Sun*

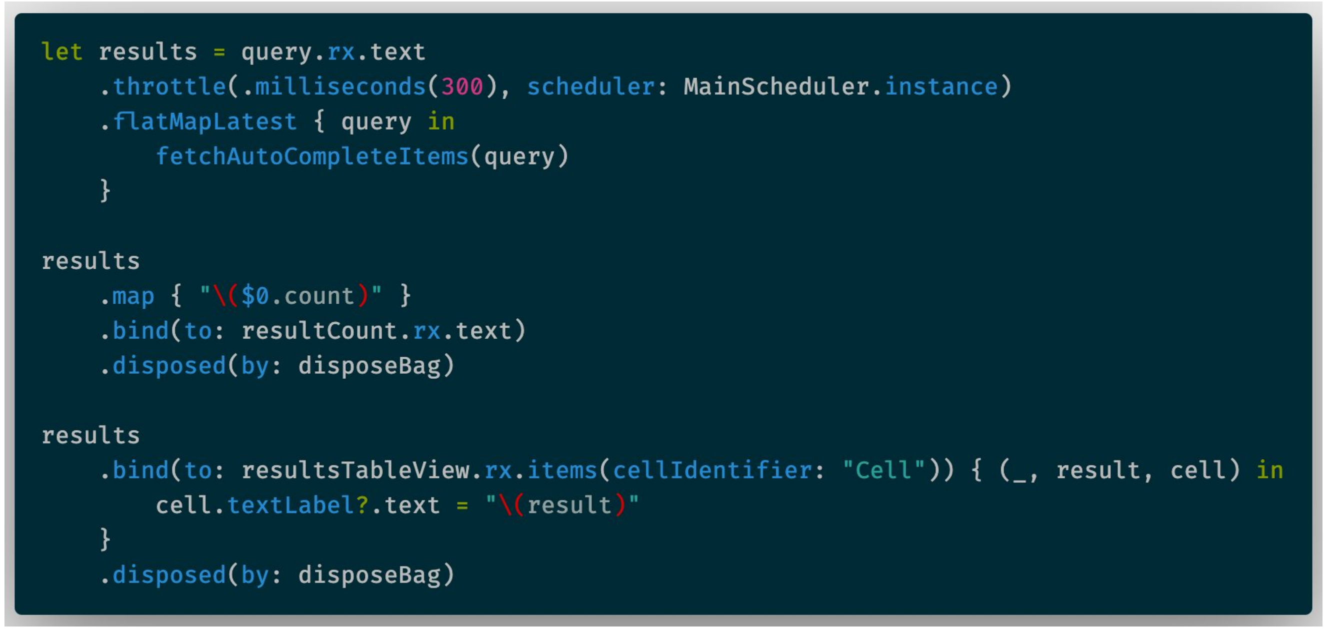
Traits

Outline

1. Observable problem
2. Driver
3. Signal
4. ControlProperty
5. ControlEvent

1. Observable problem

❖ Practical usage example:



```swift
let results = query.rx.text
    .throttle(.milliseconds(300), scheduler: MainScheduler.instance)
    .flatMapLatest { query in
        fetchAutoCompleteItems(query)
    }

results
    .map { "\($0.count)" }
    .bind(to: resultCount.rx.text)
    .disposed(by: disposeBag)

results
    .bind(to: resultsTableView.rx.items(cellIdentifier: "Cell")) { (_, result, cell) in
        cell.textLabel?.text = "\(result)"
    }
    .disposed(by: disposeBag)
```

1. Observable problem

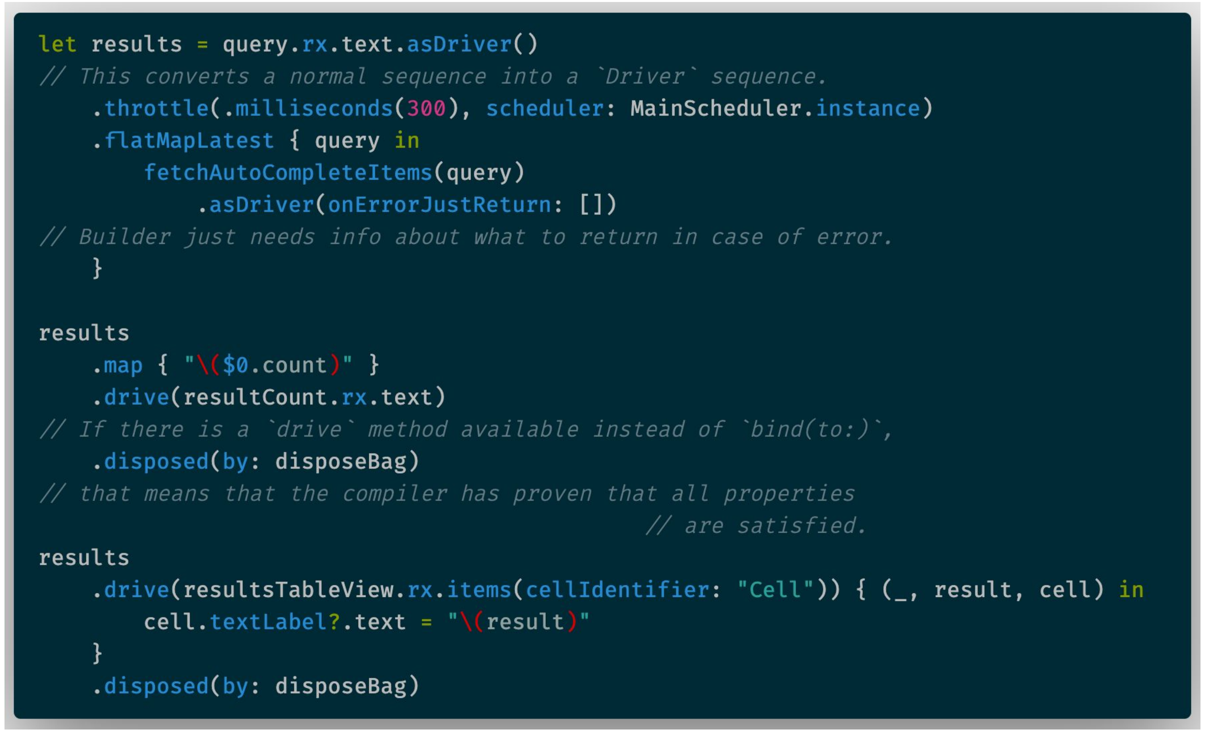
- The problems with the previous code:
  - If the fetchAutoCompleteItems observable sequence errors out (connection failed or parsing error), this error would unbind everything and the UI wouldn't respond any more to new queries.
  - If fetchAutoCompleteItems returns results on some background thread, results would be bound to UI elements from a background thread which could cause non-deterministic crashes.
  - Results are bound to two UI elements, which means that for each user query, two HTTP requests would be made, one for each UI element, which is not the intended behavior.

2. Driver

- This is the most elaborate trait. Its intention is to provide an intuitive way to write reactive code in the UI layer, or for any case where you want to model a stream of data *Driving* your application.
- Properties:
  - Can't error out.
  - Observe occurs on main scheduler.
  - Shares side effects (`share(replay: 1, scope: .whileConnected)`)

2. Driver

❖ Example:



```swift
let results = query.rx.text.asDriver()
// This converts a normal sequence into a `Driver` sequence.
    .throttle(.milliseconds(300), scheduler: MainScheduler.instance)
    .flatMapLatest { query in
        fetchAutoCompleteItems(query)
            .asDriver(onErrorJustReturn: [])
// Builder just needs info about what to return in case of error.
    }

results
    .map { "\($0.count)" }
    .drive(resultCount.rx.text)
// If there is a `drive` method available instead of `bind(to:)`,
    .disposed(by: disposeBag)
// that means that the compiler has proven that all properties
                                                // are satisfied.
results
    .drive(resultsTableView.rx.items(cellIdentifier: "Cell")) { (_, result, cell) in
        cell.textLabel?.text = "\(result)"
    }
    .disposed(by: disposeBag)
```

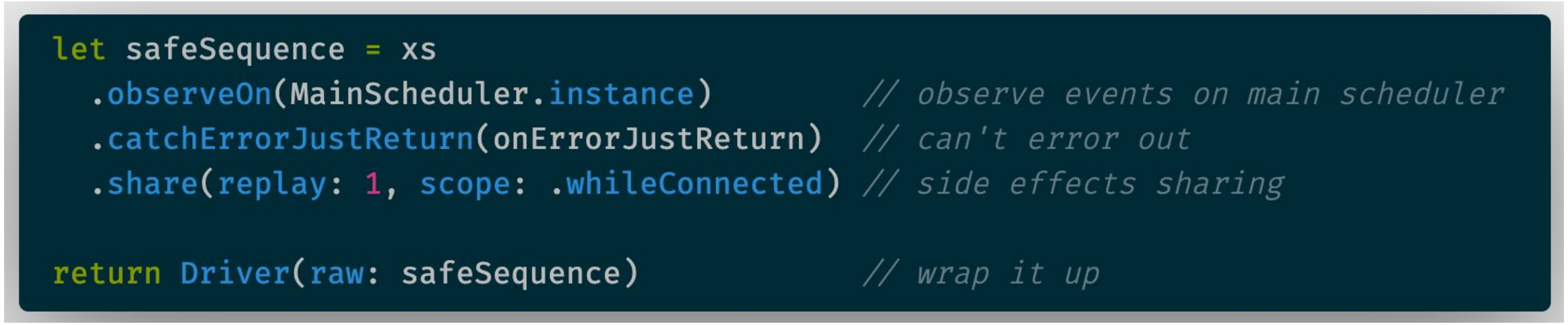
2. Driver

- Any observable sequence can be converted to **Driver** trait, as long as it satisfies 3 properties:
  - Can't error out.
  - Observe occurs on main scheduler.
  - Shares side effects (share(replay: 1, scope: .whileConnected))

Sun*

2. Driver

❖ To make sure those properties are satisfied, just use normal Rx operators. asDriver(onErrorJustReturn: []) is equivalent to following code:



```
let safeSequence = xs
  .observeOn(MainScheduler.instance)        // observe events on main scheduler
  .catchErrorJustReturn(onErrorJustReturn) // can't error out
  .share(replay: 1, scope: .whileConnected) // side effects sharing

return Driver(raw: safeSequence)            // wrap it up
```

2. Driver

❖ The final piece is using `drive` instead of `bind(to:)`

❖ `drive` is defined only on the **Driver** trait. This means that if you see drive somewhere in code, that observable sequence can never error out and it observes on the main thread, which is safe for binding to a UI element

Sun*

3. Signal

❖ A **Signal** is similar to **Driver** with one difference, it does **not** replay the latest event on subscription, but subscribers still share the sequence's computational resources.

❖ It can be considered a builder pattern to model Imperative Events in a Reactive way as part of your application.

Sun*

3. Signal

- A Signal:
  - Can't error out.
  - Delivers events on Main Scheduler.
  - Shares computational resources (share(scope: .whileConnected)).
  - Does NOT replay elements on subscription.

4. ControlProperty

- ❖ Trait for **Observable/ObservableType** that represents a property of UI element.

- ❖ Sequence of values only represents initial control value and user initiated value changes. Programmatic value changes won't be reported.

- ❖ The implementation of **ControlProperty** will ensure that sequence of events is being subscribed on main scheduler (subscribeOn(ConcurrentMainScheduler.instance) behavior).

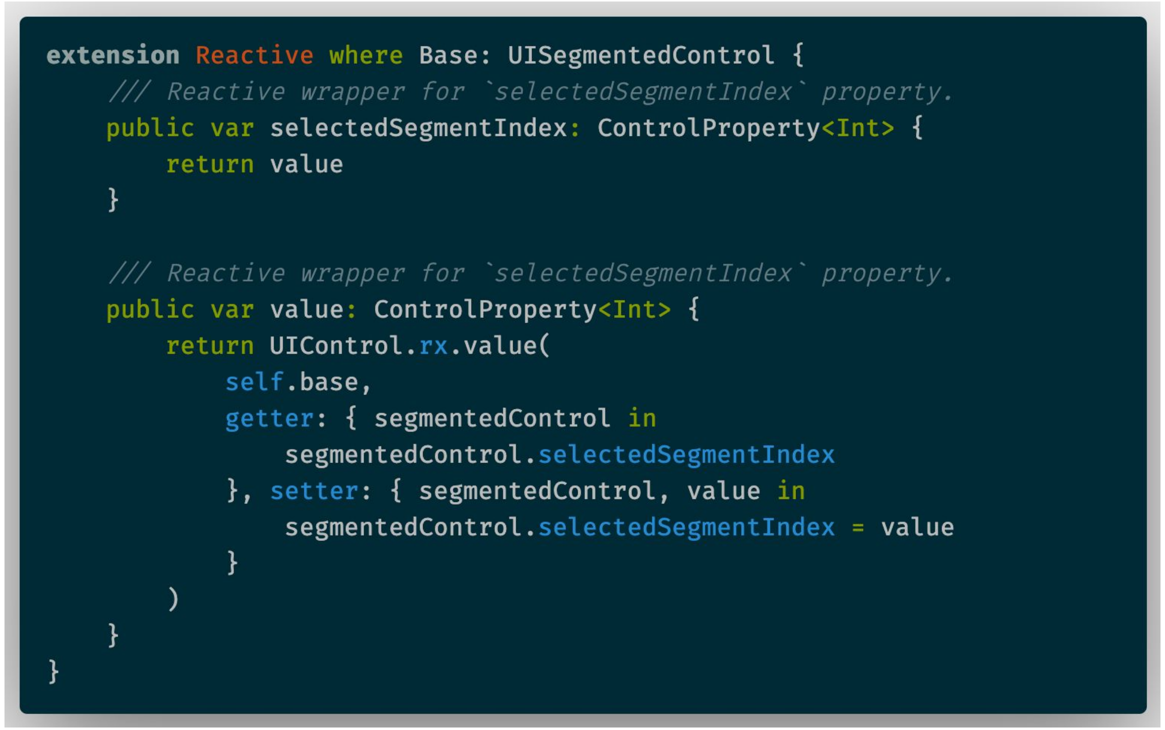
4. ControlProperty

- It's properties are:
  - it never fails
  - share(replay: 1) behavior
  - it's stateful, upon subscription (calling subscribe) last element is immediately replayed if it was produced
  - it will **Complete** sequence on control being deallocated
  - it never errors out
  - it delivers events on MainScheduler.instance

Sun*

4. ControlProperty

❖ Example:



```
extension Reactive where Base: UISegmentedControl {
    /// Reactive wrapper for `selectedSegmentIndex` property.
    public var selectedSegmentIndex: ControlProperty<Int> {
        return value
    }

    /// Reactive wrapper for `selectedSegmentIndex` property.
    public var value: ControlProperty<Int> {
        return UIControl.rx.value(
            self.base,
            getter: { segmentedControl in
                segmentedControl.selectedSegmentIndex
            }, setter: { segmentedControl, value in
                segmentedControl.selectedSegmentIndex = value
            }
        )
    }
}
```

5. ControlEvent

- Trait for **Observable/ObservableType** that represents an event on a UI element.

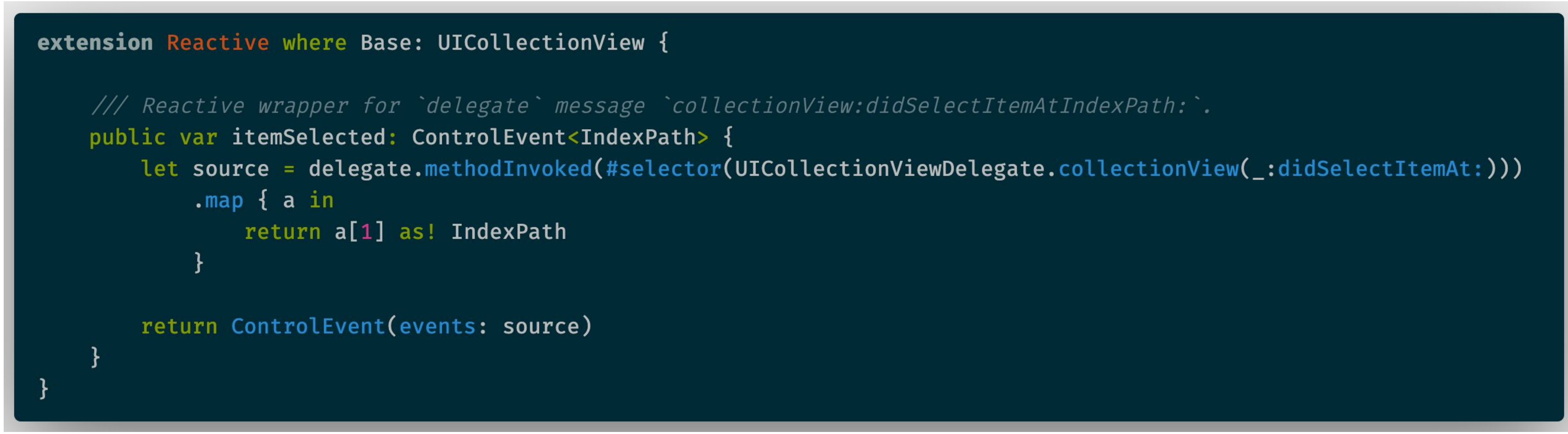
- The implementation of **ControlEvent** will ensure that sequence of events is being subscribed on main scheduler (subscribeOn(ConcurrentMainScheduler.instance) behavior).

5. ControlEvent

- It's properties are:
  - it never fails
  - it won't send any initial value on subscription
  - it will Complete sequence on control being deallocated
  - it never errors out
  - it delivers events on MainScheduler.instance

5. ControlEvent

❖ Example:



```
extension Reactive where Base: UICollectionView {

    /// Reactive wrapper for `delegate` message `collectionView:didSelectItemAtIndexPath:`.
    public var itemSelected: ControlEvent<IndexPath> {
        let source = delegate.methodInvoked(#selector(UICollectionViewDelegate.collectionView(_:didSelectItemAt:)))
            .map { a in
                return a[1] as! IndexPath
            }

        return ControlEvent(events: source)
    }
}
```

Sun*

Question & Answer?

Sun*