RXSwift - Subjects

Ref link:
https://www.raywenderlich.com/books/rxswift-reactive-programming-with-swift/v4.0/chapters/3-subjects
https://fxstudio.dev/rxswift-hello-subjects/

Definition:

Observables are a fundamental part of RxSwift, but they're essentially read-only. You may only subscribe to them to get notified of new events they produce.

A common need when developing apps is to manually add new values onto an observable during runtime to emit to subscribers. That's why we need **Subject**

- Subject can act as both **Observable sequence & Observer**
  - An **Observable sequence**, which means it can be subscribed to
  - An **Observer** that enables adding new elements onto a subject that will then be emitted to the subject subscribers

Type of subjects:

PublishSubject	Starts empty and only emits new elements to subscribers
BehaviorSubject	Starts with an initial value and replays it or the latest element to new subscribers.
ReplaySubject	Initialized with a buffer size and will maintain a buffer of elements up to that size and replay it to new subscribers.

AsyncSubject	Emits only the last ***next*** event in the sequence, and only when the subject receives a completed event. This is a seldom used kind of subject
Variable	
PublishRelay & BehaviorRelay	These wrap their respective subjects, but only accept and relay next events.
	You cannot add a completed or error event onto relays at all, so they're great for non-terminating sequences.

NOTE:
Emitting previous next events to new subscribers is called ***replaying****, and publish subjects* ***DO NOT replay****.*

🟡 Publish Subject: https://fxstudio.dev/rxswift-publish-subjects/

- **PublishSubject** will receive information and then publish it to subscribers.
  - It's of type String, so it can ONLY receive and publish strings.
  - After being initialized, it's ready to receive strings.
- Emits ONLY new next events to its subscribers.
  - Elements added to a **PublishSubject** before a subscriber subscribes will not be received by that subscriber

```swift
// 1. Create PubishSubject
var subject = PublishSubject<String>()

// 2. This puts a new string onto the 'subject', but nothing is printed out yet, because there are no observers.
subject.onNext("Yo!")

// 3. Create observer by subcribing to 'subject'
let subscription1 = subject.subscribe(
    onNext: { string in
    print("On subcriber #1: " + string)
    },
    onCompleted: {print("Completed!")},
    onDisposed: {print("Disposed!")}
  )

// 4. Now, because subject has a subscriber (subscription1), when it emit new value, subscriber will get string "Hello", "World"
subject.onNext("Hello") // add new value to sequence
subject.onNext("World")
    /// NOTE: If you subscribe to that subject after adding “Hello” and “World” using onNext(), you won’t receive these two values through events.

```

```swift
// 5. Create another observer (subcription2) subcribe to the channel

let subscription2 = subject.subscribe{ event in
    //use the nil-coalescing operator here to print the element if there is one;
        // otherwise, you print the event.
        print("On subcriber #2:", event.element ?? event)
}

// 6. When emit new value, the string is printed out twice (2x), one for subscription1 and one for subscription2
subject.onNext("subcriber #2 starts subscribing")

// 7. Dispose subscription1
subscription1.dispose()

// 8. Add another 'next' event
//  The string is only printed out one time only (on subcriber #2) because subcriber #1 was disposed
subject.onNext("subcriber #1 has left")
```

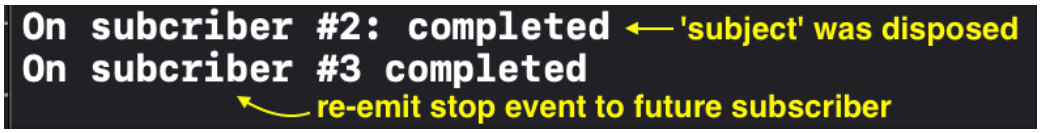
```
On subcriber #1: Hello
On subcriber #1: World
On subcriber #1: subcriber #2 starts subscribing
On subcriber #2: subcriber #2 starts subscribing
Disposed!  <-- suscriber #1 disposed
On subcriber #2: subcriber #1 has left
        <-- new emitted element only notified for subscriber #2
```

**__NOTE__: Subscribers will be notified of new events from the point at which they subscribed, until either they unsubscribe, or the subject has terminated with a completed or error event.

- When a publish subject receives a **completed** or **error** event, also known as **a stop event**, it will emit that stop event to new subscribers and it will no longer emit next events.

EX: (continue the code from above)

```
// 9. Add a completed event onto the subject
subject.onCompleted()

// 10. Add another element onto the 'subject'
//  This won't be emitted and printed, though, because the subject has already terminated.
subject.onNext("Subject is terminated")

// 11. Dispose subscription2
subscription2.dispose()

let disposeBag = DisposeBag()

// 12. Subscribe to the subject, this time adding its disposable to a dispose bag.

//  Subjects, once terminated, will re-emit their stop event to future subscribers.
//  In the ouput, you will see the 'completed' event replayed
subject
  .subscribe {
    print("On subcriber #3", $0.element ?? $0)
  }
  .disposed(by: disposeBag)

// 13. When 'subject' is terminated, it's no longer emit next event.
    // Therefore, new subscriber WILL NOT bring 'subject' back after it terminated
        // meaning, you will never get this line print out.
subject.onNext("Subscriber #3 start subscribing, but the channel is off")
```



Behavior Subject: https://fxstudio.dev/rxswift-behavior-subjects/

- **Behavior subjects** work similarly to publish subjects, except they will replay the latest next event to new subscribers
  - Subscribers will always **receive the most recent 'next' event** in the sequence even if they *subscribed after* that event was emitted

- A **BehaviorSubject** is *initialized* with a starting value
  - Because BehaviorSubject always emits its latest element, you can't create one without providing an initial value
    - If you can't provide an initial value at creation time, that probably means you need to use a **PublishSubject** instead, or model your element as an **Optional**.

- Then, it **replays** to the new **subscribers** a 'next' event containing the most recent elements
- OR the initial value if no new recent elements have been added to it beforehand.

```swift
let disposeBag = DisposeBag()

// 1. Define an error type
enum MyError: Error {
  case anError
}

// 2. Create a helper function to print the element if there is one, an error if there is one, or else the event itself.
func print<T: CustomStringConvertible>(label: String, event: Event<T>) {
  print(label, (event.element ?? event.error) ?? event)
}

// 3. Create a new BehaviorSubject instance. Its initializer takes an initial value
let behavorialSubject = BehaviorSubject(value: "Initial value") ← need initialized
```

```swift
// 4. Subscribe behavorialSubject
    // Because no other elements have been added to the subject, it replays its initial value to the subscriber.
/// NOTE: if we add an 'next' event.first before we subcribe it,
/// then the lattest element that will be printed out is the element in the 'next' event, not the initial value
behavorialSubject
  .subscribe {
    print(label: "1st Subscribing: ", event: $0)
  }
  .disposed(by: disposeBag)

// 5. Emits an error event onto behavorialSubject and terminate
behavorialSubject.onError(MyError.anError)

// 6. Create subcription #2 to behavorialSubject
    //Similar to PublishSubject, behavior subjects replay their latest value to new subscribers.
behavorialSubject
  .subscribe {
    print(label: "2nd Subscribing:", event: $0)
  }
  .disposed(by: disposeBag)
```

```
1st Subscribing:  Initial value
1st Subscribing:  anError
2nd Subscribing: anError
```

- Usage:
  - Behavior subjects are useful when you want to pre-populate a view with the most recent data.
    - EX1: you could bind controls in a user profile screen to a behavior subject, so that the latest values can be used to pre-populate the display while the app fetches fresh data.
    - EX2: In a chat app, you might use a **BehaviorSubject** to

pre-fill a new posts title text field beginning with the initial name untitled.

Replay Subject: https://fxstudio.dev/rxswift-replay-subjects/

- Replay subjects will temporarily cache, or buffer, the latest elements they emit, up to a specified size of your choosing. They will then replay that buffer to new subscribers.
- A **ReplaySubject** is ***initialized* with a buffer size** and that value cannot be changed after initialization.
- When creating a replay subject takes in an **array** of items.
  - Each emitted element will be an array, so the buffer size will buffer that many arrays.
  - It would be easy to create memory pressure here if you're not careful.
- It will also reemit its stop event to new subscribers
- Usage:
  - *You can use replay subject to display as many as the **five most recent search items** whenever a search controller is presented.*

- EX:

```
// 1. Create a new replay subject with a buffer size of 2
let replaySubject = ReplaySubject<String>.create(bufferSize: 2)
/// let disposeBag = DisposeBag()

// 2. Add 3 elements onto the subject
replaySubject.onNext("1")
replaySubject.onNext("2")
replaySubject.onNext("3")

// 3. Create 2 subscriptions to the subject.
//      + The latest two elements are replayed to both subscribers which is '2' and '3'
//          because the buffer size is 2, meaning take only 2 latest elements

replaySubject
    .subscribe {
    print(label: "Subscriber #1: ", event : $0)
    }
    .disposed(by: disposeBag)

replaySubject
    .subscribe {
    print(label: "Subscriber #2: ", event : $0)
    }
    .disposed(by: disposeBag)
```

```
Subscriber #1:  2
Subscriber #1:  3
Subscriber #2:  2
Subscriber #2:  3
```

- The latest two elements are replayed to both subscribers;
  - '1' never gets emitted, because '2' and '3' are added onto the replay subject with a buffer size of 2 before anything subscribed to it.
- When adding another 'next' event with value '4', and new 3rd subscription:

```
// 4. Add another element onto subject

replaySubject.onNext("4")

// 5. Create another subscription to the subject.
    // Now, subscriber 1 & 2 are recieved '4' normally.
        // And new subsciber 3 just come in get replay '3' and '4'
replaySubject
  .subscribe {
    print(label: "Subscriber #3: ", event: $0)
  }
  .disposed(by: disposeBag)
```

  - 1st and 2nd subscriber will receive the 'next' event (value = '4') as normal

        - 3rd subscriber (new subscriber) will get the replays of element '3' and '4'

    - If we emit an error event before 3rd subscriber subscibes

```
164 // 4. Add another element onto subject
165
166 replaySubject.onNext("4")
167
168 // 5. Emits error event
169 replaySubject.onError(MyError.anError)
170
171 // 6. Create another subscription to the subject.
172 //      + replaySubject will re-emit stop event to new subsciber #3
173 //      + buffer is also still hanging around, so it gets replayed to new subscribers as well before
        it re-emits stop event
174
175 replaySubject
176   .subscribe {
177     print(label: "Subscriber #3: ", event: $0)
178   }
179   .disposed(by: disposeBag)
180
```

```
Subscriber #1:  anError
Subscriber #2:  anError
Subscriber #3:  3          <-- replay last 2 elements
Subscriber #3:  4
Subscriber #3:  anError    <-- then, re-emit stop event
```

    - the buffer is also still hanging around, so it gets replayed to new subscribers #3 as well
    - then, the stop event is re-emitted.

Good to know:

if we call dispose() after emit error event *(not common thing to do because you have already added your subscriptions to dispose bag) :*

    - subject will re-emit the stop event, BUT NOT replay the buffer elements to new subscriber #3

```swift
// 5. Emits error event
replaySubject.onError(MyError.anError)

// 6. Explicitly call dispose() - NOT COMMON
replaySubject.dispose()

// 6. Create another subscription to the subject.
//      + replaySubject will re-emit stop event to new subsciber #3

replaySubject
  .subscribe {
    print(label: "Subscriber #3: ", event: $0)
  }
  .disposed(by: disposeBag)
```

```
Subscriber #1:  anError
Subscriber #2:  anError
Subscriber #3:  Object `RxSwift.(unknown context at $10fb28a20).ReplayMany<Swift.String>` was already disposed.
```

Relays

- A relay wraps a subject while maintaining its replay behavior.
- You add a value onto a relay by using the **accept(_:)** method. You DON'T use **onNext(_:)**
  - This is because relays can only accept values, i.e., you **CANNOT add an error or completed event** onto them.
    - Therefore, they are ***guaranteed to never terminate.***

- A **PublishRelay** wraps a **PublishSubject** and a **BehaviorRelay** wraps a **BehaviorSubject**.

- A **behavior relay** is created with an initial value, and it will replay its latest or initial value to new subscribers.
  - You can also ask it for its current value at any time.

EX of PublishRelay:

```swift
let relay = PublishRelay<String>()
//let disposeBag = DisposeBag()

//add new element to relay
relay.accept("Knock knock, anyone home?") // There are no subscribers yet, so nothing is emitted.

//create a subscriber
relay
  .subscribe(onNext: {
    print("Subscriber #1: " + $0)
  })
  .disposed(by: disposeBag)

//add another new element to relay
relay.accept("New sub #1 is added")
```

```
>>>>>PublishRelay Example:
same output as you created PublishSubject
Subscriber #1: New sub #1 is added
```

+ There is no way to add an error or completed event onto a relay. Any attempt to do so will generate a compiler error:

CANNOT DO THIS !!

```swift
relay.accept(MyError.anError)
relay.onCompleted()
```

EX of BehaviorRelay:

+ Behavior relays also will not terminate with a completed or error event.

```swift
// 1. Create a behavior relay with an initial value
    ///NOTE: The relay's type is inferred, but you could also explicitly declare the type as:
    /// BehaviorRelay<String>(value: "Initial value")
    ///
let behaviorRelay = BehaviorRelay(value: "Initial value")

// 2. Add a new element onto the relay.
behaviorRelay.accept("New initial value")

// 3. Subscribe to the relay.
    // Subscriber will recieved the latest element or initial value that is emitted before it
behaviorRelay
    .subscribe {
      print(label: "Subscription 1: ", event: $0)
    }
    .disposed(by: disposeBag)
```

```
>>>>>BehaviorReplay Example:
Subscription 1:  New initial value
```
replay latest element or initial value

(continue with the code above)

```
244
245 // 4. Add another new element
246 behaviorRelay.accept("1")
247
248 // 5. Add another subscriber
249 behaviorRelay
250   .subscribe {
251     print(label: "Subscription 2: ", event: $0)
252   }
253   .disposed(by: disposeBag)
254
255 // 6. Add another new element
256 behaviorRelay.accept("2")
```

```
Subscription 1:  1
Subscription 2:  1
Subscription 1:  2
Subscription 2:  2
```
subcriber #1 recieve new value '1' normally

NEW subcriber #2 recieve new value '1' as replaying latest emitted element

both subscribers 1 & 2 recieve new emitted value '2' normally

NOTE : You can get access to current value in Behavior Relay:

```
255 // 6. Add another new element
256 behaviorRelay.accept("2")
257
258 // 7. Behavior relays let you directly access their current value.
259 print("Current value in behavior relay: " + behaviorRelay.value)
```
current value added to relay

```
Current value in behavior relay: 2
```