Functional Programming Montréal Meetup & Swift Montréal Meetup

{ FRP in }

Fatih Nayebi | 2016-08-03 18:30 | Keatext

Agenda

- Introduction
- Why FRP?
- What is FRP?
- Building blocks of FRP
- Examples

Introduction

- Functional Reactive Programming is a programming paradigm that was created by **Conal Elliott**.
- FRP has been used for programming graphical user interfaces (GUIs), robotics, and music, aiming to simplify these problems by explicitly modelling time.
- **Reactive Programming**, focuses on asynchronous data streams, which we can listen to and react accordingly.
- **Functional Programming**, emphasizes calculations via mathematical-style functions, immutability and expressiveness, and minimizes the use of variables and state.

Why FRP?

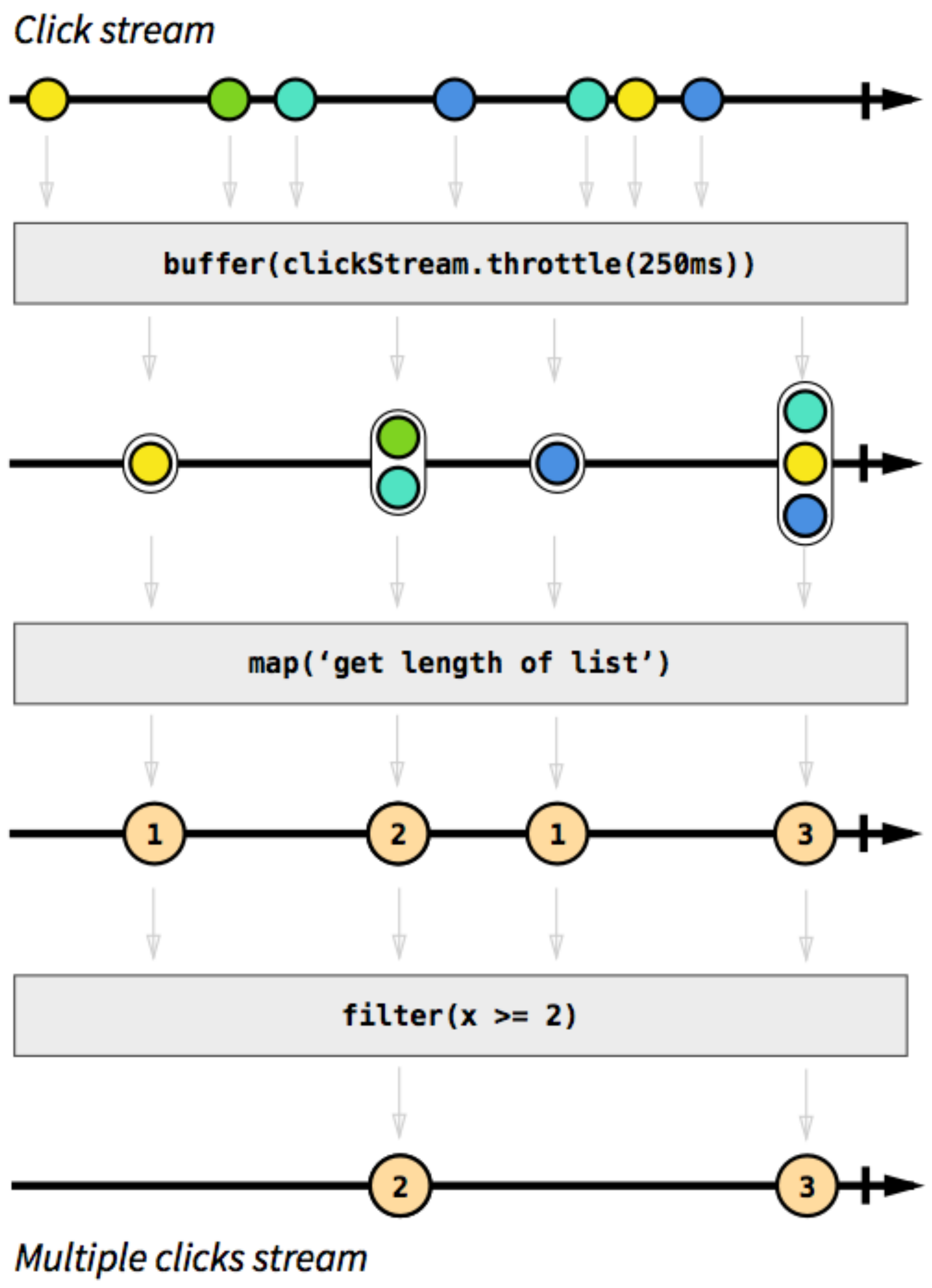
- Functional programming avoids immutability and side effects.
- Sometimes, the application should react to dynamic value/ data changes.
- Imperative programming captures these dynamic values only indirectly, through state and mutations.
- The complete history (past, present, and future) has no first-class representation.
- Only discretely-evolving values can be (indirectly) captured as the imperative paradigm is temporally discrete.
- FRP provides a way to handle dynamic value changes while still retaining the FP style.

Why FRP?

- FRP lets us reason about our program more effectively.
- Instead of worrying about state changing all over the place, we contain the ways in which it's allowed to change.
- Instead of worrying about which parts of our application might be affected, we also constrain the parts that care about any one change, and we are guaranteed that they can handle changes well.
- Instead of having to think about synchronous and asynchronous changes as two totally separate data flows, we can just handle them the same way, so there's less code we have to write.

What is FRP?

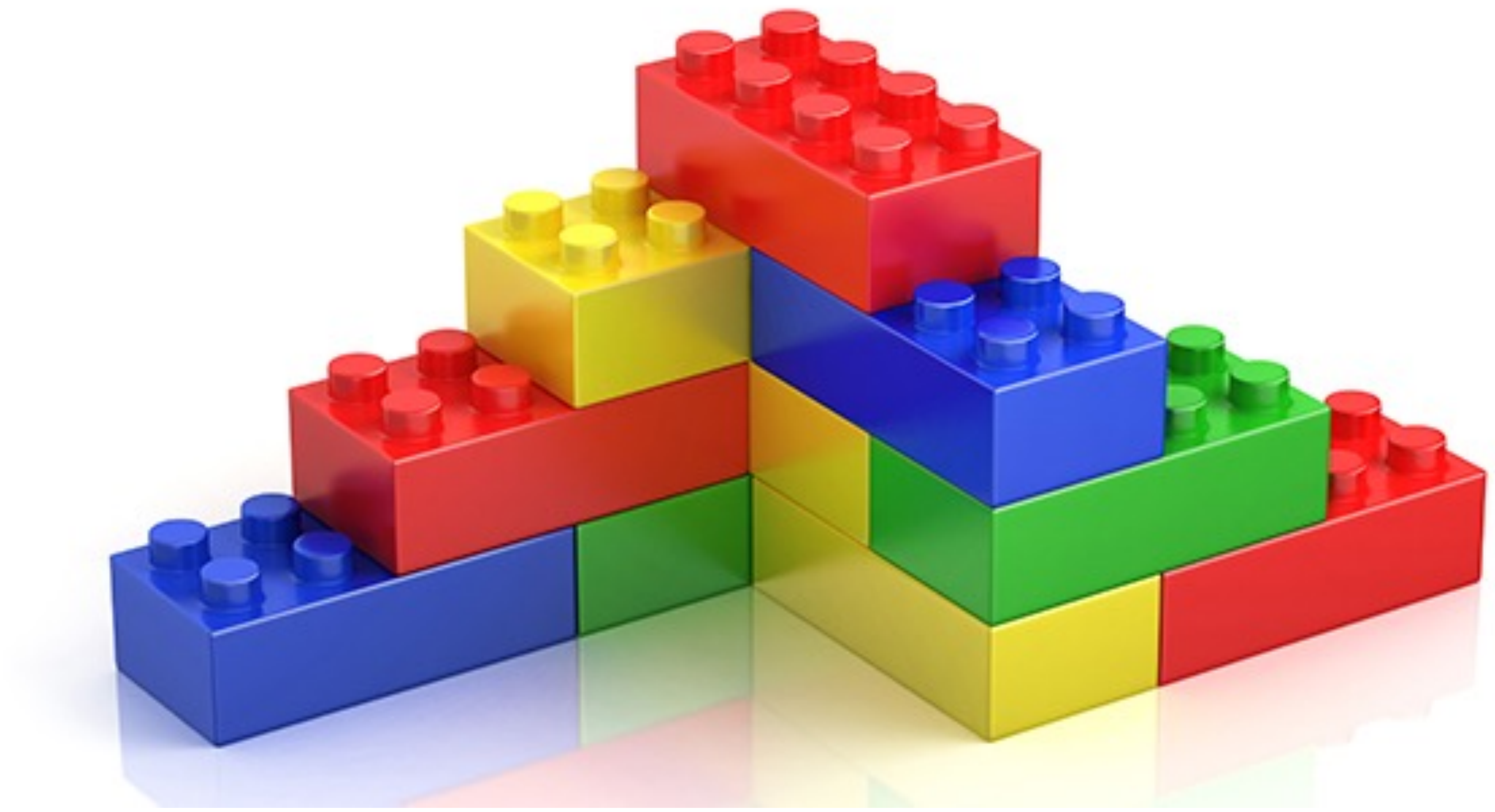
- According to Conal Elliot: tinyurl.com/zs6fblg
- A combination of FP and Reactive programming.
- Reactive programming makes it possible to deal with certain data types that represent values over time (time flow or event streams).
- Computations that involve these changing-over-time/evolving values will themselves have values that change over time. FRP captures these evolving values directly and has no difficulty with continuously evolving values.


Click stream
buffer(clickStream.throttle(250ms))
map('get length of list')
1
2
1
3
filter(x >= 2)
2
3
Multiple clicks stream

Reference: The introduction to Reactive Programming you've been missing

Very boring stuff?!

Building blocks of FRP



ReactiveCocoa

Signals

- Signals are **event streams** that send values over time that are already in progress.
- We can imagine them as pipes that send values without knowing about the previous values that they sent or future values that they are going to send.
- Signals can be composed, combined, and chained declaratively.
- Signals can **unify** all Cocoa common patterns for asynchrony and event handling:
    - Delegate methods
    - Callback blocks
    - Notifications
    - Control actions and responder chain events
    - Future and Promises
    - Key-value observing (KVO)

Pipes

- A signal that can be manually controlled is called pipe. In ReactiveCocoa, we can create a pipe by calling Signal.pipe().
- The pipe method returns signal and observer. The signal can be controlled by sending events to the observer.

Signal Producers

- Creates signals and performs side-effects.
- Can be used to represent operations or tasks such as network requests, where each invocation of start() will create a new underlying operation and allow the caller to observe the result.
- Unlike a signal, no work is started (and thus no events are generated) until an observer is attached, and the work is restarted for each additional observer.
- Starting a signal producer returns a disposable that can be used to interrupt/cancel the work associated with the produced signal.
- Signal producers can also be manipulated via operations such as map, filter, and reduce. Every signal operation can be lifted to operate upon signal producers instead, using the lift method.

Buffers

- A buffer is an optionally bounded queue for events.
- A buffer replays these events when new signals are created from SignalProducer.
- A buffer is created by calling SignalProducer.buffer().
- Similar to pipe, the method returns observer.
- Events sent to this observer will be added to the queue. If the buffer is already at capacity when a new value arrives, the oldest value will be dropped to make room for it.

Observers

- An observer is anything that observes or is capable of observing events from a signal.

- Observers can be implicitly created using the callback-based versions of the Signal.observe() or SignalProducer.start() methods.

Actions

- An action will do some work when executed with an input.

- Actions are useful in performing side-effecting work upon user interaction, such as when a button is clicked.

- Actions can also be automatically disabled based on a property, and this disabled state can be represented in a user interface by disabling any controls associated with the action.

Properties

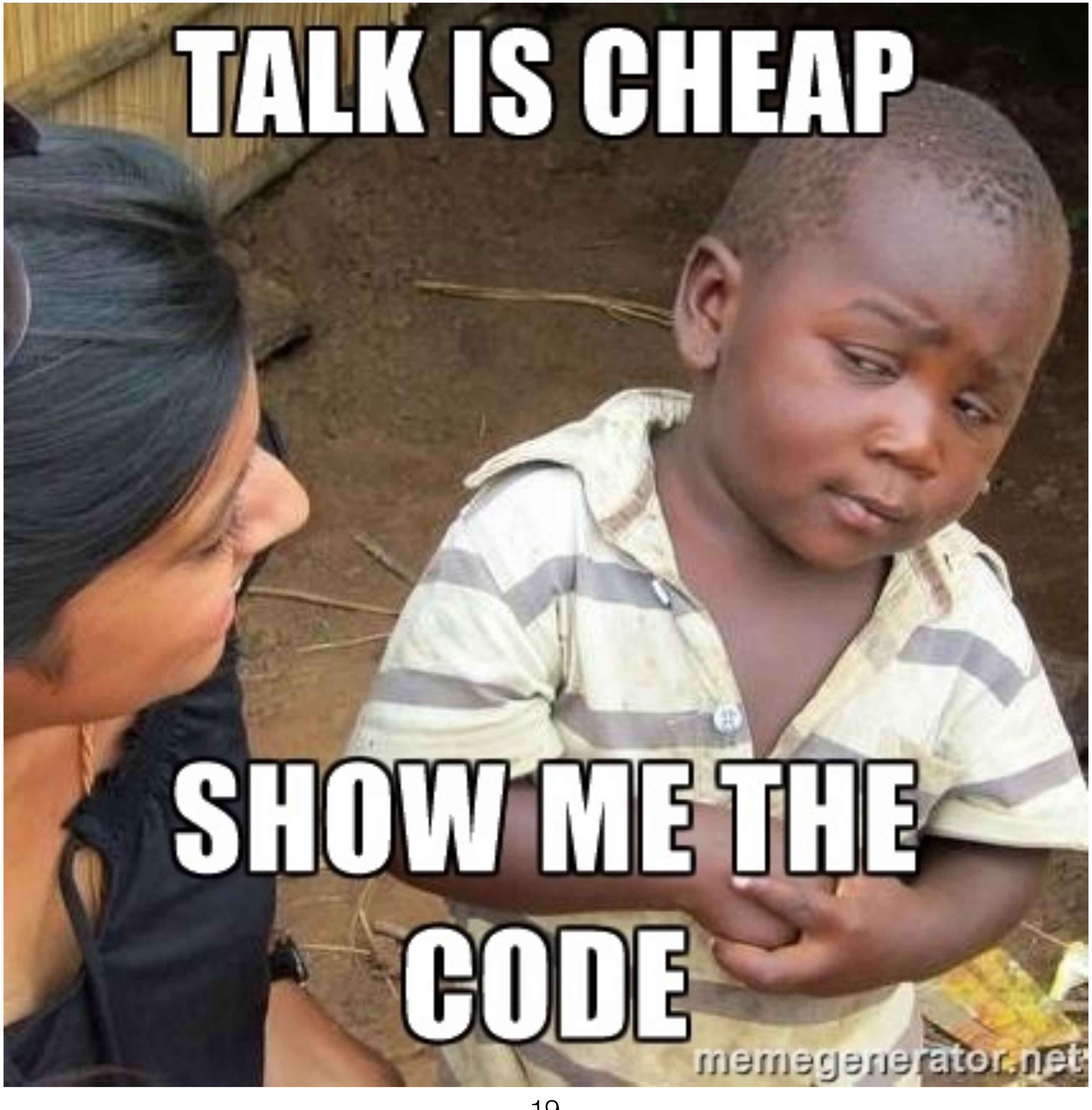
- A property stores a value and notifies observers about future changes to that value.
- The current value of a property can be obtained from the value getter.
- The producer getter returns a signal producer that will send the property's current value, followed by all changes over time.

Disposables

- A disposable is a mechanism for memory management and cancellation.
- When starting a signal producer, a disposable will be returned.
- This disposable can be used by the caller to cancel the work that has been started, clean up all temporary resources, and then send a final Interrupted event with regard to the particular signal that was created.

Schedulers

- A scheduler is a serial execution queue to perform work or deliver results upon.
- Signals and signal producers can be ordered to deliver events on a specific scheduler.
- Signal producers can additionally be ordered to start their work on a specific scheduler.
- Schedulers are similar to the **Grand Central Dispatch (GCD) queues**, but schedulers support **cancellation** via **disposables** and always execute serially.
- With the exception of ImmediateScheduler, schedulers do not offer synchronous execution. This helps avoid deadlocks and encourages the use of **signal** and **signal producer** operations instead of blocking work.
- Schedulers are also somewhat similar to NSOperationQueue, but schedulers do not allow tasks to be reordered or depend on one another.


TALK IS CHEAP
SHOW ME THE CODE
memegenerator.net

Example

- Let's suppose that we have an outlet and we want to observe its changes:

```
@IBOutlet weak var textFieldUserName: UITextField!
```

- We can create SignalProducer as follows:

```
let userNameSignalProducer = textFieldUserName.rac_textSignal().toSignalProducer.map {
    text in text as! String
}
```

- The **rac_textSignal** method is a ReactiveCocoa extension for UITextField that can be used to create the signal producer.
- Then, we can start our SignalProducer as follows:

```
userNameSignalProducer.startWithNext { results in
    print("User name:\(results)”)
}
```

- This will print any changes in our textField to the console.

Example: Email Validation

```
class EmailValidationDemoViewController: UIViewController {
    @IBOutlet weak var emailTextField: UITextField!
    @IBOutlet weak var submitButton: UIButton!

    let disposeBag = DisposeBag()

    override func viewDidLoad() {
        super.viewDidLoad()

        emailTextField.rx_text
            >- map (isEmail)
            >- submitButton.rx_subscribeEnabledTo
            >- disposeBag.addDisposable
    }
}
```

https://github.com/ashfurrow/FunctionalReactiveAwesome

Example: Online Search

- https://github.com/ReactiveCocoa/ReactiveCocoa#example-online-search

References

- **ReactiveCocoa** at GitHub (https://github.com/ReactiveCocoa/ReactiveCocoa)
- Packt Publishing - Swift 3 Functional Programming by Fatih Nayebi
- A farewell to FRP (http://elm-lang.org/blog/farewell-to-frp)
- The introduction to Reactive Programming you've been missing (tinyurl.com/mwhhx5l)
- https://realm.io/news/frp-ios-guide/
- The Swift Programming Language by Apple (swift.org)