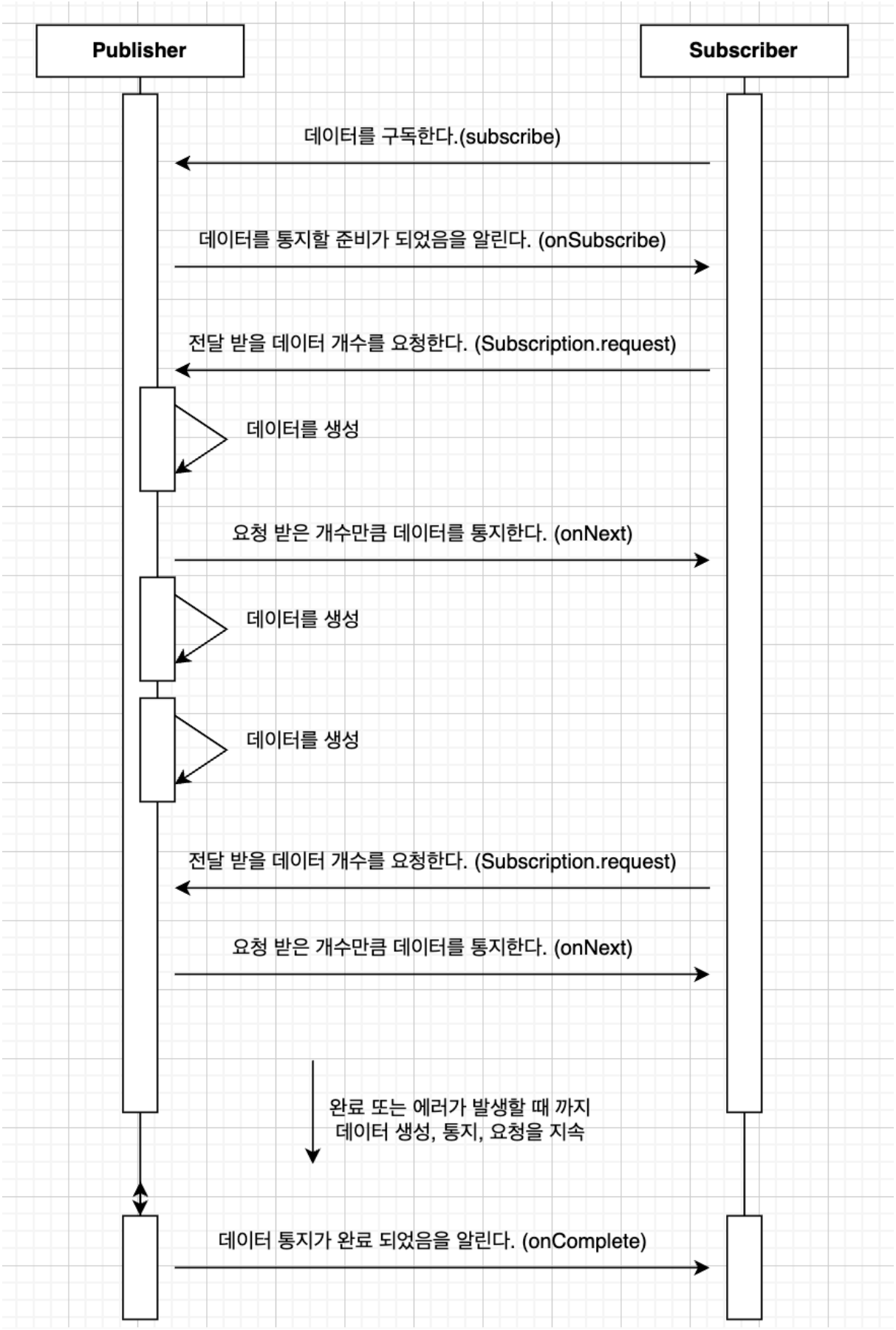
2일차 - Chapter 2. ~ Chapter 7.

Chapter 2. 리액티브 스트림즈

리액티브 스트림즈

리액티브 라이브러리를 어떻게 구현할지 정의해 놓은 별도의 표준사양.

⇒ 데이터 스트림을 Non-Blocking이면서 비동기적인 방식으로 처리하기 위한 리액티브 라이브러리 표준


Publisher
Subscriber
데이터를 구독한다.(subscribe)
데이터를 통지할 준비가 되었음을 알린다. (onSubscribe)
전달 받을 데이터 개수를 요청한다. (Subscription.request)
데이터를 생성
요청 받은 개수만큼 데이터를 통지한다. (onNext)
데이터를 생성
데이터를 생성
전달 받을 데이터 개수를 요청한다. (Subscription.request)
요청 받은 개수만큼 데이터를 통지한다. (onNext)
완료 또는 에러가 발생할 때 까지
데이터 생성, 통지, 요청을 지속
데이터 통지가 완료 되었음을 알린다. (onComplete)

서로 다른 스레드에서 비동기적으로 상호작용하기에 Subscription.request를 통해 데이터 개수를 제어하여 통신하게 됨.

Reactive streams	Description
Publisher	데이터를 생성하고 통지(발행, 게시, 방출)하는 역할을 한다.
Subscriber	구독한 Publlisher로부터 통지된 데이터를 받아서 처리하는 역할을 한다.
Subscrition	Publisher에 요청할 데이터의 개수를 지정하고, 데이터의 구독을 취소하는 역할을 한다.
Processor	Publisher, Subscriber의 기능을 모두 가지고 있다. Subscriber로서 다른 Publisher를 구독할 수 있고, Publisher로서 다른 Subscriber가 구독할 수 있다.

왜 Subscriber가 아닌 Publisher에 subscribe 메서드가 정의되어 있을까?

- 카프카는 메세지 브로커가 있고, 메세지 브로커 내의 특정 토픽으로 전송하고 Subscriber가 받기만 하는 느슨한 결합
- 리액티브 스트림즈는 Subscriber가 구독한 것은 맞는데 Publisher가 subscribe 메서드의 파라미터인 Subscriber를 동록하는 형태로 구독이 이루어짐

- **Publisher** : subscribe()
- **Subscriber** : onSubscribe(), onNext(), onError(), onComplete()
- **Subscription** : request, cancel
- **Processor** : interface Processor<T, R> extends `Subscriber<T>`, `Publisher<R>` 빼곤 메서드 X

구현 규칙

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

Chapter 3. Blocking IO vs Non-Blocking IO

I/O 작업 : 파일 읽고 기록하기, DB I/O, 네트워크 I/O, 예시는 네트워크 I/O

Blocking IO

- 말 그대로 따른 짓 못하게 블락, 그래서 멀티스레딩으로 차단 시간에 딴 짓할 수 있는데, 문제가 있음
  - **컨텍스트 스위칭** : 스레드가 여러 프로세스를 번갈아가며 실행할때 발생하는 비용
    프로세스 바꿔서 실행할 때 PCB(Process Control Block)(스레드는 TCB) 저장, 로드하는데 이 때 CPU가 대기한다.
  - **과다한 메모리 사용** : 스레드가 늘어날수록 스택 영역의 메모리 사용량이 비례하게 늘어남
    JVM 디폴트 스택 사이즈는 1024KB (요청당 하나의 스레드 할당)
  - **스레드 풀 응답 지연** : 대량의 요청이 들어오면 스레드 풀에서 사용 가능한 유휴 스레드가 없어 응답 지연이 발생할 가능성 높아짐
    스레드 생성/수거에 비용 듬, 스레드풀 다 쓰면 응답 지연

Non-Blocking IO

- IO 작업 안 기다리고 딴 일 한다 (**스레드가** 차단되지 않음)
  - 적은 수의 스레드만 사용해 스레드 전환 비용 낮음 → 효율적인 CPU 사용
  - CPU를 많이 사용하면 성능에 악영향, 요청에서 응답까지의 전체과정에서 Blocking IO 하나라도 있으면 이점이 없어짐

Spring WebFlux

spring framework 진영의 Non-Blocking I/O 구현체.

- 비동기 Non-Blocking I/O를 사용함으로써 적은 수의 쓰레드로 많은 수의 요청 처리

- Netty 같은 비동기 기반 서버 엔진 사용

Non-Blocking 통신이 적합한 시스템

1. 학습 난이도 (특히, Reactive Streams에 대한 이해)
2. 개발인력, 유지보수 고려
3. 대용량 트래픽이 발생하는 시스템
4. MSA 기반 시스템 (다른 서비스 호출이 많아 I/O구간 많으므로), 스트리밍 또는 실시간 시스템

Chapter 4. 리액티브 프로그래밍을 위한 사전 지식

리액티브 프로그래밍을 잘 사용하기 위해서는 기본적으로 함수형 프로그래밍 기법이 받쳐줘야 함.

함수형 인터페이스

함수를 값으로 취급할 수 있는 기능(일급 시민)

인터페이스에 단 하나의 추상 메서드만 정의되어 있음.

람다 표현식

익명 구현 객체를 간단하게 전달하는 방식

함수형 인터페이스를 구현한 클래스의 인스턴스를 람다 표현식으로 작성해서 전달함

ex) Comparator

메서드 레퍼런스

좀 더 간결하게 작성가능

```
(Car car) → car.getCarName() = Car::getCarName
```

함수 디스크립터

함수 서술자

일반화된 람다 표현식을 통해서 이 함수형 인터페이스가 어떤 파라미터를 가지고, 어떤 값을 리턴하는지 설명해주는 역할

람다 표현식의 파라미터 개수와 타입, 리턴 타입을 보고 어떤 함수형 인터페이스인지 알고 싶을 때 확인할때 유용함

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 등
Supplier<T>	( ) -> T	BooleanSupplier, IntSupplier, LongSupplier 등
UnaryOperator<T>	T -> T	IntUnaryOperator, LongUnaryOperator 등
BinaryOperator<T>	(T, T) -> T	IntBinaryOperator, LongBinaryOperator 등
BiPredicate<L, R>	(L, R) -> boolean	
BiConsumer<T, U>	(T, U) -> void	ObjIntConsumer<T>, ObjLongConsumer<T> 등
BiFunction<T, U, R>	(T, U) -> R	ToIntBiFunction<T>, ToLongBiFunction<T> 등

Chapter 5. Reactor 개요

Reactor는 리액티브 스트림즈 사양을 구현한 여러 구현체 중 하나.

Spring WebFlux의 핵심 역할

- **Reactive Streams**

- **Non-Blocking**
- **Java’s functional API** - 함수형 프로그래밍
- **Flux[N]** - 0개부터 N개의 데이터를 Emit 가능한 Publisher
- **Mono[0|1]** - 데이터를 0개 or 1개만 Emit 가능한 Publisher
- **Well-suited for microservices**
- **Backpressure-ready network** - Publisher로부터 전달받은 데이터를 처리하는데 과부화 방지

Chapter 6. 마블 다이어그램

Reactor에서 지원하는 Operator를 이해하는데 중요한 역할.

구슬 도표

Chapter 7. Cold Sequence와 Hot Sequence

Cold는 무언가를 새로 시작하고, Hot은 무언가를 새로 시작하지 않는 것.

ex) HotSwap - 전원을 끄지않고 교체 가능

Cold Sequence

Subscriber가 구독할 때마다 데이터 흐름이 처음부터 다시 시작되는 Sequence

구독 시점이 달라도 구독할 때마다 Publisher가 데이터를 emit하는 과정을 처음부터 다시 시작하는 데이터 흐름

⇒ 구독할 때마다 Sequence 타임라인이 생김

Hot Sequence

구독이 발생한 시점 이전에 Publisher로부터 emit 된 데이터는 Subscriber가 전달받지 못하고 구독이 발생한 시점 이후에 emit된 데이터만 전달 받을 수 있음

예를 들어 구독이 3번 이뤄질 경우 Cold Sequence는 3개의 타임라인이 생기겠지만, Hot Sequence는 한개의 타임라인만 가지고 있다는 것.

- share(), cache() 등의 Operator를 사용해서 Cold Sequence를 Hot Sequence로 변환할 수 있음
- Hot Sequence는 Suscriber의 최초 구독이 발생해야 Publisher가 데이터를 emit하는 Warm up과 Subsciber의 구독 여부와 상관없이 데이터를 emit하는 Hot으로 구분할 수 있음