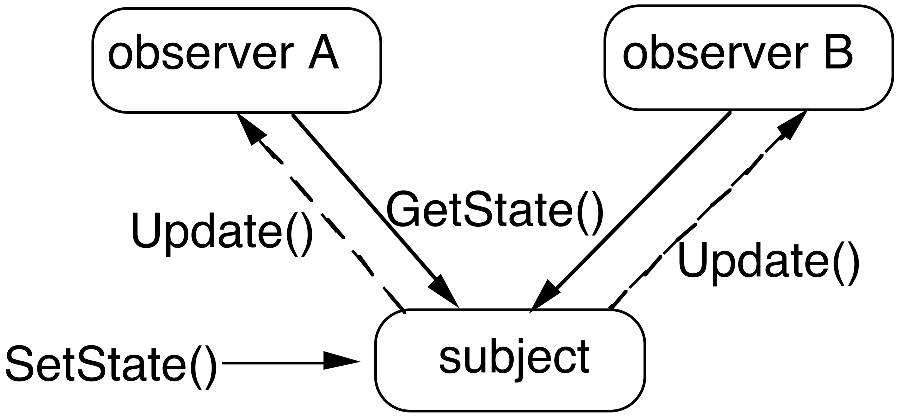
# CS 635 Advanced Object-Oriented Design & Programming
Fall Semester, 2020
Doc 7 Observer Pattern
Sep 10, 2020

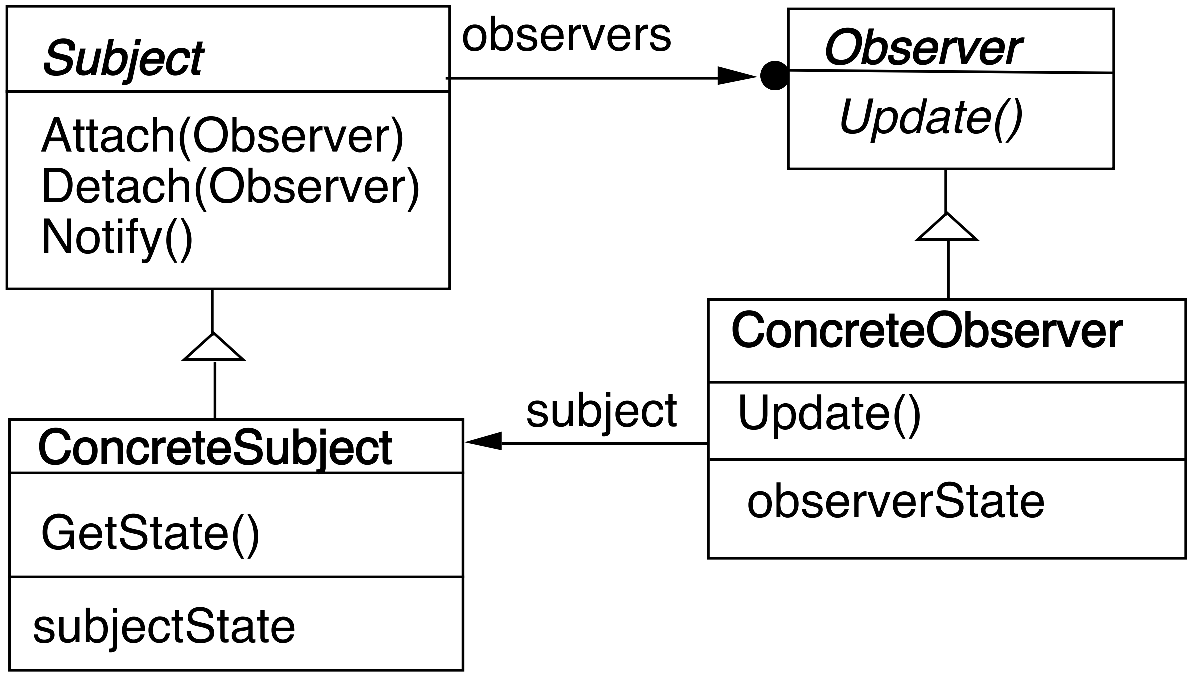
# Observer

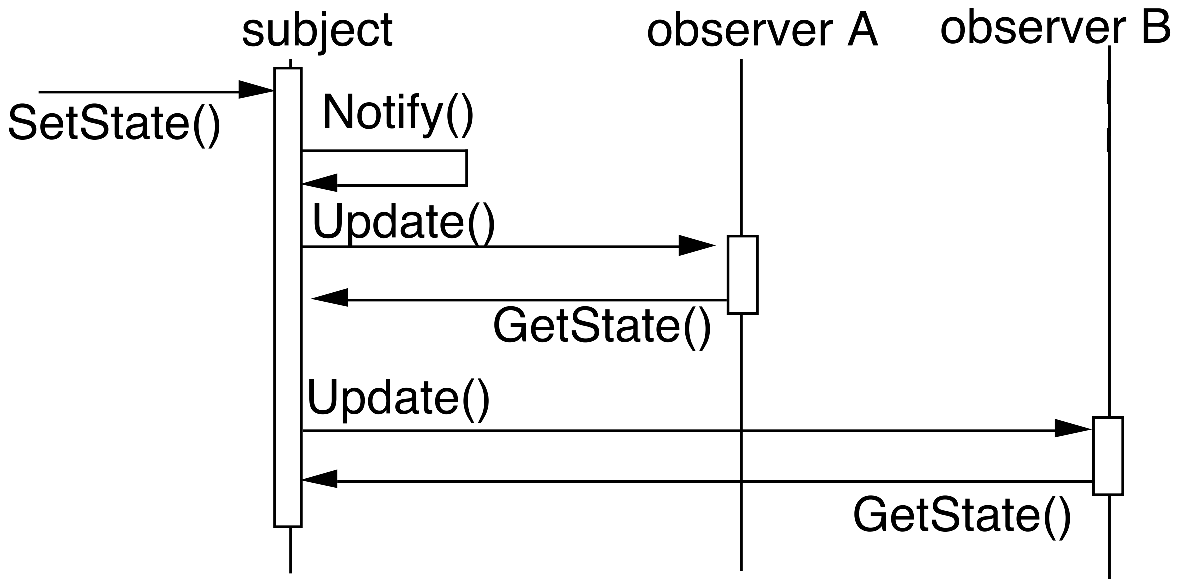
One-to-many dependency between objects

When one object changes state,
all its dependents are notified and updated automatically

# Structure


observer A
observer B
Update()
GetState()
Update()
SetState()
subject


Subject
Attach(Observer)
Detach(Observer)
Notify()
observers
Observer
Update()
ConcreteObserver
Update()
observerState
subject
ConcreteSubject
GetState()
subjectState


subject
observer A
observer B
SetState()
Notify()
Update()
GetState()
Update()
GetState()

# Common Java Example - Listeners

Java Interface

View.OnClickListener

abstract void onClick(View v)

Called when a view has been clicked.

# Java Example

```
public class CreateUIInCodeActivity extends Activity implements View.OnClickListener{
   Button test;

   @Override
   public void onCreate(Bundle savedInstanceState) {
      super.onCreate(savedInstanceState);
      setContentView(R.layout.main);
      test = (Button) this.findViewById(R.id.test);
      test.setOnClickListener(this);
   }

   public void onClick(View source) {
     Toast.makeText(this, "Hello World", Toast.LENGTH_SHORT).show();
   }
}
```

# Pseudo Java Example

```
public class Subject {
   Window display;
   public void someMethod() {
      this.modifyMyStateSomeHow();
      display.addText( this.text() );
   }
}
```

Abstract coupling - Subject & Observer

Broadcast communication

Updates can take too long

```
public class Subject {
   ArrayList observers = new ArrayList();

   public void someMethod() {
      this.modifyMyStateSomeHow();
      changed();
   }

   private void changed() {
      Iterator needsUpdate = observers.iterator();
      while (needsUpdate.hasNext() )
         needsUpdate.next().update( this );
   }
}

public class SampleWindow {
   public void update(Object subject) {
      text = ((Subject) subject).getText();
      Thread.sleep(10000).
   }
}
```

# Some Language Support

| Smalltalk | Java | Ruby | Clojure | Observer Pattern |
|---|---|---|---|---|
| Object | ~~Observer~~ |  | function | Abstract Observer class |
| Object & Model | ~~Observable~~ | Observable | watches on data | Subject class |

Smalltalk Implementation

Object implements methods for both Observer and Subject.

Actual Subjects should subclass Model

# Java's Observer

**Class java.util.Observable**

void addObserver(Observer o)
void clearChanged()
int countObservers()
void deleteObserver(Observer o)
void deleteObservers()
boolean hasChanged()
void notifyObservers()
void notifyObservers(Object arg)
void setChanged()

Deprecated in Java 9

| Java | Observer Pattern |
|---|---|
| Interface Observer | Abstract Observer class |
| Observable class | Subject class |

Observable object may have any number of Observers

Whenever the Observable instance changes,
it notifies all of its observers

Notification is done by calling the update() method on all observers.

**Interface java.util.Observer**

Allows all classes to be observable by instances of class Observer

# Flow

Java Observer & Observable are replaced by

- java beans
- Reactive Streams (Flow)

Flow

- Publisher (Subject)
- Subscriber (Observer)
- Processor (Subject & Observer)
- Subscription
  - Link between publisher & subscriber

# Coupling & Observer Pattern

Subject coupled to Observer interface

- Does not know the concrete type of the observers
- There can be 0+ observers

# Implementation Issues

# Mapping subjects(Observables) to observers

Use list in subject

Use hash table

```
public class Observable {
    private boolean changed = false;
    private Vector obs;

public Observable() {
        obs = new Vector();
    }

public synchronized void addObserver(Observer o) {
        if (!obs.contains(o)) {
            obs.addElement(o);
        }
    }
```

# Observing more than one subject

If an observer has more than one subject how does it know which one changed?

Pass information in the update method

# Deleting Subjects

In C++ the subject may no longer exist

Java/Smalltalk observer may prevent subject from garbage collection

# Who Triggers the update?

**Have methods that change the state trigger update**

```
class Counter extends Observable  {           // some code removed
   public void increase()  {
       count++;
       setChanged();
       notifyObservers( INCREASE );
   }
}
```

**Have clients call Notify at the right time**

```
class Counter extends Observable  {       // some code removed
        public void increase()  {   count++;  }
}

Counter pageHits = new Counter();
pageHits.increase();
pageHits.increase();
pageHits.increase();
pageHits.notifyObservers();
```

# Subject is self-consistent before Notification

```
class ComplexObservable extends Observable {
    Widget frontPart = new Widget();
    Gadget internalPart = new Gadget();

    public void trickyChange() {
       frontPart.widgetChange();
       internalpart.anotherChange();
       setChanged();
       notifyObservers( );
    }
}

class MySubclass extends ComplexObservable {
    Gear backEnd = new Gear();

    public void trickyChange() {
       super.trickyChange();
       backEnd.yetAnotherChange();
       setChanged();
       notifyObservers( );
    }
}
```

# Adding information about the change

push models - add parameters in the update method

```
class IncreaseDetector extends Counter implements Observer { // stuff not shown

  public void update( Observable whatChanged, Object message) {
       if ( message.equals( INCREASE) )
            increase();
  }
}

class Counter extends Observable {            // some code removed
  public void increase()  {
       count++;
       setChanged();
       notifyObservers( INCREASE );
  }
}
```

# Adding information about the change

pull model - observer asks Subject what happened

```
class IncreaseDetector extends Counter implements Observer {
 public void update( Observable whatChanged ) {
      if ( whatChanged.didYouIncrease() )
           increase();
 }
}

class Counter extends Observable {          // some code removed
 public void increase() {
      count++;
      setChanged();
      notifyObservers( );
 }
}
```

# Rate of Updates

In single threaded operation

- All observers must finish before subject can continue operation

What to do when subject changes faster than observers can handle

# Scaling the Pattern

# Java Event Model

AWT/Swing components broadcast events to Listeners

JDK1.0 AWT components broadcast an event to all its listeners

A listener normally not interested all events

Broadcasting to all listeners was too slow with many listeners

# Java 1.1+ Event Model

Each component supports different types of events:

Component supports

- ComponentEvent
- KeyEvent
- FocusEvent
- MouseEvent

Each event type supports one or more listener types:

MouseEvent

- MouseListener
- MouseMotionListener

Each listener interface replaces update with multiple methods

MouseListener

- mouseClicked()
- mousePressed()
- mouseEntered()
- mouseReleased()

Listeners

- Only register for events of interest
- Don't need case statements to determine what happened

# Small Models

Often an object has a number of fields(aspects) of interest to observers

Rather than make the object a subject make the individual fields subjects

- Simplifies the main object
- Observers can register for only the data they are interested in

VisualWorks ValueHolder

Subject for one value

ValueHolder allows you to:

- Set/get the value
  - Setting the value notifies the observers of the change
- Add/Remove dependents

# Reactive Programming

# Reactive Manifesto

https://www.reactivemanifesto.org

Organizations working in disparate domains are independently discovering patterns for building software that look the same.

These systems are more robust, more resilient, more flexible and better positioned to meet modern demands.

Reactive Systems are
- Responsive
- Resilient
  - React to failure
- Elastic
  - React to load
- Message Driven

Motivation

Need millisecond response
100% uptime
Data is measured in Petabytes
Applications run on
- Mobile
- Clusters of 1000s of multicor

# History

1997 - Elliott & Hudak

Fran - reactive animations Reactive Functional Programing

2009 Akka

Actor model + reactive streams

2009 Reactive Extension for .NET early version

2011 Reactive Extension for .NET Official release

2012 Elm - RFP for the web

2013 React

Facebook's system for Web UI components

2014 RxJava 1.0 - Port of Reactive Extensions (ReactiveX) to Java

2015 Flutter beta

2016 RxJava 2.0

2018 Flutter 1.0

2019 Swift UI, Android Compose

# ReactiveX

http://reactivex.io

Their claim

The Observer pattern done right

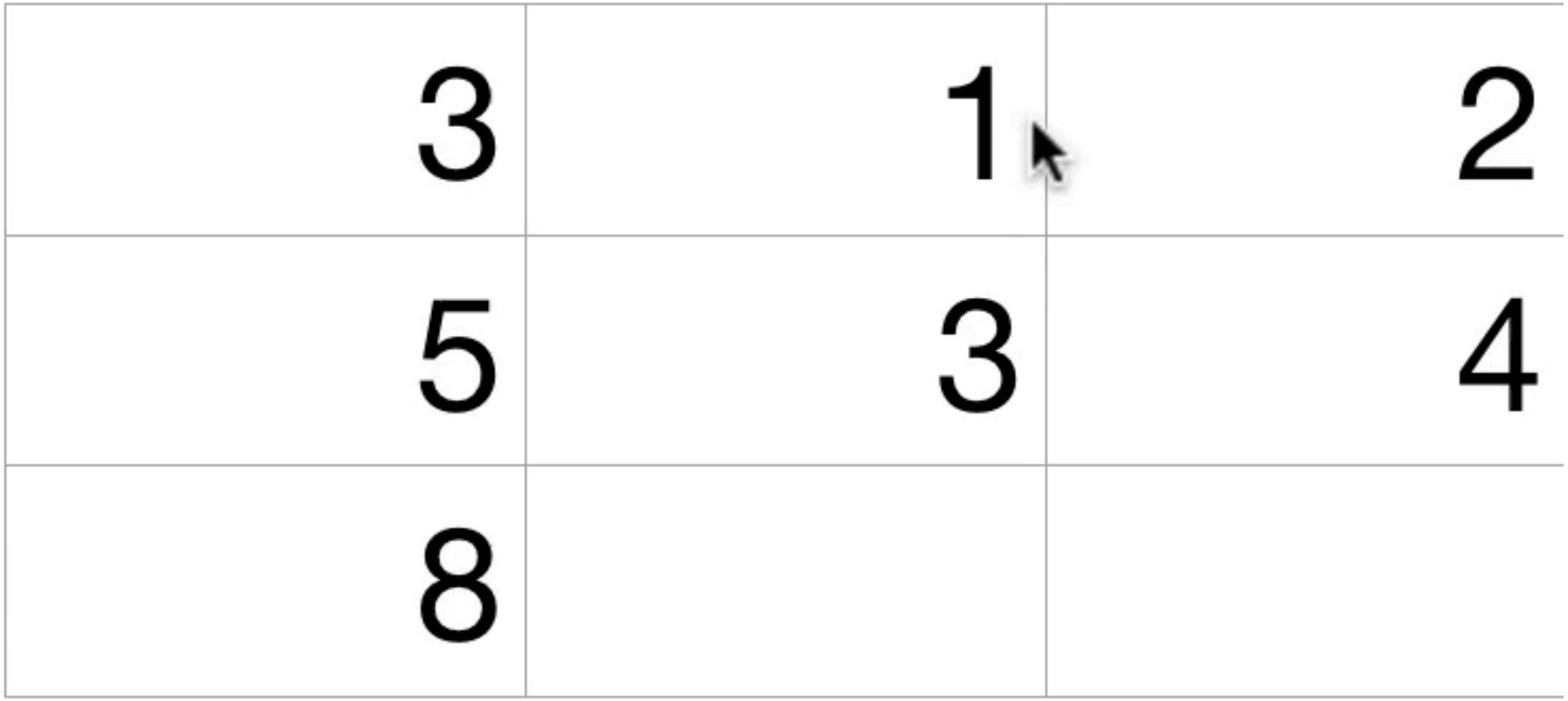
ReactiveX is a combination of the best ideas from
Observer pattern,
Iterator pattern,
Functional programming

Ported to multiple languages
- Basic ideas same
- Syntax differs

# Reactive Programming

datatypes that represent a value 'over time'

Spreadsheets



| | | |
|---|---|---|
| 3 | 1 | 2 |
| 5 | 3 | 4 |
| 8 | | |

# Reactive Programming

Spreadsheets

Elm

React (Facebook)

Reagent (Clojure)

Android Architecture Components

SwiftUI

Swift Combine

Flutter (Google)

Fuchsia (Google)

Akka

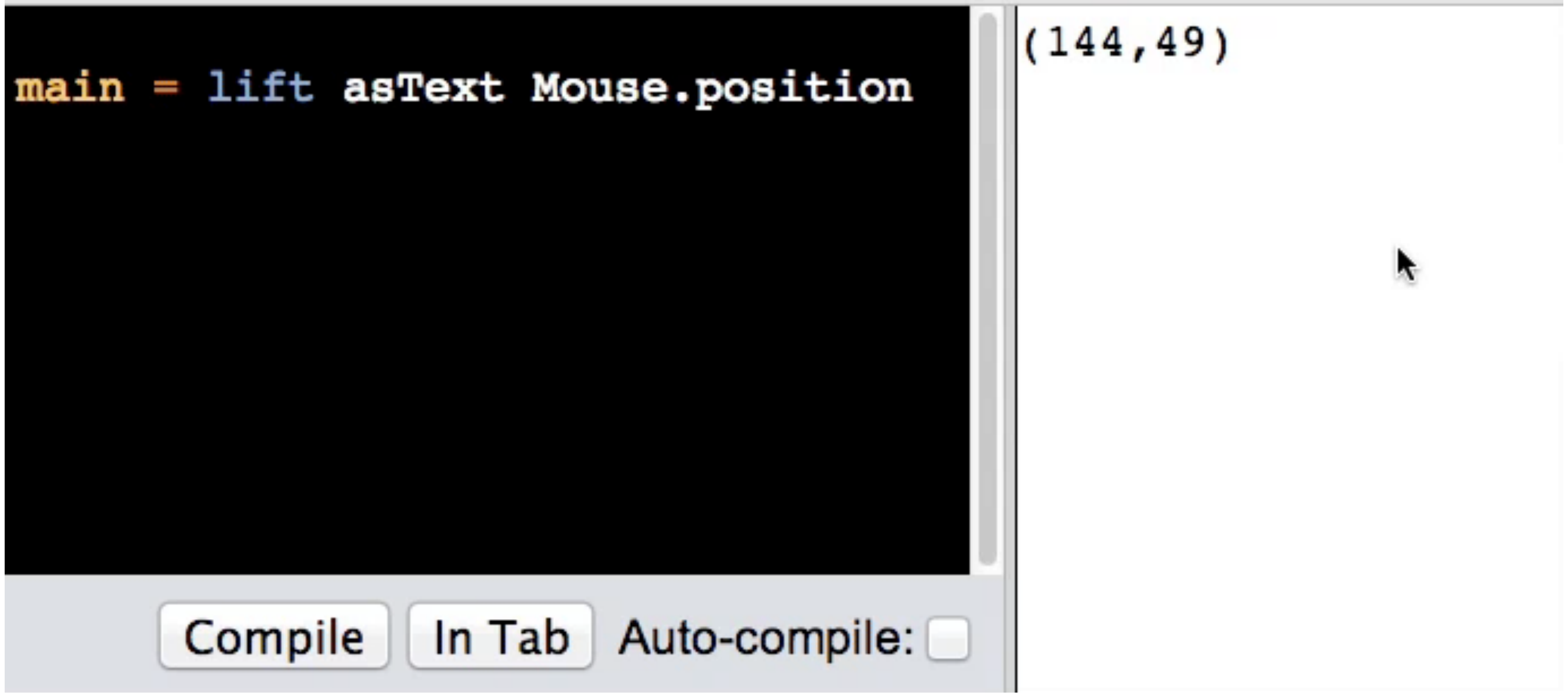
Java Flow

ReactiveX

- RxJava (35,500 GitHub stars)
- RxJS
- Rx.NET
- RxPY
- RxSwift
- RxKotlin
- RxAndroid (16,800 GitHub stars)
- RxCocoa

# Reactive Programming - Elm

datatypes that represent a value 'over time'

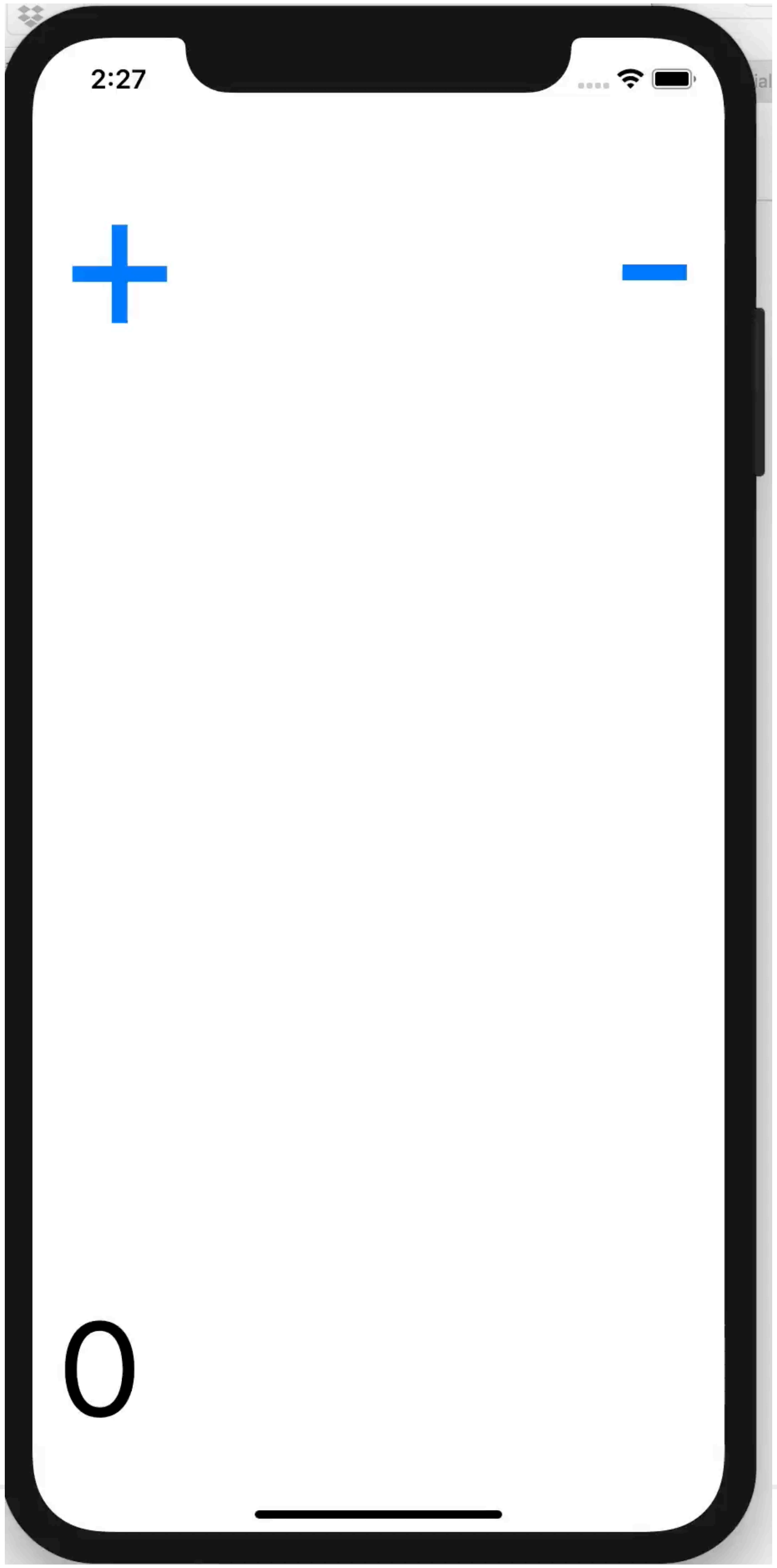


```
main = lift asText Mouse.position
```

```
(144,49)
```

# SwiftUI Example

```
import SwiftUI

struct ContentView : View {
    @State private var count : Int = 0

    var body: some View {
        VStack(alignment: .leading) {
            HStack {
                Button(action: {self.count = self.count + 1}){
                    Text("+").font(.system(size: 120))
                }
                Spacer()
                Button(action: {self.count = self.count - 1}){
                    Text("-").font(.system(size: 120))
                }
            }
            Spacer()
            Text("\(count)").font(.system(size: 80))
        }.padding()
    }
}
```



# Reactive Programming Concepts

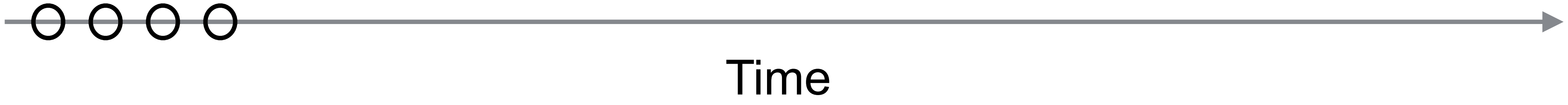
- Unify data types into stream of events/data
    - Events
    - Collections
    - Value changing
    - Asynchronous callbacks

- One-way data flows
    - React & Flux

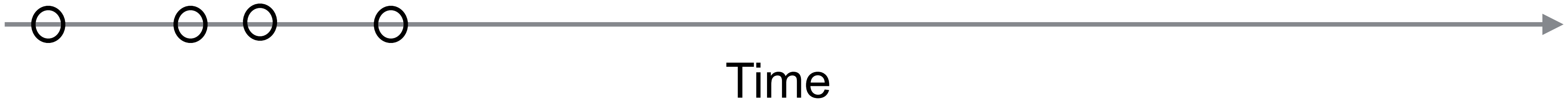
# Unify Data Types

```
Iterator<String> list = strings.iterator();
while (list.hasNext()){
    String element = list.next();
    processEachElement(element);
    }
}
```

When Elements are processes



But some elements take longer to process

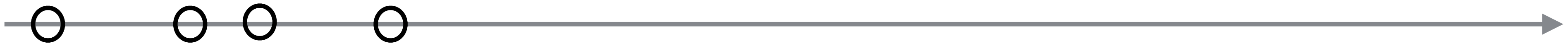


# Unify Data Types

```
class Foo {
  int bar;
```

bar changes value over time

When bar changed



Time

# The Basics

Subjects (Observables) generate a stream or flow of events/data

Streams

- Support map, filter and other functions

- Send three types of messages/events
  - onNext - the next data in the stream

  - onCompleted - The stream is at the end

  - onError - An error occurred

Observers subscribe to streams

- Some subjects give all the events/data to new subscribers
- Some give only current value and future changes
- Some subjects allow observers to tell subjects to slow down

# RxJava - Basic Classes

io.reactivex.Flowable:
    0..N flows, supporting Reactive-Streams and backpressure

io.reactivex.Observable:
    0..N flows, no backpressure

io.reactivex.Single:
    a flow of exactly 1 item or an error

io.reactivex.Completable:
    a flow without items but only a completion or error signal

io.reactivex.Maybe:
    a flow with no items, exactly one item or an error.

# RxJava HelloWorld

```
import io.reactivex.*;

public class Example {
    public static void main(String[] args) {
        Flowable.just("Hello world")
                    .subscribe(System.out::println);
    }
}
```

# RxJava Subscribe methods

```
subscribe(Consumer<? super T> onNext)

subscribe(Consumer<? super T> onNext,
          Consumer<? super Throwable> onError)

subscribe(Consumer<? super T> onNext,
          Consumer<? super Throwable> onError,
          Action onComplete)
```

Java Consumer

Lambda or function that has one argument and no return value

```
Consumer<String> print = text -> System.out.println(text);
print.accept("hello World");
```

```
import io.reactivex.*;

public class Example {
    public static void main(String[] args) {
        Flowable<Integer> flow = Flowable.range(1, 5)
                .map(v -> v * v)
                .filter(v -> v % 2 == 0);
        System.out.println("Start");
        flow.subscribe(System.out::println);
        System.out.println("Second");
        flow.subscribe(value -> System.out.println("Second " + value));
    }
}
```

Output

```
Start
4
16
Second
Second 4
Second 16
```

# Observables with Varying Number of Events

Flowable<Integer> flow = Flowable.range(1, 5)

flow has fixed number of data points

So more like iterator over a collection

How to create observable with varying number of data points/events

Emitters

Subjects

# Emitter Interface

onComplete()
onError(Throwable error)
onNext(T value)

# Example

```
import io.reactivex.*;

public class Example {
    public static void main(String[] args) {
        Observable<String> observable = Observable.create(emitter -> {
                emitter.onNext("A");
                emitter.onNext("B");
                emitter.onNext("B");
                emitter.onComplete();
                });
        System.out.println("Start");
        observable.subscribe(System.out::println,Throwable::printStackTrace,
                                                                    () -> System.out.println("Done"));
    }
}
```

# Longer Running Example

```
import io.reactivex.*;

public class Example {
    public static void main(String[] args) {
        Observable<Long> observable = Observable.create(emitter -> {
            while (!emitter.isDisposed()) {
                long time = System.currentTimeMillis();
                emitter.onNext(time);
                if (time % 2 != 0) {
                    emitter.onError(new IllegalStateException("Odd millisecond!"));
                    break;
                }
            }
        });
        System.out.println("Start");
        observable.subscribe(System.out::println,Throwable::printStackTrace);
    }
}
```

# Important Notes

Data generation all done in lambda

- But could have called a method on an object

Observable just knows to pass emitter to observer

# Subjects

Subjects are

- Observable
- Observers

Multiple Types

- BehaviorSubject
  - Sends current value and future values to observers
- PublishSubject
  - Sends future values to observers
- ReplaySubject
  - Sends past, current and future values to observers

# PublishSubject Example

```
import io.reactivex.subjects.PublishSubject;
import io.reactivex.subjects.Subject;

public class Example {
    public static void main(String[] args) {
        Subject<String> subject = PublishSubject.create();
        subject.subscribe(System.out::println,
                          Throwable::printStackTrace,
                          () ->System.out.println("Done"));

        subject.onNext("Start");
        subject.onNext("A");

        subject.subscribe(text -> System.out.println("Later " + text));
        subject.onNext("B");
        subject.onNext("C");
        subject.onComplete();
    }
}
```

Output

```
Start
A
B
Later B
C
Later C
Done
```

# BehaviorSubject Example

```
import io.reactivex.subjects.BehaviorSubject;
import io.reactivex.subjects.Subject;

public class Example {
    public static void main(String[] args) {
        Subject<String> subject = BehaviorSubject.create();
        subject.subscribe(System.out::println,
                                Throwable::printStackTrace,
                                () ->System.out.println("Done"));

        subject.onNext("Start");
        subject.onNext("A");

        subject.subscribe(text -> System.out.println("Later " + text));
        subject.onNext("B");
        subject.onNext("C");
        subject.onComplete();
    }
}
```

Output

```
Start
A
Later A
B
Later B
C
Later C
Done
```

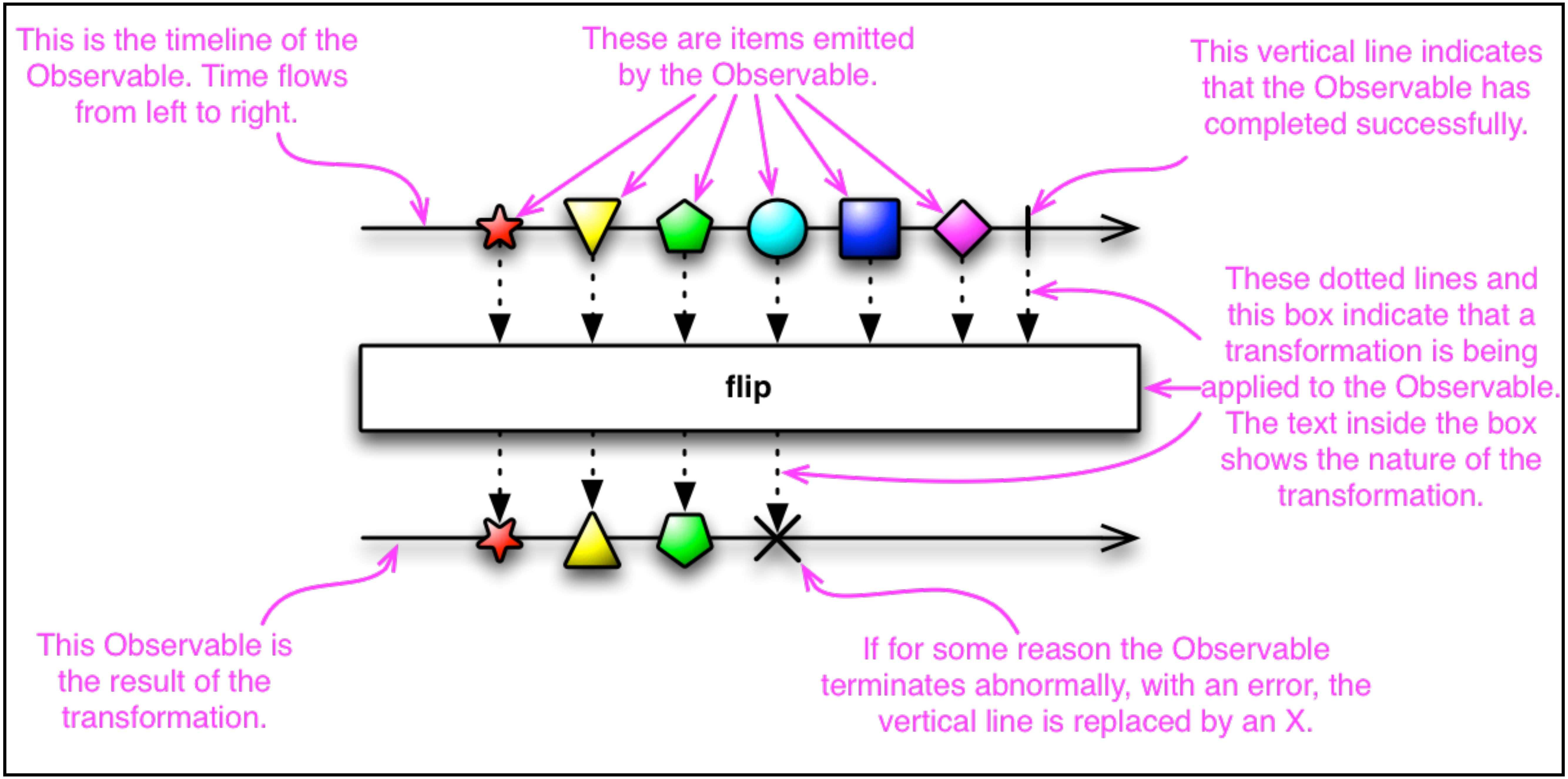
# ReplaySubject Example

```
import io.reactivex.subjects.ReplaySubject;
import io.reactivex.subjects.Subject;

public class Example {
    public static void main(String[] args) {
        Subject<String> subject = ReplaySubject.create();
        subject.subscribe(System.out::println,
                          Throwable::printStackTrace,
                          () ->System.out.println("Done"));

        subject.onNext("Start");
        subject.onNext("A");

        subject.subscribe(text -> System.out.println("Later " + text));
        subject.onNext("B");
        subject.onNext("C");
        subject.onComplete();
    }
}
```

Output

```
Start
A
Later Start
Later A
B
Later B
C
Later C
Done
```

# Diagrams



This is the timeline of the Observable. Time flows from left to right.

These are items emitted by the Observable.

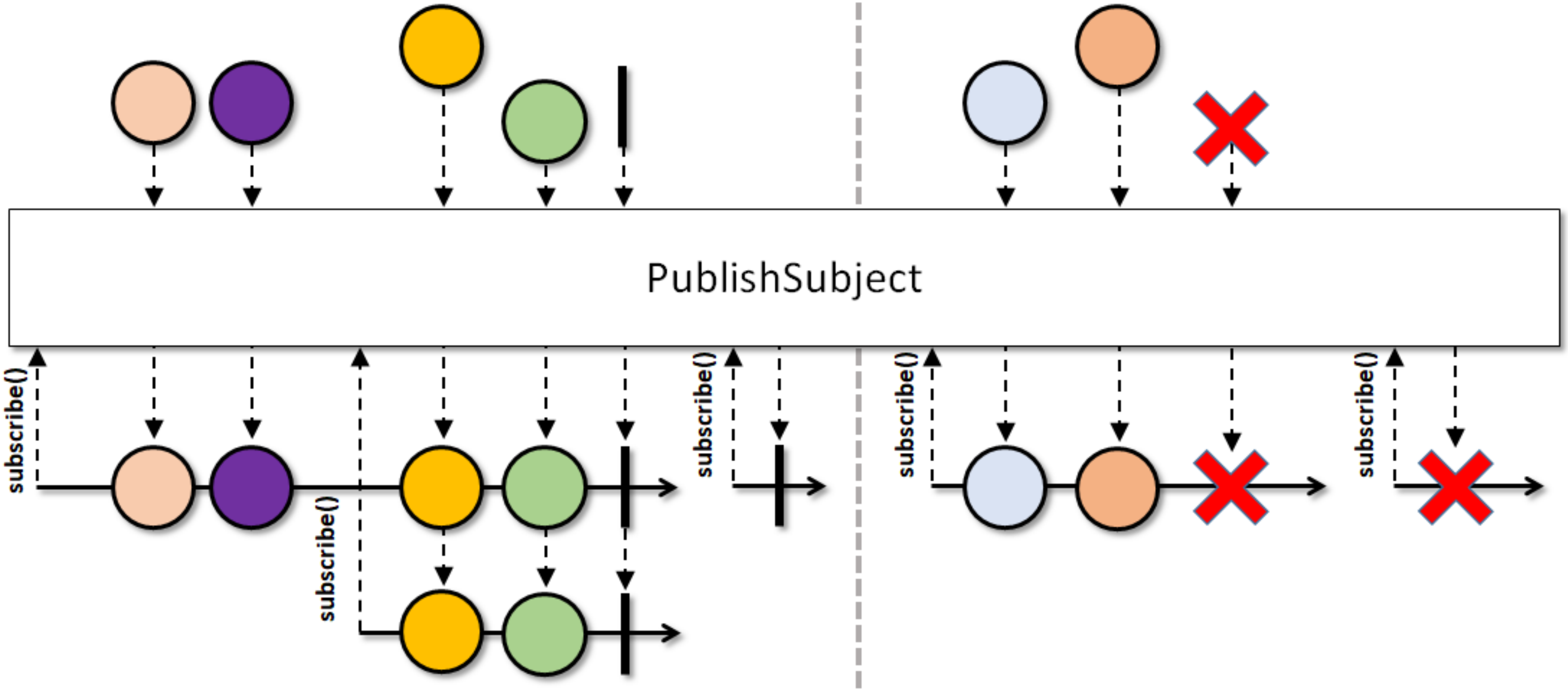
This vertical line indicates that the Observable has completed successfully.

These dotted lines and this box indicate that a transformation is being applied to the Observable. The text inside the box shows the nature of the transformation.

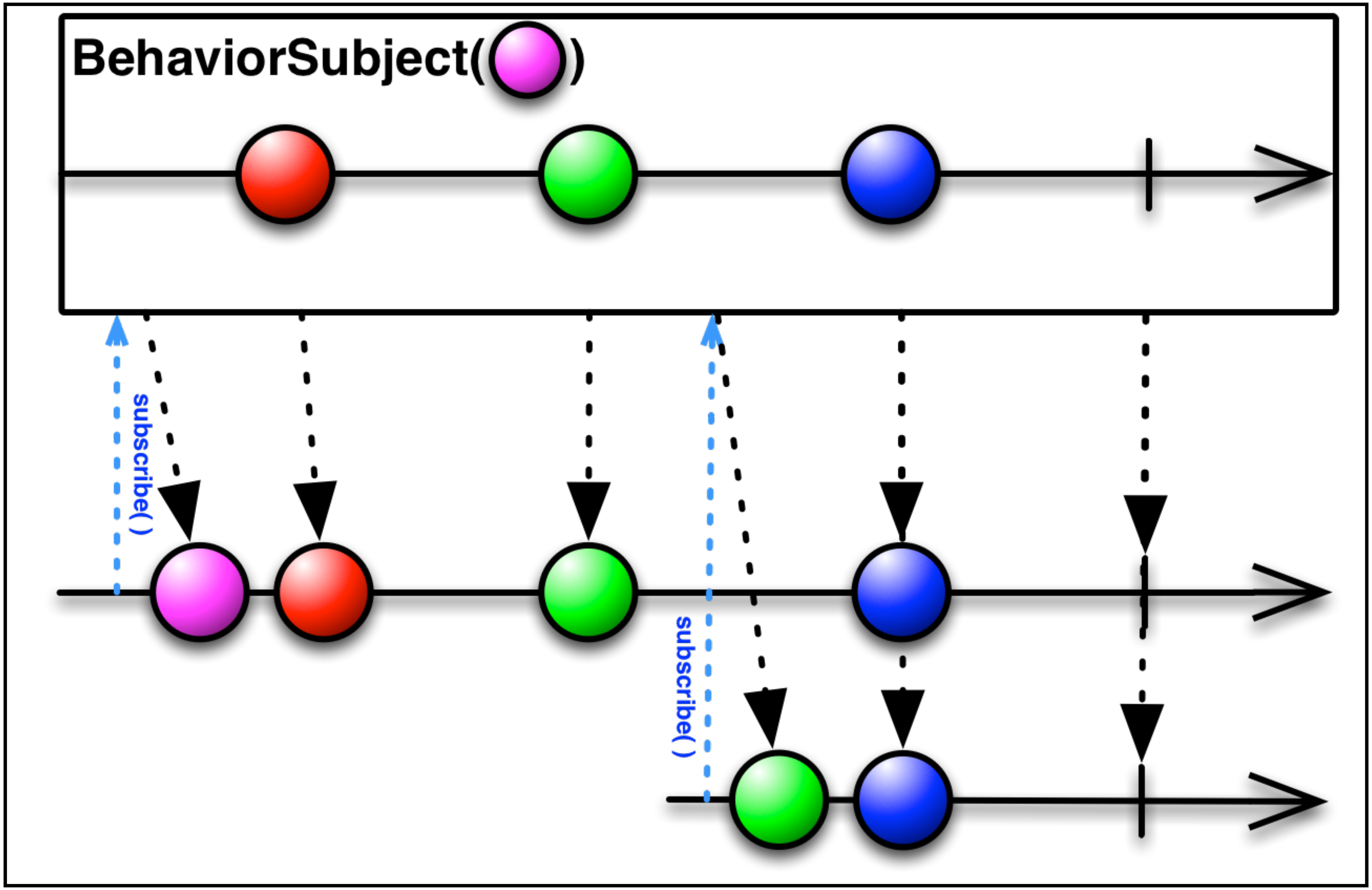
This Observable is the result of the transformation.

If for some reason the Observable terminates abnormally, with an error, the vertical line is replaced by an X.

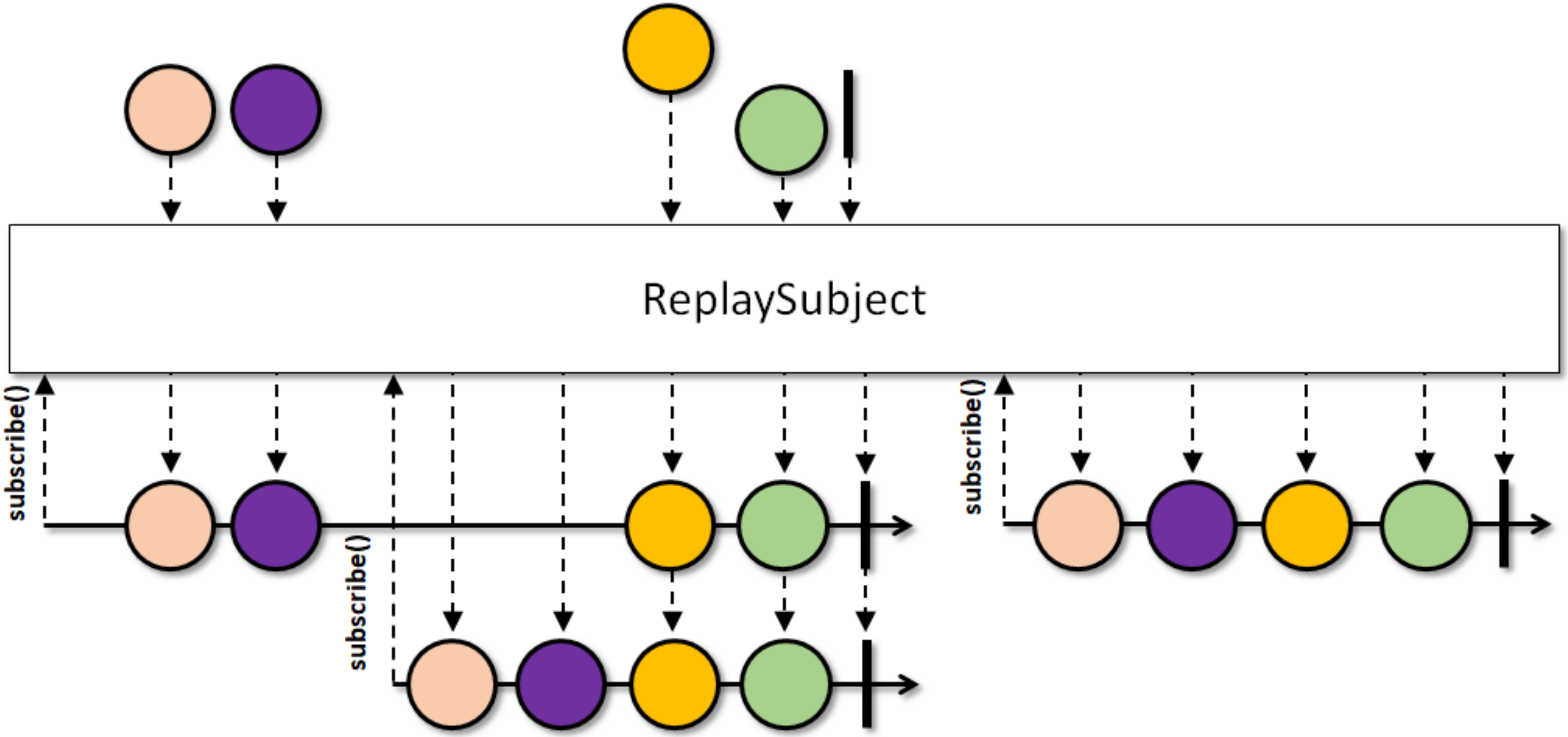
# PublishSubject



# BehaviorSubject



# ReplaySubject



# RxPy

```
from rx import Observable

source = Observable.of("Alpha", "Beta", "Gamma", "Delta", "Epsilon")

source.subscribe(on_next=lambda value: print("Received {0}".format(value)),
                 on_completed=lambda: print("Done!"),
                 on_error=lambda error: print("Error Occurred: {0}".format(error))
                 )

source.subscribe(on_completed=lambda: print("Done!"),
                   on_next=lambda value: print("Received {0}".format(value))
                 )

source.subscribe(lambda value: print("Received {0}".format(value)))


source.subscribe(print)
```

# RxPy

```
from rx import Observable

xs = Observable.from_(range(10))
d = xs.filter(lambda x: x % 2)
        .map(lambda x: x * 2)
        .subscribe(print)
```

```
2
6
10
14
18
```

```
xs = Observable.range(1, 5)
ys = Observable.from_("abcde")
zs = xs.merge(ys).subscribe(print)
```

```
a
1
b
2
c
3
d
4
e
5
```

# PublishSubject

```
from rx.subjects import Subject

stream = Subject()
stream.subscribe(on_next=lambda value: print("Received {0}".format(value)),
                 on_completed=lambda: print("Done!"),
                 on_error=lambda error: print("Error Occurred: {0}".format(error))
                 )
stream.on_next("Start")
stream.on_next("A")
d = stream.subscribe(lambda x: print("Got: %s" % x))

stream.on_next("B")

d.dispose()
stream.on_next("C")
stream.on_next(10)

stream.on_completed()
```

```
Received Start
Received A
Received B
Got: B
Received C
Received 10
Done!
```

# ReplaySubject

```
from rx.subjects import ReplaySubject

stream = ReplaySubject()
stream.subscribe(on_next=lambda value: print("Received {0}".format(value)),
                 on_completed=lambda: print("Done!"),
                 on_error=lambda error: print("Error Occurred: {0}".format(error))
                 )
stream.on_next("Start")
stream.on_next("A")
d = stream.subscribe(lambda x: print("Got: %s" % x))

stream.on_next("B")

d.dispose()
stream.on_next("C")
stream.on_next(10)

stream.on_completed()
```

```
Received Start
Received A
Got: Start
Got: A
Received B
Got: B
Received C
Received 10
Done!
```

# RxSwift

```
import RxSwift

let dataSequence = Observable.from([1, 2, 3])
dataSequence.subscribe(onNext: {print($0)})
```

```
1
2
3
```

```
dataSequence.subscribe(
                       onNext: {print($0)},
                       onCompleted: {print("Done")})
```

```
1
2
3
Done
```

```
dataSequence
   .map {$0 + 1}
   .scan(0) {$0 + $1}
   .subscribe(onNext: {print($0)},onCompleted: {print("Done")})
```

```
2
5
9
Done
```

# PublishSubject

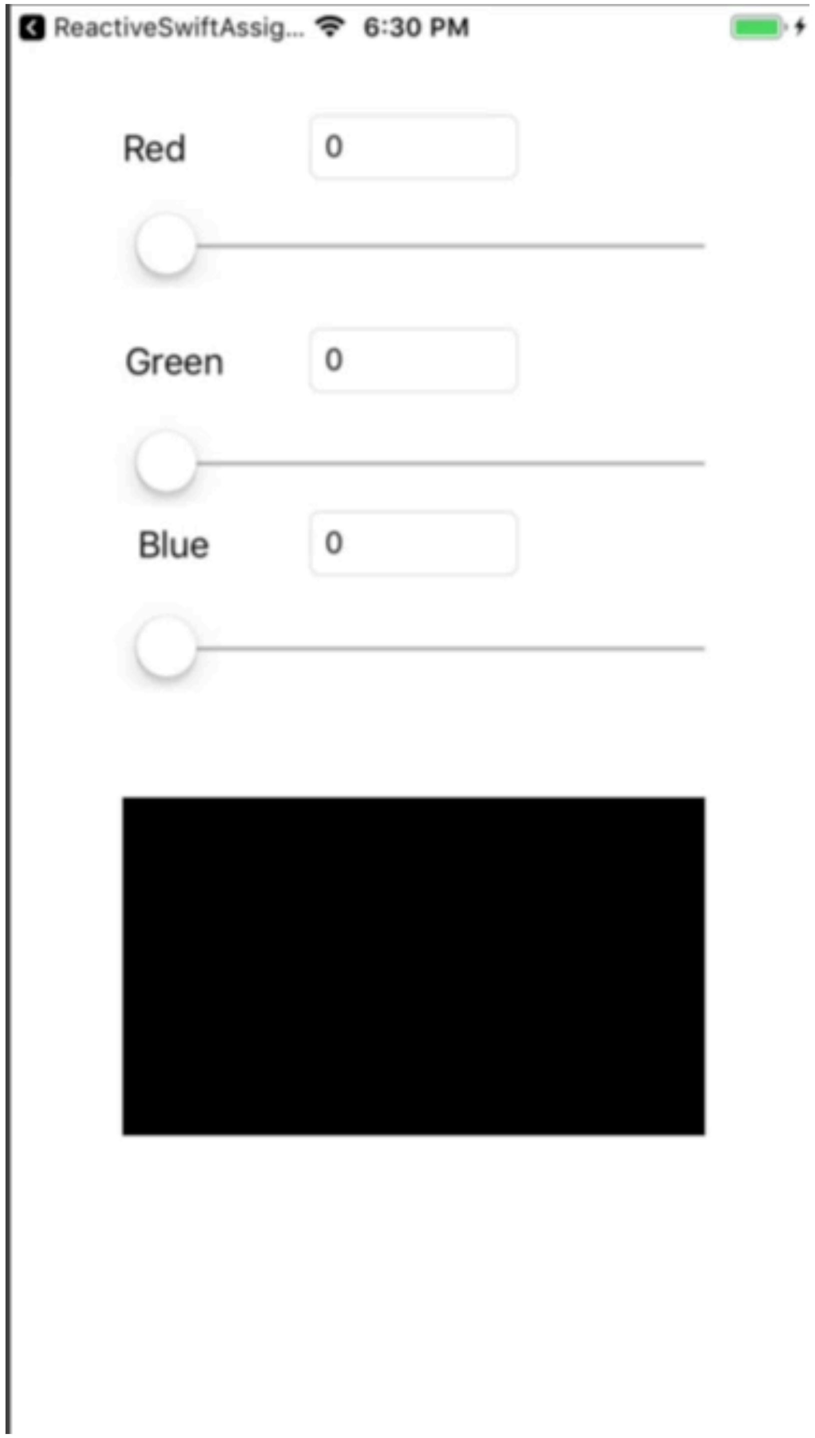
```
let subject = PublishSubject<Int>()
subject.subscribe(onNext: {print("Subject = \($0)")},
                  onCompleted: {print("Done")})

subject.map {$0 + 10}
      .subscribe(onNext: {print("Plus 10 = \($0) ")})

print("Start")
subject.onNext(2)
print("After 2")
subject.onNext(4)
print("No more")
```

```
Start
Subject = 2
Plus 10 = 12
After 2
Subject = 4
Plus 10 = 14
No more
```

# Network Calls

```
if let url = URL(string: "https://bismarck.sdsu.edu/registration/subjectlist") {
    let request = URLRequest(url: url)

    let responseJSON = URLSession.shared.rx.json(request: request)

    let cancelRequest = responseJSON.subscribe(
                                                onNext: { json in print(json) },
                                                onCompleted: {print("Done")})
}
```

# Sample App



Specs

Color values

- Integers
- 0 - 100

Change in slider

- Changes text field
- Changes color of box

Change in text field

- Changes slider
- Changes color of box

# Standard Solution

Have reference to

- redSlider
- greenSlider
- blueSlider

- redText(field)
- greenText(field)
- blueText(field)

Color class

- Stores value of red, green, blue

Have callback function called on change

- redSlider
- greenSlider
- blueSlider

- redText(field)
- greenText(field)
- blueText(field)

# Standard Solution

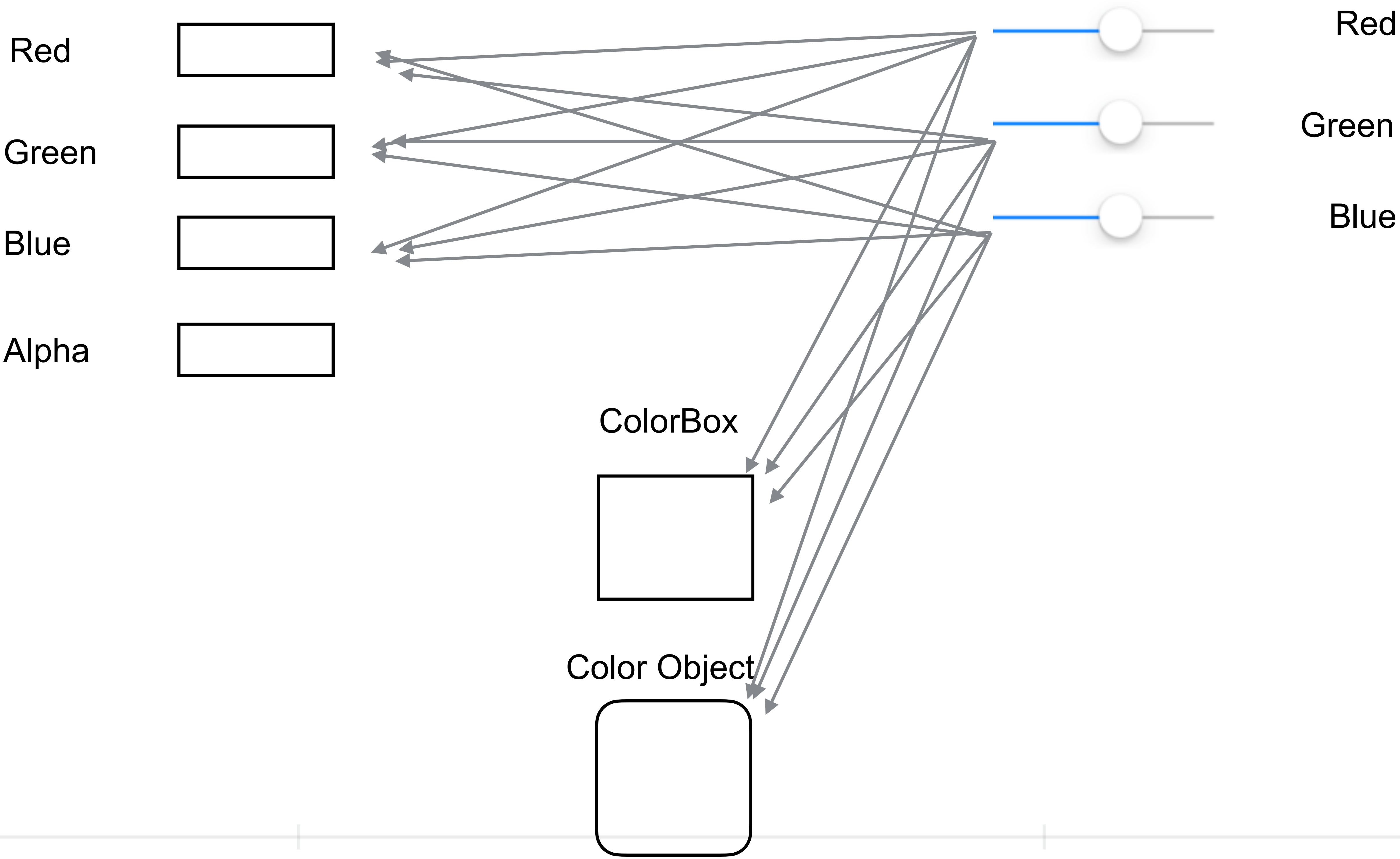
- Slider call back function - each slider
  - Called when slider changes
  - Get value of slider
  - Convert value to string
    - Set text field with string value of slider
  - Change color of box
  - Store the current color value

- Textfield call back function
  - Called when user types character or deletes a character
  - Get value of textfield
  - Convert string to float
    - Set value of slider to float value of textfield
  - Change color of box
  - Store the current color value

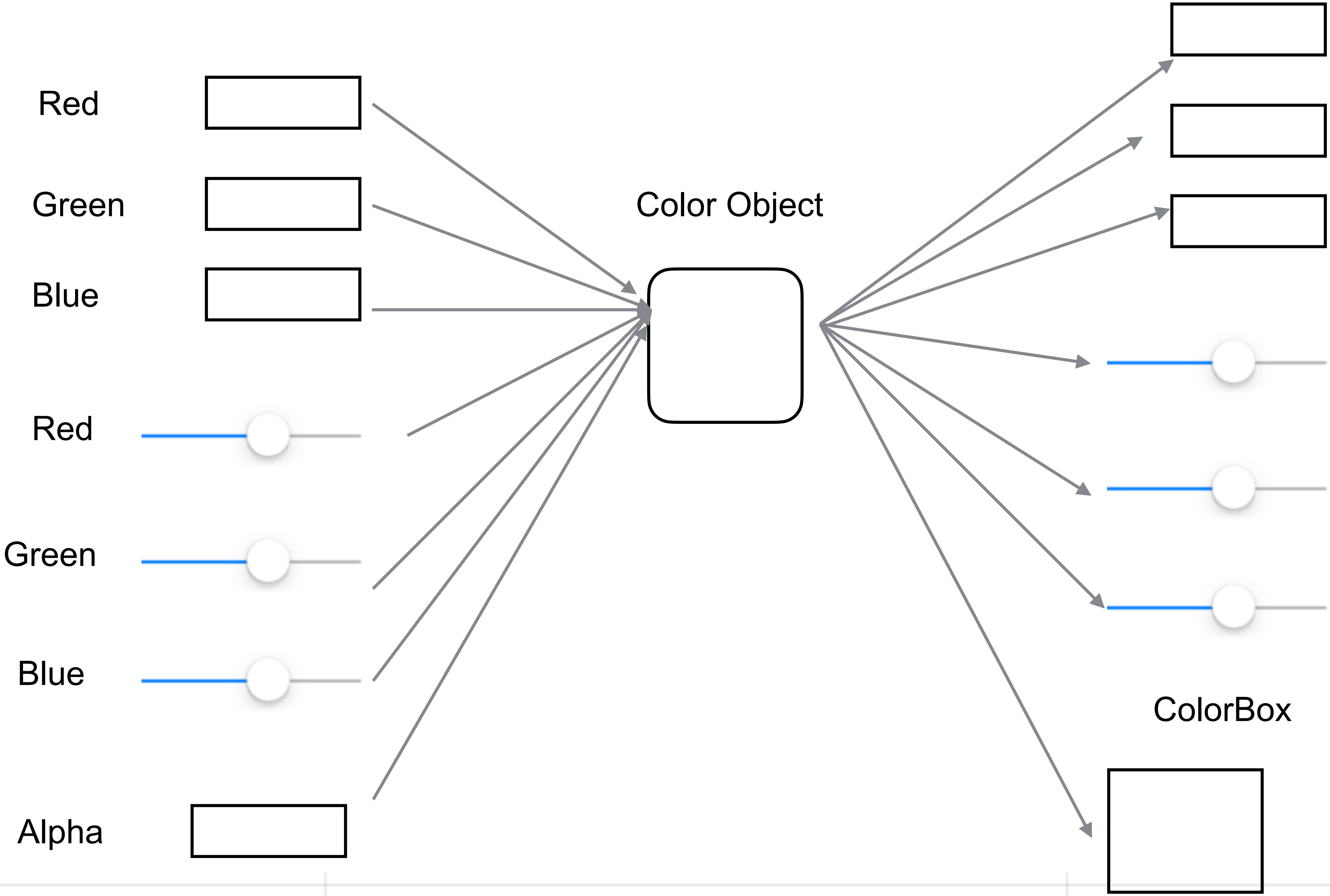
# One Slider Callback

```
@IBAction func redSliderChanged(_ sender: Any) {
    redText.text = Int(redSlider.value).description
    guard   let redString = redText?.text,
              let red = Double(redString),
              let greenString = greenText?.text,
              let green = Double(greenString),
              let blueString = blueText?.text,
              let blue = Double(blueString) else {
        return
    }
    colorBox.backgroundColor = UIColor(red: CGFloat(red)/100,
                                       green: CGFloat(green)/100,
                                       blue: CGFloat(blue)/100, alpha: 1)
    color.red = red
}
```

# Slider Callback Functions



# Color as Subject/Observable



# One Slider Callback

```
@IBAction func redSliderChanged(_ sender: Any) {
    color.red = Int(redSlider.value)
}
```

# Color Updating UI

```
override func viewDidLoad() {
    super.viewDidLoad()
    color.observable.subscribe(onNext: {(type) in
        self.colorBox.backgroundColor = self.color.asUIColor()
        switch type {
            case .Red:
                self.redText.text = String(self.color.red)
                self.redSlider.value = Float(self.color.red)
            case .Green:
                self.greenText.text = String(self.color.green)
                self.greenSlider.value = Float(self.color.green)
            case .Blue:
                self.blueText.text = String(self.color.blue)
                self.blueSlider.value = Float(self.color.blue)
       }
    })
```

# Functional Reactive Programming

Mathematical Variables

x = y

x remains equal to y

```
redSlider.value = Float(self.color.red)
```

So why can't we mean redSlider.value is always the same value as:
    Float(self.color.red)

# ReactiveSwift

Reactive library for Swift

Same ideas as ReactiveX (RxSwift)

- Uses different terms for same ideas

Not tied to ReactiveX

- So syntax is more Swift-like
- Claims simpler than RxSwift

# ReactiveSwift <~ operator

```
redSlider.reactive.value <~ color.red.map {Float($0)}
```

Whenever color.red changes then perform

```
    redSlider.reactive.value = color.red.map {Float($0)}
```

```
color.redProperty.map {Float($0)}.signal.observeValues({self.redSlider.value = $0})
```

```
overload func viewDidLoad() {
    redSlider.reactive.value <~ color.red.map {Float($0)}
    redText.reactive.text <~ color.red.map { String($0)}
    greenSlider.reactive.value <~ color.green.map {Float($0)}
    greenText.reactive.text <~ color.green.map { String($0)}
    blueSlider.reactive.value <~ color.blue.map {Float($0)}
    blueText.reactive.text <~ color.blue.map { String($0)}


    //update data when sliders move
    color.red <~ redSlider.reactive.values.map {Int($0)}
    color.green <~ greenSlider.reactive.values.map {Int($0)}
    color.blue <~ blueSlider.reactive.values.map {Int($0)}

    //update data when text fields change
    color.redProperty <~ redText.reactive.continuousTextValues.map {
                                self.stringToInt(value: $0)}
    color.greenProperty <~ greenText.reactive.continuousTextValues.map {
                                self.stringToInt(value: $0)}
    color.blueProperty <~ blueText.reactive.continuousTextValues.map {
                                self.stringToInt(value: $0)}
```

```
class Color {
    var red: MutableProperty<Int> = MutableProperty(0)
    var green: MutableProperty<Int> = MutableProperty(0)
    var blue: MutableProperty<Int> = MutableProperty(0)

    convenience init() {
        self.init(red: 30, green: 40, blue: 100)
    }

    init(red: Int, green: Int, blue: Int) {
        self.red.value = red
        self.green.value = green
        self.blue.value = blue
    }
}
```

Property generates a Signal(Channel)
Observers can listen for events
    on the signal(channel)

# What We Want Done vs How To Do it

```
@IBAction func redSliderChanged(_ sender: Any) {
     let redValue: Float = redSlider.value
    color.red = Int(redValue)
}
```

```
redSlider.reactive.value <~ color.red.map {Float($0)}
```

# Reactive Programming

New terms

- Channels, Signals
- Events
- Producers
- etc

Needs to rethink how to write code

# Aside

```
color.red.signal.observeValues
   {self.redSlider.value = Float($0)
     self.redText.text = String($0)}
color.green.signal.observeValues
   {self.greenSlider.value = Float($0)
     self.greenText.text = String($0)}
color.blue.signal.observeValues
   {self.blueSlider.value = Float($0)
     self.blueText.text = String($0)}
```

verses

```
redSlider.reactive.value <~ color.red.map {Float($0)}
redText.reactive.text <~ color.red.map { String($0)}
greenSlider.reactive.value <~ color.green.map {Float($0)}
greenText.reactive.text <~ color.green.map { String($0)}
blueSlider.reactive.value <~ color.blue.map {Float($0)}
blueText.reactive.text <~ color.blue.map { String($0)}
```