Functional Reactive Programming with Reactive Cocoa 3

Presented By: Jeff Roberts
Email: jeff@nimbleNogginSoftware.com
Twitter: @JeffBNimble
GitHub: JeffBNimble

About Me

- I'm old (compared to you)
- Coding forever (~30 years)
- Professionally since 1986
- Consulting since 1991
- Riot since 2008 (we're hiring)
- Languages
  - RPG II, III, 400 [1986]
  - Smalltalk [1992]
  - Java [1996]
  - HTML/Javascript/CSS [2002]
  - Flex/AIR [2007]
  - iOS/Android [2012]
- Sharing
  - nimbleNoggin.io

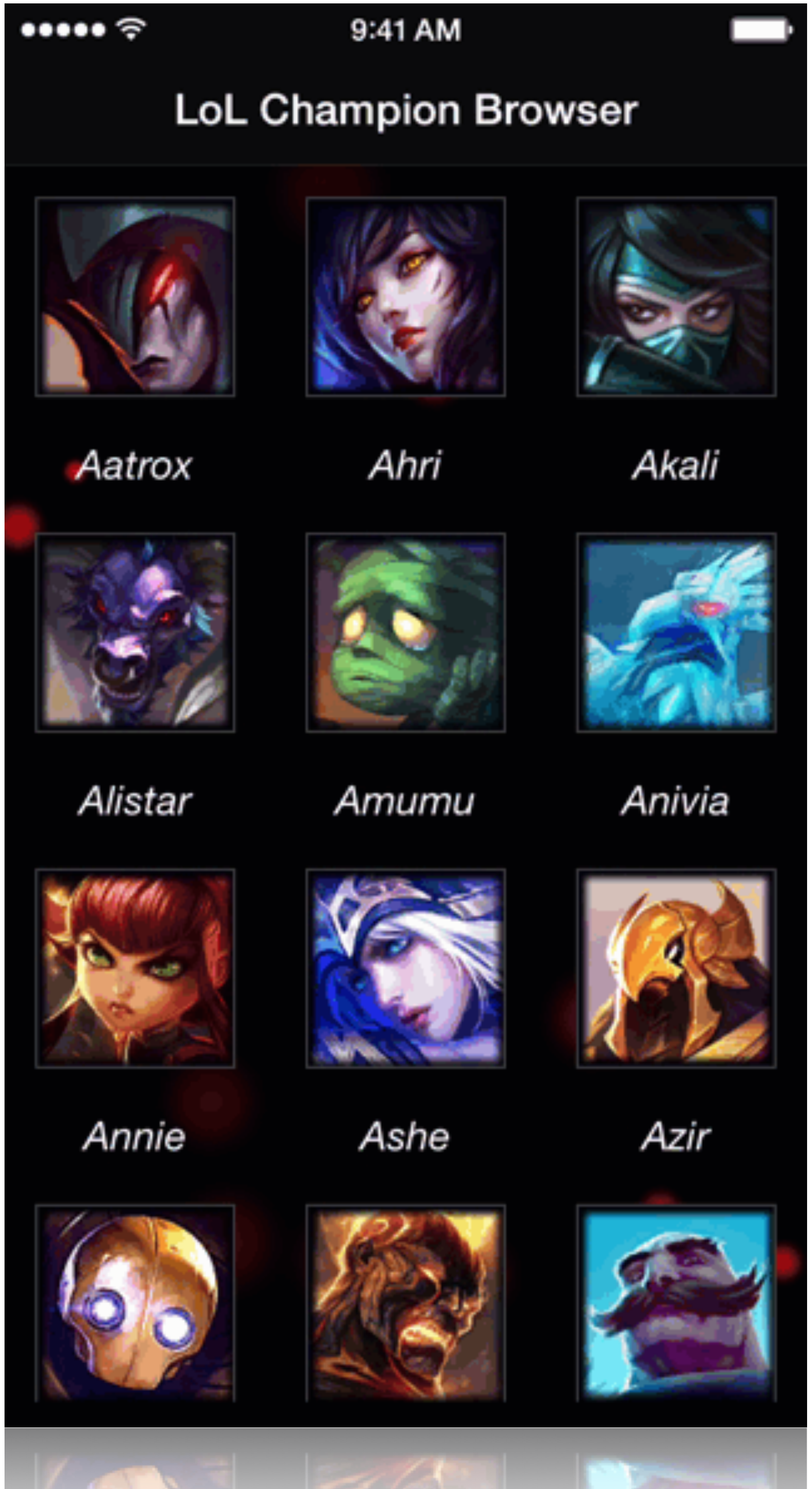
Agenda

- Gentle Intro - We will not cover everything
- Real world application demo
- Concepts & Terminology
- My thoughts and some Q&A (if we hurry)

Note: Xcode 7, Swift 2, Swift 2 branch of Reactive Cocoa
Note: Will NOT cover the RC SDK/UI Extensions

Demo Time Already

- LoL Champion Browser
- Written in Swift 2
- On GitHub
- Uses a lot of tech
- Reactive Cocoa 3

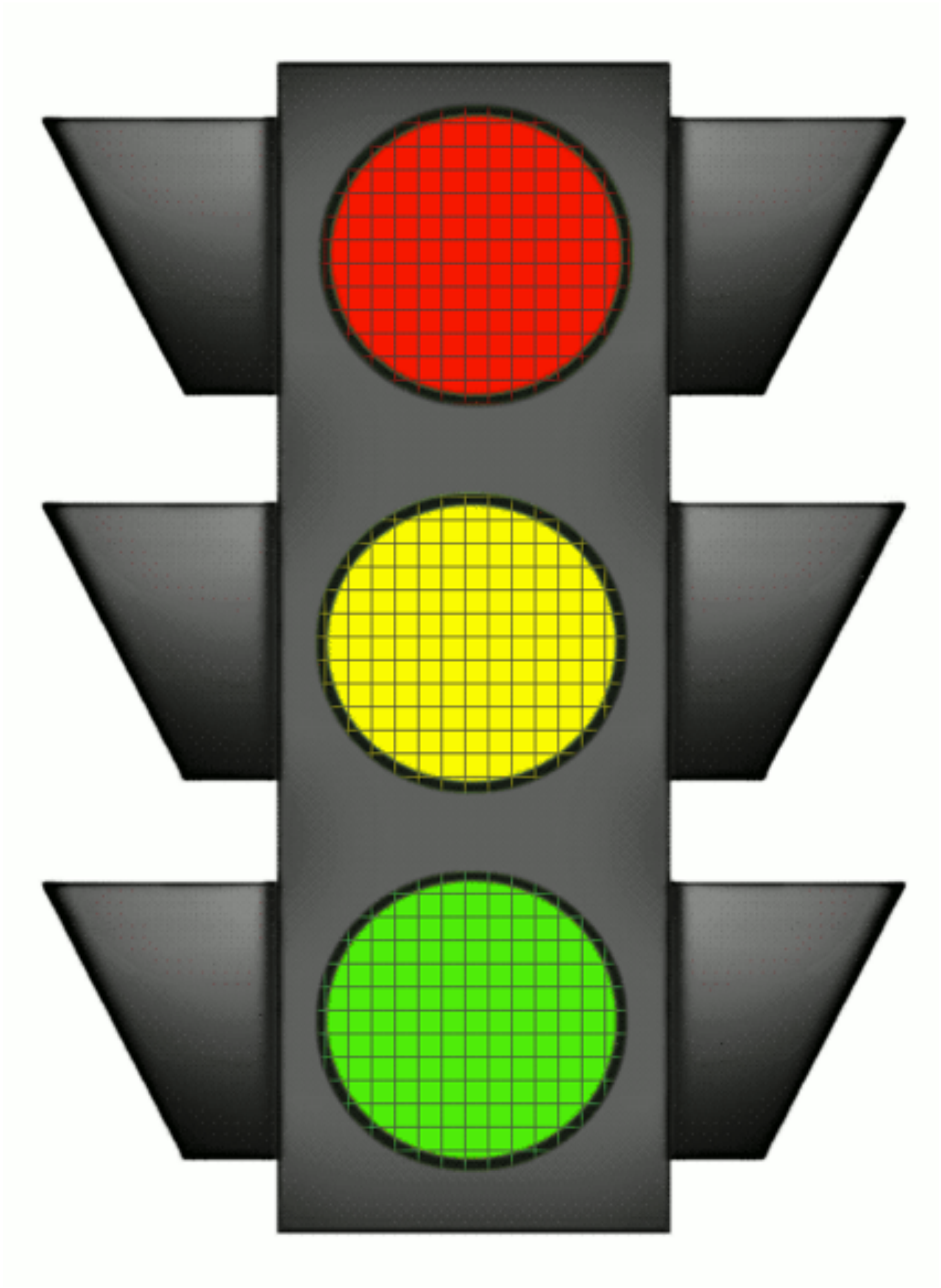


Reactive Cocoa 3

Starring

- The **Signal**
- A cast of **Operations**
- The **Signal Producer**
- The **Action**
- The **Schedulers**

Signal



- Indication something happened
- 0 to n "**next**" events, payload
- 0 or 1 "**completed**" event
- 0 or 1 "**error**" event, error payload
- 0 or 1 "**interrupted**" event
- 0 or more **observers**
- Observing returns a **Disposable**
- Dispose to **stop** observing

Creating a **Signal**

```
let (signal, sink) = Signal<String, NSError>.pipe()
```

Sink: a device or place for disposing of energy within a system, as a power-consuming device in an electrical circuit or a condenser in a steam engine.

dictionary.com

Sending Events

```
let (signal, sink) = Signal<String, NSError>.pipe()

// Send a “next” event
sendNext(sink, “Hello, World!”)

// Send an “error” event
sendError(sink, NSError())

// Send a “completed” event
sendCompleted(sink)
```

Events occur whether or not anything is observing
Referred to as “hot”
Always on

Observing a **Signal**

```
let (signal, sink) = Signal<String, NSError>.pipe()

// Observe the signal
signal.observeNext() { value in
    print("And the value is: \(value)")
}
signal.observeCompleted() {
    print("The signal is completed")
}
signal.observeError() { error in
    print("An error occurred: \(error)")
}

sendNext(sink, "Hello there")
sendCompleted(sink)
```

```
And the value is: Hello there
The signal is completed
```

Reactive Cocoa 3

Starring

- ~~The **Signal**~~
- A cast of **Operations**
- The **Signal Producer**
- The **Action**
- The **Schedulers**

A cast of **Operations**

Things that bolt onto a **Signal**
Most impact the “next” event

- Filter
- $\longrightarrow$ Alter/Manipulate/Map
- Alter the timing of
- Combine
- Chained
- Compose flows
- $f(x)$ Functional

A filter **Operation**

```
let (signal, sink) = Signal<FootballTeam, NSError>.pipe()

// Filter/Observe the signal
signal
.filter() { team in
    return team.wins > 10
}
.observeNext() { team in
    print("\(team.name) has \(team.wins) wins")
}
```

A map **Operation**

```
let (signal, sink) = Signal<FootballTeam, NSError>.pipe()

// Map/Observe the signal
signal
.map() { team in
    return team.members
}
.observeNext() { teamMembers in
    teamMembers.forEach() { member in
        print(“\(member.name) plays \(member.position)”)
    }
}
```

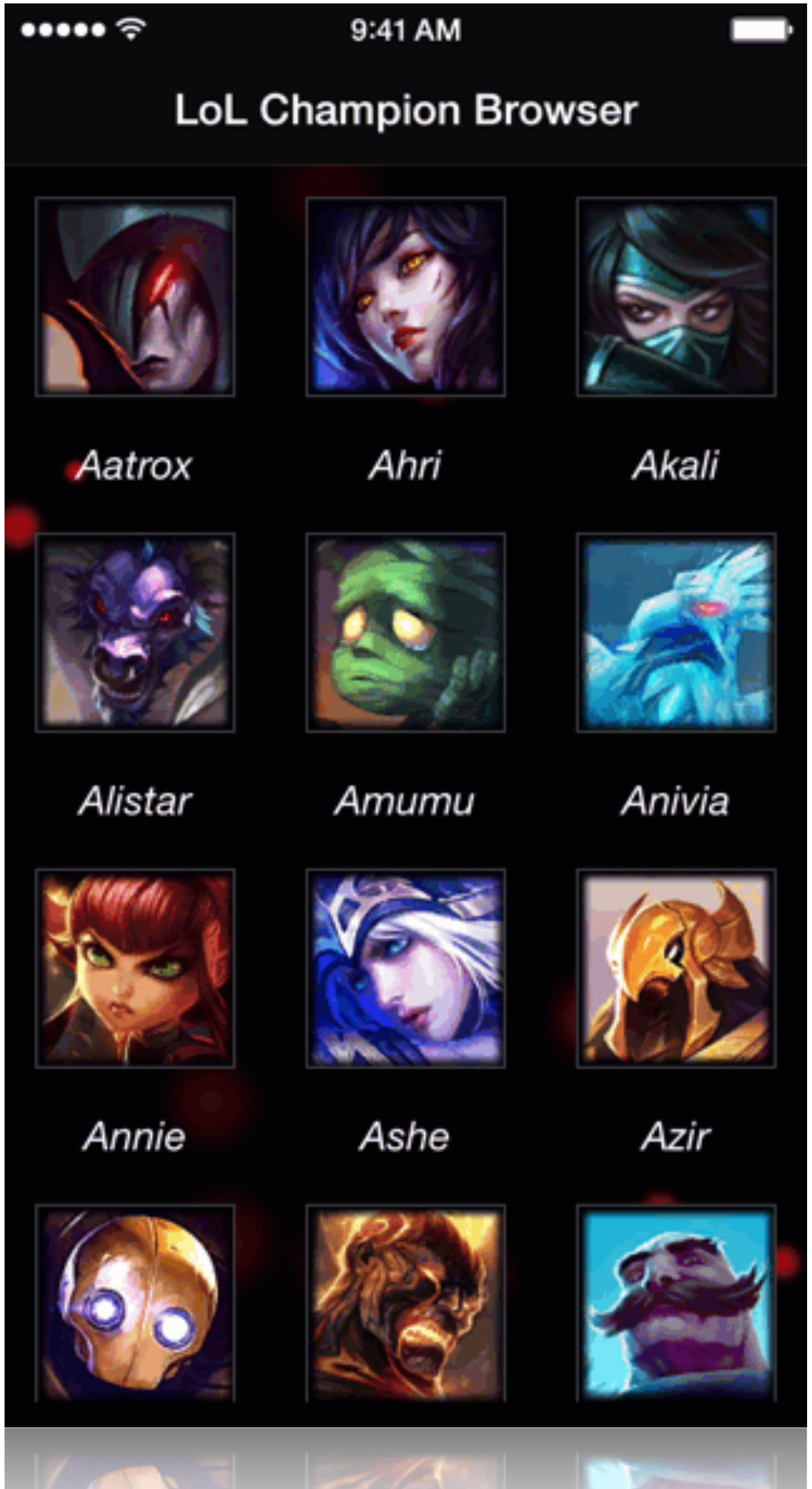
Chaining **Operations**

```
let (signal, sink) = Signal<FootballTeam, NSError>.pipe()

// Filter/Map/Observe the signal
signal
.filter() { team in
    return team.wins >= 10
}
.map() { team in
    return team.members.filter() { each in
        each.side == “defense”
    }
}
.observeNext() { teamMembers in
    teamMembers.forEach() { member in
        print(“\(member.name) plays \(member.position)”)
    }
}
```

Let’s look at some code!

- Magic
- Randomly changes color
- On orientation change
- When navigating back



OMG! **Operations**

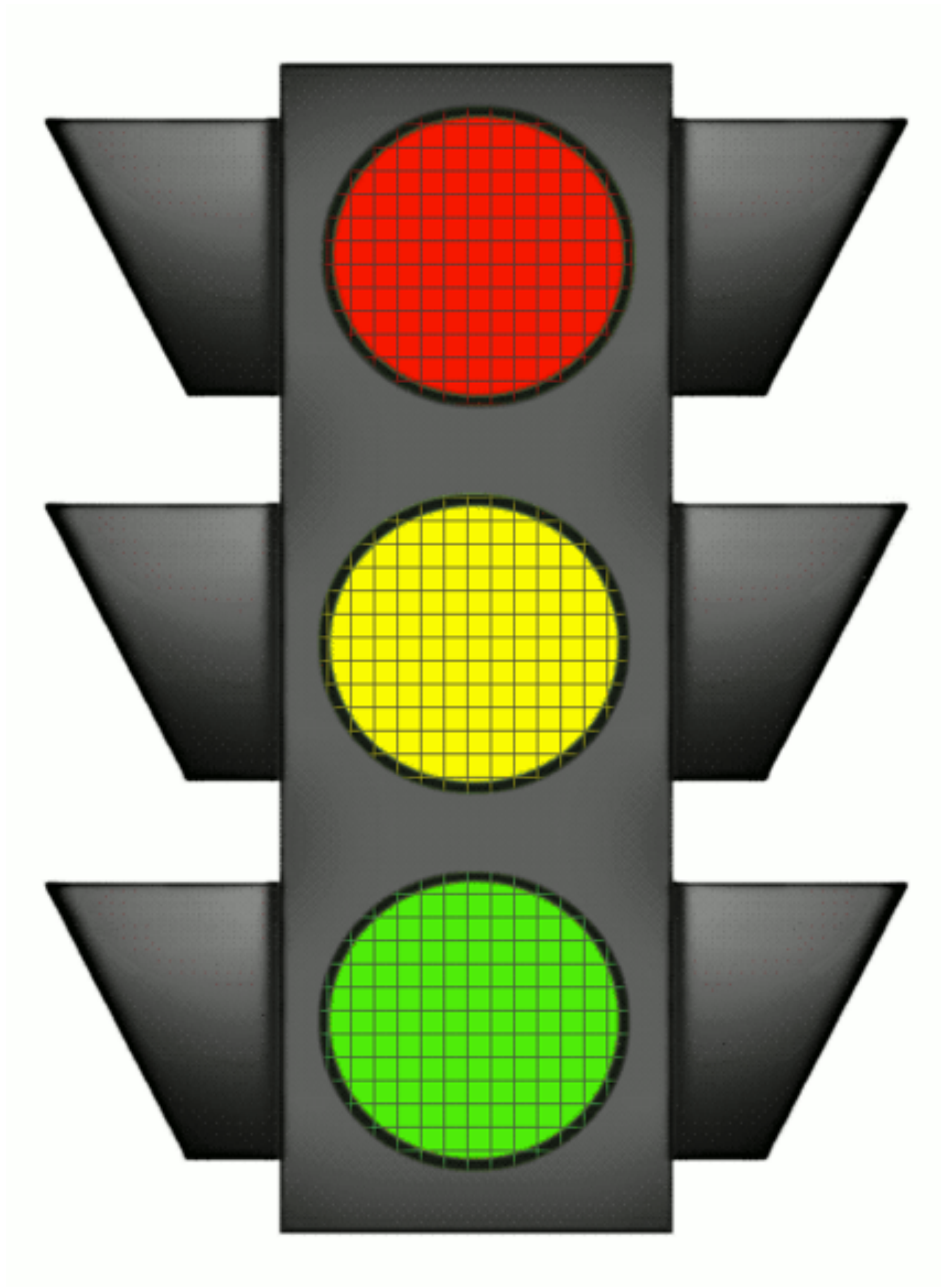
- filter()
- map()
- ignoreNil()
- take()
- collect()
- delay()
- skip()
- sampleOn()
- takeUntil()
- combinePrevious()
- reduce()
- scan()
- skipRepeats()
- skipWhile()
- takeUntilReplacement()
- takeLast()
- takeWhile()
- attempt()
- attemptMap()
- throttle()
- timeoutWithError()

Reactive Cocoa 3

Starring

- ~~The **Signal**~~
- ~~A cast of **Operations**~~
- The **Signal Producer**
- The **Action**
- The **Schedulers**

SignalProducer



- Identical to Signal, except…
- Must first be started
- **Next, Completed, Error** events
- Operations
- Cold (nothing happens until start)
- Creates a Signal when started
- Most commonly used, IMO
- I prefer in the context of **Action**s

Creating a **SignalProducer**

```
let signalProducer = SignalProducer<String, NSError>() {
    sink, disposable in
    // Do something here like…
    sendNext(sink, “Hello, World”)  // and/or
    sendCompleted(sink)
}
```

But, nothing happens here, until…
We call start

```
let disposable = signalProducer.startWithNext() {
    value in
    print(“Well \(value)”)
}
```

Well Hello, World

Reactive Cocoa 3

Starring

- ~~The **Signal**~~
- ~~A cast of **Operations**~~
- ~~The **Signal Producer**~~
- The **Action**
- The **Schedulers**

Action

- Like a Command
- Specifies **Input**, **Output** & **Error** type
- Invoke **apply()** passing **Input**
- Returns a **SignalProducer**
- **Next** payload is **Output** type
- Then can be **start**ed
- Used to perform tasks, reactively
- Like network request, queries, etc

Creating an **Action**

```
let getSkinCountAction = Action<String, Int, NSError>() {
    championKey in
    return SignalProducer<Int, NSError>() {
        sink, disposable in
        let count = // Run a query that gets row count
        sendNext(sink, count)
        sendCompleted(sink)
    }
}
```

- Nothing happens until **apply()** is called
- Returns **SignalProducer<Int, NSError>**
- Call **start** to “run” the action

Running an **Action**

```
let getSkinCountAction = Action<String, Int, NSError>() {
    championKey in
    return SignalProducer<Int, NSError>() {
        sink, disposable in
        let count = // Run a query that gets row count
        sendNext(sink, count)
        sendCompleted(sink)
    }
}
getSkinCountAction.apply(“annie”)
.startWithNext() { skinCount in
    print(“There are \(skinCount) skins”)
}
```

- Call apply() on the action and pass Input
- Returns a SignalProducer<Output, Error>
- Call start() to “run” the Action

Reactive Cocoa 3

Starring

- ~~The **Signal**~~
- ~~A cast of **Operations**~~
- ~~The **Signal Producer**~~
- ~~The **Action**~~
- The **Schedulers**

Schedulers

- RC3 approach to concurrency
- Based upon dispatch_queues
- UI, Queue, Immediate Schedulers
- SignalProducer.**startOn()**
- Signal/SignalProducer.**observeOn()**

Creating a background **QueueScheduler**

```
let dbQueue = dispatch_queue_create("dbQueue",
    DISPATCH_QUEUE_SERIAL)
let dbScheduler = QueueScheduler(queue: dbQueue,
    name: "io.nimbleNoggin.LoLBookOfChamps.dbQueue")
```

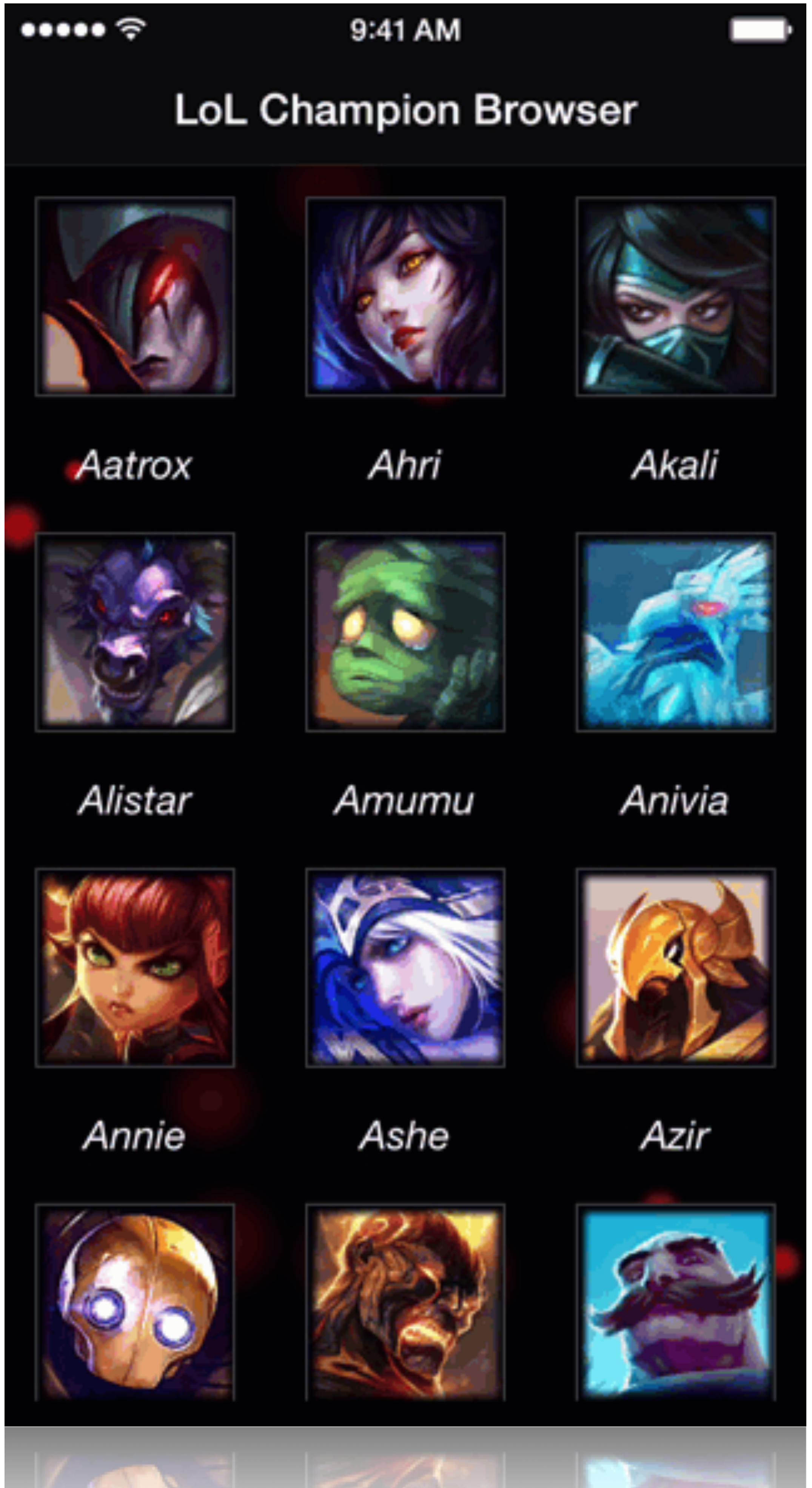
- Create a dispatch queue
- Create a scheduler
- Use it

Using **QueueSchedulers**

```
let getSkinCountAction = Action<String, Int, NSError>() {
    championKey in
    return SignalProducer<Int, NSError>() {
        sink, disposable in
        let count = // Run a query that gets row count
        sendNext(sink, count)
getSkinCountAction.apply("annie")
    .startOn(dbScheduler)
    .observeOn(UIScheduler())
    .startWithNext() { skinCount in
        print("There are \(skinCount) skins")
    }
```

Time to look at more code!

- Getting data
  - Champions
  - Champion Skins
- Asynchronous
- Actions, SignalProducers



Reactive Cocoa 3

Starring

- ~~The **Signal**~~
- ~~A cast of **Operations**~~
- ~~The **Signal Producer**~~
- ~~The **Action**~~
- ~~The **Schedulers**~~