Implementing pseudo-keywords through functional programing

Vincent Pradeilles (@v_pradeilles) – Worldline

Swift is functional

Functional Programing in Swift

Functions are first-class citizens

Functional Programing in Swift

Functions are first-class citizens

```
var increment: (Int) -> Int = { $0 + 1 }
```

Functional Programing in Swift

Functions are first-class citizens

```
var increment: (Int) -> Int = { $0 + 1 }

[1, 2, 3].map({ $0 * $0 })
```

Functional Programing in Swift

Functions are first-class citizens

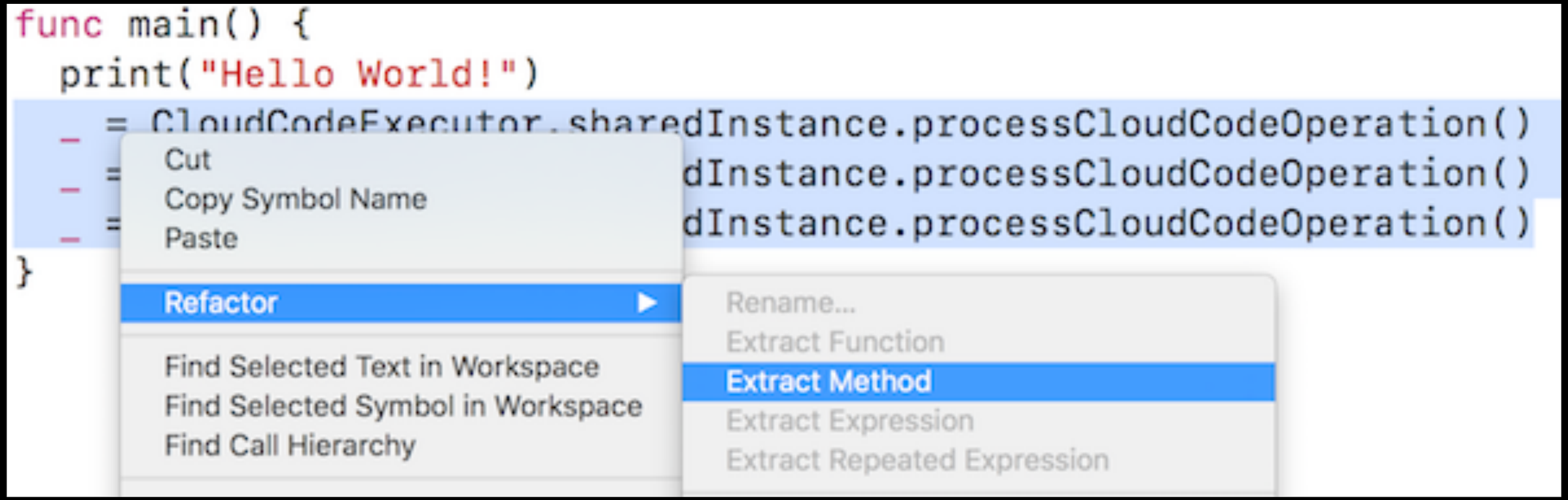
```
var increment: (Int) -> Int = { $0 + 1 }

[1, 2, 3].map({ $0 * $0 })

func buildIncrementor() -> (Int) -> Int {
    return { $0 + 1 }
}
```

Currying

```
func curry<A, B, C, D>(_ function: @escaping ((A, B, C)) -> D)
    -> (A)
    -> (B)
    -> (C)
    -> D {
    return { (a: A) -> (B) -> (C) -> D in
        return { (b: B) -> (C) -> D in
            return { (c: C) -> D in
                return function((a, b, c))
            }
        }
    }
}
```

Argo

```
extension User: Decodable {
  static func decode(_ json: JSON) -> Decoded<User> {
    return curry(User.init)
      <^> json <| "id"
      <*> json <| "name"
      <*> json <|? "email" // Use ? for parsing optional values
      <*> json <| "role" // Custom types that also conform to Decodable just work
      <*> json <| ["company", "name"] // Parse nested objects
      <*> json <|| "friends" // parse arrays of objects
  }
}
```


func main() {
print("Hello World!")
= CloudCodeExecutor.sharedInstance.processCloudCodeOperation()
dInstance.processCloudCodeOperation()
dInstance.processCloudCodeOperation()
}
Cut
Copy Symbol Name
Paste
Refactor
Find Selected Text in Workspace
Find Selected Symbol in Workspace
Find Call Hierarchy
Rename...
Extract Function
Extract Method
Extract Expression
Extract Repeated Expression

Chunks of our code can be extracted into functions...

...and functions can also take and return functions...

...so a function is able to enhance a piece of code!

Let's look at a basic example

A basic example

```
func delayed(_ action: @escaping () -> Void) -> () -> Void {
    return {
        DispatchQueue.main.asyncAfter(deadline: .now() + 5.0, execute: action)
    }
}

let helloWorld: () -> Void = delayed {
    print("Hello World!")
}

helloWorld()
```

A basic example

```
func delayed(_ action: @escaping () -> Void) -> () -> Void {
    return {
        DispatchQueue.main.asyncAfter(deadline: .now() + 5.0, execute: action)
    }
}

let helloWorld: () -> Void = delayed {
    print("Hello World!")
}

helloWorld()
```

A basic example

```
func delayed(_ action: @escaping () -> Void) -> () -> Void {
    return {
        DispatchQueue.main.asyncAfter(deadline: .now() + 5.0, execute: action)
    }
}

let helloWorld: () -> Void = delayed {
    print("Hello World!")
}

helloWorld()
```

A basic example

```
func delayed(_ action: @escaping () -> Void) -> () -> Void {
    return {
        DispatchQueue.main.asyncAfter(deadline: .now() + 5.0, execute: action)
    }
}

let helloWorld: () -> Void = delayed {
    print("Hello World!")
}

helloWorld()
```

A basic example

The trailing-closure syntax lets the function **delayed** act a whole lot like a language keyword would!

```
let helloWorld: () -> Void = delayed {
    print("Hello World!")
}
```

Now, let's look at more useful cases!

Caching

Caching

```
func cached<A: Hashable, B>(_ f: @escaping (A) -> B) -> (A) -> B {
    var cache = [A: B]()

    return { (input: A) -> B in
        if let cachedValue = cache[input] {
            return cachedValue
        } else {
            let result = f(input)
            cache[input] = result
            return result
        }
    }
}
```

Caching

```
func cached<A: Hashable, B>(_ f: @escaping (A) -> B) -> (A) -> B {
    var cache = [A: B]()

    return { (input: A) -> B in
        if let cachedValue = cache[input] {
            return cachedValue
        } else {
            let result = f(input)
            cache[input] = result
            return result
        }
    }
}
```

Caching

```
func cached<A: Hashable, B>(_ f: @escaping (A) -> B) -> (A) -> B {
    var cache = [A: B]()

    return { (input: A) -> B in
        if let cachedValue = cache[input] {
            return cachedValue
        } else {
            let result = f(input)
            cache[input] = result
            return result
        }
    }
}
```

Caching

```
func cached<A: Hashable, B>(_ f: @escaping (A) -> B) -> (A) -> B {
    var cache = [A: B]()

    return { (input: A) -> B in
        if let cachedValue = cache[input] {
            return cachedValue
        } else {
            let result = f(input)
            cache[input] = result
            return result
        }
    }
}
```

Caching

```
func cached<A: Hashable, B>(_ f: @escaping (A) -> B) -> (A) -> B {
    var cache = [A: B]()

    return { (input: A) -> B in
        if let cachedValue = cache[input] {
            return cachedValue
        } else {
            let result = f(input)
            cache[input] = result
            return result
        }
    }
}
```

Caching

```
func cached<A: Hashable, B>(_ f: @escaping (A) -> B) -> (A) -> B {
    var cache = [A: B]()

    return { (input: A) -> B in
        if let cachedValue = cache[input] {
            return cachedValue
        } else {
            let result = f(input)
            cache[input] = result
            return result
        }
    }
}
```

Caching

```
func cached<A: Hashable, B>(_ f: @escaping (A) -> B) -> (A) -> B {
    var cache = [A: B]()

    return { (input: A) -> B in
        if let cachedValue = cache[input] {
            return cachedValue
        } else {
            let result = f(input)
            cache[input] = result
            return result
        }
    }
}
```

(This **does not** work with recursive functions)

Caching

```
let cachedCos = cached { (x: Double) in cos(x) }

cachedCos(.pi * 2) // takes 48.85 ms

// value of cos for 2π is now cached

cachedCos(.pi * 2) // takes 0.13 ms
```

Debouncing

Debouncing

We want to understand how a user behaves on a scrollable screen.

To do this, we need to send the `contentOffset` to an analytics platform.

When sould we send it?

Debouncing

```
class ViewController: UIViewController, UIScrollViewDelegate {

    // ...

    func scrollViewDidScroll(_ scrollView: UIScrollView) {
        Analytics.shared.log(scrollView.contentOffset)
    }
}
```

Debouncing

Definition: waiting for a given **timespan** to elapse before performing an action.

Any new call during that timeframe **resets** the chronometer.

Debouncing

```
func debounced(delay: TimeInterval = 0.3,
               queue: DispatchQueue = .main,
               action: @escaping (() -> Void))
               -> () -> Void {
    var workItem: DispatchWorkItem?

    return {
        workItem?.cancel()
        workItem = DispatchWorkItem(block: action)
        queue.asyncAfter(deadline: .now() + delay, execute: workItem!)
    }
}
```

Debouncing

```
func debounced(delay: TimeInterval = 0.3,
               queue: DispatchQueue = .main,
               action: @escaping (() -> Void))
               -> () -> Void {
    var workItem: DispatchWorkItem?

    return {
        workItem?.cancel()
        workItem = DispatchWorkItem(block: action)
        queue.asyncAfter(deadline: .now() + delay, execute: workItem!)
    }
}
```

Debouncing

```
func debounced(delay: TimeInterval = 0.3,
               queue: DispatchQueue = .main,
               action: @escaping (() -> Void))
               -> () -> Void {
    var workItem: DispatchWorkItem?

    return {
        workItem?.cancel()
        workItem = DispatchWorkItem(block: action)
        queue.asyncAfter(deadline: .now() + delay, execute: workItem!)
    }
}
```

Debouncing

```
func debounced(delay: TimeInterval = 0.3,
               queue: DispatchQueue = .main,
               action: @escaping (() -> Void))
               -> () -> Void {
    var workItem: DispatchWorkItem?

    return {
        workItem?.cancel()
        workItem = DispatchWorkItem(block: action)
        queue.asyncAfter(deadline: .now() + delay, execute: workItem!)
    }
}
```

Debouncing

```
func debounced(delay: TimeInterval = 0.3,
               queue: DispatchQueue = .main,
               action: @escaping (() -> Void))
               -> () -> Void {
    var workItem: DispatchWorkItem?

    return {
        workItem?.cancel()
        workItem = DispatchWorkItem(block: action)
        queue.asyncAfter(deadline: .now() + delay, execute: workItem!)
    }
}
```

Debouncing

```
func debounced(delay: TimeInterval = 0.3,
               queue: DispatchQueue = .main,
               action: @escaping (() -> Void))
               -> () -> Void {
    var workItem: DispatchWorkItem?

    return {
        workItem?.cancel()
        workItem = DispatchWorkItem(block: action)
        queue.asyncAfter(deadline: .now() + delay, execute: workItem!)
    }
}
```

Debouncing

```
func debounced(delay: TimeInterval = 0.3,
               queue: DispatchQueue = .main,
               action: @escaping (() -> Void))
               -> () -> Void {
    var workItem: DispatchWorkItem?

    return {
        workItem?.cancel()
        workItem = DispatchWorkItem(block: action)
        queue.asyncAfter(deadline: .now() + delay, execute: workItem!)
    }
}
```

Debouncing

```
func debounced(delay: TimeInterval = 0.3,
               queue: DispatchQueue = .main,
               action: @escaping (() -> Void))
               -> () -> Void {
    var workItem: DispatchWorkItem?

    return {
        workItem?.cancel()
        workItem = DispatchWorkItem(block: action)
        queue.asyncAfter(deadline: .now() + delay, execute: workItem!)
    }
}
```

Debouncing

```
func debounced(delay: TimeInterval = 0.3,
               queue: DispatchQueue = .main,
               action: @escaping (() -> Void))
               -> () -> Void {
    var workItem: DispatchWorkItem?

    return {
        workItem?.cancel()
        workItem = DispatchWorkItem(block: action)
        queue.asyncAfter(deadline: .now() + delay, execute: workItem!)
    }
}
```

Debouncing

```
func debounced(delay: TimeInterval = 0.3,
               queue: DispatchQueue = .main,
               action: @escaping (() -> Void))
               -> () -> Void {
    var workItem: DispatchWorkItem?

    return {
        workItem?.cancel()
        workItem = DispatchWorkItem(block: action)
        queue.asyncAfter(deadline: .now() + delay, execute: workItem!)
    }
}
```

Debouncing

```
func debounced<T>(delay: TimeInterval = 0.3,
                  queue: DispatchQueue = .main,
                  action: @escaping ((T) -> Void))
                  -> (T) -> Void {
    var workItem: DispatchWorkItem?

    return { arg in
        workItem?.cancel()
        workItem = DispatchWorkItem(block: { action(arg) })
        queue.asyncAfter(deadline: .now() + delay, execute: workItem!)
    }
}
```

Debouncing

```
func debounced<T>(delay: TimeInterval = 0.3,
                  queue: DispatchQueue = .main,
                  action: @escaping ((T) -> Void))
                  -> (T) -> Void {
    var workItem: DispatchWorkItem?

    return { arg in
        workItem?.cancel()
        workItem = DispatchWorkItem(block: { action(arg) })
        queue.asyncAfter(deadline: .now() + delay, execute: workItem!)
    }
}
```

Debouncing

```
func debounced<T>(delay: TimeInterval = 0.3,
                  queue: DispatchQueue = .main,
                  action: @escaping ((T) -> Void))
                  -> (T) -> Void {
    var workItem: DispatchWorkItem?

    return { arg in
        workItem?.cancel()
        workItem = DispatchWorkItem(block: { action(arg) })
        queue.asyncAfter(deadline: .now() + delay, execute: workItem!)
    }
}
```

Debouncing

```
func debounced<T>(delay: TimeInterval = 0.3,
                  queue: DispatchQueue = .main,
                  action: @escaping ((T) -> Void))
                  -> (T) -> Void {
    var workItem: DispatchWorkItem?

    return { arg in
        workItem?.cancel()
        workItem = DispatchWorkItem(block: { action(arg) })
        queue.asyncAfter(deadline: .now() + delay, execute: workItem!)
    }
}
```

Debouncing

```
func debounced<T>(delay: TimeInterval = 0.3,
                  queue: DispatchQueue = .main,
                  action: @escaping ((T) -> Void))
                  -> (T) -> Void {
    var workItem: DispatchWorkItem?

    return { arg in
        workItem?.cancel()
        workItem = DispatchWorkItem(block: { action(arg) })
        queue.asyncAfter(deadline: .now() + delay, execute: workItem!)
    }
}
```

Debouncing

```
class ViewController: UIViewController, UIScrollViewDelegate {

    // ...

    let didScrollHandler = debounced { (scrollView: UIScrollView) in
        print(scrollView.contentOffset)
    }

    func scrollViewDidScroll(_ scrollView: UIScrollView) {
        self.didScrollHandler(scrollView)
    }
}
```

Debouncing

```
class ViewController: UIViewController, UIScrollViewDelegate {

    // ...

    let didScrollHandler = debounced { (scrollView: UIScrollView) in
        print(scrollView.contentOffset)
    }

    func scrollViewDidScroll(_ scrollView: UIScrollView) {
        self.didScrollHandler(scrollView)
    }
}
```

Debouncing

```
class ViewController: UIViewController, UIScrollViewDelegate {

    // ...

    let didScrollHandler = debounced { (scrollView: UIScrollView) in
        print(scrollView.contentOffset)
    }

    func scrollViewDidScroll(_ scrollView: UIScrollView) {
        self.didScrollHandler(scrollView)
    }
}
```

Debouncing

```
class ViewController: UIViewController, UIScrollViewDelegate {

    // ...

    let didScrollHandler = debounced { (scrollView: UIScrollView) in
        print(scrollView.contentOffset)
    }

    func scrollViewDidScroll(_ scrollView: UIScrollView) {
        self.didScrollHandler(scrollView)
    }
}
```

Let's keep up the good work 💪

Capturing self in closures

Capturing self in closures

```
service.call(completion: { [weak self] result in
    guard let self = self else { return }

    // use weak non-optional `self` to handle `result`
})
```

Wouldn't it be great to get rid of this extra-syntax?

Capturing self in closures

```
protocol Weakable: class { }

extension NSObject: Weakable { }

extension Weakable {

    func weakify(_ code: @escaping (Self) -> Void) -> () -> Void {
        return { [weak self] in
            guard let self = self else { return }

            code(self)
        }
    }
}
```

Capturing self in closures

```
protocol Weakable: class { }

extension NSObject: Weakable { }

extension Weakable {

    func weakify(_ code: @escaping (Self) -> Void) -> () -> Void {
        return { [weak self] in
            guard let self = self else { return }

            code(self)
        }
    }
}
```

Capturing self in closures

```
protocol Weakable: class { }

extension NSObject: Weakable { }

extension Weakable {

    func weakify(_ code: @escaping (Self) -> Void) -> () -> Void {
        return { [weak self] in
            guard let self = self else { return }

            code(self)
        }
    }
}
```

Capturing self in closures

```
protocol Weakable: class { }

extension NSObject: Weakable { }

extension Weakable {

    func weakify(_ code: @escaping (Self) -> Void) -> () -> Void {
        return { [weak self] in
            guard let self = self else { return }

            code(self)
        }
    }
}
```

Capturing self in closures

```
protocol Weakable: class { }

extension NSObject: Weakable { }

extension Weakable {

    func weakify(_ code: @escaping (Self) -> Void) -> () -> Void {
        return { [weak self] in
            guard let self = self else { return }

            code(self)
        }
    }
}
```

Capturing self in closures

```
protocol Weakable: class { }

extension NSObject: Weakable { }

extension Weakable {

    func weakify(_ code: @escaping (Self) -> Void) -> () -> Void {
        return { [weak self] in
            guard let self = self else { return }

            code(self)
        }
    }
}
```

Capturing self in closures

```
func weakify<A>(_ code: @escaping (Self, A) -> Void) -> (A) -> Void {
    return { [weak self] a in
        guard let self = self else { return }

        code(self, a)
    }
}
```

Capturing self in closures

```
service.call(completion: weakify { strongSelf, result in
    // use weak non-optional `strongSelf` to handle `result`
})
```

Asynchronous code

Asynchronous code

```
fetchUserId { id in
    fetchUserName(by: id, { firstName in
        fetchUserLastName(by: id, { lastName in
            print("Hello \(firstName) \(lastName)")
        })
    })
}
```

Aynchronous code

```
func makeSynchrone<A>(_ asyncFunction: @escaping ((A) -> Void) -> Void)
    -> () -> A {
    return {
        let lock = NSRecursiveLock()

        var result: A? = nil

        asyncFunction() {
            result = $0
            lock.unlock()
        }

        lock.lock()

        return result!
    }
}
```

Aynchronous code

What's the point?

Haven't we just lost some encapsultion?

For sure, but only when we are performing a single call.

Asynchronous code

Using the function `makeSynchrone`, we are able to extract the "asynchronicity".

How about we try to factor it out?

Asynchronous code

```
let queue = DispatchQueue(label: "asyncqueue", attributes: .concurrent)

typealias CompletionHandler<T> = (T) -> Void
```

Asynchronous code

```
func await<A>(_ body: @escaping (CompletionHandler<A>) -> Void)
    -> A {
    return makeSynchrone(body)()
}

func await<A, B>(_ body: @escaping (A, CompletionHandler<B>) -> Void)
    -> (A) -> B {
    return { a in
        makeSynchrone(body)(a)
    }
}
```

Asynchronous code

```
func await<A>(_ body: @escaping (CompletionHandler<A>) -> Void)
    -> A {
    return makeSynchrone(body)()
}

func await<A, B>(_ body: @escaping (A, CompletionHandler<B>) -> Void)
    -> (A) -> B {
    return { a in
        makeSynchrone(body)(a)
    }
}
```

Asynchronous code

```
func await<A>(_ body: @escaping (CompletionHandler<A>) -> Void)
    -> A {
    return makeSynchrone(body)()
}

func await<A, B>(_ body: @escaping (A, CompletionHandler<B>) -> Void)
    -> (A) -> B {
    return { a in
        makeSynchrone(body)(a)
    }
}
```

Asynchronous code

```
func async(_ body: @escaping () -> Void) {
    queue.async(execute: body)
}
```

Asynchronous code

```
async {
    let userId = await(fetchUserId)
    let firstName = await(fetchUserFirstName)(userId)
    let lastName = await(fetchUserLastName)(userId)

    print("Hello \(firstName) \(lastName)")
}
```

Asynchronous code

What did we do?

We extracted the asynchronicity using `await`, and we factored it out using `async`.

Asynchronous code

This technique works well with code that uses completion handlers, but it works even better with code that returns promises.

Asynchronous code

```
func fetchUser(username: String) -> Promise<User>

async {
  do {
    let user = try await(fetchUser(username: "vincent"))

    // do something with user
  }
  catch {
    print(error)
  }
}
```

Recap

Recap

Recap

→ We can build build functions that enhance or alter other functions

Recap

→ We can build build functions that enhance or alter other functions

→ This technique is great to reduce boilerplate

Recap

→ We can build build functions that enhance or alter other functions

→ This technique is great to reduce boilerplate

→ As it's backed by functions, it's highly composable

Recap

→ We can build build functions that enhance or alter other functions

→ This technique is great to reduce boilerplate

→ As it's backed by functions, it's highly composable

→ It makes our code more declarative and reusable

Recap

- → We can build build functions that enhance or alter other functions
- → This technique is great to reduce boilerplate
- → As it's backed by functions, it's highly composable
- → It makes our code more declarative and reusable
- → It encourages us to look at our code algebraically

Where to go from here?

The examples shown in these slides are only the tip of the iceberg.

There are many more use cases within your apps waiting to be discovered.

(Please do let me know when you find them!)

See you in 2019!

Some links

Some links

→ https://www.pointfree.co

Some links

- → https://www.pointfree.co
- → https://medium.com/@vin.pradeilles/an-elegant-pattern-to-craft-cache-efficient-functions-in-swift-c1a18f73e28c

Some links

- → https://www.pointfree.co
- → https://medium.com/@vin.pradeilles/an-elegant-pattern-to-craft-cache-efficient-functions-in-swift-c1a18f73e28c
- → https://github.com/vincent-pradeilles/weakable-self/

Some links

- → https://www.pointfree.co
- → https://medium.com/@vin.pradeilles/an-elegant-pattern-to-craft-cache-efficient-functions-in-swift-c1a18f73e28c
- → https://github.com/vincent-pradeilles/weakable-self/
- → https://github.com/yannickl/AwaitKit

Questions?