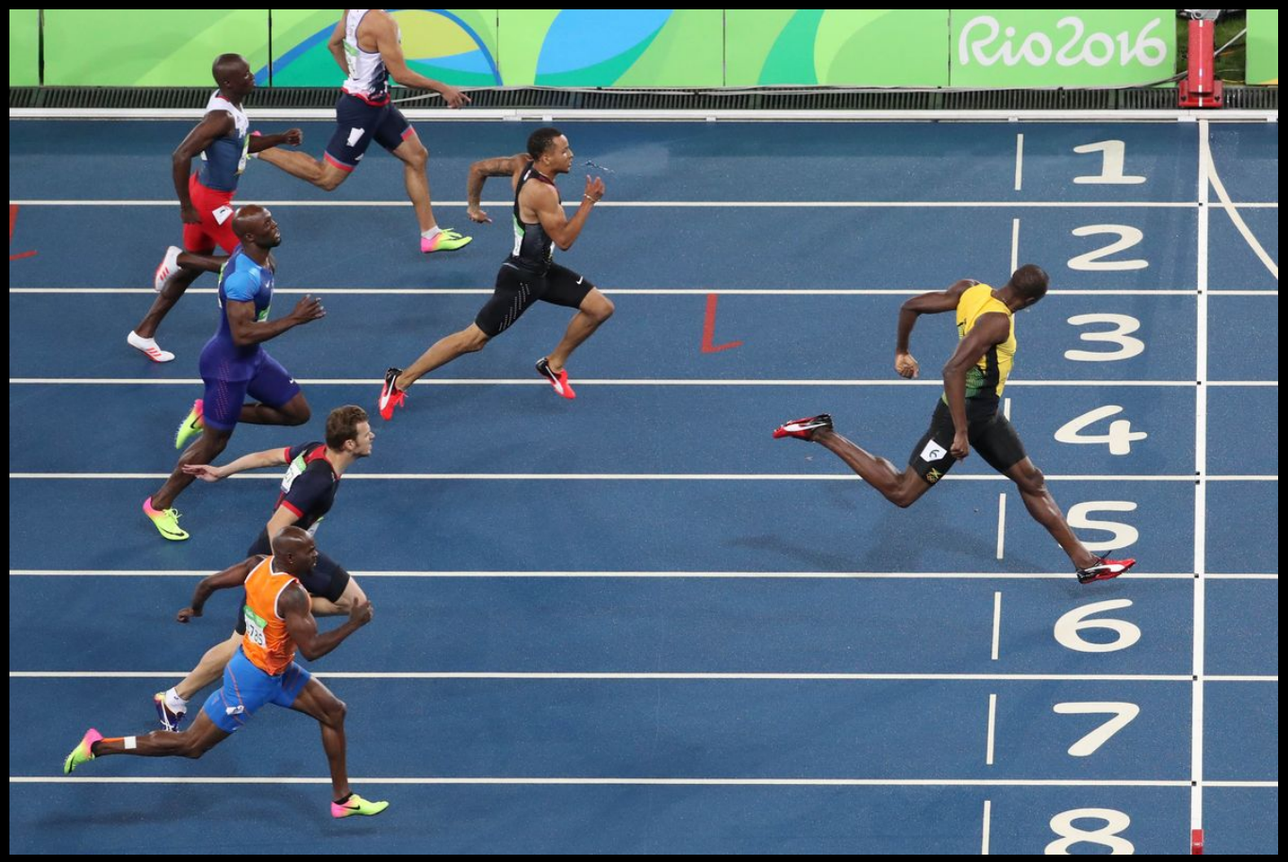
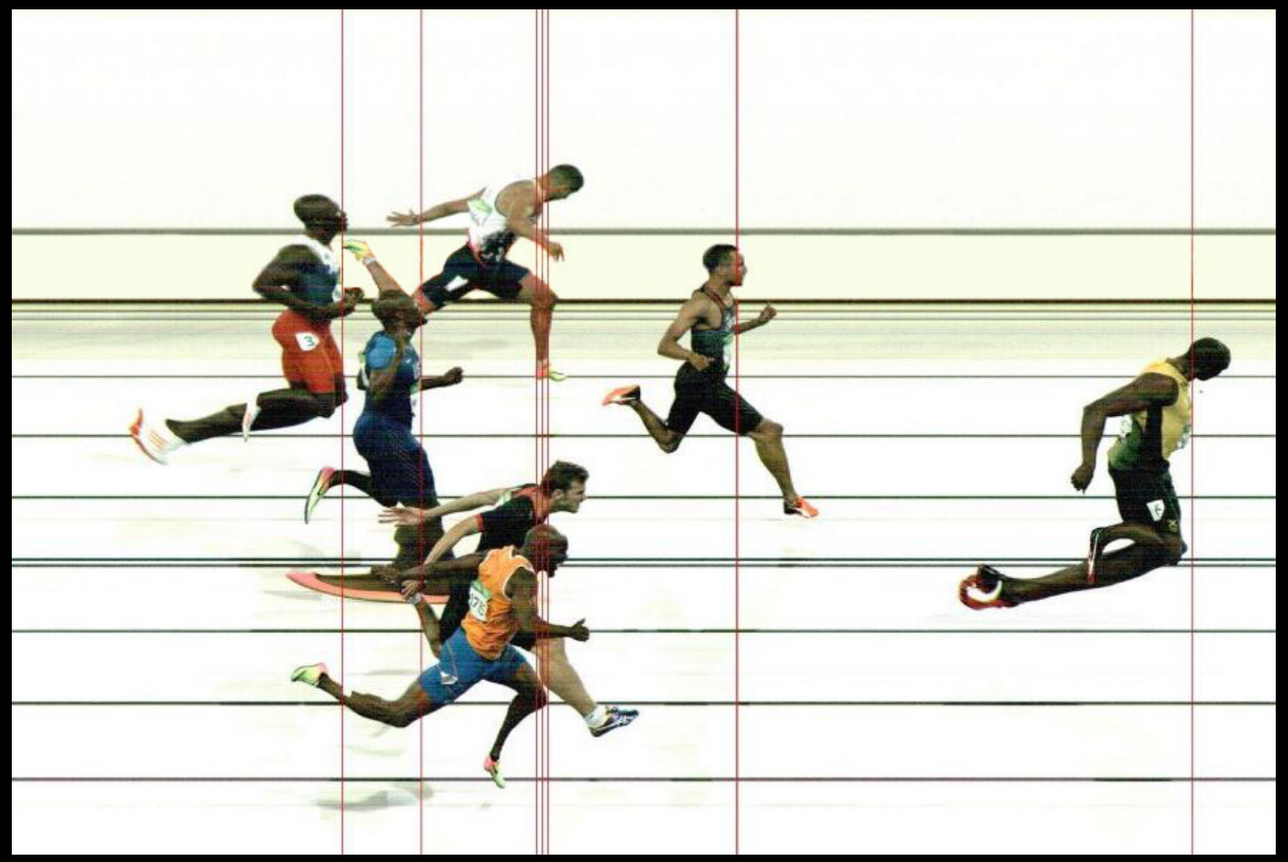
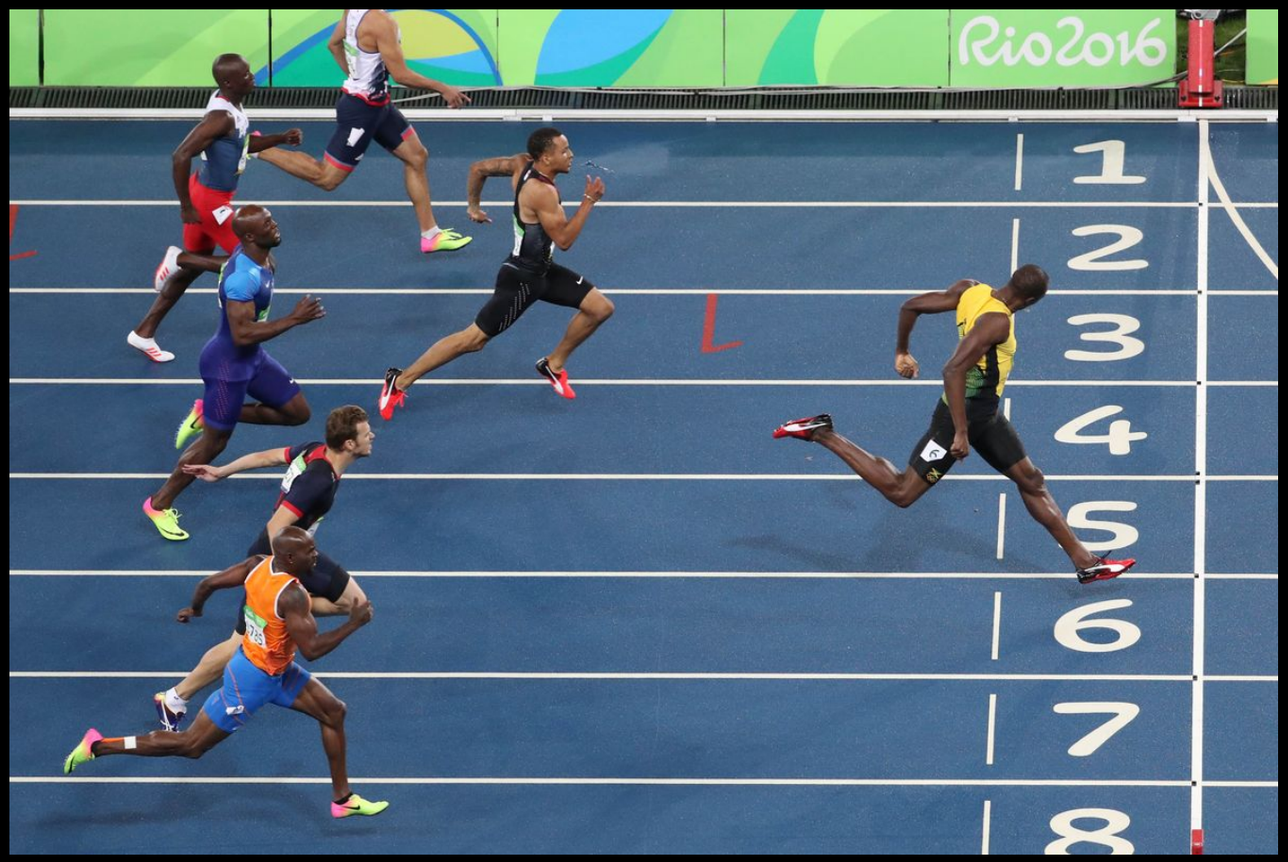
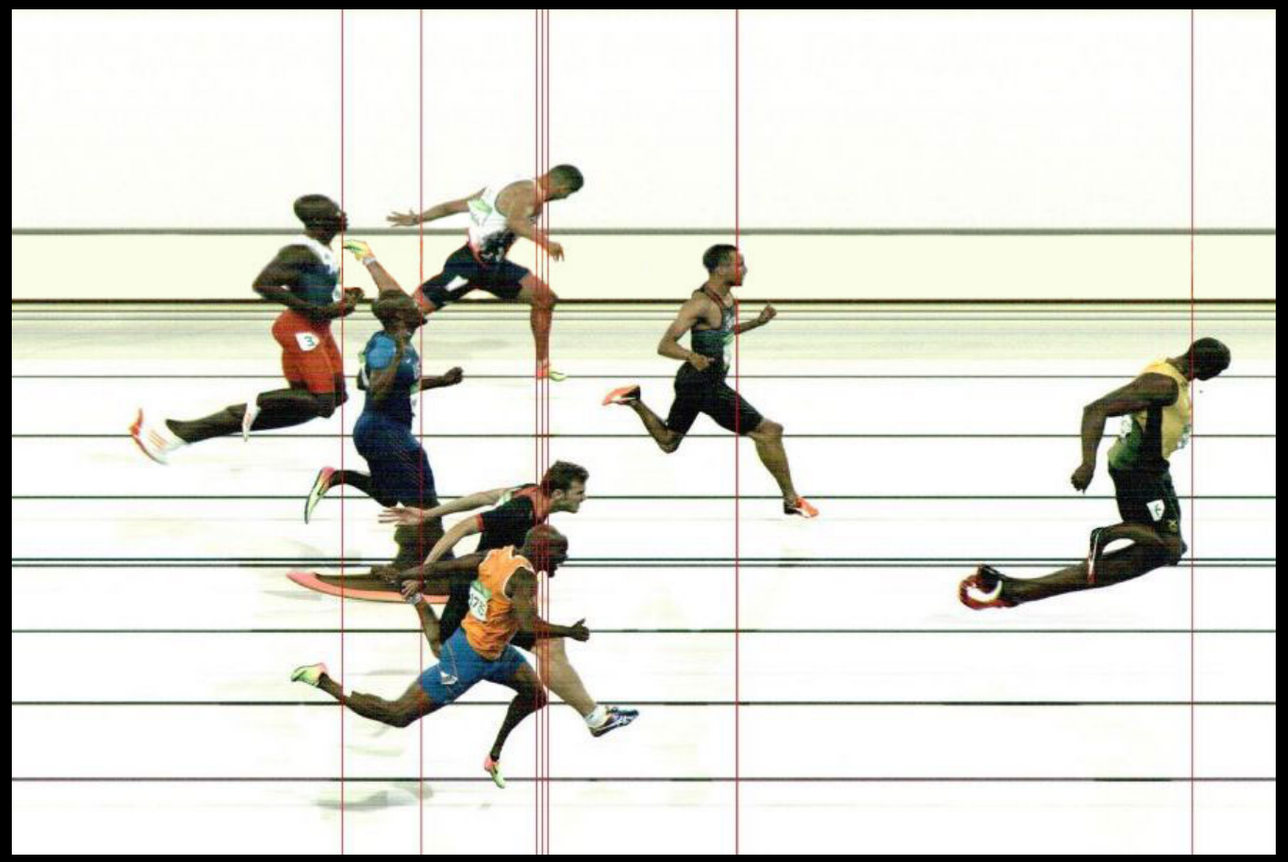
Reactive programming from scratch

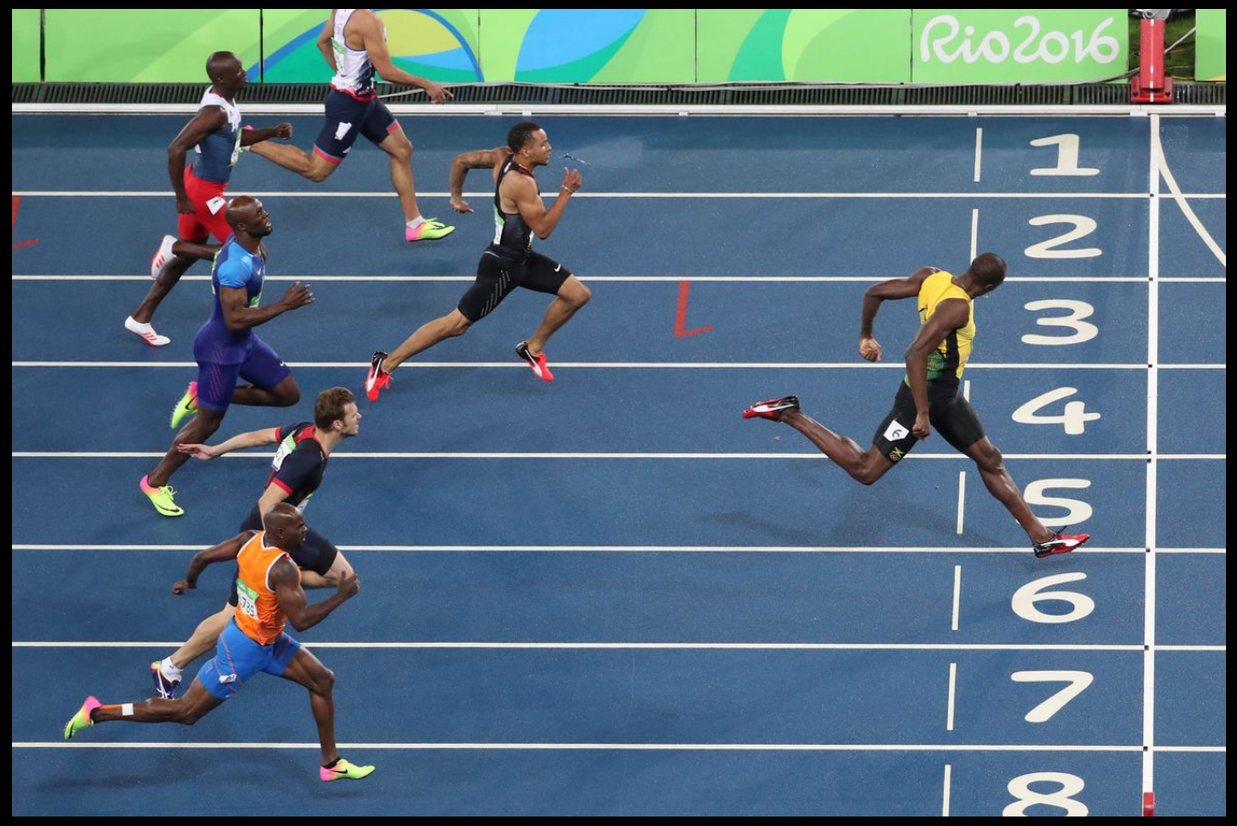
@thomvis


Rio2016
1
2
3
4
5
6
7
8

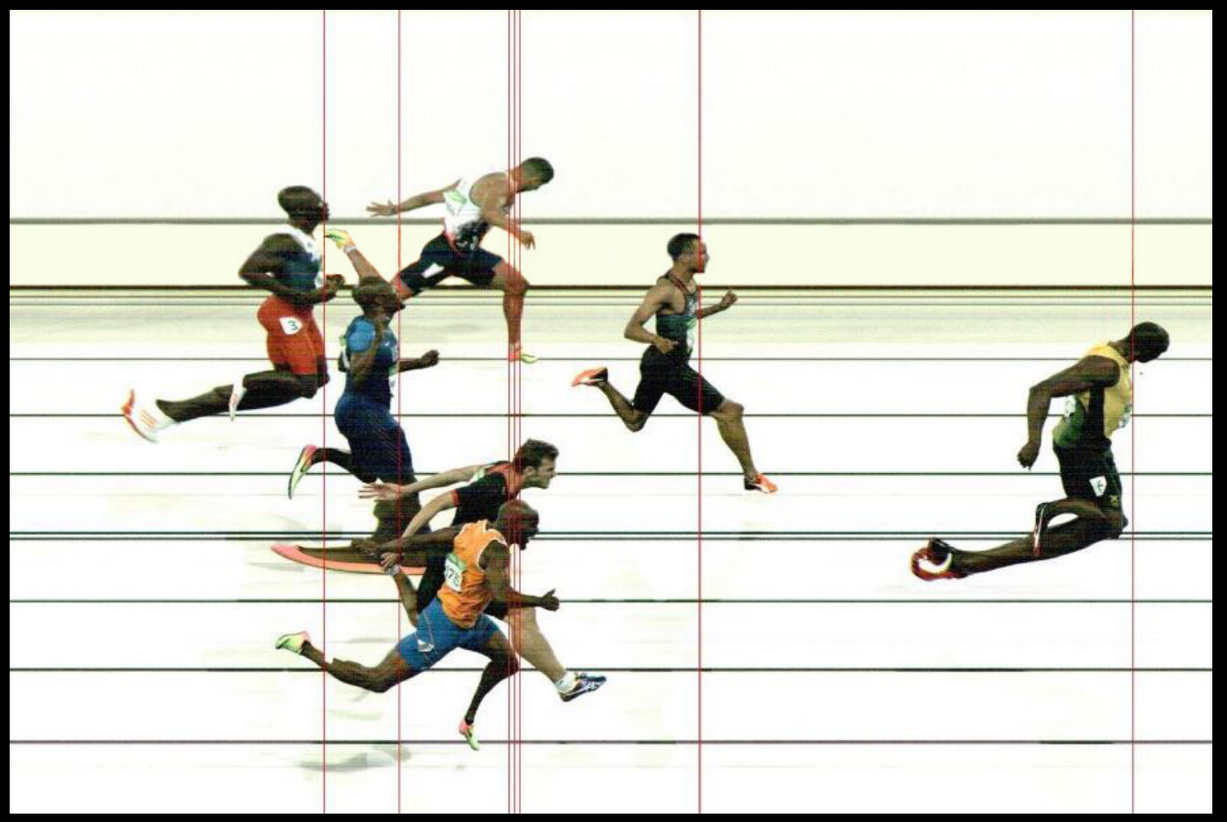



Rio2016
1
2
3
4
5
6
7
8

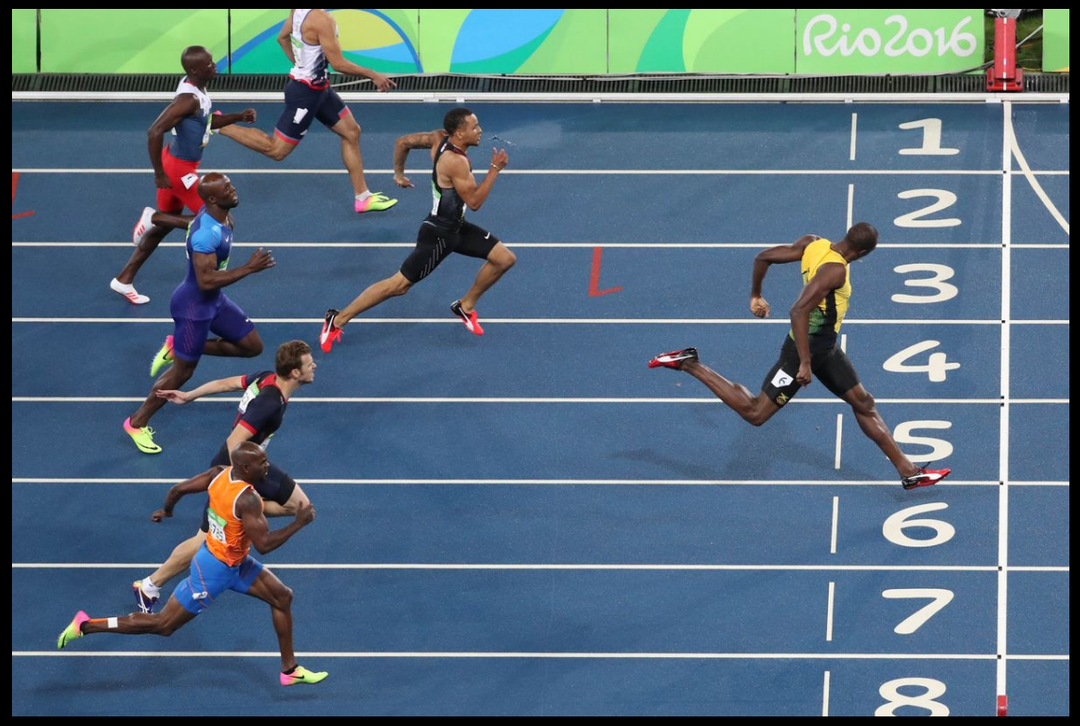


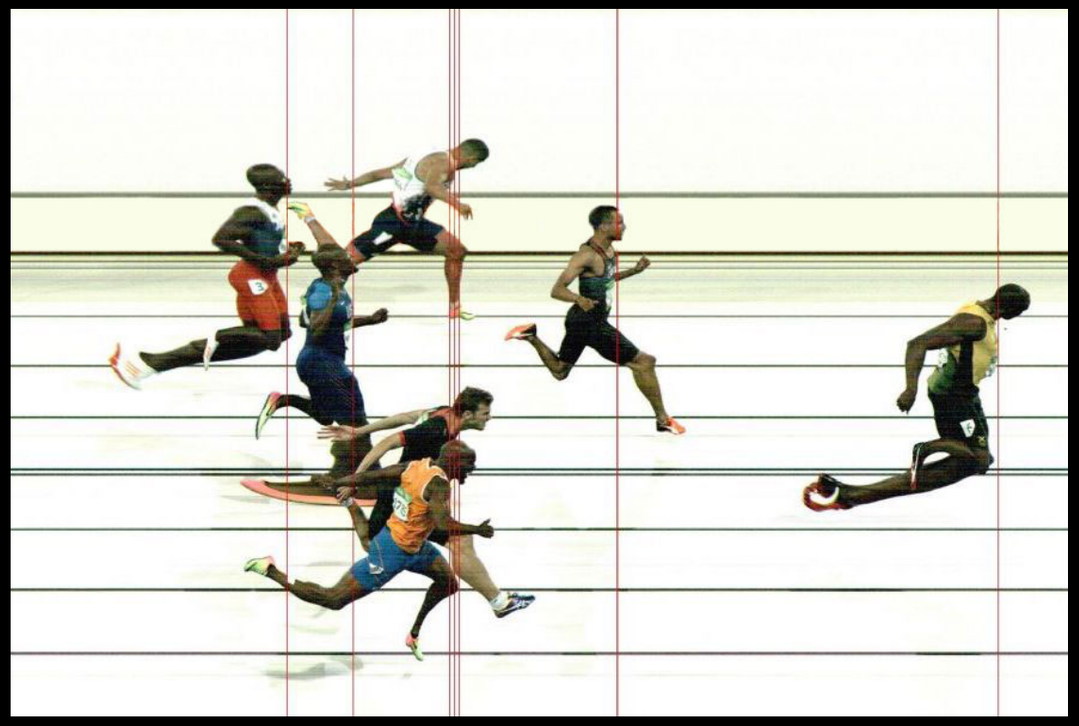



```
let result: [String] = getRaceResult()
// ["Bolt", "De Grasse", "Lemaitre", "Gemili", "Martina"]
```



```
race.onAthleteDidFinish { athlete in
    // athlete is Bolt, De Grasse, Lemaitre, Gemili and Martina
}
```





```
["Bolt", "De Grasse", "Lemaitre", "Gemili", "Martina"]
```

Everyting is a Sequence

```swift
getRaceResult()
    .prefix(3)
    .enumerated()
    .map { offset, elem in
        return "\(offset+1). \(elem)"
    }
    .joined(separator: "\n")

// 1. Bolt
// 2. De Grasse
// 3. Lemaitre
```

```swift
var athletes: [String] = []
var res: String? = nil
race.onAthleteDidFinish { athlete in
    if athletes.count < 3 {
        athletes.append("\(athletes.count + 1). \(athlete)")
    } else if res == nil {
        res = athletes.joined(separator: "\n")
    }
}

// 1. Bolt
// 2. De Grasse
// 3. Lemaitre
```

```swift
button.addTarget(self, action: #selector(buttonTap), for: [.touchUpInside])
// [tap, tap tap]
```

```swift
let manager = CLLocationManager()
manager.delegate = self
// [(52.47695990, 13.43936600), (52.51716100, 13.44998000)]
```

```swift
URLSession.shared.dataTask(with: URL(string: "http://www.uikonf.com")!) { d, r, e in
    // ["<html><head><title>UIKonf</title>..."]
}.resume()
```

```
for elem in seq {

}
```

```
public protocol Sequence {
    associatedtype Iterator : IteratorProtocol

    public func makeIterator() -> Self.Iterator
}
```

```
public protocol IteratorProtocol {
    associatedtype Element

    public mutating func next() -> Self.Element?
}
```

```
var z = 0
let s = AnyIterator { () -> Int in
    defer { z += 1 }
    return z
}
```

```swift
var z = 0
let s = AnyIterator { () -> Int in
    defer { z += 1 }
    return z
}

for i in s {
    print(i)
}

// prints:
// 0
// 1
// 2
// 3
// etc.
```

```swift
var z = 0
let s = AnyIterator { () -> Int in
    defer { z += 1 }
    return z
}

for i in s.prefix(3) {
    print(i)
}

// prints:
// 0
// 1
// 2
```

```
var z = 0
let s = AnyIterator { () -> Int in
    defer { z += 1 }
    return z
}

for i in s.lazy.map({ $0 % 2 == 0 }) {
    print(i)
}

// prints:
// true
// false
// true
// etc.
```

```
class AsyncSequence<E>: Sequence {

    func makeIterator() -> AnyIterator<E> {
        return AnyIterator {
            while noNextValueAvailable {
                self.semaphore.wait()
            }

            return nextValue
        }
    }
}
```

```
protocol IteratorProtocol {
    associatedtype Element

    public mutating func next() -> Self.Element?
}
```

```
protocol Observer {
    associatedtype Element

    func next(_ element: Element)
}
```

```
typealias Observer<E> = (E) -> Void
```

Iterator : Sequence

Iterator : Sequence

::

Observer : Observable

```
class Observable<E> {
    var values: [E]

    init(values: [E]) {
        self.values = values
    }

    func subscribe(_ observer: Observer<E>) {
        for v in values {
            observer(v)
        }
    }

}
```

```
let o = Observable<Int>(values: [1, 2, 3])

o.subscribe {
    print($0)
}
```

```
// prints:
// 1
// 2
// 3
```

```
class Observable<E> {
    var values: [E]


    init(values: [E]) {
        self.values = values
    }

    func subscribe(_ observer: Observer<E>) {

        for v in values {
            observer(v)
        }
    }


}
```

```
class Observable<E> {
    var values: [E]
    var observers: [Observer<E>] = []

    init(values: [E]) {
        self.values = values
    }

    func subscribe(_ observer: @escaping Observer<E>) {
        observers.append(observer)
        for v in values {
            observer(v)
        }
    }

    func append(_ newElement: E) {
        values.append(newElement)
        for o in observers {
            o(newElement)
        }
    }

}
```

```swift
let o = Observable<Int>(values: [1, 2, 3])

o.subscribe {
    print($0)
}

o.append(4)

// prints:
// 1
// 2
// 3
// 4
```

```
class Observable<E> {
    var values: [E]
    var observers: [Observer<E>] = []

    init(values: [E]) {
        self.values = values
    }

    func subscribe(_ observer: @escaping Observer<E>) {
        observers.append(observer)
        for v in values {
            observer(v)
        }
    }

    func append(_ newElement: E) {
        values.append(newElement)
        for o in observers {
            o(newElement)
        }
    }

}
```

```
class Observable<E> {

    var observers: [Observer<E>] = []



    func subscribe(_ observer: @escaping Observer<E>) {
        observers.append(observer)
    }

    func append(_ newElement: E) {

        for o in observers {
            o(newElement)
        }
    }

}
```

```
let o = Observable<String>()


o.subscribe {
    print($0)
}

o.append("Foo")

// prints:
// Foo
```

```
let o = Observable<String>()

o.append("Bar")

o.subscribe {
    print($0)
}

o.append("Foo")

// prints:
// Foo
```

```
class Observable<E> {

    var observers: [Observer<E>] = []

    func subscribe(_ observer: @escaping Observer<E>) {
        observers.append(observer)
    }

    func append(_ newElement: E) {
        for o in observers {
            o(newElement)
        }
    }
}
```

```
class Observable<E> {

    typealias SubscriptionHandler = (@escaping Observer<E>) -> Void

    let handler: SubscriptionHandler

    init(subscriptionHandler: @escaping SubscriptionHandler) {
        self.handler = subscriptionHandler
    }

    func subscribe(_ observer: @escaping Observer<E>) {
        self.handler(observer)
    }

}
```

```
let o = Observable<Int> { obs in
    obs(1)
    obs(2)
    obs(3)
    obs(4)
}
```

```
//
```

```
let o = Observable<Int> { obs in
    obs(1)
    obs(2)
    obs(3)
    obs(4)
}

o.subscribe {
    print($0)
}

// prints:
// 1
// 2
// 3
// 4
```

```
let o = Observable<Int> { obs in
    obs(1)
    obs(2)
    obs(3)
    obs(4)
}

o.subscribe { }

// --------

let s = [1, 2, 3, 4]

for elem in s { }

s.forEach { }
```

```
let o = Observable<Int> { obs in
    obs(1)
    dispatchQueue.asyncAfter(deadline: .now() + .seconds(1)) {
        obs(2)
    }
}

o.subscribe {
    print($0)
}

// prints:
// 1
// 2
```

```
func timer(delay: Int) -> Observable<Int> {
    return Observable { obs in
        DispatchQueue.main.asyncAfter(deadline: .now() + .seconds(delay)) {
            obs(delay)
        }
    }
}


//
```

```
func timer(delay: Int) -> Observable<Int> {
    return Observable { obs in
        DispatchQueue.main.asyncAfter(deadline: .now() + .seconds(delay)) {
            obs(delay)
        }
    }
}

timer(delay: 360).subscribe { _ in
    print("Your soft boiled egg is ready!")
}
```

```
func response(url: URL) -> Observable<(Data, URLResponse)> {
    return Observable { obs in

        URLSession.shared.dataTask(with: url) { d, r, e in

            guard let data = d, let response = r else { return }
            obs((data, response))

        }.resume()

    }
}
```

```swift
let uikonf = response(url: URL(string: "http://www.uikonf.com")!)

uikonf.subscribe { data, response in
    print(data)
}
```

```
extension Observable {
    func same() -> Observable<E> {
        return Observable { observer in
            self.subscribe(observer)
        }
    }
}
```

```swift
extension Observable {
    func same() -> Observable<E> {
        return Observable { observer in
            self.subscribe { elem in
                observer(elem)
            }
        }
    }
}
```

```swift
extension Observable where E == Int {
    func plusOne() -> Observable<Int> {
        return Observable { observer in
            self.subscribe { elem in
                observer(elem + 1)
            }
        }
    }
}
```

```swift
extension Observable {
    func map<U>(_ f: @escaping (E) -> U) -> Observable<U> {
        return Observable<U> { observer in
            self.subscribe { elem in
                observer(f(elem))
            }
        }
    }
}
```

```
let o = Observable<Int> { obs in
    obs(1)
    obs(2)
    obs(3)
}

let o1 = o.plusOne().map { $0 % 2 == 0 }
```

```
o1.subscribe {
    print($0)
}


//
```

```
let o = Observable<Int> { obs in
    obs(1)
    obs(2)
    obs(3)
}

let o1 = o.plusOne().map { $0 % 2 == 0 }
```

```
o1.subscribe {
    print($0)
}

o1.subscribe {
    print($0)
}
```

```swift
let uikonf = response(url: URL(string: "http://www.uikonf.com")!)

uikonf.subscribe { data, response in
    print(data)
}
```

```swift
uikonf.subscribe { data, response in
    print(data)
}
```

Reactive programming

Making duplicate networking requests has never been easier

Reactive programming

~~Making duplicate networking requests has never been easier~~

Sharing side-effects has never been easier

```
extension Observable {

    func share() -> Observable<E> {
        return Observable<E> { observer in
            // do something
        }
    }
}
```

```
extension Observable {

    func share() -> Observable<E> {

        return Observable<E> { observer in

            }

        }

}
```

```
extension Observable {

    func share() -> Observable<E> {
        var subscribed = false

        return Observable<E> { observer in

            if !subscribed {
                self.subscribe { e in
                    // ...
                }
                subscribed = true
            }
        }
    }
}
```

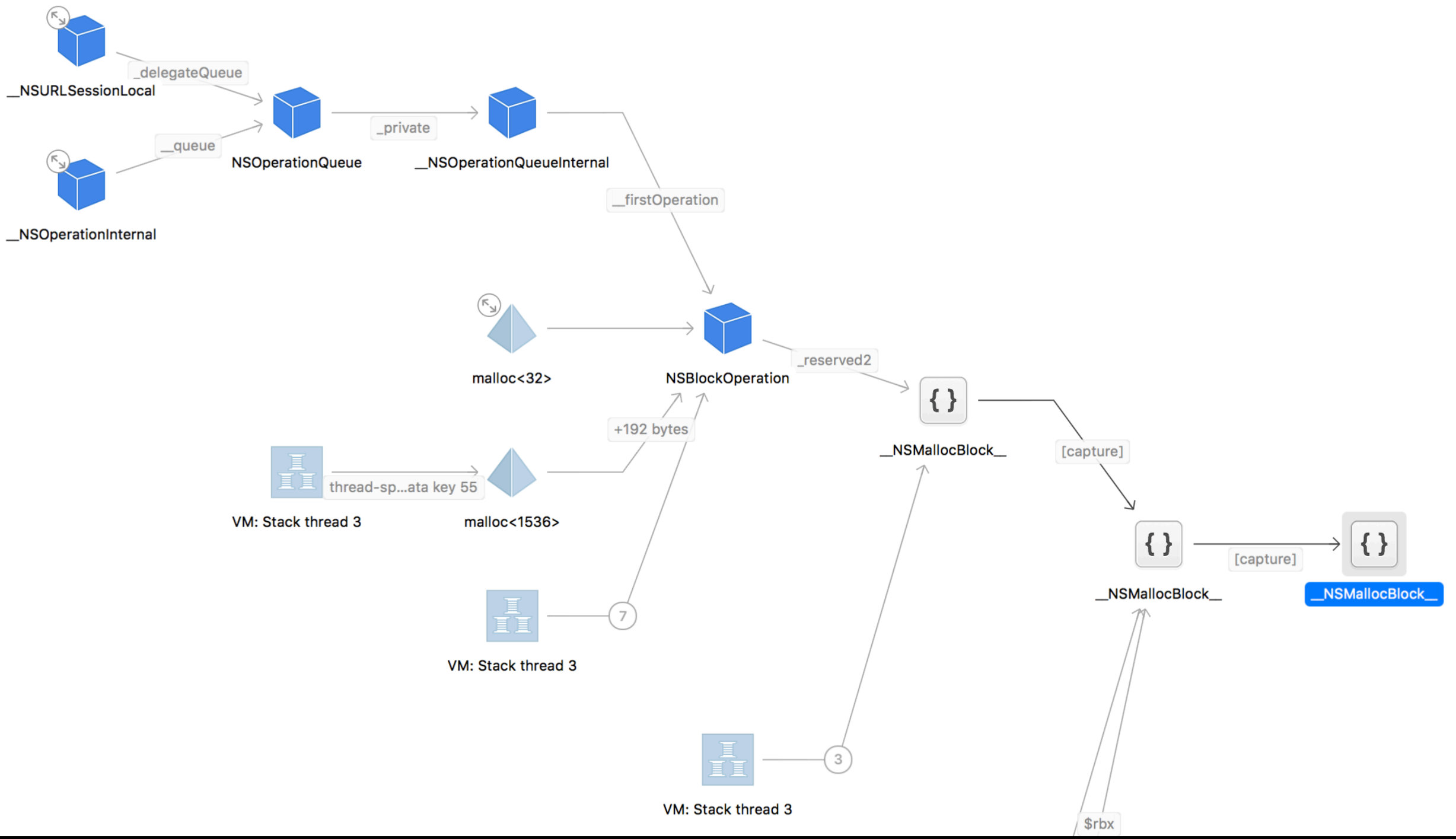
```
extension Observable {

    func share() -> Observable<E> {
        var subscribed = false
        var observers: [Observer<E>] = []
        return Observable<E> { observer in
            observers.append(observer)
            if !subscribed {
                self.subscribe { e in
                    for o in observers {
                        o(e)
                    }
                }
                subscribed = true
            }
        }
    }
}
```

```swift
let uikonf = response(url: URL(string: "http://www.uikonf.com")!).share()

uikonf.subscribe { data, response in
    print(data)
}
```


```swift
uikonf.subscribe { data, response in
    print(data)
}
```

```
for e in seq {
    break
}
```

```
let uikonf = response(url: URL(string: "http://www.uikonf.com")!).share()

uikonf.subscribe { data, response in
    print(data)
}
```


__NSURLSessionLocal
_delegateQueue
__queue
__NSOperationInternal
NSOperationQueue
_private
__NSOperationQueueInternal
__firstOperation
malloc<32>
NSBlockOperation
_reserved2
__NSMallocBlock__
[capture]
+192 bytes
thread-sp...ata key 55
VM: Stack thread 3
malloc<1536>
7
VM: Stack thread 3
__NSMallocBlock__
[capture]
__NSMallocBlock__
3
VM: Stack thread 3
$rbx

```
//

class Observable<E> {

    typealias SubscriptionHandler = (@escaping Observer<E>) -> Void

    let handler: SubscriptionHandler

    init(subscriptionHandler: @escaping SubscriptionHandler) {
        self.handler = subscriptionHandler
    }

    func subscribe(_ observer: @escaping Observer<E>) {
        self.handler(observer)
    }

}
```

```swift
typealias Disposable = () -> Void

class Observable<E> {

    typealias SubscriptionHandler = (@escaping Observer<E>) -> (Disposable)

    let handler: SubscriptionHandler

    init(subscriptionHandler: @escaping SubscriptionHandler) {
        self.handler = subscriptionHandler
    }

    func subscribe(_ observer: @escaping Observer<E>) -> Disposable {
        return self.handler(observer)
    }

}
```

```
func response(url: URL) -> Observable<(Data, URLResponse)> {
    return Observable { observer in

        let t = URLSession.shared.dataTask(with: url) { d, r, e in

            guard let data = d, let response = r else { return }
            observer((d, r))
        }
        t.resume()

        return {
            t.cancel()
        }
    }
}
```

```
let uikonf = response(url: URL(string: "http://www.uikonf.com")!)

let disposable = uikonf.subscribe { data, response in
    print(data)
}

// later...
disposable()
```

```
class View {
    var disposable: Disposable?

    func bind(to viewModel: ViewModel) {
        disposable = viewModel.title().subscribe { [weak self] in
            self.titleLabel.text = $0
        }
    }

    deinit {
        disposable?()
    }
}
```

```
class ViewController {


    func submitButtonTapped() {
        Observable<Void> { obs in
            // ?
        }
    }

}
```

```
class ViewController {

    let buttonTaps = Observable<Void>()

    func submitButtonTapped() {
        buttonTaps.send()
    }


}
```

40 slides ago

in a Kosmos not so faraway...

```
class Observable<E> {
    var observers: [Observer<E>] = []

    func append(_ newElement: E) {
        for o in observers {
            o(newElement)
        }
    }

    func subscribe(_ observer: @escaping Observer<E>) {
        observers.append(observer)
    }

}
```

```
class Subject<E>: Observable<E> {

    let observers: [Observer<E>] = []

    init() {
        super.init { observer in
            observers.append(observer)
            return {
                observers.remove(observer)
            }
        }
    }

    func send(_ newElement: E) {
        for o in observers {
            o(newElement)
        }
    }
}
```

```
class Subject<E>: Observable<E> {

    let observers: Box<[Observer<E>?]>

    init() {
        let observers = Box<[Observer<E>?]>([])
        self.observers = observers
        super.init { observer in
            let i = observers.value.count
            observers.value.append(observer)
            return {
                observers.value[i] = nil
            }
        }
    }

    func send(_ newElement: E) {
        for o in observers.value {
            o?(newElement)
        }
    }
}
```

```
class ViewController {

    let buttonTaps = Subject<Void>()

    func submitButtonTapped() {
        buttonTaps.send()
    }


}
```

```
class ViewController {

    let buttonTaps = Subject<Void>()

    func submitButtonTapped() {
        buttonTaps.send()
    }

    init() {
        let d = buttonTaps.subscribe {



        }

    }
}
```

```
class ViewController {

    let buttonTaps = Subject<Void>()

    func submitButtonTapped() {
        buttonTaps.send()
    }

    init() {
        let d = buttonTaps.subscribe {
            let url = URL(string: "http://www.uikonf.com")!
            let d1 = response(url: url).subscribe { d, r in
                print(d)
            }
        }


    }
}
```

```
class ViewController {

    let buttonTaps = Subject<Void>()

    func submitButtonTapped() {
        buttonTaps.send()
    }

    init() {
        let d = buttonTaps.map { _ in
            let url = URL(string: "http://www.uikonf.com")!
            return response(url: url)


        }.subscribe {
            // recieves Observable<(Data, URLResponse)>
        }
    }
}
```

```
extension Observable where E == ObservableProtocol {
    func flatten() -> Observable<E.Element> {
        // ...
    }
}
```

```
protocol ObservableProtocol {
    associatedtype Element

    func subscribe(_ observable: @escaping Observer<Element>)
        -> Disposable
}

extension Observable: ObservableProtocol { }
```

```swift
extension Observable where E == ObservableProtocol {
    func flatten() -> Observable<E.Element> {
        return Observable<E.Element> { observer in

            self.subscribe { innerObservable in
                innerObservable.subscribe(observer)

            }

            return {

            }
        }
    }
}
```

```swift
extension Observable where E == ObservableProtocol {
    func flatten() -> Observable<E.Element> {
        return Observable<E.Element> { observer in
            var disposables: [Disposable] = []
            let outerD = self.subscribe { innerObservable in
                let innerD = innerObservable.subscribe(observer)
                disposables.append(innerD)
            }
            disposables.append(outerD)

            return {
                for d in disposables { d() }
            }
        }
    }
}
```

```
class ViewController {

    let buttonTaps = Subject<Void>()

    func submitButtonTapped() {
        buttonTaps.send()
    }

    init() {
        let d = buttonTaps.map { _ in
            let url = URL(string: "http://www.uikonf.com")!
            return response(url: url)
        }.flatten().subscribe { data, response in
            print(data)
        }
    }
}
```

✅ Observer
✅ Observable
✅ Operators
✅ Subscribing & Disposing
✅ Sharing side-effects
✅ Subject
✅ Combining Observables

→ .next(E), .error(Error) and .completed

→ Concurrency & Thread Safety

→ Operators, operators, operators

The introduction to Reactive Programming you've been missing by @andrestaltz (https://gist.github.com/staltz/868e7e9bc2a7b8c1f754)

Reactive Programming Workshop during the Unconference Day

RxSwift, ReactiveSwift, ReactKit

Thanks!