Pseudo-Keywords

Functional Programing at its Finest

Vincent Pradeilles (@v_pradeilles) – Worldline

I'm Vincent 👋 🇫🇷

```
lazy var tableView: UITableView = { UITableView() }()
```

var

lazy

lazy

`lazy` is a Swift keyword

lazy

`lazy` is a Swift keyword

It encapsulates a complex behaviour

lazy

`lazy` is a Swift keyword

It encapsulates a complex behaviour

Using it avoids lots of boilerplate

lazy

`lazy` is a Swift keyword

It encapsulates a complex behaviour

Using it avoids lots of boilerplate

Having it built into the language makes sense

But lazy loading is definitely not the only useful abstraction for developers!

And of course, Swift cannot implement every useful abstraction as a keyword...

So how about we try to do it ourselves? 💪

Swift ❤️ 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 })
```

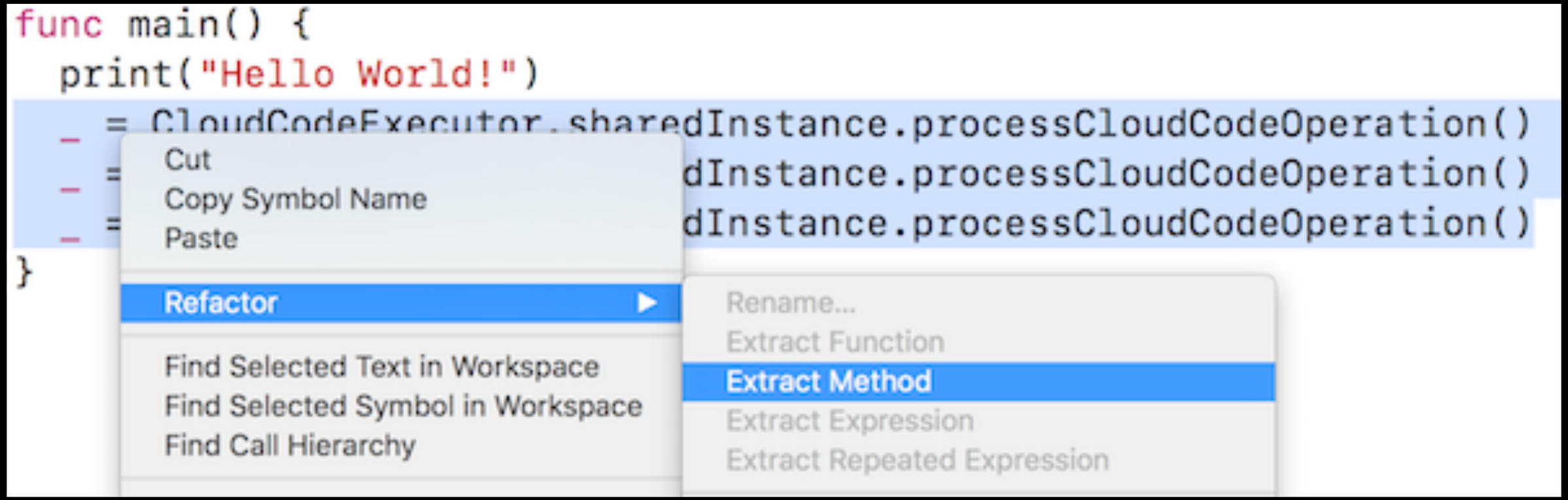
Functional Programing in Swift

Functions are first-class citizens

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

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

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


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 timer.

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
        Analytics.shared.log(scrollView.contentOffset)
    }

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

Debouncing

```
class ViewController: UIViewController, UIScrollViewDelegate {

    // ...

    let didScrollHandler = debounced { (scrollView: UIScrollView) in
        Analytics.shared.log(scrollView.contentOffset)
    }

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

Debouncing

```
class ViewController: UIViewController, UIScrollViewDelegate {

    // ...

    let didScrollHandler = debounced { (scrollView: UIScrollView) in
        Analytics.shared.log(scrollView.contentOffset)
    }

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

Debouncing

```
class ViewController: UIViewController, UIScrollViewDelegate {

    // ...

    let didScrollHandler = debounced { (scrollView: UIScrollView) in
        Analytics.shared.log(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!
    }
}
```

Asynchronous code

What's the point?

We definitely don't want to block our threads!

Asynchronous code

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

How can we leverage this?

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`.

Just like we would factor a product out of a sum.

Pretty cool, right?

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

One More Thing

Swift 5.1 

Property Wrappers

Property Wrappers

```
import SwiftUI

struct MyView: View {

    @State var name = "John"

    var body: some View {
        Text(name)
    }
}
```

Property Wrappers

```
import SwiftUI

struct MyView: View {

    @State var name = "John"

    var body: some View {
        Text(name)
    }
}
```

Property Wrappers

`@State` is a property wrapper

Property Wrappers

`@State` is a property wrapper

A property wrapper decorates a property with a custom behavior

Property Wrappers

`@State` is a property wrapper

A property wrapper decorates a property with a custom behavior

However, you can use it like any other property, it's completely transparent

Property Wrappers

It looks really close to our pseudo-keywords!

Let's try and implement `debounced` with a property wrapper 💪

Property Wrappers

```
struct Debounced {

}
```

Property Wrappers

```
struct Debounced {

    let delay: TimeInterval
    let queue: DispatchQueue
    var action: () -> Void = { }

}
```

Property Wrappers

```
struct Debounced {

    let delay: TimeInterval
    let queue: DispatchQueue
    var action: () -> Void = { }

}
```

Property Wrappers

```
struct Debounced {

    let delay: TimeInterval
    let queue: DispatchQueue
    var action: () -> Void = { }

}
```

Property Wrappers

```
struct Debounced {

    let delay: TimeInterval
    let queue: DispatchQueue
    var action: () -> Void = { }

}
```

Property Wrappers

```
struct Debounced {

    let delay: TimeInterval
    let queue: DispatchQueue
    var action: () -> Void = { }

    init(delay: TimeInterval, queue: DispatchQueue = .main) {
        self.delay = delay
        self.queue = queue
    }
}
```

Property Wrappers

```
struct Debounced {

    let delay: TimeInterval
    let queue: DispatchQueue
    var action: () -> Void = { }

    init(delay: TimeInterval, queue: DispatchQueue = .main) {
        self.delay = delay
        self.queue = queue
    }
}
```

Property Wrappers

```
@propertyWrapper
struct Debounced {

    let delay: TimeInterval
    let queue: DispatchQueue
    var action: () -> Void = { }

    init(delay: TimeInterval, queue: DispatchQueue = .main) {
        self.delay = delay
        self.queue = queue
    }
}
```

Property Wrappers

```
@propertyWrapper
struct Debounced {

    let delay: TimeInterval
    let queue: DispatchQueue
    var action: () -> Void = { }

    init(delay: TimeInterval, queue: DispatchQueue = .main) {
        self.delay = delay
        self.queue = queue
    }
}
```

Property Wrappers

```
@propertyWrapper
struct Debounced {

    let delay: TimeInterval
    let queue: DispatchQueue
    var action: () -> Void = { }

    init(delay: TimeInterval, queue: DispatchQueue = .main) {
        self.delay = delay
        self.queue = queue
    }

    var wrappedValue: () -> Void {
        get { return action }
        set {
            var workItem: DispatchWorkItem?

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

Property Wrappers

```
@propertyWrapper
struct Debounced {
    /* ... */
    var wrappedValue: () -> Void {
        get { return action }
        set {
            var workItem: DispatchWorkItem?

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

Property Wrappers

```
@propertyWrapper
struct Debounced {
    /* ... */
    var wrappedValue: () -> Void {
        get { return action }
        set {
            var workItem: DispatchWorkItem?

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

Property Wrappers

```
@propertyWrapper
struct Debounced {
    /* ... */
    var wrappedValue: () -> Void {
        get { return action }
        set {
            var workItem: DispatchWorkItem?

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

Property Wrappers

```
@propertyWrapper
struct Debounced {
    /* ... */
    var wrappedValue: () -> Void {
        get { return action }
        set {
            var workItem: DispatchWorkItem?

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

Property Wrappers

```
@propertyWrapper
struct Debounced {
    /* ... */
    var wrappedValue: () -> Void {
        get { return action }
        set {
            var workItem: DispatchWorkItem?

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

Property Wrappers

```
@propertyWrapper
struct Debounced {
    /* ... */
    var wrappedValue: () -> Void {
        get { return action }
        set {
            var workItem: DispatchWorkItem?

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

Property Wrappers

```
@propertyWrapper
struct Debounced {
    /* ... */
    var wrappedValue: () -> Void {
        get { return action }
        set {
            var workItem: DispatchWorkItem?

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

Property Wrappers

```
@propertyWrapper
struct Debounced {
    /* ... */
    var wrappedValue: () -> Void {
        get { return action }
        set {
            var workItem: DispatchWorkItem?

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

Property Wrappers

```
@propertyWrapper
struct Debounced {
    /* ... */
    var wrappedValue: () -> Void {
        get { return action }
        set {
            var workItem: DispatchWorkItem?

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

Property Wrappers

```
@propertyWrapper
struct Debounced {
    /* ... */
    var wrappedValue: () -> Void {
        get { return action }
        set {
            var workItem: DispatchWorkItem?

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

Property Wrappers

```
@propertyWrapper
struct Debounced {
    /* ... */
    var wrappedValue: () -> Void {
        get { return action }
        set {
            var workItem: DispatchWorkItem?

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

Property Wrappers

```
@propertyWrapper
struct Debounced {
    /* ... */
    var wrappedValue: () -> Void {
        get { return action }
        set {
            var workItem: DispatchWorkItem?

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

Now let's use it!

Property Wrappers

```
struct Handler {
    @Debounced(delay: 1.0) private var action: () -> Void

    init(action: @escaping () -> Void) {
        self.action = action
    }

    func handle() {
        action()
    }
}
```

Property Wrappers

```
struct Handler {
    @Debounced(delay: 1.0) private var action: () -> Void

    init(action: @escaping () -> Void) {
        self.action = action
    }

    func handle() {
        action()
    }
}
```

Property Wrappers

```
struct Handler {
    @Debounced(delay: 1.0) private var action: () -> Void

    init(action: @escaping () -> Void) {
        self.action = action
    }

    func handle() {
        action()
    }
}
```

Property Wrappers

```
struct Handler {
    @Debounced(delay: 1.0) private var action: () -> Void

    init(action: @escaping () -> Void) {
        self.action = action
    }

    func handle() {
        action()
    }
}
```

Property Wrappers

```
struct Handler {
    @Debounced(delay: 1.0) private var action: () -> Void

    init(action: @escaping () -> Void) {
        self.action = action
    }

    func handle() {
        action()
    }
}
```

Property Wrappers

```
struct Handler {
    @Debounced(delay: 1.0) private var action: () -> Void

    init(action: @escaping () -> Void) {
        self.action = action
    }

    func handle() {
        action()
    }
}
```

Property Wrappers

```
let handler = Handler(action: { print("Action performed!") })

handler.handle()
handler.handle()
handler.handle()

// After a 1 second delay, this gets
// printed to the console:

// Action performed!
```

This time it's over!

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! Twitter handle: @v_pradeilles)

Some links

Some links

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

Some links

- → 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://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?