# RxSwift and RxCocoa

In software engineering, **dependency injection** is a design pattern in which an object or function receives other objects or functions that it depends on. A form of inversion of control, dependency injection aims to separate the concerns of constructing objects and using them, leading to loosely coupled programs. The pattern ensures that an object or function which wants to use a given service should not have to know how to construct those services.

## Swinject

Swinject is a lightweight dependency injection framework for Swift apps. It allows you to split your app into loosely-coupled components, which can then be maintained and tested more easily

## How it works

The Swinject is pretty straightforward in terms of using it. First you have to register your dependencies then you just call resolve to access them.

```
import Swinject

protocol Animal {
  var name: String { get }
}

struct Cat: Animal {
  let name: String
```

```swift
}

struct PetOwner {
  var pet = Injector.container.resolve(Animal.self)

  func play() {
    print("I'm playing with \(pet?.name ?? "some pet").")
  }
}

struct Injector {
  static let container = Container()

  static func registerDependencies() {
    container.register(Animal.self) { _ in
      Cat(name: "Persan")
    }
  }
}

// MARK: - Usage
Injector.registerDependencies()
let petOwner = PetOwner()
petOwner.play() // prints:  I'm playing with Persan
```

In the above example you can see how the pet gets injected in the PetOwner, which has no whereabouts on how the pet property is constructed. Also one important note first you need to register the dependencies usually is recommended to be done in the appDidFinishLaunching to be sure all dependencies are properly set before any use.

## RxSwift

RxSwift is a library for composing asynchronous and event-based code by using observable sequences and functional style operators, allowing for parameterized execution via schedulers. RxSwift in its essence simplifies developing asynchronous programs by allowing code to react to new data and process it in a sequential and isolated manner.

Asynchronous programming is a means of parallel programming in which a unit of work runs separately from the main application thread and notifies the calling thread of its completion, failure or progress.
The main benefits one can gain from using asynchronous programming are improved application performance and responsiveness.
To explain asynchronous programming in a simple way, let’s give an example from an iOS app. In the iOS app, at any moment, we might be doing any of the following things;

- Reacting to button taps
- Animating the UIView transitions
- Downloading bits of data from remote source
- Playing audio or video
- Saving bits of data to disk

Let’s say you have a block of code inside updateUI function. You only want to execute it once the value of isLiked has changed.

**Without RxSwift(**Imperative Programming**):**

```
var isLiked: Bool = false {
    didSet {
        if isLiked != oldValue {
            updateUI()
        }
    }
}
```

**With RxSwift:**

```
let isLiked = Variable(false)isLiked.asObservable()
    .distinctUntilChanged()
    .subscribe(onNext: {
        updateUI()
})
```

Let’s say you don’t want to respond if the value is updating too fast. For example, you don’t want to take action if the value of ***isLiked*** has been updated more than once within one second.

**Without RxSwift:**

```
var likedCount = 0
var isLikedEnabled = true
var isLiked: Bool = false {
    didSet {
        if isLiked != oldValue && likedCount < 5 &&
isLikedEnabled {
            updateUI()
            likedCount += 1
            isLikedEnabled = false
Timer.scheduledTimer(withTimeInterval: 1000, repeats:
false) { _ in                isLikedEnabled = true
            }
        }
    }
}
```

**With RxSwift:**

```
let isLiked = Variable(false)isLiked.asObservable()
    .debounce(1)
    .distinctUntilChanged()
    .take(5)
    .subscribe(onNext: {
        updateUI()
})
```

## RxCocoa

**RxCocoa** library allows us to use Cocoa APIs used in iOS and OS X with reactive technics.

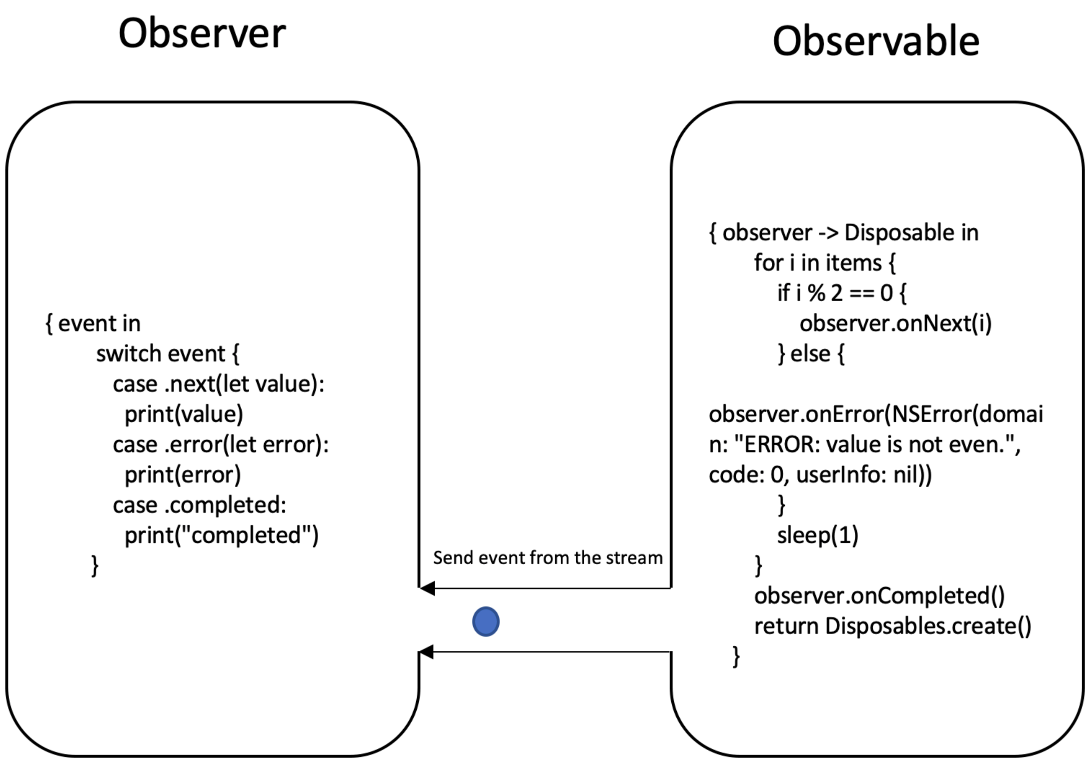
### *Observables and Observers*

One of the concepts you've got to know in this article is *Observable* and the other is *Observer.*

**Observable** means the constructs that bring out the changes.

**Observer** constructs are the ones that subscribe to Observable constructs and are informed when there's a change.

## Example of Observable and Observer.



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

```
{ observer -> Disposable in
      for i in items {
         if i % 2 == 0 {
            observer.onNext(i)
         } else {

observer.onError(NSError(domai
n: "ERROR: value is not even.",
code: 0, userInfo: nil))
         }
         sleep(1)
      }
      observer.onCompleted()
      return Disposables.create()
   }
```

## Observer

```
 checkIsEvenNumberObservable(items: [2,4,6,7,10])
 .subscribe{ event in
        switch event {
           case .next(let value):
             print(value)
           case .error(let error):
             print(error)
           case .completed:
             print("completed")
       }
}.disposed(by: bag)
```

You know now, Observer subscribes Observable.

**Closure after .subscribe method is Observer.**

**By forking switch cases (next, error, completed), Observer can implement actions for each cases.**

print is action here.

## Observable

We can see Observable now. You also can see **observer parameter** in Observable.

**Observer is closure after .subscribe method from previous code.**

```
func checkIsEvenNumberObservable(items: [Int]) ->
Observable<Int> {
    return Observable<Int>.create{ observer ->
Disposable in
        for i in items {
            if i % 2 == 0 {
                observer.onNext(i)
            } else {
                 observer.onError(NSError(domain: "ERROR:
value is not even.", code: 0, userInfo: nil))
            }
            sleep(1)
        }
        observer.onCompleted()
        return Disposables.create()
    }
```

From the above explaination, we can predict each .onNext(i), .onCompleted(), .onCompleted() sent to observer as event.

Last thing what we should know is just each behavior protocol for sending event. There are **onNext, onError, and onComplete**

- **onNext**

onNext is stream which emits one or sequences. Because Observer still subscribes Observable.

Observer can do some action depend on events from Observable.

- **onError**

When Observerble gets error, it emits error and makes Observerble stream stopped.

It is important to know you that stopped emittion by error is not same with completion.

Stream is not completed, and just stopped.

- **onComplete**

Observable emits event which means **completion of stream**, it is also same meaning that there are **no more next event to emit** from Observable.

***DisposeBag***
RxSwift and RxCocoa contain a tool named **DisposeBag** which helps ARC and memory management. We can think DisposeBag as a virtual bag that carries Observer objects. We can use DisposeBag tool to properly dispose of Observers when the parent

objects, which we define Observers with, are deallocated.