Observable

ObservableType & Observable

```
protocol ObservableType {
    associatedtype E
    func subscribe<O: ObserverType>(_ observer: O) -> Disposable where O.E == E
}

class Observable<Element> : ObservableType {
    public typealias E = Element
    public func subscribe<O>(_ observer: O) -> Disposable where Element == O.E, O : ObserverType
}
```

RxSwift.Event

- ObservableType의 값이 될 수 있는 타입

```
// RxSwift.Event
enum Event<Element> {
    case next(Element)
    case error(Error)
    case completed
}

// Swift.Optional
enum Optional<T> {
    case some(T)
    case none
}

// Swift.Result
enum Result<Success, Failure> where Failure : Error {
    case success(Success)
    case failure(Failure)
}
```

각 Event 타입에 해당하는 Observable의 생성

```
enum MyError: Error { case unknown }

// 생성
let one: Observable<Int> = Observable.just(1)
let unknown: Observable<Int> = Observable.error(MyError.unknown)
let completed: Observable<Int> = Observable.empty()
```

ObserverType

```
protocol ObserverType {
    associatedtype E
    func on(_ event: Event<E>)
}

class IntegerPrinter: ObserverType {
    func on(_ event: Event<Int>) {
        switch event {
        case let .next(value):
            print("value: ", value)
        case let .error(error):
            print("error: ", error.localizedDescription)
        case .completed:
            print("completed")
        }
    }
}

let x: Observable<Int> = Observable.just(1)
let integerPrinter = IntegerPrinter()
let disposeBag = DisposeBag()

x.subscribe(integerPrinter).disposed(by: disposeBag)
```

Anonymous Observer

```
let numbers$: Observable<Int> = Observable.from([1,2,3])

numbers$.subscribe { event in
    switch event {
    case let .next(value):
        print("value: ", value)
    case let .error(error):
        print("error: ", error.localizedDescription)
    case .completed:
        print("completed")
    }
}
.disposed(by: disposeBag)

// Array 사용
let numbers: Array<RxSwift.Event<Int>> = [.next(1), .next(2), .next(3)]

numbers.forEach { event in
  switch event {
    case let .next(value):
        print("value: ", value)
    case let .error(error):
        print("error: ", error.localizedDescription)
    case .completed:
        print("completed")
    }
}
```

ObservableType의 규칙

- `next`가 0번 이상 발생할 수 있음.
- `completed` 또는 `error`가 발생하면 종료됨.
- next가 한 번도 발생하지 않고 completed나 error로 종료될 수 있음
- next, error, completed 중 아무것도 발생하지 않을 수 있음.

Observable은 Event를 값으로 갖는 Push Collection으로 값이 발생할 때마다 자신을 subscribe하고 있는 observer에게 Event 타입의 값을 보낸다.

Hot & Cold Observable

Observable은 언제 값을 처리하기 시작하는지는 Observable이 Hot인지 Cold인지에 따라 다르다.

Hot Observables	Cold observables
옵저버가 있건 없건 상관없이 자원(CPU, 메모리)를 사용	옵저버가 구독할 때까지 자원을 사용하지 않음
변수 / 프로퍼티 / 상수, 터치 좌표, 마우스 좌표, UIControl 값, 현재 시간 등을 표현할 때 사용	비동기 연산, HTTP 연결, TCP 연결 등을 표현할 때 사용
보통 1개 이상의 값을 가짐	보통 1개의 값을 가짐
옵저버가 있건 없건 상관없이 값이 전달됨	옵저버가 있을 경우에만 값이 전달됨
보통 계산된 결과를 구독하는 옵저버들이 공유한다	보통 새로운 옵저버가 구독할 때마다 계산을 다시 한다.
일반적으로 상태를 가지고 있다.	일반적으로 상태를 가지지 않는다

Disposable

```
public protocol Disposable {
    func dispose()
}

let getGooglePage: Observable<Data> = Observable.create { (observer: AnyObserver<Data>) in
    let url: URL = URL(string: "https://google.com")!
    let dataTask = URLSession.shared.dataTask(with: url, completionHandler: { (data, response, error) in
        if let data = data {
            observer.onNext(data)
            observer.onCompleted()
        }

        if let error = error {
            observer.onError(error)
        }
    })
    dataTask.resume()

    return Disposables.create {
        dataTask.cancel()
    }
}

let subscription: Disposable = getGooglePage
    .subscribe(onNext: { (data: Data) in
        print(data)
    })

subscription.dispose()
```

DisposeBag

```
import RxCocoa

class MyViewController {
    let button1: UIButton = UIButton()
    let button2: UIButton = UIButton()

    var subscriptions: [Disposable] = []

    func viewDidLoad() {
        super.viewDidLoad()
        let subscription1 = button1.rx.tap
            .subscribe(onNext: { [weak self] in
                // ...
            })
        subscriptions.append(subscription1)

        let subscription2 = button2.rx.tap
            .subscribe(onNext: { [weak self] in
                // ...
            })
        subscriptions.append(subscription2)
    }

    deinit {
        subscriptions.forEach { $0.dispose() }
    }
}
```

```
import RxCocoa

class MyViewController {
    let button1: UIButton = UIButton()
    let button2: UIButton = UIButton()

    var disposeBag: DisposeBag = DisposeBag()

    func viewDidLoad() {
        super.viewDidLoad()
        button1.rx.tap
            .subscribe(onNext: { [weak self] in
                // ...
            })
            .disposed(by: disposeBag)

        button2.rx.tap
            .subscribe(onNext: { [weak self] in
                // ...
            })
            .disposed(by: disposeBag)
    }
}
```

Subject

- `ObservableType`이자 `ObserverType`

PublishSubject

```
import Foundation
import RxSwift

let disposeBag = DisposeBag()
let numbers$ = PublishSubject<Int>()

numbers$
    .subscribe(onNext: { print("Observer 1: \($0)") })
    .disposed(by: disposeBag)

numbers$.onNext(0)
numbers$.onNext(1)

numbers$.subscribe(onNext: { print("Observer 2: \($0)") })
    .addDisposableTo(disposeBag)

numbers$.onNext(2)
numbers$.onNext(3)
numbers$.onCompleted()

/*
Observer 1: 0
Observer 1: 1
Observer 1: 2
Observer 2: 2
Observer 1: 3
Observer 2: 3
*/
```

- 이벤트를 표현할 때 유용함

ReplaySubject

```
import Foundation
import RxSwift

let disposeBag = DisposeBag()
let numbers$ = ReplaySubject<Int>.create(bufferSize: 2)

numbers$
    .subscribe(onNext: { print("Observer 1: \($0)") })
    .disposed(by: disposeBag)

numbers$.onNext(0)
numbers$.onNext(1)

numbers$.subscribe(onNext: { print("Observer 2: \($0)") })
    .addDisposableTo(disposeBag)

numbers$.onNext(2)
numbers$.onNext(3)
numbers$.onCompleted()

/*
Observer 1: 0
Observer 1: 1
Observer 2: 0
Observer 2: 1
Observer 1: 2
Observer 2: 2
Observer 1: 3
Observer 2: 3
*/
```

- onNext 이후에 구독이 이루어져도 bufferSize만큼 next가 발행됨

BehaviorSubject

```swift
import Foundation
import RxSwift

let disposeBag = DisposeBag()
let numbers$ = BehaviorSubject(value: -1)

numbers$
    .subscribe(onNext: { print("Observer 1: \($0)") })
    .disposed(by: disposeBag)

numbers$.onNext(0)
numbers$.onNext(1)

numbers$.subscribe(onNext: { print("Observer 2: \($0)") })
    .addDisposableTo(disposeBag)

numbers$.onNext(2)
numbers$.onNext(3)
numbers$.onCompleted()

/*
Observer 1: -1
Observer 1: 0
Observer 1: 1
Observer 2: 1
Observer 1: 2
Observer 2: 2
Observer 1: 3
Observer 2: 3
*/
```

- 생성시 최초값을 전달해야 함
- onNext 이후에 구독이 이루어져도 최근 값 1개가 next가 발행됨
- 상태를 표현할 때 유용함

Subject는 왜 사용하는가?

- 명령형 프로그래밍과 반응형 프로그래밍간 전환이 필요할 때 사용
- Erik Meijer가 싫어함
- `shamefullySendNext`

그 외의 타입들

- Relay:
  - Subject를 감싼 타입. completed나 error가 발생하지 않음
  - 즉, next 타입만 발생시키며 종료되지 않음
- Trait
  - Single: success나 error 중 하나가 한 번만 발생. 다른 언어의 Promise, Task와 같음
  - Completable: complete나 error 중 하나가 한 번만 발생. Single<Void>와 같다.
  - Maybe: next, complete, error 중 하나가 한 번만 발생
  - Driver: error가 발생할 수 없음. 메인 스레드에서 동작. 사이드 이펙트 실행을 공유
  - Signal: Driver와 같으나 Subscription시 마지막 값을 Replay하는 동작이 없음.
  - ControlProperty:
  - ControlEvent: