Operators

- Push Collection(= Observable)의 값(= Event)을 조작하는 함수.
- Observable의 생성, 변형, 필터링, 결합, 에러 처리, 비동기 처리 등 수십개의 연산자가 존재.
- Rx Documentation 참고.

RxMarble

이벤트 스트림(Observable)을 시각적으로 이해하기 쉽게 표현

```
/*
-    시간의 흐름(편의상 1초)
1    .next
|    .completed
x    .error
*/
```

Observable 생성

```swift
let one$: Observable<Int> = Observable.just(1)
// 1|

let error$: Observable<Int> = Observable.error("error!")
// X

let empty$: Observable<Int> = Observable.empty()
// |

let flag$: Observable<Bool> = Observable.of(true, false)
// TF|

let number$: Observable<Int> = Observable.from([1,2,3])
// 123|

let interval$: Observable<Int> = Observable<Int>.interval(.seconds(1), scheduler: MainScheduler.instance)
// 0-1-2-3-4-5-6-7...

let getGooglePage$: Observable<Data> = Observable.create { _ in //... }
// ---....---(Data)|
// ---....---X
```

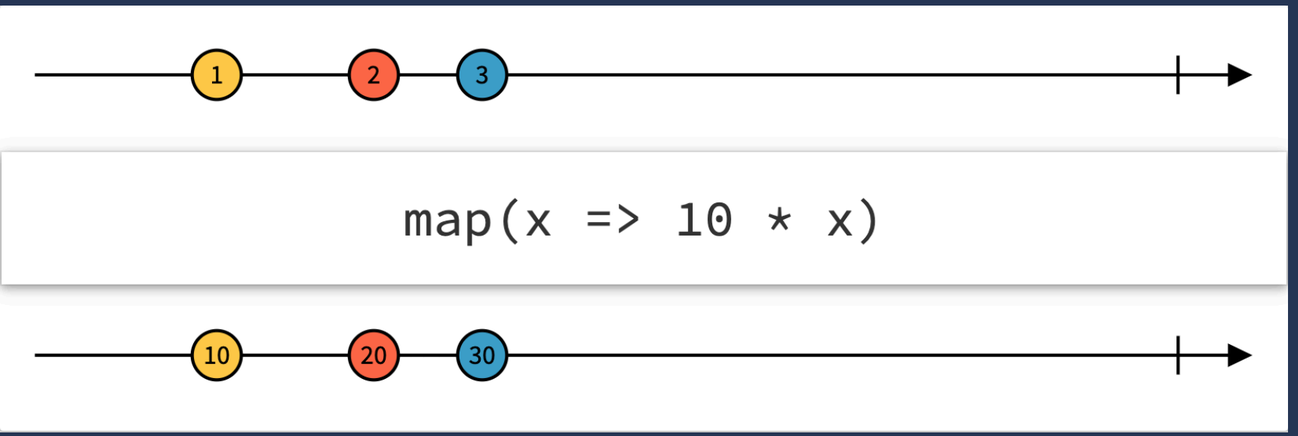
Observable Filtering

```
let interval$: Observable<Int> = Observable<Int>.interval(.seconds(1), scheduler: MainScheduler.instance)
// 0-1-2-3-4-5-6-7...

let oddNumber$: Observable<Int> = interval$.filter { $0 % 2 == 1 }
// --1---3---5---7

let lessThan4$: Observable<Int> = interval$.filter { $0 < 4 }
// 0-1-2-3--------...

Let firstThree$: Observable<Int> = interval$.take(3)
// 0-1-2|

Let skipThree$: Observable<Int> = interval$.skip(3)
// ------3-4-5-6-7...

let numbers = [1,2,3,4,4,5,5,3]
let number$ = Observable<Int>.interval(.seconds(1), scheduler: MainScheduler.instance)
    .map { i in numbers[i%numbers.count] }
// 1-2-3-4-4-5-5-3-1-2-3-4...

number$.distinctUntilChanged()
// 1-2-3-4---5---3-1-2-3-4...
```

Observable 변형

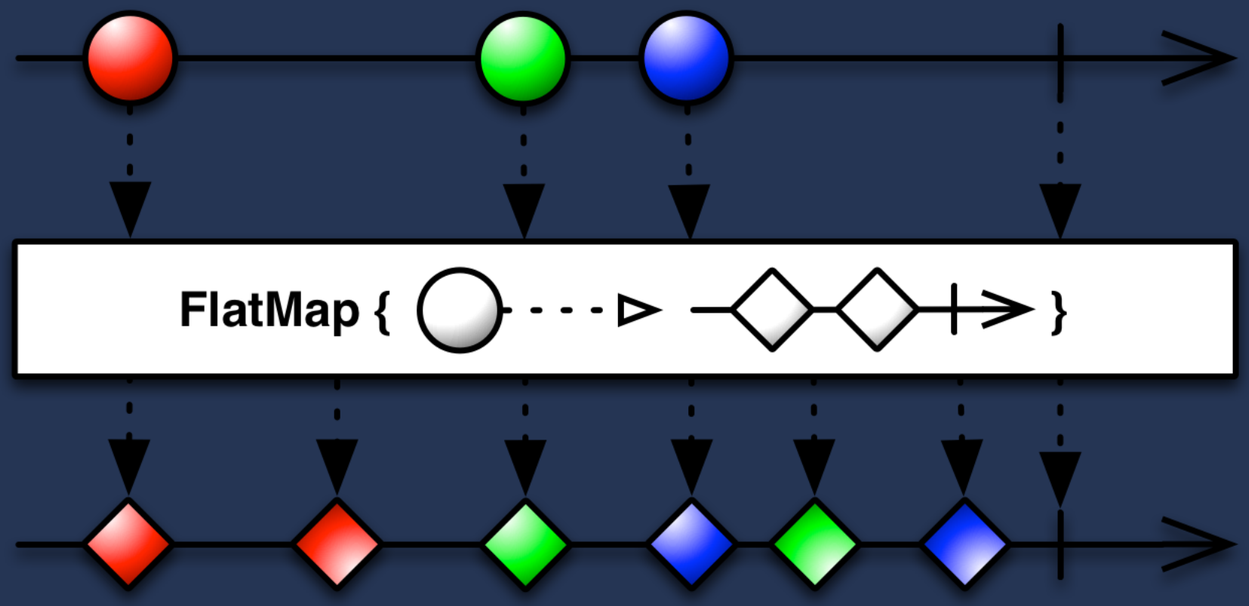
```
let number$: Observable<Int> = Observable.from([1,2,3])
// 123|

let numbers$ = number$.toArray()
// [1,2,3]|

let interval$: Observable<Int> = Observable<Int>.interval(.seconds(1), scheduler: MainScheduler.instance)
// 0-1-2-3-4-5-6-7...

let string$ = interval$.map { $0.description }
// "0"-"1"-"2"-"3"-"4"-"5"...

let plusTen$ = interval$.map { $0 + 10 }
// 10-11-12-13-14-15-16...
```

map



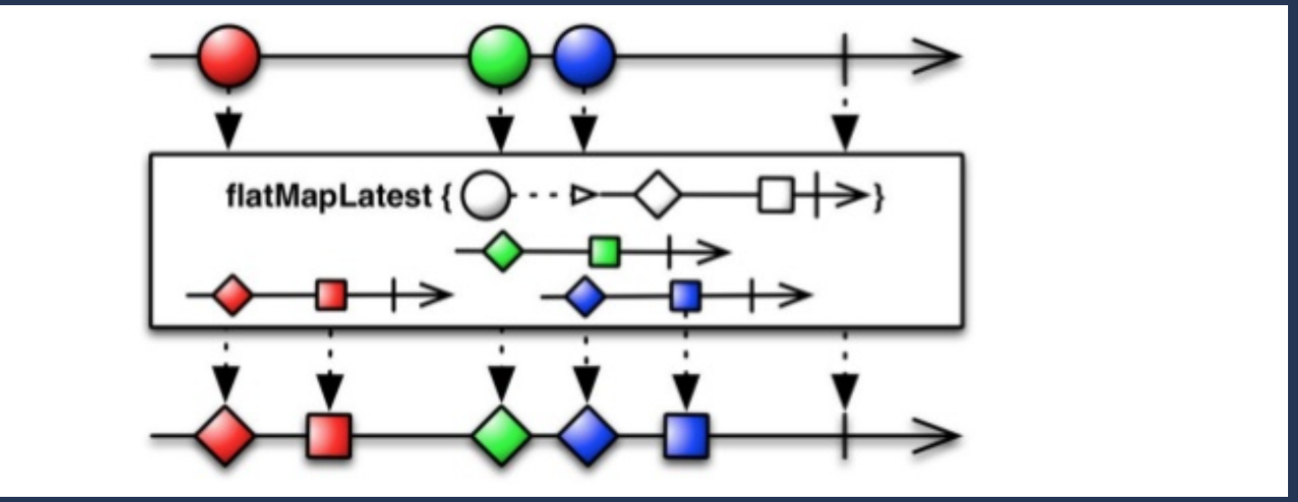

```
let numbers = Observable.from([1,2,3])
let disposeBag = DisposeBag()

numbers
    .map { $0 * 10 }
    .subscribe(onNext: {
        print($0)
    })
    .disposed(by: disposeBag)
```

flatMap



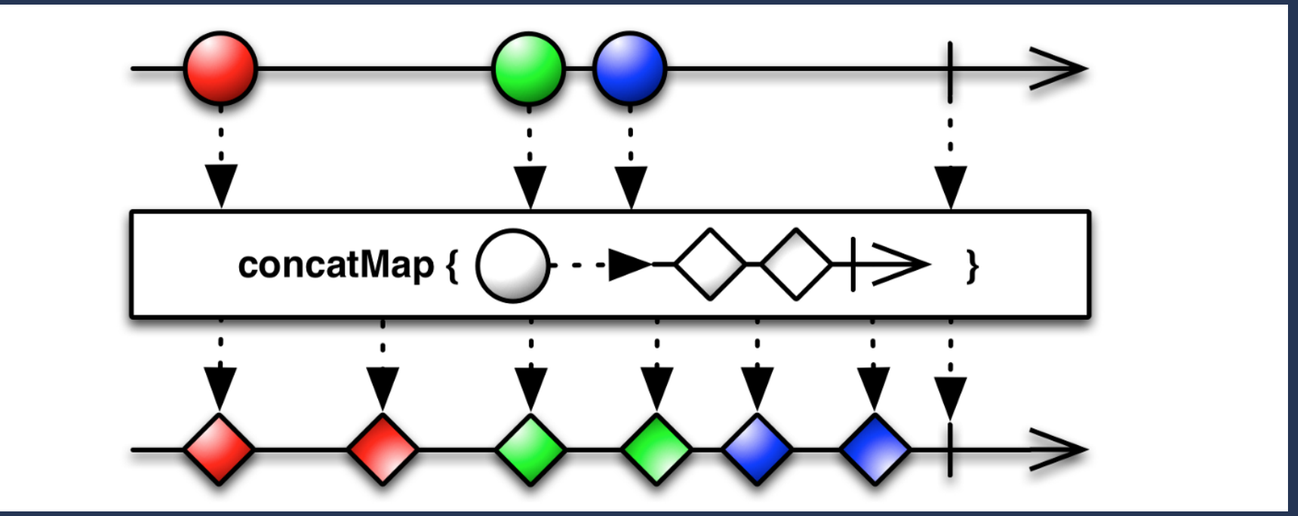

```
let source$ = Observable<Int>
    .interval(.seconds(2), scheduler: MainScheduler.instance)
    .take(2)

let target$ = source$.flatMap { value in
    return Observable<Int>
        .interval(.seconds(1), scheduler: MainScheduler.instance)
        .take(3)
        .map { innerValue in "source: \(value), target: \(innerValue)" }
}

target$
    .subscribe(onNext: { print($0) })
    .disposed(by: disposeBag)
```

flatMapLatest




```
let source$ = Observable<Int>
    .interval(.seconds(2), scheduler: MainScheduler.instance)
    .take(2)

let target$ = source$.flatMapLatest { value in
    return Observable<Int>
        .interval(.seconds(1), scheduler: MainScheduler.instance)
        .take(3)
        .map { innerValue in "source: \(value), target: \(innerValue)" }
}

target$
    .subscribe(onNext: { print($0) })
    .disposed(by: disposeBag)
```

concatMap



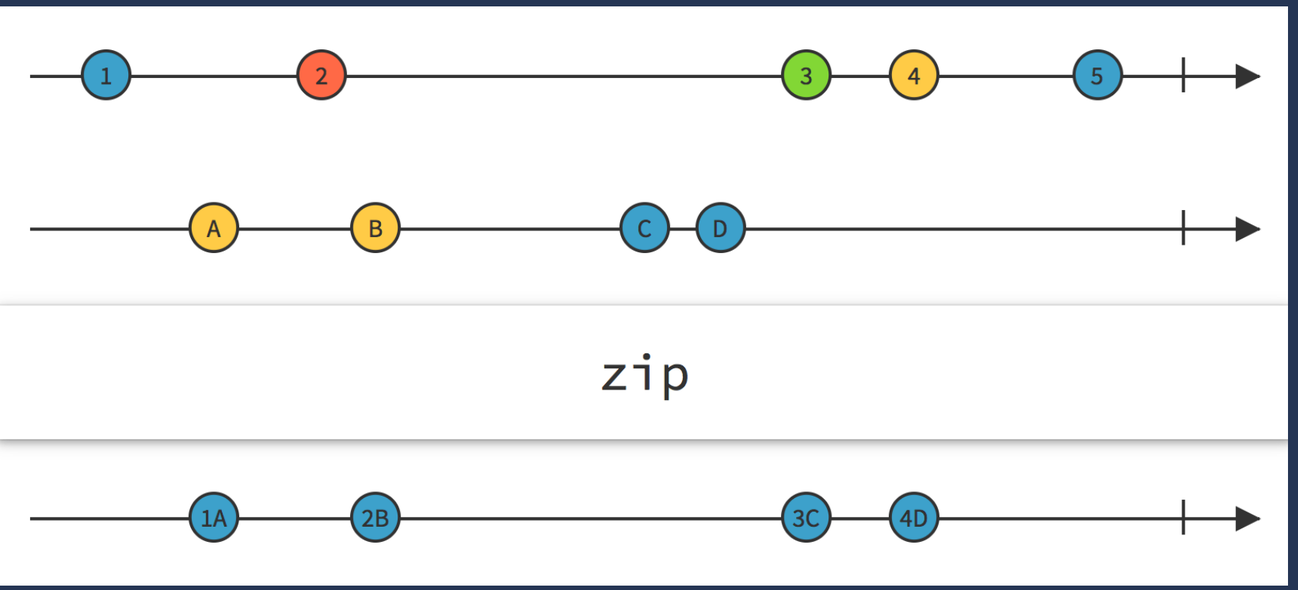

```
let source$ = Observable<Int>
    .interval(.seconds(2), scheduler: MainScheduler.instance)
    .take(2)

let target$ = source$.concatMap { value in
    return Observable<Int>
        .interval(.seconds(1), scheduler: MainScheduler.instance)
        .take(3)
        .map { innerValue in "source: \(value), target: \(innerValue)" }
}

target$
    .subscribe(onNext: { print($0) })
    .disposed(by: disposeBag)
```

map & flatMap

- 가장 중요한 연산자
- `Optional`, `Result`, `Array`등의 타입 모두 `map`과 `flatMap`을 가지고 있음.
  - map (함자)
  - flatMap (모나드)
- `flatMap`, `flatMapLatest`, `concatMap`을 잘 구분해서 사용할 것

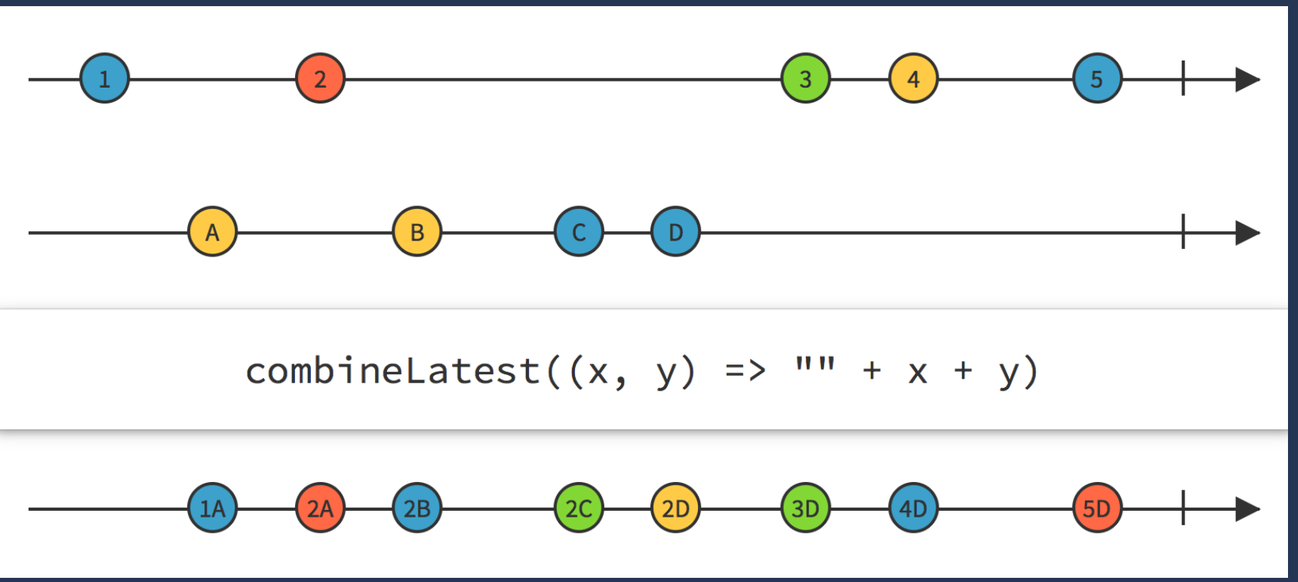
Observable 결합

Observable.zip



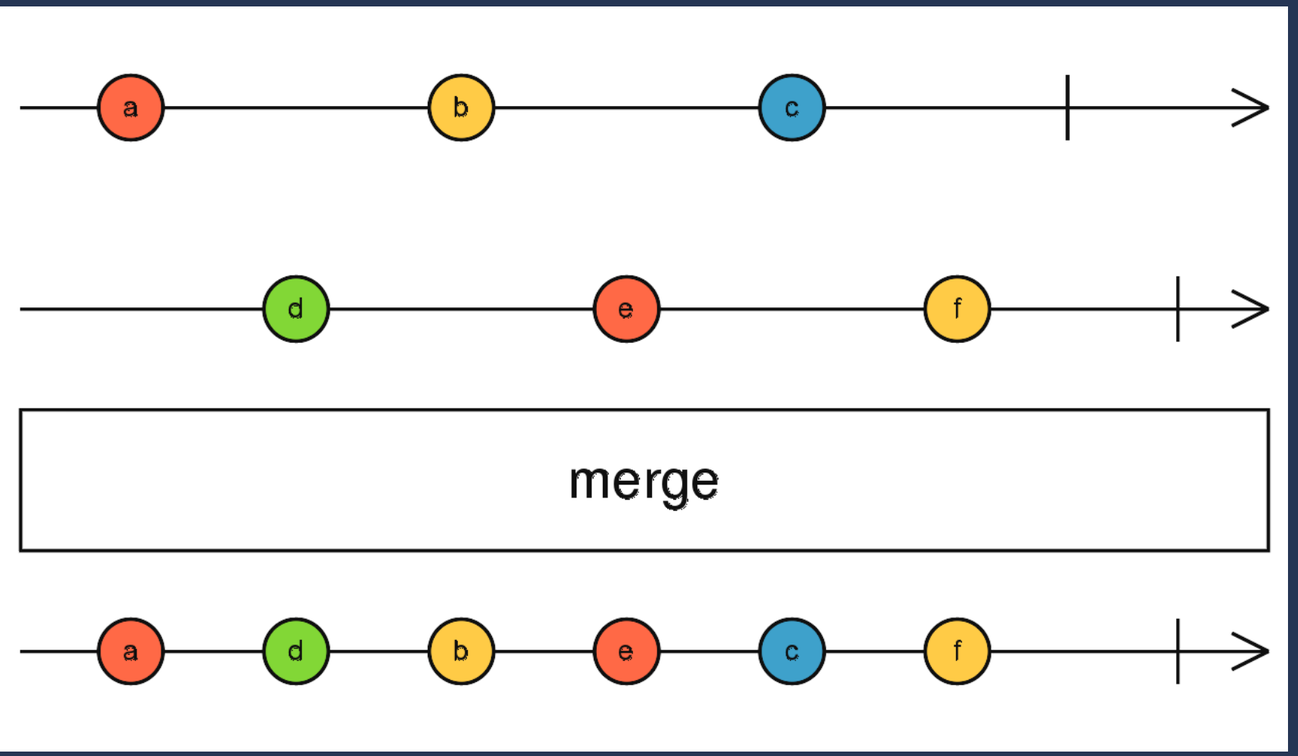

```
let disposeBag = DisposeBag()

let number$ = Observable<Int>
    .interval(.milliseconds(300), scheduler: MainScheduler.instance)
    .take(5)

let letter$ = Observable<Int>
    .interval(.milliseconds(500), scheduler: MainScheduler.instance)
    .take(3)
    .map { ["A", "B", "C"][$0] }

let combined$ = Observable.zip(number$, letter$)

combined$
    .subscribe(onNext: { print($0) })
    .disposed(by: disposeBag)
```

Observable.combineLatest




```
let disposeBag = DisposeBag()

let number$ = Observable<Int>
    .interval(.milliseconds(300), scheduler: MainScheduler.instance)
    .take(5)

let letter$ = Observable<Int>
    .interval(.milliseconds(500), scheduler: MainScheduler.instance)
    .take(3)
    .map { ["A", "B", "C"][$0] }

let combined$ = Observable.zip(number$, letter$)

combined$
    .subscribe(onNext: { print($0) })
    .disposed(by: disposeBag)
```

Observable.merge




```
let disposeBag = DisposeBag()

let number$ = Observable<Int>
    .interval(.milliseconds(300), scheduler: MainScheduler.instance)
    .take(5)

let number2$ = Observable<Int>
    .interval(.milliseconds(500), scheduler: MainScheduler.instance)
    .take(3)

let combined$: Observable<Int> = Observable.merge(number$, number2$)

combined$
    .subscribe(onNext: { print($0) })
    .disposed(by: disposeBag)
```

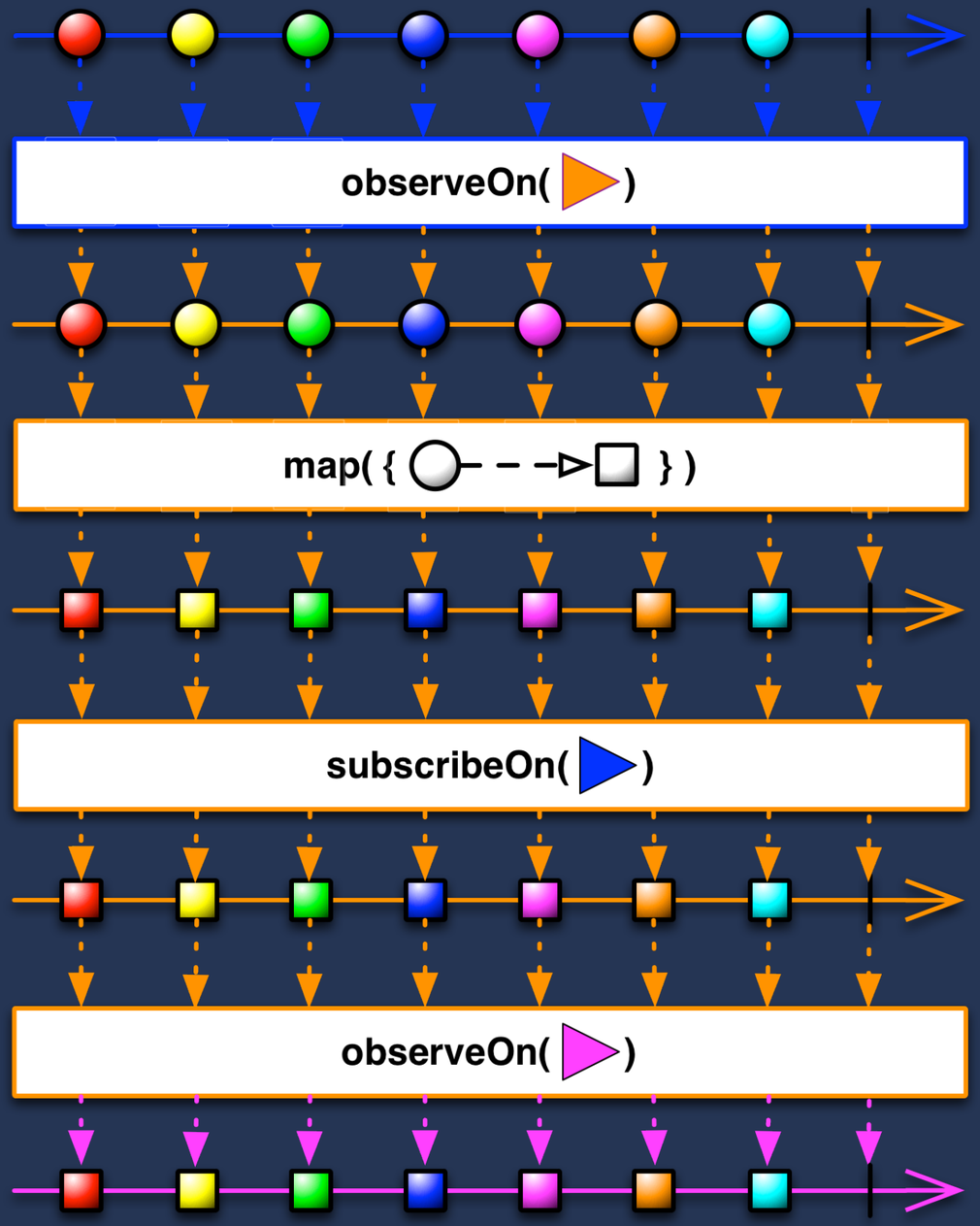
그 밖에 자주 사용하는 Operator

- scan
- delay
- debounce
- withLatestFrom

언어마다 조금씩 이름이 다를 수 있음

Schedulers

- 코드를 무슨 스레드에서 어떻게 실행할지 설정할 때 사용.
- 직렬(Serial)과 병렬(Concurrent) 타입이 있다.
- `observeOn` & `subscribeOn` 연산자를 이용해 설정한다.



observeOn vs subscribeOn

```
let numbers: Observable<Int> = Observable.from([1,2,3])
let orangeScheduler = SerialDispatchQueueScheduler(qos: .background)
let blueScheduler = ConcurrentDispatchQueueScheduler(qos: .default)
let pinkScheduler = MainScheduler.instance
let disposeBag = DisposeBag()

numbers
  .observeOn(orangeScheduler)
  .map { $0.description }
  .subscribeOn(blueScheduler)
  .observeOn(pinkScheduler)
  .subscribe(onNext: {
    print($0)
  })
  .disposed(by: disposeBag)
```

subscribeOn

- subscribeOn 연산자를 이용해 다른 스레드에서 실행을 **시작**할 수 있다.
- 기본적으로 subscribe 메소드가 호출된 스레드에서 코드가 실행된다.

observeOn

- 다음 Observable 체인의 코드가 실행될 스케쥴러를 지정한다.
- 일반적으로 subscribeOn보다 자주 사용한다.
- observeOn을 사용해 스케쥴러를 따로 지정하지 않으면 이벤트가 넘어온 스케쥴러에서 실행된다.

내장 Scheduler

- CurrentThreadScheduler - 직렬
- MainScheduler - 직렬
  - UI 관련 코드를 실행
- SerialDispatchQueueScheduler - 직렬
- ConcurrentDispatchQueueScheduler - 병렬
  - 백그라운드에서 병렬로 작업을 실행하기에 좋음
- OperationQueueScheduler - 병렬
  - NSOperationQueue 기반의 스케쥴러
  - maxConcurrentOperationCount를 설정할 수 있어 큰 데이터를 병렬처리하기에 좋음.

구현 Tips

1. 사이드 이펙트는 subscribe에서만 실행해라.
2. 연산자에서 사용하는 함수는 순수함수여야 한다.
3. doOn 함수는 사용하지 않는다.
4. subscribe를 중복해 사용하지 않는다.
    - flatMap을 사용한다.