Introduction to Functional Reactive Programming in Swift

- Sebastian Grail
- iOS Developer at Canva
- 🐦 @sebastiangrail
- github: sebastiangrail

We're Hiring!

- Java Engineers
- Front End Engineers
- Mobile Developers (end of the year)
- sebastian@canva.com

Why use Functional Reactive Programming

Why use Functional Reactive Programming

- Keeping track of state is hard
- Multiple patterns to notify of changes
- Cocoa APIs tend to spread concerns over multiple places
- Manuel said that state is bad and signals are awesome!

What is Reactive Programming?

- Reactive programming abstracts changes over time into signals
- Signals send values until they either complete or error out
- Signals can be observed
- Observers receive new values from their signals

Function returning single value

- Synchronous

```
T -> U
```

Function returning single value

- Synchronous

  ```
  T -> U
  ```

- Asynchronous

  ```
  (T, U -> Void) -> Void
  ```

Function returning single value

- Synchronous

  ```
  T -> U
  ```

- Asynchronous

  ```
  T -> Future<U>
  ```

Function returning multiple values

- Synchronous

```
T -> [U]
```

Function returning multiple values

- Synchronous

  ```
  T -> [U]
  ```

- Asynchronous

  ```
  T -> Future<[U]>
  ```

Function returning multiple values

- Synchronous

  ```
  T -> [U]
  ```

- Asynchronous

  ```
  T -> [Future<U>]
  ```

Function returning multiple values

- Synchronous

```
T -> [U]
```

- Asynchronous

```
(T, U -> Void) -> Void
```

Function returning multiple values

- Synchronous

```
T -> [U]
```

- Asynchronous

```
(T, U -> Void, () -> Void) -> Void
```

Function returning multiple values

- Synchronous

```
T -> [U]
```

- Asynchronous

```
(T, U -> Void, E -> Void, () -> Void) -> Void
```

Function returning multiple values

- Synchronous

```
T -> [U]
```

- Asynchronous

```
T -> Signal<U, E>
```

Implementing an MVS*

*Minimum Viable Signal

Where can Signals be used?

- Button: Replace target/action with Signal
- Table view: Signal replaces delegate methods like `didSelect...`
- Network request: Signal that sends data when available
- Replace mutable variables: Push new values to observers

Hot Signal

- Always on
- Subscription does not trigger side effects
- All observers get the same events
- Usually long lived

Cold Signal

- Short life cycle
- Subscription triggers side effects
- Every subscriber gets its own events
- Usually completes after work is done
- "Producer of Signals"

Reactive Cocoa

Reactive Cocoa

- Original Objective-C API started in 2012
- Inspired by Reactive Extensions for .Net
- RAC 3 introduces new Swift API
- ~~Still in beta!~~ ~~RC1 released!~~ 3.0 released this week!
- RAC 4 targets Swift 2

Event

```
enum Event<T, E : ErrorType> {
    case Next(T)
    case Error(E)
    case Completed
    case Interrupted
}
```

Signal

```
final class Signal<T, E : ErrorType>
```

Creating a Signal

```
init(generator: SinkOf<Event<T, E>> -> Disposable?)
```

Creating a Signal

```
let signal = Signal<String, NoError> { sink in
    NSOperationQueue().addOperationWithBlock {
        while true {
            sleep(1)
            sendNext(sink, "Hello World!")
        }
    }
    return nil
}
```

Observing a signal

```
func observe<T, E>(
    next: (T -> ())? = nil,
    error: (E -> ())? = nil,
    completed: (() -> ())? = nil,
    interrupted: (() -> ())? = nil)
    (signal: Signal<T, E>) -> Disposable?
```

Observing a signal

```
signal |> observe(next: println)
signal |> observe(next: println, error: handleError)
signal |> observe(completed: signalCompleted)
```

Signals are hot

```
let signal = Signal<Int, NoError> { sink in
    NSOperationQueue().addOperationWithBlock {
        for i in 0...Int.max {
            sleep(1)
            println("sending")
            sendNext(sink, i)
        }
    }
    return nil
}
// sending - sending - sending - ...
```

Signals are hot

```
let signal = Signal<Int, NoError> { sink in
    NSOperationQueue().addOperationWithBlock {
        for i in 0...Int.max {
            sleep(1)
            println("sending")
            sendNext(sink, i)
        }
    }
    return nil
}
signal |> observe(next: println)
signal |> observe(next: println)
// sending - 1 - 1 - sending - 2 - 2
```

SignalProducer

```
struct SignalProducer<T, E : ErrorType>
```

Creating a SignalProducer

```
init(startHandler: (SinkOf<Event<T, E>>,
                    CompositeDisposable) -> ())
```

Creating a SignalProducer

```
let producer = SignalProducer<String, NoError> { sink, disposable in
    sendNext(sink, "Hello World")
    sendCompleted(sink)
}
```

Starting a SignalProducer

```
func start<T, E>(
    next: (T -> ())? = nil,
    error: (E -> ())? = nil,
    completed: (() -> ())? = nil,
    interrupted: (() -> ())? = nil)
    (producer: SignalProducer<T, E>) -> Disposable
```

Starting a SignalProducer

```
producer |> start(next: println)
producer |> start(next: println, error: handleError)
producer |> start(completed: signalCompleted)
```

SignalProducers are cold

```
let producer = SignalProducer<Int, NoError> { sink, disposable in
    println("sending")
    sendNext(sink, 1)
    sendNext(sink, 2)
    sendCompleted(sink)
    disposable.addDisposable { println("disposing") }
}
// No output
```

SignalProducers are cold

```
let producer = SignalProducer<Int, NoError> { sink, disposable in
    println("sending")
    sendNext(sink, 1)
    sendNext(sink, 2)
    sendCompleted(sink)
    disposable.addDisposable { println("disposing") }
}
producer |> start(next: println)
producer |> start(next: println)
// "sending" - 1 - 2 - "disposing" - "sending" - 1 - 2 - "disposing"
```

Other ways to create Signals

MutableProperty<T>

- Encapsulates a mutable property
- Exposes a SignalProducer

```
let property = MutableProperty<Int>(0)
...
property <~ someSignal
```

DynamicProperty

- Mostly for legacy Objective-C code
- Wraps a KVO property

```
DynamicProperty(object: someObject, keyPath: "keyPath") <~ someSignal
```

`RACSignal`

- ReactiveCocoa 2 signal
- Can be converted using

  ```
  toSignalProducer() ->
  SignalProducer<AnyObject?, NSError>
  ```

RACSignal

```
UITextField
    .rac_textSignal()

UIControl
    .rac_signalForControlEvents(UIControlEvents)

NSNotificationCenter
    .rac_addObserverForName(String, object: AnyObject)

UIGestureRecognizer
    .rac_gestureSignal
```

Composing Signals

Composing Signals

- Functions to compose signals
- Defined both as methods and top level functions
- Top level functions are curried to work with |>
- Most functions defined for `Signal`
- Can be lifted to work on `SignalProducers`

filter

```
func filter<T, E>(predicate: T -> Bool)
    (signal: Signal<T, E>)
    -> Signal<T, E>
```

map

```
func map<T, U, E>(transform: T -> U)
    (signal: Signal<T, E>)
    -> Signal<U, E>
```

mapError

```
func mapError<T, E, F>(transform: E -> F)
    (signal: Signal<T, E>)
    -> Signal<T, F>
```

flatten

```
func flatten<T, E>(strategy: FlattenStrategy)
    (producer: SignalProducer<SignalProducer<T, E>, E>)
    -> SignalProducer<T, E>
```

flatMap

```
func flatMap<T, U, E>(strategy: FlattenStrategy,
                      transform: T -> SignalProducer<U, E>)
    (producer: SignalProducer<T, E>)
    -> SignalProducer<U, E>
```

sampleOn

```
func sampleOn<T, E>(sampler: SignalProducer<(), NoError>)
    (producer: SignalProducer<T, E>)
    -> SignalProducer<T, E>
```

- Forwards the latest value from `producer` whenever `sampler` sends a Next event.

combineLatest

```
func combineLatest<A, B, Error>(a: SignalProducer<A, Error>,
                                b: SignalProducer<B, Error>)
    -> SignalProducer<(A, B), Error>
```

- Sends the latest value when either signal sends a value

Putting it all together

iOS Simulator - iPhone 6 - iPh...
Carrier
3:29 PM
Login

```
@IBOutlet weak var nameField: UITextField!
@IBOutlet weak var passwordField: UITextField!
@IBOutlet weak var loginButton: UIButton!
```

```
let name = nameField
    .rac_textSignal()
    .toSignalProducer()
    |> map { $0 as? String }
    |> ignoreNil
    |> discardError
```

```
extension UITextField {
    func textSignal () -> SignalProducer<String, NoError> {
        return self.rac_textSignal()
            .toSignalProducer()
            |> map { $0 as? String }
            |> ignoreNil
            |> discardError
    }
}
```

```
let name = nameField.textSignal()
let password = passwordField.textSignal()
let tap = loginButton
    .rac_signalForControlEvents(.TouchUpInside)
    .toSignalProducer()
    |> discardError
    |> map { _ in () }
```

```
let enabled = combineLatest(
        name |> map(not(isEmpty)),
        password |> map(not(isEmpty)))
    |> map({ $0 && $1 })
    |> map { NSNumber(bool: $0) as AnyObject? }
    |> discardError

DynamicProperty(object: loginButton, keyPath: "enabled") <~ enabled
```

```
login: (String, String) -> SignalProducer<User, NSError>

combineLatest(name, password)
    |> sampleOn(tap)
    |> flatMap(FlattenStrategy.Concat, login)
    |> on(error: showError)
    |> retryForever
    |> start(next: showUser)
```

Resources

- github.com/ReactiveCocoa/ReactiveCocoa
- www.quora.com/ReactiveCocoa
- blog.scottlogic.com/ceberhardt
- nomothetis.svbtle.com/an-introduction-to-reactivecocoa
- Have a look at Haskell to really learn FP

Conclusion

- FRP encapsulates side effects in a composable way
- Signals can replace many different OO patterns
- Still a lot of boilerplate code for bridging
- Most of it can be removed by extending existing APIs

Thanks!