Sun*

Observable

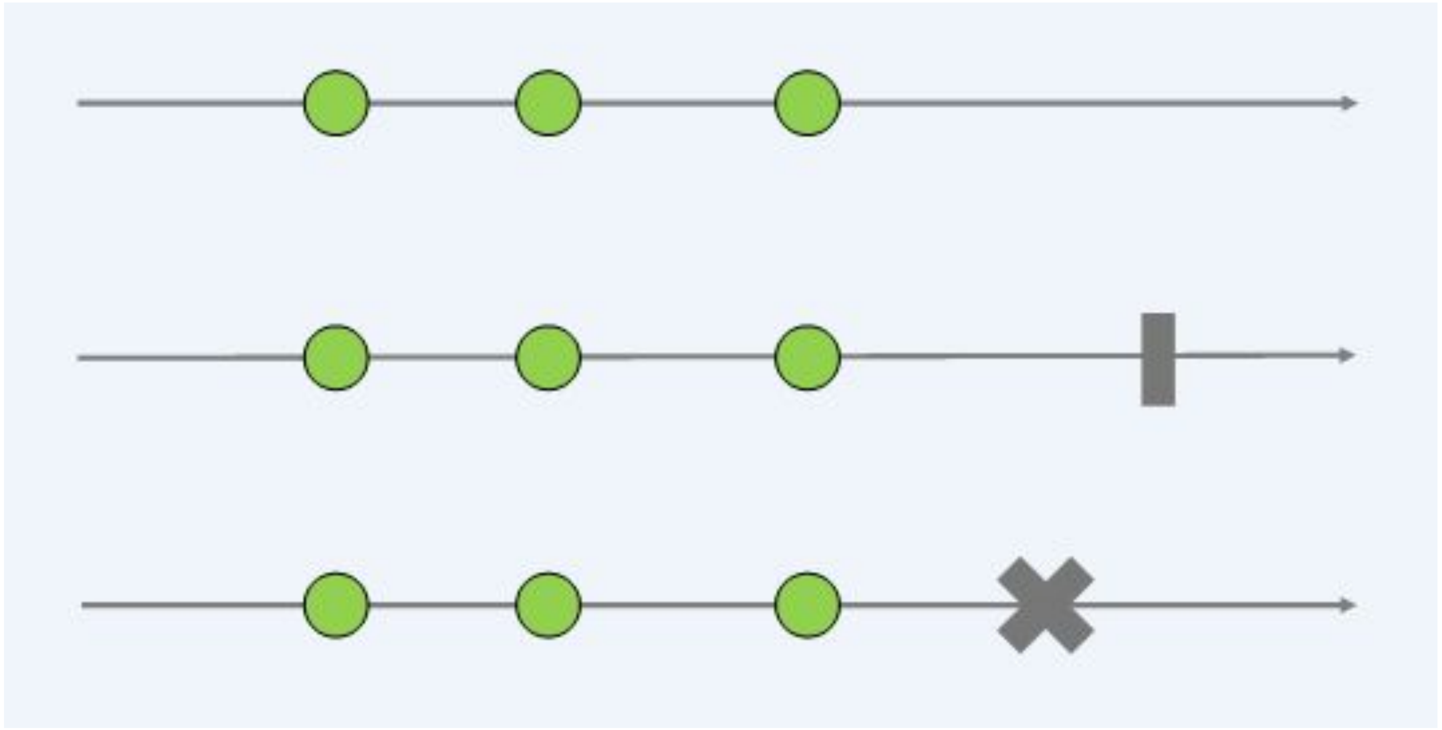
Outline

Sun*

1. What is an Observable?

- ❖ **Observables** are the heart of Rx. Every Observable sequence is just a sequence.
- ❖ The key advantage for an Observable vs Swift's Sequence is that it can also receive elements asynchronously.
- ❖ **Observables** produce events, the process of which is referred to as **emitting**, over a period of time.

2. Lifecycle of an observable



Sun*

2. Lifecycle of an observable (cont)

- An observable emits **next** events that contain elements.
- It can continue to do this until a **terminating event** is emitted. E.g. either an **error** event or a **completed** event.
- Once an observable is terminated, it can no longer emit events.

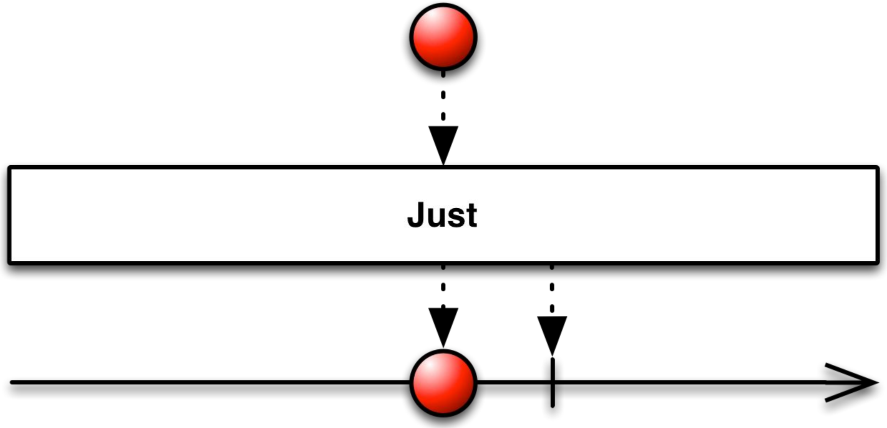
3. Creating Observables

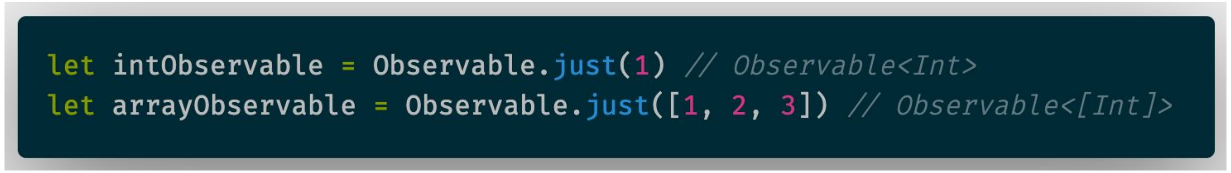
1. Observable.just()
2. Observable.of()
3. Observable.from()

Sun*

3.1. Observable.just()

❖ Create an Observable that emits a particular item



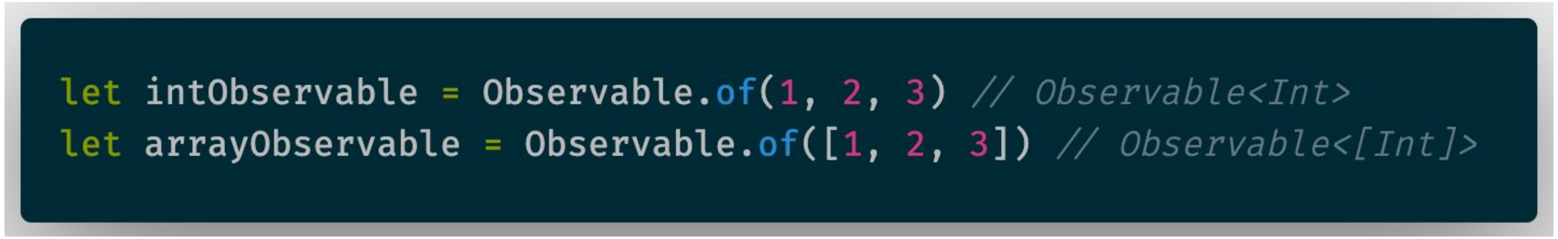


```
let intObservable = Observable.just(1) // Observable<Int>
let arrayObservable = Observable.just([1, 2, 3]) // Observable<[Int]>
```

Sun*

3.2. Observable.of()

- Creates a new Observable instance with a variable number of elements

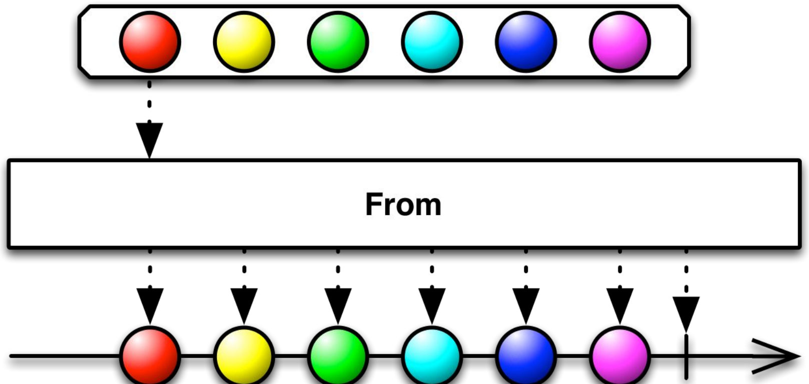


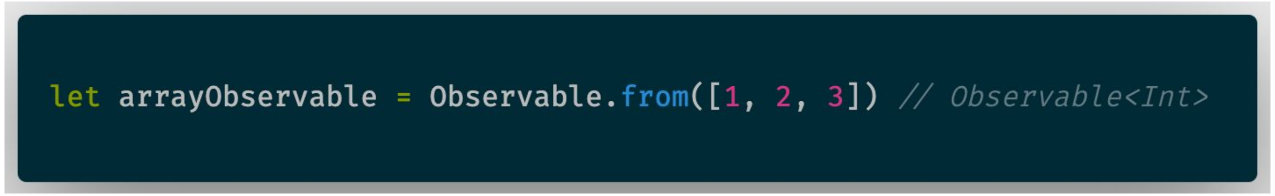
```
let intObservable = Observable.of(1, 2, 3) // Observable<Int>
let arrayObservable = Observable.of([1, 2, 3]) // Observable<[Int]>
```

Sun*

3.3. Observable.from()

❖ Converts an array to an observable sequence.





```
let arrayObservable = Observable.from([1, 2, 3]) // Observable<Int>
```

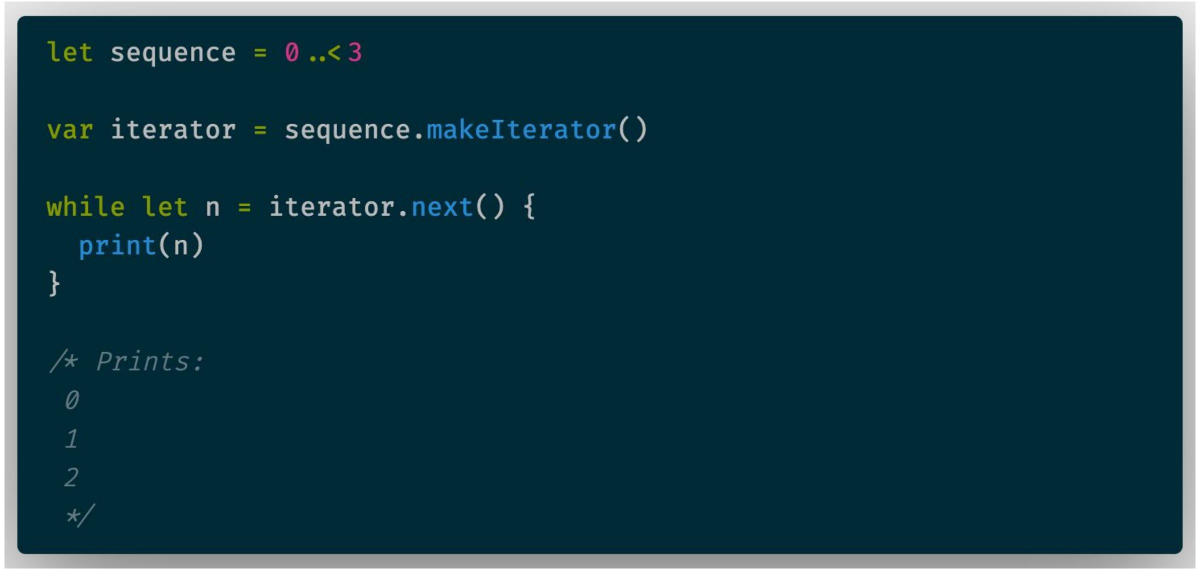
4. Subscribing to observables

- ❖ As an iOS developer, you may be familiar with NotificationCenter; it broadcasts notifications to observers, which are different than RxSwift Observables.
- ❖ Subscribing to an RxSwift observable is fairly similar; you call observing an observable subscribing to it. So instead of addObserver(), you use subscribe(). Unlike NotificationCenter, where developers typically use only its .default singleton instance, each observable in Rx is different.
- ❖ More importantly, an observable won’t send events, or perform any work, until it has a subscriber.

Sun*

4. Subscribing to observables (cont)

❖ An observable is really a sequence definition; subscribing to an observable is really more like calling next() on an Iterator in the Swift standard library.

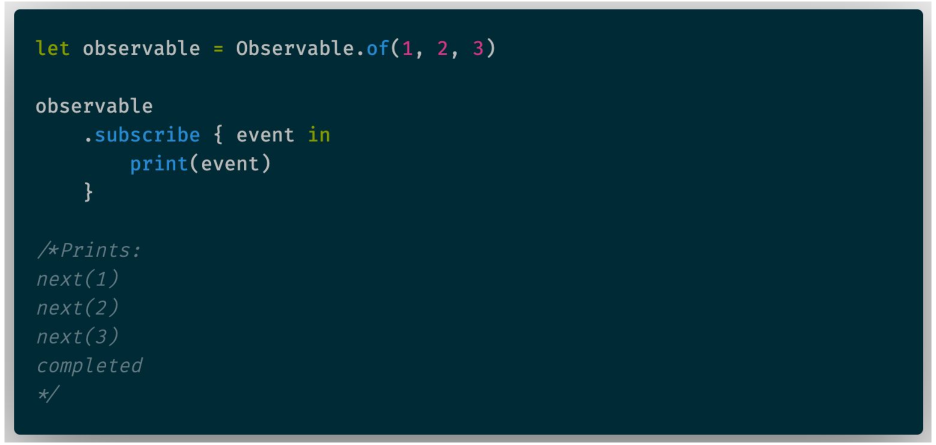


```
let sequence = 0 ..< 3

var iterator = sequence.makeIterator()

while let n = iterator.next() {
  print(n)
}

/* Prints:
 0
 1
 2
 */
```

Sun*

4. Subscribing to observables (cont)

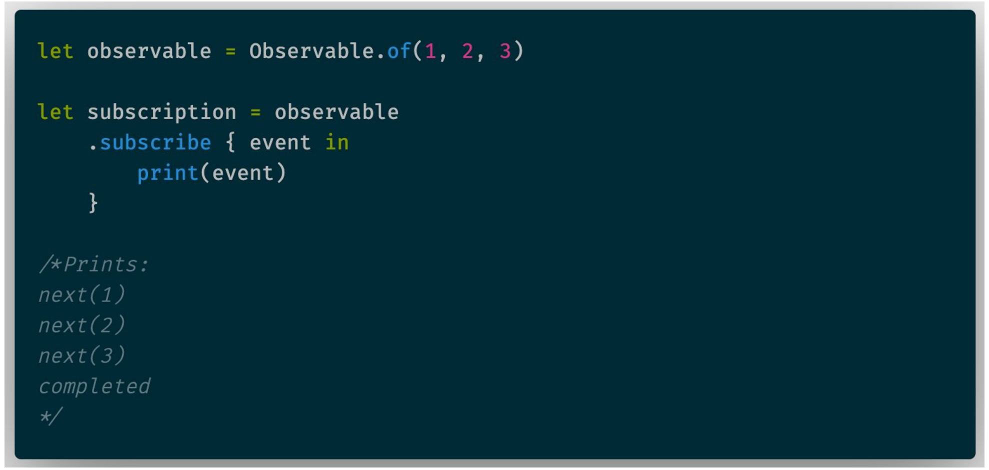
- Subscribing to observables is more streamlined than this, though. You can also add handlers for each event type an observable can emit. Recall that an observable emits .next, .error and .completed events. A .next event passes the element being emitted to the handler, and an .error event contains an error instance.



```
let observable = Observable.of(1, 2, 3)

observable
    .subscribe { event in
        print(event)
    }

/*Prints:
next(1)
next(2)
next(3)
completed
*/
```

5. Disposing and terminating

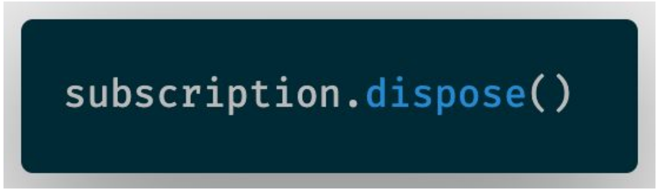
❖ An observable doesn’t do anything until it receives a subscription. It’s the subscription that triggers an observable's work, which emits new events, up until it emits an .error or .completed event and is terminated. You can manually cause an observable to terminate by canceling a subscription to it.



```
let observable = Observable.of(1, 2, 3)

let subscription = observable
    .subscribe { event in
        print(event)
    }

/*Prints:
next(1)
next(2)
next(3)
completed
*/
```

5. Disposing and terminating (cont)

❖ To explicitly cancel a subscription, call dispose() on it. After you cancel the subscription, or dispose of it, the observable in the current example will stop emitting events.

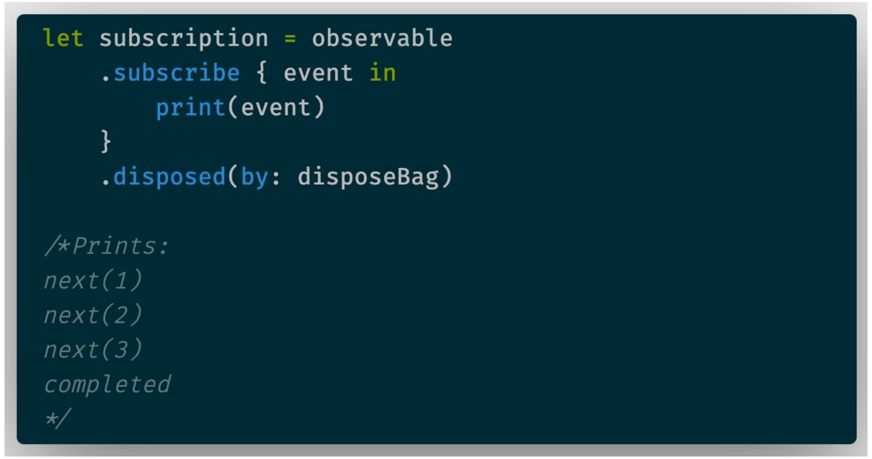


```
subscription.dispose()
```

Sun*

5. Disposing and terminating (cont)

❖ Managing each subscription individually would be tedious, so RxSwift includes a DisposeBag type. A dispose bag holds disposables — typically added using the .disposed(by:) method — and will call dispose() on each one when the dispose bag is about to be deallocated.



```
let subscription = observable
    .subscribe { event in
        print(event)
    }
    .disposed(by: disposeBag)

/*Prints:
next(1)
next(2)
next(3)
completed
*/
```

5. Disposing and terminating (cont)

❖ If you forget to add a subscription to a dispose bag, or manually call dispose on it when you’re done with the subscription, or in some other way cause the observable to terminate at some point, you will probably leak memory.

❖ Don’t worry if you forget; the Swift compiler should warn you about unused disposables.

6. Using Traits

1. Single
2. Completable
3. Maybe

Sun*

6. Traits

❖ Traits are observables with a narrower set of behaviors than regular observables. Their use is optional; you can use a regular observable anywhere you might use a trait instead.

❖ Their purpose is to provide a way to more clearly convey your intent to readers of your code or consumers of your API.

❖ The context implied by using a trait can help make your code more intuitive.

6.1. Single

- Single will emit either a .success(value) or .error event.
- .success(value) is actually a combination of the .next and .completed events.
- This is useful for one-time processes that will either succeed and yield a value or fail, such as downloading data or loading it from disk.

Sun*

6.2. Completable

❖ A Completable will only emit a .completed or .error event.

❖ Completable doesn’t emit any values.

❖ You could use a completable when you only care that an operation completed successfully or failed, such as a file write.

6.3. Maybe

- ❖ Maybe is a mashup of s Single and a Completable.

- ❖ It can either emit a .success(value), .completed or .error.

- ❖ If you need to implement an operation that could either succeed or fail, and optionally return a value on success, then Maybe is your ticket.

Sun*

Question & Answer?

Sun*