

From Zero to Rx Hero

Kinda

A world before Rx

- NotificationCenter
- Delegation
- GCD
- Clojures

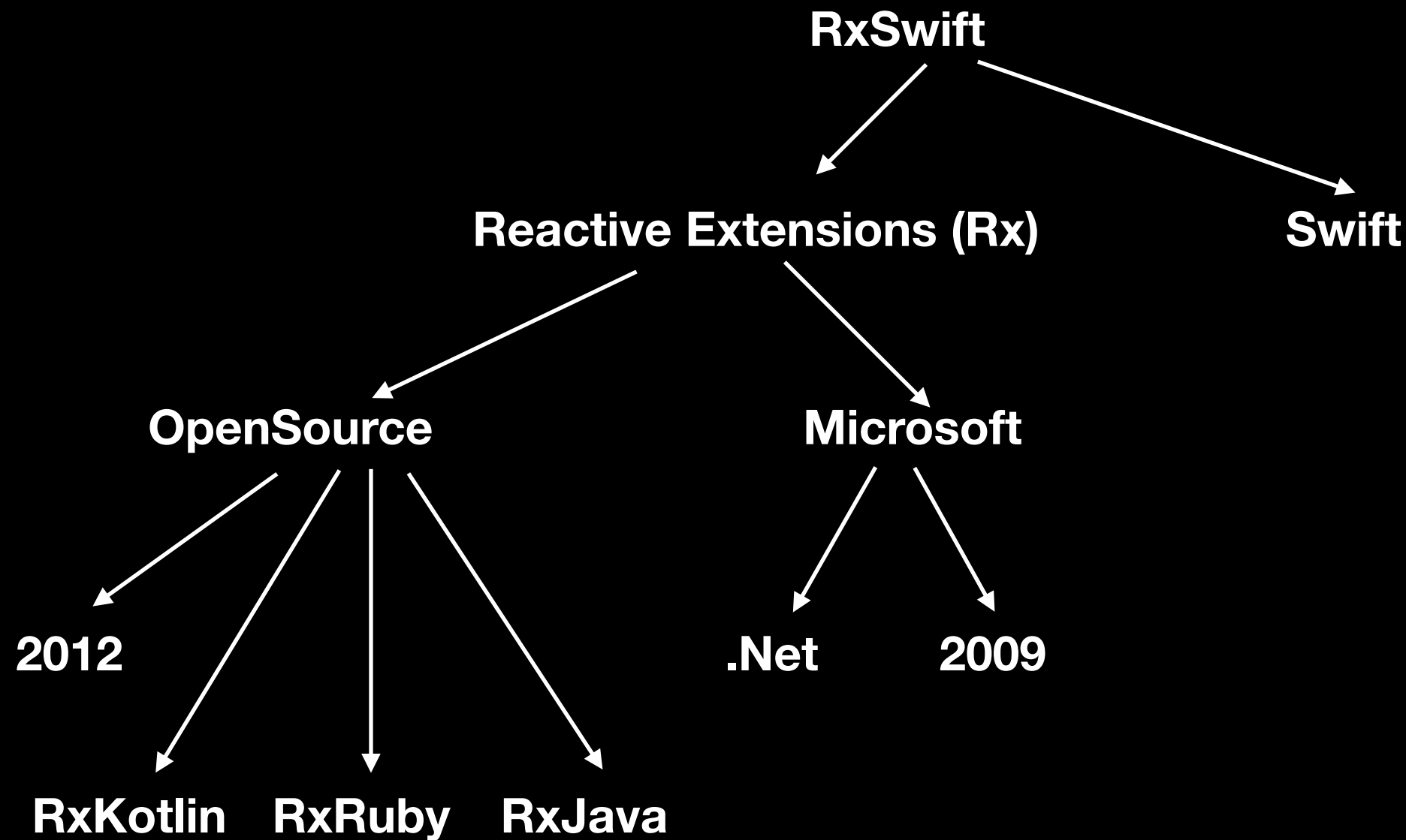
Daily Struggles

- Robust applications
- Concurrent tasks
 - * **Handle User Input**
 - * **Make network calls**
 - * **Update User location**
 - * **Update screen**
- Passing data between processes
- Asynchronously at the correct order

RxSwift to the rescue

- Simplify asynchronous programming
- React to new data
- Processed in isolation and in sequence

What is this Rx thing?



Reactive Extensions

ReactiveX is an API for composing **asynchronous** and **event-based** code by using **observable sequences** and **functional** style **operators**, allowing for parameterised execution via **schedulers**.

It uses the **Iterator** pattern, **Observer** pattern, **Functional** programming, and **Imperative** programming.

Synchronous & Asynchronous Programming

Synchronous

```
for number in array {  
    print(number)  
    array = [4, 5, 6]  
}  
print(array)
```

Asynchronous

```
var array = [1, 2, 3]  
var currentIndex = 0  
  
@IBAction func printNext(_ sender: Any) {  
    print(array[currentIndex])  
    if currentIndex != array.count - 1 {  
        currentIndex += 1  
    }  
}
```

Pitfalls of Async code

- Hard to know the order in which pieces of work are performed
- Shared mutable data (state

These are Rx's strongest features

Before we dive in

- State, and specifically, shared mutable state
- Imperative programming
- Side effects
- Declarative (functional) code
- Reactive Systems
 - **Responsive**
 - **Resilient**
 - **Elastic**
 - **Message driven**

The building blocks of Rx

- Observables

the ability to asynchronously produce a sequence of events that can “carry” an immutable snapshot of generic data of type T

- Operators

These are the methods of the Observable class that abstract discrete pieces of asynchronous work, they can be composed together to implement more complex logic. Because they are highly decoupled and composable, these methods are most often referred to as operators.

- Schedulers

are the Rx equivalent of dispatch queues — just on steroids and much easier to use.

Observable Types

- Finite Observable Sequences

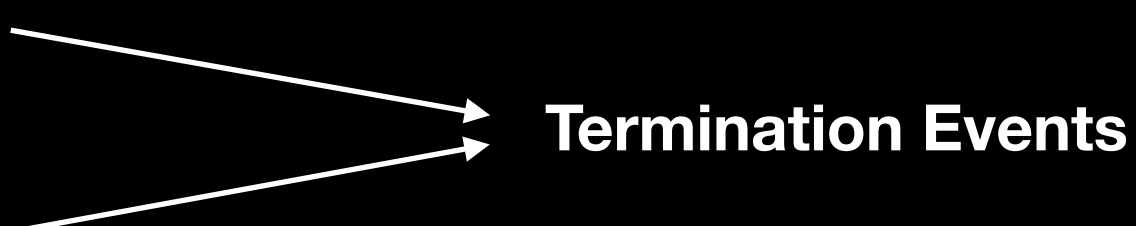
emit zero, one or more values, and, at a later point, either terminate successfully or terminate with an error.

- Infinite Observable Sequences

Unlike finite sequences, which are supposed to terminate either naturally or forcefully, infinite sequences stay "forever".

Observable Lifecycle

To start “listening” to an observable stream you “subscribe” to it.
This is done with an Observer.

- Next event
 - Completed event
 - Error Event
- 
- Termination Events
- The diagram consists of two white arrows on a black background. The top arrow originates from the text 'Completed event' and points to the right. The bottom arrow originates from the text 'Error Event' and also points to the right, slightly below the first arrow. Both arrows converge towards the text 'Termination Events'.

Traits

RxSwift

- Single
- Maybe
- Completable

RxCocoa

- Driver
- Signal

The reason for RxCocoa

RxCocoa is a standalone framework (usually shipped with RxSwift) that allows the usage of reactive programming with the OS specific components.

- RxSwift only contains Rx API methods
- RxCocoa adds support to platform specific features

Let be Subjects

Subjects are both Observers and Observables

- PublishSubject / PublishRelay
- BehaviorSubject / BehaviorRelay
- ReplaySubject
- AsyncSubject

Emits only the last “next” event received, only when receiving a “completed” event.

Relays can’t receive a “completed” or “error” event. Hence, they don’t terminate.

Dispose of your trash

Subscriptions live until they are terminated or disposed of.

They can be disposed of in 2 ways:

- Manually: (.dispose)
- Automatically: (adding to a disposeBag)

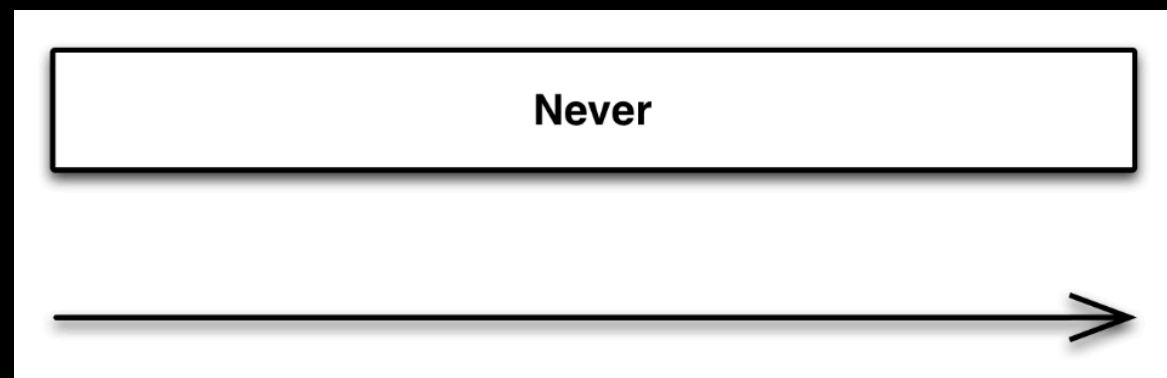
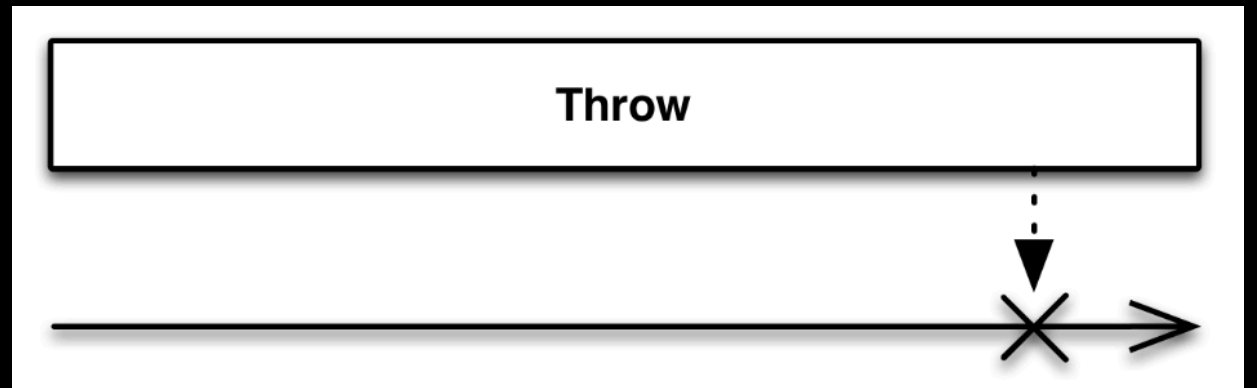
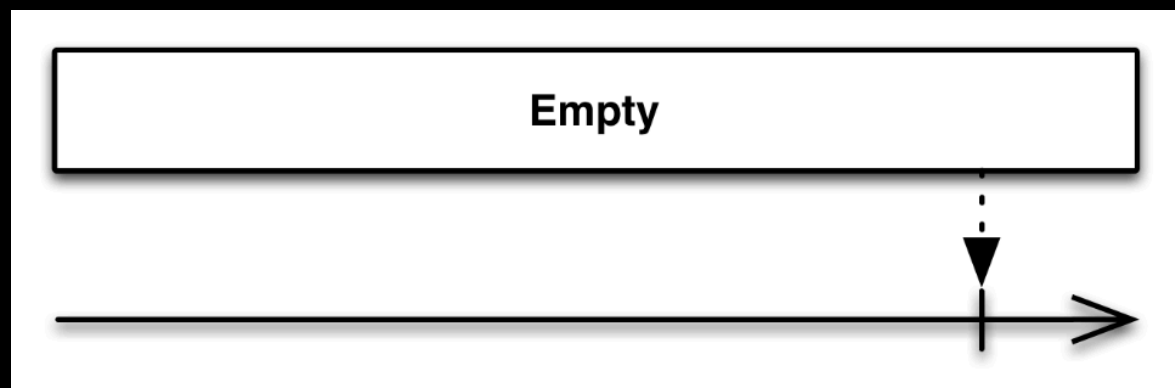
Operator Categories

- Creating
- Transforming
- Filtering
- Combining
- Error Handling
- Utility, and more

Creating Observables

Empty / Never / failWith

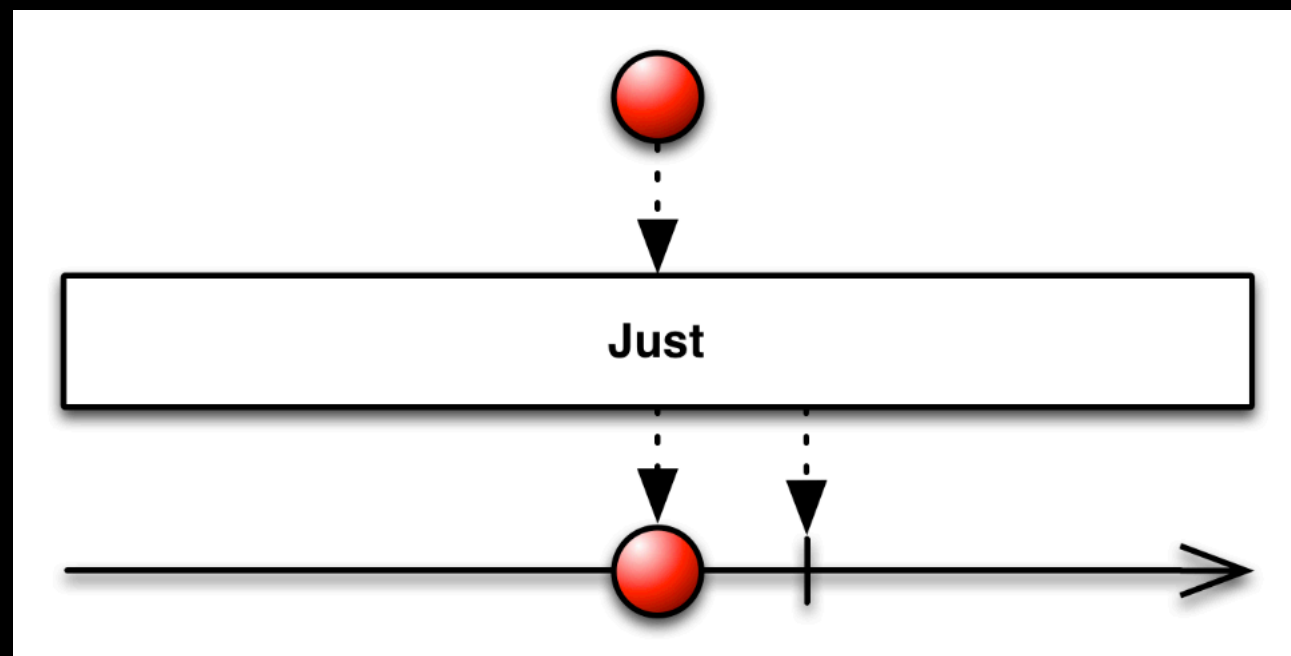
Creates an Observable from scratch by means of a function



Creating Observables

Just

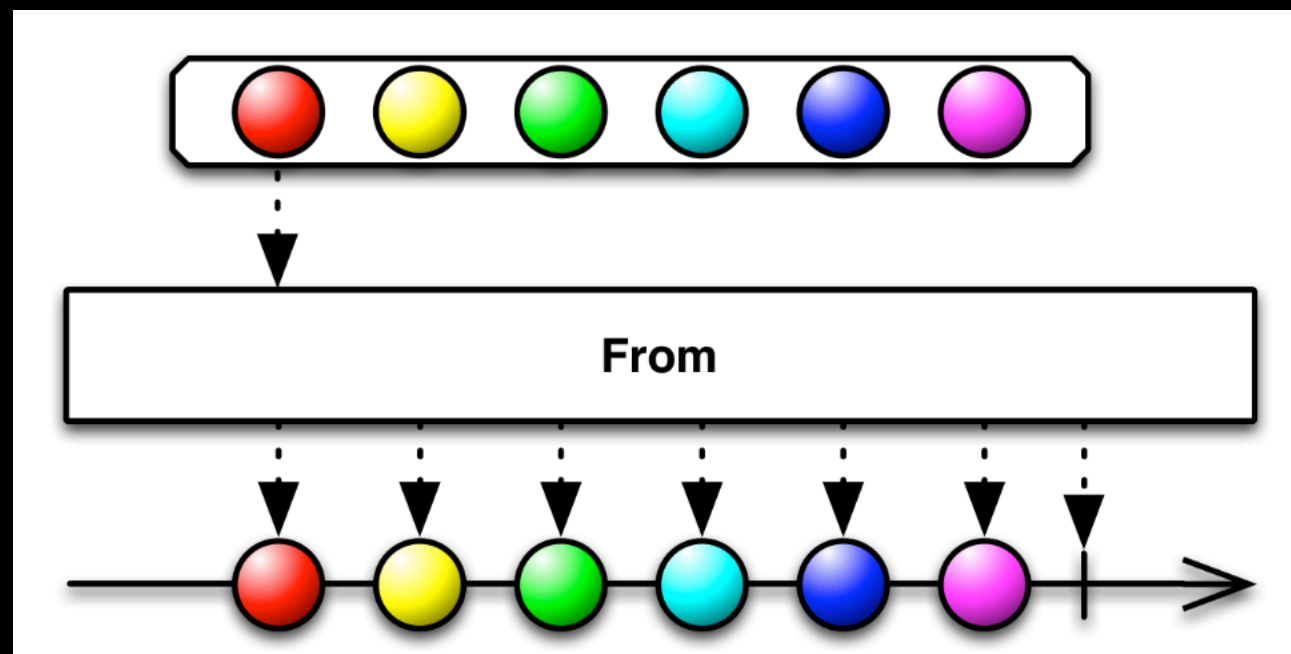
Creates an Observable that emits a particular item and completes



Creating Observables

From

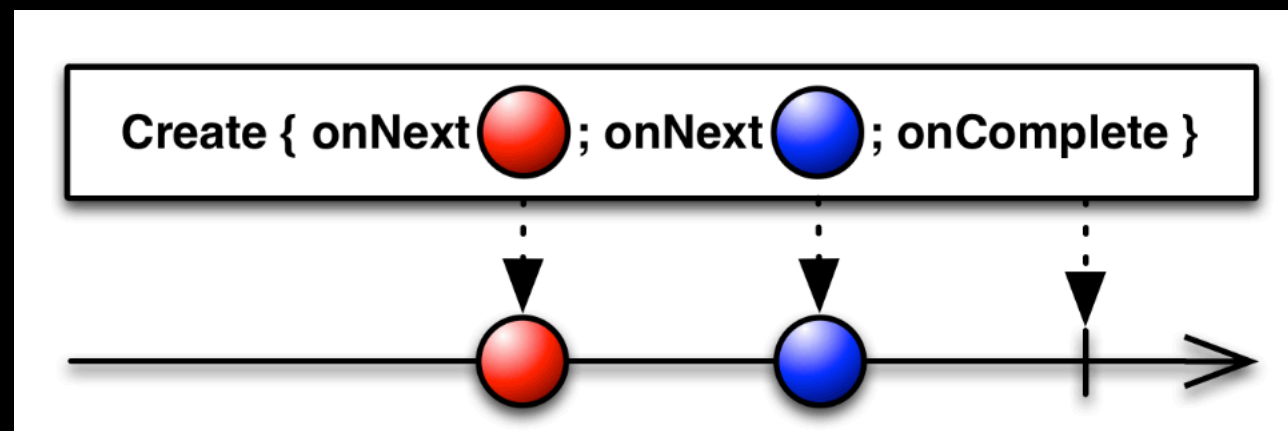
Converts various other objects and data types into Observables



Creating Observables

Create

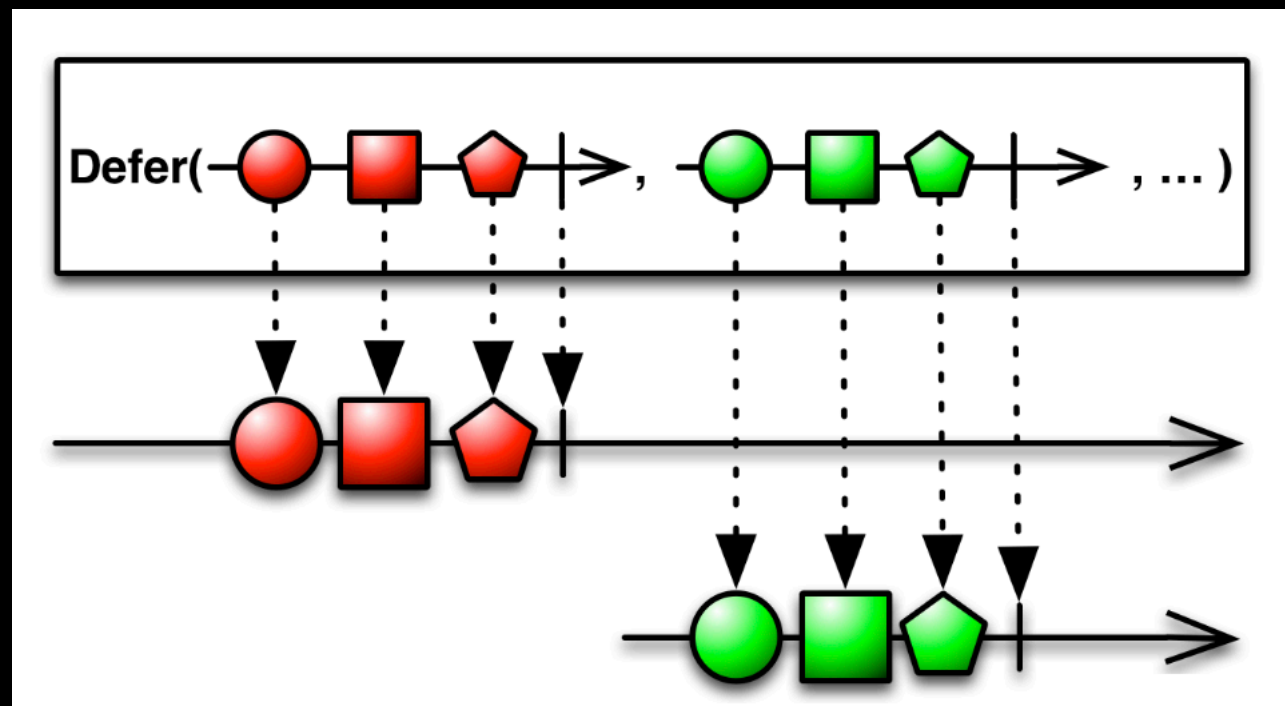
Creates an Observable from scratch by means of a function



Creating Observables

Defer

Do not create the Observable until the observer subscribes, and create a fresh Observable for each observer



Creating Observables

Interval

Creates an Observable that emits a sequence of integers spaced by a given time interval

Range

Creates an Observable that emits a particular range of sequential integers

Repeat

Creates an Observable that emits a particular item multiple times

Timer

Creates an Observable that emits a particular item after a given delay

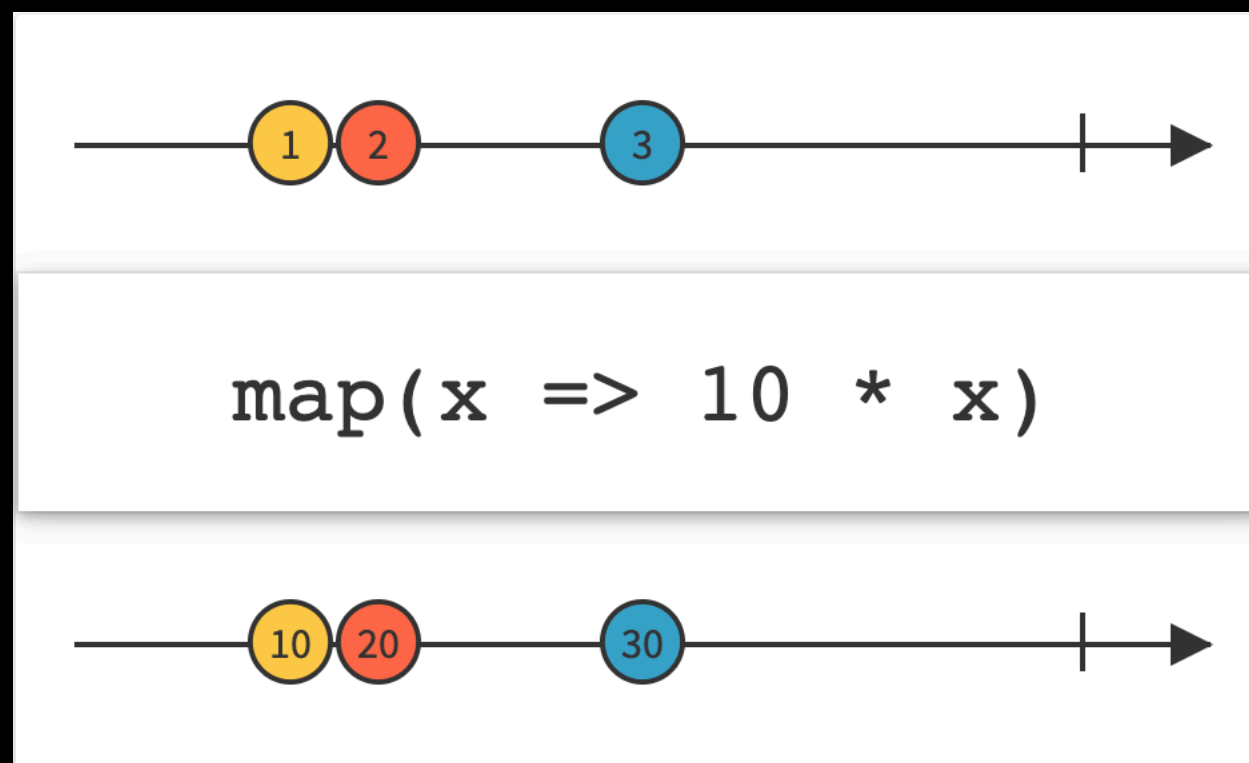
Hot & Cold Observables

- Cold observables only start emitting elements when they are subscribed to
- Hot observables emit events if they have an object subscribed to them or not

Transforming Observables

Map

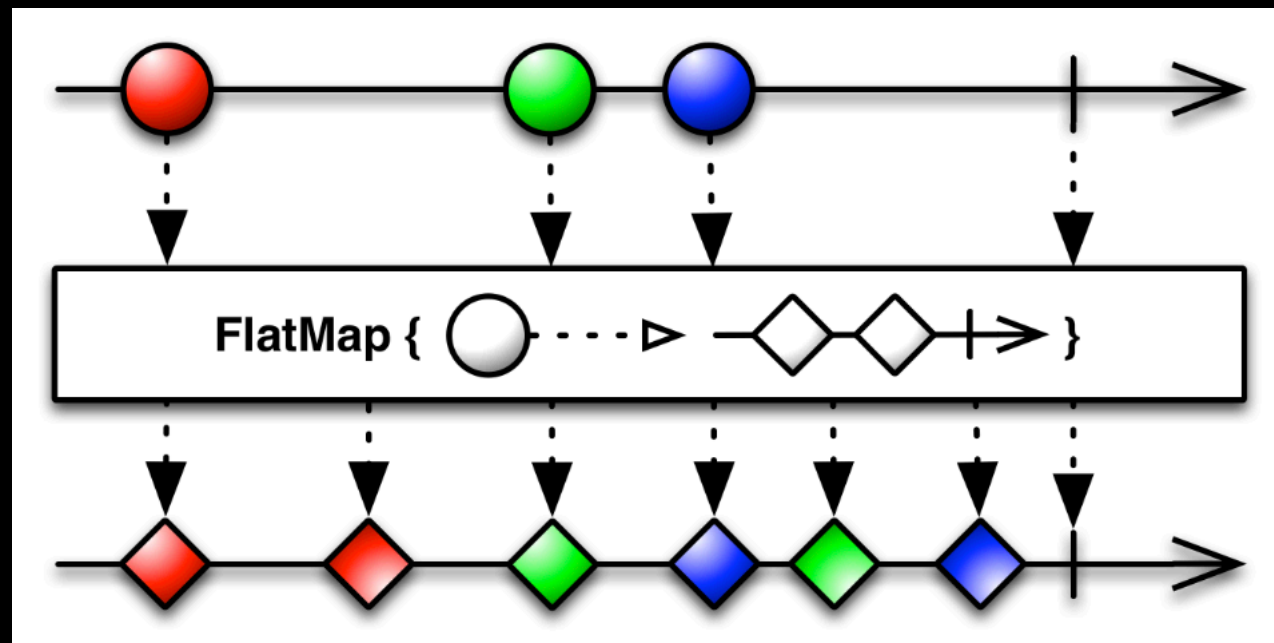
Transform the items emitted by an Observable by applying a function to each item



Transforming Observables

FlatMap

Transform the items emitted by an Observable into Observables, then flatten the emissions from those into a single Observable



Transforming Observables

Scan & Reduce

Scan applies a function to each item emitted by an Observable, sequentially, and emit each successive value

Reduce apply a function to each item emitted by an Observable, sequentially, and emit the final value



```
scan( (x, y) => x + y )
```



```
reduce( (x, y) => x + y )
```

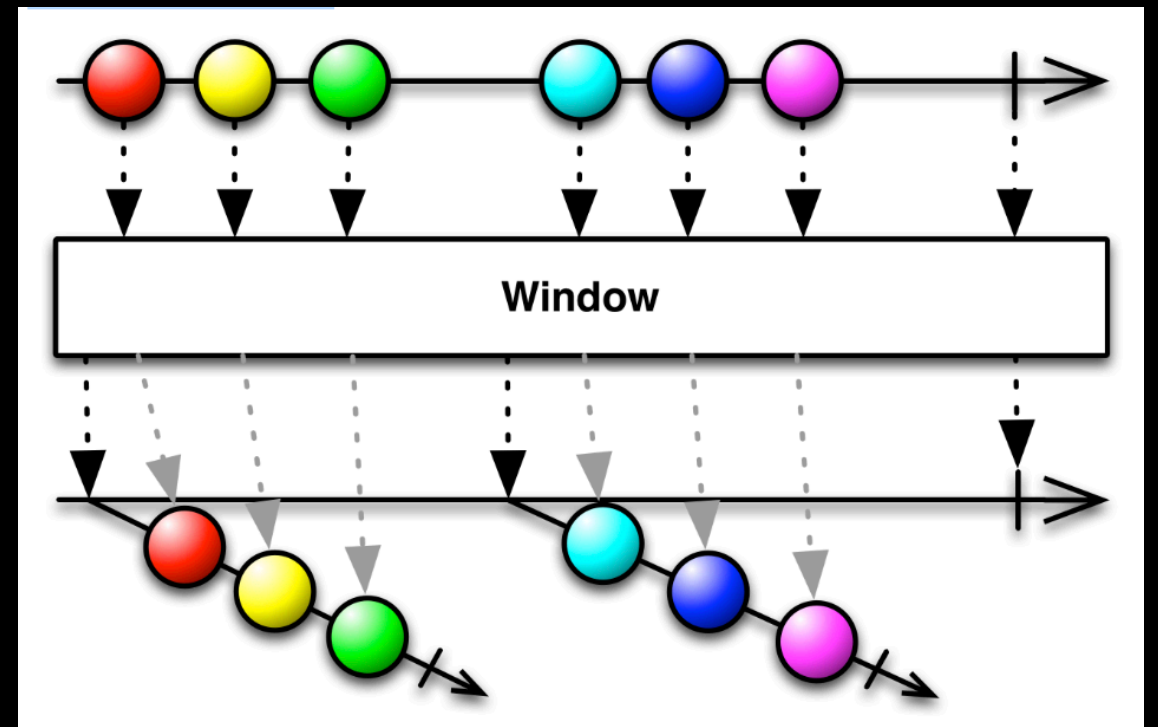
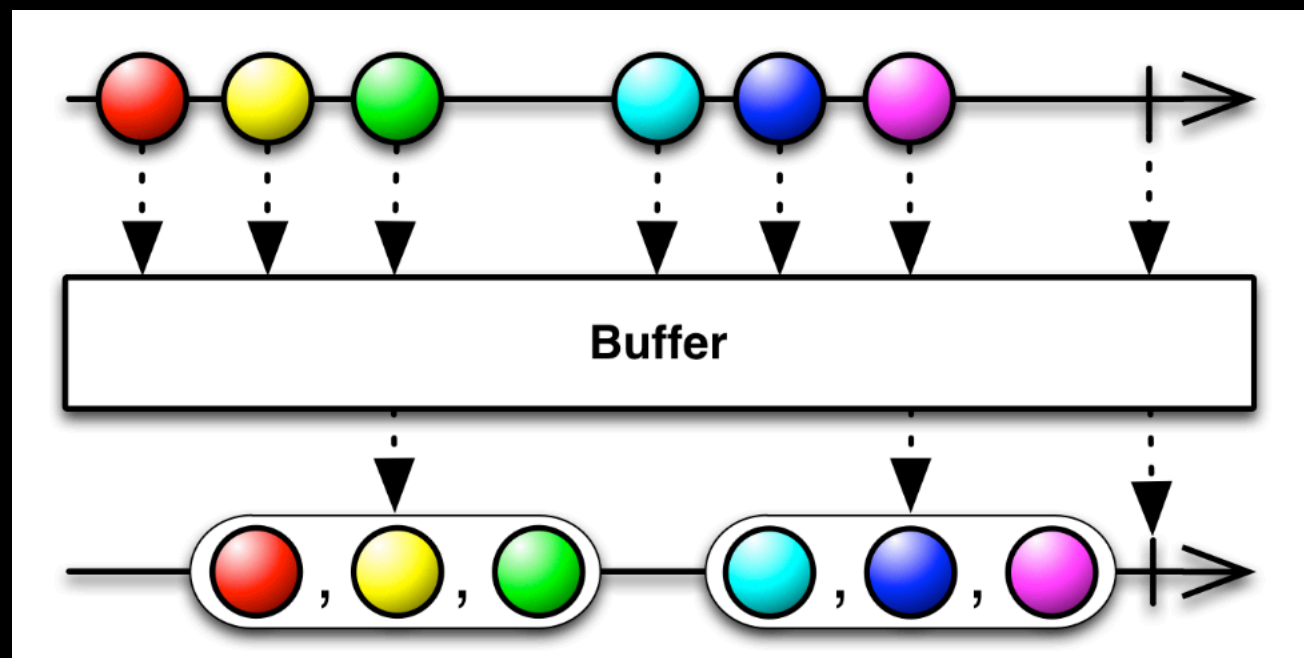


Transforming Observables

Buffer & Window

Buffer periodically gather items emitted by an Observable into bundles and emit these bundles rather than emitting the items one at a time

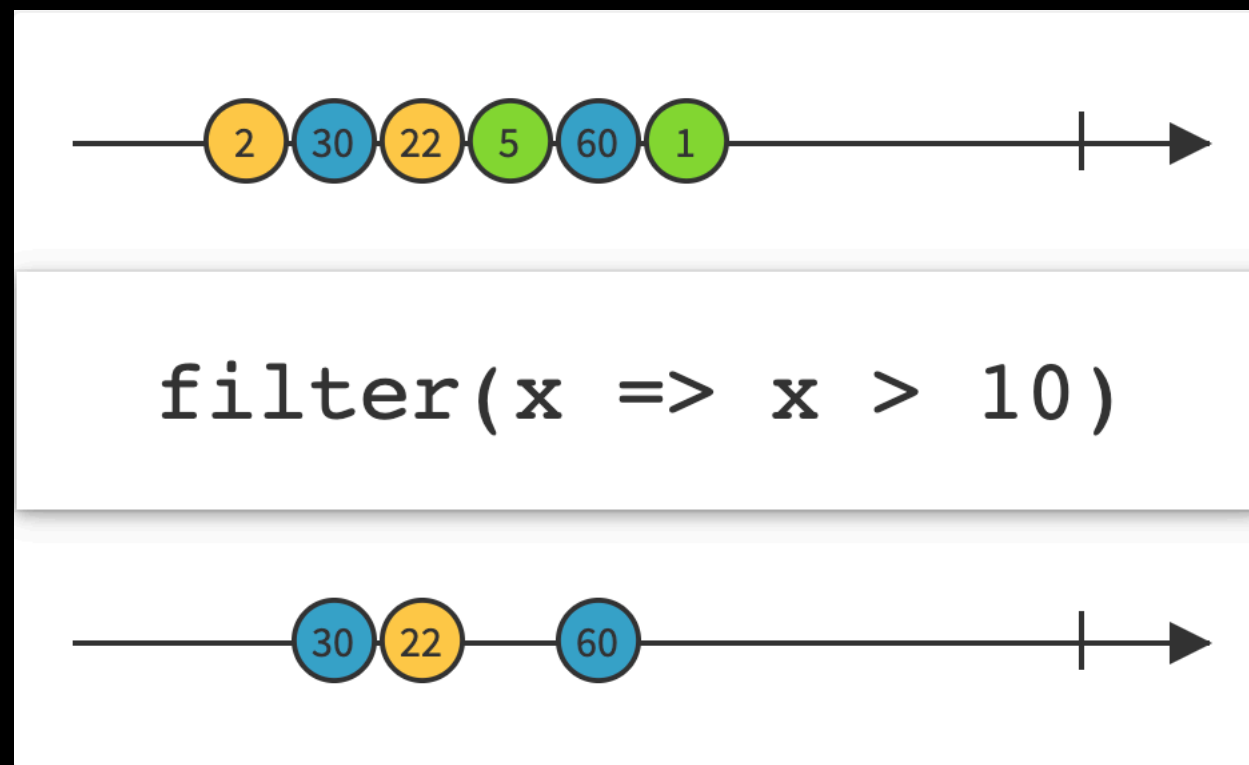
Window periodically subdivide items from an Observable into Observable windows and emit these windows rather than emitting the items one at a time



Filtering Observables

Filter

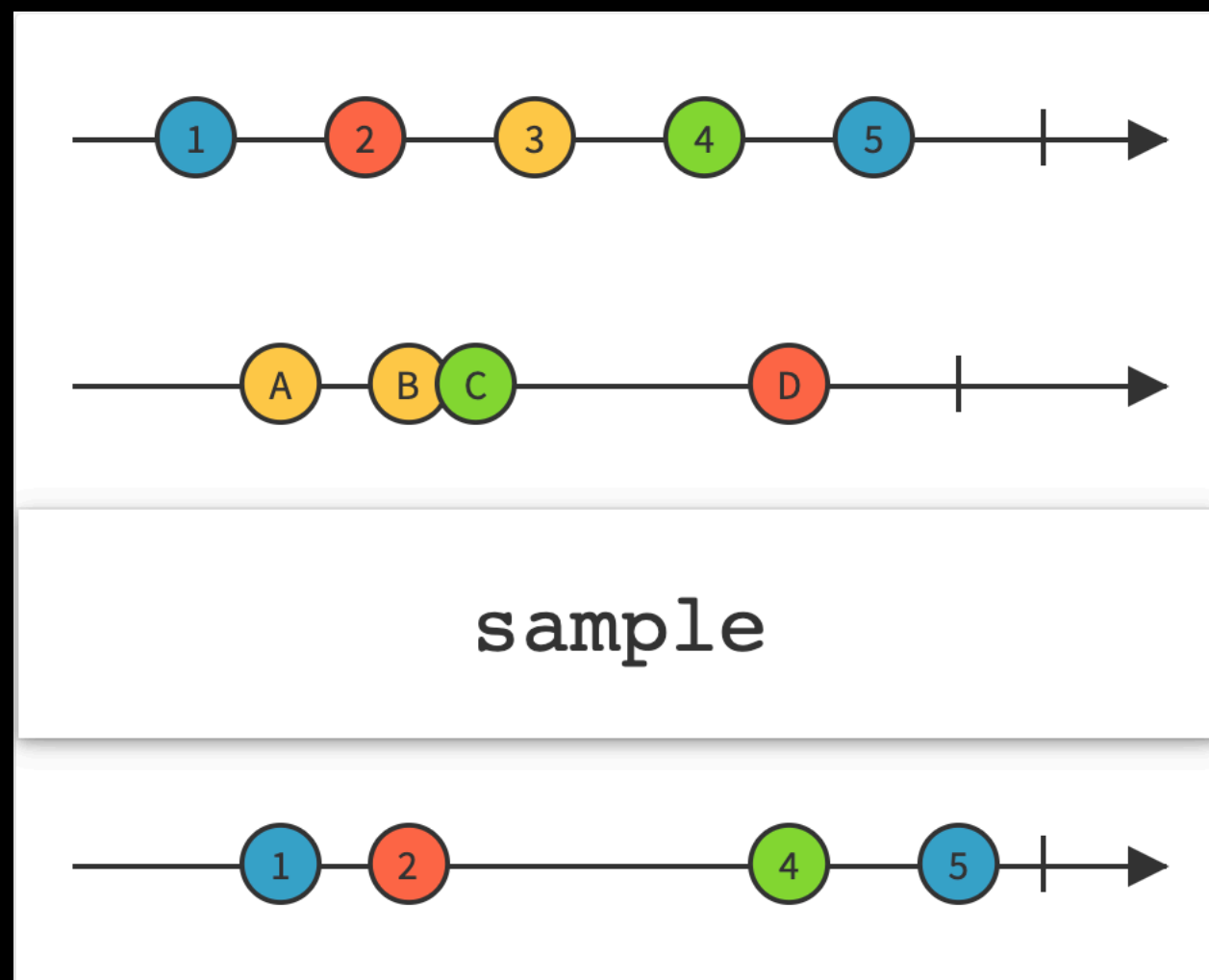
emit only those items from an Observable that pass a predicate test



Filtering Observables

Sample

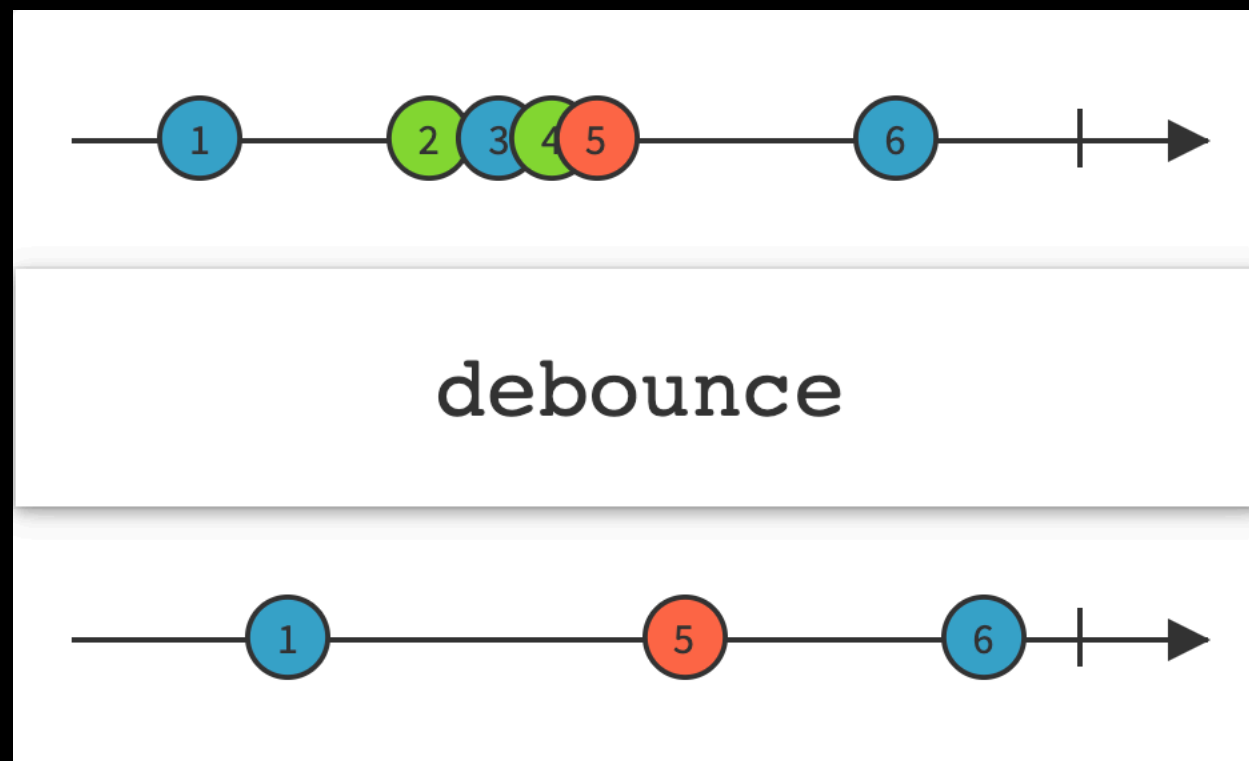
emit the most recent item emitted by an Observable within periodic time intervals



Filtering Observables

Debounce

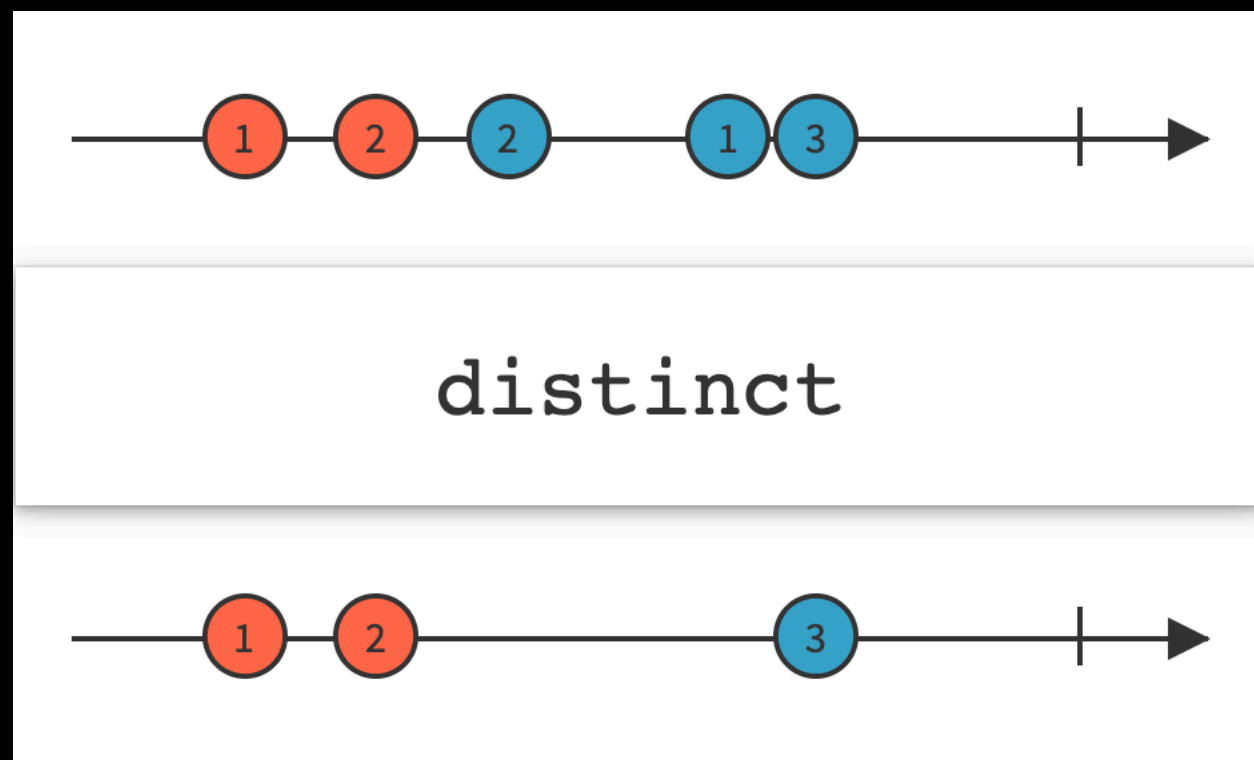
only emit an item from an Observable if a particular timespan has passed without it emitting another item



Filtering Observables

DistinctUntilChanged

suppress duplicate items emitted by an Observable



Filtering Observables

ElementAt & Skip



`elementAt(2)`



`skip(2)`



Filtering Observables

Take & TakeLast



`take(2)`



`takeLast(1)`



Combining Observables

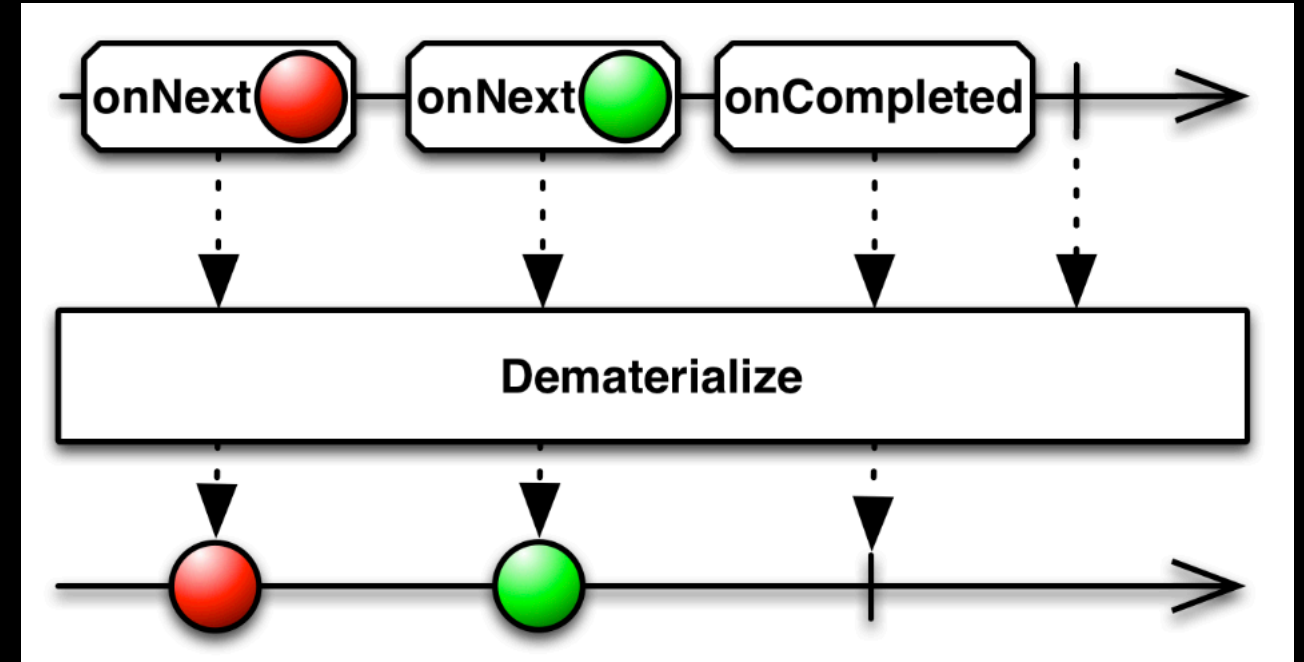
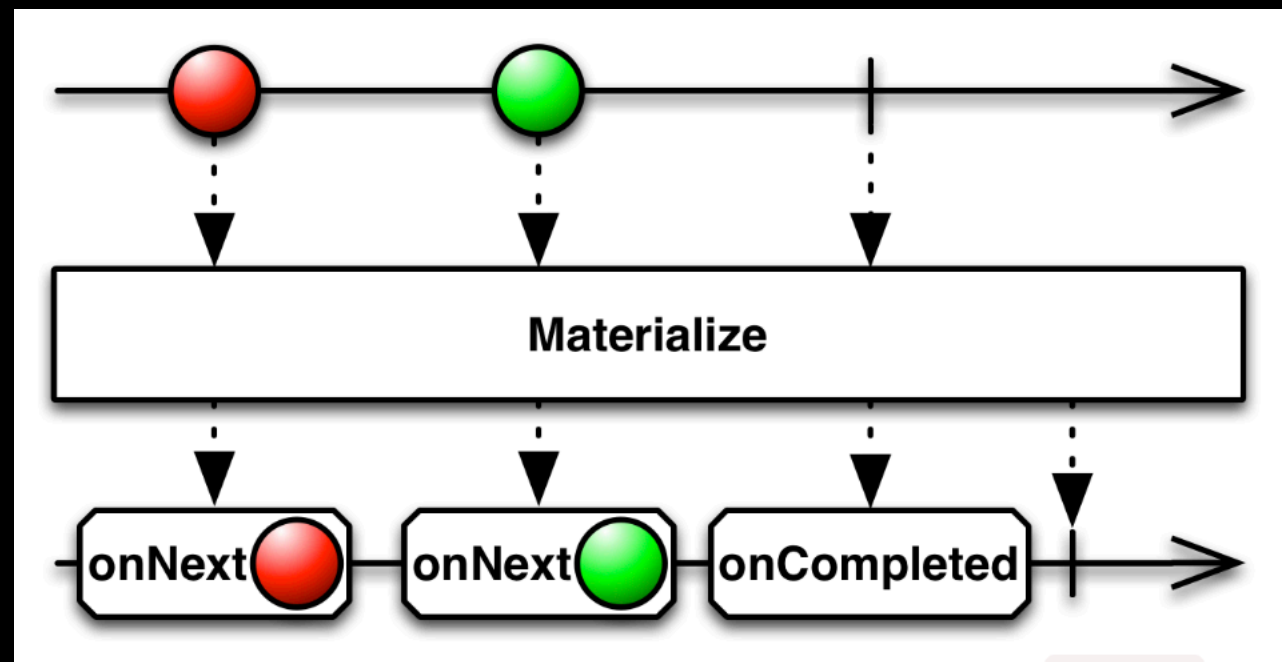
- **CombineLatest / WithLatestFrom**
- **Merge**
- **StartWith**
- **SwitchLatest**
- **Zip**

Error Handling Observables

- **Catch**
- **Retry**

Observable Utility Operators

Materialize & Dematerialize



Observable Utility Operators

- **DelaySubscription**
- **Do**
- **ObserveOn**
- **SubscribeOn**
- **Timeout**
- **Using**

Schedulers

- **Change where a piece of work is performed**
- **By default subscriptions are created on the current thread**
- **Where code is executed can be altered by using `ObserveOn` and `SubscribeOn` Operators**
- **`MainScheduler` is one of the schedulers available**

Resources

- <https://www.reactivex.io/>
- <http://slack.rxswift.org>
- <http://community.rxswift.org>