Enemy of the State

Presentation available at

github.com/jspahrsummers/enemy-of-the-state

Programming is all about abstraction

We want to be using the

best possible

abstractions for development

What even is state?

State

Your stored values at any given time

Mutation

The act of updating some state in place

Variables are state

```
var x: Int

// Store a value into the variable
a = 5

// Update (mutate) the variable, by replacing the value
a = 2

// Mutate the variable again
a++
```

State is

easy

Easy and Simple are not the same

See Rich Hickey's talk, "Simple Made Easy"

Easy

Familiar* or *approachable

Simple

Fewer concepts and concerns

0 0 0 0 0 0 0 F

0 0 0 0 0 0 0 3

Complexity

Mixing (“complecting”) concepts or concerns

State is familiar

(but complex)

All systems have

essential complexity

State also adds

incidental complexity

See Moseley and Marks' paper, "Out of the Tar Pit"

`var visible` → **2 states**

`var enabled` → **4 states**

`var selected` → **8 states!**

`var highlighted` → **16 states!!**

State is just a glorified cache

There are only two hard problems in Computer Science: ***cache invalidation*** *and naming things.*

– Phil Karlton

Table views as a cache

```
@IBAction func addBlankRow(sender: AnyObject!) {
    self.items.append("")

    let indexPath = NSIndexPath(forRow: self.items.count,
        inSection: 0)

    self.tableView.insertRowsAtIndexPaths([ indexPath ],
        withRowAnimation: UITableViewRowAnimation.Automatic)
}
```

See Andy Matuschak's post, "Mutability, aliasing, and the caches you didn't know you had"

State is unpredictable


race conditions
theprofoundprogrammer.com

[State is] ***spooky action at a distance***

– Albert Einstein *(probably)*

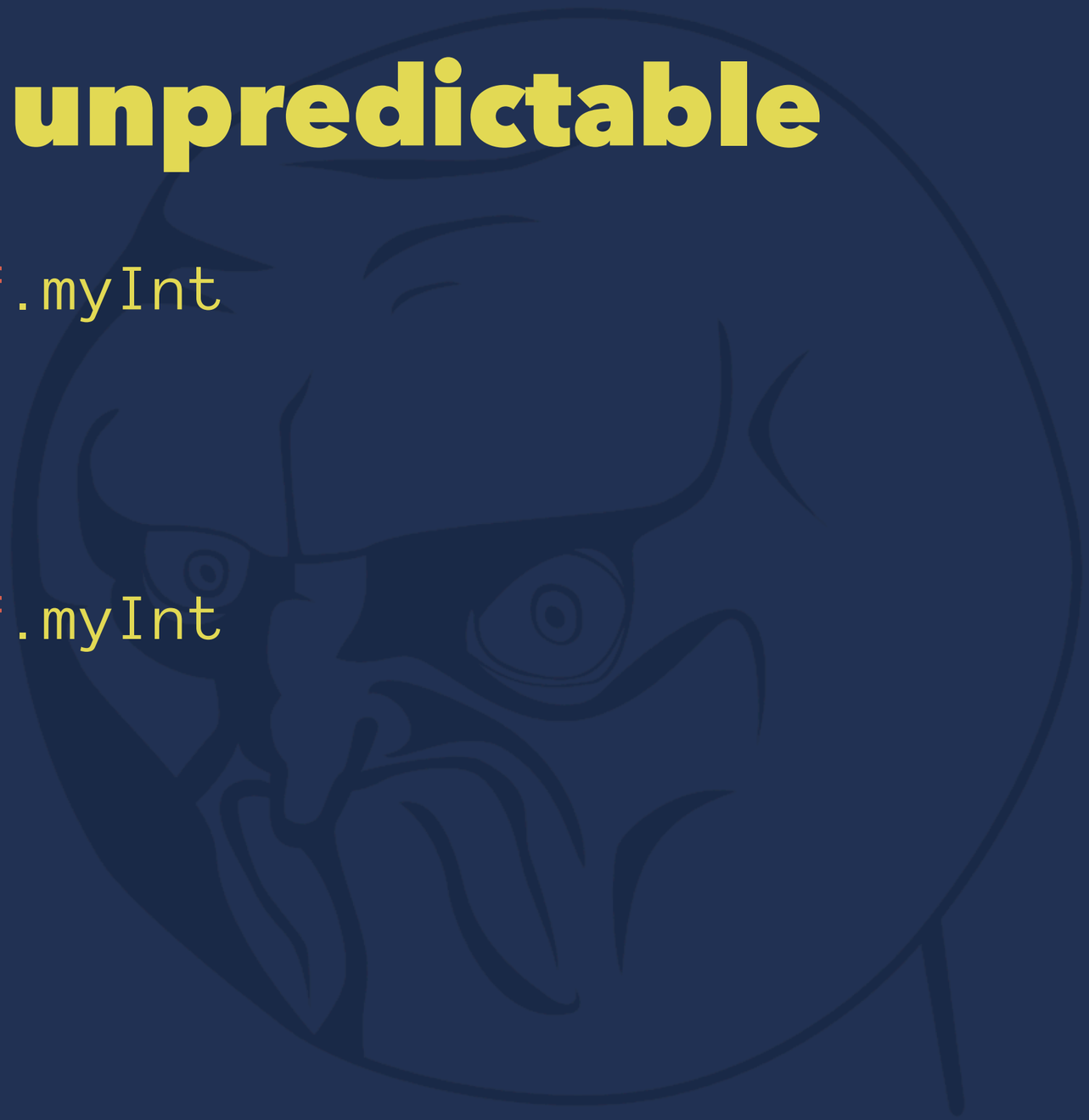
State is unpredictable

```
let x = self.myInt
println(x)
```

==> 5

```
let y = self.myInt
println(y)
```

==> 10 (?!?!)



State is hard to test

Tests

verify expected outputs for certain inputs

State

is an *implicit* input that can change

Example: Testing Core Data

```
id managedObject = [OCMockObject mockForClass:[NSManagedObject class]];
id context = [OCMockObject
    mockForClass:[NSManagedObjectContext class]];

[[context expect] deleteObject:managedObject];
[[[context stub] andReturnValue:@YES] save:[OCMArg anyObjectRef]];

id resultsController = [OCMockObject
    mockForClass:[NSFetchedResultsController class]];

[[[resultsController stub] andReturn:context] managedObjectContext];
[[[resultsController stub] andReturn:managedObject]
    objectAtIndexPath:OCMOCK_ANY];

id viewController = partialMockForViewController();
[[[viewController stub] andReturn:resultsController]
    fetchedResultsController];

[viewController deleteObjectAtIndexPath:nil];
[context verify];
```

from Ash Furrow's C-41 project (sorry, Ash!)


State! State! State! State!

Hey, state happens

- Preferences
- Open documents
- Documents saved to disk
- In-memory or on-disk caches
- UI appearance and content

Minimize state
Minimize complexity

Values
Purity
Isolation

Values

Purity

Isolation

Structs

Enums

Copied

(not shared)

Value types are immutable in Swift

But I can set the properties of a struct in Swift! This guy doesn't know what he's talking about.

– You, the audience

“Mutating” a struct in Swift

```
struct Point {
    var x = 0.0
    var y = 0.0

    mutating func scale(factor: Double) {
        self.x *= factor
        self.y *= factor
    }
}
```

“Mutating” a struct in Swift

```
var p = Point(x: 5, y: 10)
```

“Mutating” a struct in Swift

```
var p = Point(x: 5, y: 10)
p.x = 7       // p = (7, 10)
p.scale(2)    // p = (14, 20)
```

“Mutating” a struct in Swift

```
var p = Point(x: 5, y: 10)
let q = p
```

“Mutating” a struct in Swift

```
var p = Point(x: 5, y: 10)
let q = p

p.scale(2)  // p = (10, 20)
            // q = (5, 10)
```

Here's the key:

Variables mutate
Values never change

“Mutating” a struct in Swift

```
var p = Point(x: 5, y: 10)
let q = p

p.scale(2)  // p = (10, 20)
            // q = (5, 10)

q.x = 2
q.scale(2)  // Error!
```

“Mutating” functions

```
func pointByScaling(factor: Double) -> Point {
    return Point(self.x * factor, self.y * factor)
}

mutating func scale(factor: Double) {
    self.x *= factor
    self.y *= factor
}
```

“Mutating” functions

```
func pointByScaling(self: Point, factor: Double) -> Point {
    return Point(self.x * factor, self.y * factor)
}

mutating func scale(self: Point, factor: Double) {
    self.x *= factor
    self.y *= factor
}
```

“Mutating” functions

```
func pointByScaling(self: Point, factor: Double) -> Point {
    return Point(self.x * factor, self.y * factor)
}

mutating func scale(inout self: Point, factor: Double) {
    self.x *= factor
    self.y *= factor
}
```

Variables mutate
Values never change

So what?

Values are automatically thread-safe

Values are automatically

predictable

Values are predictable

```
let value = self.myData

let x = value.someInt
println(x)
```

==> 5

```
let y = value.someInt
println(y)
```

==> still 5!

Values

Purity

Isolation

Pure functions

Same inputs always yield the same result

Must not have *observable* side effects

```swift
// Pure: concatenates two input strings and
// returns the result.
func +(lhs: String, rhs: String) -> String

protocol GeneratorType {
    // Impure: advances to the next element
    // and returns it.
    mutating func next() -> Element?
}

struct Array {
    // Pure(?)
    var count: Int { get }
}
```

Impure functions are surprising

Insanity is doing the same thing over and over again but expecting ***different results***.

Pure functions are easily tested

Example: Impure formatting

```
func formattedCurrentTime() -> String {
    let now = NSDate()

    let formatter = NSDateFormatter()
    formatter.timeStyle =
        NSDateFormatterStyle.MediumStyle

    return formatter.stringFromDate(now)
}
```

Example: Pure formatting

```
func formattedTimeFromDate(date: NSDate) -> String {
    let formatter = NSDateFormatter()
    formatter.timeStyle =
        NSDateFormatterStyle.MediumStyle

    return formatter.stringFromDate(date)
}
```

Values

Purity

Isolation

Objects should have

only one

reason to change

Isolate

unrelated

pieces of state

Isolation Done Wrong™

```
class MyViewController: UIViewController {
    // When logging in
    var username: String?
    var password: String?

    // After logging in
    var loggedInUser: User?
}
```

Isolation Done Right™

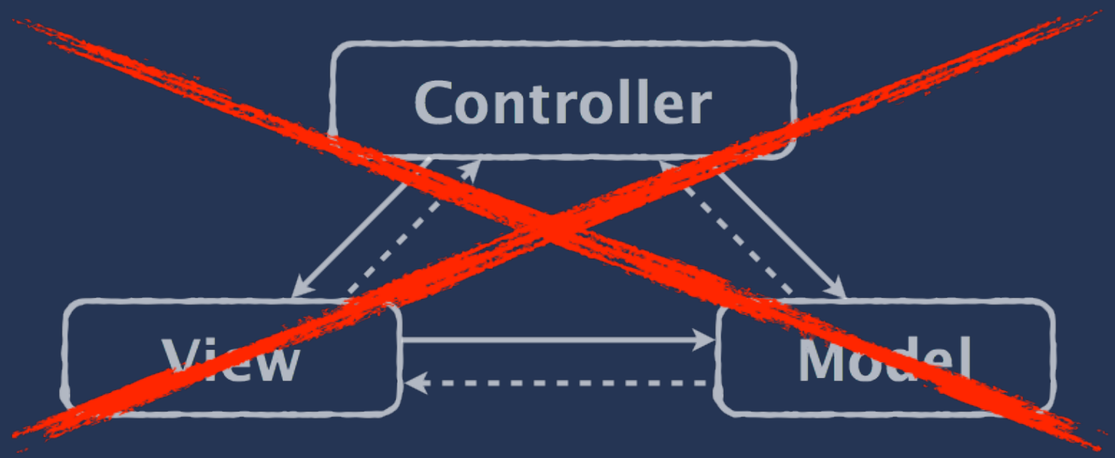
```
// For logging in
class LoginViewModel {
    var username: String?
    var password: String?

    func logIn() -> UserViewModel?
}

// After logging in
class UserViewModel {
    var loggedInUser: User?
}
```

Stateless core, stateful shell

- Keep core **domain logic** in completely immutable value types
- Add **stateful shell objects** with mutable references to the immutable data

See Gary Bernhardt's talk, "Boundaries"

Model-View-ViewModel



Model-View-ViewModel

```
struct User { … }

class UserViewModel {
    var loggedInUser: User?

    func logOut() {
        loggedInUser = nil
    }
}
```

Globals:
Just Say No

Globals get mixed in to

every part

of your program

Isolation reduces complexity
Globals compound it

Singletons are global state

Let's just pass instances around instead

Example: Singleton Networking

```
class APIClient {
    // Access the singleton with APIClient.sharedClient
    class var sharedClient: APIClient {
        struct Singleton {
            static let instance = APIClient()
        }

        return Singleton.instance
    }

    // Fetches the top-level list of categories
    func fetchCategories() -> [Category]
}
```

Example: Instance Networking

```
class APIClient {
    // Fetches the top-level list of categories
    func fetchCategories() -> [Category]
}
```

Example: Instance Networking

```
class MyViewController: UIViewController {
    let client: APIClient

    designated init(client: APIClient) {
        super.init(nibName: nil, bundle: nil)
    }

    override func viewWillAppear(animated: Bool) {
        super.viewWillAppear(animated)

        client.fetchCategories()
    }
}
```

Easily testable
Explicit dependencies
More flexible

Values
Purity
Isolation

TL;DR

Minimize state
Minimize complexity

Learning More

WWDC 2014

Session 229

"Advanced iOS Application Architecture and Patterns"

Check out ReactiveCocoa

github.com/ReactiveCocoa/ReactiveCocoa

Haskell

book.realworldhaskell.org

Elm

elm-lang.org

Thanks to...

Andy Matuschak, Dave Lee, Rob Rix, Matt Diephouse, Josh Vera, Josh Abernathy, Robert Böhnke, Keith Duncan, Landon Fuller, Maxwell Swadling, Alan Rogers, Luke Hefson

and you!