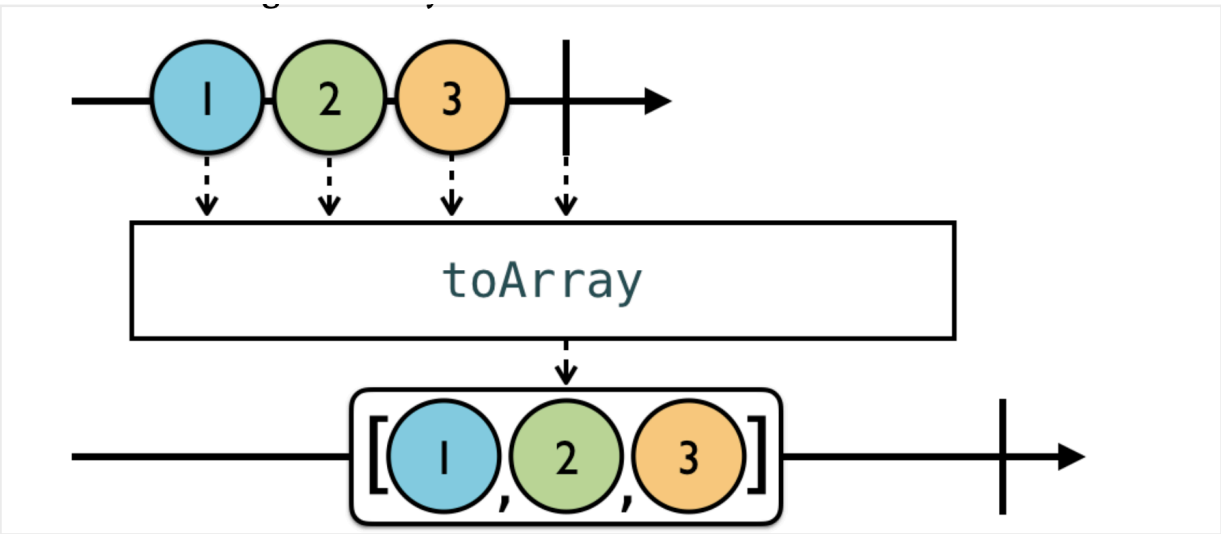
RXSwift - Transforming Operator

✅ Transforming elements

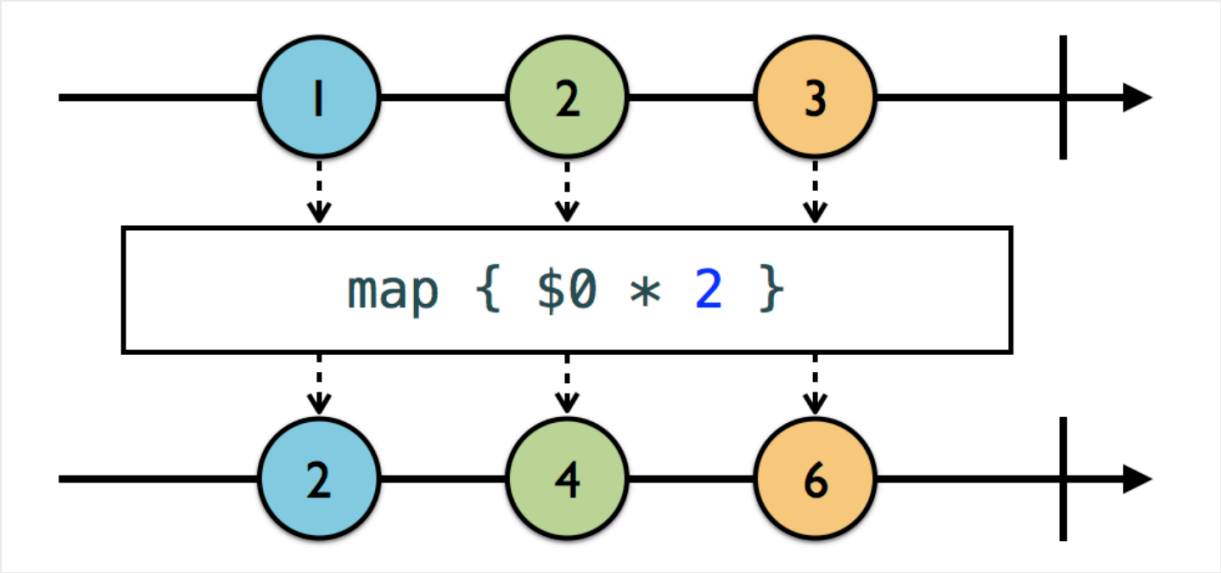
- **toArray**
  - **Convert** an **observable sequence** of elements **into an array** of those elements once the observable completes
  - And emit a .next event containing that array to subscribers.




```
// 1. Create a finite observable of letters.
Observable.of("A", "B", "C")
    //2. Use toArray to transform the elements into an array.
    .toArray()
    .subscribe{
        print("Observable 1: ", $0)
    }
    .disposed(by: disposeBag)
```

```
Observable 1:  success(["A", "B", "C"])
```

- **map**:
  - RxSwift's map operator works just like Swift's standard map, except it operates on observables.
  - It applies our given implementation (closure) to each item emitted from the source observable.
    - And, returns an observable with the items that we get after implementing on the items of source observable.



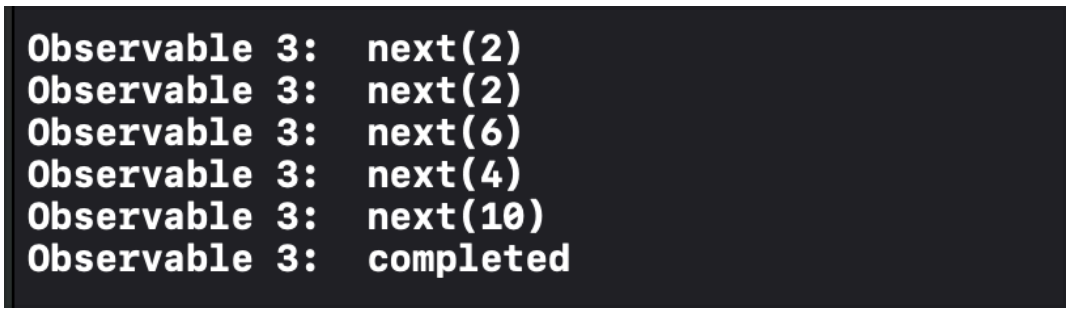
EX1: Spell out element value:

```
41  let formatter = NumberFormatter()
42  formatter.numberStyle = .spellOut
43
44  Observable<NSNumber>.of(123,5,32)
45      .map { item in
46          formatter.string(from: item) ?? ""
47      }
48      .subscribe{
49          print("Observable 2: ", $0)
50      }
51      .disposed(by: disposeBag)
52
```

```
Observable 2:  next(one hundred twenty-three)
Observable 2:  next(five)
Observable 2:  next(thirty-two)
Observable 2:  completed
```

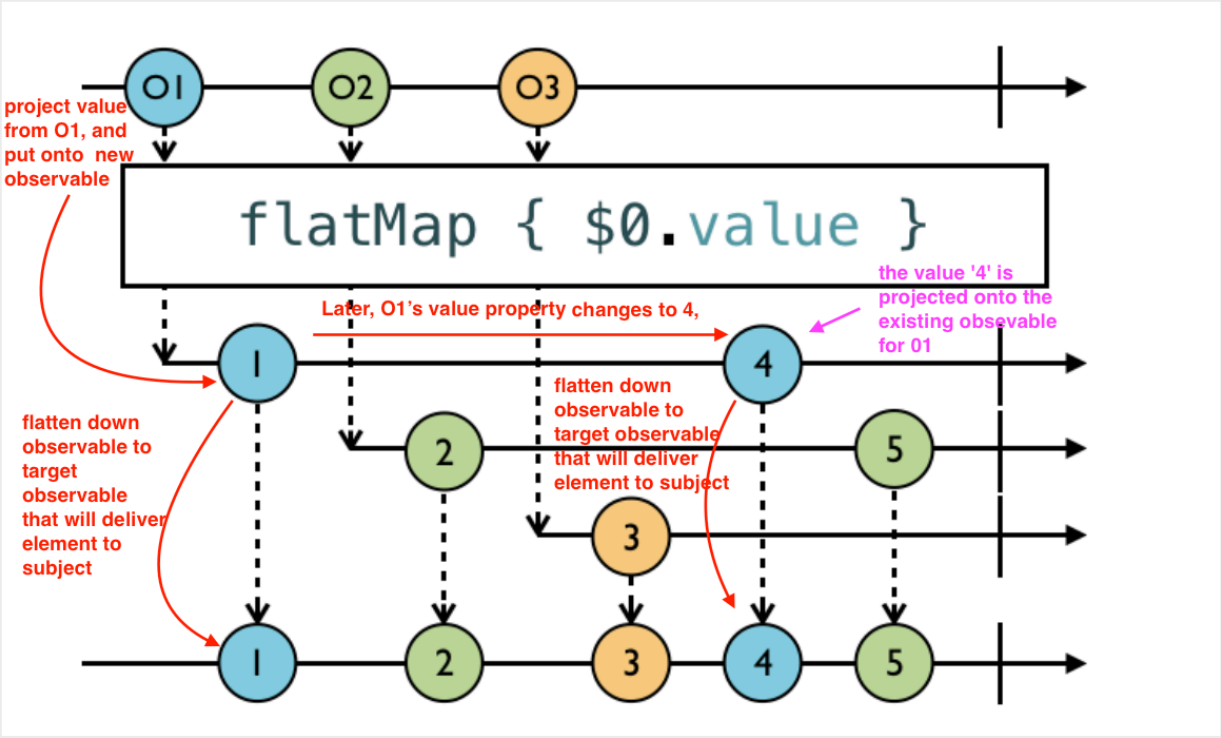
EX2: Multiple odd elements by 2:

```swift
Observable.of(1,2,3,4,5)
    // use enumerated() to make tuple (index,item)
    .enumerated()
    // use map to multiply odd number by 2
    .map { index, item in
        index % 2 == 0 ? item * 2 : item
    }
    .subscribe{
        print("Observable 3: ", $0)
    }
    .disposed(by: disposeBag)
```



Transforming inner observables

- **flatMap**
- **flatMap** projects and **transforms** an observable value *of* an observable, and then **flattens** it **down** to a target observable.



- **flatMap** keeps up with each and every observable it creates, *one for each element* added onto the source observable
    - it will project changes from each and every observables inside it.

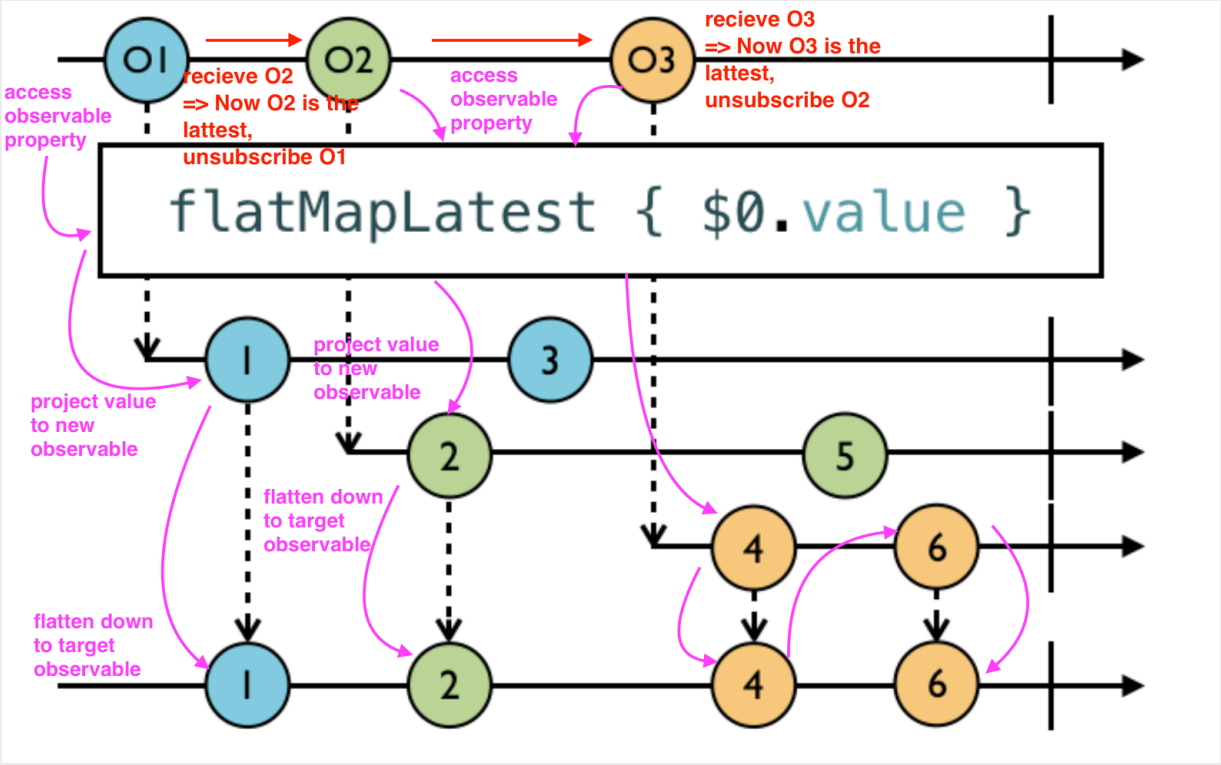
EX:

```swift
// 1. Create two instances of Student, laura and charlotte.
  let laura = Student(score: BehaviorSubject(value: 80))
  let charlotte = Student(score: BehaviorSubject(value: 90))

// 2. Create a Source subject of type 'Student'
  let student = PublishSubject<Student>()

  student
    // 3. Use flatMap to reach into the student subject and project its
        content(score).
    .flatMap {
        observable element type Student
        $0.score                    get value (score) and put it as target observable
    }                               => in subscribe, we access score value= $0.element
    .subscribe{
        print("Subject1 : ",$0)
    }                               observable element type Int (for holding score)
    .disposed(by: disposeBag)

// 4. Add laura as an element onto Source Subject
student.onNext(laura)

// 5. Change laura's score
    // The change will be notified and emit to Source Subject as new .next event
laura.score.onNext(85)

// 6. Add charlotte as an element onto Source Subject
student.onNext(charlotte)

// 7. Change charlotte & laura score and see new values emitted to subject
charlotte.score.onNext(20)
laura.score.onNext(100)
```

```
Subject1 :  next(80)
Subject1 :  next(30)
Subject1 :  next(90)
Subject1 :  next(20)
Subject1 :  next(100)
```

- **flapMapLatest**
  - a combination of two operators, map and switchLatest (in Combine operator)
    - switchLatest will **produce** values **from the most recent** observable, and **unsubscribe from the previous observable.**
  - **flatMapLatest** works just like flatMap
    - reach into an observable element to access its observable property
    - project it onto a new sequence for each element of the source observable.
    - Then, those elements are flattened down into a target observable that will provide elements to the subscriber.

- The difference for **flapMapLatest** is that it it will automatically ***switch*** **to the latest observable** and ***unsubscribe*** **from the the previous one.**



```swift

let john = Student(score: BehaviorSubject(value: 99))
let andy = Student(score: BehaviorSubject(value: 49))

let student2 = PublishSubject<Student>()

student2
    .flatMapLatest{
        $0.score
    }
    .subscribe{
        print("Subject1 : ",$0)
    }
    .disposed(by: disposeBag)

//1. add 'john' as element onto subject
student2.onNext(john)

//2. change 'john' score
john.score.onNext(21)

//3. add 'andy' as element onto subject => automatically UNSUBSCRIBE 'john'
student2.onNext(andy)

//4. change 'john' score <- this will not be emitted because subject has switch to
    'andy' and unsuscribe 'john'
john.score.onNext(100)

//5. change 'andy' score -> the change is emitted onto subject
andy.score.onNext(78)
```

'100' will not be emitted because subject has already unsubscribe 'john'

```
Subject1 :  next(99)
Subject1 :  next(21)
Subject1 :  next(49)
Subject1 :  next(78)
```

🟡 Observing events

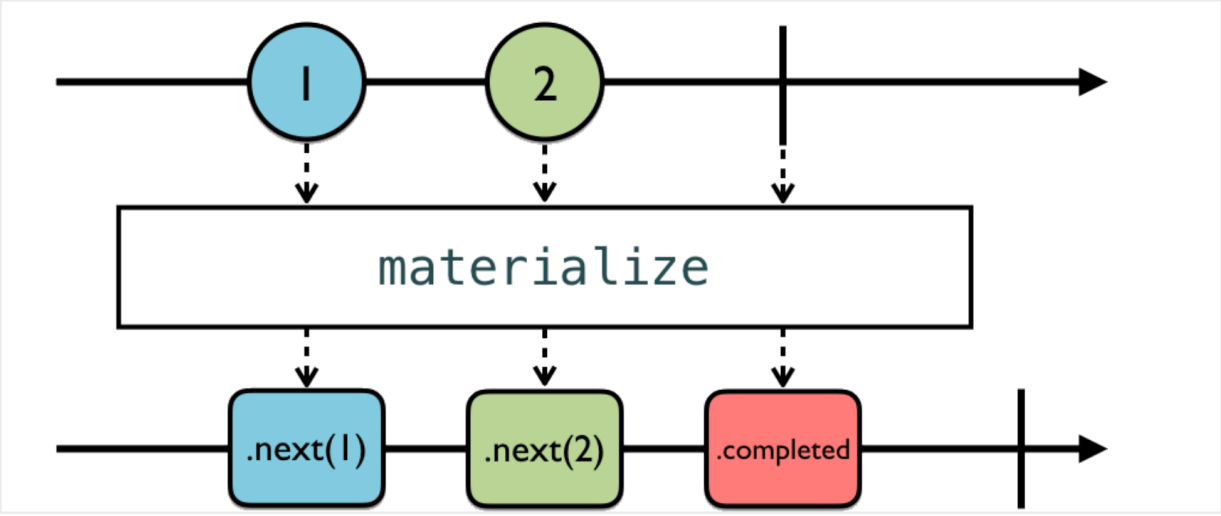
- Convert an observable into an observable of its events.
- Scenario: this is useful is when you do not have control over an observable that has observable properties, and you want to handle error events to avoid terminating outer sequences.

Ex: As if we have this code similar to above:

```swift
// 1. Create an error type.
enum MyError: Error {
  case anError
}

// 2. Create two instances of Student
let mimi = Student(score: BehaviorSubject(value: 80))
let ted = Student(score: BehaviorSubject(value: 100))

// 3. Create student behavior subject with the 1st student 'mimi' as its initial value.
let student3 = BehaviorSubject(value: mimi)

let studentScore = student3
    .flatMapLatest{
        $0.score
    }
//4. Create a studentScore observable using flatMapLatest to reach into the student observable and access its score observable property.
studentScore
    .subscribe{
    print("Subject 1: ",$0)
    }
    .disposed(by: disposeBag)

// 3. Add a score, error, and another score onto 'mimi'.
  mimi.score.onNext(85)
  mimi.score.onError(MyError.anError) ← when we throw error here, it cannot handle the error
  mimi.score.onNext(90)

// 4. Add the 2nd student 'ted' onto the student observable.
//      + Because you used flatMapLatest, this will switch to this new student and subscribe to his score.
  student.onNext(ted)
```

```
Subject 1:  next(80)
Subject 1:  next(85)
Subject 1:  error(anError)
```

- Materialize operator: you can wrap each event emitted by an observable *in* an observable.



we can modify the example above a bit:

```swift
// 1. Create an error type.
enum MyError: Error {
  case anError
}

// 2. Create two instances of Student
let mimi = Student(score: BehaviorSubject(value: 80))
let ted = Student(score: BehaviorSubject(value: 100))

// 3. Create student behavior subject with the 1st student 'mimi' as its initial value.
let student3 = BehaviorSubject(value: mimi)

let studentScore = student3
    .flatMapLatest{
        $0.score.materialize() // convert observable to any observable of its event
        ///**NOTE :
        ///Now, studentScore is  an Observable<Event<Int>>. And the subscription to it now emits EVENTS instead Observable<Int>
    }
//4. Create a studentScore observable using flatMapLatest to reach into the student observable and access its score observable property.
studentScore
    .subscribe{
    print("Subject 1: ",$0)
    }
    .disposed(by: disposeBag)

// 3. Add a score, error, and another score onto 'mimi'.
  mimi.score.onNext(85)
  mimi.score.onError(MyError.anError)
  mimi.score.onNext(90)

// 4. Add the 2nd student 'ted' onto the student observable.
//      + Because you used flatMapLatest, this will switch to this new student and subscribe to his score.
  student.onNext(ted)
```

```
Subject 1:  next(next(80))
Subject 1:  next(next(85))
Subject 1:  next(error(anError))
Subject1 :  next(100)
```

- **studentScore** is now an **Observable<Event<Int>>** instead Observable<Int>.
  - And the subscription to it now **emits events.**
- The error still causes the studentScore to terminate, but not the outer student observable, so when you switch to the new student, its score is successfully received and printed.

- Dematerialized: convert a materialized observable back into its original form.

```swift
let studentScore2 = student4
    .flatMapLatest{
        $0.score.materialize() // convert observable to any observable of its event
        ///**NOTE :
        ///Now, studentScore is  an Observable<Event<Int>>. And the subscription to it now emits EVENTS instead Observable<Int>
    }

studentScore2
    // 1. You print and filter out any errors.
    .filter{
        guard $0.error == nil else {
            print($0.error!)
            return false
        }
        return true
    }
    // 2. Use dematerialize to return the studentScore observable to its original form, emitting scores and stop
        events, not events of scores and stop events.
    /// NOTE: your student observable is protected by errors on its inner score observable.
    ///     + The error is printed and hayes's studentScore is terminated, so adding a new score onto her does nothing.
    ///     + But when you add charlotte onto the student subject, her score is printed.
    .dematerialize()  ← revert to original form
    .subscribe{
    print("Subject 2: ",$0)
    }
    .disposed(by: disposeBag)

// 3. Add a score, error, and another score onto 'mimi'.
    hayes.score.onNext(85)
    hayes.score.onError(MyError.anError)
    hayes.score.onNext(90)

// 4. Add the 2nd student 'ted' onto the student observable.
//      + Because you used flatMapLatest, this will switch to this new student and subscribe to his score.
  student.onNext(erik)

// 5. Change value of both erik and hayes
  hayes.score.onNext(15) //already unsubscribed so no value is printted out
  erik.score.onNext(88) // value is emitted and printed out
```

```
Subject 2:  next(80)
Subject 2:  next(85)
anError  ←
Subject1 :  next(100)
Subject1 :  next(88)
```