

Reactive Programming

Index

1. Preliminary of Asynchronous Programming

- Preliminary
- Aysnchronous Programming
- When do we have to use Async?

2. Reactive Programming

- Code Scalability
- Reactive Programming (Reactive Extension)

Index

1. Preliminary of Asynchronous Programming

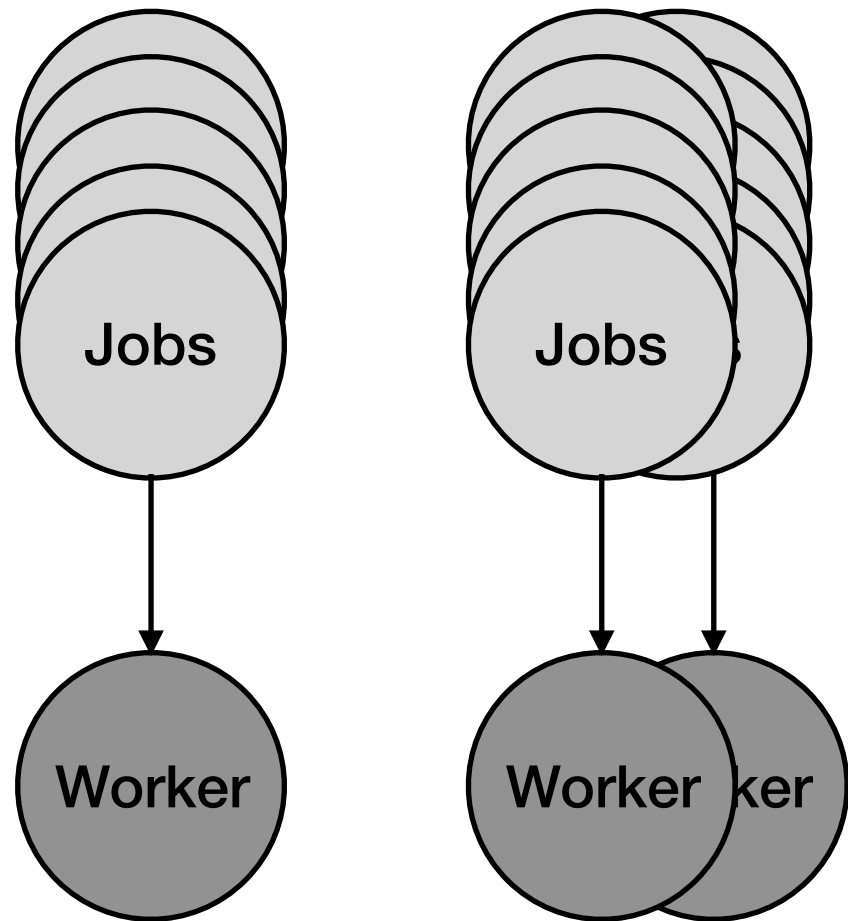
- **Preliminary**
- Aysnchronous Programming
- When do we have to use Async?

2. Reactive Programming

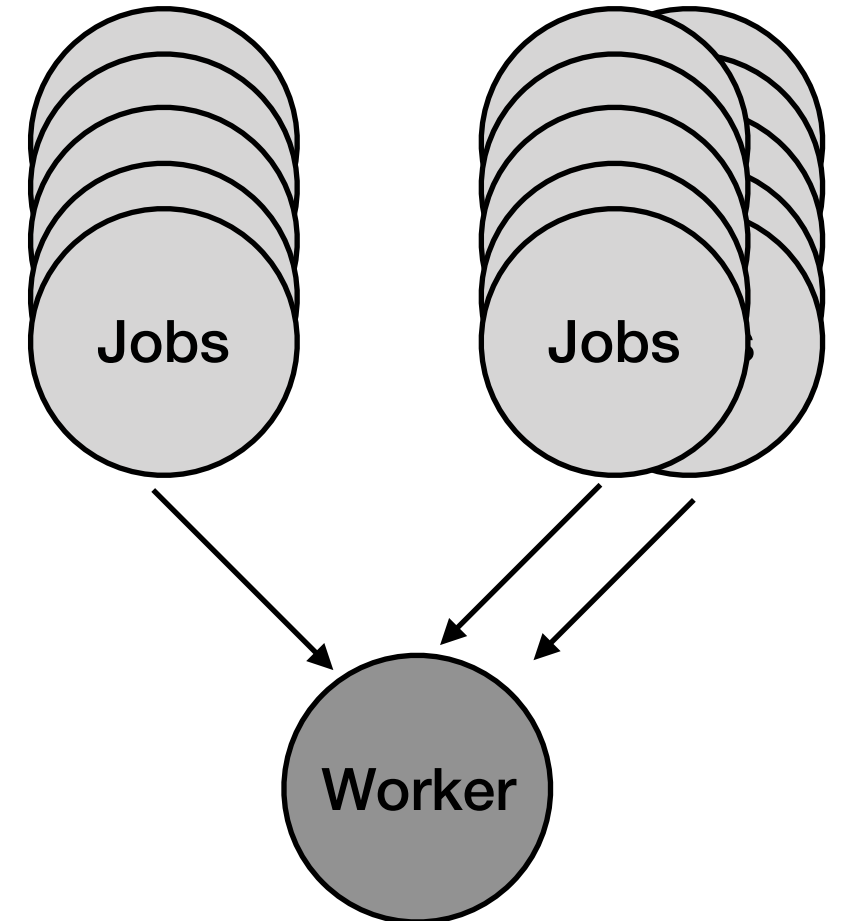
- Code Scalability
- Reactive Programming (Reactive Extension)

Preliminary

Parallelism - Concurrency



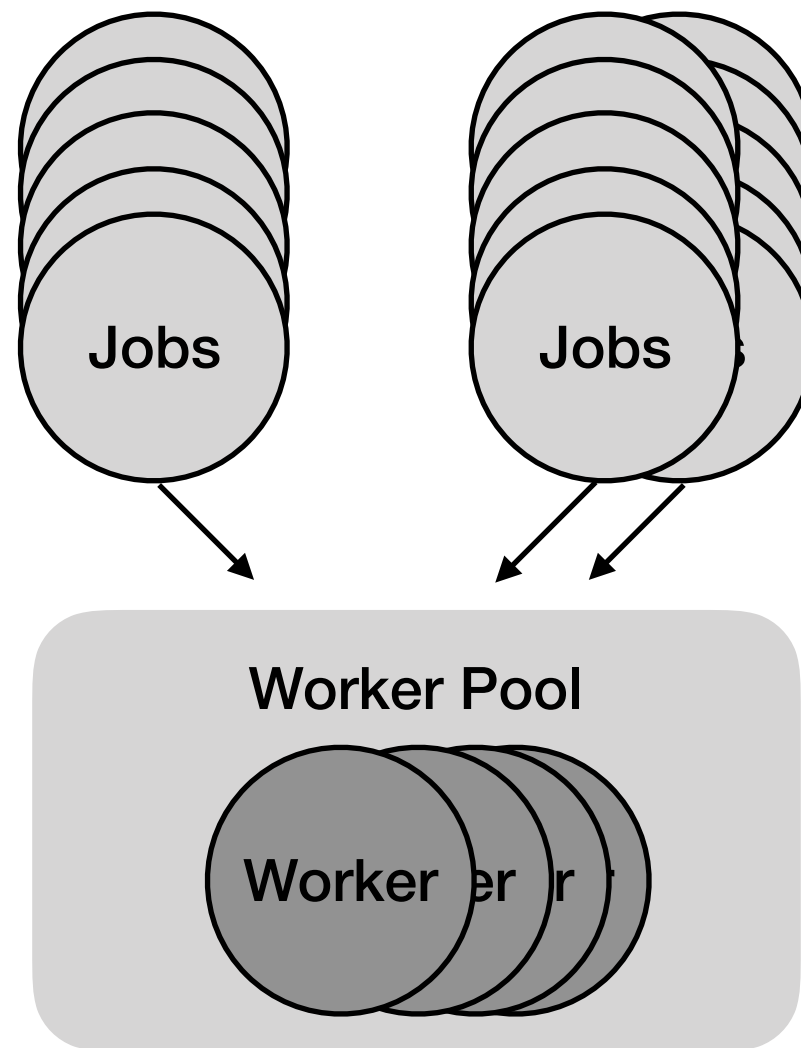
일꾼이 여러개
여러개 일꾼이 “동시에” 일을 하면서 처리



일꾼 갯수 상관 X
일을 스케줄링을 통해 처리하면서
마치 동시에 하는 것처럼 처리

Preliminary

Parallelism + Concurrency



당연히 2가지 다 확보가능

Preliminary

Blocking I/O - Non Blocking I/O

```
while not_finished:  
    data = socket.recv(buf_size)  
    do_something(data)
```

소켓에서 데이터를 받고 로직을 진행 (Imperative 하게)

Preliminary

Blocking I/O - **Non Blocking I/O**

```
while not_finished:
    try:
        data = socket.recv(buf_size)
        do_something(data)
    except socket.error as e:
        if e.args[0] in _ERRNO_WOULDBLOCK:
            # Do something else
```

데이터를 읽거나 쓰는 과정에서 기다리지 않음

Preliminary

Asynchronous Programming

```
data = yield tornado.iostream.read_until('\r\n')
```

⋮

Implementation Details

Blocking or Non-Blocking socket I/O

**Asynchronous programming 의 Implementation detail 를
Non-blocking i/o 든지 blocking i/o 이든지 상관 X**

Index

1. Preliminary of Asynchronous Programming

- Preliminary
- **Aysnchronous Programming**
- When do we have to use Async?

2. Reactive Programming

- Code Scalability
- Reactive Programming (Reactive Extension)

Asynchronous Programming

Providing **Concurrency** by Scheduling Events

이벤트를 스케줄링하면서 동시성을 제공하는 것이 목적

Asynchronous Programming

How communicate with a scheduler?

Callback, Future, Promise, Await

**Asynchronous framework 를 사용할 때 중요한건 이 프레임워크가
어떻게 유저 코드랑 스케줄링 해주는 스케줄러랑 통신을 하느냐이다**

Index

1. Preliminary of Asynchronous Programming

- Preliminary
- Aysnchronous Programming
- **When do we have to use Async?**

2. Reactive Programming

- Code Scalability
- Reactive Programming (Reactive Extension)

When do we have to use Async?

Massive I/O

CPU bound job에서는 쓸모 X

Index

1. Preliminary of Asynchronous Programming

- Preliminary
- Aysnchronous Programming
- When do we have to use Async?

2. Reactive Programming

- **Code Scalability**
- Reactive Programming (Reactive Extension)

Code Scalability

Everything must be **compositional**

Code Scalability

Remind

Asynchronous Frameworks (tornado, asyncio)

```
data = yield tornado.iostream.read_until('\r\n')
```

much more compositional

Non-Blocking I/O

```
while not_finished:
    try:
        data = socket.recv(buf_size)
        do_something(data)
    except socket.error as e:
        if e.args[0] in _ERRNO_WOULDBLOCK:
            # Do something else
```


Code Scalability

Asynchronous Frameworks (tornado, asyncio)

```
data = yield tornado.iostream.read_until('\r\n')
```

Is It enough?

NO !

Code Scalability

Multiple Async HTTP Calls

Multiple Async HTTP Calls and Take Fastest Response

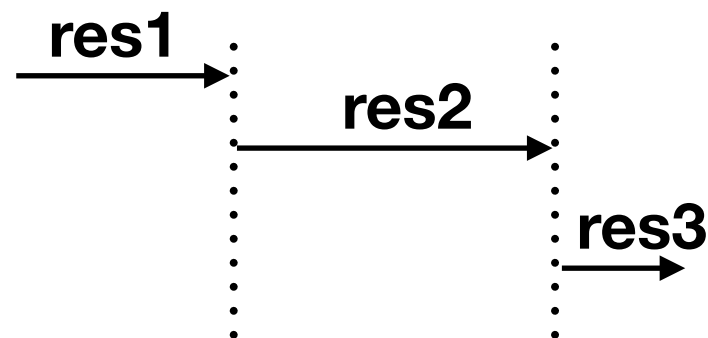
Code Scalability

Multiple Async HTTP Calls - Asynchronous Frameworks

```
res1 = yield service1_api_call()  
res2 = yield service2_api_call()  
res3 = yield service3_api_call()
```

```
data = res1 + res2 + res3
```

Must be refactored !

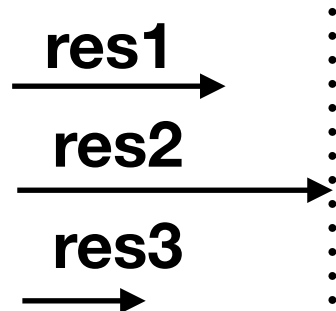


Code Scalability

Multiple Async HTTP Calls - Asynchronous Frameworks

```
futures = [service1_api_call(),  
           service2_api_call(),  
           service3_api_call()]
```

```
data = []  
for future in futures:  
    data.append(yield future))
```



Code Scalability

Multiple Async HTTP Calls - Rx

```
Observable.merge(service1_api_call(),  
                 service2_api_call(),  
                 service3_api_call())  
            .map(lambda data: ...)
```

Code Scalability

Multiple Async HTTP Calls and Take Fastest Response
- Asynchronous Frameworks

.... ???

Code Scalability

**Multiple Async HTTP Calls and Take Fastest Response
- Rx**

```
Observable.merge(service1_api_call(),  
                 service2_api_call(),  
                 service3_api_call())  
    .take(1)  
    .map(lambda data: ...)
```

Code Scalability

Reactive Programming (Rx)

much more compositional

Asynchronous Frameworks (tornado, asyncio)

```
data = yield tornado.iostream.read_until('\r\n')
```

much more compositional

Non-Blocking I/O

```
while not_finished:
    try:
        data = socket.recv(buf_size)
        do_something(data)
    except socket.error as e:
        if e.args[0] in _ERRNO_WOULDBLOCK:
            # Do something else
```


Index

1. Preliminary of Asynchronous Programming

- Preliminary
- Aysnchronous Programming
- When do we have to use Async?

2. Reactive Programming

- Code Scalability
- **Reactive Programming (Reactive Extension)**

Reactive Programming

Async 한 상황에서 Async 한 데이터를 어떻게 처리할 것인지에 대한 아이디어
Async 한 작업을 Functional 하게 처리하는 아이디어

아이디어 => Stream 이라는 것으로 연결하고 그 Stream 에 데이터를 흘려보내라

Reactive Programming

Reactive Libraries

Reactive Extension, Sodium, ReactiveCocoa ...

Reactive Programming

Observable

Generator

Operator

map

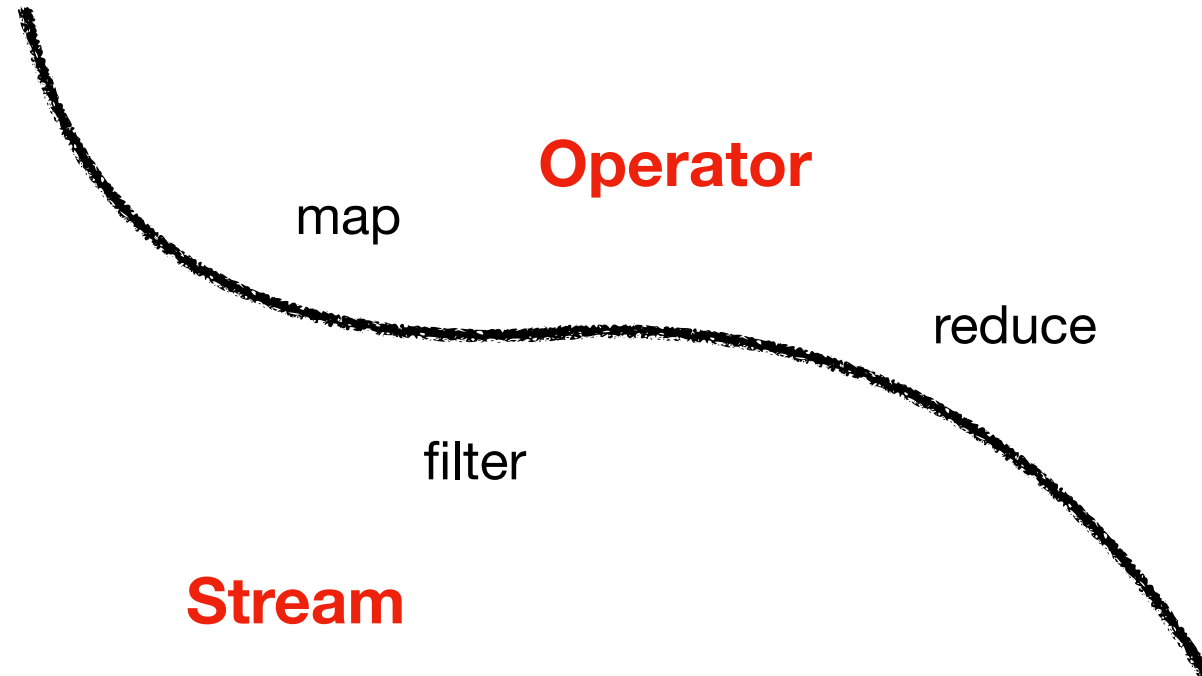
reduce

filter

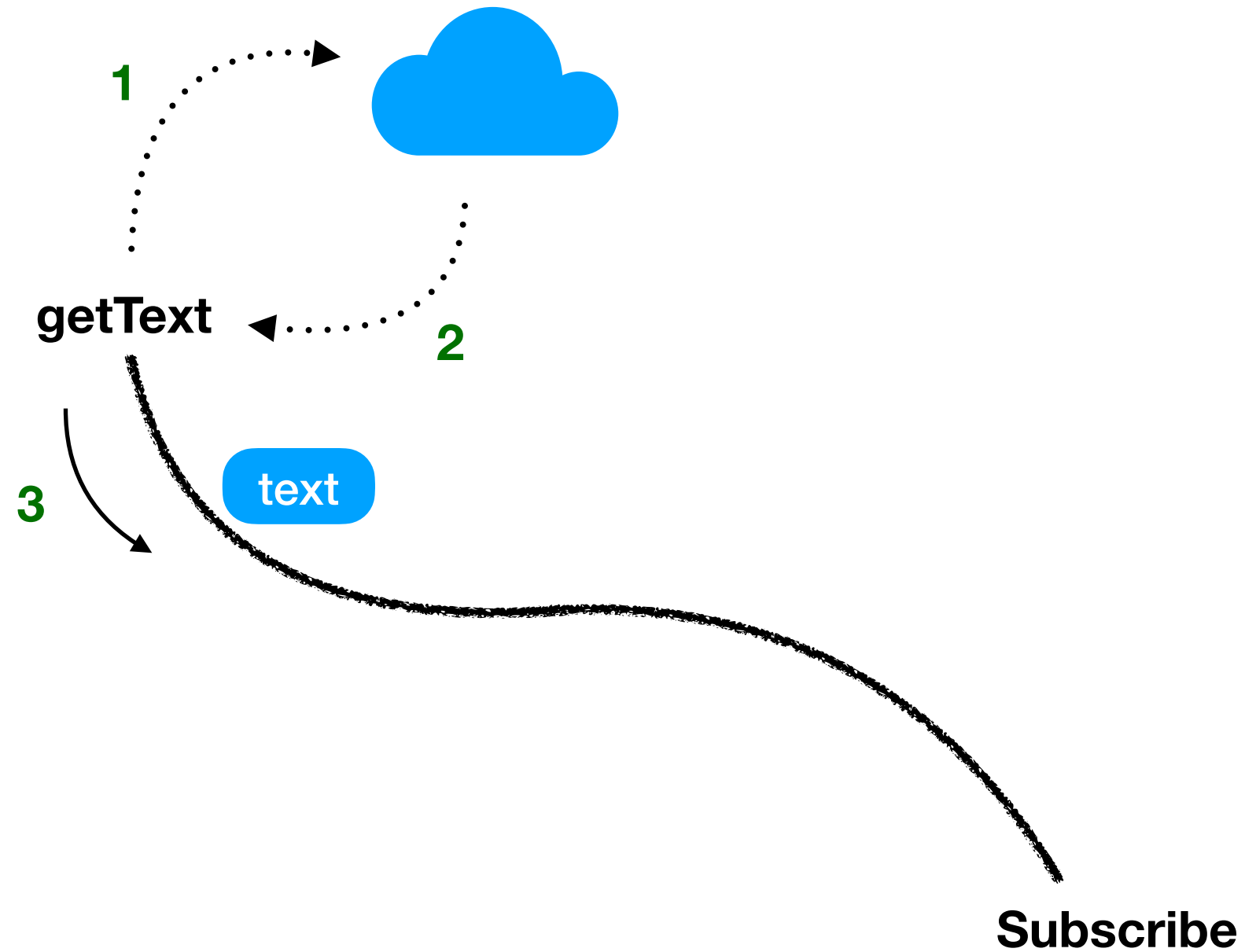
Stream

Subscriber

Consumer



Reactive Programming



Reactive Programming

Async 한 처리를 Functional 하게 처리하자

리턴값 Stream 은 Observable 을 반환하자

Stream 에 흐르는 Data/Event 를 Operator 로 처리하자

Stream 과 Stream 을 연결하자

Reactive Programming

Providing Concurrency by Scheduling Events

+

Asynchronous framework 가 해주는 일

Providing Operators by Calling Functions

Reduce Complexity using functional programming patterns, disciplines

Reactive Programming

Rx Operators

Observable<Data>

+

Operator

⋮

timer, defer, interval, repeat,
map, flat_map, filter, zip,
merge, delay, timeout

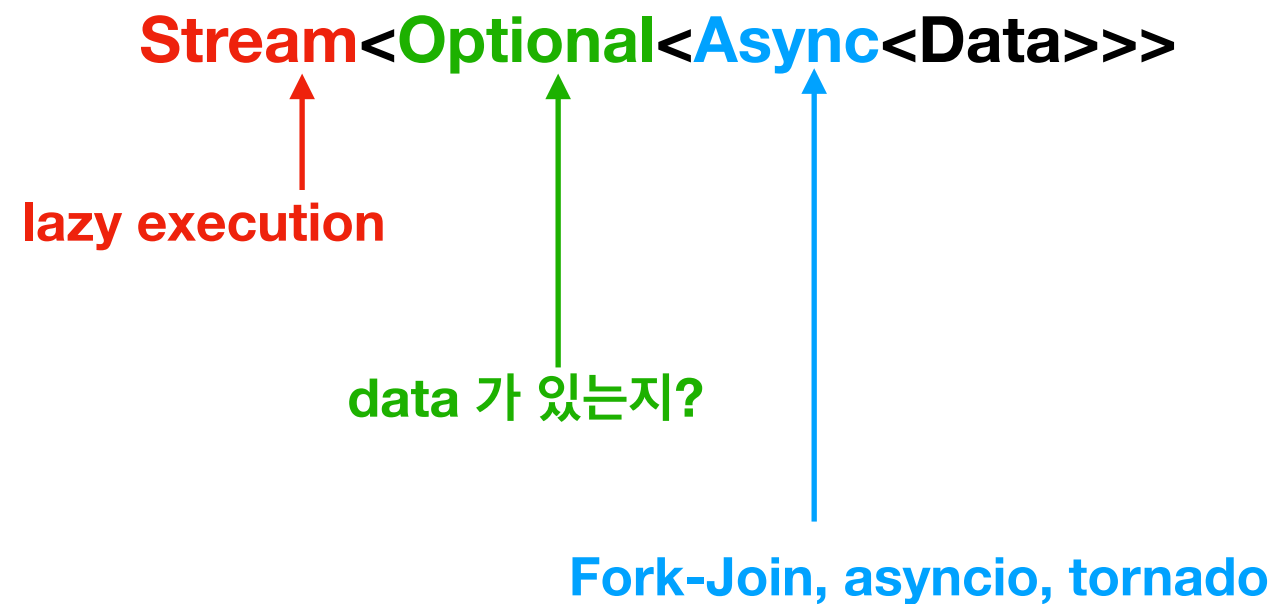
...

시간 관련 operator

데이터를 다루는 operator

Reactive Programming

Essence of Observable



Reactive Programming

Essence of Observable

Rx : **Stream**<**Optional**<**Async**<**Data**>>>
→ (Data → **Stream**<**Optional**<**Async**<**Data**>>>)
→ **Stream**<**Optional**<**Async**<**Data**>>>

Reactive Programming

Essence of Observable

Rx : **Stream**<**Optional**<**Async**<**Data**>>>
→ (Data → **Stream**<**Optional**<**Async**<**Data**>>>)
→ (Data → **Stream**<**Optional**<**Async**<**Data**>>>)
→ (Data → **Stream**<**Optional**<**Async**<**Data**>>>)
→ **Stream**<**Optional**<**Async**<**Data**>>>

Reactive Programming

Essence of Observable

Scheduler Rx : **Stream**<**Optional**<**Async**<**Data**>>>
→ (Data → **Stream**<**Optional**<**Async**<**Data**>>>)
→ (Data → **Stream**<**Optional**<**Async**<**Data**>>>) **User**
→ (Data → **Stream**<**Optional**<**Async**<**Data**>>>)
→ **Stream**<**Optional**<**Async**<**Data**>>>

Reactive Programming

Reduce Complexity using functional programming patterns, disciplines

1. Stream

```
Observable.from(get_obj)
            .filter(is_a)
            .map(...)
            .flat_map(...)
            .map(...)
            .subscription(success, error, complete)
```

Reactive Programming

Reduce Complexity using functional programming patterns, disciplines

2. Compositionality

```
Observable.from(get_obj)
    .filter(is_b)
    .map(...)
    .flat_map(...)
    .map(...)
    .subscription(success, error, complete)
```

← 이렇게 바꾸더라도 로직에 영향 X
딱만 맞추면 됨

Reactive Programming

Reduce Complexity using functional programming patterns, disciplines

3. Compositionality of Operators

`operator = some_other_operator : Observable<a> -> * -> Observable`

```
feature_a = Observable.from(get_obj)
                .filter(is_b)
                .map(...)
                .flat_map(...)
                .map(...)
                .some_other_operator(...)
```

```
feature_b = Observable.from(get_obj)
                .flat_map(feature_a)
                .map(...)
                .some_other_operator(...)
                .subscription(success, error, complete)
```

Reactive Programming

Reduce Complexity using functional programming patterns, disciplines

4. ROP, First Class Effect

```
Observable.from(get_obj)
    .filter(is_b)
    .map(...)
    .flat_map(...)
    .map(...)
    .subscription(success, error, complete)
```


Reactive Programming

Reduce Complexity using functional programming patterns, disciplines

5. Abstracted Data Type based on Type Theory - First Class Effect

```
Observable.from(get_obj)
            .filter(is_b)
            .map(...)
            .flat_map(...)
            .map(...)
            .subscription(success, error, complete)
```

Reactive Programming

Reduce Complexity using functional programming patterns, disciplines

5. Abstracted Data Type based on Type Theory - First Class Effect

```
Observable.from(get_obj)
            .filter(is_b)
            .map(...)
            .flat_map(...)
            .map(...)
            .subscription(success, error, complete)
```

```
from : a -> Observable<a>
filter : Observable<a> -> (a -> bool) -> Observable<a>
map : Observable<a> -> (a -> b) -> Observable<b>
flat_map : Observable<a> -> (a -> Observable<b>) -> Observable<b>
```

```
operators : Observable<a> -> * -> Observable<b> => flat_map
```

Reactive Programming

Reduce Complexity using functional programming patterns, disciplines

5. Abstracted Data Type based on Type Theory - First Class Effect

```
Observable.from(get_obj)
            .flat_map(filter_is_a)
            .flat_map(...)
            .flat_map(...)
            .flat_map(...)
            .subscription(success, error, complete)
```

```
from : a -> Observable<a>
flat_map : Observable<a> -> (a -> Observable<b>) -> Observable<b>
```

```
Rx : Stream<Optional<Async<Data>>>
    -> (Data -> Stream<Optional<Async<Data>>> )
    -> (Data -> Stream<Optional<Async<Data>>> )
    -> (Data -> Stream<Optional<Async<Data>>> )
    -> Stream<Optional<Async<Data>>>
```

Reactive Programming

Reduce Complexity using functional programming patterns, disciplines

5. Abstracted Data Type based on Type Theory - First Class Effect

```
Stream.from(Optional.from(Async.from(get_obj)))  
  .flat_map(Optional.flat_map(Async.flat_map(filter_is_a)))  
  .flat_map(Optional.flat_map(Async.flat_map(...)))  
  .flat_map(Optional.flat_map(Async.flat_map(...)))  
  .flat_map(Optional.flat_map(Async.flat_map(...)))
```

```
from : a -> Stream<Optional<Async<a>>>  
flat_map : Stream<Optional<Async<a>>>  
          -> (a -> Stream<Optional<Async<b>>>  
          -> Stream<Optional<Async<b>>>
```

```
Rx : Stream<Optional<Async<Data>>>  
    -> (Data -> Stream<Optional<Async<Data>>> )  
    -> (Data -> Stream<Optional<Async<Data>>> )  
    -> (Data -> Stream<Optional<Async<Data>>> )  
    -> Stream<Optional<Async<Data>>>
```

Reactive Programming

Reduce Complexity using functional programming patterns, disciplines

5. Abstracted Data Type based on Type Theory - First Class Effect

```
Observable.from(get_obj)
            .flat_map(filter_is_a)
            .flat_map(...)
            .flat_map(...)
            .flat_map(...)
            .subscription(success, error, complete)
```

이런식으로 쪽 연결된 형태가 Rx 의 본질

Asynchronous Programming 을 한단계 래핑을 하여
이런식으로 프로그램을 짜서 돌릴 수 있도록 하겠다하는 규칙

SUMMARY

Code Scalability

Reactive Programming (Rx)

much more compositional

Asynchronous Frameworks (tornado, asyncio)

```
data = yield tornado.iostream.read_until('\r\n')
```

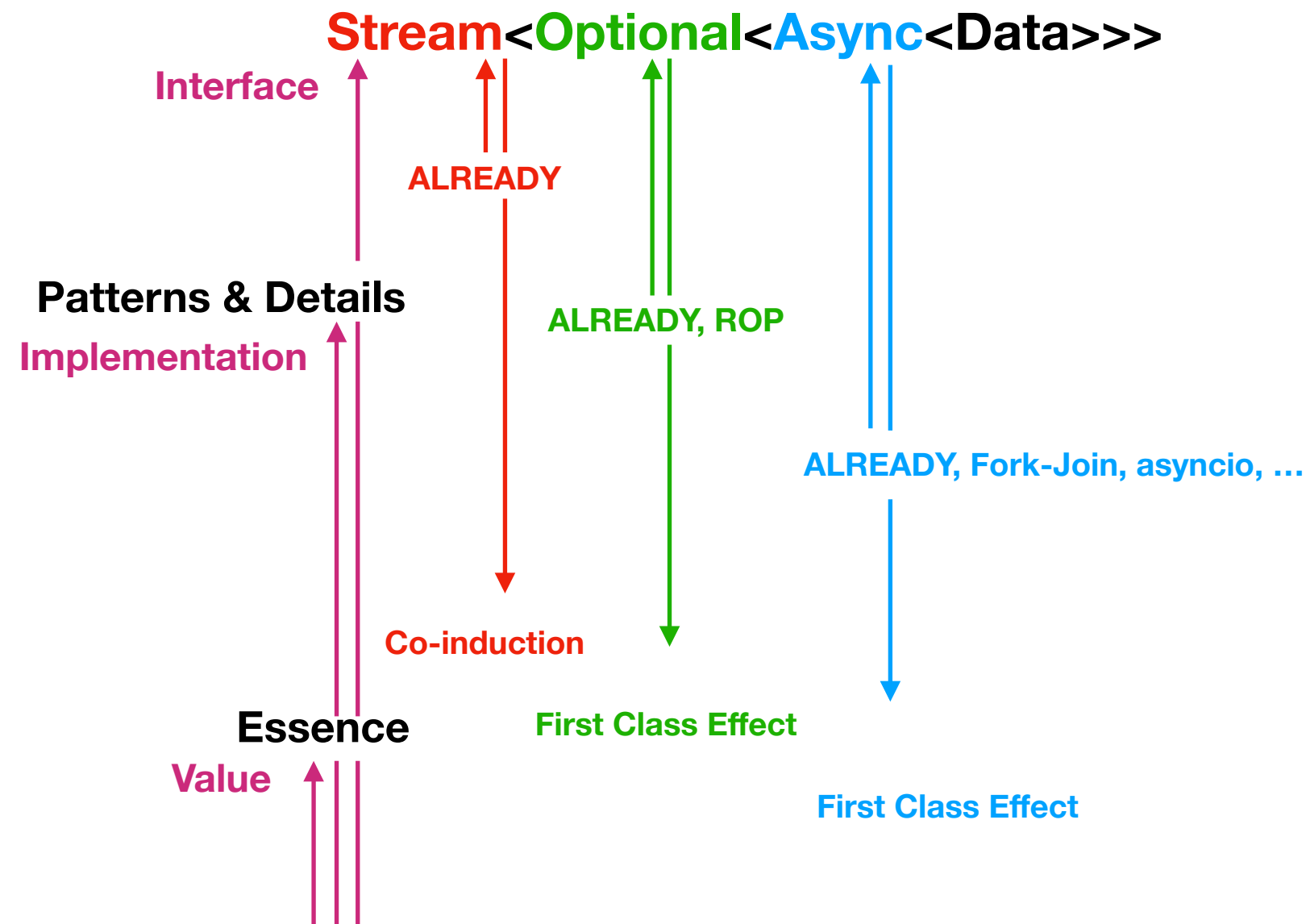
much more compositional

Non-Blocking I/O

```
while not_finished:  
    try:  
        data = socket.recv(buf_size)  
        do_something(data)  
    except socket.error as e:  
        if e.args[0] in _ERRNO_WOULDBLOCK:  
            # Do something else
```

Reactive Programming

Essence of Observable



what we can obtain by following the principles can be explained based on mathematics.

Reference

Salt Stack 과 RxPY 로 살펴보는 파이썬 비동기 프로그래밍

Functional Reactive Programming 패러다임