

いまさら聞けない RxSwift入門

株式会社 tech vein

猪俣 充央

自己紹介

株式会社 tech vein 代表

猪俣 充央 (いのまた みつひろ)

会社HP: <https://www.tech-vein.com/>

private twitter: @ino2222

公開中のAndroid/iOSアプリ

ソウルアンリーシュ

自分のオリキャラを作るスマホRPG

<http://soun.tech-vein.com/>



©2014 QUALIASOFT Inc. All Rights Reserved



ChiiQ

地域で助け合える匿名相談アプリ

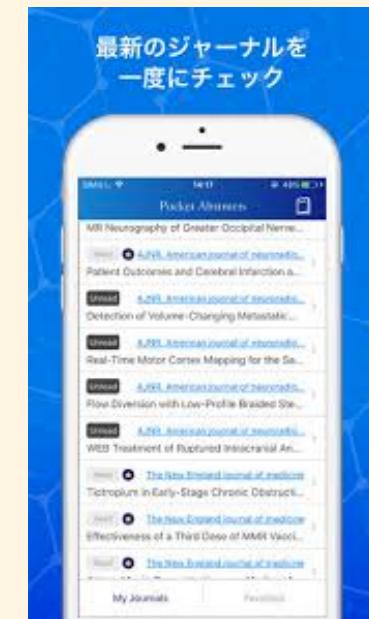
<https://bit.ly/2HZ5auR>



ポケットアブストラクト

医療従事者向けの最新ジャーナル閲覧アプリ

<https://apple.co/2K6zhxc>



今日お話すこと

- RxSwiftとは
- RxSwiftの基本
- これだけ覚えたら使えるようになるかも？

今日お話ししないこと(知ってる人向け)

- Cold -> Hot 変換
- Reactive Streams(backPressure など)

RxSwiftとは

- 各言語で実装されている「Rx(Reactive Extension)」のSwift向けライブラリ
- Reactive Extension ... もともとは C#.NET で使われだした Rx.NET がオリジナル
- Java(RxJava), JavaScript(RxJS), C++(RxCpp), Ruby(Rx.rb), PHP(RxPHP) などなど多数

ReactiveXとは何か？

ReactiveX = Reactive Extension

Reactive Programming = 反応的プログラミングをするための

Extension = 拡張

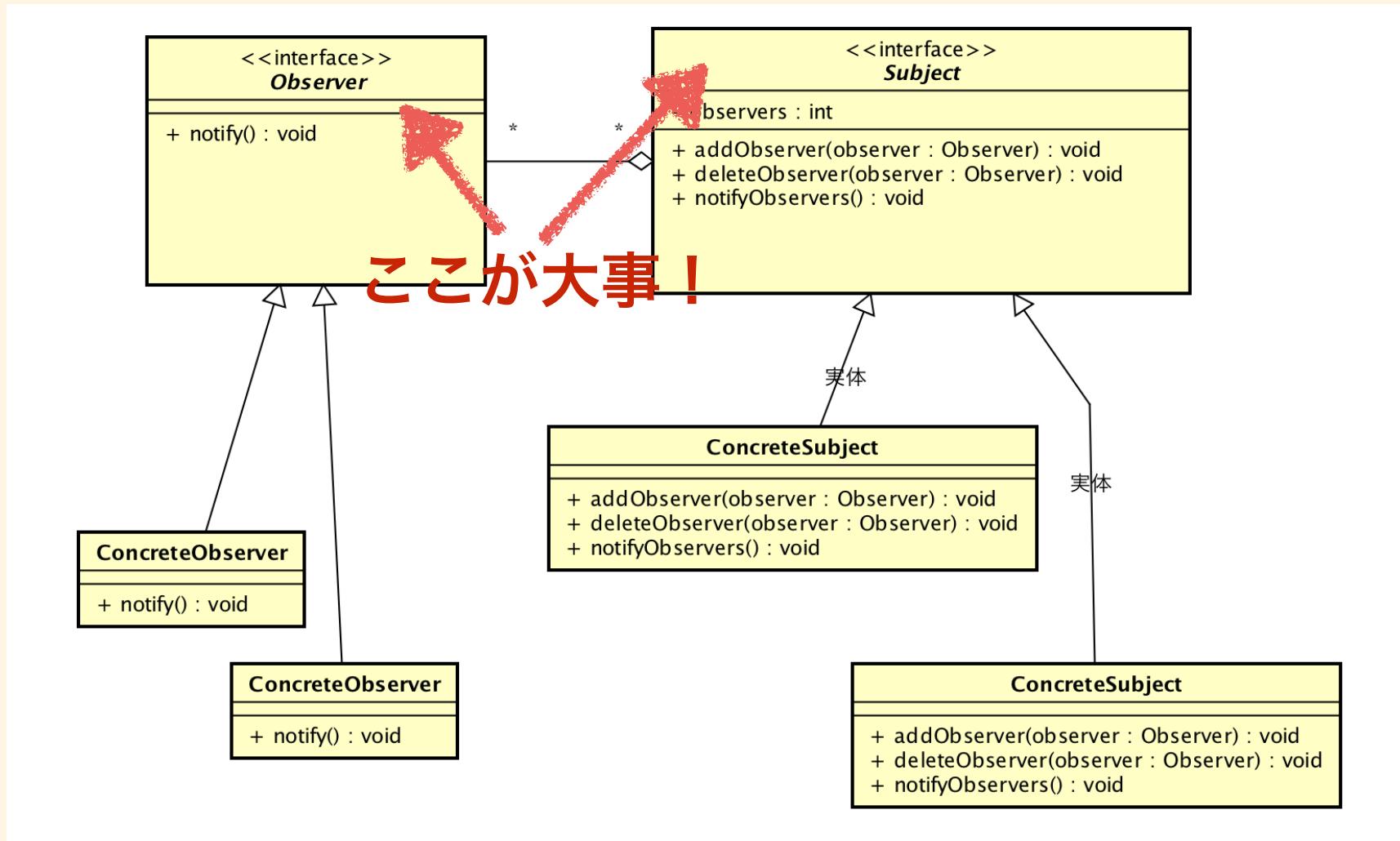
ReactiveXは

- デザインパターンの **Observable** パターン
- デザインパターンの **Iterator** パターン
- 関数型プログラミング

を組み合わせて作られたライブラリのこと

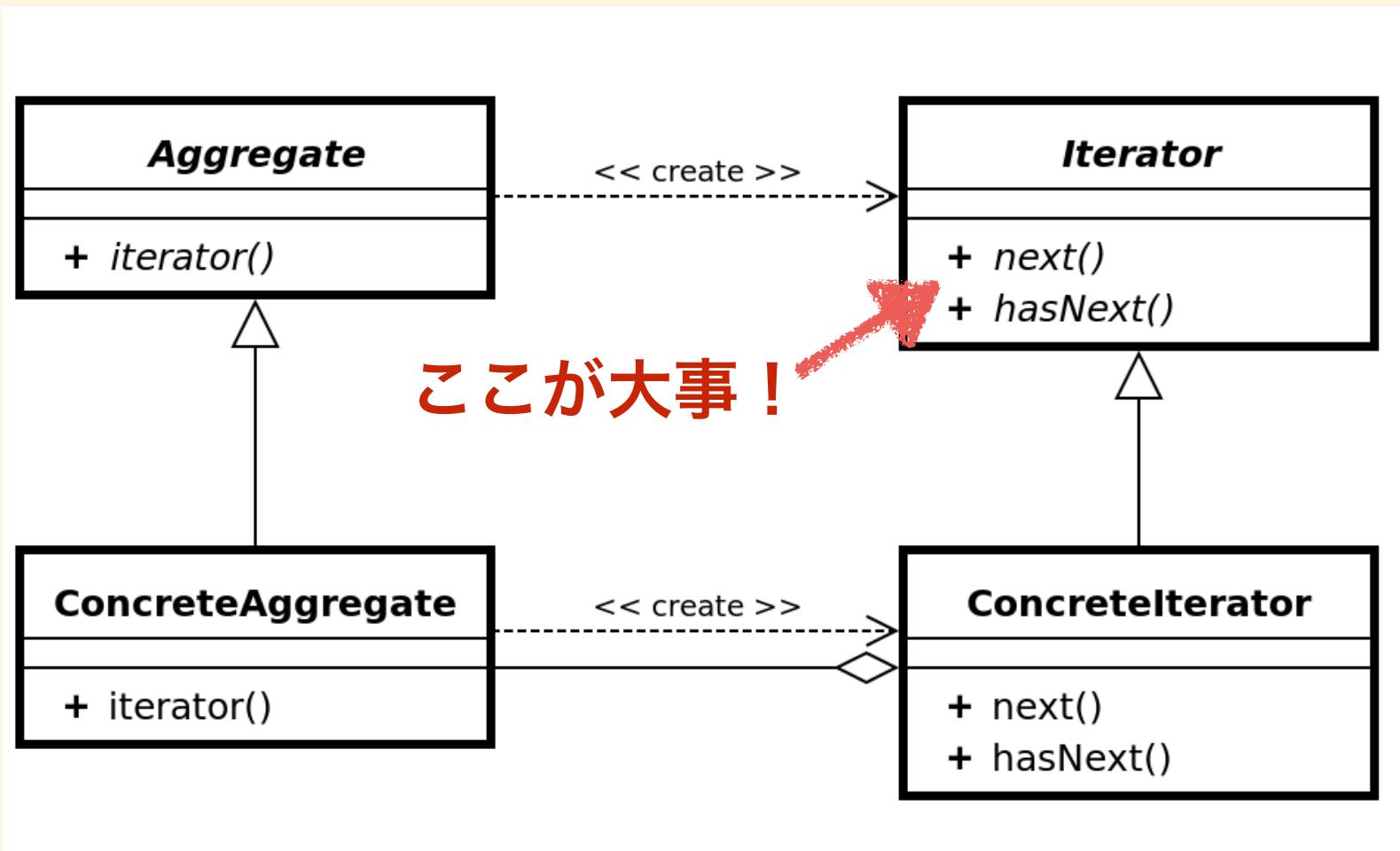
Observerパターン

データを監視する人と伝える仕組みを
抽象化したデザインパターン

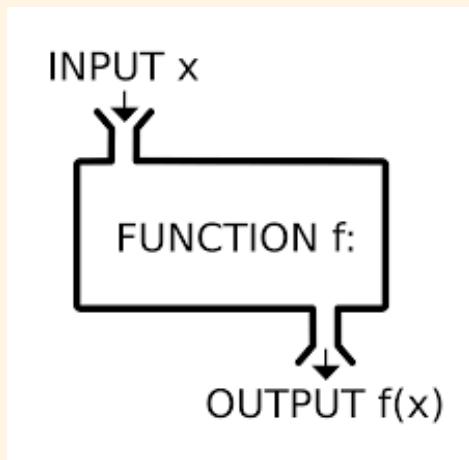


Iterator パターン

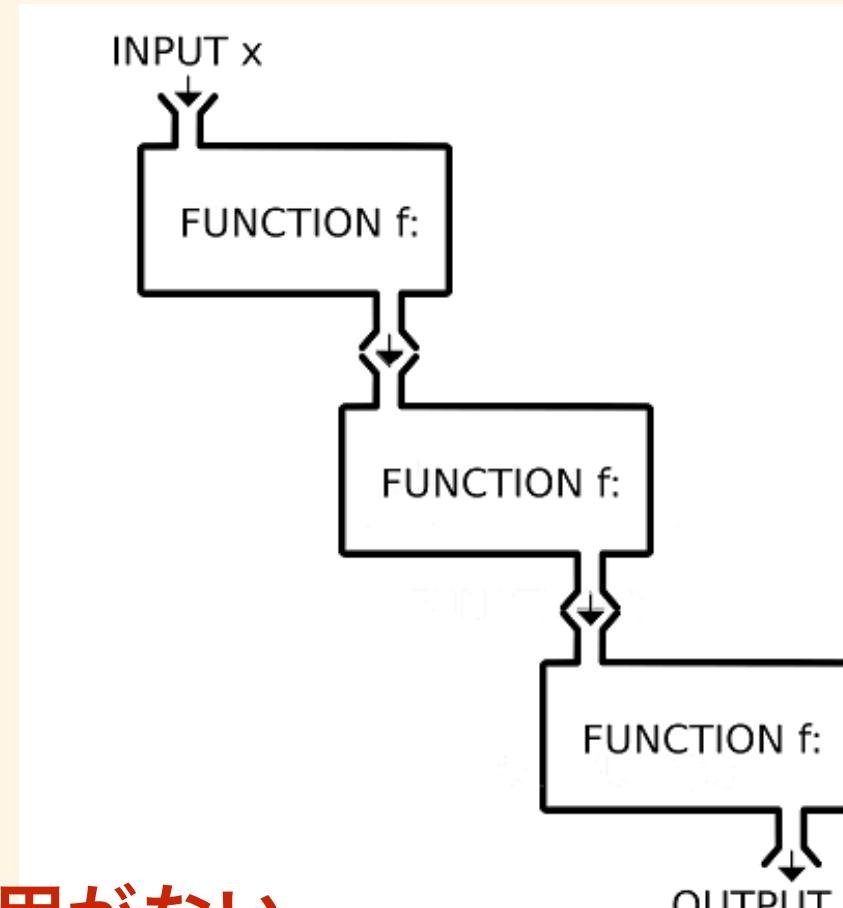
一連のデータを順番に処理するデザインパターン



関数型プログラミング



→

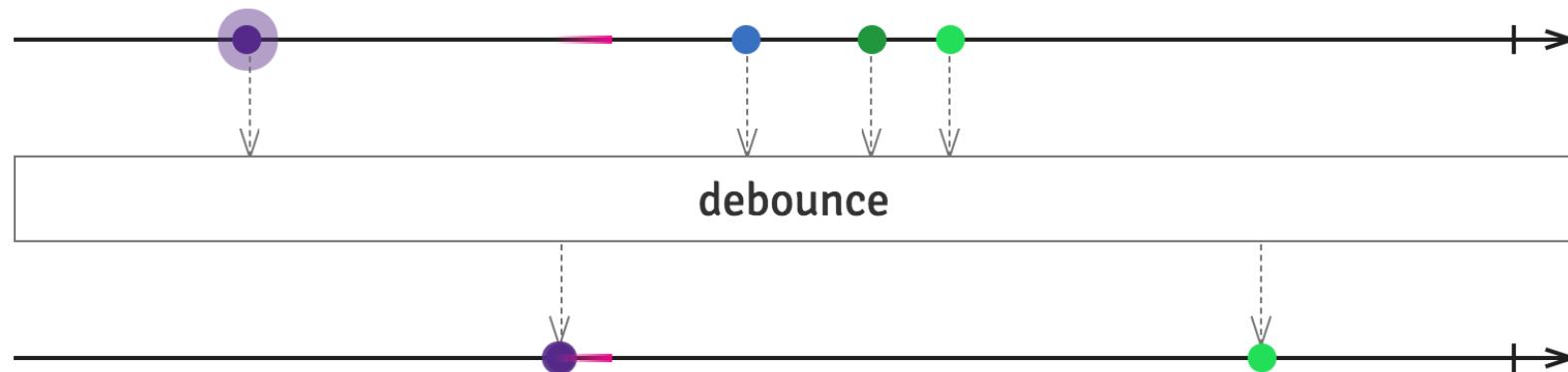


それぞれの関数は
IN/OUTにしか
影響しない =副作用がない

そしてRxへ...

The Observer pattern done right

ReactiveX is a combination of the best ideas from
the **Observer** pattern, the **Iterator** pattern, and **functional programming**



ここが嬉しいRxSwift

時系列データを含む、**あらゆるデータをストリームとして**スマートに処理できる。
ストリームとして順番に処理できることで、ややこしい非同期制御がシンプルになる。

→コールバック地獄からの解放

抽象化により送り手・受け取り手を直接知らないまま処理が出来るので**疎結合**になる。

→改変の影響範囲が小さくなる

→部品化・共通化が進む

各種ストリームを扱う便利な道具が揃っている。

→ストリームの生成・変換・合成を行うオペレータ群

→ストリームのスレッド切り替え機能

RxSwiftの基本

RxSwiftの基本

これだけ知ってたら
Rxチョット デキルって言える

RxSwiftの基本

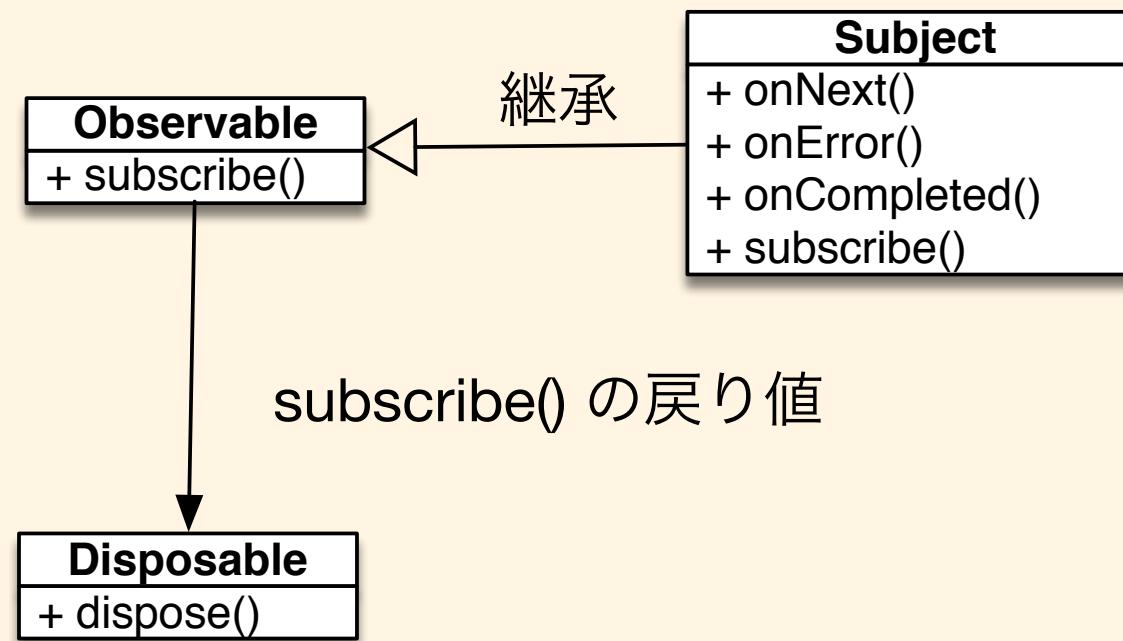
これだけ知ってたら
Rxチョット(ダケ)デキルって言えるかも？

Observable, Subject, Disposable

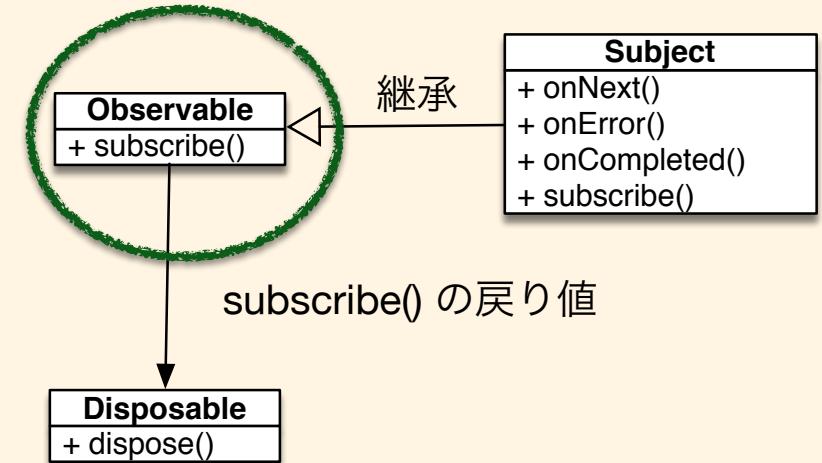
キモになるのは、
この3系統のクラス(プロトコル)。

クラスは色々ありますが、
だいたいこれらの仲間です。

Observable, Subject, Disposable



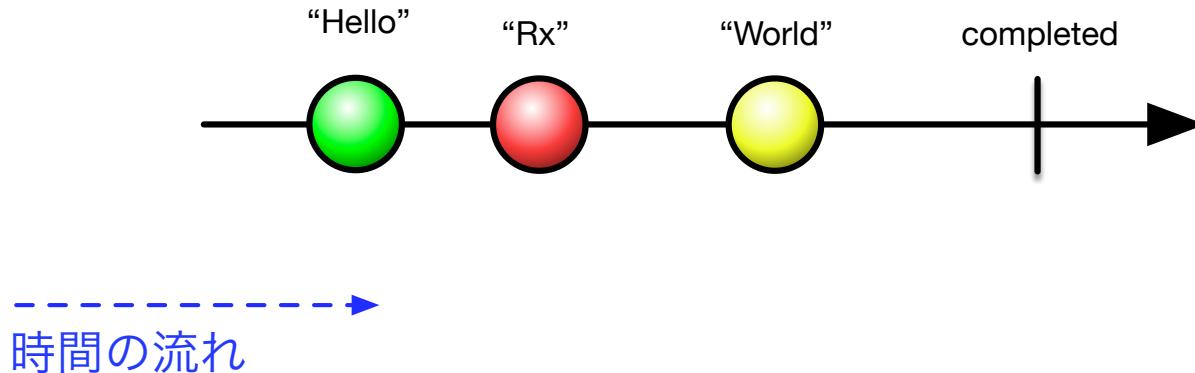
Observable<T>



直訳すると、監視することができる対象。
任意の型 T について、監視してデータ（イベント）を受け取
る事ができるクラス・インスタンスです。

時系列でデータが Observable に流れることから、
Observable は **ストリーム** (Stream) とも呼びます。

Observable<String>

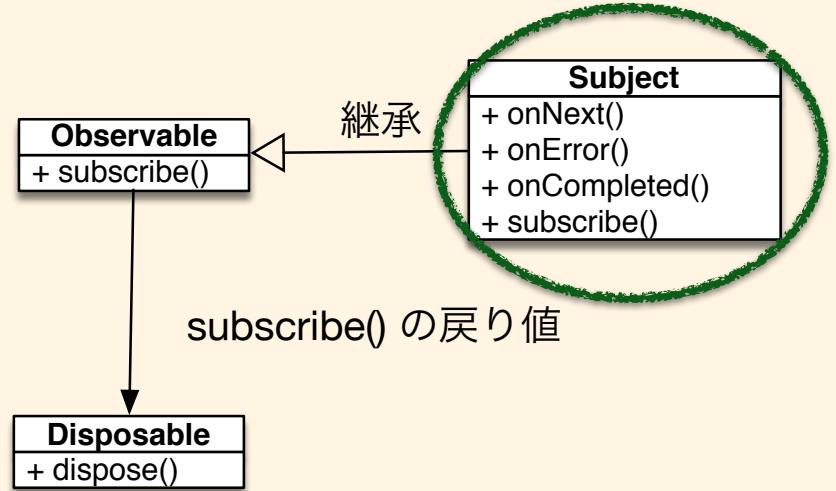


```
func sampleObservable(observable: Observable<String>) {  
    _ = observable.subscribe(  
        onNext: { print("next(\($0))") },  
        onCompleted: { print("completed") }  
    )  
}
```

実行結果：下の順に出力される。

next(Hello), next(Rx), next(World), completed

Subject



簡単に言うと、任意のデータを流す機能(Observerとしての機能)を持ったObservable実装。

RxSwift のSubjectは

PublishSubject, **BehaviorSubject**, **ReplaySubject**の3種類。

SubjectはObserverを継承しているので
Observableとして扱える。

```
let subject = PublishSubject<String>()
let observable: Observable<String> = subject
```

SubjectはonNext(), onComplete(), onError()
などのデータ・イベント発信メソッドを持つ。

```
let subject = PublishSubject<String>()
subject.onNext("Hello")
subject.onNext("Rx")
subject.onNext("World")
subject.onCompleted()
```

ストリームのイベント

- **next**

データが正しく流れてきたというイベント。任意の型のデータを引数を持つ。

- **error**

ストリーム中がエラー(例外)で異常停止した時に流れるイベント。発生したエラーを引数を持つ。

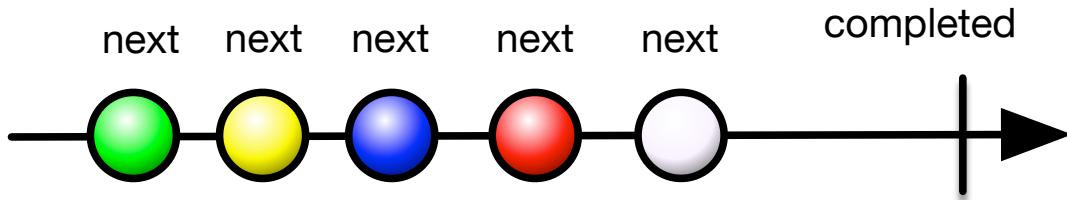
- **completed**

ストリームが正しく完了した時に発生するイベント。ストリームの最後に一度だけ発生する。

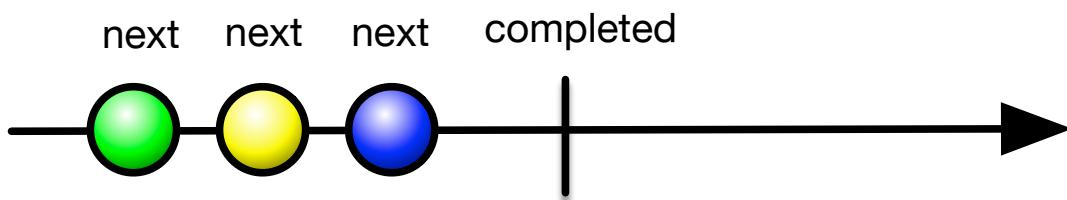
(入力を待ち続けたいときはcompletedを流さない)

ストリームのイベント

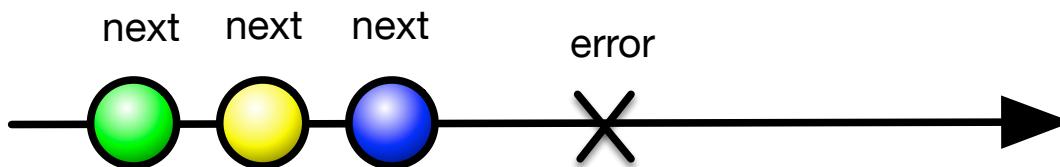
ストリームは next でデータを流し、completed で終了する



completed が来たら、その先は next/ error は流せない



error (例外)が来たら、その先は next / completed / error は流せない



RxSwift の Subject使い分け

- **PublishSubject**

過去のデータを保持しないSubject。

利用例：ボタンのタップイベント

- **BehaviorSubject**

過去のデータのうち直近の1つを保持しているSubject。

利用例：スイッチON/OFF状態・ユーザステータスなど

- **ReplaySubject.create(bufferSize: Int)**

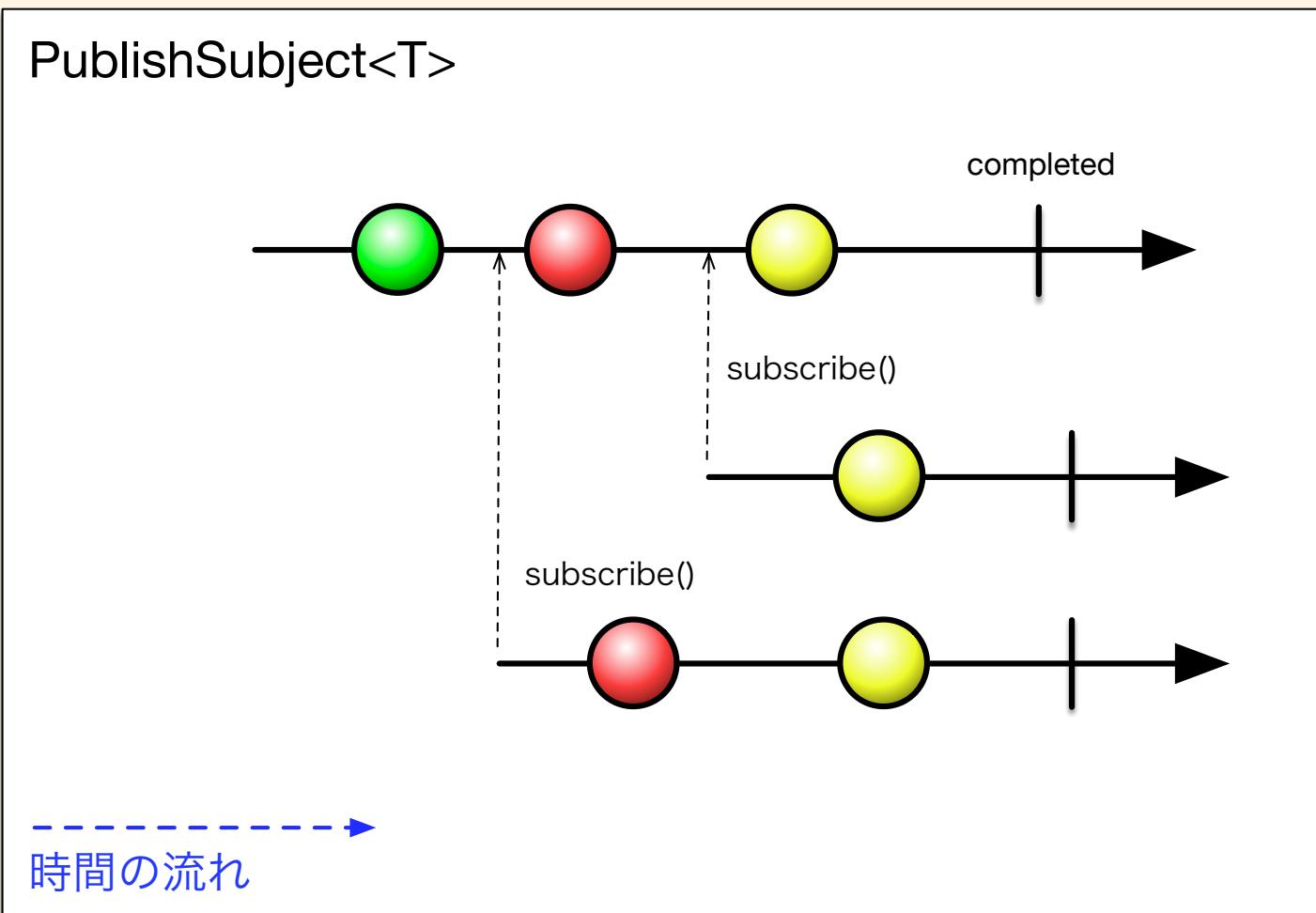
過去の直近データを指定バッファに入るだけ保持しているSubject。

バッファサイズ0ならPublishSubjectと同じ。

バッファサイズ1ならBehaviorSubjectと同じ。

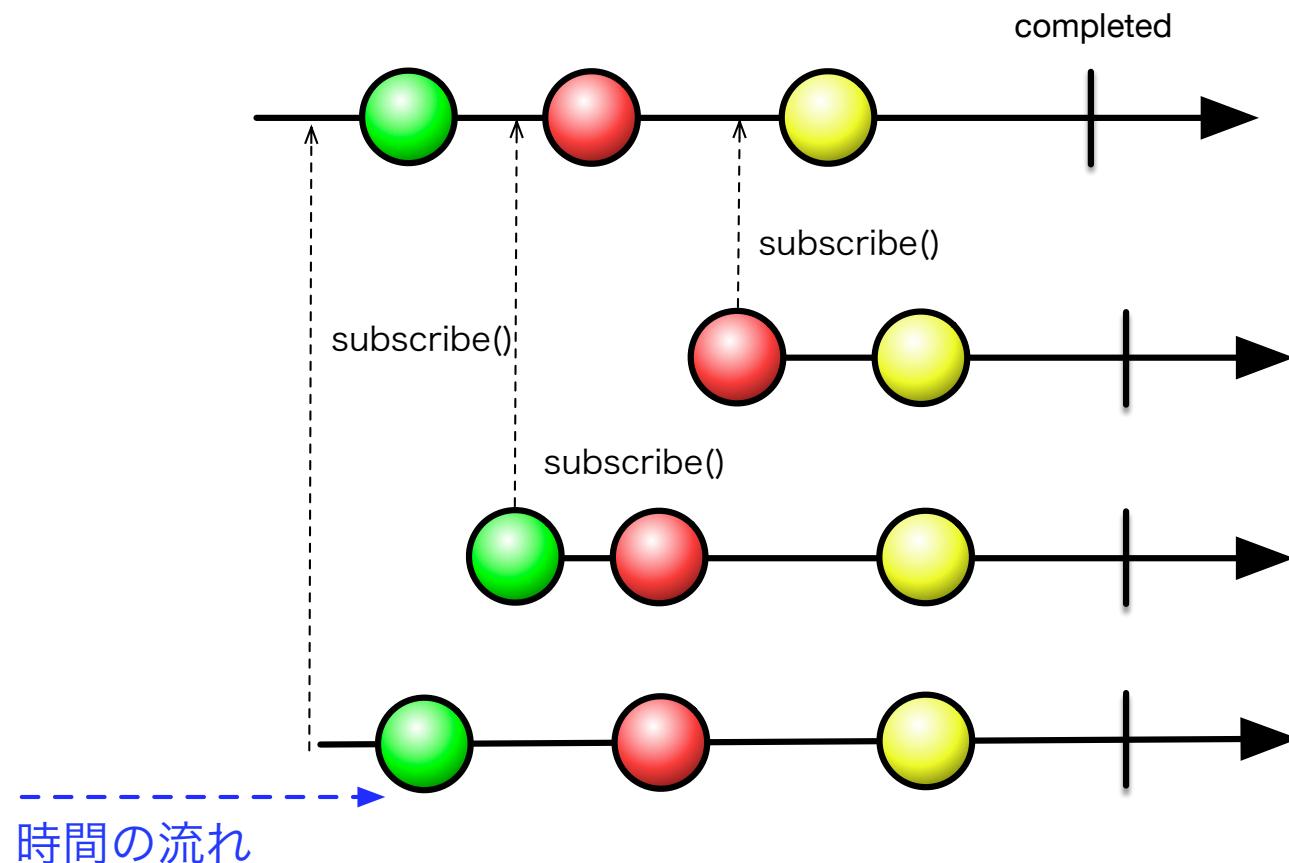
利用例：思いつかない...履歴系で使えるかも？

PublishSubject



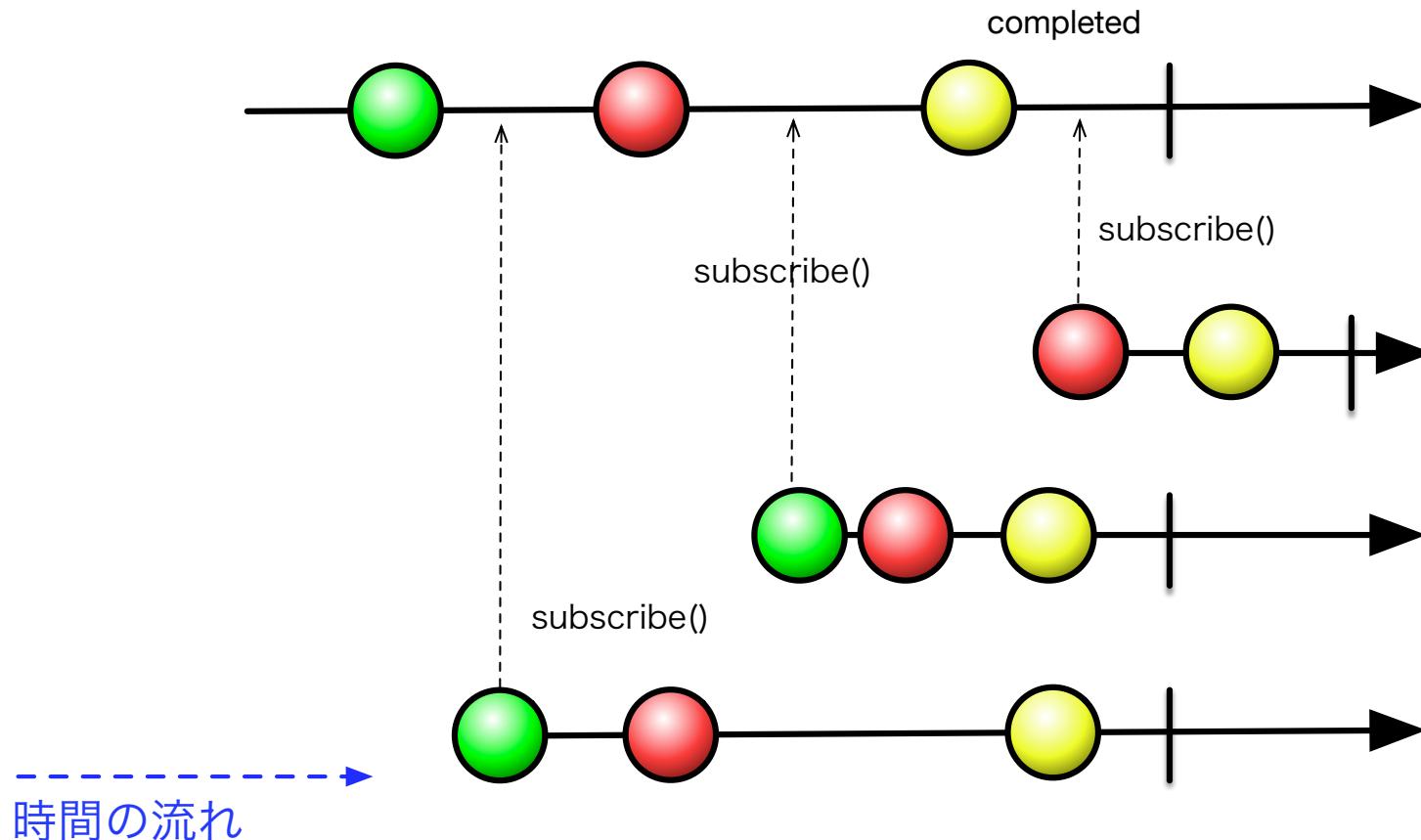
BehaviorSubject

BehaviorSubject<T>

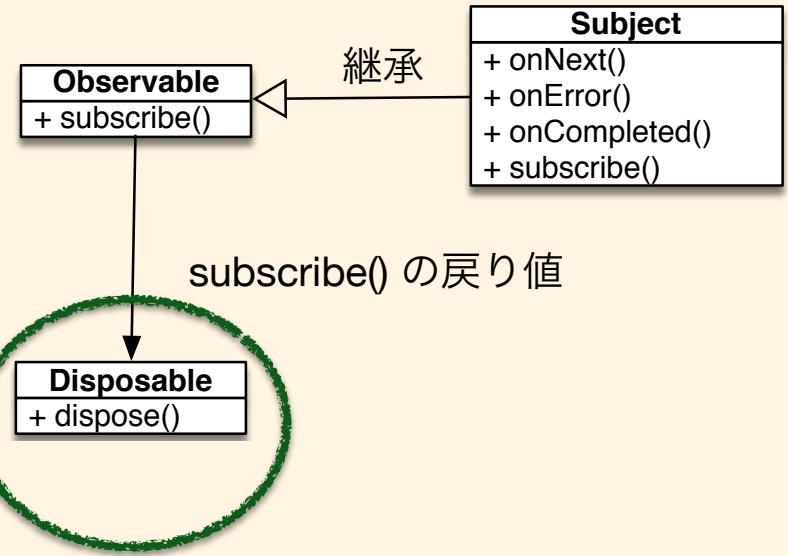


ReplaySubject

ReplaySubject<T>.create(bufferSize: **2**)



Disposable



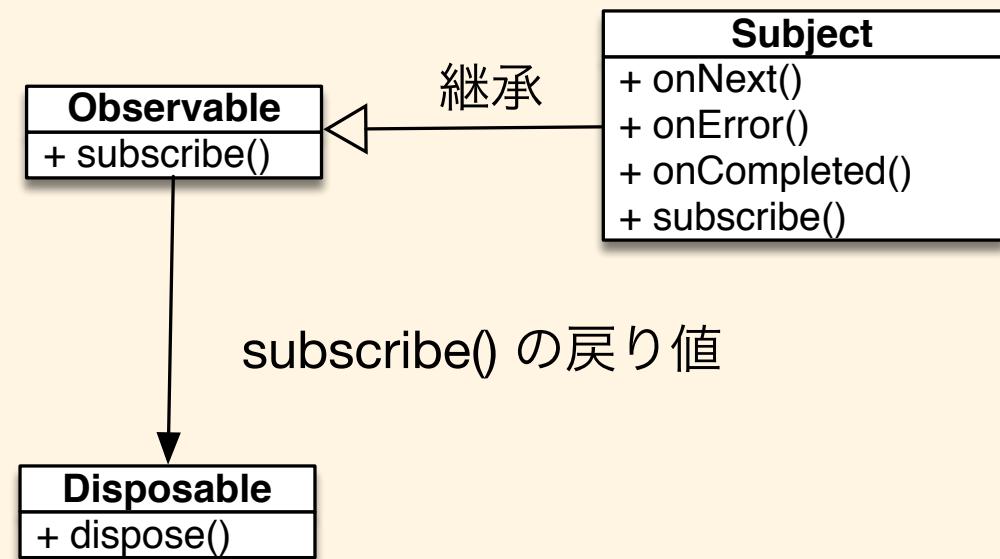
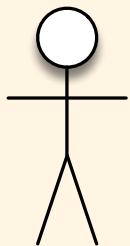
直訳すると、廃棄・処分できるもの。

Observableを subscribe(購読) すると戻り値として Disposable を返します。

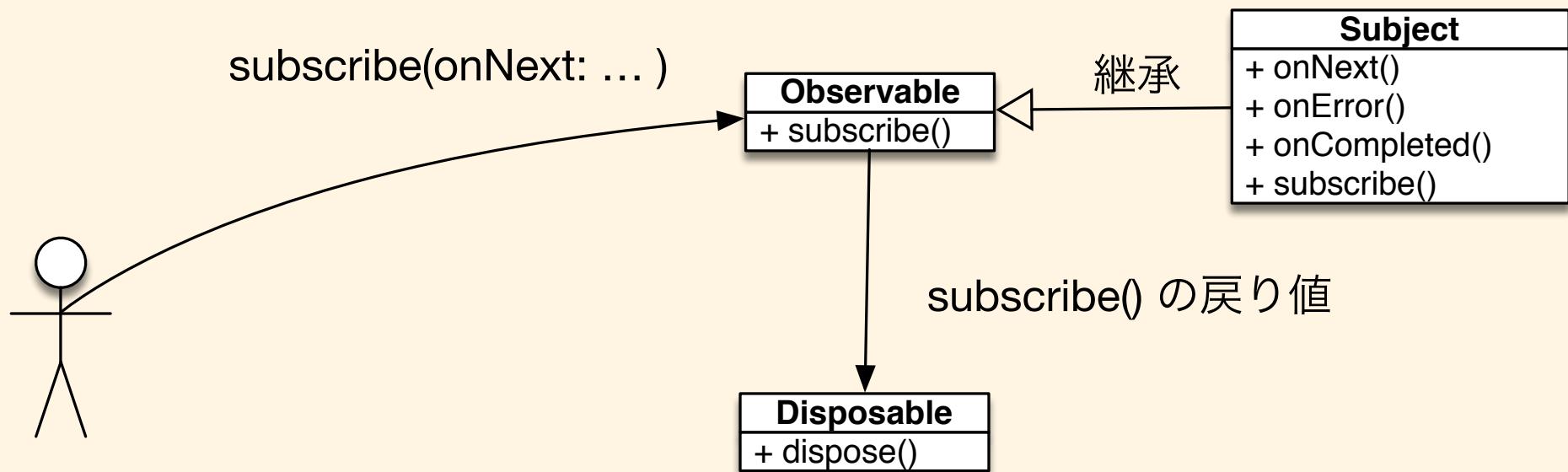
Disposableに対して dispose()を呼ぶことで、非同期処理のキャンセルなど、監視する側として処理を止める事ができます。

例：画面を閉じたら通信キャンセルしたり、入力待ちを止めたりする。

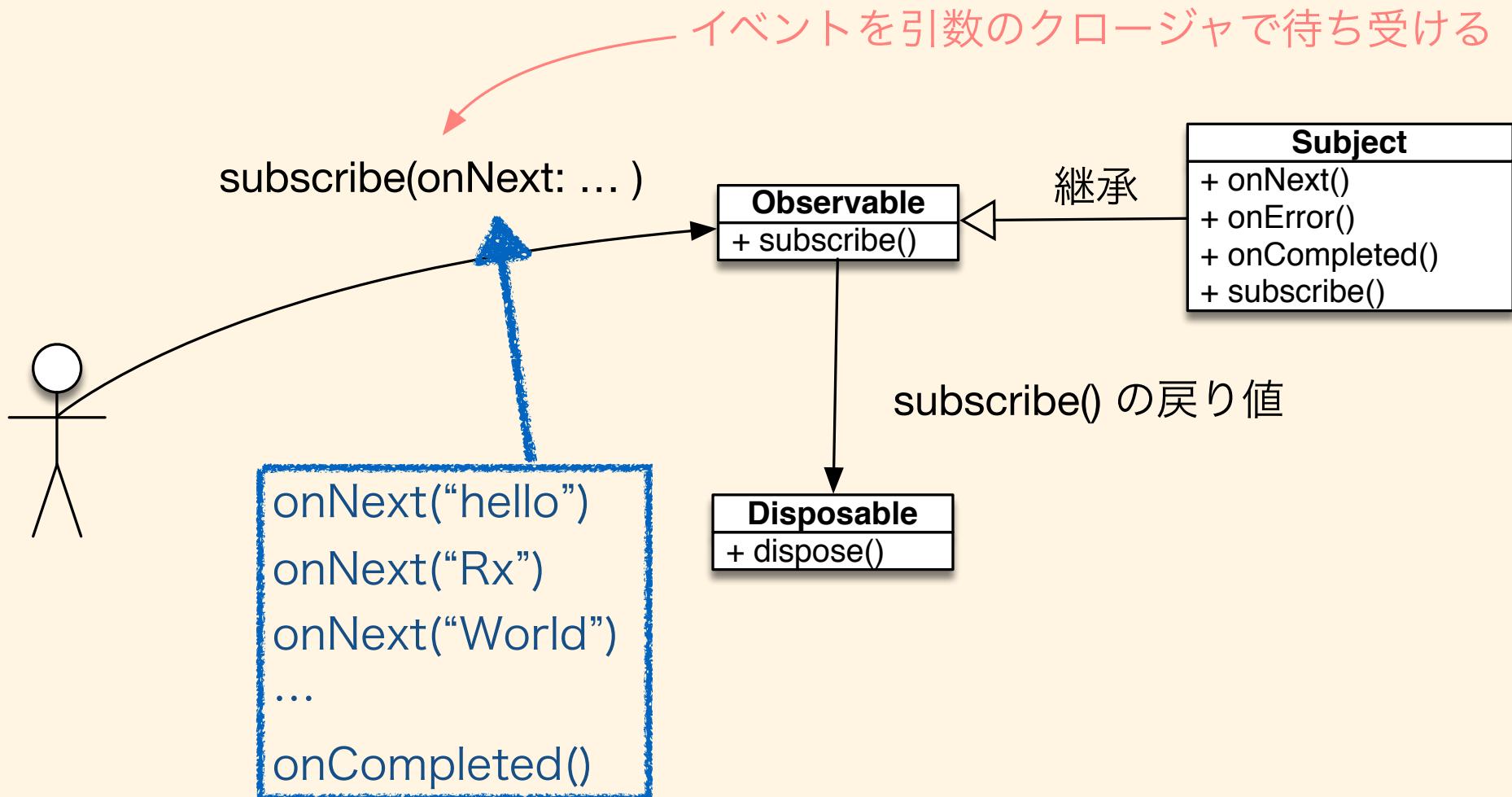
処理の流れ



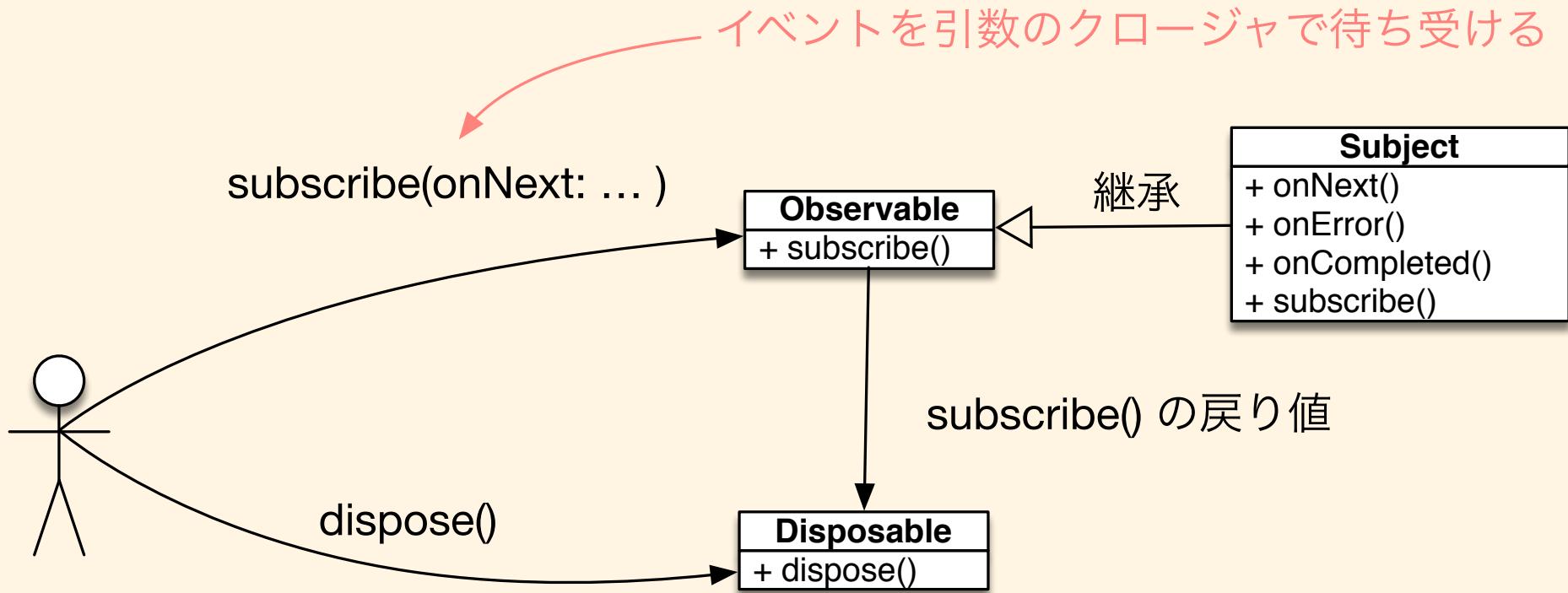
処理の流れ



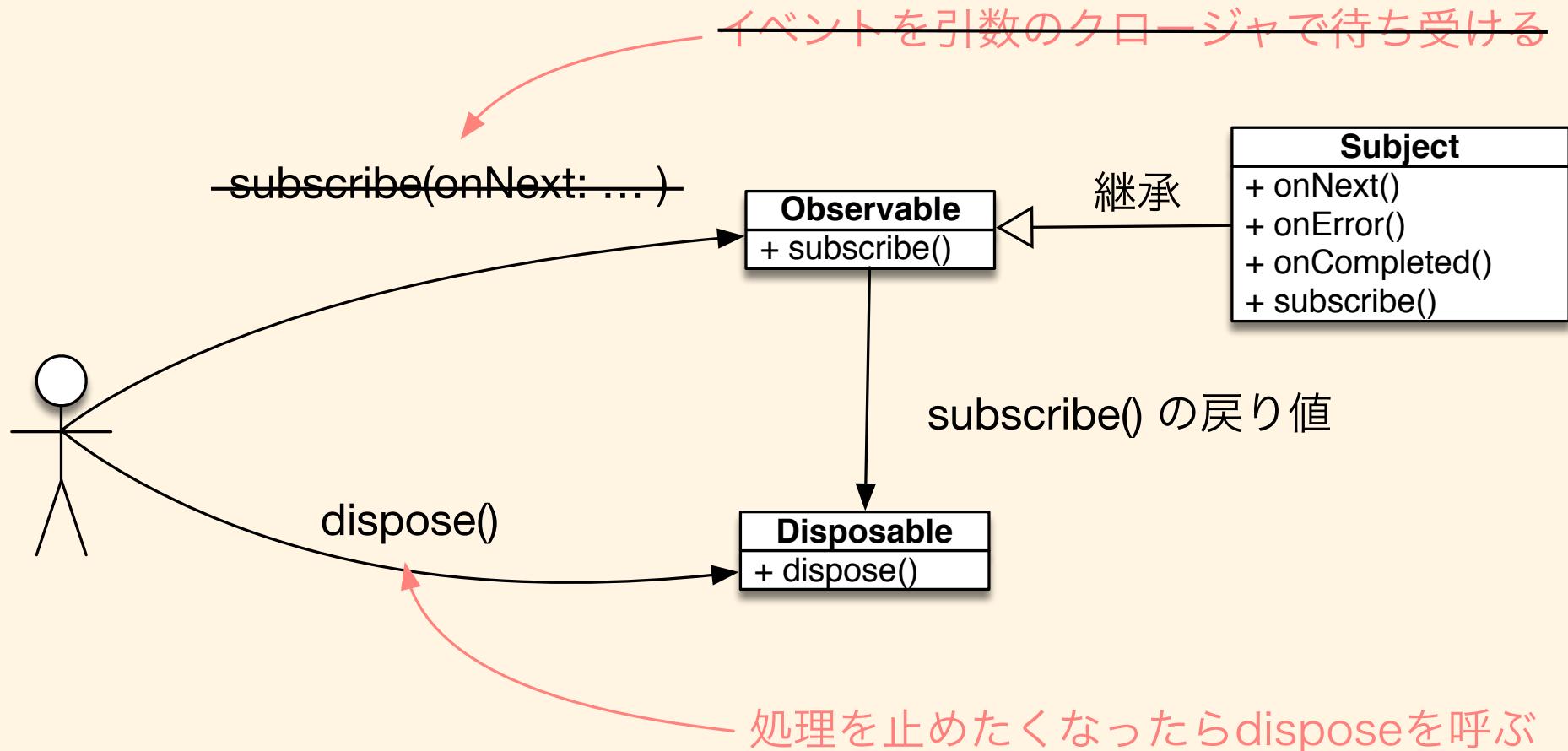
処理の流れ



処理の流れ

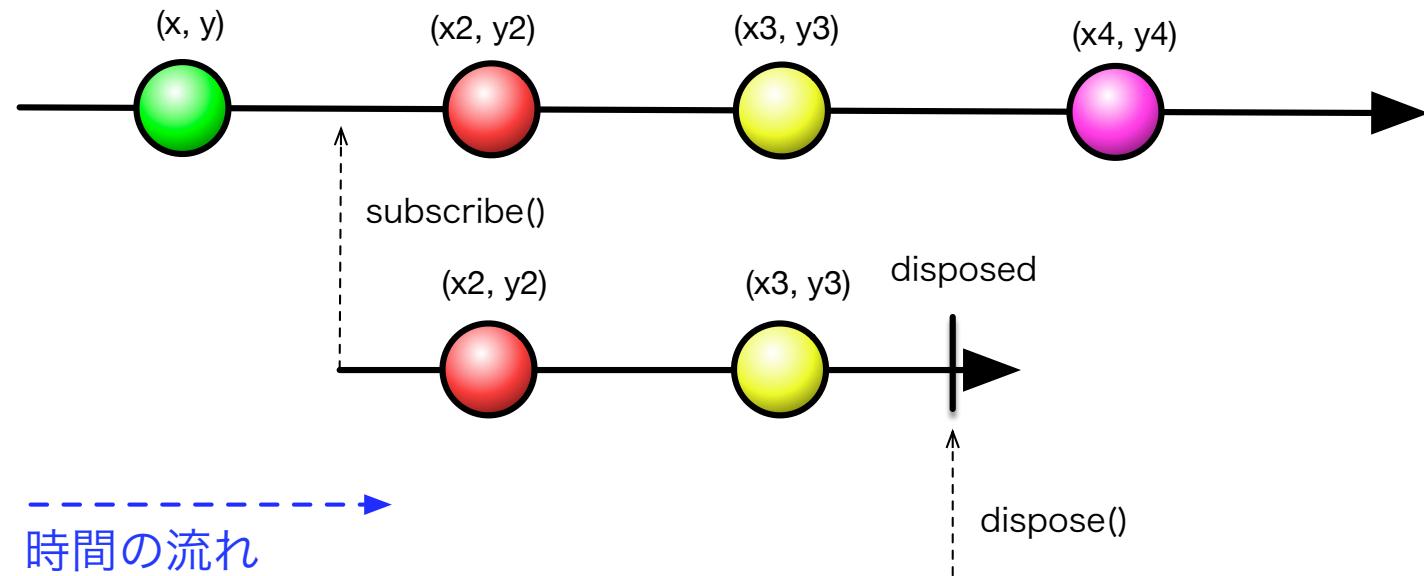


処理の流れ



Disposable.dispose()

例：タッチイベントの監視(Observable<Point>)



DisposeBag

disposeをいちいちするのはめんどくさい。

そんなときはDisposeBag。

DisposeBag

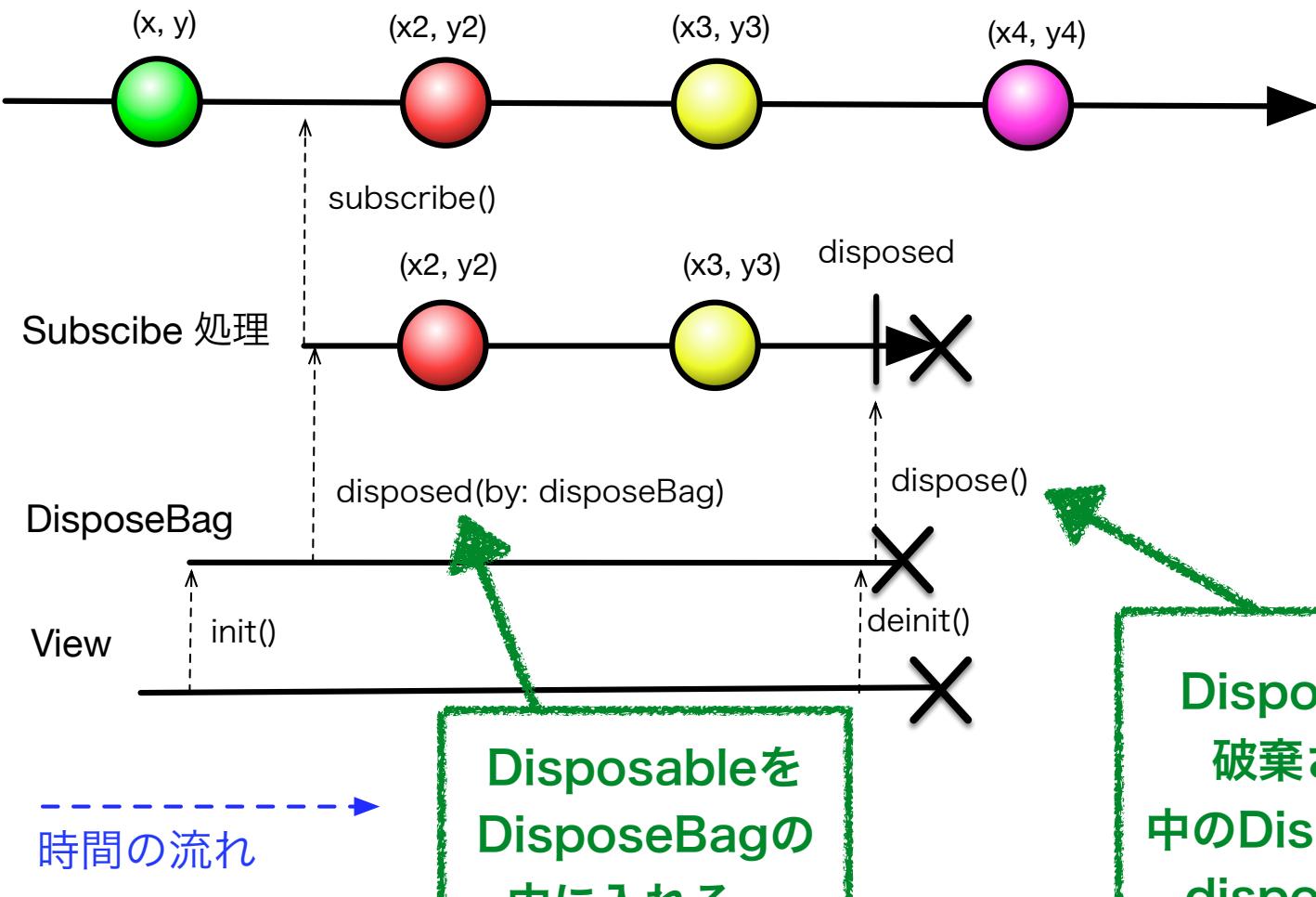
画面など、監視元インスタンスが死んだら自動でぜんぶdisposeしてほしい時に使える便利クラス。

DisposeBagインスタンスに突っ込んでおくと、
disposeBagが解放される時にまとめてdisposeBagの中のdisposableをdisposeしてくれる。

→Rx標準ではないですがめちゃくちゃ使います。

とりあえずDisposeBag作って放り込んでおけばいいと思う。

例：タッチイベントの監視2(Observable<Point>)



DisposeBagの利用例

```
class UISampleViewController: UIViewController {  
    private let disposeBag = DisposeBag()
```

```
    override func viewDidLoad() {  
        super.viewDidLoad()  
        resetAllInput()  
        setupResetButton()  
        setupInputEvents()  
        setupImageViewEvents()  
    }
```

←UIViewController の
フィールドにdisposeBagを置いておく
→画面が閉じたらUIViewController が死ぬ。
→UIViewControllerが死んだらDisposeBagも死ぬ。
→DisposeBagが死んだら紐づくdisposableが
dispose()される。

```
/// 入力イベント欄のイベント設定  
private func setupInputEvents() {
```

```
    // *inputTextField.text -> previewLabel.text を直接つなぐ。  
    inputTextField.rx.text // 入力イベント  
        .bind(to: previewLabel.rx.text) // 入力イベントをそのまま previewLabel.text に適用  
        .disposed(by: disposeBag) // DisposeBag に後始末を任せる
```

<https://github.com/ecoopnet/rxswift-beginner/blob/master/RxSwiftSample/RxSwiftSample/UISampleViewController.swift>

オペレータ(Operator)

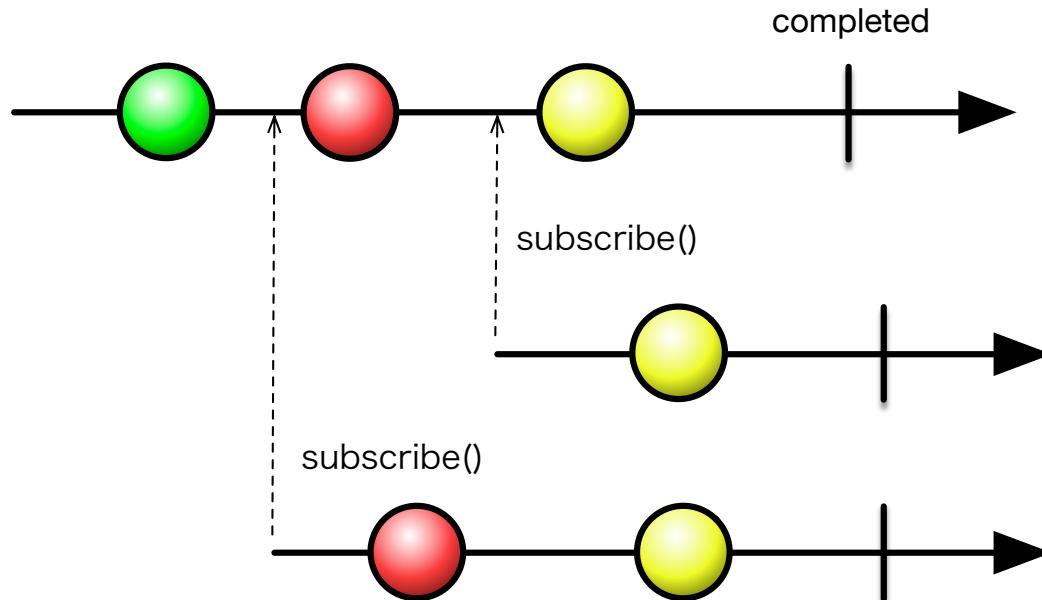
Rxには Observable, Subjectで作ったストリームを操作する機能が提供されています。

ストリーム(=Observable)を変換・合成・生成する操作のことを「オペレータ」と呼びます。

ストリームを作る

- 1 ... Subject クラスから作る
- 2 ... from(), of(), just() などで任意の配列・値を変換して作る
- 3 ... Observable.create() で任意の処理から作る

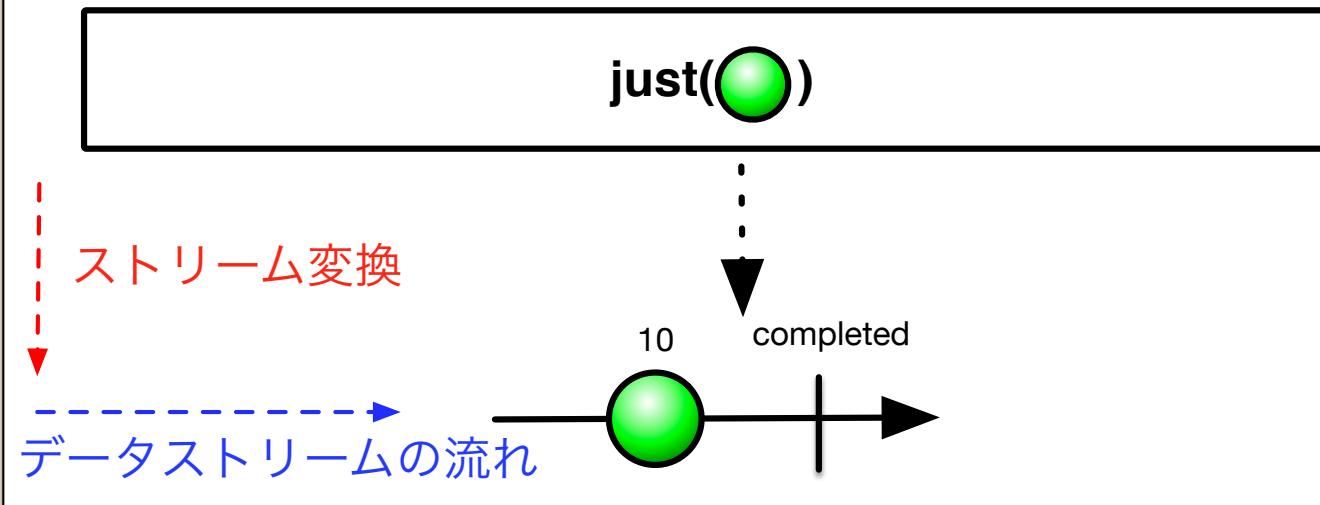
PublishSubject<T>



時間の流れ

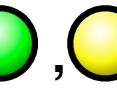
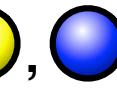
```
let subject = PublishSubject<String>()
subject.onNext("Hello")
subject.onNext("Rx")
subject.onNext("World")
subject.onCompleted()
```

Observable.just(10)



```
let observable1: Observable<Int> = Observable.just(10)
observable1
    .subscribe(
        onNext: { print($0) },
        onError: { print("error: \"\$0\"") },
        onCompleted: { print("complete") }
    ).disposed(by: disposeBag)
// -> 10, complete
```

Observable.of("a", "b", "c", "d", "e")

of(, , , , )

ストリーム変換

データストリームの流れ

"a" "b" "c" "d" "e" completed

```
// Observable.of(_:T, ...) ... 任意の個数の引数からObservableを生成する
let observable3: Observable<String> = Observable.of("a", "b", "c", "d", "e")
observable3
    .subscribe(
        onNext: { print($0) },
        onError: { print("error: \"\($0)\"") },
        onCompleted: { print("complete") }
    ).disposed(by: disposeBag)
// -> "a", "b", "c", "d", "e", complete
```

Observable.from([1, 3, 5, 7, 9])

from([, , , , ])

ストリーム変換

データストリームの流れ

1

3

5

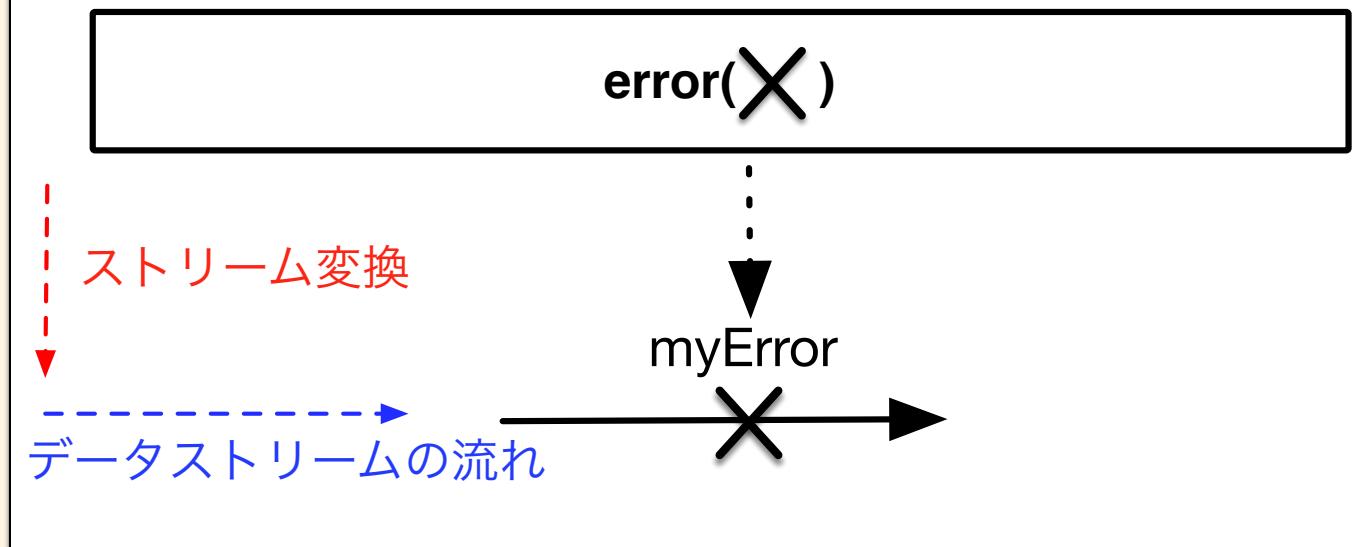
7

9

completed

```
// Observable.from(_: [T]) ... 配列からObservableを生成する
let observable2: Observable<Int> = Observable.from([1,3,5,7,9])
observable2
    .subscribe(
        onNext: { print($0) },
        onError: { print("error: \"\($0)\"") },
        onCompleted: { print("complete") }
    ).disposed(by: disposeBag)
// -> 1, 3, 5, 7, 9, complete
```

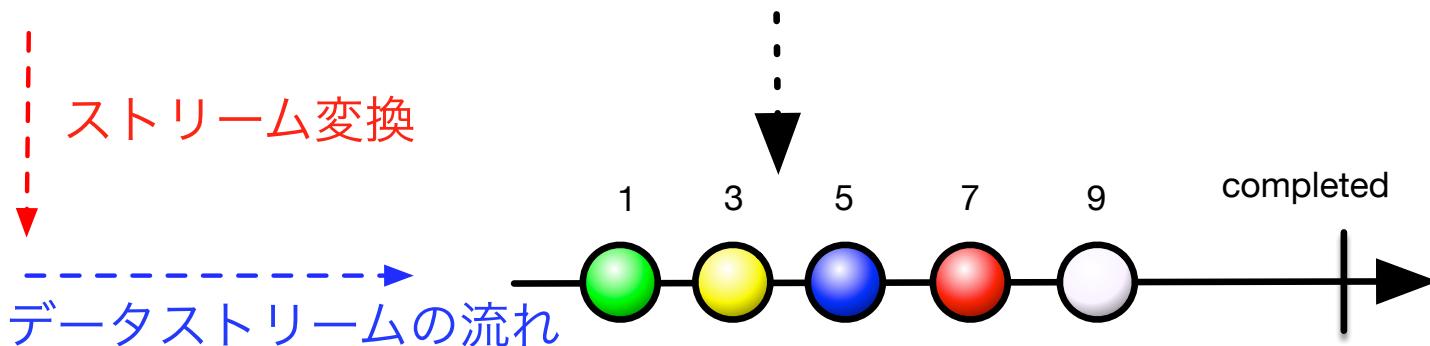
Observable.error(MyError())



```
// Observable.error(_: Error) ... エラーを返すだけのObservableを生成する
let observable4: Observable<Int> = Observable.error(AppError.sample)
observable4
    .subscribe(
        onNext: { print($0) },
        onError: { print("error: \"\($0)\"") },
        onCompleted: { print("complete") }
    ).disposed(by: disposeBag)
// -> error (completeしない)
```

Observable.create(任意の処理)

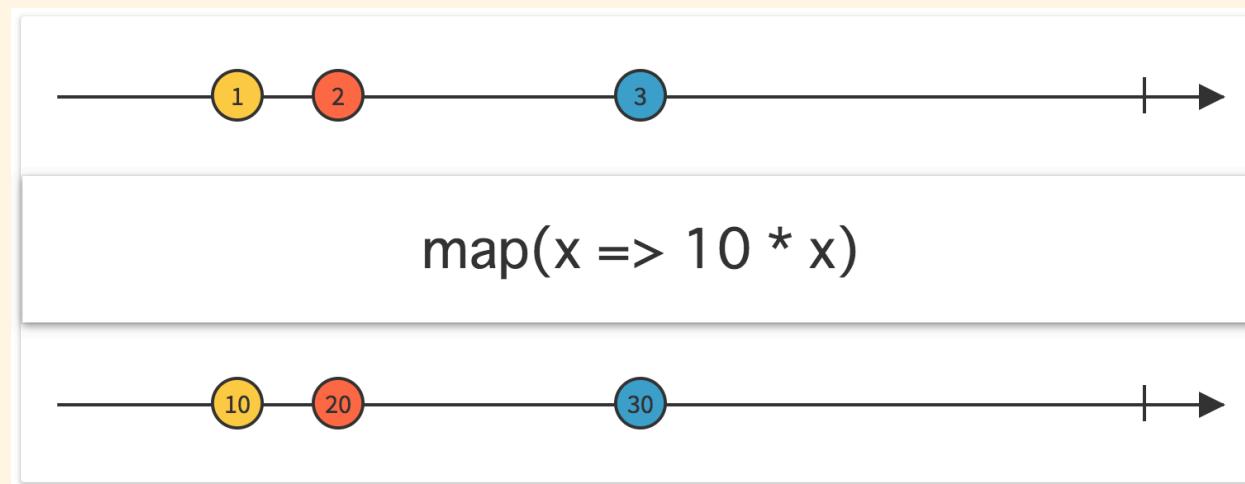
create(任意の処理)



```
let observable5 = Observable<Int>.create { observer in
    observer.onNext(1)
    return Disposables.create()
}
observable5
    .subscribe(
        onNext: { print($0) },
        onCompleted: { print("complete") }
    ).disposed(by: disposeBag)
// -> 1, complete
```

map: ストリームを変換する

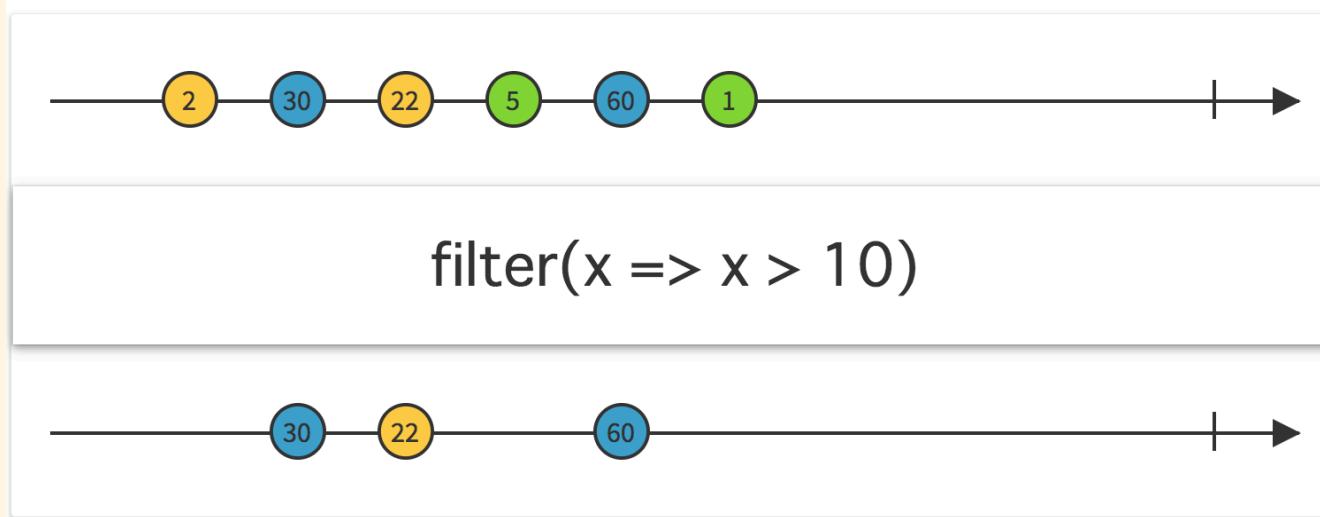
map(_:(T) -> R)) -> Observable<R>



任意の型・値のストリームを加工して別の型・値のストリームに変換します。

filter: ストリームをフィルタする

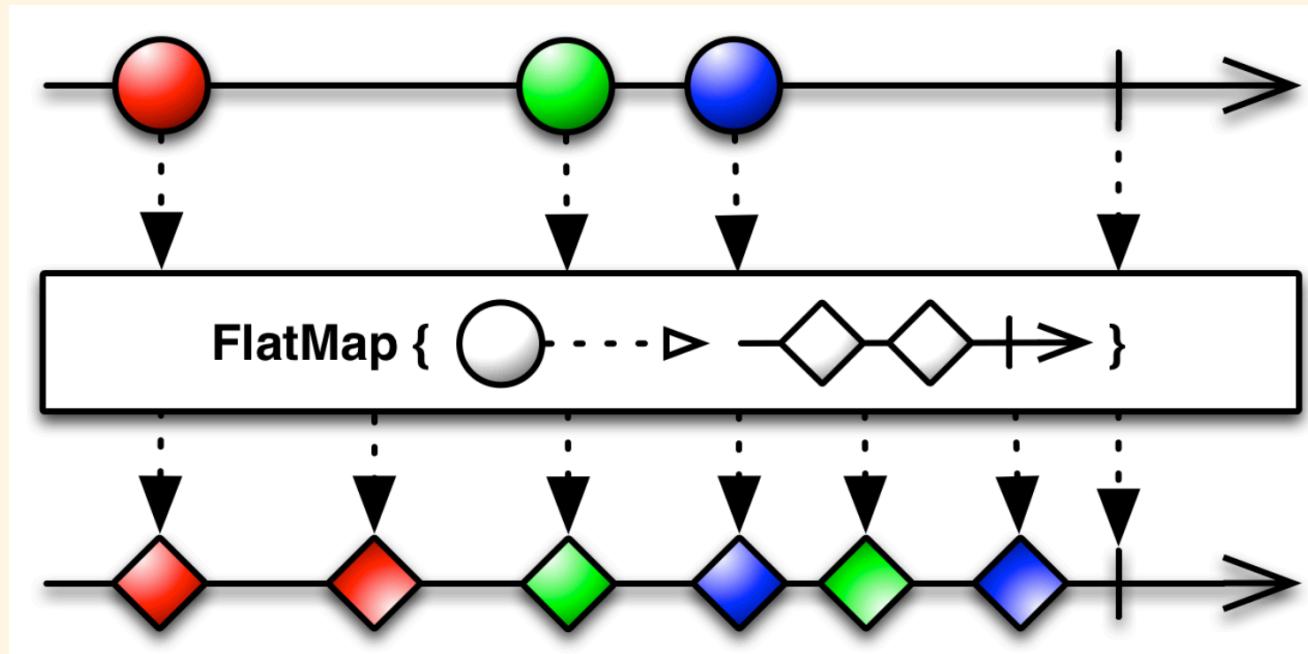
filter(_:(T) -> Bool)) -> Observable<T>



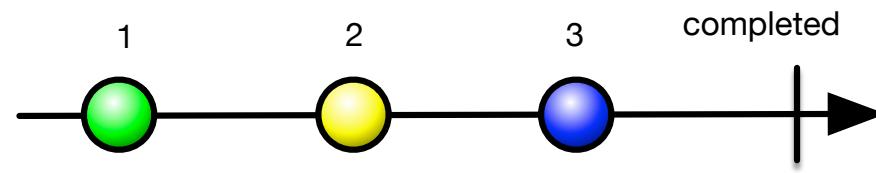
流れてきたストリームを引数の関数で順番に処理し、
結果がtrue になるデータだけのストリームに変換します。

flatMap: ストリームを変換する

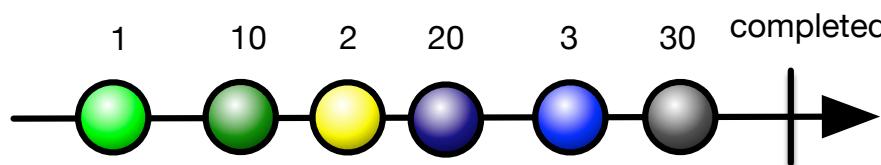
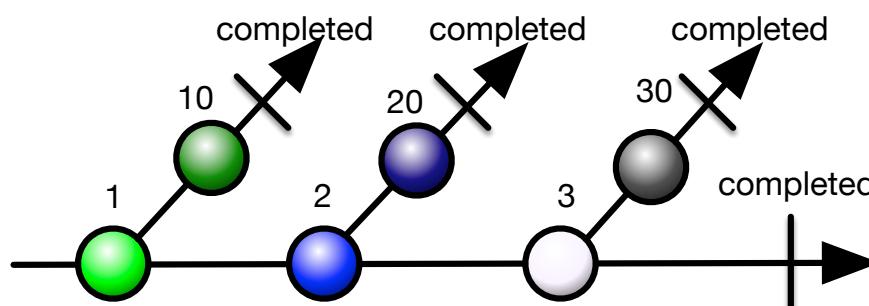
`flatMap(_:(T) -> Observable<R>) -> Observable<R>`



任意の型・値のストリームを加工して別の型・値のストリームに変換します。`map`とほぼ同じですが、クロージャの返り値が`Observable`になっている点が異なります。



flatMap({x in Observable.of(x, x*10)})



filter, map, flatMap の例

```
let observable: Observable<Int> = Observable.of(1,2,3,4,5,6,7,8,9,10)
observable
    .filter { n in n % 2 == 1 } // 1,3,5,7,9
    .map { n in n * 2 } // 2,6,10,14,18
    .flatMap { n in Observable.just("v\((n)") } // "v2","v6","v10","v14","v18"
    .subscribe(onNext: { print($0) }) //
    .disposed(by: disposeBag)
// -> "v2","v6","v10","v14","v18", complete
```

map と flatMap の使い分け

map ... 同期的に処理できるデータ変換に使います。

例 : String型 -> URL型、Int 型を順番に計算して変換する

flatMap ... 非同期処理など、別のストリームを使って変換する時に使います。

例 : twitterのツイートIDストリームをflatMapして、順番にAPIを叩いて内容を取得してタイムライン表示する

map と subscribe の使い分け

(簡単な説明)

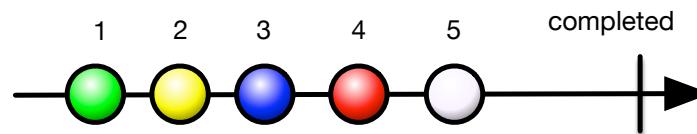
どちらでもonNextについて処理が出来ますが、

subscribe()は繋いだストリームを動かし始めるという大事な役割があります。

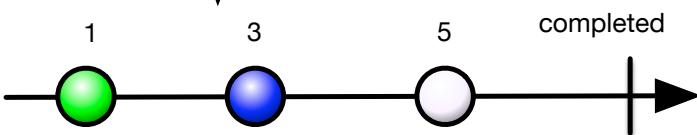
(ストリーム変換の終端)

subscribe() はストリーム変換でなくストリーム実行になるため、Observable
でなくDisposableを返します。このためmapと違うしろにストリームをつな
げることができません。

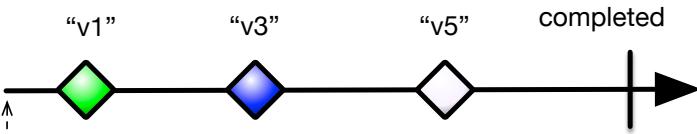
`Observable.from([1, 2, 3, 4, 5])`



`filter({ x in x % 2 == 1})`



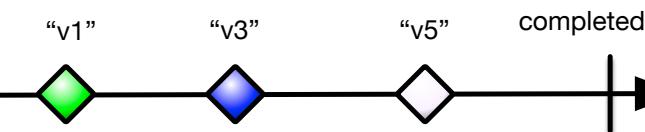
`map({ x in "v\u2028(x)"})`



ストリーム変換

データストリームの流れ

subscribe()



他にも 便利なオペレータが たくさんあります

参考：

<http://reactivex.io/documentation/operators.html>

<http://rxmarbles.com/>



Transforming Observables

Operators that transform items that are emitted by an Observable.

- **Buffer** — periodically gather items from an Observable into bundles and emit these bundles rather than emitting the items one at a time
- **FlatMap** — transform the items emitted by an Observable into Observables, then flatten the emissions from those into a single Observable
- **GroupBy** — divide an Observable into a set of Observables that each emit a different group of items from the original Observable, organized by key
- **Map** — transform the items emitted by an Observable by applying a function to each item
- **Scan** — apply a function to each item emitted by an Observable, sequentially, and emit each successive value
- **Window** — periodically subdivide items from an Observable into Observable windows and emit these windows rather than emitting the items one at a time

Filtering Observables

Operators that selectively emit items from a source Observable.

- **Debounce** — only emit an item from an Observable if a particular timespan has passed without it emitting another item
- **Distinct** — suppress duplicate items emitted by an Observable
- **ElementAt** — emit only item n emitted by an Observable
- **Filter** — emit only those items from an Observable that pass a predicate test
- **First** — emit only the first item, or the first item that meets a condition, from an Observable
- **IgnoreElements** — do not emit any items from an Observable but mirror its termination notification
- **Last** — emit only the last item emitted by an Observable
- **Sample** — emit the most recent item emitted by an Observable within periodic time intervals
- **Skip** — suppress the first n items emitted by an Observable
- **SkipLast** — suppress the last n items emitted by an Observable
- **Take** — emit only the first n items emitted by an Observable
- **TakeLast** — emit only the last n items emitted by an Observable

Combining Observables

Operators that work with multiple source Observables to create a single Observable

- **And / Then / When** — combine sets of items emitted by two or more Observables by means of **Pattern** and **Plan** intermediaries
- **CombineLatest** — when an item is emitted by either of two Observables, combine the latest item emitted by each Observable via a specified function and emit items based on the results of this function
- **Join** — combines items emitted by two Observables whenever an item from one Observable is emitted

便利なオペレータの一例

- **zip, merge, combineLatest ...**

複数のストリームを並列に動かして1つのストリームにまとめる。

- **catchError ...**

エラーが発生したときにリカバリ処理をしてonNextに変換する。再度エラーを投げる(throw)ことで、別のエラーに変換することもできる

- **distinctUntilChanged ...**

重複除外。ストリーム中に変更があったときだけ通知する

- **throttle, debounce ...**

連打防止に使えるフィルタ。データを受け取ったら一定時間間を開けないと次が発火しないたりする

- **toArray ...**

配列にまとめる。completedまで待って1つの配列に変換する

- **groupBy ...**

ストリームのデータを好きな条件でグループ化する

Try Rx!

RxSwift/RxCocoa の インストール

① CocoaPods → Podfile:

```
pod 'RxSwift'  
pod 'RxCocoa'
```

② pod install

③ .swift ファイルでimport

```
import RxSwift
```

または

```
import RxSwift  
import RxCocoa
```

(RxCocoaについては後述)

*Carthage / Swift Package Manager にも対応しています。

<https://github.com/ReactiveX/RxSwift>

Hello, Rx world

```
import RxSwift
```

```
let subject = PublishSubject<String>()

let disposable = subject.subscribe(onNext: {
    print($0)
})

subject.onNext("Hello")
subject.onNext("Rx")
subject.onNext("World")
subject.onCompleted()

disposable.dispose()
```



Hello
Rx
World

Hello, Rx world2

```
import RxSwift

let subject = PublishSubject<String>()

let disposable = subject
    .filter { $0.count >= 2 }      // 2文字以上だけを通す
    .map { $0.lowercased() }       // 英字小文字に変換
    .subscribe(onNext: {
        print($0)                // 流れてきたものを順番に出力
    })

subject.onNext("Hello")
subject.onNext("a")
subject.onNext("Rx")
subject.onNext("b")
subject.onNext("WORLD")
subject.onCompleted()

disposable.dispose()
```



hello
rx
world

Hello, Rx world3

```
let observable1 = Observable.from([true, false, true, true, false, true, false])
let observable2 = Observable.from(["Hell", "Heaven", "o", "Rx", "Fx", "World", "Dog"])
let disposable = Observable
    .zip(observable1, observable2)          // (Observable<Bool>, Observable<String>) to Observable<(Bool, String)>
    .filter { enable, _ in return enable } // Observable<(enable:Bool, text:String)> のうち enable == false を除去
    .map { _, text in return text}        // Observable<(Bool, String)> to Observable<String> ... Hell, o, Rx, World
    .toArray()                          // Observable<String> to Observable<[String]> ... ["Hell", "o", "Rx", "World"]
    .subscribe(onNext: { texts in print(texts.joined(separator: " ")) }) // 配列を文字列として結合して出力 "Hell o Rx World"

disposable.dispose()
```



Hell o Rx World

RxCocoa について

RxSwift のサブセット的なライブラリです。

RxSwift が純粋な Rx 実装で、RxCocoa は iOS/Mac 開発に便利な付属クラス・拡張を提供しています。

RxCocoa を使うには RxSwift とは別にライブラリインストールが必要になります。

```
> pod 'RxCocoa'
```

で pod install して、swift コード内で import RxCocoa を書いておくと使えます。

RxCocoa が提供する機能の例

UIKit 提供クラスの rx 化

UIView, URLSession などUIKitが提供するクラスのプロパティ・イベントをrx化して、Observableインスタンスとの相互バインドを可能にする。

例：

- 数字をテキスト入力したらラベルに計算結果を表示する。
- ボタンを押したらAPI通信して、API実行結果をラベルに表示する。
- URLSession のAPI通信結果JSONを整形してプロフィール画面に表示

知っておくと良いかも。

- RxAlamofire など探せば先人がRx化してくれてるもの、公開されてるコードが結構あります。

まとめ

まとめ

- データのやり取りをストリームとして考えよう
- とりあえずRxのストリーム(Observable)にできた
らこっちのもの。どう加工して目的の形式にする
かを考えよう。
- Rxを極めたらロジックがシンプルに副作用なく、
スマートに書けるようになる。非同期処理も怖く
ない（はず）

さいごに

- tech vein ではこんなアプリエンジニア、
フリーランス・パートナー様を募集しています。
- RxSwift, RxJava を使いたい
- MVVM、CleanArchitecture、DI を活用した設計に興味がある

興味がある方、ちょっと話を聞いてみたい方
ただRxの話をもっと聞きたい方も
ぜひお声がけ下さい！



参考

- 今回のサンプルコード置き場 + RxCocoa を使った簡単なアプリ
 - GitHub:
<https://github.com/ecoopnet/rxswift-beginner>
- 参考記事
 - 何となくRxJavaを使ってるけど正直よく分かってない人が読むと良さそうな記事・基礎編
<https://qiita.com/k-mats/items/4d374460a3f6284dd09f>

時間があればサンプルアプリデモ

ご清聴

ありがとうございました

```
// Observable の結果を subscribe で受け取る
let observable = Observable.from(["a", "b", "c", "d", "e"])
observable.subscribe(
    onNext: { v in
        print(v)
    }).disposed(by: disposeBag)

// Driver の結果を drive で受け取る
let driver = observable.asDriver(onErrorJustReturn: "")
driver.drive(
    onNext: { v in
        print(v)
    }).disposed(by: disposeBag)
```

Driver の使い方は Observable とほぼ同じ

```
// Observable の結果を subscribe で受け取る
let observable = Observable.from(["a", "b", "c", "d", "e"])
observable.subscribe(
    onNext: { v in
        print(v)
    },
    onError: { error in
        // 必要ならエラー処理をここに書く
        print(error)
    }
).disposed(by: disposeBag)

// Driver の結果を drive で受け取る
let driver = observable.asDriver(onErrorJustReturn: "")
driver.drive(
    onNext: { v in
        print(v)
    }
    // 絶対にエラーしないので drive() には onError: がない
    // (エラーになったら onErrorJustReturn: "" の効果で "" になる)
).disposed(by: disposeBag)
```

Driver の使い方は Observable とほぼ同じ？

→ エラー処理が要らない等の細かい違いはある

*

Driver (補足)

具体的には observable.asDriver(onErrorJustReturn: "") とすると

- UIスレッドで処理 :

```
.observeOn(MainScheduler.instance)
```

- Cold -> Hot 変換 :

```
.share(replay:1)
```

- エラー時に監視が止まるのを防ぐためのリカバリ処理 :

```
.catchError({ _ in "" })
```

をまとめてやってくれるようになります。