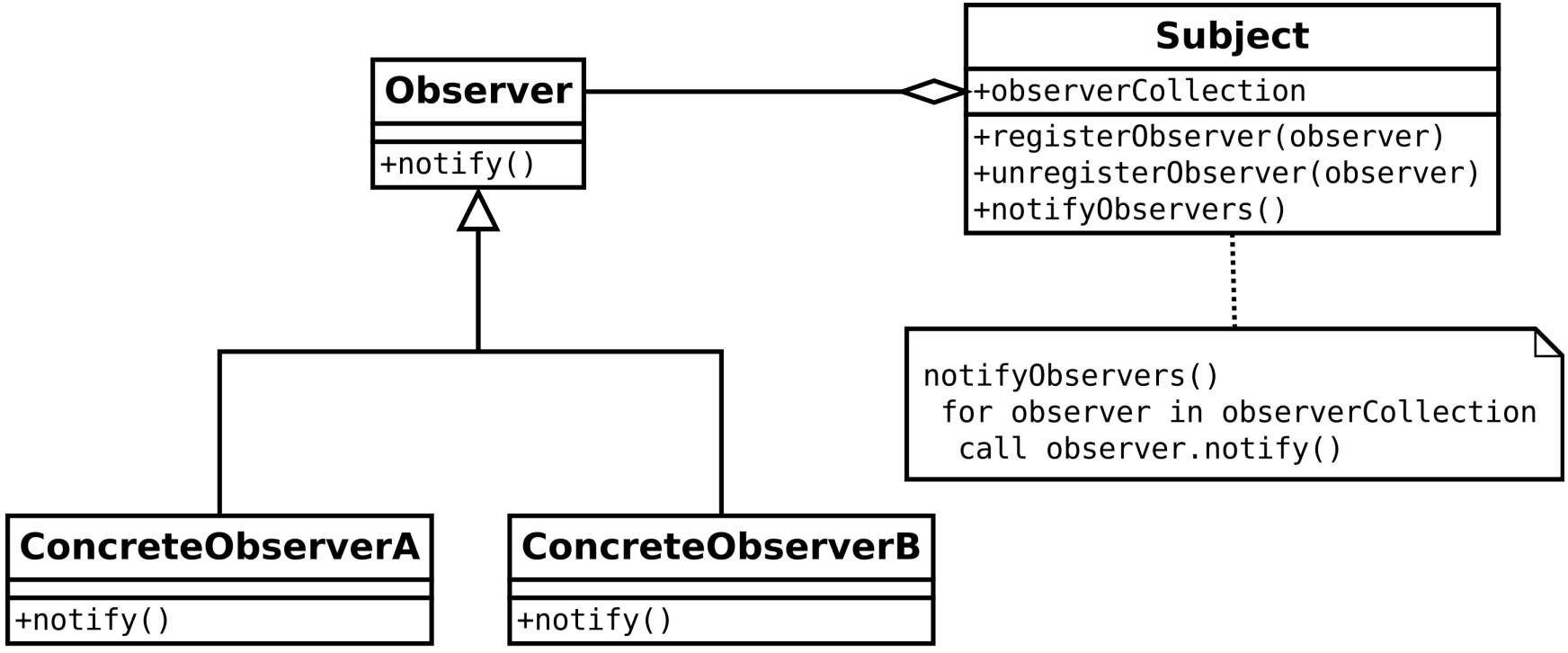
Observable & Scheduler

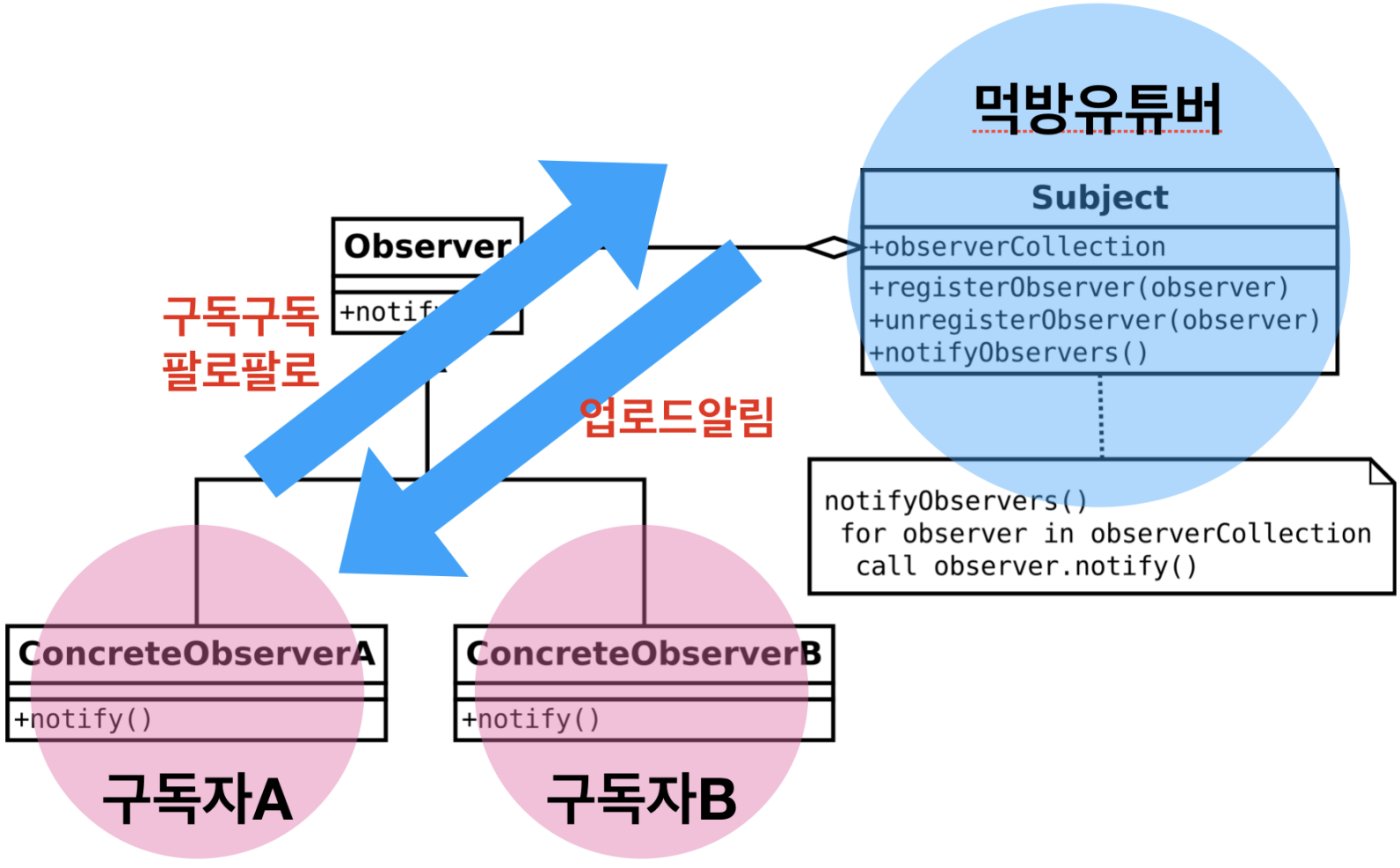
지난 번에 ReactiveX가 무엇인지 알아보았다. **관측 가능한 스트림을 사용한 비동기 프로그래밍 용 API** 이라는 말에서 Observer 패턴을 사용한다는 얘기가 있었는데, 일단 Observable을 알아보기 전에 *Observer 패턴*이 무엇인지부터 짚고 가보자.

Observer Pattern

google에 "Observer 패턴"이라고 검색하면 제일 많이 나오는 그림부터 등판하면서 시작해보자.



이게 이해 쉽게 설명하려고 그린 그림인지 모르겠는데, 약간 변경해보겠다.



그러니까 Observer 패턴은 유튜브에 비교하면 이해하기가 쉬운데,

- Subject : 컨텐츠를 만들어내는 유투버 -> RxJava - Observable
- Observer : 유투버를 구독하는 구독자 -> RxJava - Subscriber

> **옵서버 패턴**(observer pattern)은 객체의 상태 변화를 관찰하는 관찰자들, 즉 Observer들의 목록을 객체에 등록하여 상태 변화가 있을 때마다 메서드 등을 통해 객체(Subject)가 직접 목록의 각 Observer에게 통지하도록 하는 디자인 패턴이다. 주로 분산 이벤트 핸들링 시스템을 구현하는 데 사용된다. 발행/구독 모델로 알려져 있기도 하다.

위키백과 "옵저버 패턴" 참고

- Observable은 데이터의 변화가 발생하는, 이벤트를 발생시키는 데이터 소스이고 모든 Rx는 이 녀석으로부터 시작된다.
- Subscriber는 발생된 이벤트를 받아서 처리하는 녀석으로, Observable은 subscribe()함수를 호출해야지만 데이터를 구독자에게 발행한다.

이제부터 우리는 컨텐츠를 만들어내는 유투버가 되는거다.
그리고 거기에 구독을 거는 구독자들도 만들어본다.

Observable 사용하기

전에
RxJava 1.x와 RxJava 2.x에서의 데이터소스 비교가 있겠습니다.

RxJava 1.x	RxJava 2.x	
Observable	Observable	
	Maybe	데이터가 발행될 수 있거나 발행되지 않고도 완료되는 경우
	Flowable	데이터가 발행되는 속도 > 구독자가 처리하는 속도 -> 배압 이슈 대응 기능
Single	Single	

Observable은 세 가지 알림을 전달할 수 있다.

- onNext : 이벤트 발행
- onComplete : 이벤트 발행 완료
- onError : 에러 발생. Observable 실행 종료.

그러니까, 구독자(subscriber)는 위의 세 가지 타입의 이벤트를 받아서 처리할 수 있는거다.

Observable 이벤트 발행하기

Observable을 생성할 때는 특정 함수들을 통해 호출할 수 있는데, 그걸 팩토리 함수라고 한댄다. 여러 종류의 함수들 중에 사용할 목적에 따라 맞는걸로 골라서 내가 원하는 Observable을 만들면 된다.
먹방 유투버를 할지, 뷰티 유투버를 할지, 여행 유투버를 할지 구독자들을 고려하여 적당한 컨텐츠를 선택하는 것과 같다.

팩토리 함수 종류

- RxJava 1.x : create(), just(), ~~from()~~
- RxJava 2.x : fromArray(), fromIterable(), fromCallable(), fromFuture(), fromPublisher()
- 기타 : interval(), range(), timer(), defer() 등등...

Observable 생성하기

just() 를 사용하여 Observable을 생성해보자.

```
System.out.println("create Observable")
val observable = Observable.just("마블")
```

결과값

```
I/System.out: create Observable
```

Observable은 subscribe()함수를 호출해야지만 데이터를 구독자에게 발행하기 때문에, 구독자를 만들어서 "마블"이라는 키워드를 받아보자.

subscribe()를 사용하여 Observable에서 이벤트를 받아보자.

```
        System.out.println("create Observable")
        val observable = Observable.just("마블")
        System.out.println("do subscribe")
        observable.subscribe(object : Subscriber<String>(
            override fun onNext(t: String?) {
                System.out.println("onNext : $t")
            }

            override fun onCompleted() {
                System.out.println("onCompleted")
            }

            override fun onError(e: Throwable?) {
                System.out.println("onError : $e")
            }
        })
```

결과출력

```
I/System.out: create Observable
I/System.out: do subscribe
I/System.out: onNext : 마블
I/System.out: onCompleted
```

Observable을 생성하고 그로부터 이벤트를 받아서 처리하는 subscribe()까지 호출해보았다.

subscribe() 함수

- subscribe() : onError 이벤트가 발생했을 때만 OnErrorNotImplementedException을 던진다.
- subscribe(인자1) : onNext 이벤트를 받아서 처리. onError가 발생하면 OnErrorNotImplementedException을 던진다.
- subscribe(인자1, 인자2) : onNext, onError 처리
- subscribe(인자1, 인자2, 인자3) : onNext, onError, onComplete 처리

위의 함수들은 모두 Disposable 인터페이스 객체를 리턴한다.

> **Disposable**

> 구독해지의 기능을 하는 객체. Observable이 onComplete를 이벤트를 보내면 자동으로 dispose()를 호출하여 더 이상 데이터를 발행하지 않도록 구독을 해지한다.

여기까지 정리

1. Observable은 데이터를 발행하는 데이터 소스이며, create(), just(), fromXXX(), interval(), range() 등의 팩토리 함수를 활용해 목적에 맞는 Observable을 생성할 수 있다.
2. Subscriber는 Observable에서 발행한 데이터를 onNext, onCompleted, onError 세 가지 함수를 통해 받아서 처리할 수 있다.
3. Observable은 subscribe()로 호출받기 전까지는, 이벤트를 발행하지 않는다.

just() 함수

- 인자로 넣은 데이터를 차례로 발행하는 Observable을 생성한다.
- 인자값은 최대 10개의 값까지 넣을 수 있다. (같은 타입에 한하여)

create() 함수

- 자동으로 데이터를 발행하는 just()와는 달리, create()함수는 onNext, onComplete, onError 알림을 개발자가 직접 호출해야한다.
- 모든 데이터를 발행하면 onComplete()함수를 호출해야 한다.

```
        System.out.println("create Observable")
        val observable = Observable.create<String> { s->
            s.onNext("HYDRxA")
            s.onNext("viSIon")
            s.onNext("Captain Marble")
            s.onNext("thanOS")
            s.onCompleted()
        }
        System.out.println("do subscribe")
        observable.subscribe(System.out::println)
```

결과출력

```
I/System.out: create Observable
I/System.out: do subscribe
I/System.out: HYDRxA
I/System.out: viSIon
I/System.out: Captain Marble
I/System.out: thanOS
```

Single 클래스

- Observable의 특수한 형태. 오직 1개의 데이터만 발행하도록 한정됨. (Observable은 무한하게 데이터를 발행할 수 있다.)
- 보통 결과가 유일한 서버 API를 호출할 경우 유용하게 사용한다.
- onSuccess, onError함수로 구성.
- 데이터 하나가 발행과 동시에 종료된다
  -> onNext + onComplete => onSuccess

Maybe 클래스

- 역시 Observable의 특수한 형태로, 최대 1개의 데이터를 발행할 수도, 발행 안 하고 종료할 수도 있는 클래스. (그래서 'Maybe'인가봄)
- onSuccess, onError, onComplete 함수로 구성.

Flowable 클래스

- Observable의 성능을 향상시키기 위해 도입된 클래스.
- 배압(backpressure)이슈 해결.
- 설명 읽어도 잘 모르겠으니 다음에 자세히 알아보자.

뜨거운 Observable과 차가운 Observable

여태까지 예제는 subscribe()를 함수를 호출하지 않으면 데이터를 발행하지 않는 Observable을 보았는데, 그런 Observable을 차가운 Observable이라고 한다.

> 누군가가 구독할 때까지 데이터를 준비해두고 있다가 구독이 이루어지면 데이터를 처음부터 발행받음.

반면, 뜨거운 Observable은 구독 존재 여부와는 상관없이 데이터를 발행한다.

> 누군가가 구독을 하면 그 시점에 발행되는 데이터를 받을 수 있다.

그러니까, 차옵은 일반 유투브를 시청하는 것, 뜨옵은 유투브 라이브 방송을 시청하는 느낌인거다.
미리 준비된 고정된 데이터가 아니라 실시간으로 갱신되어야하는 데이터를 처리할 경우에 뜨옵을 활용할 수 있다.

뜨거운 Observable 사용 시 주의점

- 배압을 고려해야 한다.
- 데이터 발행 속도 - 구독자 처리 속도 차이가 클 때 문제 발생.

Subject 클래스

차옵 -> 뜨옵 으로 객체를 변환하는 방법 중 하나.
Subject 클래스는 Observable의 속성과 Subscriber의 속성 모두를 갖고 있기 때문에 데이터를 발행할 수도, 발행된 데이터를 바로 처리할 수도 있다.

> 유투버가 영상을 보는 방송을 라이브로 브로드캐스트 한다면?

지금까지 Observable에 대해서 깊게는 아니지만, 대략적인 컨셉을 보는 정도의 느낌적인 느낌으로 알아보았다.
생각보다 양이 많아서 Scheduler는 다음에 알아보아야겠다.