Property Wrappers

or how Swift decided to become Java 🙃

Vincent Pradeilles (@v_pradeilles) – Worldline 🇫🇷

I'm Vincent 👋 

Property Wrappers!

But first...

```kotlin
// Kotlin

data class MyServiceResponse(/* ... */)

interface MyService {
    @FormUrlEncoded
    @POST("/myservice/endpoint")
    fun call(@Header("Authorization") authorizationHeader: String,
             @Field("first_argument") firstArgument: String,
             @Field("second_argument") secondArgument: Int
            ): Observable<MyServiceResponse>
}
```

```kotlin
// Kotlin

data class MyServiceResponse(/* ... */)

interface MyService {
    @FormUrlEncoded
    @POST("/myservice/endpoint")
    fun call(@Header("Authorization") authorizationHeader: String,
             @Field("first_argument") firstArgument: String,
             @Field("second_argument") secondArgument: Int
            ): Observable<MyServiceResponse>
}
```

```kotlin
// Kotlin

data class MyServiceResponse(/* ... */)

interface MyService {
    @FormUrlEncoded
    @POST("/myservice/endpoint")
    fun call(@Header("Authorization") authorizationHeader: String,
             @Field("first_argument") firstArgument: String,
             @Field("second_argument") secondArgument: Int
            ): Observable<MyServiceResponse>
}
```

```kotlin
// Kotlin

data class MyServiceResponse(/* ... */)

interface MyService {
    @FormUrlEncoded
    @POST("/myservice/endpoint")
    fun call(@Header("Authorization") authorizationHeader: String,
             @Field("first_argument") firstArgument: String,
             @Field("second_argument") secondArgument: Int
            ): Observable<MyServiceResponse>
}
```

```kotlin
// Kotlin

data class MyServiceResponse(/* ... */)

interface MyService {
    @FormUrlEncoded
    @POST("/myservice/endpoint")
    fun call(@Header("Authorization") authorizationHeader: String,
             @Field("first_argument") firstArgument: String,
             @Field("second_argument") secondArgument: Int
            ): Observable<MyServiceResponse>
}
```

```kotlin
// Kotlin

data class MyServiceResponse(/* ... */)

interface MyService {
    @FormUrlEncoded
    @POST("/myservice/endpoint")
    fun call(@Header("Authorization") authorizationHeader: String,
             @Field("first_argument") firstArgument: String,
             @Field("second_argument") secondArgument: Int
            ): Observable<MyServiceResponse>
}
```

```kotlin
// Kotlin

data class MyServiceResponse(/* ... */)

interface MyService {
    @FormUrlEncoded
    @POST("/myservice/endpoint")
    fun call(@Header("Authorization") authorizationHeader: String,
             @Field("first_argument") firstArgument: String,
             @Field("second_argument") secondArgument: Int
            ): Observable<MyServiceResponse>
}
```

Kotlin & Java Annotations

Through these **annotations**, Kotlin developers can **decorate** their code.

Kotlin & Java Annotations

Through these **annotations**, Kotlin developers can **decorate** their code.

These annotations carry a **meaning** that will **enrich the behavior** of the code.

Kotlin & Java Annotations

Through these **annotations**, Kotlin developers can **decorate** their code.

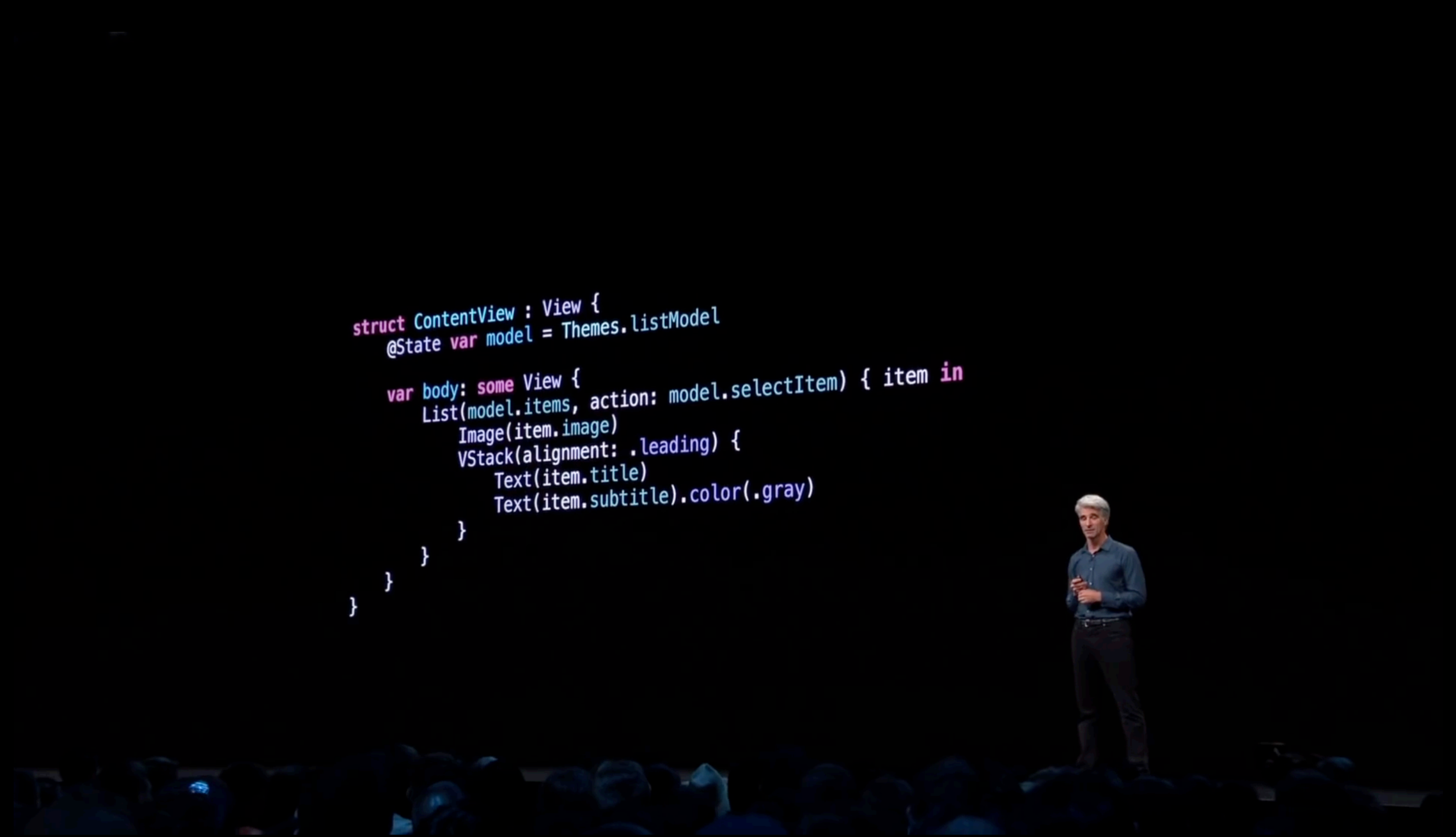
These annotations carry a **meaning** that will **enrich the behavior** of the code.

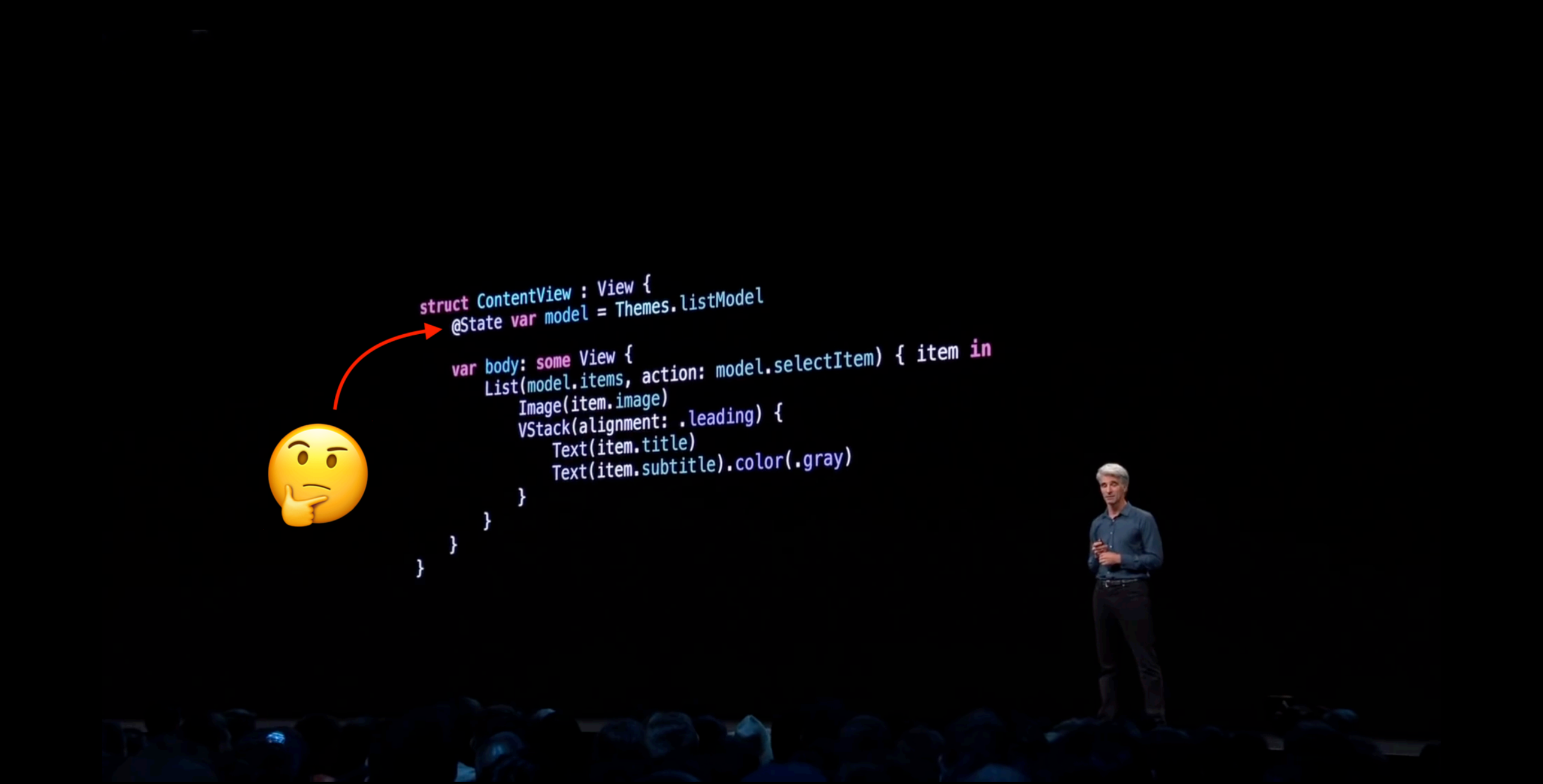
They let developers **extend** their code in a very **declarative** way.

Swift Developers 😢

WWDC 2019!


SwiftUI


struct ContentView : View {
@State var model = Themes.listModel
var body: some View {
List(model.items, action: model.selectItem) { item in
Image(item.image)
VStack(alignment: .leading) {
Text(item.title)
Text(item.subtitle).color(.gray)
}
}
}
}


struct ContentView : View {
@State var model = Themes.listModel
var body: some View {
List(model.items, action: model.selectItem) { item in
Image(item.image)
VStack(alignment: .leading) {
Text(item.title)
Text(item.subtitle).color(.gray)
}
}
}
}

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.

A property wrapper **decorates** a **property** with a **custom behavior**.

(This sounds very **similar** to **annotations** in Kotlin/Java)

Property Wrappers

To **understand** how they work under the hood, let's take a **look** at their **implementation**!

The **best place to start** is the Swift Evolution proposal (SE-0258) where they have been **initially pitched**.

SE-0258

By reading the **introduction**, we can are already get some **insight** in what **motivated** their **addition** to the language.

SE-0258

*There are **property implementation patterns** that come up **repeatedly**.*

*Rather than **hardcode a fixed set of patterns into the compiler** (as we have done for lazy and @NSCopying), we should **provide a general "property wrapper" mechanism** to allow these patterns to be **defined as libraries**.*

Let's implement some Property Wrappers 👩🏼‍💻 👨🏻‍💻

(Courtesy of https://nshipster.com/propertywrapper/)

Clamping Numerical Values

Clamping Numerical Values

```
struct Solution {
    @Clamping(0...14) var pH: Double = 7.0
}
```

Clamping Numerical Values

```
struct Clamping<Value: Comparable> {
    var value: Value
    let range: ClosedRange<Value>
}
```

Clamping Numerical Values

```
struct Clamping<Value: Comparable> {
    var value: Value
    let range: ClosedRange<Value>
}
```

Clamping Numerical Values

```
struct Clamping<Value: Comparable> {
    var value: Value
    let range: ClosedRange<Value>
}
```

Clamping Numerical Values

```
struct Clamping<Value: Comparable> {
    var value: Value
    let range: ClosedRange<Value>
}
```

Clamping Numerical Values

```
struct Clamping<Value: Comparable> {
    var value: Value
    let range: ClosedRange<Value>

    init(wrappedValue: Value, _ range: ClosedRange<Value>) {
        precondition(range.contains(wrappedValue))
        self.value = wrappedValue
        self.range = range
    }
}
```

Clamping Numerical Values

```
struct Clamping<Value: Comparable> {
    var value: Value
    let range: ClosedRange<Value>

    init(wrappedValue: Value, _ range: ClosedRange<Value>) {
        precondition(range.contains(wrappedValue))
        self.value = wrappedValue
        self.range = range
    }
}
```

Clamping Numerical Values

```
struct Clamping<Value: Comparable> {
    var value: Value
    let range: ClosedRange<Value>

    init(wrappedValue: Value, _ range: ClosedRange<Value>) {
        precondition(range.contains(wrappedValue))
        self.value = wrappedValue
        self.range = range
    }
}
```

Clamping Numerical Values

```
struct Clamping<Value: Comparable> {
    var value: Value
    let range: ClosedRange<Value>

    init(wrappedValue: Value, _ range: ClosedRange<Value>) {
        precondition(range.contains(wrappedValue))
        self.value = wrappedValue
        self.range = range
    }
}
```

Clamping Numerical Values

```
struct Clamping<Value: Comparable> {
    var value: Value
    let range: ClosedRange<Value>

    init(wrappedValue: Value, _ range: ClosedRange<Value>) {
        precondition(range.contains(wrappedValue))
        self.value = wrappedValue
        self.range = range
    }
}
```

Clamping Numerical Values

```
struct Clamping<Value: Comparable> {
    var value: Value
    let range: ClosedRange<Value>

    init(wrappedValue: Value, _ range: ClosedRange<Value>) {
        precondition(range.contains(wrappedValue))
        self.value = wrappedValue
        self.range = range
    }
}
```

Clamping Numerical Values

```
struct Clamping<Value: Comparable> {
    var value: Value
    let range: ClosedRange<Value>

    init(wrappedValue: Value, _ range: ClosedRange<Value>) {
        precondition(range.contains(wrappedValue))
        self.value = wrappedValue
        self.range = range
    }
}
```

Clamping Numerical Values

```
@propertyWrapper
struct Clamping<Value: Comparable> {
    var value: Value
    let range: ClosedRange<Value>

    init(wrappedValue: Value, _ range: ClosedRange<Value>) {
        precondition(range.contains(wrappedValue))
        self.value = wrappedValue
        self.range = range
    }
}
```

Clamping Numerical Values

```
@propertyWrapper
struct Clamping<Value: Comparable> {
    var value: Value
    let range: ClosedRange<Value>

    init(wrappedValue: Value, _ range: ClosedRange<Value>) {
        precondition(range.contains(wrappedValue))
        self.value = wrappedValue
        self.range = range
    }
}
```

Clamping Numerical Values

```
@propertyWrapper
struct Clamping<Value: Comparable> {
    var value: Value
    let range: ClosedRange<Value>

    init(wrappedValue: Value, _ range: ClosedRange<Value>) {
        precondition(range.contains(wrappedValue))
        self.value = wrappedValue
        self.range = range
    }
}
```

Clamping Numerical Values

```
@propertyWrapper
struct Clamping<Value: Comparable> {
    var value: Value
    let range: ClosedRange<Value>

    init(wrappedValue: Value, _ range: ClosedRange<Value>) {
        precondition(range.contains(wrappedValue))
        self.value = wrappedValue
        self.range = range
    }

    var wrappedValue: Value {
        get { value }
        set { value = min(max(range.lowerBound, newValue), range.upperBound) }
    }
}
```

Clamping Numerical Values

```
@propertyWrapper
struct Clamping<Value: Comparable> {
    var value: Value
    let range: ClosedRange<Value>

    init(wrappedValue: Value, _ range: ClosedRange<Value>) {
        precondition(range.contains(wrappedValue))
        self.value = wrappedValue
        self.range = range
    }

    var wrappedValue: Value {
        get { value }
        set { value = min(max(range.lowerBound, newValue), range.upperBound) }
    }
}
```

Clamping Numerical Values

```
@propertyWrapper
struct Clamping<Value: Comparable> {
    var value: Value
    let range: ClosedRange<Value>

    init(wrappedValue: Value, _ range: ClosedRange<Value>) {
        precondition(range.contains(wrappedValue))
        self.value = wrappedValue
        self.range = range
    }

    var wrappedValue: Value {
        get { value }
        set { value = min(max(range.lowerBound, newValue), range.upperBound) }
    }
}
```

Clamping Numerical Values

```
@propertyWrapper
struct Clamping<Value: Comparable> {
    var value: Value
    let range: ClosedRange<Value>

    init(wrappedValue: Value, _ range: ClosedRange<Value>) {
        precondition(range.contains(wrappedValue))
        self.value = wrappedValue
        self.range = range
    }

    var wrappedValue: Value {
        get { value }
        set { value = min(max(range.lowerBound, newValue), range.upperBound) }
    }
}
```

Clamping Numerical Values

```
@propertyWrapper
struct Clamping<Value: Comparable> {
    var value: Value
    let range: ClosedRange<Value>

    init(wrappedValue: Value, _ range: ClosedRange<Value>) {
        precondition(range.contains(wrappedValue))
        self.value = wrappedValue
        self.range = range
    }

    var wrappedValue: Value {
        get { value }
        set { value = min(max(range.lowerBound, newValue), range.upperBound) }
    }
}
```

Clamping Numerical Values

```
@propertyWrapper
struct Clamping<Value: Comparable> {
    var value: Value
    let range: ClosedRange<Value>

    init(wrappedValue: Value, _ range: ClosedRange<Value>) {
        precondition(range.contains(wrappedValue))
        self.value = wrappedValue
        self.range = range
    }

    var wrappedValue: Value {
        get { value }
        set { value = min(max(range.lowerBound, newValue), range.upperBound) }
    }
}
```

That's it 

Clamping Numerical Values

```
struct Solution {
    @Clamping(0...14) var pH: Double = 7.0
}

var solution = Solution(pH: 7.0)

solution.pH = -1
solution.pH // 0
```

Clamping Numerical Values

```
struct Solution {
    @Clamping(0...14) var pH: Double = 7.0
}

var solution = Solution(pH: 7.0)

solution.pH = -1
solution.pH // 0
```

Clamping Numerical Values

```
struct Solution {
    @Clamping(0...14) var pH: Double = 7.0
}

var solution = Solution(pH: 7.0)

solution.pH = -1
solution.pH // 0
```

Clamping Numerical Values

```
struct Solution {
    @Clamping(0...14) var pH: Double = 7.0
}

var solution = Solution(pH: 7.0)

solution.pH = -1
solution.pH // 0
```

Magic?

Compiler Magic!

Compiler Magic

Whenever we write `solution.pH`, the compiler actually **replaces** it by `solution.pH.wrappedValue`.

This is what makes Property Wrapper incredibly **seamless** to use!

And because this happens at **compile time**, they have **no minimum OS requirement** 🎉

Compiler Magic

The `initializer` also gets some compiler magic:

```
init(wrappedValue: Value, _ range: ClosedRange<Value>)

@Clamping(0...14) var pH: Double = 7.0
```

By convention, the argument `wrappedValue` will be set to the initial value of the property (here `7.0`).

Compiler Magic

If you need to **access** the **Wrapper itself**, you can do so by **implementing** the property `projectedValue`.

```
var projectedValue: Clamping<Value> {
    get { return self }
}
```

Which you would then **call** like this:

```
solution.$pH // is of type Clamping<Double>
```

Numerical Values

Now that we've implemented `@clamping`, we can easily think of other useful wrappers to deal with numerical values:

Numerical Values

Now that we've implemented `@clamping`, we can easily think of other useful wrappers to deal with numerical values:

→ `@Normalized`

Numerical Values

Now that we've implemented `@clamping`, we can easily think of other useful wrappers to deal with numerical values:

- → `@Normalized`
- → `@Rounded`

Numerical Values

Now that we've implemented `@clamping`, we can easily think of other useful wrappers to deal with numerical values:

- → `@Normalized`
- → `@Rounded`
- → `@Truncated`

Numerical Values

Now that we've implemented `@clamping`, we can easily think of other useful wrappers to deal with numerical values:

- → `@Normalized`
- → `@Rounded`
- → `@Truncated`
- → `@Quantized`

Numerical Values

Now that we've implemented `@clamping`, we can easily think of other useful wrappers to deal with numerical values:

- → `@Normalized`
- → `@Rounded`
- → `@Truncated`
- → `@Quantized`
- → etc.

Trimming Characters

Trimming Characters

When we deal with **standardized formats**, lingering **whitespaces** can be very **tricky**:

Trimming Characters

When we deal with **standardized formats**, lingering **whitespaces** can be very **tricky**:

```
URL(string: " https://nshipster.com") // nil (!)

ISO8601DateFormatter().date(from: " 2019-06-24") // nil (!)
```

Trimming Characters

When we deal with **standardized formats**, lingering **whitespaces** can be very **tricky**:

```
URL(string: " https://nshipster.com") // nil (!)

ISO8601DateFormatter().date(from: " 2019-06-24") // nil (!)
```

So let's introduce a **Property Wrapper** that will **trim** such characters!

Trimming Characters

```
@propertyWrapper
struct Trimmed {
    private(set) var value: String = ""

    var wrappedValue: String {
        get { value }
        set { value = newValue.trimmingCharacters(in: .whitespacesAndNewlines) }
    }

    init(wrappedValue: String) {
        self.wrappedValue = wrappedValue
    }
}
```

Trimming Characters

```
@propertyWrapper
struct Trimmed {
    private(set) var value: String = ""

    var wrappedValue: String {
        get { value }
        set { value = newValue.trimmingCharacters(in: .whitespacesAndNewlines) }
    }

    init(wrappedValue: String) {
        self.wrappedValue = wrappedValue
    }
}
```

Trimming Characters

```
@propertyWrapper
struct Trimmed {
    private(set) var value: String = ""

    var wrappedValue: String {
        get { value }
        set { value = newValue.trimmingCharacters(in: .whitespacesAndNewlines) }
    }

    init(wrappedValue: String) {
        self.wrappedValue = wrappedValue
    }
}
```

Trimming Characters

```
@propertyWrapper
struct Trimmed {
    private(set) var value: String = ""

    var wrappedValue: String {
        get { value }
        set { value = newValue.trimmingCharacters(in: .whitespacesAndNewlines) }
    }

    init(wrappedValue: String) {
        self.wrappedValue = wrappedValue
    }
}
```

Trimming Characters

```
@propertyWrapper
struct Trimmed {
    private(set) var value: String = ""

    var wrappedValue: String {
        get { value }
        set { value = newValue.trimmingCharacters(in: .whitespacesAndNewlines) }
    }

    init(wrappedValue: String) {
        self.wrappedValue = wrappedValue
    }
}
```

Trimming Characters

```
@propertyWrapper
struct Trimmed {
    private(set) var value: String = ""

    var wrappedValue: String {
        get { value }
        set { value = newValue.trimmingCharacters(in: .whitespacesAndNewlines) }
    }

    init(wrappedValue: String) {
        self.wrappedValue = wrappedValue
    }
}
```

Trimming Characters

```
@propertyWrapper
struct Trimmed {
    private(set) var value: String = ""

    var wrappedValue: String {
        get { value }
        set { value = newValue.trimmingCharacters(in: .whitespacesAndNewlines) }
    }

    init(wrappedValue: String) {
        self.wrappedValue = wrappedValue
    }
}
```

Trimming Characters

```
@propertyWrapper
struct Trimmed {
    private(set) var value: String = ""

    var wrappedValue: String {
        get { value }
        set { value = newValue.trimmingCharacters(in: .whitespacesAndNewlines) }
    }

    init(wrappedValue: String) {
        self.wrappedValue = wrappedValue
    }
}
```

Trimming Characters

```
@propertyWrapper
struct Trimmed {
    private(set) var value: String = ""

    var wrappedValue: String {
        get { value }
        set { value = newValue.trimmingCharacters(in: .whitespacesAndNewlines) }
    }

    init(wrappedValue: String) {
        self.wrappedValue = wrappedValue
    }
}
```

Trimming Characters

```
struct Post {
    @Trimmed var title: String
    @Trimmed var body: String
}

let quine = Post(title: "  Swift Property Wrappers  ", body: "...")
quine.title // "Swift Property Wrappers" (no leading or trailing spaces!)

quine.title = "      @propertyWrapper      "
quine.title // "@propertyWrapper" (still no leading or trailing spaces!)
```

Trimming Characters

```
struct Post {
    @Trimmed var title: String
    @Trimmed var body: String
}

let quine = Post(title: "  Swift Property Wrappers  ", body: "...")
quine.title // "Swift Property Wrappers" (no leading or trailing spaces!)

quine.title = "      @propertyWrapper      "
quine.title // "@propertyWrapper" (still no leading or trailing spaces!)
```

Trimming Characters

```
struct Post {
    @Trimmed var title: String
    @Trimmed var body: String
}

let quine = Post(title: "  Swift Property Wrappers  ", body: "...")
quine.title // "Swift Property Wrappers" (no leading or trailing spaces!)

quine.title = "      @propertyWrapper      "
quine.title // "@propertyWrapper" (still no leading or trailing spaces!)
```

Trimming Characters

```
struct Post {
    @Trimmed var title: String
    @Trimmed var body: String
}

let quine = Post(title: "  Swift Property Wrappers  ", body: "...")
quine.title // "Swift Property Wrappers" (no leading or trailing spaces!)

quine.title = "      @propertyWrapper      "
quine.title // "@propertyWrapper" (still no leading or trailing spaces!)
```

Trimming Characters

```
struct Post {
    @Trimmed var title: String
    @Trimmed var body: String
}

let quine = Post(title: "  Swift Property Wrappers  ", body: "...")
quine.title // "Swift Property Wrappers" (no leading or trailing spaces!)

quine.title = "      @propertyWrapper      "
quine.title // "@propertyWrapper" (still no leading or trailing spaces!)
```

Trimming Characters

```
struct Post {
    @Trimmed var title: String
    @Trimmed var body: String
}

let quine = Post(title: "  Swift Property Wrappers  ", body: "...")
quine.title // "Swift Property Wrappers" (no leading or trailing spaces!)

quine.title = "      @propertyWrapper      "
quine.title // "@propertyWrapper" (still no leading or trailing spaces!)
```

Trimming Characters

```
struct Post {
    @Trimmed var title: String
    @Trimmed var body: String
}

let quine = Post(title: "  Swift Property Wrappers  ", body: "...")
quine.title // "Swift Property Wrappers" (no leading or trailing spaces!)

quine.title = "      @propertyWrapper      "
quine.title // "@propertyWrapper" (still no leading or trailing spaces!)
```

Data Versioning

Data Versioning

We can even go further, and **implement** an entire **business requirement** through a Property Wrapper.

Let's implement a Property Wrapper that will manage an **history** of the **values assigned** to a given **variable**.

(Developers working on **server-side** might find it particularly useful!)

Data Versioning

```
@propertyWrapper
struct Versioned<Value> {
    private var value: Value
    private(set) var timestampedValues: [(Date, Value)] = []

    var wrappedValue: Value {
        get { value }

        set {
            value = newValue
            timestampedValues.append((Date(), value))
        }
    }

    init(wrappedValue: Value) {
        self.wrappedValue = wrappedValue
    }
}
```

Data Versioning

```
@propertyWrapper
struct Versioned<Value> {
    private var value: Value
    private(set) var timestampedValues: [(Date, Value)] = []

    var wrappedValue: Value {
        get { value }

        set {
            value = newValue
            timestampedValues.append((Date(), value))
        }
    }

    init(wrappedValue: Value) {
        self.wrappedValue = wrappedValue
    }
}
```

Data Versioning

```
@propertyWrapper
struct Versioned<Value> {
    private var value: Value
    private(set) var timestampedValues: [(Date, Value)] = []

    var wrappedValue: Value {
        get { value }

        set {
            value = newValue
            timestampedValues.append((Date(), value))
        }
    }

    init(wrappedValue: Value) {
        self.wrappedValue = wrappedValue
    }
}
```

Data Versioning

```
@propertyWrapper
struct Versioned<Value> {
    private var value: Value
    private(set) var timestampedValues: [(Date, Value)] = []

    var wrappedValue: Value {
        get { value }

        set {
            value = newValue
            timestampedValues.append((Date(), value))
        }
    }

    init(wrappedValue: Value) {
        self.wrappedValue = wrappedValue
    }
}
```

Data Versioning

```
@propertyWrapper
struct Versioned<Value> {
    private var value: Value
    private(set) var timestampedValues: [(Date, Value)] = []

    var wrappedValue: Value {
        get { value }

        set {
            value = newValue
            timestampedValues.append((Date(), value))
        }
    }

    init(wrappedValue: Value) {
        self.wrappedValue = wrappedValue
    }
}
```

Data Versioning

```
@propertyWrapper
struct Versioned<Value> {
    private var value: Value
    private(set) var timestampedValues: [(Date, Value)] = []

    var wrappedValue: Value {
        get { value }

        set {
            value = newValue
            timestampedValues.append((Date(), value))
        }
    }

    init(wrappedValue: Value) {
        self.wrappedValue = wrappedValue
    }
}
```

Data Versioning

```
@propertyWrapper
struct Versioned<Value> {
    private var value: Value
    private(set) var timestampedValues: [(Date, Value)] = []

    var wrappedValue: Value {
        get { value }

        set {
            value = newValue
            timestampedValues.append((Date(), value))
        }
    }

    init(wrappedValue: Value) {
        self.wrappedValue = wrappedValue
    }
}
```

Data Versioning

```
@propertyWrapper
struct Versioned<Value> {
    private var value: Value
    private(set) var timestampedValues: [(Date, Value)] = []

    var wrappedValue: Value {
        get { value }

        set {
            value = newValue
            timestampedValues.append((Date(), value))
        }
    }

    init(wrappedValue: Value) {
        self.wrappedValue = wrappedValue
    }
}
```

Data Versioning

```
@propertyWrapper
struct Versioned<Value> {
    private var value: Value
    private(set) var timestampedValues: [(Date, Value)] = []

    var wrappedValue: Value {
        get { value }

        set {
            value = newValue
            timestampedValues.append((Date(), value))
        }
    }

    init(wrappedValue: Value) {
        self.wrappedValue = wrappedValue
    }
}
```

Data Versioning

```
@propertyWrapper
struct Versioned<Value> {
    private var value: Value
    private(set) var timestampedValues: [(Date, Value)] = []

    var wrappedValue: Value {
        get { value }

        set {
            value = newValue
            timestampedValues.append((Date(), value))
        }
    }

    init(wrappedValue: Value) {
        self.wrappedValue = wrappedValue
    }
}
```

Data Versioning

```
@propertyWrapper
struct Versioned<Value> {
    private var value: Value
    private(set) var timestampedValues: [(Date, Value)] = []

    var wrappedValue: Value {
        get { value }

        set {
            value = newValue
            timestampedValues.append((Date(), value))
        }
    }

    init(wrappedValue: Value) {
        self.wrappedValue = wrappedValue
    }
}
```

Data Versioning

```
class ExpenseReport {
    enum State { case submitted, received, approved, denied }

    @Versioned var state: State = .submitted
}
```

Data Versioning

A real world back-end application could go even further.

We could implement another wrapper `@Audited`, that would **track which user read or wrote** a given variable.

Data Versioning

However there are still some **limitations**: Property Wrapper **cannot throw errors**, which makes some interesting use cases **out of reach**.

Data Versioning

However there are still some **limitations**: Property Wrapper **cannot throw errors**, which makes some interesting use cases **out of reach**.

If it **were possible**, we could write some **pretty cool things**, like extending `@Versioned` to **implement** a **workflow mechanism** 🤩

We've used Property Wrappers to implement standalone features...

...Now, let's try to interact with existing APIs

Let's look at UserDefaults 🧐

(Example from https://www.avanderlee.com/swift/property-wrappers/)

UserDefaults

```
extension UserDefaults {

    public enum Keys {
        static let hasSeenAppIntroduction = "has_seen_app_introduction"
    }

    /// Indicates whether or not the user has seen the on-boarding.
    var hasSeenAppIntroduction: Bool {
        set {
            set(newValue, forKey: Keys.hasSeenAppIntroduction)
        }
        get {
            return bool(forKey: Keys.hasSeenAppIntroduction)
        }
    }
}
```

UserDefaults

```
UserDefaults.standard.hasSeenAppIntroduction = true

guard !UserDefaults.standard.hasSeenAppIntroduction else { return }
showAppIntroduction()
```

UserDefaults

Very clean call sites 👍

Requires boilerplate 👎

Perfect Use Case for a Property Wrapper

```swift
@propertyWrapper
struct UserDefault<T> {
    let key: String
    let defaultValue: T

    init(_ key: String, defaultValue: T) {
        self.key = key
        self.defaultValue = defaultValue
    }

    var wrappedValue: T {
        get {
            return UserDefaults.standard.object(forKey: key) as? T ?? defaultValue
        }
        set {
            UserDefaults.standard.set(newValue, forKey: key)
        }
    }
}
```

```swift
@propertyWrapper
struct UserDefault<T> {
    let key: String
    let defaultValue: T

    init(_ key: String, defaultValue: T) {
        self.key = key
        self.defaultValue = defaultValue
    }

    var wrappedValue: T {
        get {
            return UserDefaults.standard.object(forKey: key) as? T ?? defaultValue
        }
        set {
            UserDefaults.standard.set(newValue, forKey: key)
        }
    }
}
```

```swift
@propertyWrapper
struct UserDefault<T> {
    let key: String
    let defaultValue: T

    init(_ key: String, defaultValue: T) {
        self.key = key
        self.defaultValue = defaultValue
    }

    var wrappedValue: T {
        get {
            return UserDefaults.standard.object(forKey: key) as? T ?? defaultValue
        }
        set {
            UserDefaults.standard.set(newValue, forKey: key)
        }
    }
}
```

```swift
@propertyWrapper
struct UserDefault<T> {
    let key: String
    let defaultValue: T

    init(_ key: String, defaultValue: T) {
        self.key = key
        self.defaultValue = defaultValue
    }

    var wrappedValue: T {
        get {
            return UserDefaults.standard.object(forKey: key) as? T ?? defaultValue
        }
        set {
            UserDefaults.standard.set(newValue, forKey: key)
        }
    }
}
```

```swift
@propertyWrapper
struct UserDefault<T> {
    let key: String
    let defaultValue: T

    init(_ key: String, defaultValue: T) {
        self.key = key
        self.defaultValue = defaultValue
    }

    var wrappedValue: T {
        get {
            return UserDefaults.standard.object(forKey: key) as? T ?? defaultValue
        }
        set {
            UserDefaults.standard.set(newValue, forKey: key)
        }
    }
}
```

UserDefaults

```
struct UserDefaultsConfig {
    @UserDefault("has_seen_app_introduction", defaultValue: false)
    static var hasSeenAppIntroduction: Bool
}

UserDefaultsConfig.hasSeenAppIntroduction = false
print(UserDefaultsConfig.hasSeenAppIntroduction) // Prints: false
UserDefaultsConfig.hasSeenAppIntroduction = true
print(UserDefaultsConfig.hasSeenAppIntroduction) // Prints: true
```

UserDefaults

```
struct UserDefaultsConfig {
    @UserDefault("has_seen_app_introduction", defaultValue: false)
    static var hasSeenAppIntroduction: Bool
}

UserDefaultsConfig.hasSeenAppIntroduction = false
print(UserDefaultsConfig.hasSeenAppIntroduction) // Prints: false
UserDefaultsConfig.hasSeenAppIntroduction = true
print(UserDefaultsConfig.hasSeenAppIntroduction) // Prints: true
```

UserDefaults

```
struct UserDefaultsConfig {
    @UserDefault("has_seen_app_introduction", defaultValue: false)
    static var hasSeenAppIntroduction: Bool
}

UserDefaultsConfig.hasSeenAppIntroduction = false
print(UserDefaultsConfig.hasSeenAppIntroduction) // Prints: false
UserDefaultsConfig.hasSeenAppIntroduction = true
print(UserDefaultsConfig.hasSeenAppIntroduction) // Prints: true
```

UserDefaults

```
struct UserDefaultsConfig {
    @UserDefault("has_seen_app_introduction", defaultValue: false)
    static var hasSeenAppIntroduction: Bool
}

UserDefaultsConfig.hasSeenAppIntroduction = false
print(UserDefaultsConfig.hasSeenAppIntroduction) // Prints: false
UserDefaultsConfig.hasSeenAppIntroduction = true
print(UserDefaultsConfig.hasSeenAppIntroduction) // Prints: true
```

UserDefaults

```
struct UserDefaultsConfig {
    @UserDefault("has_seen_app_introduction", defaultValue: false)
    static var hasSeenAppIntroduction: Bool
}

UserDefaultsConfig.hasSeenAppIntroduction = false
print(UserDefaultsConfig.hasSeenAppIntroduction) // Prints: false
UserDefaultsConfig.hasSeenAppIntroduction = true
print(UserDefaultsConfig.hasSeenAppIntroduction) // Prints: true
```

UserDefaults & More

This example focused on **smoothing out interactions** with `UserDefaults`.

UserDefaults & More

This example focused on **smoothing out interactions** with `UserDefaults`.

But it's an **approach** that could be **applied** to **any data store**.

Property Wrappers operate on properties

Thank you,
Captain Obvious 

Wouldn't it be cool to have the same mechanism for functions?

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 }
}
```

We can store a function in a property...

...so Property Wrappers can work with functions 🎉

Let's start with a simple use case

Caching Pure Functions

Caching Pure Functions

```
@propertyWrapper
struct Cached<Input: Hashable, Output> {

    private var functionWithCache: (Input) -> Output

    init(wrappedValue: @escaping (Input) -> Output) {
        self.functionWithCache = Cached.addCachingLogic(to: wrappedValue)
    }

    var wrappedValue: (Input) -> Output {
        get { return self.functionWithCache }
        set { self.functionWithCache = Cached.addCachingLogic(to: newValue) }
    }

    private static func addCachingLogic(to function: @escaping (Input) -> Output) -> (Input) -> Output {
        var cache: [Input: Output] = [:]

        return { input in
            if let cachedOutput = cache[input] {
                return cachedOutput
            } else {
                let output = function(input)
                cache[input] = output
                return output
            }
        }
    }
}
```

Caching Pure Functions

```
@propertyWrapper
struct Cached<Input: Hashable, Output> {

    private var functionWithCache: (Input) -> Output

    init(wrappedValue: @escaping (Input) -> Output) {
        self.functionWithCache = Cached.addCachingLogic(to: wrappedValue)
    }

    var wrappedValue: (Input) -> Output {
        /* ... */
    }

    private static func addCachingLogic(to function: @escaping (Input) -> Output) -> (Input) -> Output {
        /* ... */
    }
}
```

Caching Pure Functions

```
@propertyWrapper
struct Cached<Input: Hashable, Output> {

    private var functionWithCache: (Input) -> Output

    init(wrappedValue: @escaping (Input) -> Output) {
        self.functionWithCache = Cached.addCachingLogic(to: wrappedValue)
    }

    var wrappedValue: (Input) -> Output {
        /* ... */
    }

    private static func addCachingLogic(to function: @escaping (Input) -> Output) -> (Input) -> Output {
        /* ... */
    }
}
```

Caching Pure Functions

```
@propertyWrapper
struct Cached<Input: Hashable, Output> {

    private var functionWithCache: (Input) -> Output

    init(wrappedValue: @escaping (Input) -> Output) {
        self.functionWithCache = Cached.addCachingLogic(to: wrappedValue)
    }

    var wrappedValue: (Input) -> Output {
        /* ... */
    }

    private static func addCachingLogic(to function: @escaping (Input) -> Output) -> (Input) -> Output {
        /* ... */
    }
}
```

Caching Pure Functions

```
@propertyWrapper
struct Cached<Input: Hashable, Output> {

    private var functionWithCache: (Input) -> Output

    init(wrappedValue: @escaping (Input) -> Output) {
        self.functionWithCache = Cached.addCachingLogic(to: wrappedValue)
    }

    var wrappedValue: (Input) -> Output {
        /* ... */
    }

    private static func addCachingLogic(to function: @escaping (Input) -> Output) -> (Input) -> Output {
        /* ... */
    }
}
```

Caching Pure Functions

```
@propertyWrapper
struct Cached<Input: Hashable, Output> {

    private var functionWithCache: (Input) -> Output

    init(wrappedValue: @escaping (Input) -> Output) {
        self.functionWithCache = Cached.addCachingLogic(to: wrappedValue)
    }

    var wrappedValue: (Input) -> Output {
        /* ... */
    }

    private static func addCachingLogic(to function: @escaping (Input) -> Output) -> (Input) -> Output {
        /* ... */
    }
}
```

Caching Pure Functions

```
@propertyWrapper
struct Cached<Input: Hashable, Output> {

    /* ... */

    var wrappedValue: (Input) -> Output {
        get { return self.functionWithCache }
        set { self.functionWithCache = Cached.addCachingLogic(to: newValue) }
    }

    private static func addCachingLogic(to function: @escaping (Input) -> Output) -> (Input) -> Output {
        /* ... */
    }
}
```

Caching Pure Functions

```
@propertyWrapper
struct Cached<Input: Hashable, Output> {

    /* ... */

    var wrappedValue: (Input) -> Output {
        get { return self.functionWithCache }
        set { self.functionWithCache = Cached.addCachingLogic(to: newValue) }
    }

    private static func addCachingLogic(to function: @escaping (Input) -> Output) -> (Input) -> Output {
        /* ... */
    }
}
```

Caching Pure Functions

```
@propertyWrapper
struct Cached<Input: Hashable, Output> {

    /* ... */

    var wrappedValue: (Input) -> Output {
        get { return self.functionWithCache }
        set { self.functionWithCache = Cached.addCachingLogic(to: newValue) }
    }

    private static func addCachingLogic(to function: @escaping (Input) -> Output) -> (Input) -> Output {
        /* ... */
    }
}
```

Caching Pure Functions

```
@propertyWrapper
struct Cached<Input: Hashable, Output> {

    /* ... */

    var wrappedValue: (Input) -> Output {
        get { return self.functionWithCache }
        set { self.functionWithCache = Cached.addCachingLogic(to: newValue) }
    }

    private static func addCachingLogic(to function: @escaping (Input) -> Output) -> (Input) -> Output {
        /* ... */
    }
}
```

Caching Pure Functions

```
@propertyWrapper
struct Cached<Input: Hashable, Output> {

    /* ... */

    var wrappedValue: (Input) -> Output {
        get { return self.functionWithCache }
        set { self.functionWithCache = Cached.addCachingLogic(to: newValue) }
    }

    private static func addCachingLogic(to function: @escaping (Input) -> Output) -> (Input) -> Output {
        /* ... */
    }
}
```

```swift
@propertyWrapper
struct Cached<Input: Hashable, Output> {

    /* ... */

    var wrappedValue: (Input) -> Output {
        /* ... */
    }

    private static func addCachingLogic(to function: @escaping (Input) -> Output) -> (Input) -> Output {
        var cache: [Input: Output] = [:]

        return { input in
            if let cachedOutput = cache[input] {
                return cachedOutput
            } else {
                let output = function(input)
                cache[input] = output
                return output
            }
        }
    }
}
```

```swift
@propertyWrapper
struct Cached<Input: Hashable, Output> {

    /* ... */

    var wrappedValue: (Input) -> Output {
        /* ... */
    }

    private static func addCachingLogic(to function: @escaping (Input) -> Output) -> (Input) -> Output {
        var cache: [Input: Output] = [:]

        return { input in
            if let cachedOutput = cache[input] {
                return cachedOutput
            } else {
                let output = function(input)
                cache[input] = output
                return output
            }
        }
    }
}
```

```swift
@propertyWrapper
struct Cached<Input: Hashable, Output> {

    /* ... */

    var wrappedValue: (Input) -> Output {
        /* ... */
    }

    private static func addCachingLogic(to function: @escaping (Input) -> Output) -> (Input) -> Output {
        var cache: [Input: Output] = [:]

        return { input in
            if let cachedOutput = cache[input] {
                return cachedOutput
            } else {
                let output = function(input)
                cache[input] = output
                return output
            }
        }
    }
}
```

```swift
@propertyWrapper
struct Cached<Input: Hashable, Output> {

    /* ... */

    var wrappedValue: (Input) -> Output {
        /* ... */
    }

    private static func addCachingLogic(to function: @escaping (Input) -> Output) -> (Input) -> Output {
        var cache: [Input: Output] = [:]

        return { input in
            if let cachedOutput = cache[input] {
                return cachedOutput
            } else {
                let output = function(input)
                cache[input] = output
                return output
            }
        }
    }
}
```

```swift
@propertyWrapper
struct Cached<Input: Hashable, Output> {

    /* ... */

    var wrappedValue: (Input) -> Output {
        /* ... */
    }

    private static func addCachingLogic(to function: @escaping (Input) -> Output) -> (Input) -> Output {
        var cache: [Input: Output] = [:]

        return { input in
            if let cachedOutput = cache[input] {
                return cachedOutput
            } else {
                let output = function(input)
                cache[input] = output
                return output
            }
        }
    }
}
```

Caching Pure Functions

```
struct Trigo {
    @Cached static var cachedCos = { (x: Double) in cos(x) }
}

Trigo.cachedCos(.pi * 2) // takes 48.85 µs

// value of cos for 2π is now cached

Trigo.cachedCos(.pi * 2) // takes 0.13 µs
```

Caching Pure Functions

```
struct Trigo {
    @Cached static var cachedCos = { (x: Double) in cos(x) }
}

Trigo.cachedCos(.pi * 2) // takes 48.85 µs

// value of cos for 2π is now cached

Trigo.cachedCos(.pi * 2) // takes 0.13 µs
```

Caching Pure Functions

```
struct Trigo {
    @Cached static var cachedCos = { (x: Double) in cos(x) }
}

Trigo.cachedCos(.pi * 2) // takes 48.85 µs

// value of cos for 2π is now cached

Trigo.cachedCos(.pi * 2) // takes 0.13 µs
```

Caching Pure Functions

```
struct Trigo {
    @Cached static var cachedCos = { (x: Double) in cos(x) }
}

Trigo.cachedCos(.pi * 2) // takes 48.85 µs

// value of cos for 2π is now cached

Trigo.cachedCos(.pi * 2) // takes 0.13 µs
```

Property Wrappers & Functions

`@Cached` is a very good example to **grasp** how **Property Wrappers and functions** can **work together**.

But we could devise **many others**:

Property Wrappers & Functions

`@Cached` is a very good example to **grasp** how **Property Wrappers and functions** can **work together**.

But we could devise **many others**:

→ `@Delayed(delay: 0.3)` and `@Debounced(delay: 0.3)`, to deal with timing

Property Wrappers & Functions

`@Cached` is a very good example to **grasp** how **Property Wrappers and functions** can **work together**.

But we could devise **many others**:

→ `@Delayed(delay: 0.3)` and `@Debounced(delay: 0.3)`, to deal with timing

→ `@ThreadSafe`, to wrap a piece of code around a `Lock`

Property Wrappers & Functions

`@Cached` is a very good example to **grasp** how **Property Wrappers and functions** can **work together**.

But we could devise **many others**:

→ `@Delayed(delay: 0.3)` and `@Debounced(delay: 0.3)`, to deal with timing

→ `@ThreadSafe`, to wrap a piece of code around a `Lock`

→ etc.

Last one 

```kotlin
// Kotlin

data class MyServiceResponse(/* ... */)

interface MyService {
    @FormUrlEncoded
    @POST("/myservice/endpoint")
    fun call(@Header("Authorization") authorizationHeader: String,
             @Field("first_argument") firstArgument: String,
             @Field("second_argument") secondArgument: Int
            ): Observable<MyServiceResponse>
}
```

```kotlin
// Kotlin

data class MyServiceResponse(/* ... */)

interface MyService {
    @FormUrlEncoded
    @POST("/myservice/endpoint")
    fun call(@Header("Authorization") authorizationHeader: String,
             @Field("first_argument") firstArgument: String,
             @Field("second_argument") secondArgument: Int
            ): Observable<MyServiceResponse>
}
```

Property Wrappers can't be composed (yet)...

...so we're going to implement GET instead

```swift
typealias Service<Response> = (_ completionHandler: @escaping (Result<Response, Error>) -> Void) -> ()

@propertyWrapper
struct GET {
    private var url: URL

    init(url: String) {
        self.url = URL(string: url)!
    }

    var wrappedValue: Service<String> {
        get {
            return { completionHandler in
                let task = URLSession.shared.dataTask(with: self.url) { (data, response, error) in
                    guard error == nil else { completionHandler(.failure(error!)); return }

                    let string = String(data: data!, encoding: .utf8)!

                    completionHandler(.success(string))
                }

                task.resume()
            }
        }
    }
}
```

```swift
typealias Service<Response> = (_ completionHandler: @escaping (Result<Response, Error>) -> Void) -> ()

@propertyWrapper
struct GET {
    private var url: URL

    init(url: String) {
        self.url = URL(string: url)!
    }

    var wrappedValue: Service<String> {
        get {
            return { completionHandler in
                let task = URLSession.shared.dataTask(with: self.url) { (data, response, error) in
                    guard error == nil else { completionHandler(.failure(error!)); return }

                    let string = String(data: data!, encoding: .utf8)!

                    completionHandler(.success(string))
                }

                task.resume()
            }
        }
    }
}
```

```swift
typealias Service<Response> = (_ completionHandler: @escaping (Result<Response, Error>) -> Void) -> ()

@propertyWrapper
struct GET {
    private var url: URL

    init(url: String) {
        self.url = URL(string: url)!
    }

    var wrappedValue: Service<String> {
        get {
            return { completionHandler in
                let task = URLSession.shared.dataTask(with: self.url) { (data, response, error) in
                    guard error == nil else { completionHandler(.failure(error!)); return }

                    let string = String(data: data!, encoding: .utf8)!

                    completionHandler(.success(string))
                }

                task.resume()
            }
        }
    }
}
```

```swift
typealias Service<Response> = (_ completionHandler: @escaping (Result<Response, Error>) -> Void) -> ()

@propertyWrapper
struct GET {
    private var url: URL

    init(url: String) {
        self.url = URL(string: url)!
    }

    var wrappedValue: Service<String> {
        get {
            return { completionHandler in
                let task = URLSession.shared.dataTask(with: self.url) { (data, response, error) in
                    guard error == nil else { completionHandler(.failure(error!)); return }

                    let string = String(data: data!, encoding: .utf8)!

                    completionHandler(.success(string))
                }

                task.resume()
            }
        }
    }
}
```

```swift
typealias Service<Response> = (_ completionHandler: @escaping (Result<Response, Error>) -> Void) -> ()

@propertyWrapper
struct GET {
    private var url: URL

    init(url: String) {
        self.url = URL(string: url)!
    }

    var wrappedValue: Service<String> {
        get {
            return { completionHandler in
                let task = URLSession.shared.dataTask(with: self.url) { (data, response, error) in
                    guard error == nil else { completionHandler(.failure(error!)); return }

                    let string = String(data: data!, encoding: .utf8)!

                    completionHandler(.success(string))
                }

                task.resume()
            }
        }
    }
}
```

```swift
typealias Service<Response> = (_ completionHandler: @escaping (Result<Response, Error>) -> Void) -> ()

@propertyWrapper
struct GET {
    private var url: URL

    init(url: String) {
        self.url = URL(string: url)!
    }

    var wrappedValue: Service<String> {
        get {
            return { completionHandler in
                let task = URLSession.shared.dataTask(with: self.url) { (data, response, error) in
                    guard error == nil else { completionHandler(.failure(error!)); return }

                    let string = String(data: data!, encoding: .utf8)!

                    completionHandler(.success(string))
                }

                task.resume()
            }
        }
    }
}
```

```swift
typealias Service<Response> = (_ completionHandler: @escaping (Result<Response, Error>) -> Void) -> ()

@propertyWrapper
struct GET {
    private var url: URL

    init(url: String) {
        self.url = URL(string: url)!
    }

    var wrappedValue: Service<String> {
        get {
            return { completionHandler in
                let task = URLSession.shared.dataTask(with: self.url) { (data, response, error) in
                    guard error == nil else { completionHandler(.failure(error!)); return }

                    let string = String(data: data!, encoding: .utf8)!

                    completionHandler(.success(string))
                }

                task.resume()
            }
        }
    }
}
```

```swift
typealias Service<Response> = (_ completionHandler: @escaping (Result<Response, Error>) -> Void) -> ()

@propertyWrapper
struct GET {
    private var url: URL

    init(url: String) {
        self.url = URL(string: url)!
    }

    var wrappedValue: Service<String> {
        get {
            return { completionHandler in
                let task = URLSession.shared.dataTask(with: self.url) { (data, response, error) in
                    guard error == nil else { completionHandler(.failure(error!)); return }

                    let string = String(data: data!, encoding: .utf8)!

                    completionHandler(.success(string))
                }

                task.resume()
            }
        }
    }
}
```

```swift
typealias Service<Response> = (_ completionHandler: @escaping (Result<Response, Error>) -> Void) -> ()

@propertyWrapper
struct GET {
    private var url: URL

    init(url: String) {
        self.url = URL(string: url)!
    }

    var wrappedValue: Service<String> {
        get {
            return { completionHandler in
                let task = URLSession.shared.dataTask(with: self.url) { (data, response, error) in
                    guard error == nil else { completionHandler(.failure(error!)); return }

                    let string = String(data: data!, encoding: .utf8)!

                    completionHandler(.success(string))
                }

                task.resume()
            }
        }
    }
}
```

```swift
typealias Service<Response> = (_ completionHandler: @escaping (Result<Response, Error>) -> Void) -> ()

@propertyWrapper
struct GET {
    private var url: URL

    init(url: String) {
        self.url = URL(string: url)!
    }

    var wrappedValue: Service<String> {
        get {
            return { completionHandler in
                let task = URLSession.shared.dataTask(with: self.url) { (data, response, error) in
                    guard error == nil else { completionHandler(.failure(error!)); return }

                    let string = String(data: data!, encoding: .utf8)!

                    completionHandler(.success(string))
                }

                task.resume()
            }
        }
    }
}
```

```swift
typealias Service<Response> = (_ completionHandler: @escaping (Result<Response, Error>) -> Void) -> ()

@propertyWrapper
struct GET {
    private var url: URL

    init(url: String) {
        self.url = URL(string: url)!
    }

    var wrappedValue: Service<String> {
        get {
            return { completionHandler in
                let task = URLSession.shared.dataTask(with: self.url) { (data, response, error) in
                    guard error == nil else { completionHandler(.failure(error!)); return }

                    let string = String(data: data!, encoding: .utf8)!

                    completionHandler(.success(string))
                }

                task.resume()
            }
        }
    }
}
```

```swift
typealias Service<Response> = (_ completionHandler: @escaping (Result<Response, Error>) -> Void) -> ()

@propertyWrapper
struct GET {
    private var url: URL

    init(url: String) {
        self.url = URL(string: url)!
    }

    var wrappedValue: Service<String> {
        get {
            return { completionHandler in
                let task = URLSession.shared.dataTask(with: self.url) { (data, response, error) in
                    guard error == nil else { completionHandler(.failure(error!)); return }

                    let string = String(data: data!, encoding: .utf8)!

                    completionHandler(.success(string))
                }

                task.resume()
            }
        }
    }
}
```

```swift
typealias Service<Response> = (_ completionHandler: @escaping (Result<Response, Error>) -> Void) -> ()

@propertyWrapper
struct GET {
    private var url: URL

    init(url: String) {
        self.url = URL(string: url)!
    }

    var wrappedValue: Service<String> {
        get {
            return { completionHandler in
                let task = URLSession.shared.dataTask(with: self.url) { (data, response, error) in
                    guard error == nil else { completionHandler(.failure(error!)); return }

                    let string = String(data: data!, encoding: .utf8)!

                    completionHandler(.success(string))
                }

                task.resume()
            }
        }
    }
}
```

```swift
typealias Service<Response> = (_ completionHandler: @escaping (Result<Response, Error>) -> Void) -> ()

@propertyWrapper
struct GET {
    private var url: URL

    init(url: String) {
        self.url = URL(string: url)!
    }

    var wrappedValue: Service<String> {
        get {
            return { completionHandler in
                let task = URLSession.shared.dataTask(with: self.url) { (data, response, error) in
                    guard error == nil else { completionHandler(.failure(error!)); return }

                    let string = String(data: data!, encoding: .utf8)!

                    completionHandler(.success(string))
                }

                task.resume()
            }
        }
    }
}
```

```
struct API {
    @GET(url: "https://samples.openweathermap.org/data/2.5/weather?id=2172797&appid=b6907d289e10d714a6e88b30761fae22")
    static var getCurrentWeather: Service<String>
}

API.getCurrentWeather { result in
    print(result)
}

success("{\"coord\":{\"lon\":145.77,\"lat\":-16.92},\"weather\":[{\"id\":802,\"main\":\"Clouds\",\"description\":\"scattered clouds\",
\"icon\":\"03n\"}],\"base\":\"stations\",\"main\":{\"temp\":300.15,\"pressure\":1007,\"humidity\":74,\"temp_min\":300.15,\"temp_max\":
300.15},\"visibility\":10000,\"wind\":{\"speed\":3.6,\"deg\":160},\"clouds\":{\"all\":40},\"dt\":1485790200,\"sys\":{\"type\":1,\"id\":
8166,\"message\":0.2064,\"country\":\"AU\",\"sunrise\":1485720272,\"sunset\":1485766550},\"id\":2172797,\"name\":\"Cairns\",\"cod\":200}")
```

```
struct API {
    @GET(url: "https://samples.openweathermap.org/data/2.5/weather?id=2172797&appid=b6907d289e10d714a6e88b30761fae22")
    static var getCurrentWeather: Service<String>
}

API.getCurrentWeather { result in
    print(result)
}

success("{\"coord\":{\"lon\":145.77,\"lat\":-16.92},\"weather\":[{\"id\":802,\"main\":\"Clouds\",\"description\":\"scattered clouds\",
\"icon\":\"03n\"}],\"base\":\"stations\",\"main\":{\"temp\":300.15,\"pressure\":1007,\"humidity\":74,\"temp_min\":300.15,\"temp_max\":
300.15},\"visibility\":10000,\"wind\":{\"speed\":3.6,\"deg\":160},\"clouds\":{\"all\":40},\"dt\":1485790200,\"sys\":{\"type\":1,\"id\":
8166,\"message\":0.2064,\"country\":\"AU\",\"sunrise\":1485720272,\"sunset\":1485766550},\"id\":2172797,\"name\":\"Cairns\",\"cod\":200}")
```

```
struct API {
    @GET(url: "https://samples.openweathermap.org/data/2.5/weather?id=2172797&appid=b6907d289e10d714a6e88b30761fae22")
    static var getCurrentWeather: Service<String>
}

API.getCurrentWeather { result in
    print(result)
}

success("{\"coord\":{\"lon\":145.77,\"lat\":-16.92},\"weather\":[{\"id\":802,\"main\":\"Clouds\",\"description\":\"scattered clouds\",
\"icon\":\"03n\"}],\"base\":\"stations\",\"main\":{\"temp\":300.15,\"pressure\":1007,\"humidity\":74,\"temp_min\":300.15,\"temp_max\":
300.15},\"visibility\":10000,\"wind\":{\"speed\":3.6,\"deg\":160},\"clouds\":{\"all\":40},\"dt\":1485790200,\"sys\":{\"type\":1,\"id\":
8166,\"message\":0.2064,\"country\":\"AU\",\"sunrise\":1485720272,\"sunset\":1485766550},\"id\":2172797,\"name\":\"Cairns\",\"cod\":200}")
```

Time to Recap 💪

Recap

Recap

→ Property Wrappers let us **wrap properties** with **custom behavior**

Recap

→ Property Wrappers let us **wrap properties** with **custom behavior**

→ They are **very good** at **removing boilerplate**

Recap

→ Property Wrappers let us **wrap properties** with **custom behavior**

→ They are **very good** at **removing boilerplate**

→ They provide **great ergonomics** for **ubiquitous constructs**

Recap

- → Property Wrappers let us **wrap properties** with **custom behavior**
- → They are **very good** at **removing boilerplate**
- → They provide **great ergonomics** for **ubiquitous constructs**
- → They can also be **applied** on **functions** 🎉

Recap

- → Property Wrappers let us **wrap properties** with **custom behavior**
- → They are **very good** at **removing boilerplate**
- → They provide **great ergonomics** for **ubiquitous constructs**
- → They can also be **applied** on **functions** 🎉
- → They can **hurt code readability** if **abused**

Recap

- → Property Wrappers let us **wrap properties** with **custom behavior**
- → They are **very good** at **removing boilerplate**
- → They provide **great ergonomics** for **ubiquitous constructs**
- → They can also be **applied** on **functions** 🎉
- → They can **hurt code readability** if **abused**
- → They will **probably lead** to **new and exciting frameworks** 🤩

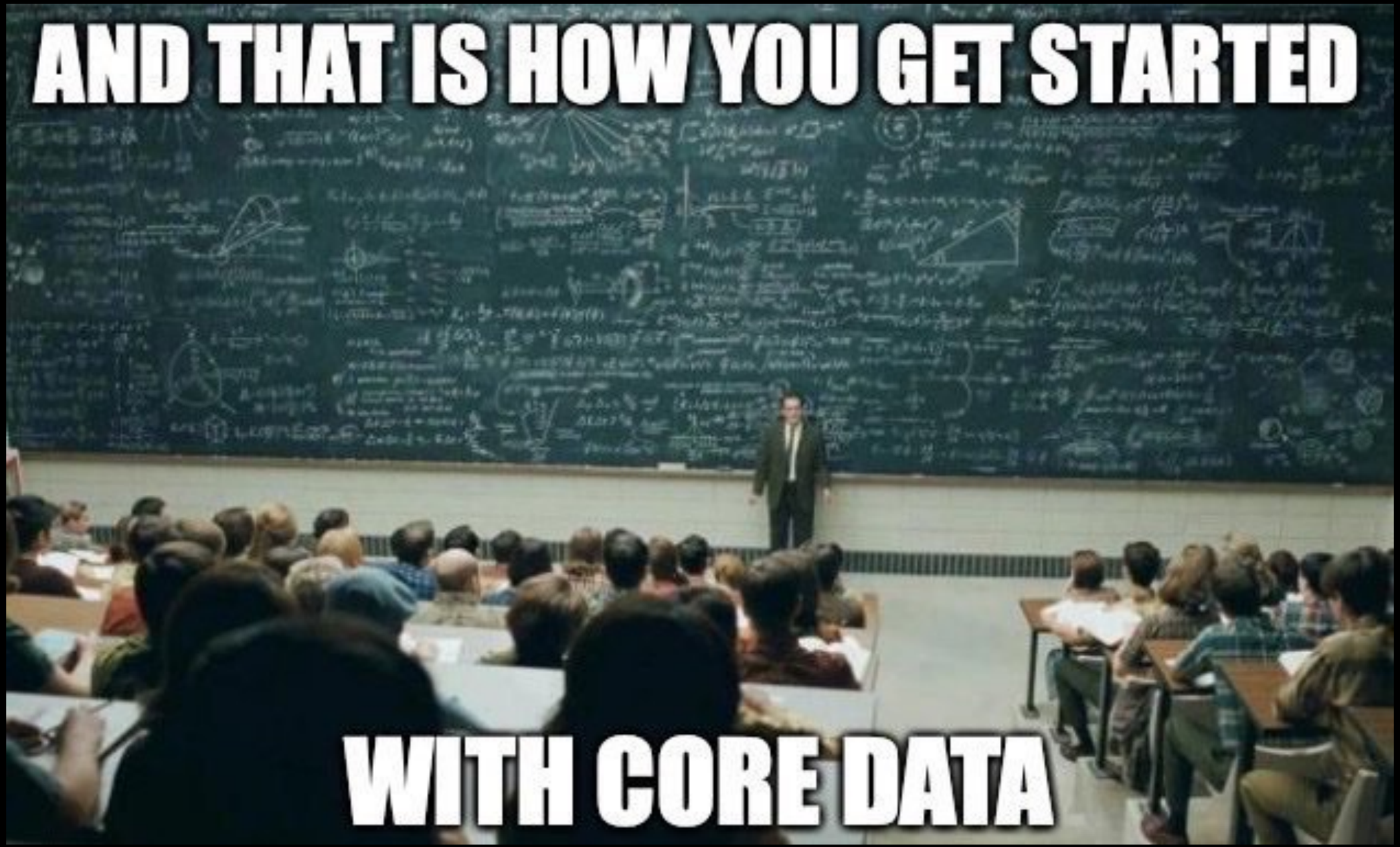
Property Wrappers

or how Swift decided to become Java 🙃

Vincent Pradeilles @v_pradeilles – Worldline 🇫🇷

Up for more iOS Memes?


IOS DEVELOPER
Why is it, when something happens, it is always you three?
UIWEBVIEW
CODE SIGNING
APP REVIEW
imgflip.com


AND THAT IS HOW YOU GET STARTED
WITH CORE DATA


SwiftUI
Combine
Diffable
Deployment Target
iOS Deployment Target
9.0