This morning we're going to talk about reactive programming. We'll cover some of the what, why, and how, hopefully with a bend towards grasping the fundamentals.

We'll have some simple iOS examples using the open source library, ReactiveCocoa, but you should be able to take the presented lessons and apply them to all sorts of platforms and environments, ranging from fat GUI clients (Android, web browser, the Mac) to less user facing distributed systems.

Thanks to my friend Andrew Sardone, who made most of these slides for a talk he gave in January (but which I helped write!)

dzombak@nytimes.com

chris dzombak

iOS @ NY Times

@cdzombak

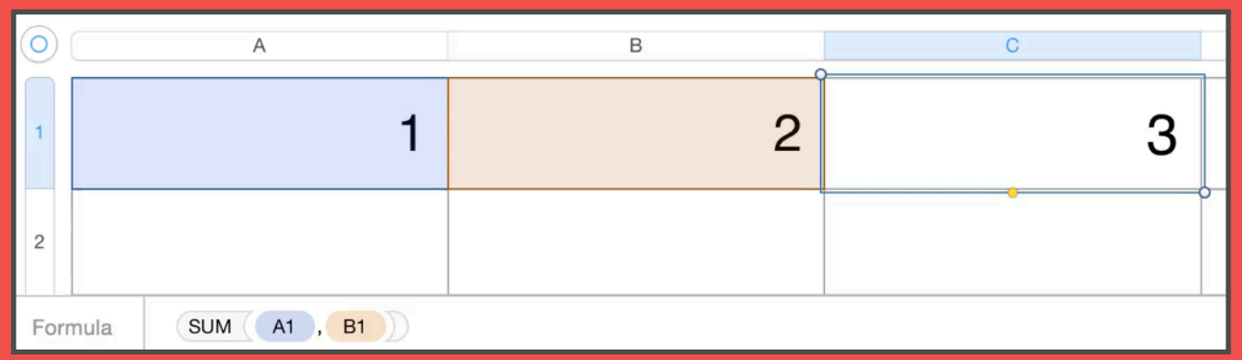
github.com/cdzombak

To start off, we'll ask ourselves what *is* reactive programming.

We can turn to the textbook definition, one that's routinely pointed to to start the conversation

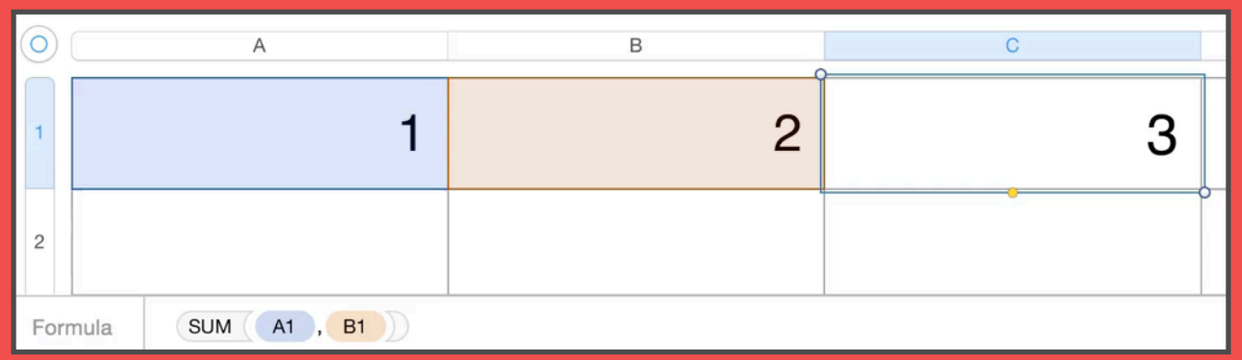
> ... a programming paradigm oriented around data flows and the propagation of change

reactive |rē'aktiv|



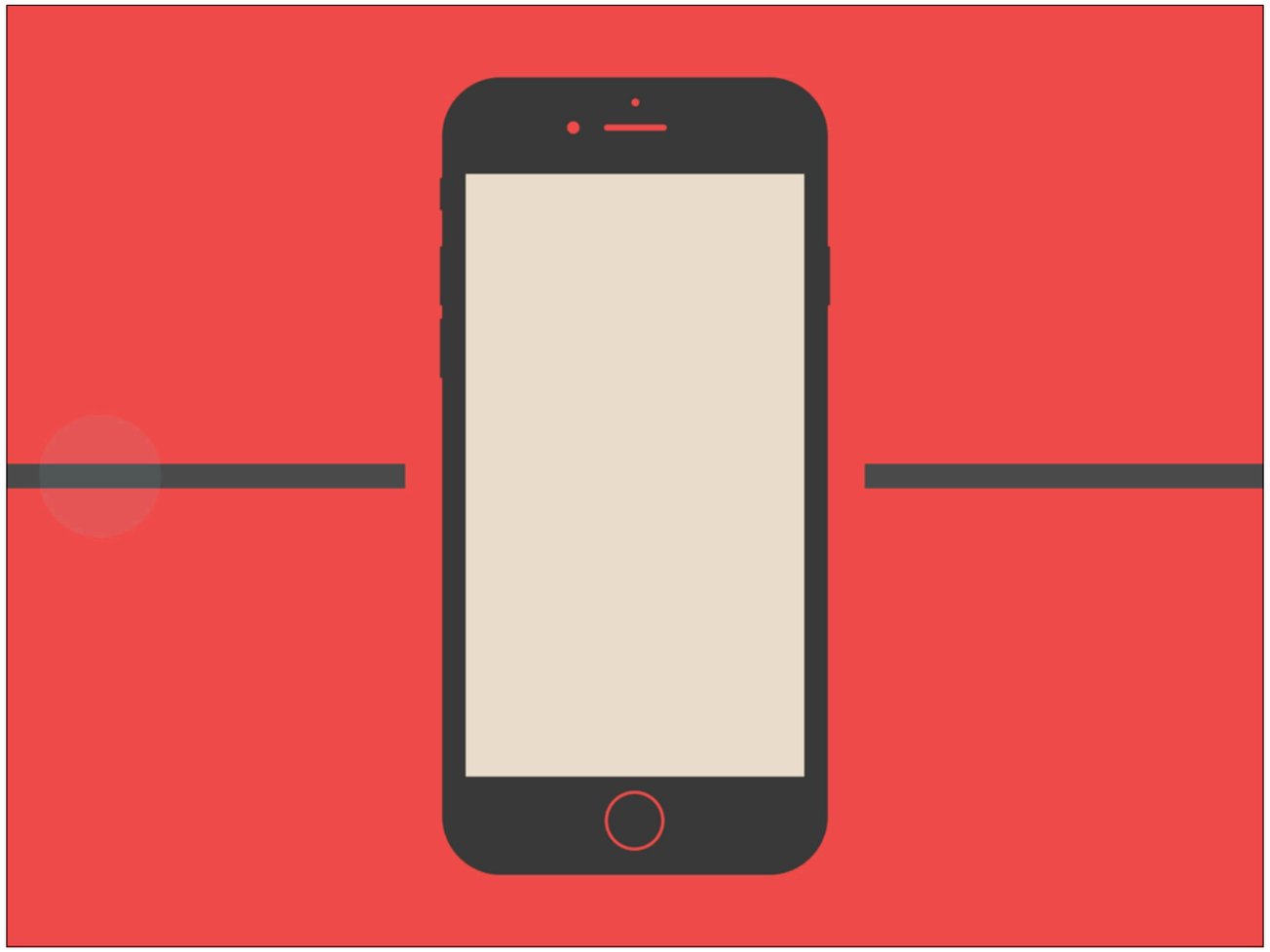
A common analogy people cite when it comes to explaining reactive programming is the lowly spreadsheet.

You set up relationships of cells, and when you change a value all dependent values automatically update

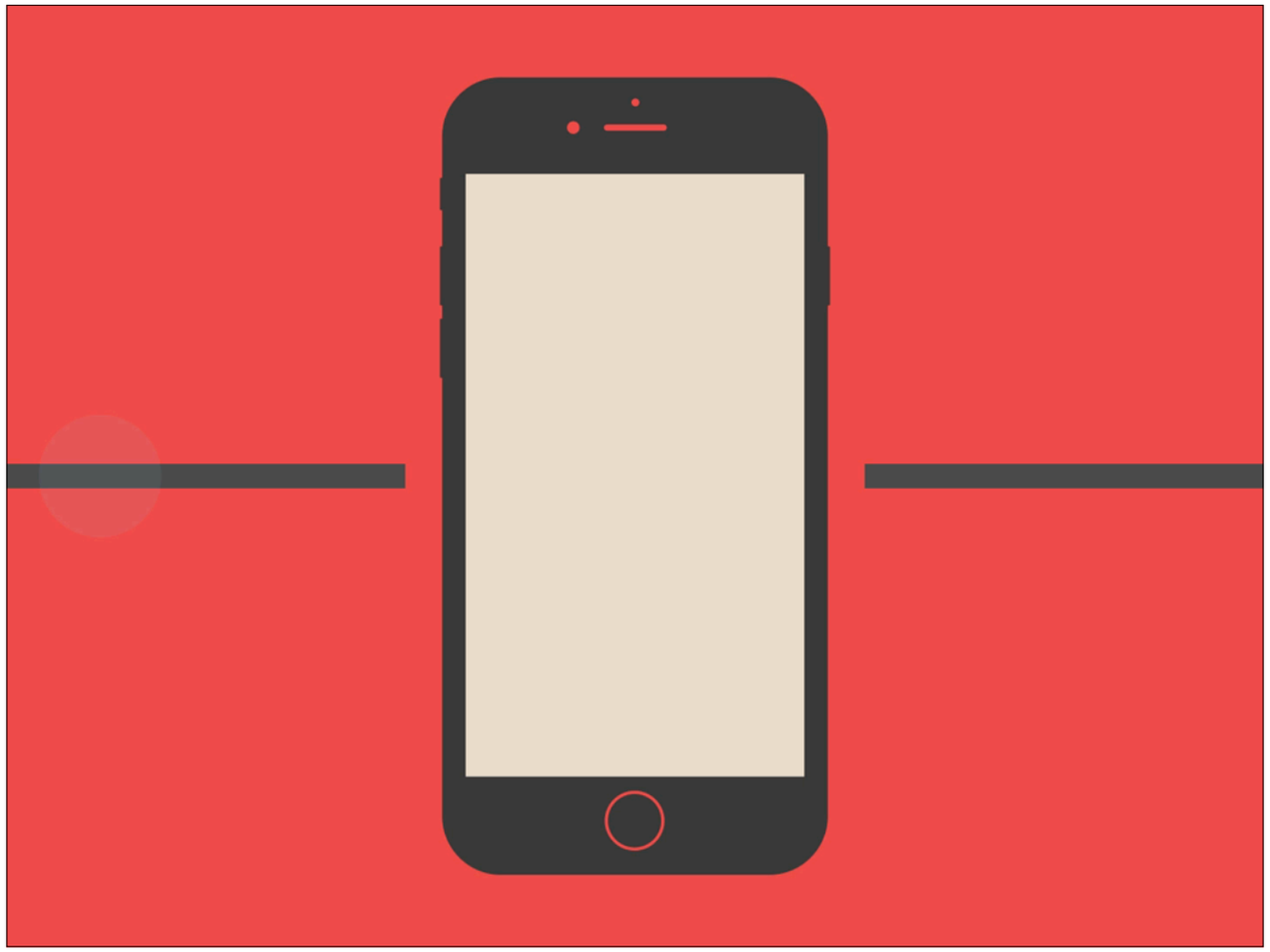


A common analogy people cite when it comes to explaining reactive programming is the lowly spreadsheet.

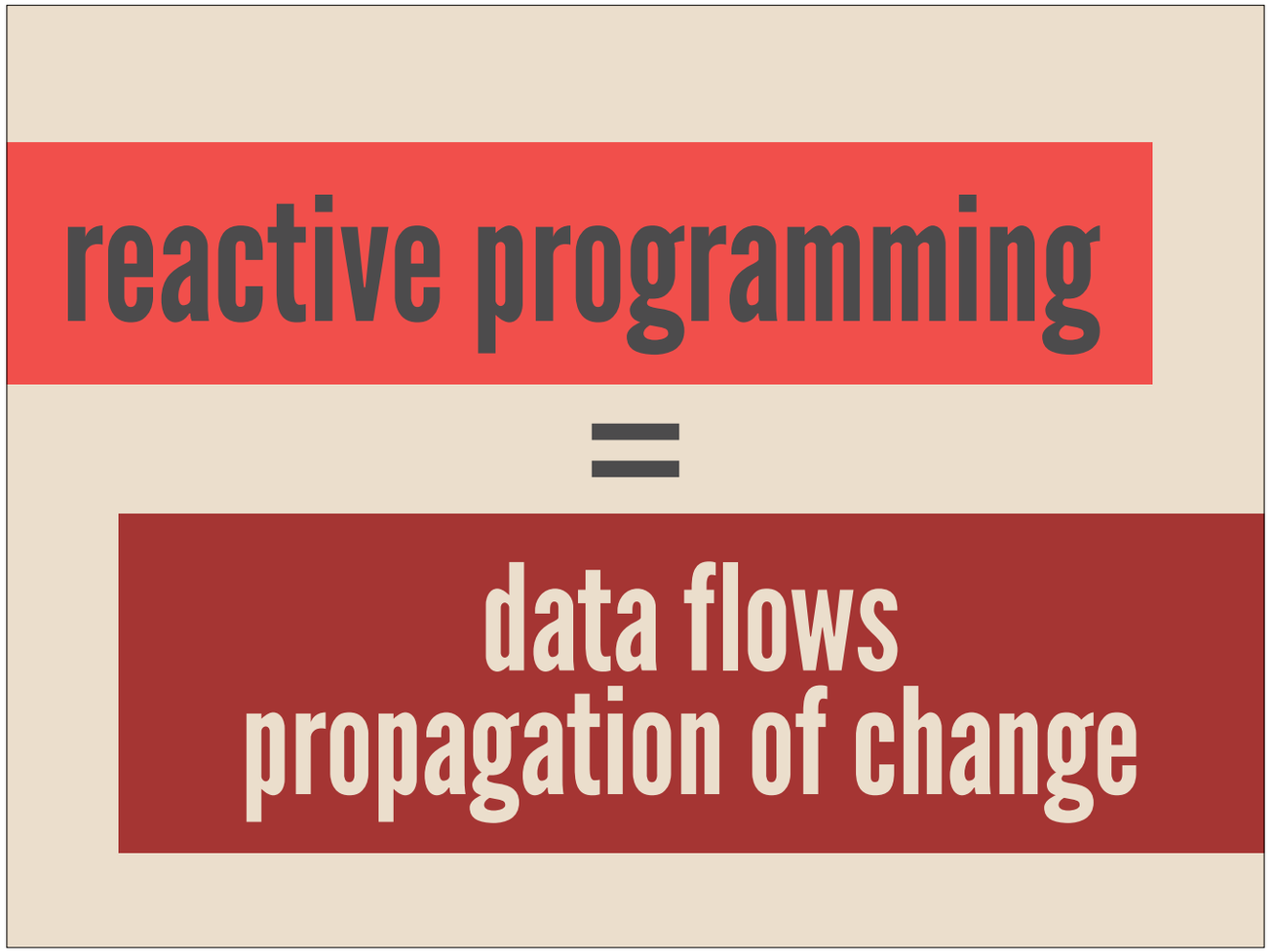
You set up relationships of cells, and when you change a value all dependent values automatically update



Just like a spreadsheet, reactive programming is about setting up data relationships within your app, and watching the changes to all of your dependencies propagate through the app

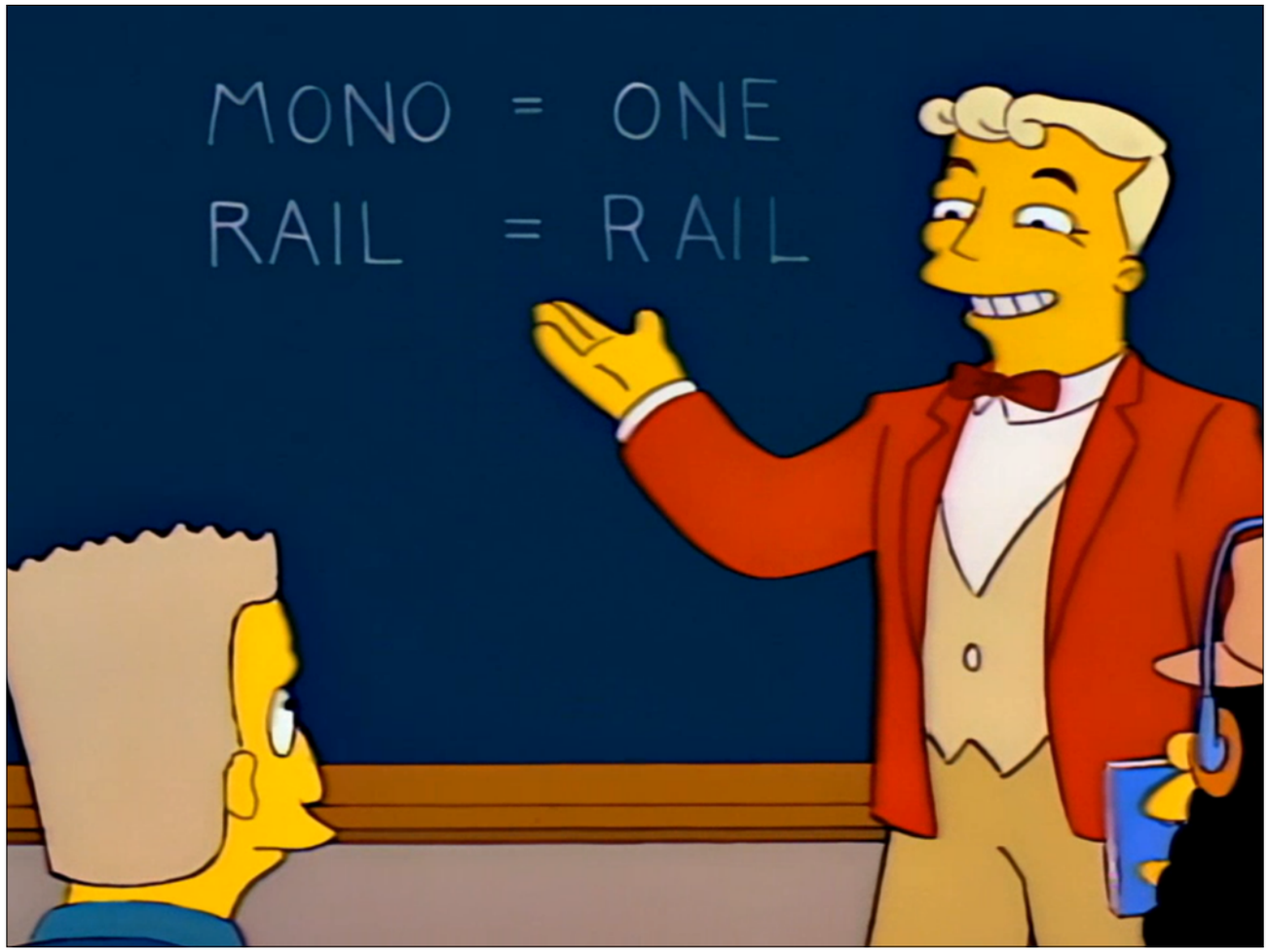


Just like a spreadsheet, reactive programming is about setting up data relationships within your app, and watching the changes to all of your dependencies propagate through the app



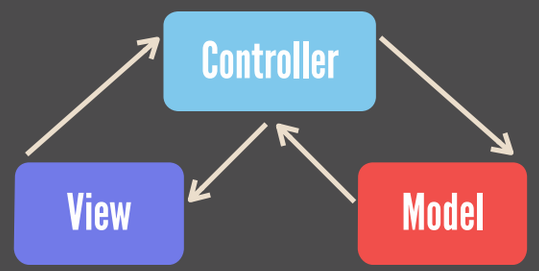
So, reactive programming is all about data flows and the propagation of change

But frankly, this kind of sounds like B.S., and is a bit reductive

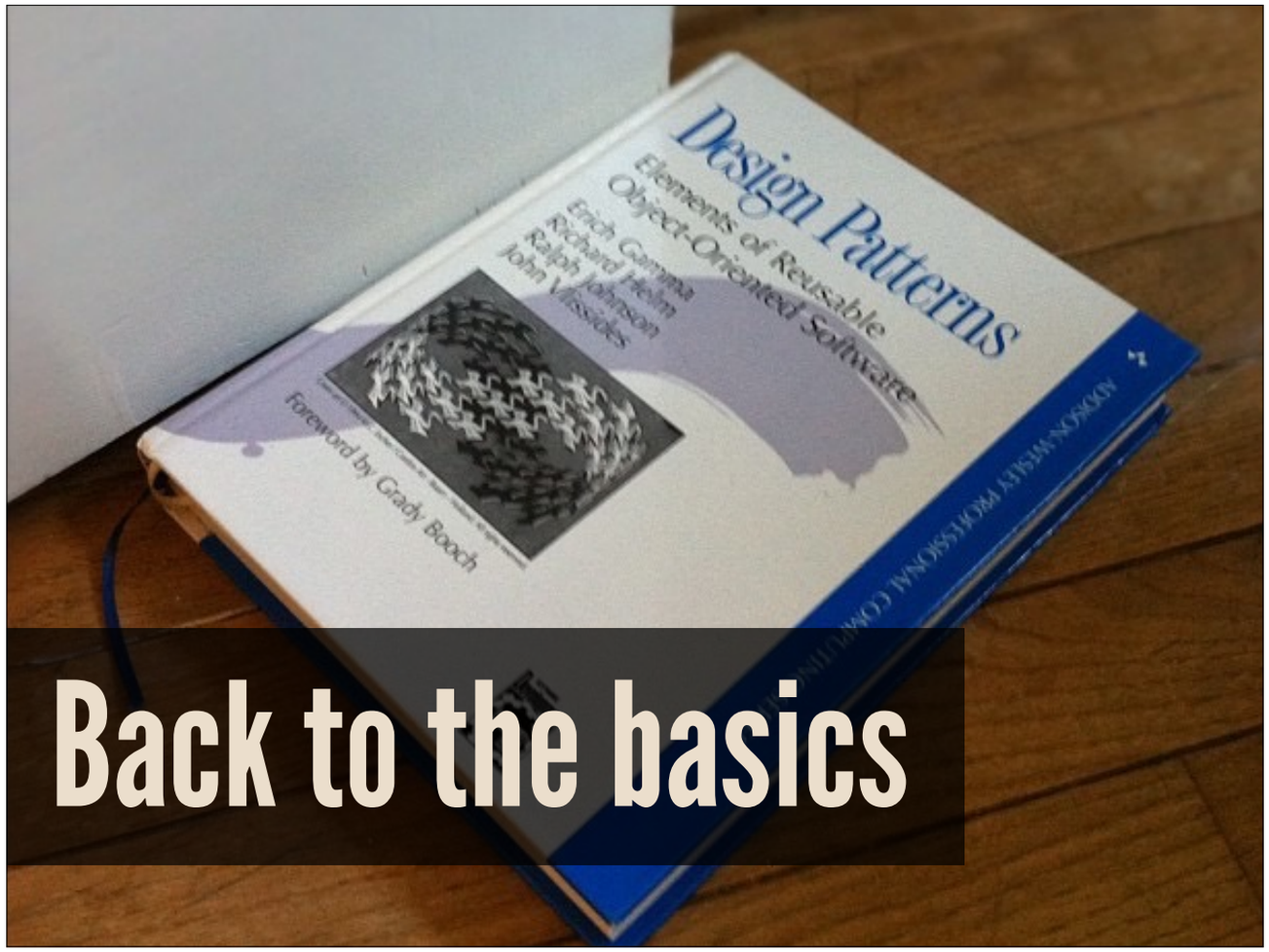


this is a reductive definition; it’s like saying a monorail is one rail

we all
respond to flowing data
and react to changes

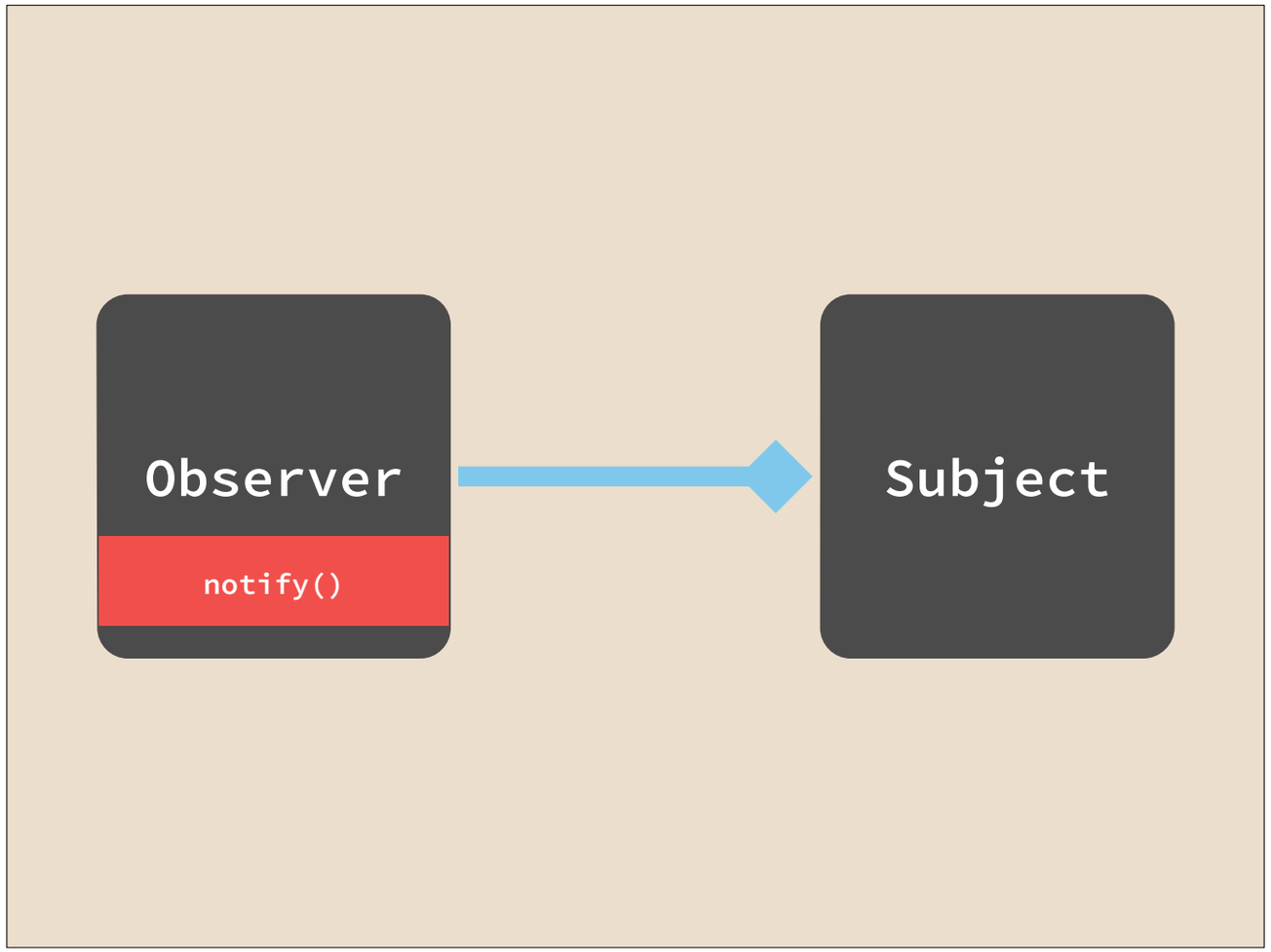


How is reactive programming different from…



Let’s get to the basics, and note that reactive programming, at first, isn’t all that different than our previous options.

It builds heavily off of the observer pattern, which we all know from the classic Gang of Four ‘Design Patterns’ book.



The observer pattern

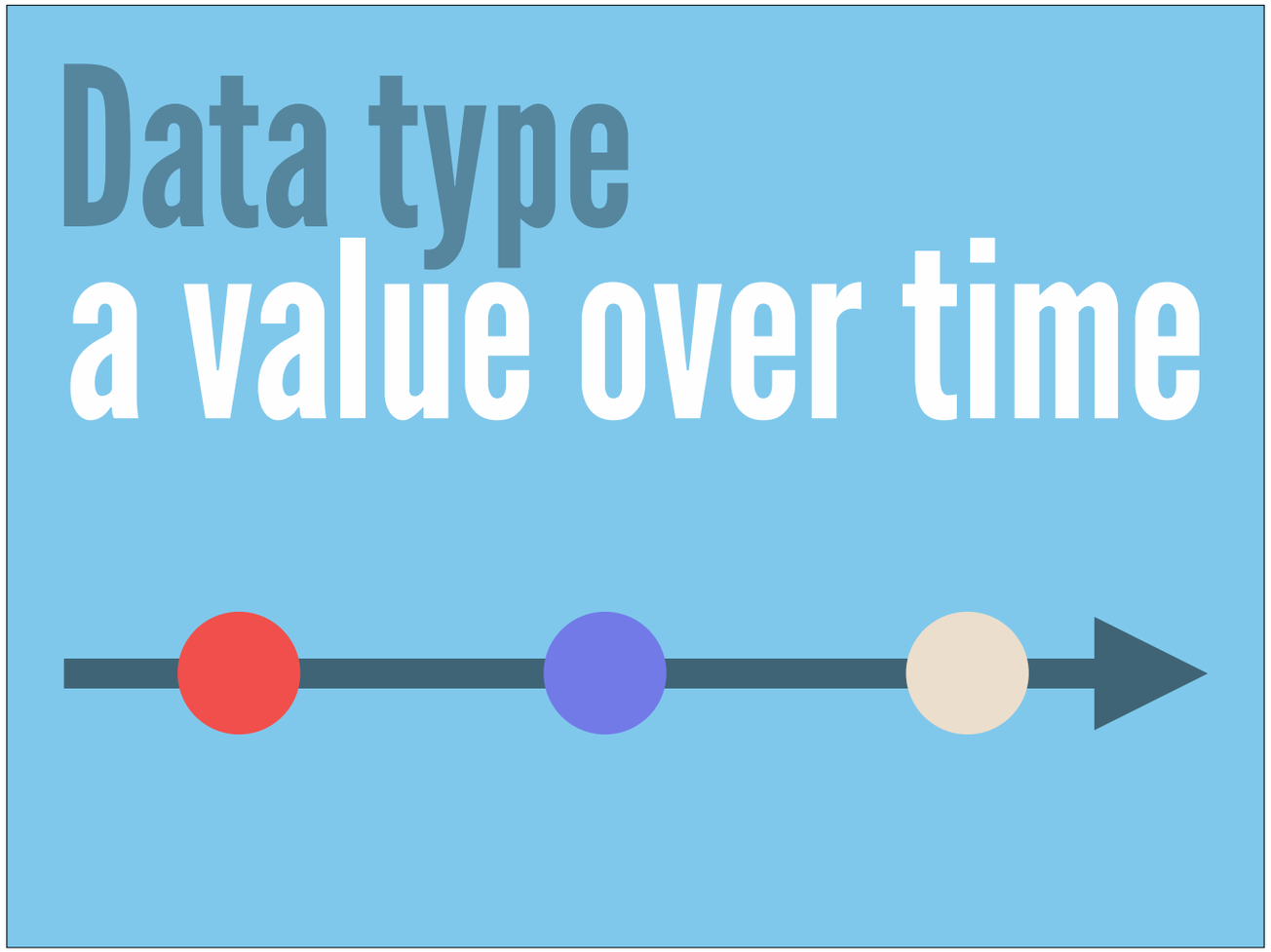
But reactive programming takes the observer pattern and cranks it up to 11, and it does so using……

(data streams)

Let’s start with something that’s just as abstract, but hopefully will give us a better jumping off point.

Reactive programming is all about data streams, and they’re a pretty simple concept.

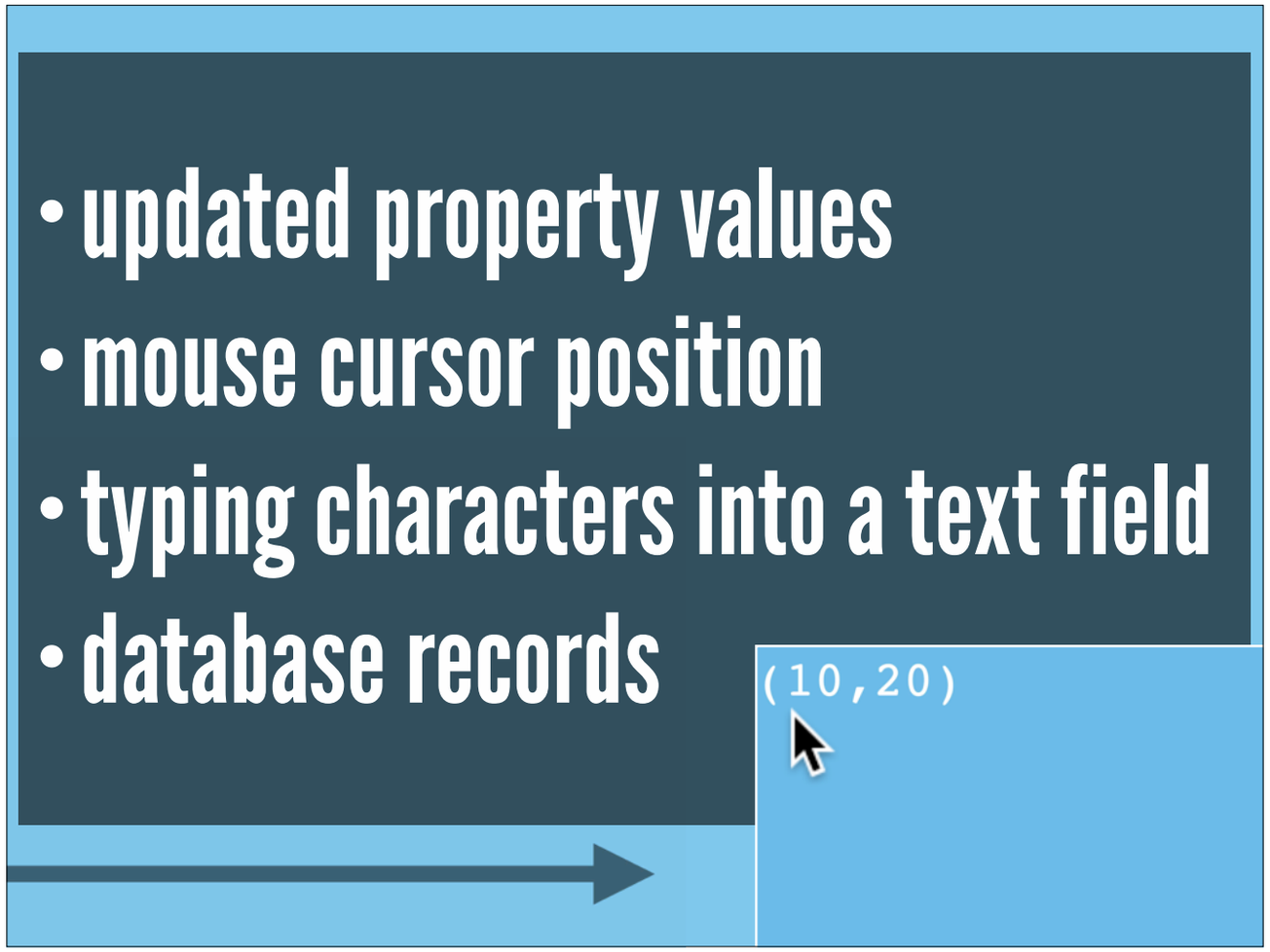
So what are data streams?



A simple data type that represents a single value over time.

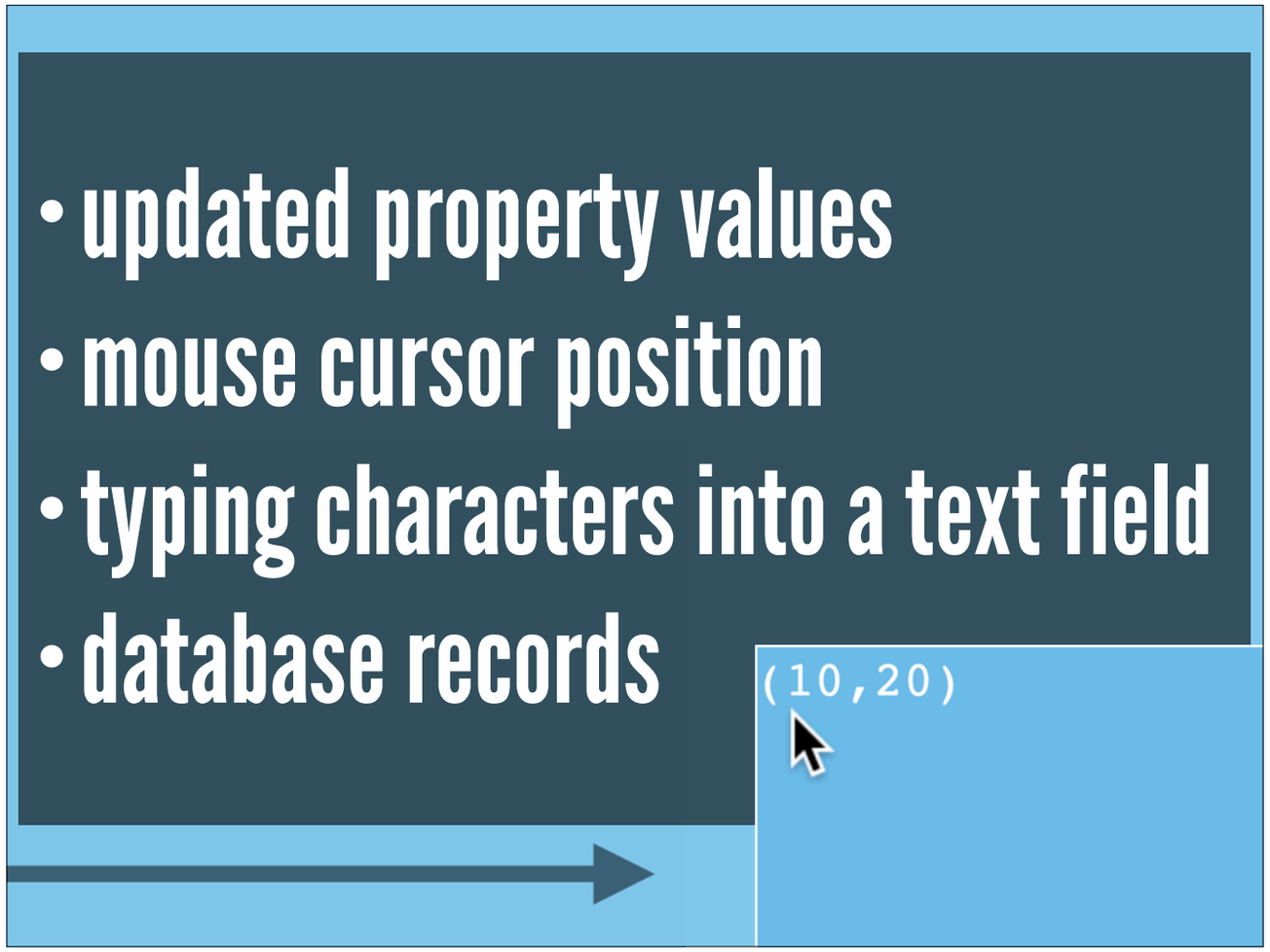
In the Observer pattern, what we're observing are changes, and stream puts the concept of 'change' front and center.

We see a lot of things in the lifecycle of our applications have different values over time, so they can be modeled with streams



- A variable that you reassign to different values
- UI properties like the position of your mouse cursor on screen
- User input like typing in a text field
- Database records
- Your Twitter feed

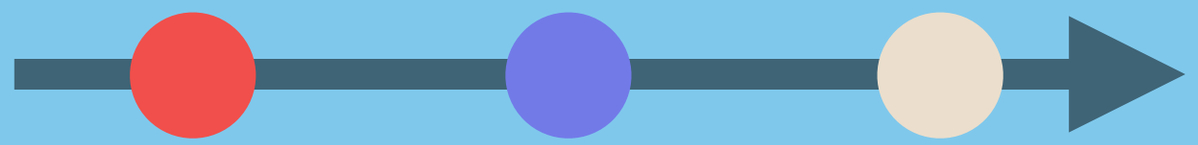
From the perspective of the observer pattern, they’re emitting new values for someone to observe

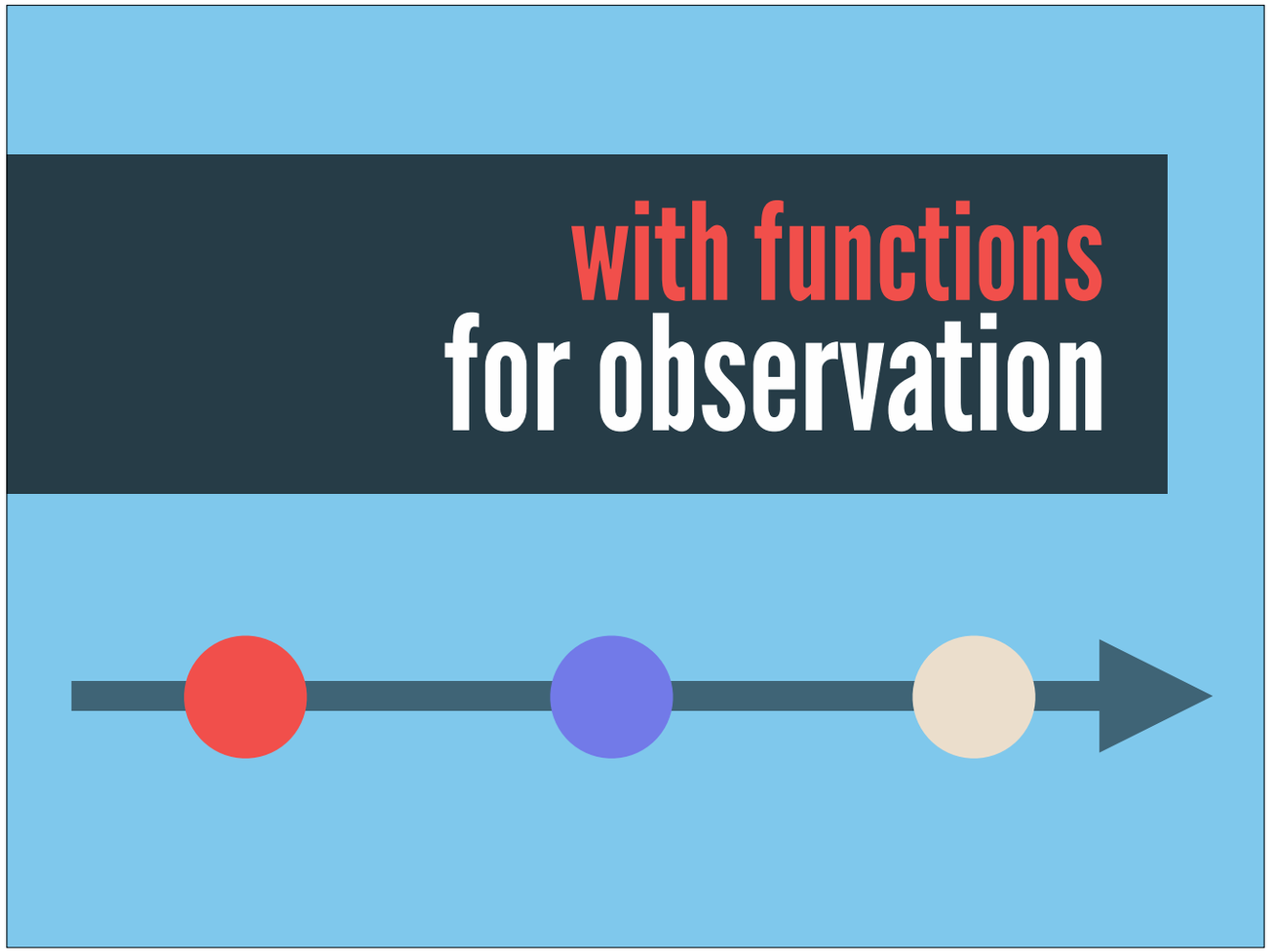


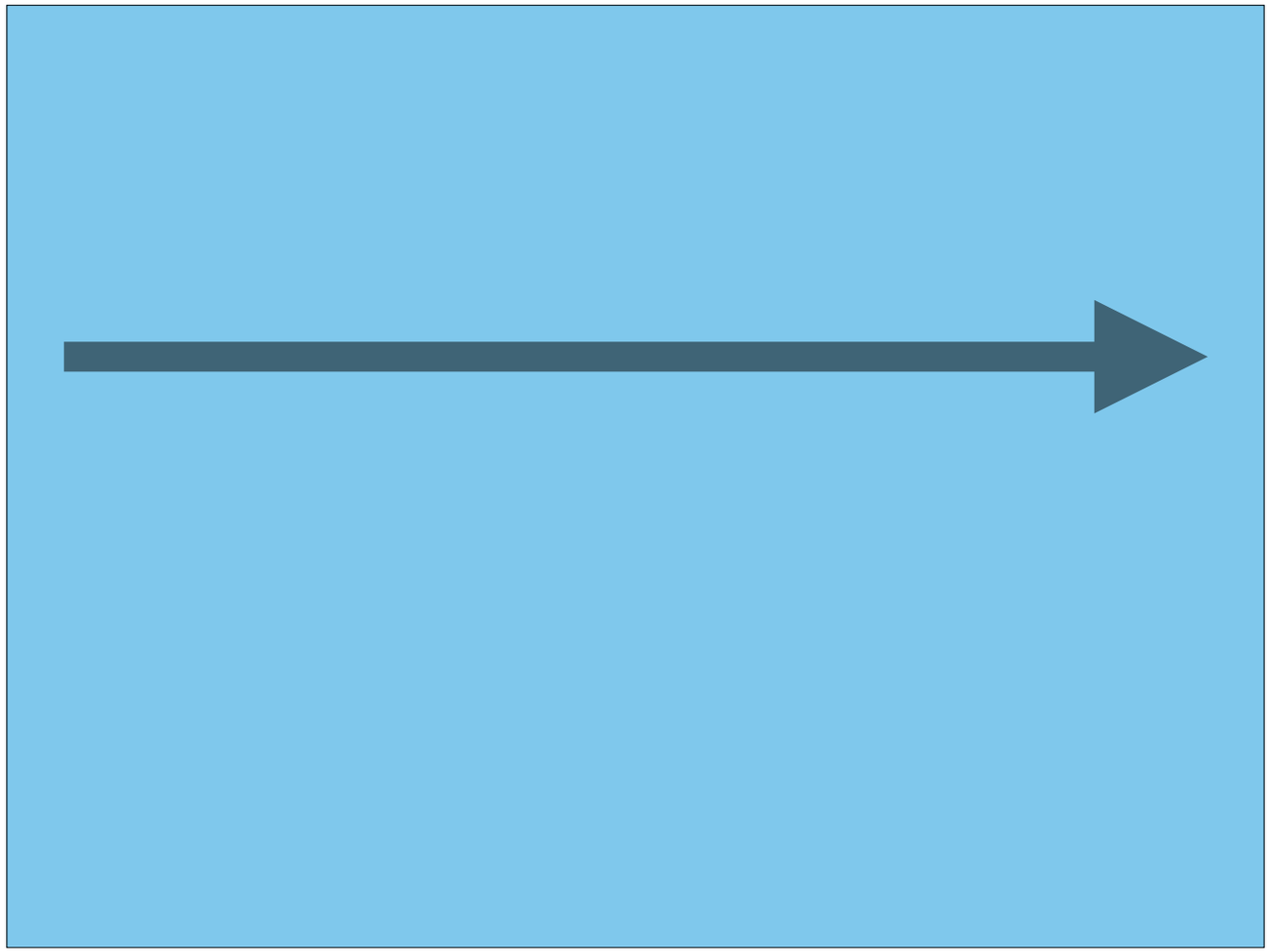
- A variable that you reassign to different values
- UI properties like the position of your mouse cursor on screen
- User input like typing in a text field
- Database records
- Your Twitter feed

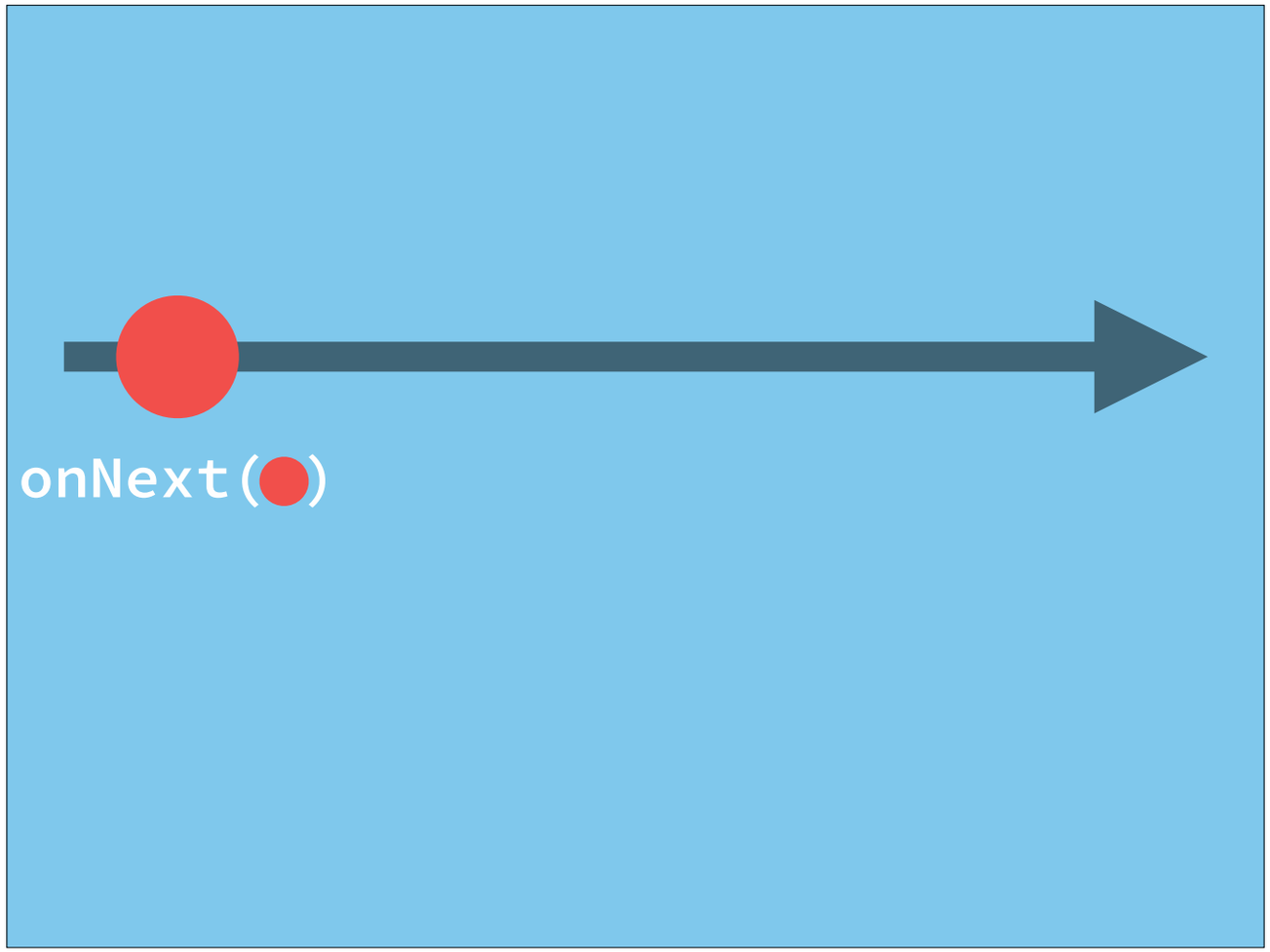
From the perspective of the observer pattern, they’re emitting new values for someone to observe

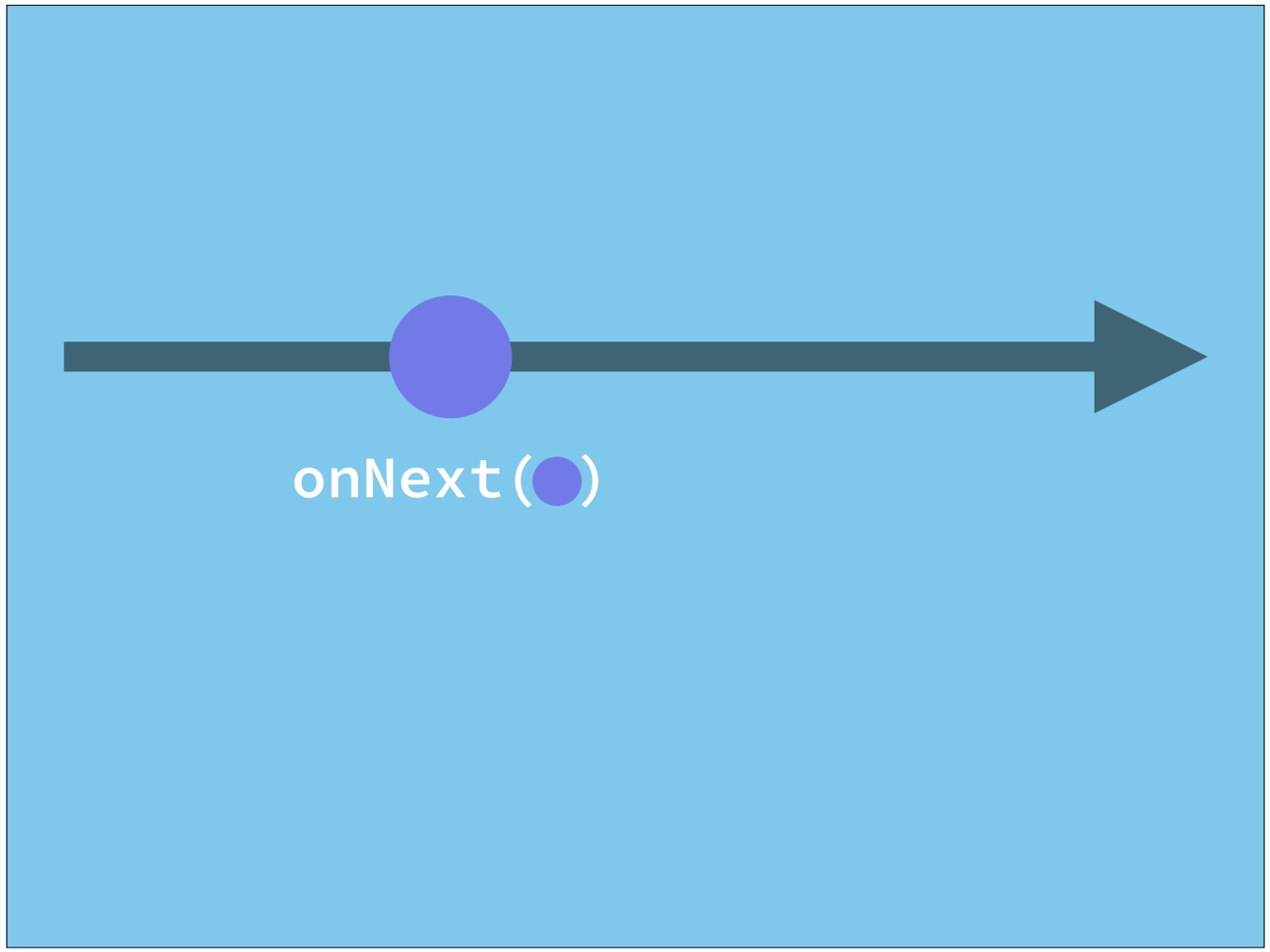
Collection of emitted events
ordered in time

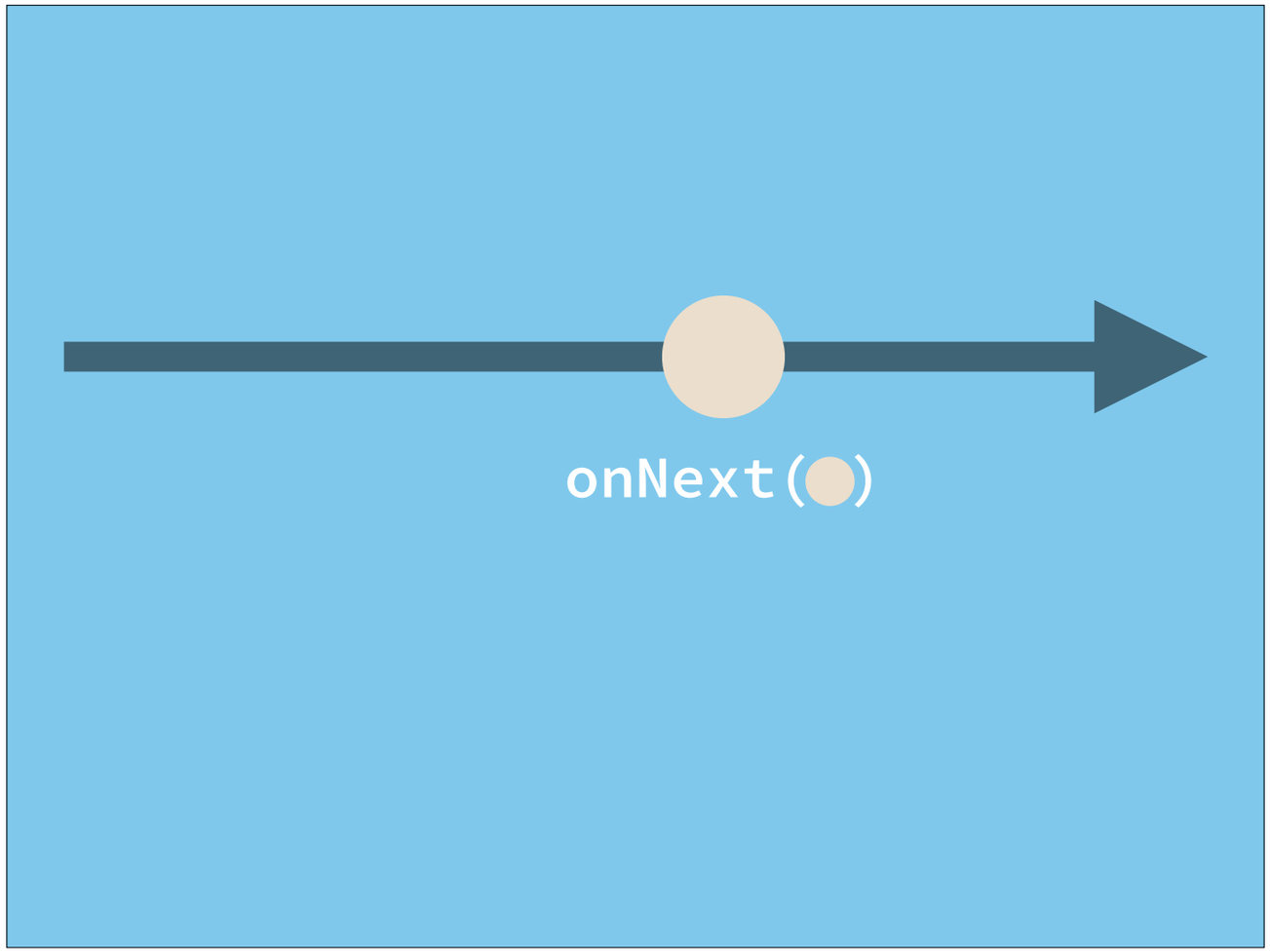


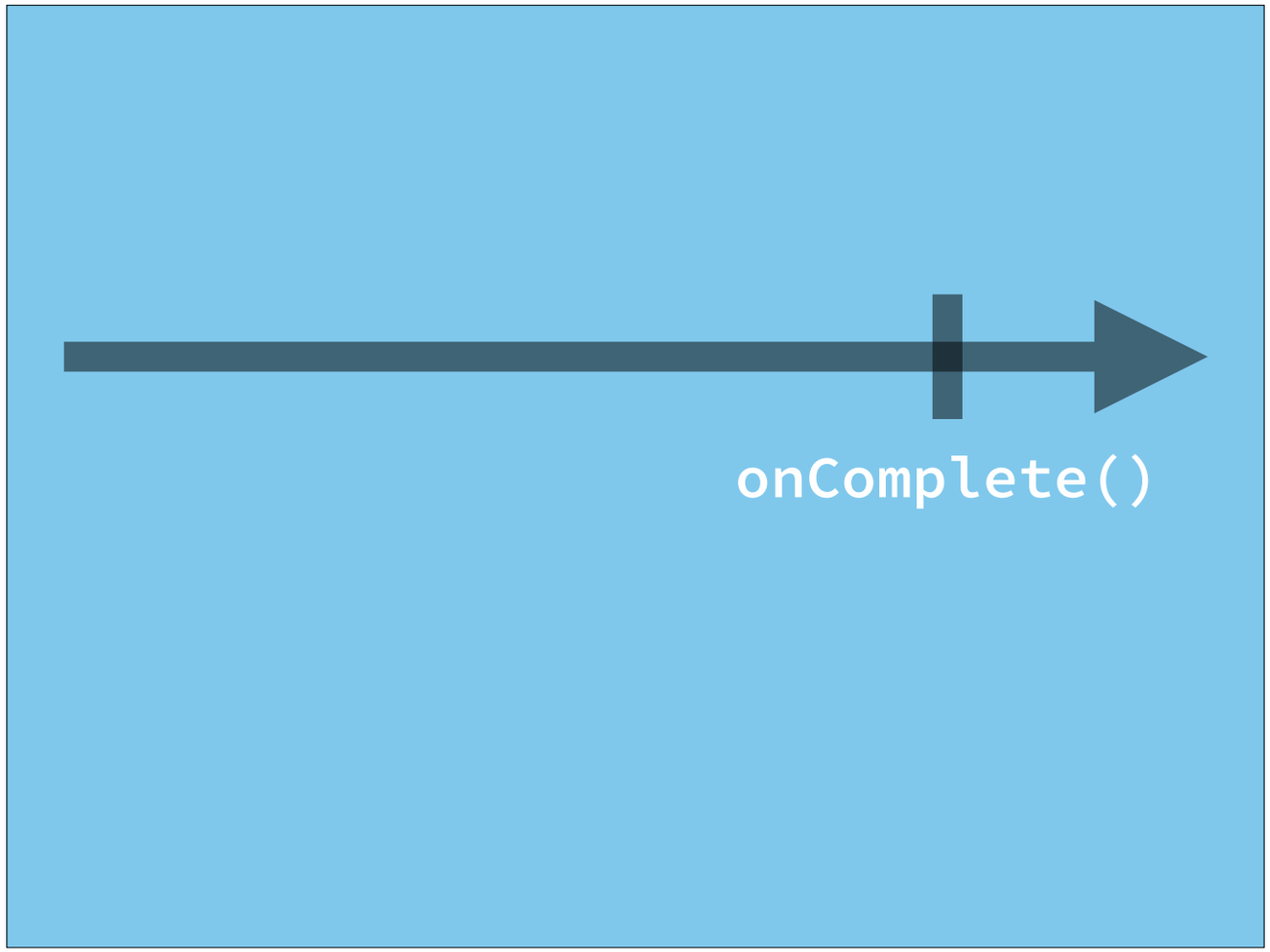

with functions
for observation

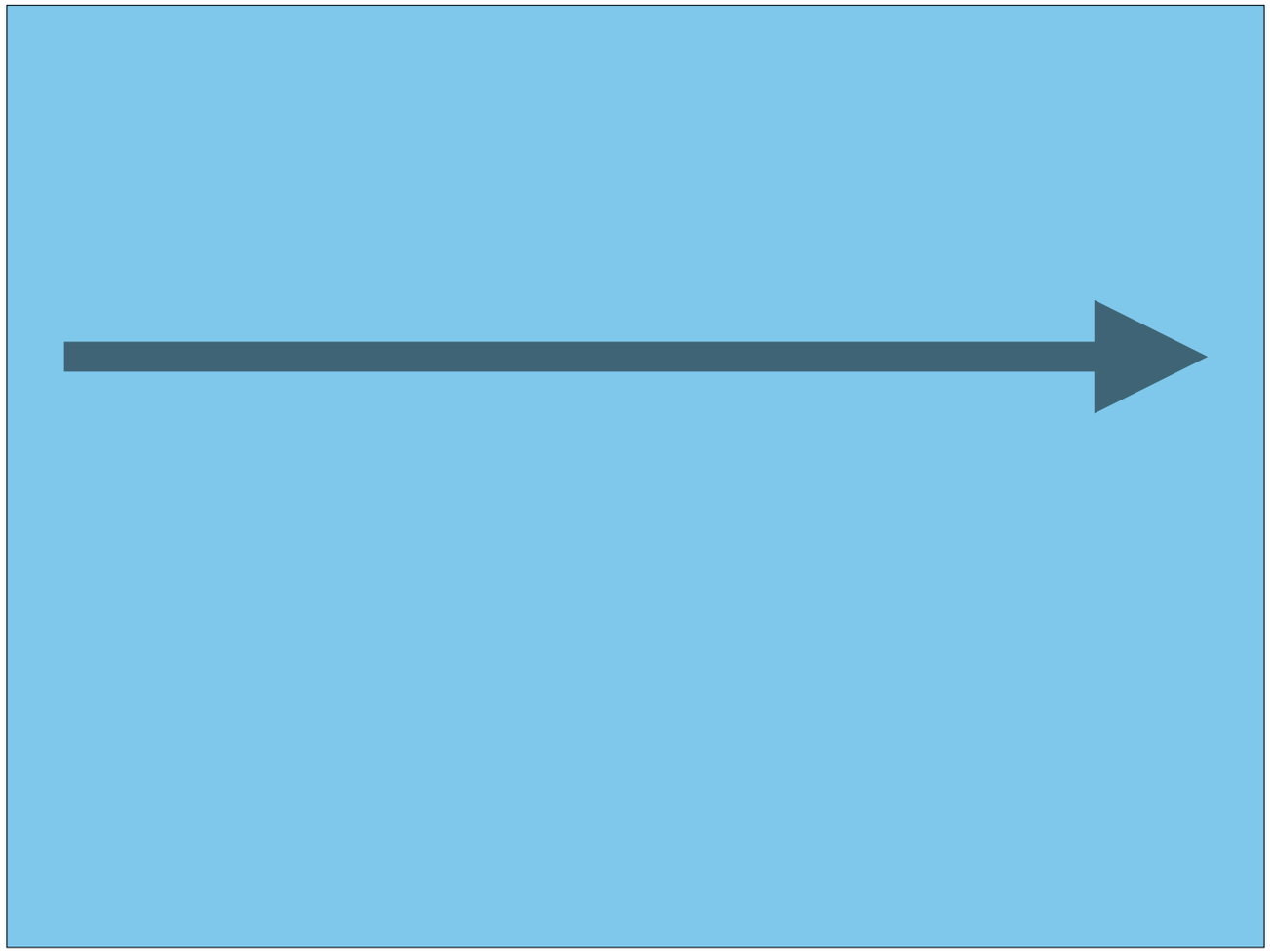



onNext(●)

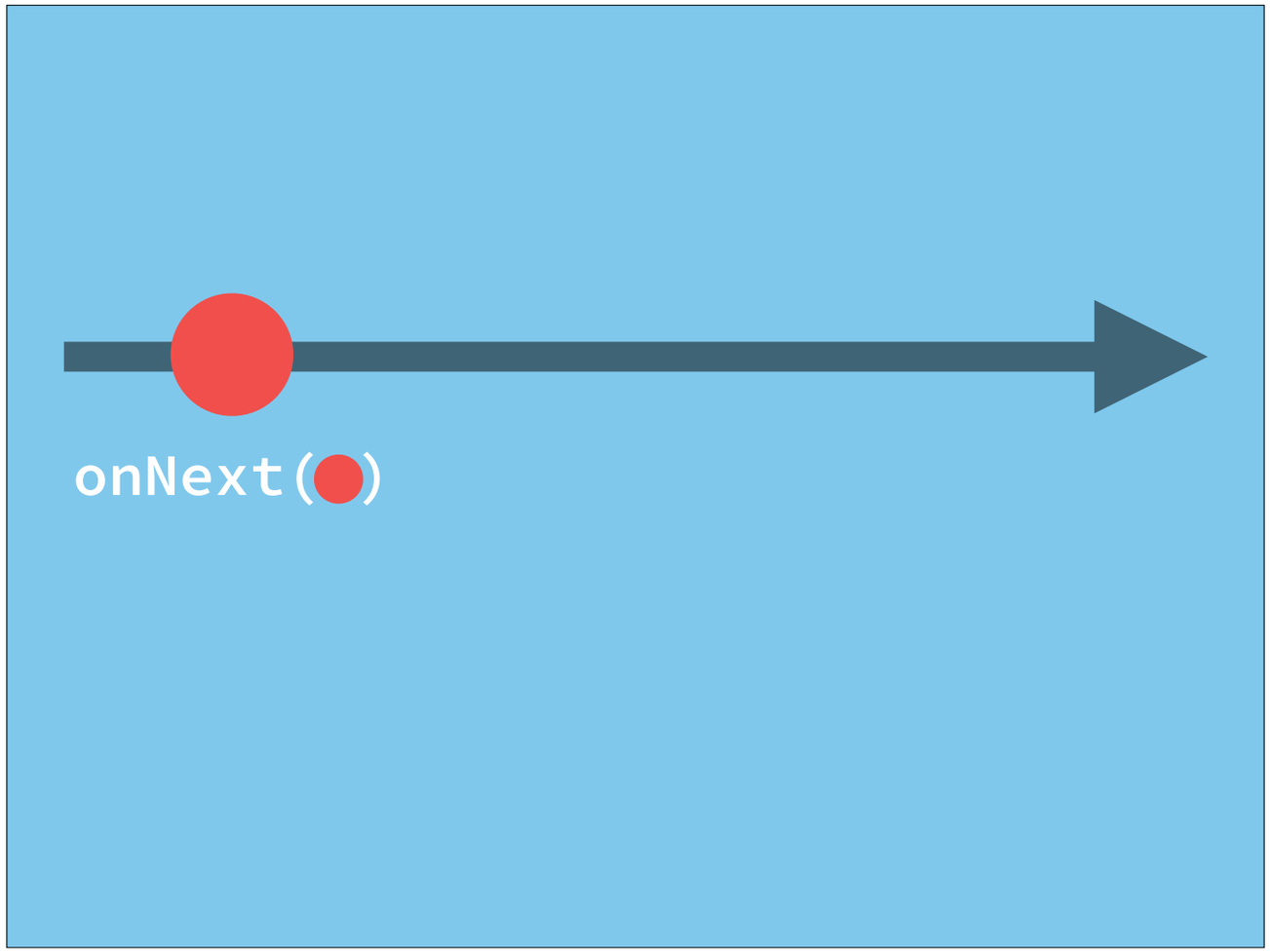

onNext(●)


onNext(●)

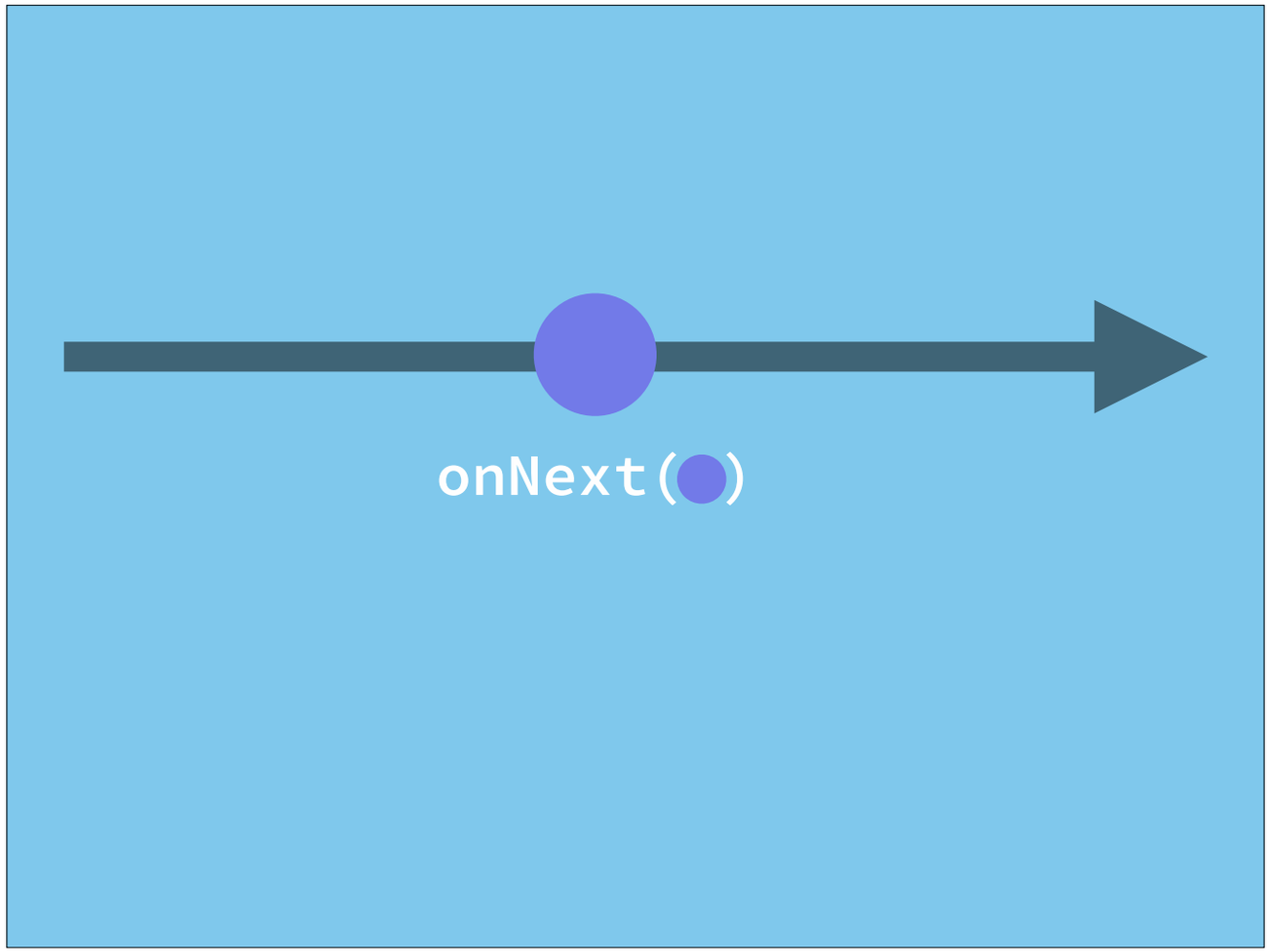

onComplete()



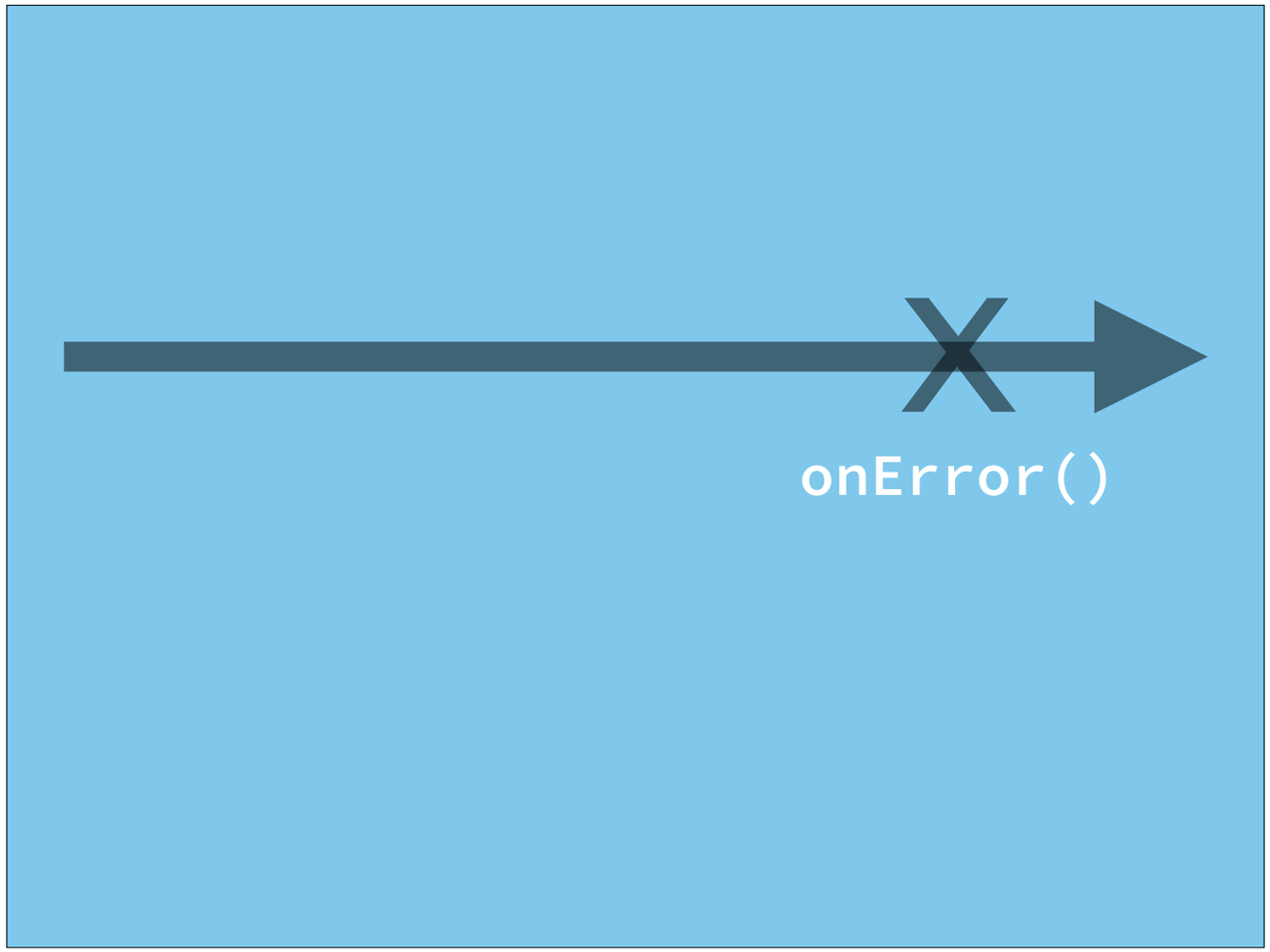
or error



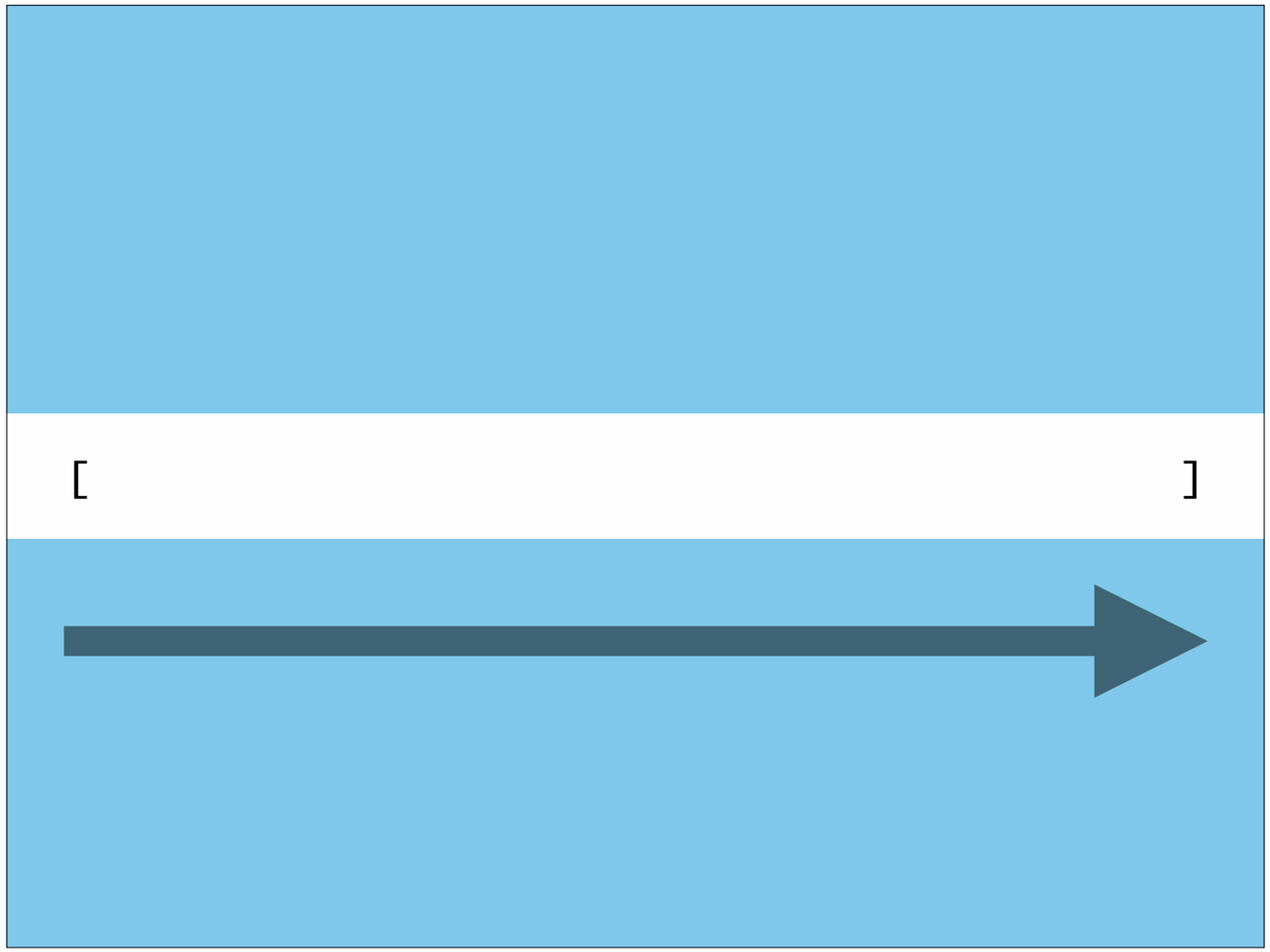
or error



or error



or error

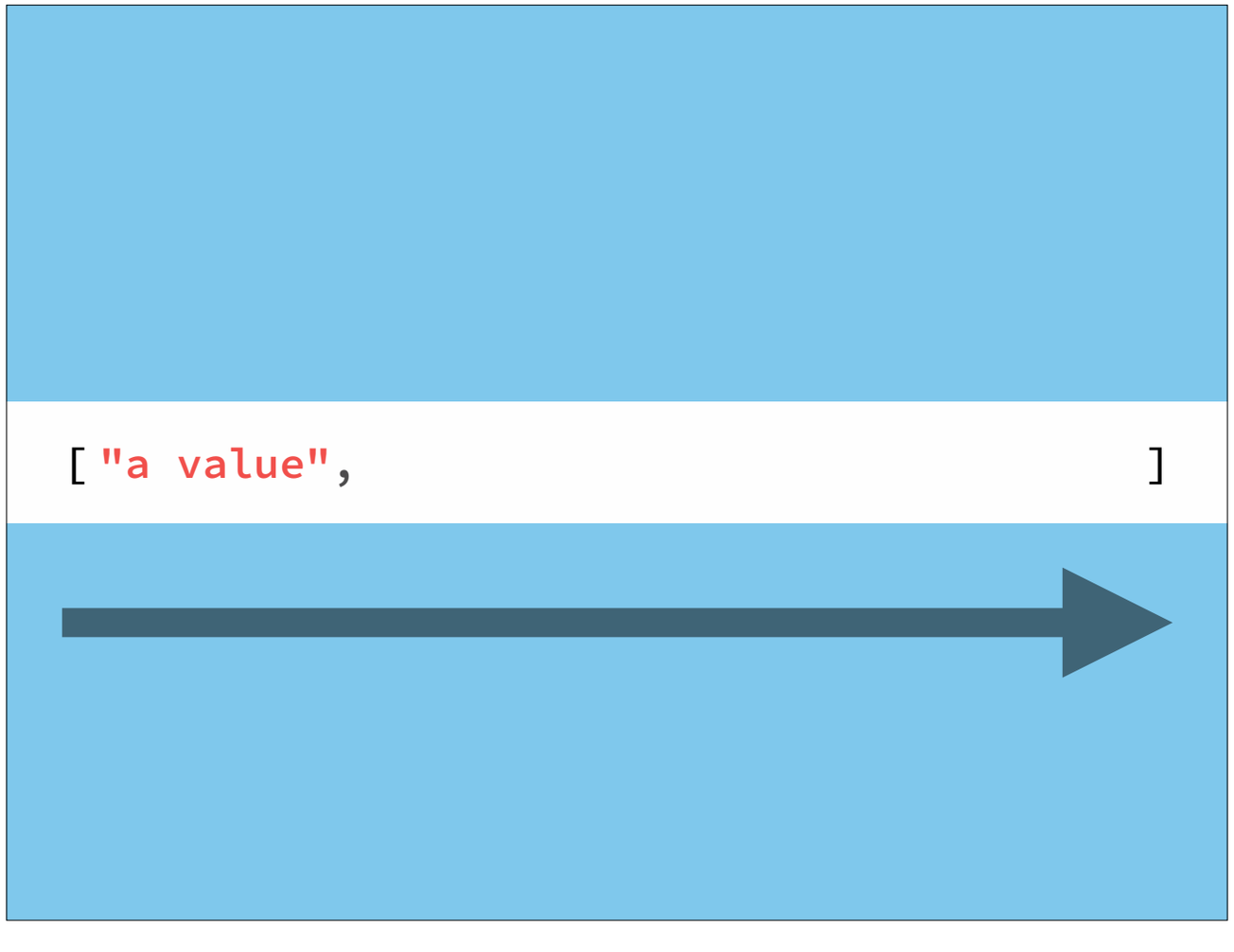


Think of the parent data stream of changes almost like an ordered collection, like an array

Your onNext() events are just like the iterator pattern, being called each time a new value is emitted.

Your next, error, and completed events allow you to both observe and iterate across changes over time

It’s not dissimilar from an array!

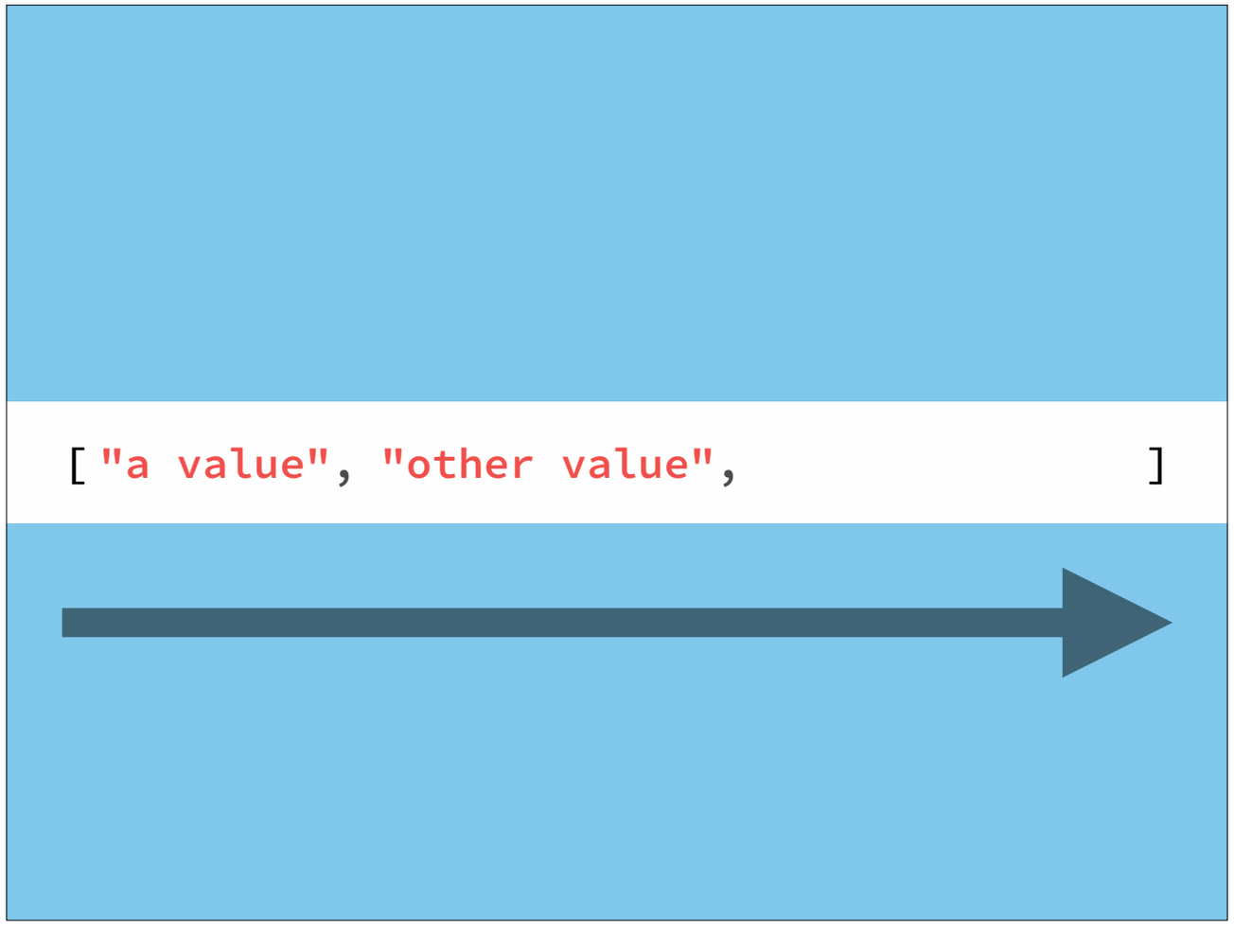


Think of the parent data stream of changes almost like an ordered collection, like an array

Your onNext() events are just like the iterator pattern, being called each time a new value is emitted.

Your next, error, and completed events allow you to both observe and iterate across changes over time

It’s not dissimilar from an array!

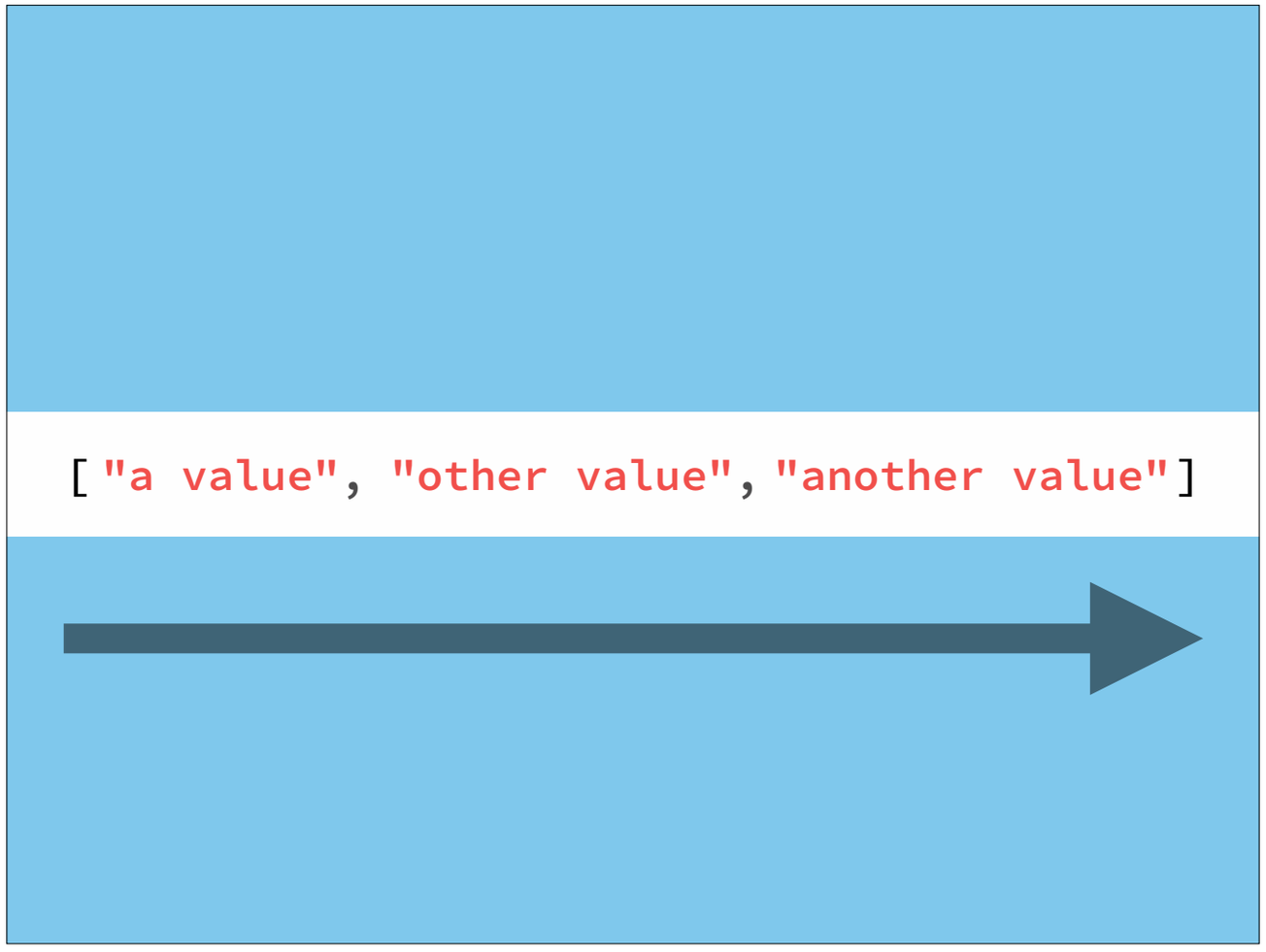


Think of the parent data stream of changes almost like an ordered collection, like an array

Your onNext() events are just like the iterator pattern, being called each time a new value is emitted.

Your next, error, and completed events allow you to both observe and iterate across changes over time

It’s not dissimilar from an array!



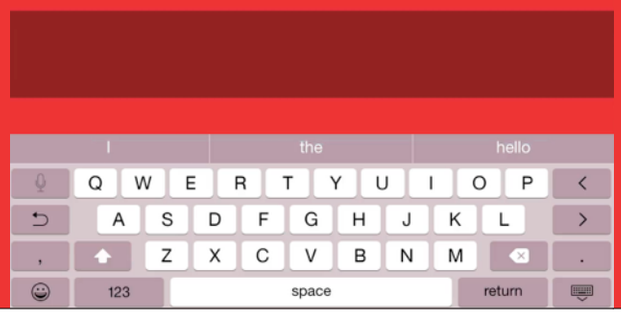
Think of the parent data stream of changes almost like an ordered collection, like an array

Your onNext() events are just like the iterator pattern, being called each time a new value is emitted.

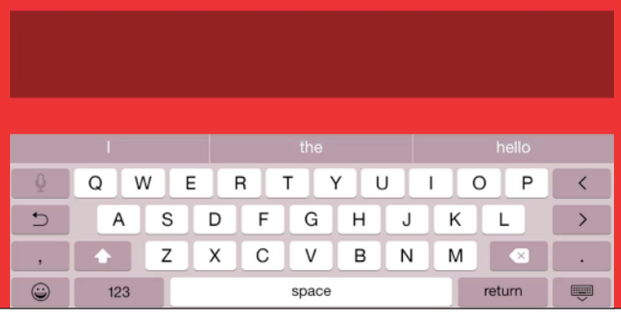
Your next, error, and completed events allow you to both observe and iterate across changes over time

It’s not dissimilar from an array!

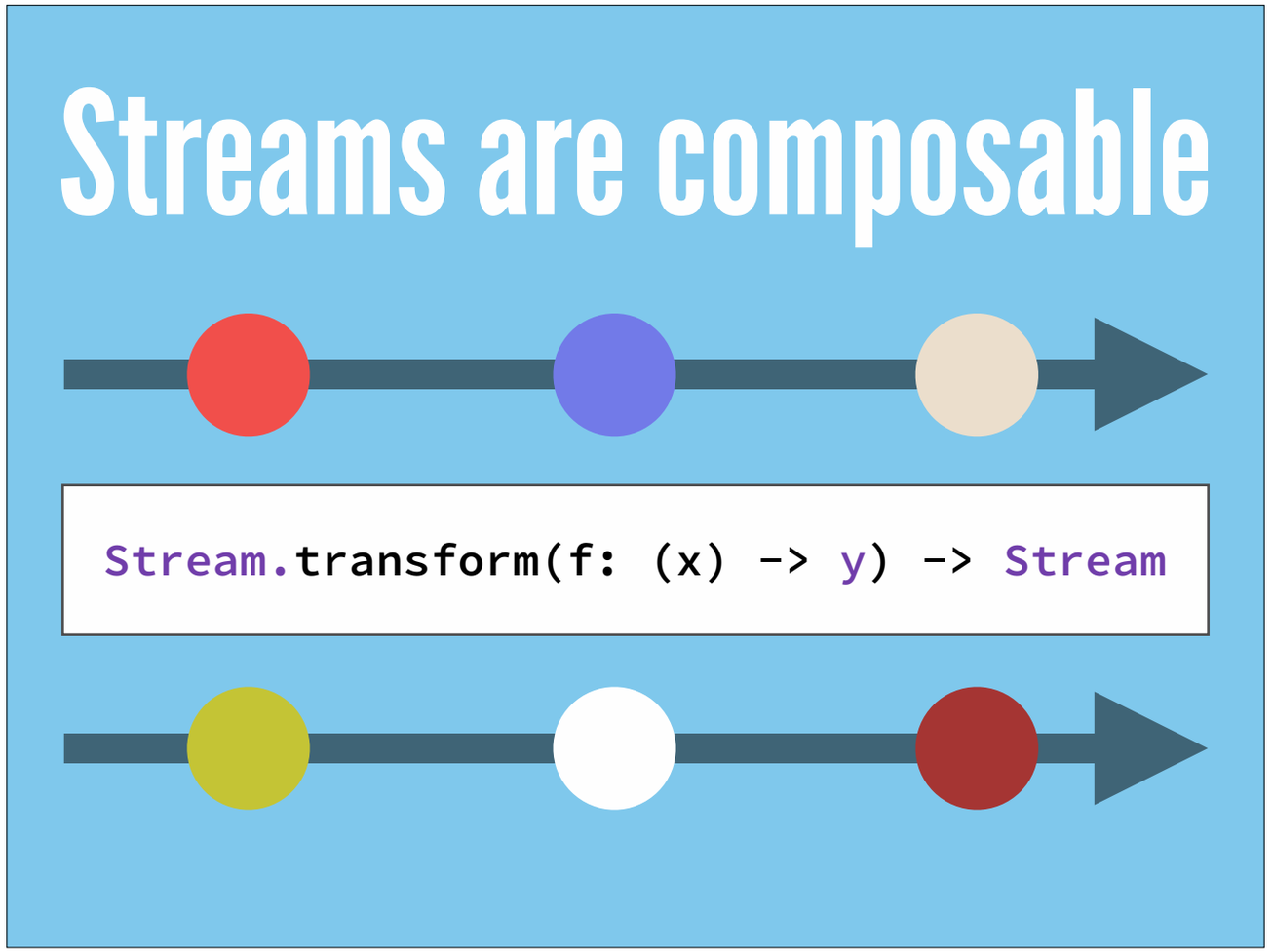
```
textField.rac_textSignal()
    .subscribeNext({ (value) -> Void in
        // value = “H”, “He”, …
    })
```



```
textField.rac_textSignal()
    .subscribeNext({ (value) -> Void in
        // value = “H”, “He”, …
    })
```



```
apiClient.performRequest(request)
    .subscribeNext({ (response) -> Void in
        // do something with response
    },
    error: { (error) -> Void in
        println("Error: \(error)")
    },
    completed: { () -> Void in
        println("Done.")
    })
```



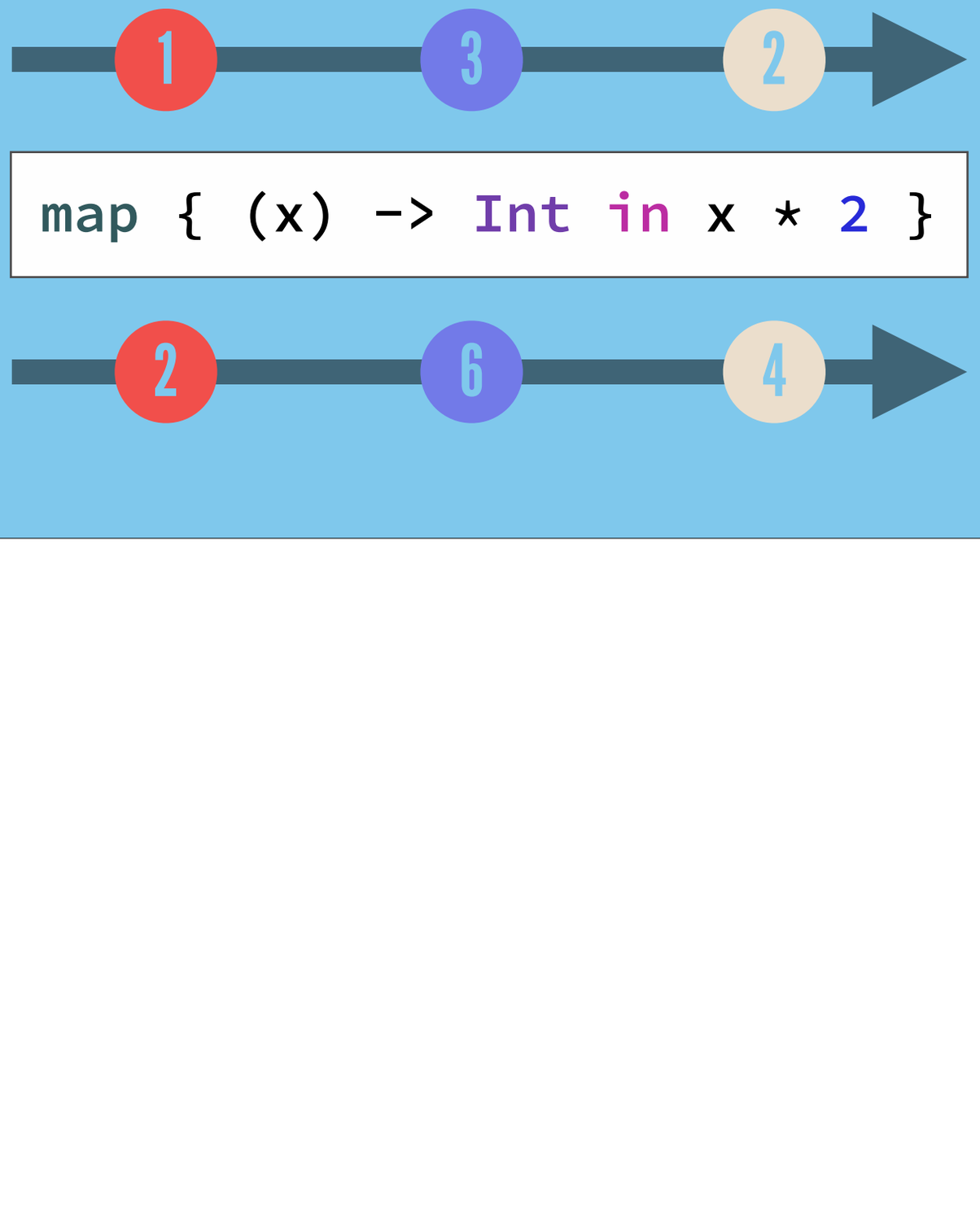
You are given an amazing toolbox of functions to combine, create and filter any of those streams.

Just like the functional tools we have on collections, these operations take in streams and output streams

It’s a simple, unified fluent interface that allows you to compose small functions to build up larger behavior.

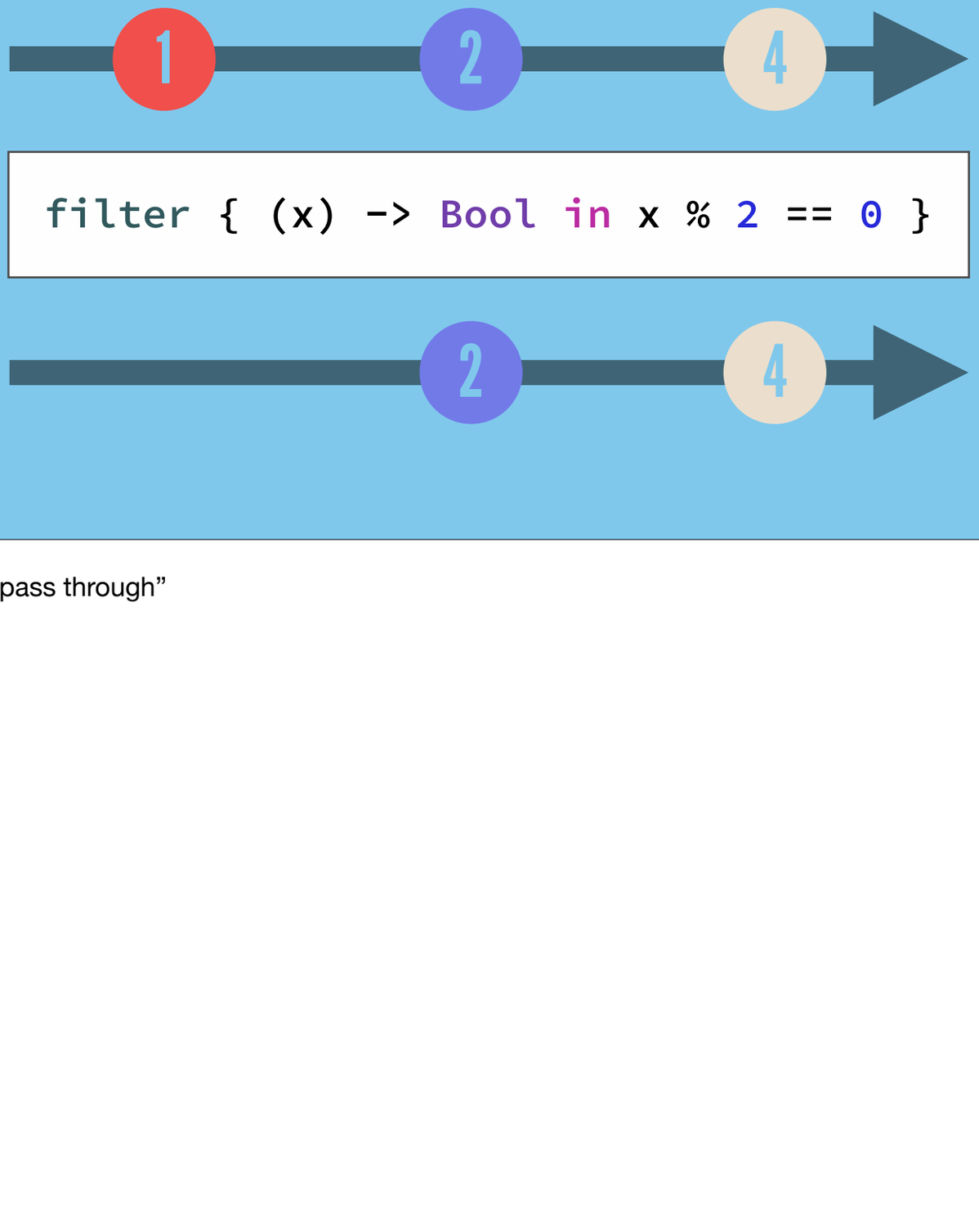
It’s not unlike Unix pipes!

Streams are composable



map for transformation

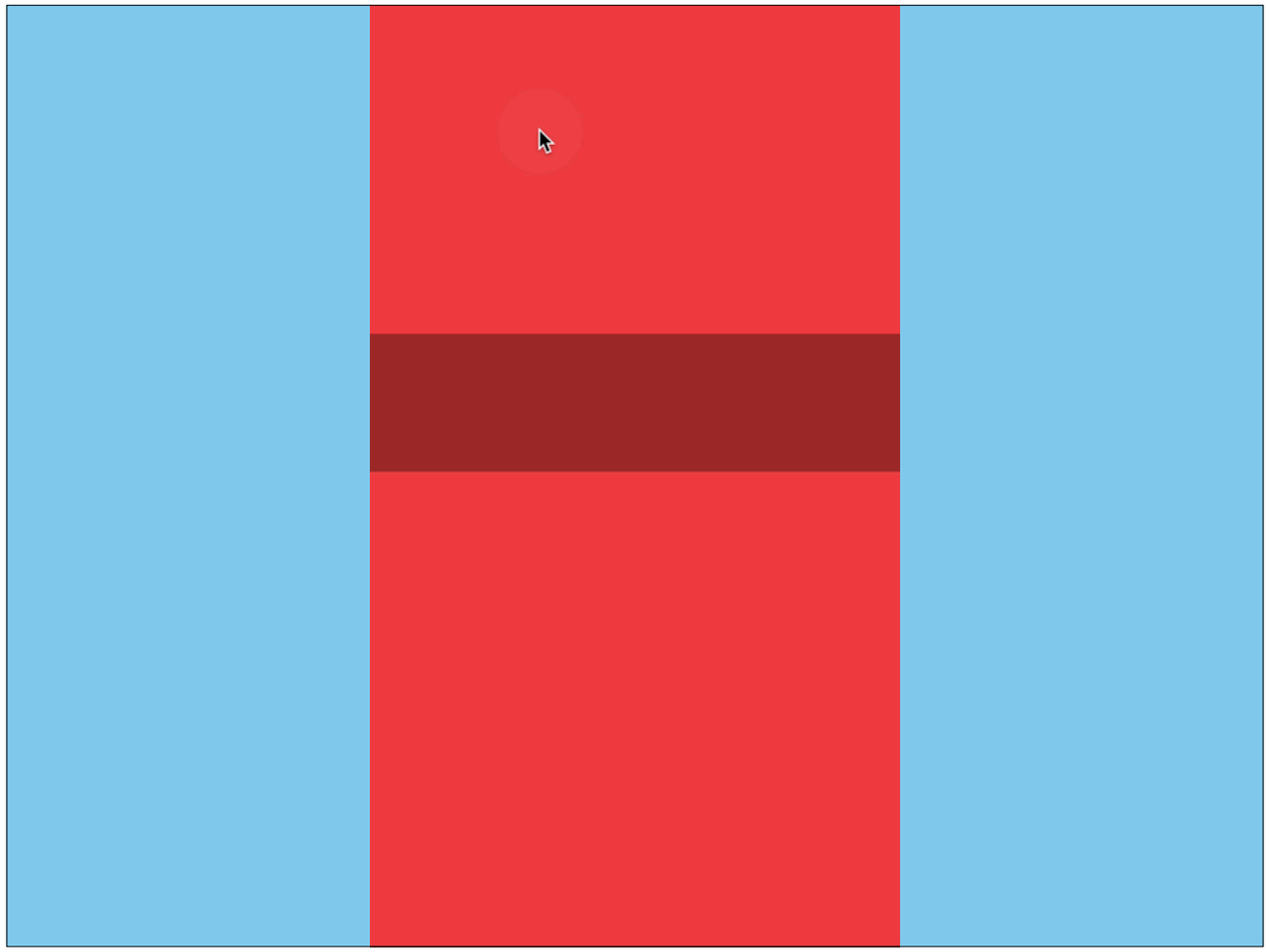
Streams are composable

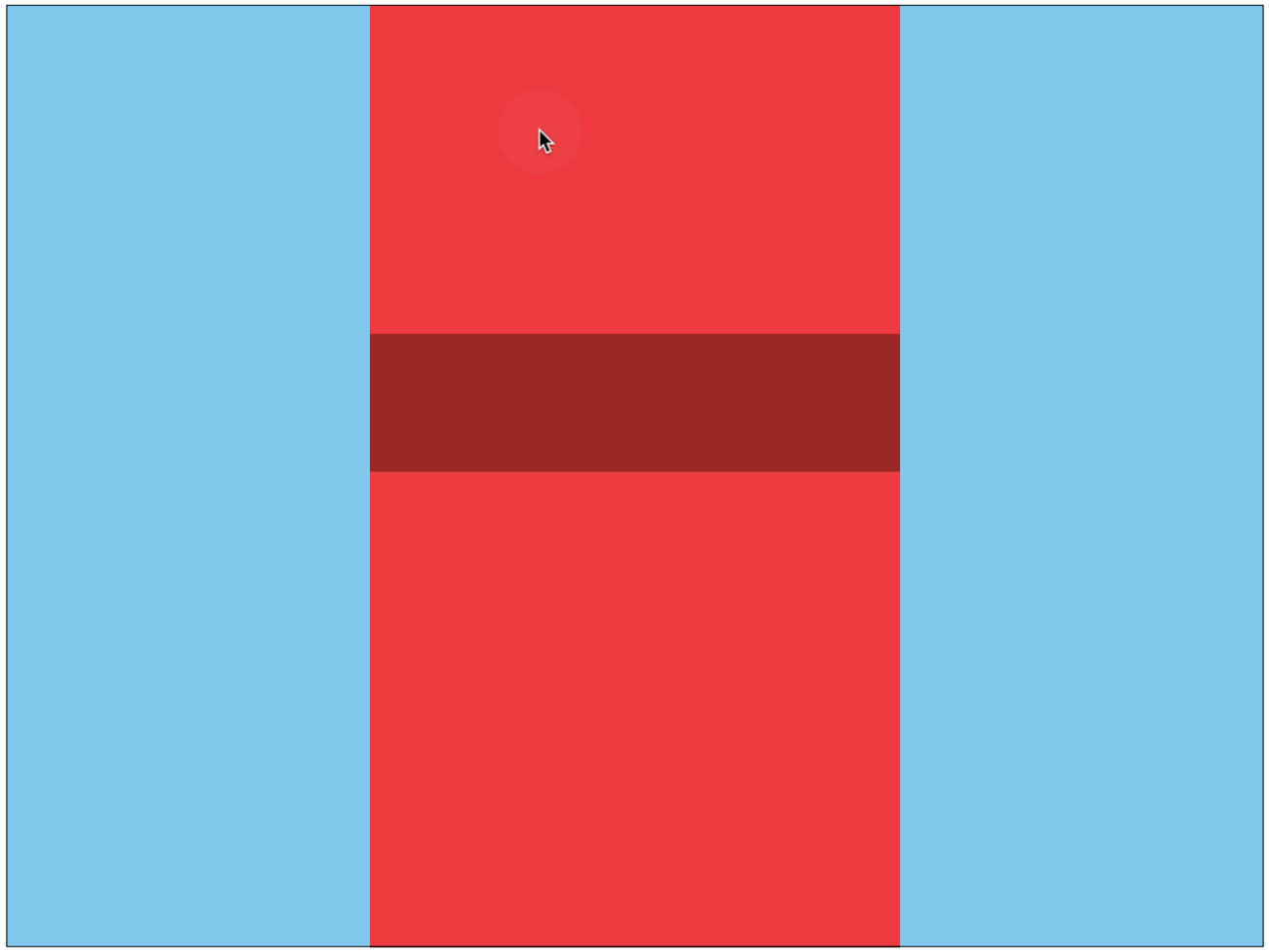


filter for allowing values that pass a test to “pass through”

Demo

Double tap!





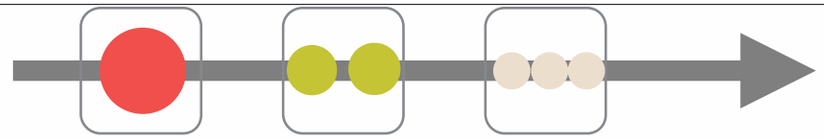
```
taps
    .bufferWithTime(0.5)

    .map { ($0 as RACTuple).count }

    .filter { ($0 as Int) == 2 }

    .subscribeNext { (_) in
        println("Double tap!")
    }
```

…next up let’s take a look at coordinating two pieces of asynchronous work whose results need to be processed at the same time…

taps

```
    .bufferWithTime(0.5)

    .map { ($0 as RACTuple).count }

    .filter { ($0 as Int) == 2 }

    .subscribeNext { (_) in
        println("Double tap!")
    }
```

…next up let’s take a look at coordinating two pieces of asynchronous work whose results need to be processed at the same time…

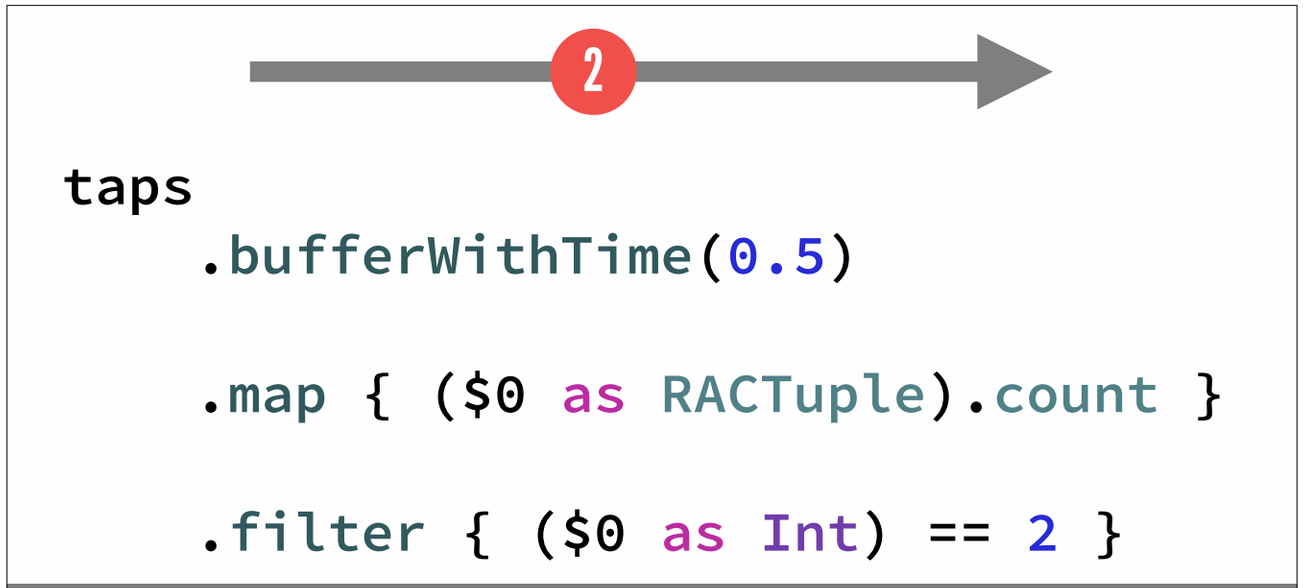


```
taps
    .bufferWithTime(0.5)
    .map { ($0 as RACTuple).count }
    .filter { ($0 as Int) == 2 }
    .subscribeNext { (_) in
        println("Double tap!")
    }
```

…next up let’s take a look at coordinating two pieces of asynchronous work whose results need to be processed at the same time…


```
taps
    .bufferWithTime(0.5)

    .map { ($0 as RACTuple).count }

    .filter { ($0 as Int) == 2 }

    .subscribeNext { (_) in
        println("Double tap!")
    }
```

…next up let’s take a look at coordinating two pieces of asynchronous work whose results need to be processed at the same time…



…next up let’s take a look at coordinating two pieces of asynchronous work whose results need to be processed at the same time…

Double tap!

```
taps
    .bufferWithTime(0.5)

    .map { ($0 as RACTuple).count }

    .filter { ($0 as Int) == 2 }

    .subscribeNext { (_) in
        println("Double tap!")
    }
```

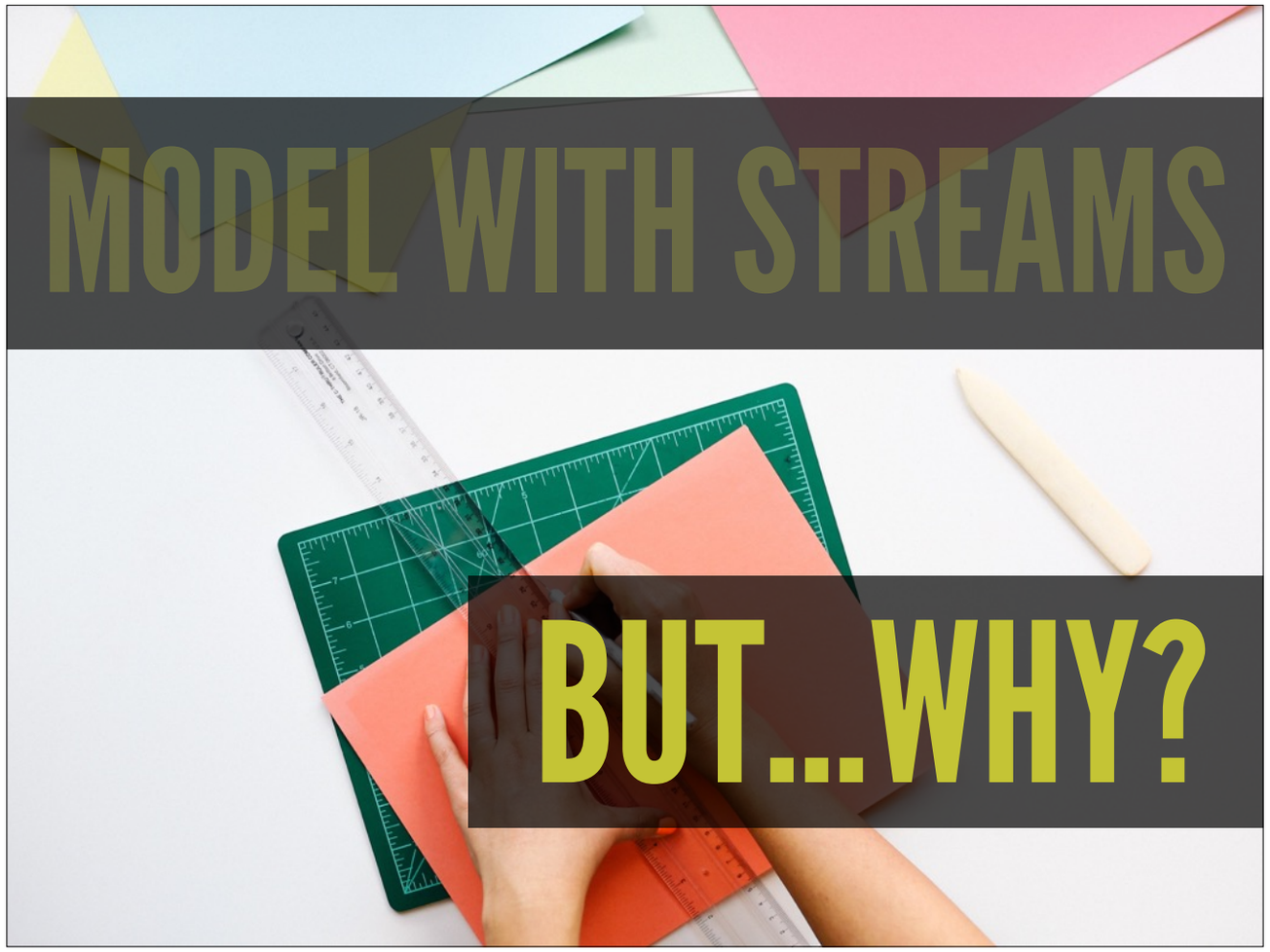
…next up let’s take a look at coordinating two pieces of asynchronous work whose results need to be processed at the same time…

```
RACSignal.combineLatest([
        slowToEmitStream,
        evenSlowerToEmitStream,
    ])
    .doNext {
        let tuple = $0 as RACTuple
        processResults(
            tuple.first,
            tuple.second)
    }
    .deliverOn(RACScheduler.mainThreadScheduler())
    .subscribeCompleted { () -> Void in
        println("The work is done!")
    }
```

MODEL WITH STREAMS

Recap: reactive programming is about modeling things with streams

Look at your program, and try to understand what things change over time, and encapsulate them into streams



But why would you use this?

We have a baseline understanding of…

- reactive programming
- the usage data streams, primarily for asynchronous work and data transformation

But going back to our starting premise, our apps already do a series of data transformations, respond to changes, handle asynchronous work.

- Why is reactive programming better?
- Why is it worth learning a new concept?
- Why is it worth adding yet another tool to my toolbox?
- What are the improvements over event handlers, callbacks, or futures and promises?

Claim

code should communicate intent

Claim: Your code should communicate what’s changing and what isn’t, what is actually happening

Your code needs to communicate what is actually happening

Your toolbox needs to be filled with tools that help us reason about our code

I’d like to set a baseline example, a precedent of how additional concepts, higher abstractions, can help us reason about our code

For event driven software, for modern applications, the current tooling is insufficient

And to help demonstrate this, I’d first like to draw an analogy with basic data transformation, and set a precedent for how additional concepts, higher level abstractions, can help us reason about our code

map

```
extension Array {

    func map<U>(transform: (T) -> U) -> [U]

}
```

Why is map a good thing?

It seems like it just adds an extra concept

Especially given that writing a for-loop is pretty easy, made even easier given that a lot of languages now have a lightweight syntax to perform iteration, like Swift's for-loop.

Why use a higher-order function?

```
for i in 0..5 {
  println("Iteration #\(i)")
}
```

```swift
let input = [1, 2, 3]

var output = [Int]()

for number in input {
    output.append(number * 2)
}
// output: [2, 4, 6]
```

```swift
let output = [1, 2, 3].map { $0 * 2 }
// output: [2, 4, 6]
```

Why is map a good thing?

It seems like it just adds an extra concept

Especially given that writing a for-loop is pretty easy, made even easier given that a lot of languages now have a lightweight syntax to perform iteration, like Swift's for-loop.

Why use a higher-order function?

```
for i in 0..5 {
  println("Iteration #\(i)")
}
```

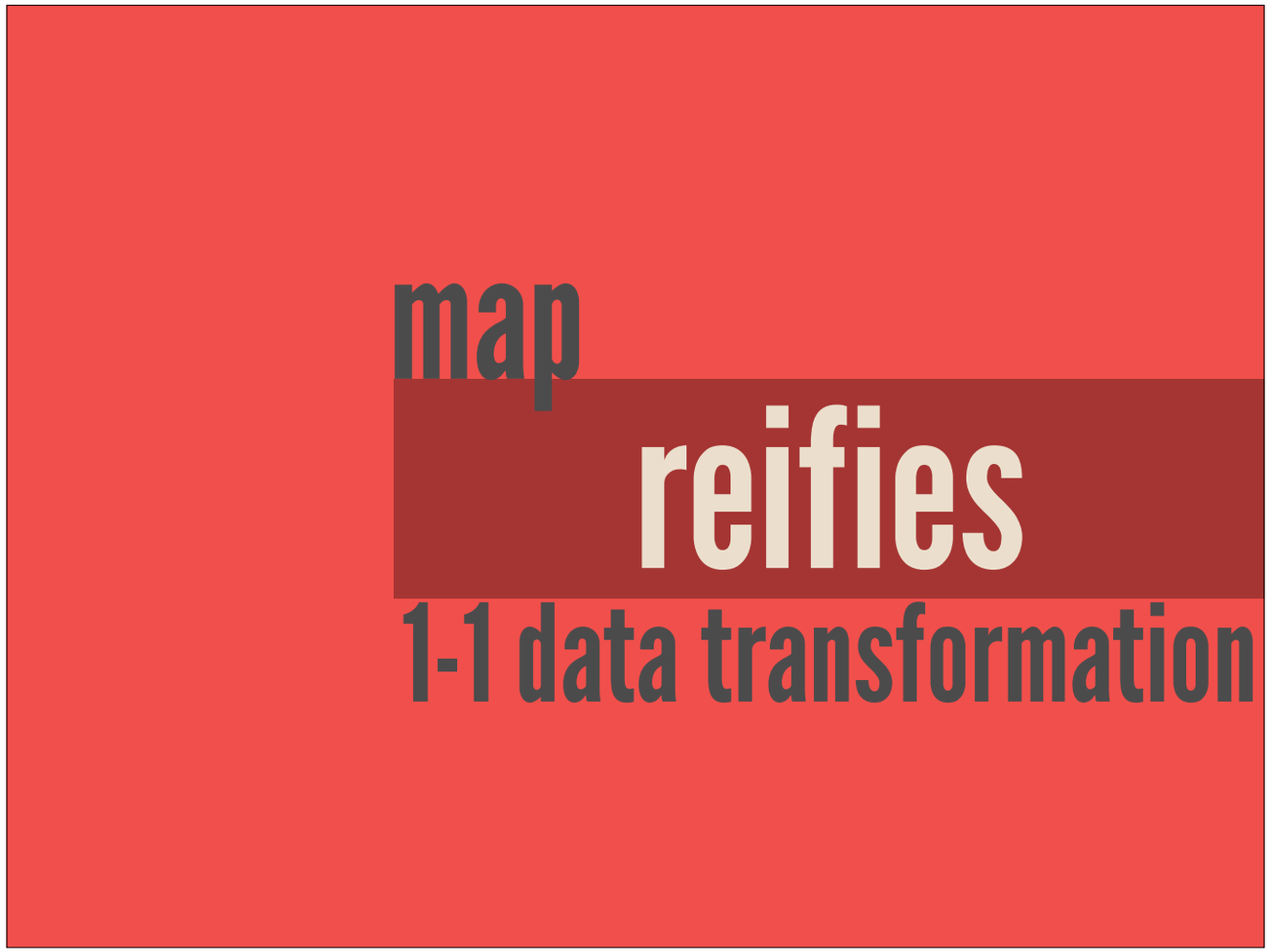
```swift
let input = [1, 2, 3]

var output = [Int]()

for number in input {
    output.append(number * 2)
}
// output: [2, 4, 6]
```

Concept: 1-1 mapping

```swift
let output = [1, 2, 3].map { $0 * 2 }
// output: [2, 4, 6]
```

When you see a map, you know exactly what is happening structurally to a collection

You're going to take some collection as an input, and for every one of its elements, there's going to be exactly one element in an output collection
It's a 1-1 mapping
Contrast with a traditional iteration to map a collection into a separate collection
There's all this boilerplate setup cruft
There's no distinct concept that you're mapping data
It's hidden in the implementation



map reifies 1-1 data transformation

Good Thing™

make (something abstract)
more concrete or real

reify |ˈrēəˌfī|

Finding something of essence, and "making something a first-class citizen"

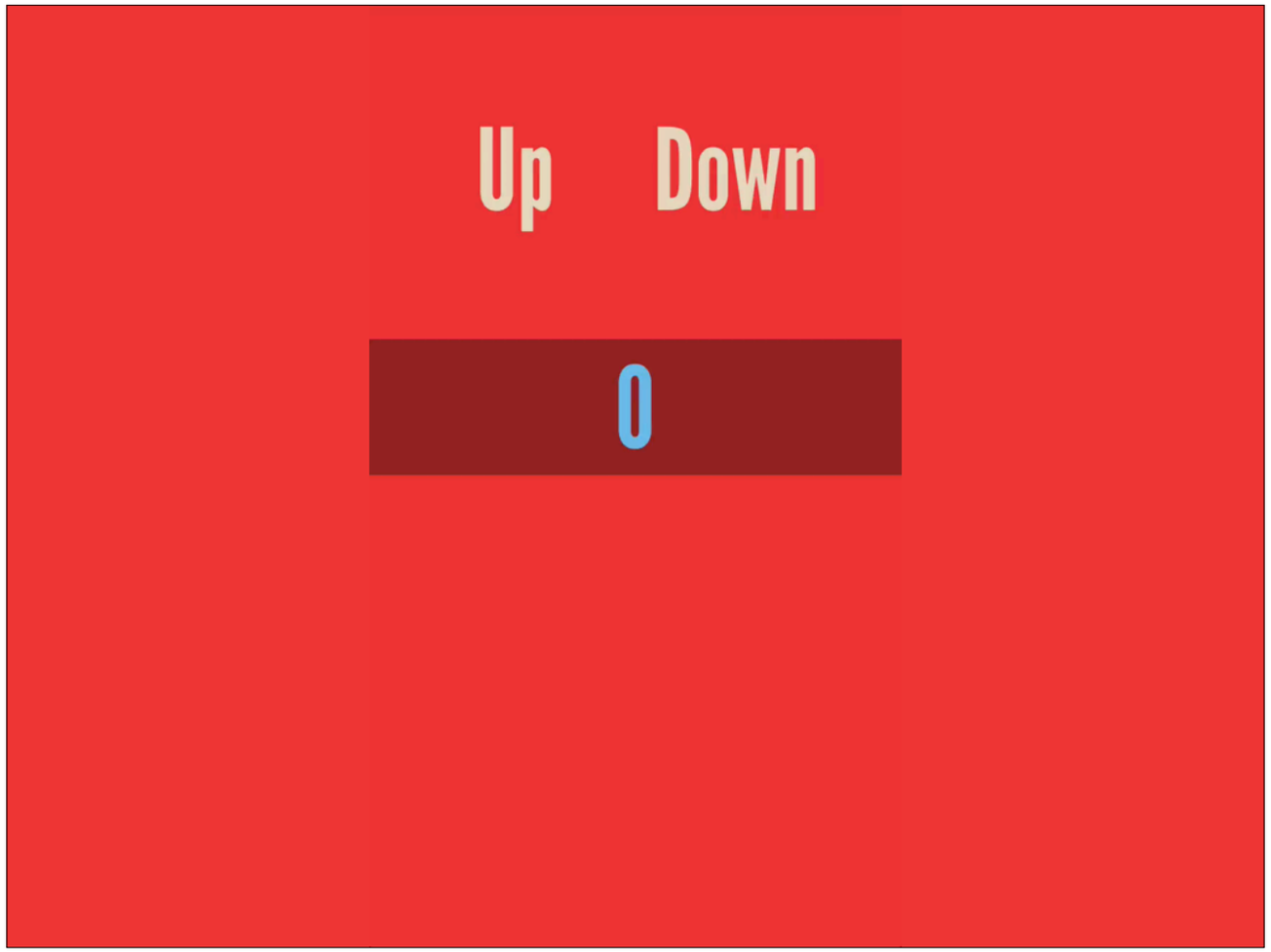
Just like map with transforming collections, if we're writing highly event-driven apps, we need to reify data streams

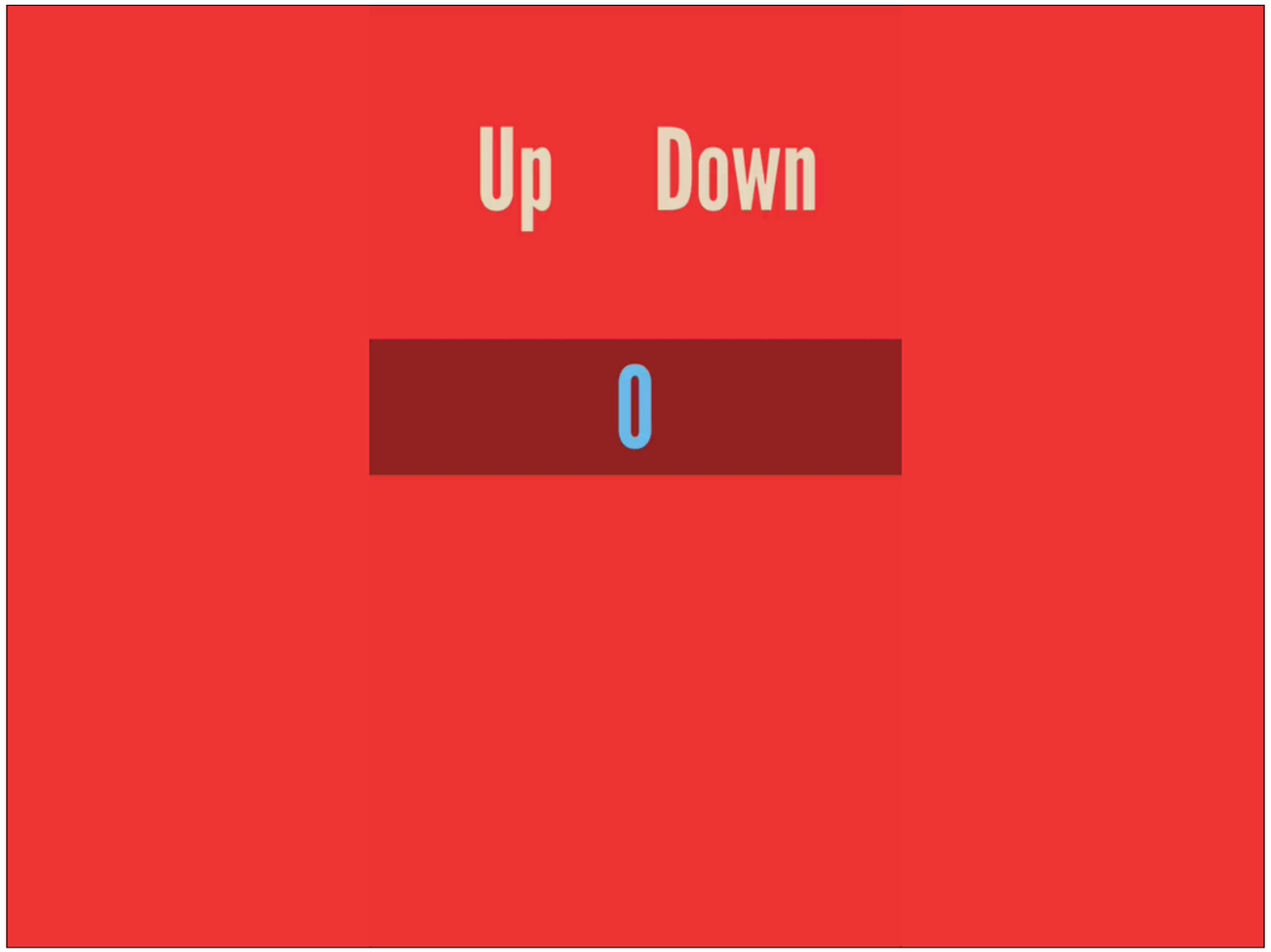
Both synchronous and asynchronous

We need to raise them up from an implicit concept that's handled by flags, timers, callbacks, while loops and other control flow primitives and into an explicit first-class citizen amongst our tools

It really is a paradigm that you can use when programming any event-driven software.

Not necessarily that helpful if your program doesn't have a lot of interaction
writing a blocking shell script
a more traditional web app that accepts some form data, handles some backend processing, and then renders a stateless web page to a frontend browser
But that's not the kind of applications we're building anymore
Our apps today are highly interactive with all sorts of real-time events
Lots of event handling
UI events
Direct user input
Backend data events
GPS
Accelerometer


Up
Down
0


Up
Down
0

```swift
var count = 0

upButton.addTarget(self, action: “upButtonTouched:”,
    forControlEvents: .TouchUpInside)

downButton.addTarget(self, action: “downButtonTouched:",
    forControlEvents: .TouchUpInside)

countLabel.text = String(count)

func upButtonTouched(sender: UIButton) {
    count++
    countLabel.text = String(count)
}

func downButtonTouched(sender: UIButton) {
    count--
    countLabel.text = String(count)
}
```

typical code

```swift
let upTaps = upButton
    .rac_signalForControlEvents(.TouchUpInside)
let downTaps = downButton
    .rac_signalForControlEvents(.TouchUpInside)

let count = RACSignal.merge([
        RACSignal.return(0),
        upTaps.mapReplace(1),
        downTaps.mapReplace(-1)
    ])
    .scanWithStart(0, reduce: { $0 + $1 })
    .map { String($0) }

count ~> RAC(self, "countLabel.text")
```

reactive code!

let, not var

```
client.loginWithSuccess({
    client.loadCachedTweetsWithSuccess({ (tweets) in
        client.fetchTweetsAfterTweet(tweets.last,
            success: { (tweets) -> Void in
                // Now we can show our tweets
            },
            failure: { (error) -> Void in
                presentError(error)
            })
        },
        failure: { (error) -> Void in
            presentError(error)
        })
    },
    failure: { (error) -> Void in
        presentError(error)
    })
```

avoid right-drifting callback hell!

```
client.login()
    .then {
        return client.loadCachedTweets()
    }
    .flattenMap { (tweets) -> RACStream in
        return client.fetchTweetsAfterTweet(tweets.last)
    }
    .subscribeError({ (error) -> Void in
        presentError(error)
    },
    completed: { () -> Void in
        // Now we can show our tweets
    })
```

TODO: Unified error handling and completion

For those of you programming in swift, or something else with a real type system. Think about how your type system actually helps you deal with change: it probably doesn’t.

Now that we’ve reified the concept of change over time, with data streams, reactive programming can bring this concept into the type system. This helps communicate what things are expected to change, enforces that clients will deal with changes properly, and provides tooling to do both.

```swift
/// Clients should observe via property
    observer:
public var title: String
```

```swift
public let title: Signal<String, NoError>
```

Borrowing syntax from RAC 3, which is nearly complete and uses generics…

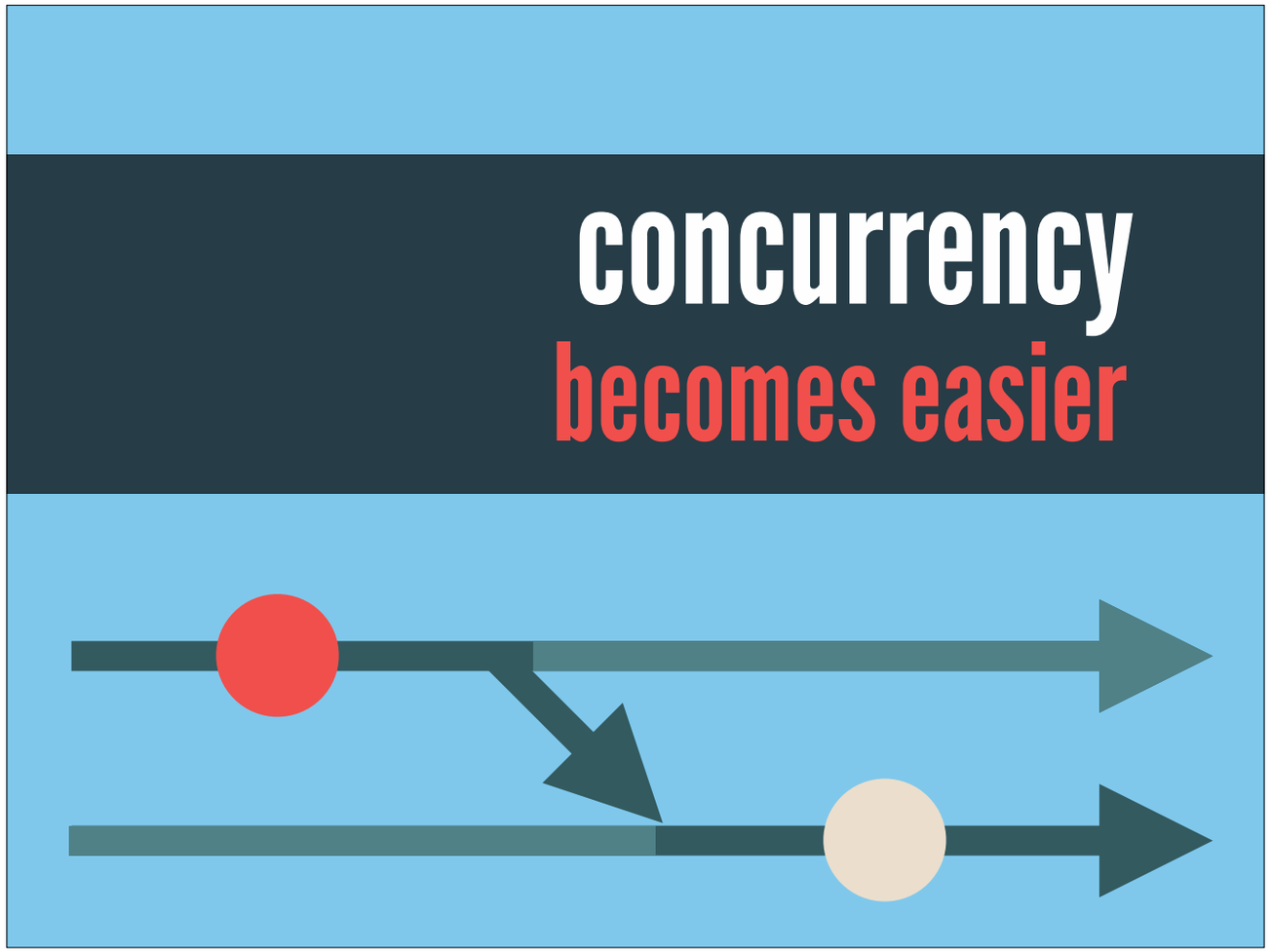
Note that the first version can change, but clients need to know that, and they must manually set up observation. Not only that, they don’t have a rich toolbox for processing the title as it changes. And they only have imperative tools for binding these changes to the UI.

In the second, Reactive version, the type system communicates that this value may change, and it ensures that clients deal with that change. Because this is a Signal, the client has powerful functional tools for transforming this data and dealing with the change. And they set up the binding to the UI once, in a declarative way, as we saw in the counter example on slide 46.

```swift
func loginWithSuccess(
    success: () -> Void,
    failure: (NSError) -> Void)
{
    // function doesn't return anything
    // it jumps into two callbacks based on return
}
```

```swift
func login() -> RACSignal {
    // function has a native return value
    // a data stream object that we can observe
}
```

also lets us avoid the void-return + callback block smell



Reactive libraries, and ReactiveCocoa specifically, provide powerful tools for dealing with concurrency via data streams.

I left this until late in the talk, and I'm not going into detail, because these tools actually turn out to be less important in this paradigm. Because we're not writing imperative code, we don't need heavyweight concurrency tooling. It turns out to be relatively easy.

When we're thinking in terms of a stream of data, which we might transform into a different stream or which we might attach side effects to, we can effectively just tell the stream to "jump" between threads and start delivering its values on a different thread. Then the transformations or side effects happen on whatever thread the stream jumps to.

reactive concurrency

`Scheduler`

schedules when and where work is performed

All reactive frameworks that deal with concurrency (RAC, RxJava, Rx in .NET) have the concept of a Scheduler. A Scheduler determines when and where a block of work is performed: what thread, whether immediately or after a delay.

The work could be transforming values in order to transform one signal into a new one. Or it could be performing some side effects like a network request or UI updates.

A lot of the time, you don’t have to think about Schedulers. Work happens on the same thread where a signal originates by default, which is usually fine.

reactive concurrency

`Scheduler`

schedules when and where work is performed

`deliverOn(Scheduler)`

`subscribeOn(Scheduler)`

But if you need to make some code concurrent, there are two key operators.

(describe subscribeOn and deliverOn)

So both of these basically force the work before or after them to be performed on another thread. And that's all.

Now that we're thinking about change and work in our application as a stream of values, with transformations embedded in it and side effects attached, we need only these tools to determine where the stream flows to have those side effects and transformations happen.

You can also use a stream to wrap existing asynchronous code, and any streams which provide the result of a long-running or heavy task (networking, image processing) can do their work in the background by default. So really, 95% of the time, you solely need deliverOn.

Note that if you really try, you can write code that will deadlock. But this is much less error-prone than basically any other mainstream concurrency coordination system.


SO...
HOW DO I DO THIS?
(photo by Andrew Sardone)

ReactiveCocoa	github.com/ReactiveCocoa/ReactiveCocoa
ReactiveExtensions	github.com/Reactive-Extensions
RxJava	github.com/ReactiveX/RxJava
Bacon.js	github.com/baconjs/bacon.js
Elm	elm-lang.org

SOURCES

FURTHER READING

Missing RX Intro	j.mp/missingrxintro
Input & Output	j.mp/joshaberio
RxMarbles	rxmarbles.com
Other	pinboard.in/u:cdzombak/t:reactiveprogramming


BYE
(photo by Chris Dzombak)
(puppy by Shane Bliemaster)