RXSwift - Observable

Ref Link:
https://www.raywenderlich.com/books/rxswift-reactive-programming-with-swift/v4.0/chapters/2-observables

https://fxstudio.dev/rxswift-observables/

- **Definition**
  - An *observable sequence* can emit "**things**" known as events. It can emit 0 or more events
- Observable life cycle :
  + Value : value or collection of values that is emitted
  + Error
  + Complete

    - When a value or collection of values (called *elements*) is added to or put onto a sequence, it will send **a next event** *containing that value or collection to its observers* - **emitting process.**
      - *If there's an error, a sequence can* ***emit an error event*** *containing an error type; this will terminate the sequence.*
      - *A sequence can also terminate normally, it'll emit a* ***completed event****.*

    - To stop observing a sequence, you can cancel the subscription by calling **dispose()**
      - OR add subscriptions to an instance of **DisposeBag**, which will take care of properly canceling subscriptions on **deinit()**.

- **To create an observable, we use operators : just, of , from**
  - **just:**
    - Create an observable sequence with **ONLY 1 element.**
      - The observable sequence emits ONLY 1 value, and then terminate

- Have to declare its value type

```
let observable1 = Observable<Int>.just(iOS)
```

- **of:**
  - No need to declare its value type. It will automatically identify the value type of the data inside of(...)
  - **of** operator has a **variadic** parameter

Declaration

```
static func of(_ elements: Int..., scheduler:
ImmediateSchedulerType = CurrentThreadScheduler.instance) ->
Observable<Int>
```

  - Ex1: observable 2 's value type : Observable<**Int**> : Single int values will be emitted one by one

```
let observable2 = Observable.of(iOS, android, flutter)
```

  - Ex2: observable 3 's value type Observable<**[Int]**>. Taking an array as its single element; each emitted value is an array of intergers:

```
let observable3 = Observable.of([iOS, android, flutter])
```

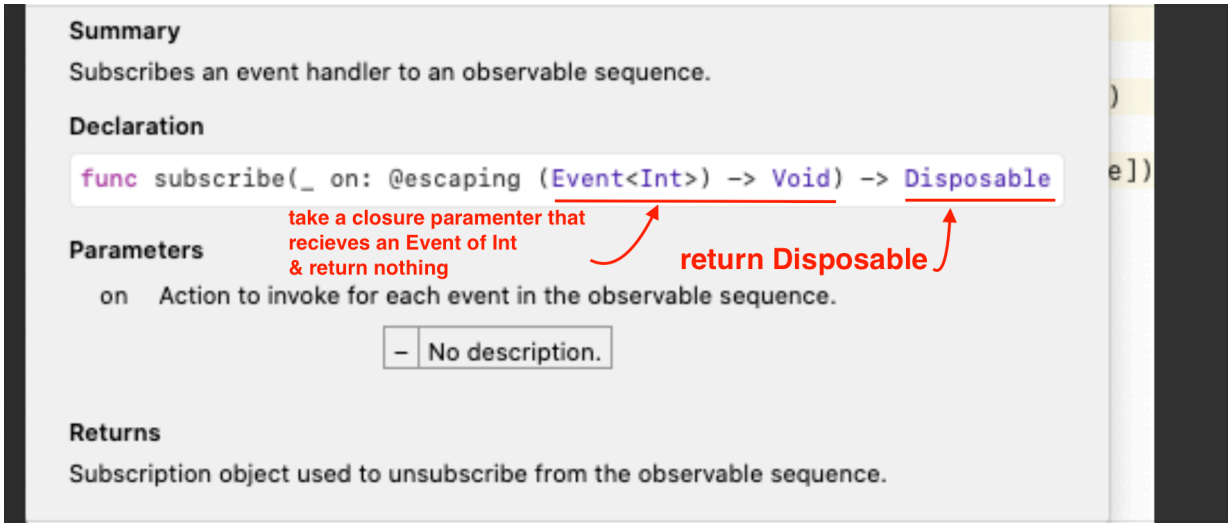
- **from**
  - Creates an observable of individual elements from an array of typed elements.
  - ONLY take array type
  - EX: observable4 is an Observable of Int, not [Int]

```
let observable4 = Observable.from([one, two, three])
```

just	of	from
an observable sequence with **ONLY 1 element**	an observable sequence with **single elements will be emitted one by one**	observable of individual elements from an array of typed elements.

	if we put an array as parameter, whole array will be considered as 1 single element	

To subscribe an observable :



We do:

```
let one = 1
let two = 2
let three = 3

let observable = Observable.of(one, two, three)
```

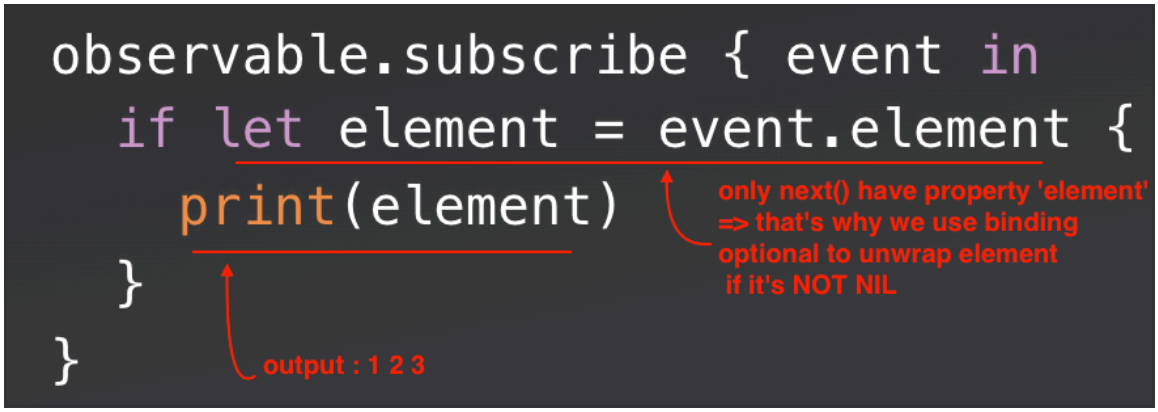
```
observable.subscribe { event in
  print(event)
}
```

This subscription will print out each event emitted by observable:

```
next(1)
next(2)
next(3)
completed
```

- There's a subscribe operator for each type of event an observable emits: **next**, **error** and **completed**.
  - The observable emits a next event for **each element**, then emits a completed event and is terminated (if does me

- To access the elements directly, we can do:



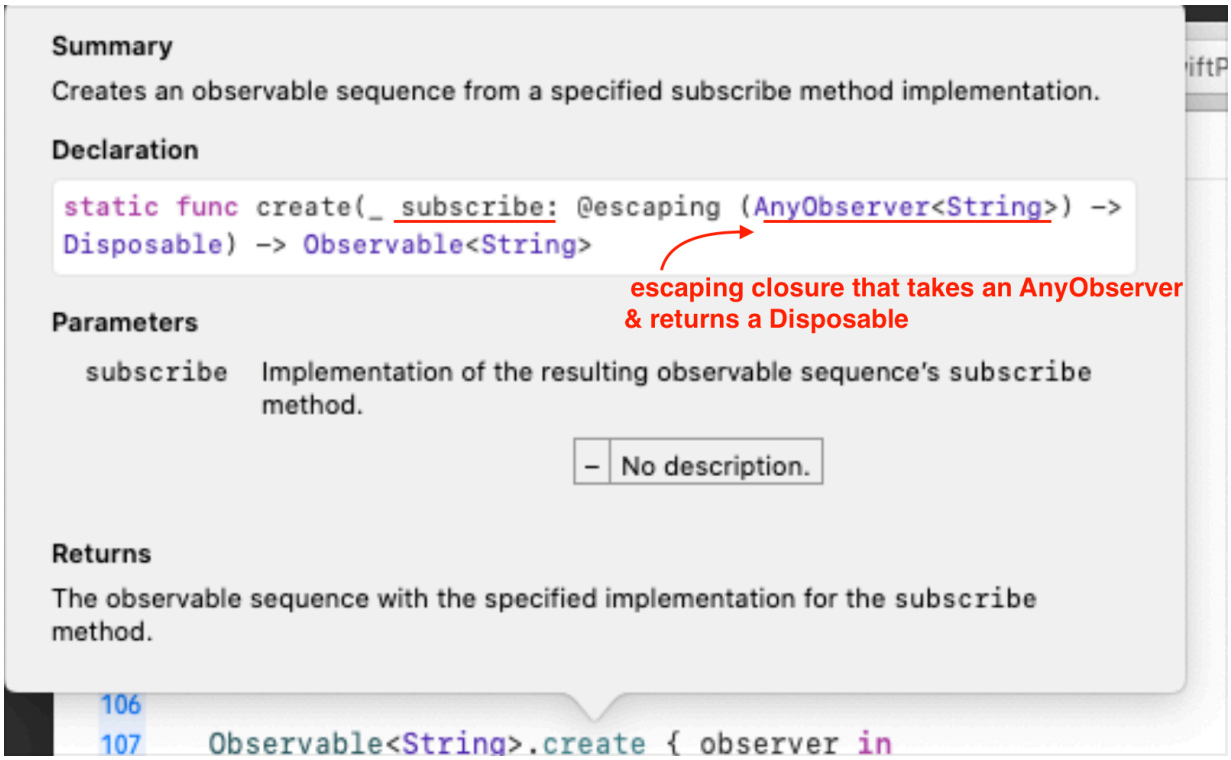
** Event has an **element property**. It's an optional value, because only next events have an element

- To handling specifically with next event , and ignore other stuff we do:

```
observable.subscribe(onNext: { element in
  print(element)
})
```

The onNext closure receives the next event's element as an argument, so you don't have to manually extract it from the event like you did before.

- Use **create** operator to create observable:
  - This is another way to specify all the events an observable will emit to subscribers.



  - '***subscribe***' parameter => Its job is to provide the implementation of calling subscribe on the observable.
    - '***subscribe***' parameter is an escaping closure that takes an AnyObserver and returns a Disposable.
      - It defines all the events that will be emitted to subscribers.

    - *AnyObserver* is a generic type that facilitates adding values onto an observable sequence, which will then be emitted to subscribers.

EX1 : Use create to create observable

```
Observable<String>.create { observer in
    // 1 Add a next event onto the observer.
    --> onNext(_:) is a convenience method for on(.next(_:)).
    observer.onNext("1")

    // 2 Add a completed event onto the observer.
    --> onCompleted is a convenience method for on(.completed).
    observer.onCompleted()

    // 3 Add another next event onto the observer.
    observer.onNext("?")

    // 4 Return a disposable, defining what happens when your observable is
    terminated or disposed of;
    In this case, no cleanup is needed so you return an empty disposable.
    return Disposables.create()
}
```

create observable

the value type when adding new value MUST matched with declared type - String

NOTE: Remember that 'subscribe' MUST return a disposable representing the subscription (check create(...) definition above).
=> Therefore, we use Disposables.create() to create a disposable.

```
// 5. subscribe to the observable
.subscribe(
    onNext: { print($0) },
    onError: { print($0) },
    onCompleted: { print("Completed") },
    onDisposed: { print("Disposed") }
)
.disposed(by: disposeBag)
```

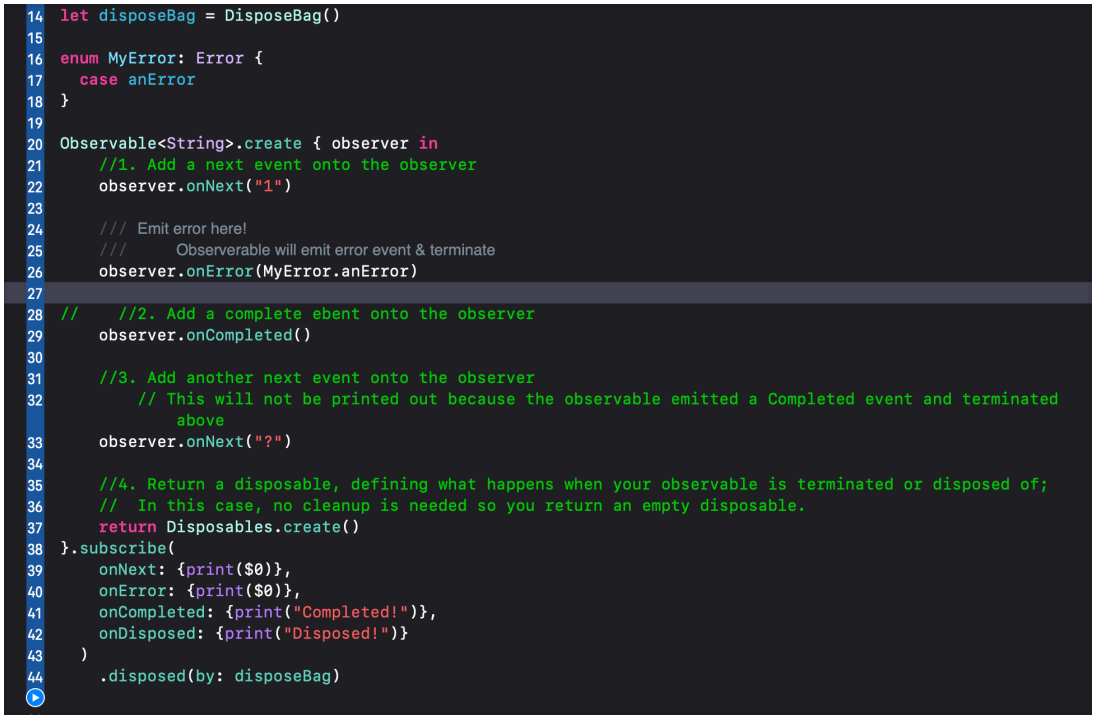
implement all handlers

add subscription to Dispose bag

```
--- Example of: create ---
1
Completed
Disposed
```

NOTE : The second next event is not printed because the observable emitted a completed event and terminated before it is added.

EX2: Observable emit Error Event



```
let disposeBag = DisposeBag()

enum MyError: Error {
  case anError
}

Observable<String>.create { observer in
    //1. Add a next event onto the observer
    observer.onNext("1")

    /// Emit error here!
    ///      Observerable will emit error event & terminate
    observer.onError(MyError.anError)

//    //2. Add a complete ebent onto the observer
    observer.onCompleted()

    //3. Add another next event onto the observer
        // This will not be printed out because the observable emitted a Completed event and terminated above
    observer.onNext("?")

    //4. Return a disposable, defining what happens when your observable is terminated or disposed of;
    //  In this case, no cleanup is needed so you return an empty disposable.
    return Disposables.create()
}.subscribe(
    onNext: {print($0)},
    onError: {print($0)},
    onCompleted: {print("Completed!")},
    onDisposed: {print("Disposed!")}
  )
    .disposed(by: disposeBag)
```

```
--- Example of: create ---
1
anError ←—— Observable emitted Error event & terminate
Disposed
```

EX3: Memory leak case when not adding DisposeBag or call Dispose()

```
Observable<String>.create { observer in
    //1. Add a next event onto the observer
    observer.onNext("1")

    /// Emit error here!
    ///      Observerable will emit error event & terminate
//    observer.onError(MyError.anError)

//    //2. Add a complete ebent onto the observer
//    observer.onCompleted()

    //3. Add another next event onto the observer
        // This will not be printed out because the observable emitted a Completed event and terminated
            above
    observer.onNext("?")

    //4. Return a disposable, defining what happens when your observable is terminated or disposed of;
    //  In this case, no cleanup is needed so you return an empty disposable.
    return Disposables.create()
}.subscribe(
    onNext: {print($0)},
    onError: {print($0)},
    onCompleted: {print("Completed!")},
    onDisposed: {print("Disposed!")}
  )
//    .disposed(by: disposeBag)
```

commented out

commented out

commented out

=> This will leak memory! The observable will never finish, and the disposable will never be disposed.

🟡 Observables Special Types:

- **empty**:

The empty operator creates an **empty observable sequence** with zero elements; it will ***only emit a completed event.***

```
let observable = Observable<Void>.empty()
```

```
observable.subscribe(
  // 1
  onNext: { element in
    print(element)
  },

  // 2
  onCompleted: {
    print("Completed")
  }
)
```

Q: Why to use empty?
A: They're handy when you want to return an observable that immediately terminates or intentionally has zero values.

- **never:**

The never operator creates an observable that **doesn't emit anything** and **never terminates**
It can be use to represent an infinite duration.

```
let observable = Observable<Void>.never()
```

```
  observable.subscribe(
    onNext: { element in
      print(element)
    },
    onCompleted: {
      print("Completed")
    }
  )
}
```

=> Nothing is printed. Not even "Completed"

- **range:** to generate an observable from a range of values.

EX:

1. Create an observable using the range operator, which takes a start integer value and a count of sequential integers to generate.
2. Calculate and print the **nth** Fibonacci number for each emitted element.

```
// 1
let observable = Observable<Int>.range(start: 1, count: 10

observable
  .subscribe(onNext: { i in
    // 2
    let n = Double(i)

    let fibonacci = Int(
      ((pow(1.61803, n) - pow(0.61803, n)) /
        2.23606).rounded()
    )

    print(fibonacci)
})
```

**NOTE: Except for the never() example, up to this point we've worked with observables that automatically emit a completed event and naturally terminate.

🟡 Disposing and terminating

- Subscription is what triggers an observable's work, causing it to emit new events until an error or completed event terminates the observable.
- You can also manually cause an observable to terminate by canceling a subscription to it (dispose it)
  - After you cancel the subscription, or **dispose** of it, the observable in the will stop emitting events.

EX:

```
// 1 Create an observable of strings.
let observable = Observable.of("A", "B", "C")

// 2 Subscribe to the observable, this time saving the returned Disposable
as a local constant called 'subscription'
let subscription = observable.subscribe { event in
  // 3 Print each emitted event in the handler.
  print(event)
}
```

```
// explicitly cancel a subscription
subscription.dispose()
```

- **DisposeBag:**

A dispose bag holds **Disposables** — typically added using the ***disposed(by:)*** method — and this will call ***dispose()*** on each one when the dispose bag is about to be deallocated
EX:

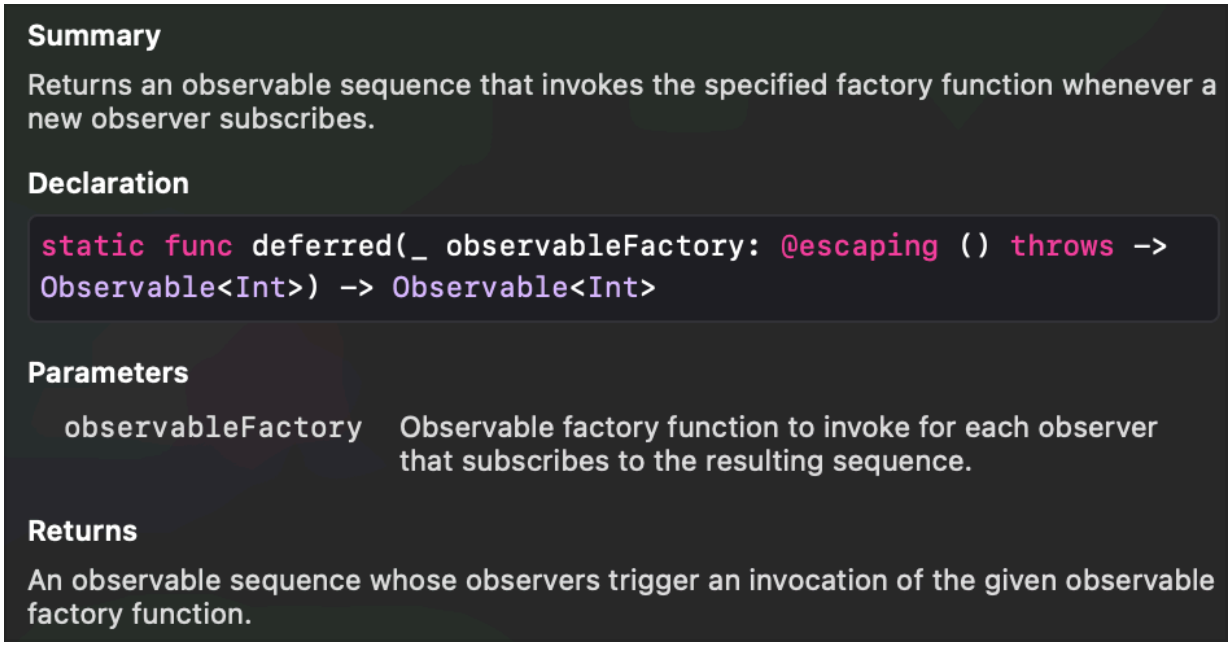
```
// 1 Create Dispose Bag
let disposeBag = DisposeBag()

// 2 Create an observable
Observable.of("A", "B", "C")
  .subscribe { // 3 Subscribe to the observable
    print($0)  & print out the emitted events using
               the default argument name $0
  } Add the returned Disposable from subscribe to the dispose bag
  .disposed(by: disposeBag) // 4
```

=> This is the pattern you'll use most frequently:
creating and subscribing to an observable, and immediately adding the subscription to a dispose bag.

Q: Why we need to dispose?
A:
If we not add subscription to dispose bag or manually call dispose() when we'ew done with the subscription, we will **experience memory leak.**

🟡 Create observable factories (Using deferred)

- Create observable factories that vend a new observable to each subscriber.
- Sometimes you want to **defer** some slow calculations until an Observable has an active subscription. In such cases, you may use ***Observable.deferred***.
    - ***Observable.deferred*** create an observable (observable factories), but defers reading of the value(emit) until a subscription.



EX1:

```swift
let disposeBag = DisposeBag()

// 1. Create a Bool flag to flip which observable to return.
var flip = false

// 2. Create an observable of Int factory using the deferred operator.
let factory: Observable<Int> = Observable.deferred {

    //3. Toggle flip, which happens each time factory is subscribed to.
    flip.toggle()

    //4. Return different observables based on whether flip is true or false.
    if flip {
        return Observable.of(1,2,3)
    }
    else {
        return Observable.of(4,5,6)
    }
}

// 5. To subscribe to 'factory' four times
    // Each time you subscribe to factory, you get the opposite observable.
for _ in 0...3 {
  factory.subscribe(onNext: {
    print($0, terminator: "")
  })
  .disposed(by: disposeBag)

  print() // print next output to next line
}
```

```
--- Example of: deferred ---
123
456
123
456
```

EX2: IOS Permission Defer Until Response

Sometimes you should read a value after the subscription. Take any iOS permission as the example:

```swift
func permissionObservable() -> Observable<PHAuthorizationStatus> {
    return .just(PHPhotoLibrary.authorizationStatus())
}
```

- Here, you read the authorizationStatus while creating the Observable not when an observer subscribes to it.
- Imagine a user changes the permission in iOS settings before an observer subscribes for the permissionObservable. The

observer will receive **the old value** of the permission
=> Therefore, we can use **deferred** in this case:

```
func permissionObservable() -> Observable<PHAuthorizationStatus> {
    return Observable.deferred {
        return .just(PHPhotoLibrary.authorizationStatus())
    }
}
```

Deferred defers reading of the value until a subscription => This means the observer will have a fresh permission status and your calculation will be calculated when the result is indeed needed.

Using Traits : https://github.com/ReactiveX/RxSwift/blob/main/Documentation/Traits.md

- Traits are **observables with a narrower set of behaviors** than regular observables.
- Their use is optional
- Type of Traits:
  1. **Single:**
     - A **Single** is a variation of Observable that, instead of emitting a series of elements, is always guaranteed to emit either *a single element* or *an error*.
       - ***Single*** will emit either a **success(value)** or **failure(error)** event.

     - **success(value)** is a combination of the **next** and **completed** events.
       - It's useful for one-time processes that will either succeed and yield a value or fail. Such as when downloading data or loading it from disk.
       - One common use case for using Single is for performing HTTP Requests that could only return a response or an error, but a Single can be used to model any case where you only care for a single element, and not for an infinite stream of elements.

EX: LoadText file function using Single

```
// 1. Create a dispose bag to use later
let disposeBAG = DisposeBag()

// 2. Define an Error enum to model some possible errors that can occur in
    reading data from a file on disk.

enum FileReadError : Error {
    case fileNotFound, unreadable, encodingFailed
}
```

```
// 3. Implement a function to load text from a file on disk that returns a
   Single.
func loadText(from name : String) -> Single<String> {
   // 4. Create and return a Single.
    return Single.create { single in

        // a. Create a Disposable, because the subscribe closure of create
            expects it as its return type
        let disposable = Disposables.create()

        // b. Get the path for the filename, or else add a file not found error
            onto the Single and return the disposable you created.
        guard let path = Bundle.main.path(forResource: name, ofType: "txt")
            else {
            single(.failure(FileReadError.fileNotFound))
          return disposable
        }

        // c. Get the data from the file at that path, or add an unreadable
            error onto the Single and return the disposable.
        guard let data = FileManager.default.contents(atPath: path) else {
          single(.failure(FileReadError.unreadable))
          return disposable
        }

        // d.Convert the data to a string; otherwise, add an encoding failed
            error onto the Single and return the disposable.
        guard let contents = String(data: data, encoding: .utf8) else {
          single(.failure(FileReadError.encodingFailed))
          return disposable
        }

        // e. Add the contents onto the Single as a success, and return the
            disposable.
        single(.success(contents))

        return disposable
    }
}
```

```
135 // 5. Call loadText(from:) and pass the root name of the text file.
136 loadText(from: "Copyright")
137 // 6. Subscribe to the Single it returns.
138   .subscribe {
139     // Switch on the event and print the string if it was successful, or print
          the error if not.
140     switch $0 {
141     case .success(let string):
142       print(string)
143     case .failure(let error):
144       print(error)
145     }
146   }
147   .disposed(by: disposeBag)
```

2. **Maybe:**
   - ***Maybe*** is a mashup of a Single and Completable.
   - It can either emit
     a success(value), completed or error(error). I
   - Usage: if you need to implement an operation that could either succeed or fail, and optionally return a value on success

3. **Completable:**
   - ***Completable*** will only emit
     a **completed** or **error(error**) event. It will not emit any values.
   - We could use a completable when we only care that an operation completed successfully or failed, such as a file write.