Getting Started With SwiftUI

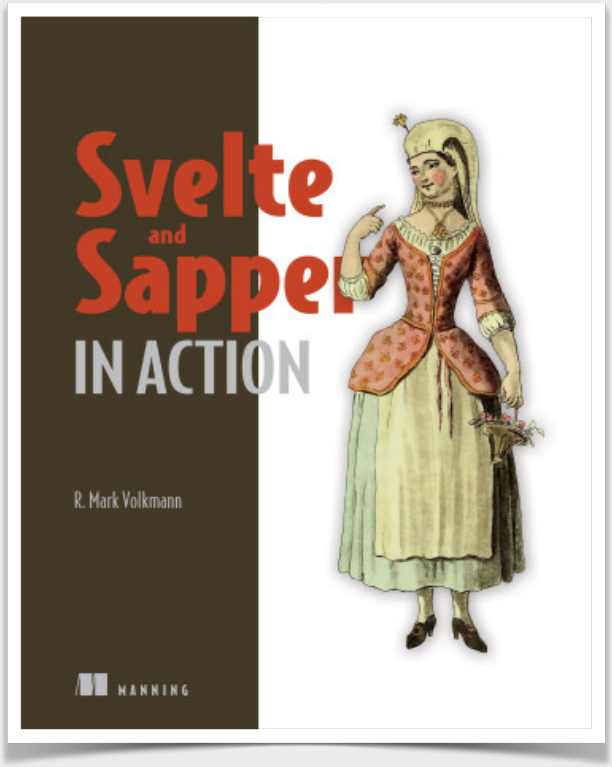
R. Mark Volkmann
Object Computing, Inc.
https://objectcomputing.com
mark@objectcomputing.com
@mark_volkmann

OBJECT COMPUTING
YOUR OUTCOMES ENGINEERED

Slides at https://github.com/mvolkmann/talks/swift/

About Me

- Partner and Distinguished Software Engineer at Object Computing, Inc. in St. Louis, Missouri USA
- 43 years of professional software development experience
- Writer and teacher
- Blog at https://mvolkmann.github.io/blog/
- Author of Manning book “Svelte and Sapper in Action”
- Heavily into SwiftUI development since late 2021



SwiftUI Overview

- A Swift framework for building apps that target many Apple platforms including iOS, iPadOS, watchOS, macOS, and tvOS
  - some features are only available on specific platforms
- Alternative to predecessor UIKit
  - many SwiftUI views are built on UIKit components

https://developer.apple.com/xcode/swiftui/

Comparison to UIKit

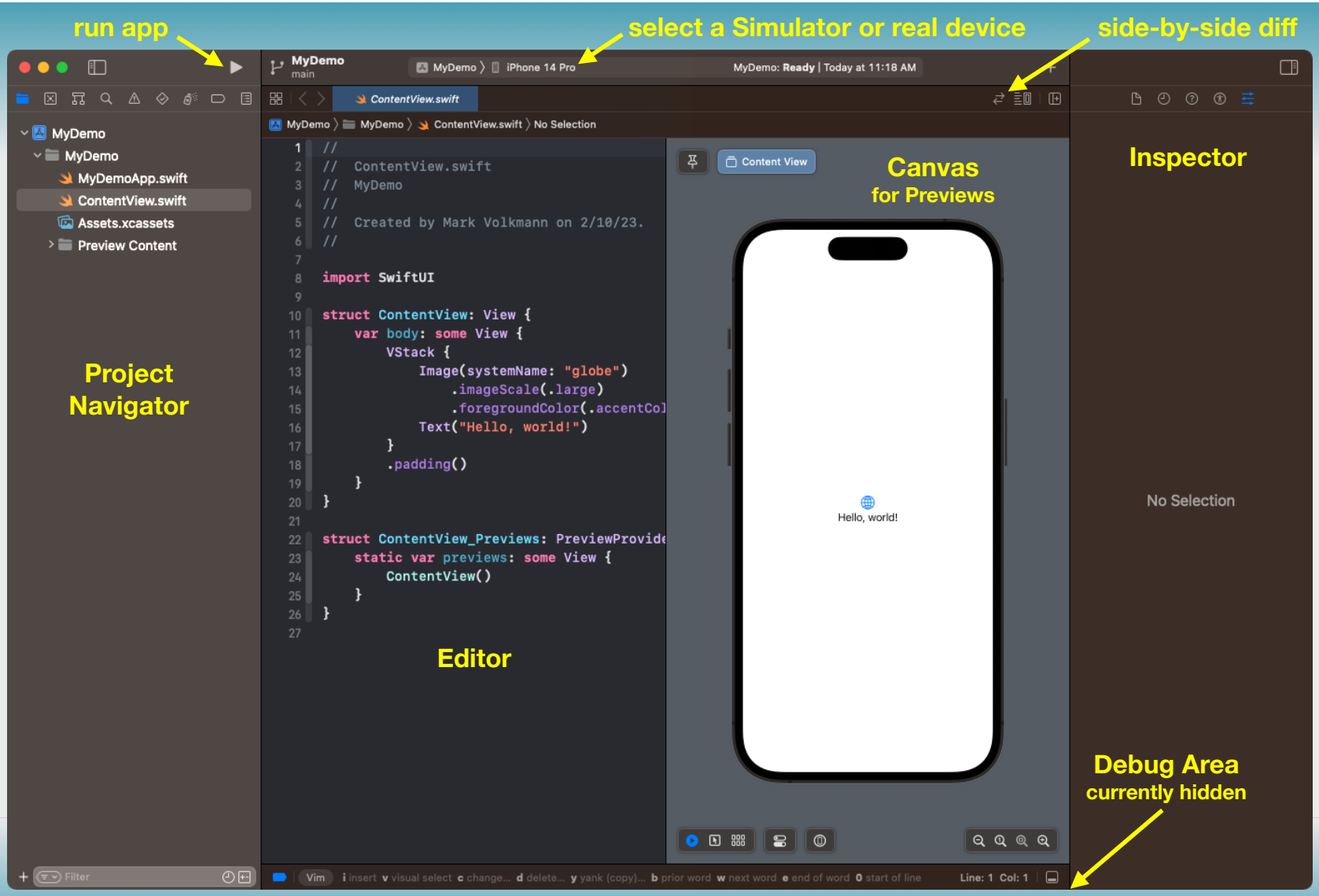
- Unlike UIKit, SwiftUI ...
  - is declarative in nature rather than imperative
  - emphasizes use of structs over classes
  - requires far less code to do the same things
  - has somewhat better performance
  - requires iOS 13 or above
  - is currently missing some features of UIKit
  - doesn't use Storyboard to build views

Xcode

- Preferred IDE for Apple platform development
- Download from https://developer.apple.com/xcode/
  - very large so takes a while
  - initially only includes SDKs for iOS and macOS
  - 23GB after installing watchOS SDK
- Has excellent Git integration
- See my blog for notes on Xcode
  - https://mvolkmann.github.io/blog/
  - see Swift ... Xcode

Creating a New Project

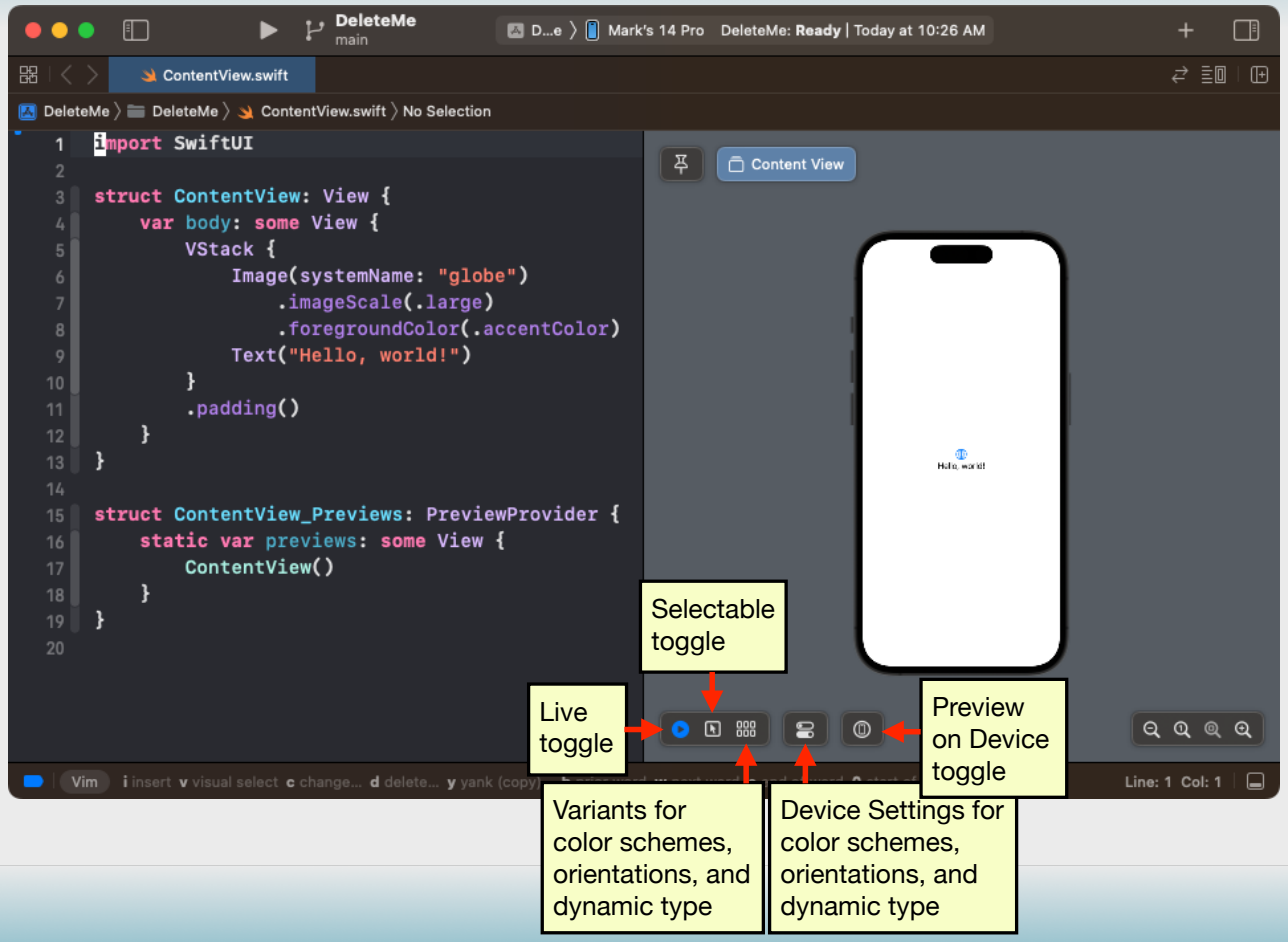
- In Xcode
  - select File ... New ... Project...
  - select “App” template
  - in “Product Name” enter a name
  - keep default “Interface” of “SwiftUI” and default “Language” of “Swift”
  - select directory where project will be created
  - optionally select Editor ... Vim Mode
  - optionally configure use of SwiftFormat to format on save
    - see Swift ... SwiftFormat in my blog


run app
select a Simulator or real device
side-by-side diff
MyDemo
main
MyDemo 〉 iPhone 14 Pro
MyDemo: Ready | Today at 11:18 AM
ContentView.swift
MyDemo 〉 MyDemo 〉 ContentView.swift 〉 No Selection
MyDemo
MyDemo
MyDemoApp.swift
ContentView.swift
Assets.xcassets
Preview Content
Project Navigator
1 //
2 // ContentView.swift
3 // MyDemo
4 //
5 // Created by Mark Volkmann on 2/10/23.
6 //
7
8 import SwiftUI
9
10 struct ContentView: View {
11 var body: some View {
12 VStack {
13 Image(systemName: "globe")
14 .imageScale(.large)
15 .foregroundColor(.accentCol
16 Text("Hello, world!")
17 }
18 .padding()
19 }
20 }
21
22 struct ContentView_Previews: PreviewProvide
23 static var previews: some View {
24 ContentView()
25 }
26 }
27
Editor
Content View
Canvas
for Previews
Hello, world!
Inspector
No Selection
Debug Area
currently hidden
Filter
Vim
i insert v visual select c change... d delete... y yank (copy)... b prior word w next word e end of word 0 start of line
Line: 1 Col: 1

Xcode Previews ...

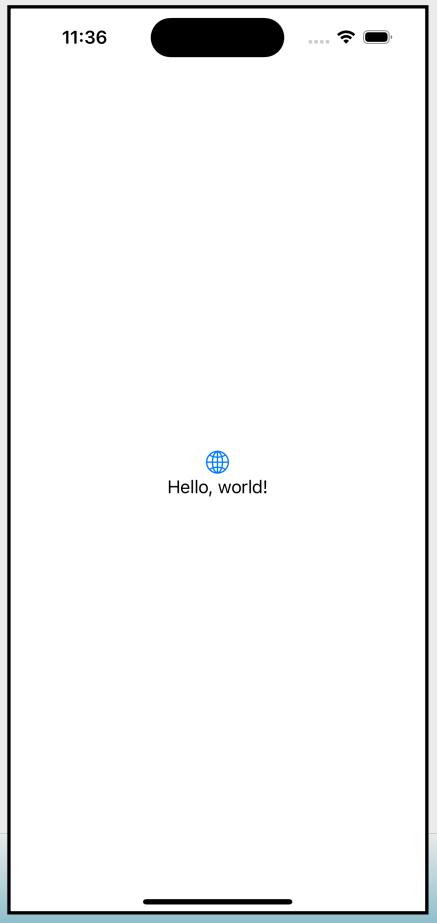
- Typically defined at bottom of **`View`** subtype source files
  - can define any number of previews, but typically only one
- Renders a single view inside Xcode
- Updates automatically after code changes are saved
- Speeds up development
  - compared to rebuilding entire app and running in Simulator
- Some functionality doesn't work in previews
- Keyboard shortcuts
  - toggle display with cmd-option-return
  - restart with cmd-option-p

... Xcode Previews



Running App

- To run in Simulator app
  - select Simulator for a specific device type from dropdown at top
- To run on a real device
  - attach iPhone or iPad to Mac using USB cable
    - can also run wirelessly, but slower
  - select device from dropdown at top
- Click right-pointing triangle in upper-left or press cmd-r



Initial Code

MyDemoApp.swift

```
import SwiftUI

@main
struct MyDemoApp: App {
    var body: some Scene {
        WindowGroup {
            ContentView()
        }
    }
}
```

marks app starting point

ContentView.swift

```
import SwiftUI

struct ContentView: View {
    var body: some View {
        VStack {
            Image(systemName: "globe")
                .imageScale(.large)
                .foregroundColor(.accentColor)
            Text("Hello, world!")
        }
        .padding()
    }
}

struct ContentView_Previews: PreviewProvider {
    static var previews: some View {
        ContentView()
    }
}
```

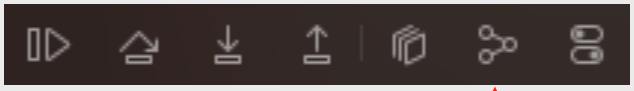
computed property required by `View` protocol

Debug Area

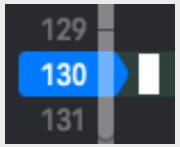
- To open
  - select View ... Debug Area ... Show Debug Area (cmd-shift-y) or drag up from bottom
- Two sides
  - left side displays variable names and values
    - can also hover over variables in source code to see their values
  - right side displays **`print`** function output
- To run app with debugger

default scheme “Run” settings have “Build Configuration” that defaults to “Debug”

  - set one or more breakpoints by clicking on source file line numbers
  - run app
  - click debugger buttons



continue/ pause | step over | step into | step out | view hierarchy | environment overrides | memory graph



right-click breakpoint for options to delete or disable

Swift Programming Language

- Compiled OO and functional programming language
- Strongly typed with type inference
- Introduced by Apple in 2014
- Open source since December 2015
- Goals are to be safe, fast, and expressive
- Built on Low Level Virtual Machine (LLVM)
- Interoperates with Objective-C
- Used for UI development and server-side development with frameworks like Vapor

From Wikipedia, "**LLVM** is a set of compiler and toolchain technologies that can be used to develop a front end for any programming language and a back end for any instruction set architecture."

Proficiency in using SwiftUI requires proficiency in Swift. The next several slides provide a Swift overview.

Swift Types

- Primitives
  - **`Bool`**, **`Int`**, **`Float`**, **`Double`**, **`Character`**, and **`String`**
- Non-primitives
  - **`func`**, **`struct`**, **`class`**, **`enum`**, and **`protocol`**
- Collections (all generic)
  - **`Array`**, **`Set`**, **`Dictionary`**, **`Range`**, tuple, and more

similar to an interface in other languages, but instead of “implementing an interface” you “conform to a protocol”

Syntax Basics

- Variables are declared with **`let`** and **`var`** keywords
  - **`let`** for immutable, **`var`** for mutable
- Statements
  - **do not need** to be terminated with a **semicolon**
- Control flow statements
  - **condition does not need** to be surrounded with **parentheses**
  - **code does need** to be surrounded by **curly braces**

nice tradeoff!

Type Inference

- Basic examples

```
let b = true // inferred type is Bool
let s = "test" // inferred type is String
let i = 19 // inferred type is Int
let d = 3.14 // inferred type is Double
let a = [1, 2, 3] // inferred type is [Int]; same as Array<Int>
let scores = ["Mark": 19, "Tami": 21] // inferred type is [String: Int]
```

Dictionary

- Enum cases
  - in example below, **`border`** is a SwiftUI view modifier method that takes a **`Color enum`** where **`red`** is one of the cases
  - can pass **`Color.red`** or just **`.red`** to take advantage of type inference

```
Text("Hello, World!").border(.red)
```

Control Flow Syntax

```
if condition {
    ...
}

if condition {
    ...
} else {
    ...
}
```

```
for item in collection {
    ...
}

for number in 1...10 {
    ...
}
```

range

```
switch expression {
case value1:
    ...
case value2:
    ...
default:
    ...
}
```

case code doesn't fall through;
break only required if
no other statements

```
while condition {
    ...
}

repeat {
    ...
} while condition
```

Functions ...

- Defined with **`func`** keyword
- Each parameter has a name and an optional “argument label”
  - argument labels are used by callers
  - parameter names are used in function body

goal is to make function calls read like English

- Parameters can be
  - named with no argument label - defaults to name
  - named with an argument label
  - positional indicated by underscore for argument label
    - not usually used, but when they are typically only the first is positional and its meaning is clear from the function name

... Functions

- Argument labels in function calls
  - must appear in the order in which the parameters were defined
  - can improve readability
- Examples

```
func someName(
    _ p1: Type1, // positional
    p2 Type2, // argument label defaults to parameter name
    a3 p3 Type3 // argument label differs from parameter name
) -> ReturnType {
    ...
}
let result = someName(v1, p2: v2, a3: v3)
```

```
func add(_ n1: Int, to n2: Int) -> Int {
    n1 + n2
}
print(add(2, to: 3)) // 5
```

return keyword is optional when body is a single expression

```
func getDailyPercentage(of ingredient: Ingredient, in food: Food) -> Double {
    ...
}
let percent = getDailyPercentage(of: sugar, in: iceCream)
```

argument label

Struct vs. Class ...

- **What they have in common**
  - define a named group of properties and methods
    that can be per-instance (default) or per-type (with **`static`** keyword)
  - can conform to any number of protocols
  - can define any number of initializers (**`init`**) like constructors in other languages
  - define properties with **`let`** (immutable) or **`var`** (mutable) keyword
  - can define computed properties with **`var`** keyword
  - can define methods with **`func`** keyword
  - instances are created by calling the type name
    like a function without a **`new`** keyword

... Struct vs. Class

- **How they differ**
  - structs are value types and classes are reference types
    - multiple variables can refer to the same instance of a class
  - structs are immutable and classes are mutable
    - assigning a struct instance to another variable creates a copy-on-write copy
    - mutating a struct instance property creates a copy of the struct instance
  - structs cannot inherit from another struct or class, but classes can inherit from a single class
  - if no initializer is defined, structs are given a memberwise-initializer, but classes are not

Struct and Class Examples

```
struct Point {
    // Stored properties
    var x: Double
    var y: Double

    // Computed property
    var distanceFromOrigin: Double {
        (x * x + y * y).squareRoot()
    }

    // Instance method
    func log() {
        print("(\(x), \(y))")
    }

    // Mutating instance method
    mutating func translate(dx: Double, dy: Double) {
        x += dx
        y += dy
    }
}

var pt = Point(x: 3, y: 4)
print(pt.distanceFromOrigin) // 5.0
pt.translate(dx: 2, dy: 3)
pt.log() // (5.0, 7.0)
```

uses string interpolation

mutating keyword is only needed in structs

uses free memberwise initializer

creates a copy

```
class Point {
    var x: Double
    var y: Double

    init(x: Double, y: Double) {
        self.x = x
        self.y = y
    }

    var distanceFromOrigin: Double {
        (x * x + y * y).squareRoot()
    }

    func log() {
        print("(\(x), \(y))")
    }

    func translate(dx: Double, dy: Double) {
        x += dx
        y += dy
    }
}

var pt = Point(x: 3, y: 4)
print(pt.distanceFromOrigin) // 5.0
pt.translate(dx: 2, dy: 3)
pt.log() // (5.0, 7.0)
```

must supply an initializer

does not create a copy

Extensions

- Provided and custom types can be extended with new computed properties and methods

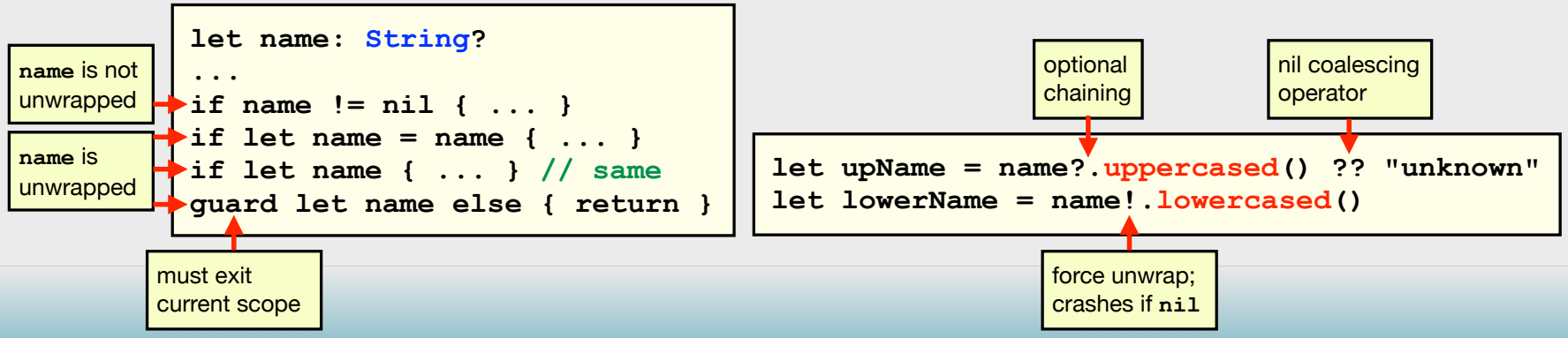
```
import Foundation

extension Date {
    // Returns an abbreviated day of the week (ex. Sun).
    var dayOfWeek: String {
        let dateFormatter = DateFormatter()
        dateFormatter.dateFormat = "EEE"
        return dateFormatter.string(from: self)
    }

    // Returns a new Date that is a given number of days later.
    func daysAfter(_ days: Int) -> Date {
        Calendar.current.date(
            byAdding: .day,
            value: days,
            to: self
        )!
    }

    // Returns the month number and day of the month.
    var md: String {
        let dateFormatter = DateFormatter()
        dateFormatter.dateFormat = "M/d"
        return dateFormatter.string(from: self)
    }
}
```

adds computed properties and a method to the `Date` struct

```
let d = Date.now
// This was run on February 10, 2023 at 1:44 PM
// in the CST time zone which is six hours behind GMT.
print(d) // 2023-02-10 19:44:46 +0000 (GMT)
print(d.dayOfWeek) // Fri
print(d.daysAfter(3)) // 2023-02-13 19:44:46 +0000 (GMT)
print(d.md) // 2/10 (February 10)
```

Optional Types

- Swift uses **`nil`** to represent not having a value
- By default variables cannot be set to **`nil`**
- To enable, declare an optional type with **`?`** suffix; ex. **`String?`**
- When variables have an optional type,
  the Swift compiler requires checking for **`nil`**
- Unwrapping using **`if let`** or **`guard let`** is one way to check



Closures

- Anonymous functions
- Often used to pass a function to a function
- Syntax
  - surrounded by curly braces, just like a block of statements

```
// Using type inference
{ p1, p2 in
    ...
}

// Using explicit types
{ (p1: Type1, p2: Type2) in
    ...
}
```

omit parameter list and `in` keyword if no parameters

Trailing Closures

- Can pass a function as an argument to another function
  - supply a function name or a closure
- Trailing closures provide an alternate syntax
  - only when last parameter expects a function
  - looks like a block of statements - genius syntax choice!
  - for example, consider the **`Array reduce`** method

```
let prices = [1.23, 2.34, 3.45]
// Passing a closure as the last argument
let total = prices.reduce(0, { result, price in
    result + price
})
// Using a trailing closure
let total = prices.reduce(0) { result, price in
    result + price
}
// Using shorthand argument names
let total = prices.reduce(0) { $0 + $1 }
```

KeyPaths

- Provide a path to a property in objects
  - can be data or a function
- Used to retrieve a property value from an object
- Can be passed to a function
  - ex. passing to **`Array`** methods like **`map`** and **`filter`** instead of a closure

```
struct Employee {
    let name: String
    let isDeveloper: Bool
}

let employees = [
    Employee(name: "Ann Able", isDeveloper: true),
    Employee(name: "Bob Barker", isDeveloper: false),
    Employee(name: "Charlie Chaplin", isDeveloper: true),
    Employee(name: "Darla Denny", isDeveloper: false)
]

// Using closures with explicit argument names
let names = employees
    .filter { employee in employee.isDeveloper }
    .map { employee in employee.name }

// Using closures with shorthand argument names
let names = employees.filter { $0.isDeveloper }.map { $0.name }

// Using KeyPaths
let names = employees.filter(\.isDeveloper).map(\.name)
```

Error Handling ...

- **Calling**
  - call functions that can throw in a **`do`** block using **`try`** keyword or from a function that also throws
- **Catching**
  - catch errors in catch blocks that can catch a specific kind of error or any error
- **Throwing**
  - can **`throw`** a value of any type that inherits from **`Error`**
  - there are a small number of provided **`Error`** subtypes
  - new **`Error`** subtypes are typically enums
  - can enable throwing **`String`** values

```
do {
    let v1 = nonThrowingFunction()
    let v2 = try throwingFunction()
    let v3 = try await throwingAsyncFunction()
} catch {
    print("error:", error)
}
```

explained soon

```
enum IntError: Error {
    case negative(_ n: Int)
    case tooHigh(_ n: Int, max: Int)
    case zero
}
```

cases can have "associated data"

```
extension String: LocalizedError {
   public var errorDescription: String? { self }
}
throw "the jig is up"
```

... Error Handling

- **`do`** ... **`catch`** has no **`finally`** like in many other languages
  - but can use a **`defer`** statement before **`do`**
- **`defer`**
  - specifies a block of code to run when current scope exits
  - example uses include
    - hiding a progress indicator
    - closing a resource

async/await

- Asynchronous functions
  - define with **`async`** keyword
  - call in a task with **`await`** keyword

```
func getCurrentCity() async throws -> String {
    let coordinates = await getCurrentCoordinates()
    let address = await getAddress(of: coordinates)
    return address.city
}

Task {
    do {
        let city = try await getCurrentCity()
        // Do something with city.
    } catch {
        print("error:", error)
    }
}
```

can also use `.task { ... }` view modifier on SwiftUI views

Swift Standard Library

- Open source
- Implemented in Swift
- Available by default - **no need to import**
- Defines fundamental types, protocols, and global functions
  - types like **`Bool`**, **`Int`**, **`Float`**, **`Double`**, **`String`**, **`Regex`**, **`KeyPath`**, and **`Optional`**
  - collections like **`Array`**, **`Set`**, **`Dictionary`**, and **`Range`**
  - protocols like **`Comparable`**, **`Equatable`**, **`Hashable`**, **`CaseIterable`**, and **`Codable`**
  - global functions like **`abs`**, **`max`**, **`min`**, **`print`**, **`dump`**, **`readLine`**, and **`assert`**

Foundation Framework

- Not open source
- Mostly implemented in Objective-C
  - but currently being rewritten in Swift and will be open source
- Not available by default - **must import**
  - unless importing another framework that imports Foundation which is typical
- Defines many types including
  - `AttributedString`, `Bundle`, `Calendar`, `CGFloat`, `Data`, `Date`, `DateFormatter`, `DateInterval`, `Decimal`, `Dimension`, `Error`, `FileManager`, `HTTPURLResponse`, `InputStream`, `Locale`, `Measurement`, `NumberFormatter`, `NumberFormatter`, `ObservableObject`, `OutputStream`, `Pipe`, `Port`, `Process`, `ProcessInfo`, `Published`, `RunLoop`, `Stream`, `Thread`, `TimeInterval`, `Timer`, `TimeZone`, `Unit`, `URL`, `URLRequest`, `URLResponse`, `URLSession`, `UserDefaults`, `UUID`, and more

SwiftUI

- Everything that renders on the screen is a kind of **`View`**
- Provides many prebuilt views
  - **`Button`**, **`Color`**, **`Image`**, **`Picker`**, **`Slider`**, **`Stepper`**, **`Text`**, **`TextEditor`**, **`TextField`**, **`Toggle`**, and more
- Provides many container views
  - **`HStack`** (horizontal), **`VStack`** (vertical), and **`ZStack`** (overlapping)
  - **`DisclosureGroup`**, **`Form`**, **`List`**, **`Section`**, **`ScrollView`**, **`TabView`**, and more
- Can define custom views that combine provided views

SF Symbols

- A macOS app from Apple that provides over 4,000 icons
- Rendered using **`Image`** view with **`systemName`** argument
  - ex. `Image(systemName: "heart.filled")`
- Some symbols support multiple rendering modes that enable using different colors for parts of the icon

View Modifiers

- Methods invoked on a view that return another view
- Used for styling (rather than CSS) and many other things
- Can chain multiple view modifier calls
- Order matters
  - ex. adding padding and then a border is not the same as adding a border and then padding
- See many examples ahead two slides

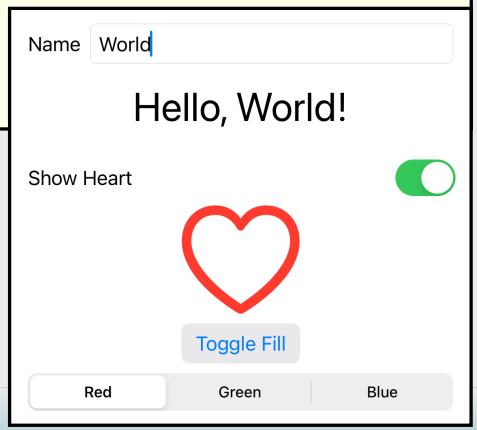
Managing State w/ @State

- SwiftUI views are defined by structs which are immutable
- But often these need mutable state
- For state only used by a single view,
  declare as a property using `@State` property wrapper
  - creates a constant reference to non-constant data
    held in the property wrapper
- See many examples on next slide

Custom SwiftUI Views

```
import SwiftUI

struct ContentView: View {
    @State private var color: Color = .red
    @State private var filled = false
    @State private var name = ""
    @State private var showHeart = false

    let colors: [Color] = [.red, .green, .blue]

    var body: some View {
        VStack {
            // insert code here
            Spacer()
        }
        .padding()
    }
}
```

trailing closure



preceding a `@State` property name with `$` gets a **binding**

```
LabeledContent("Name") {
    TextField("Name", text: $name)
        .textFieldStyle(.roundedBorder)
}
if !name.isEmpty {
    Text("Hello, \(name)!")
        .font(.largeTitle)
}
Toggle("Show Heart", isOn: $showHeart)
if showHeart {
    Image(systemName:
        filled ? "heart.fill" : "heart")
        .resizable()
        .scaledToFit()
        .frame(width: 100)
        .foregroundColor(color)
    Button("Toggle Fill") {
        filled.toggle()
    }
    .buttonStyle(.bordered)
    Picker("Color", selection: $color) {
        Text("Red").tag(Color.red)
        Text("Green").tag(Color.green)
        Text("Blue").tag(Color.blue)
    }
    .pickerStyle(.segmented)
}
```

trailing closure

State With View Models ...

- View models store and manage data that can be shared by multiple views
- Defined by classes that
  - inherit from **`ObservableObject`**
  - have **`@Published`** properties
- Three ways to share
  - inject into environment, use singleton pattern, or pass as argument
- Can inject into environment with **`.environmentObject`** view modifier
- Can access in descendant views with **`@EnvironmentObject`** properties

... State With View Models

```
@main
struct BlackjackApp: App {
    var body: some Scene {
        WindowGroup {
            ContentView().environmentObject(MyViewModel())
        }
    }
}
```

```
class MyViewModel: ObservableObject {
    @Published var score = 0
}
```

can have any number of `@Published` properties

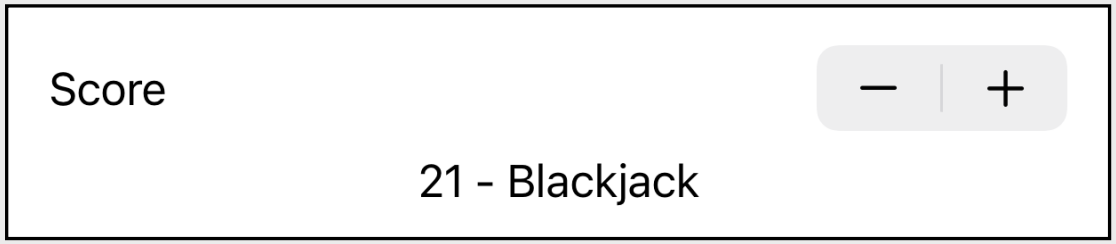
```
struct ContentView: View {
    var body: some View {
        VStack {
            Form()
            Report()
        }
        .padding()
    }
}
```

Each of these files need `import SwiftUI`

```
struct Form: View {
    @EnvironmentObject private var vm: MyViewModel

    var body: some View {
        Stepper("Score", value: $vm.score, in: 0 ... 30)
    }
}
```

```
struct Report: View {
    @EnvironmentObject private var vm: MyViewModel

    private var action: String {
        let s = vm.score
        return s < 18 ? "Hit" :
            s > 21 ? "Bust" :
            s == 21 ? "Blackjack" :
            "Stand"
    }

    var body: some View {
        Text("\(vm.score) - \(action)")
    }
}
```

computed property



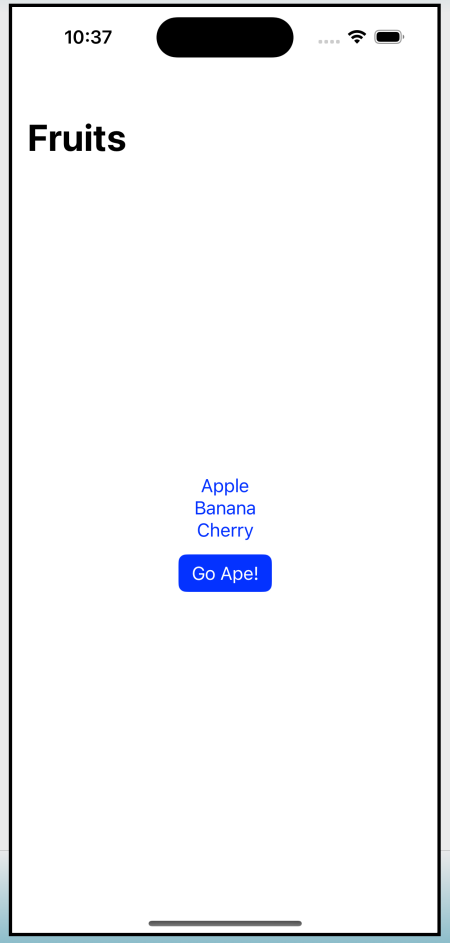
Change Detection

- Two approaches
- Property Observers
  - can define **`willSet`** and **`didSet`** methods on any property
  - supported by Swift
- **`onChange`** view modifier
  - supported by SwiftUI

```
var someProperty = 0 {
    willSet {
        print("old value = \(someProperty)")
        print("new value = \(newValue)")
    }
    didSet {
        print("old value = \(oldValue)")
        print("new value = \(someProperty)")
    }
}
```

```
SomeView() // if only new value is needed
    .onChange(of: someState) { newValue in
        print("new value = \(newValue)")
    }

SomeView() // if both old and new values are needed
    .onChange(of: someState) { [someState] newValue in
        print("old value = \(someState)")
        print("new value = \(newValue)")
    }
```

Navigation ...

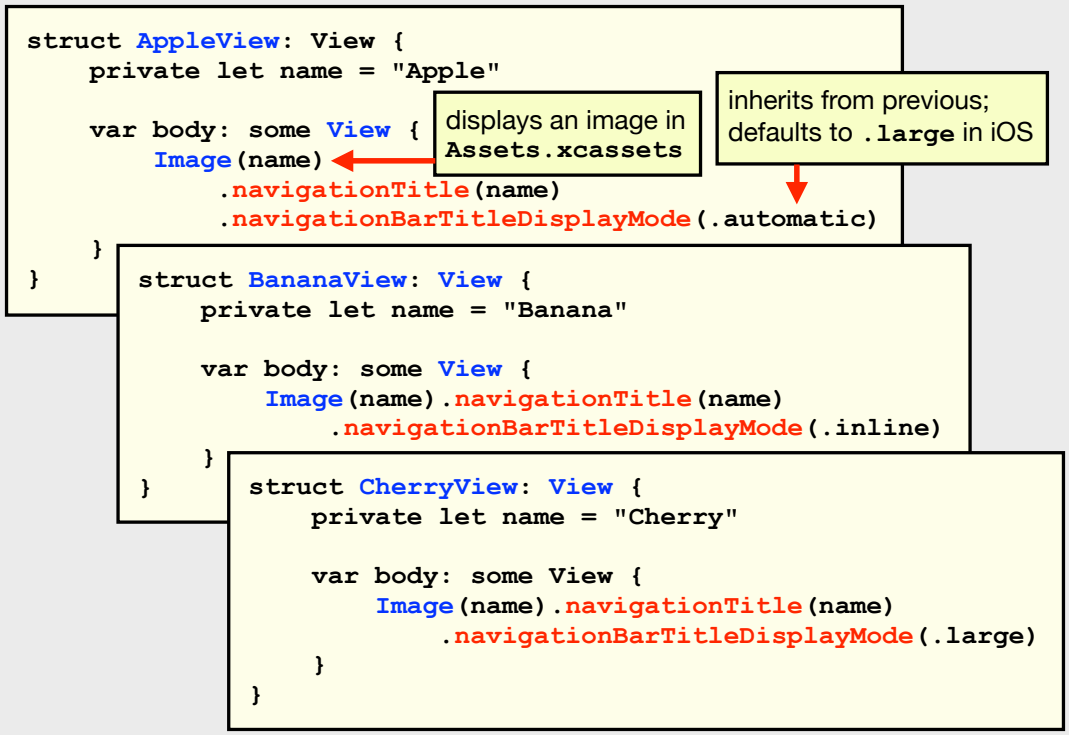



```
import SwiftUI

struct ContentView: View {
    @State private var showingBanana = false
    private let fruits = ["Apple", "Banana", "Cherry"]

    var body: some View {
        NavigationStack {
            VStack {
                ForEach(fruits, id: \.self) { fruit in
                    NavigationLink(fruit, value: fruit)
                }
                Button("Go Ape!") { showingBanana.toggle() }
                    .buttonStyle(.borderedProminent)
            }
            .navigationTitle("Fruits")
            .navigationDestination(for: String.self) { item in
                switch item {
                case "Apple":
                  AppleView()
                case "Banana":
                    BananaView()
                case "Cherry":
                    CherryView()
                default:
                    Text("unsupported fruit")
                }
            }
            .navigationDestination(isPresented: $showingBanana) {
                BananaView()
            }
        }
    }
}
```

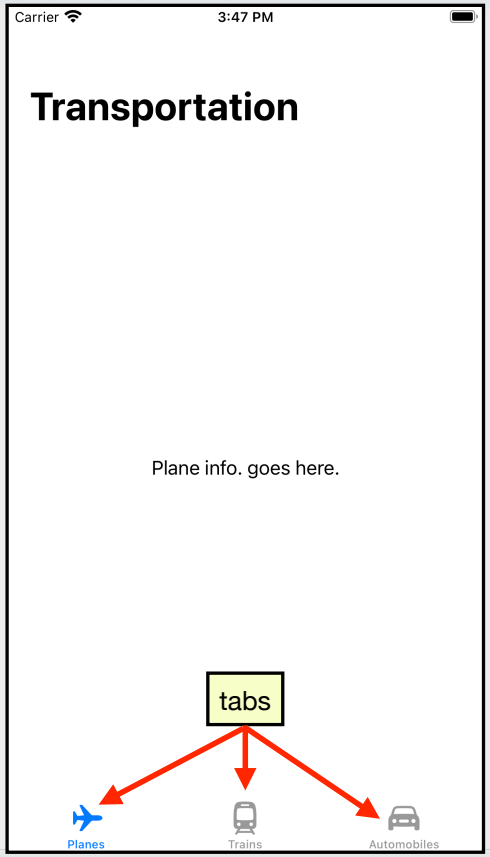
can also use a custom **enum** type here

second way to render this view

... Navigation


struct AppleView: View {
private let name = "Apple"
var body: some View {
Image(name)
.navigationTitle(name)
.navigationBarTitleDisplayMode(.automatic)
}
}
displays an image in Assets.xcassets
inherits from previous; defaults to .large in iOS
struct BananaView: View {
private let name = "Banana"
var body: some View {
Image(name).navigationTitle(name)
.navigationBarTitleDisplayMode(.inline)
}
}
struct CherryView: View {
private let name = "Cherry"
var body: some View {
Image(name).navigationTitle(name)
.navigationBarTitleDisplayMode(.large)
}
}

TabView




```
import SwiftUI

struct ContentView: View {
    var body: some View {
        NavigationStack {
            TabView {
                Planes()
                    .tabItem {
                        Label("Planes", systemImage: "airplane")
                    }
                Trains()
                    .tabItem {
                        Label("Trains", systemImage: "tram")
                    }
                Automobiles()
                    .tabItem {
                        Label("Automobiles", systemImage: "car.fill")
                    }
            }
            .navigationTitle("Transportation")
            // for a smaller, centered title
            // .navigationBarTitleDisplayMode(.inline)
        }
    }
}
```

SF Symbol name

... TabView

```
struct Automobiles: View {
    var body: some View {
        Text("Automobiles info. goes here.")
    }
}
```

```
struct Planes: View {
    var body: some View {
        Text("Plane info. goes here.")
    }
}
```

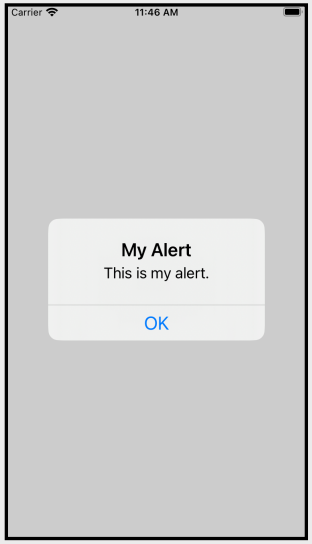
```
struct Trains: View {
    var body: some View {
        Text("Trains info. goes here.")
    }
}
```

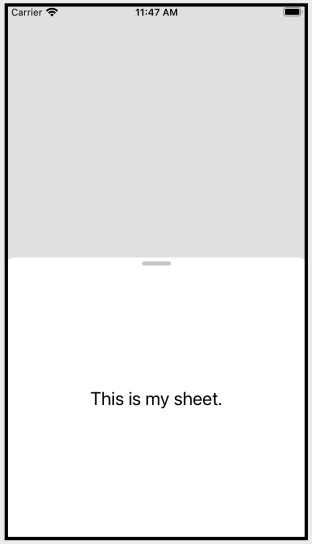
Alerts and Sheets

```
struct ContentView: View {
    @State private var isAlertPresented = false
    @State private var isSheetPresented = false

    var body: some View {
        VStack {
            Button("Toggle Alert") {
                isAlertPresented.toggle()
            }
            Button("Toggle Sheet") {
                isSheetPresented.toggle()
            }
        }
        .padding()
        .alert(
            "My Alert",
            isPresented: $isAlertPresented,
            actions: {},
            message: {
                Text("This is my alert.")
            }
        )
        .sheet(isPresented: $isSheetPresented) {
            Text("This is my sheet.")
                .presentationDetents([.medium])
                .presentationDragIndicator(.visible)
        }
    }
}
```

for custom buttons

controls height





Network Requests

```
private let apiURL = URL(
    string: "https://official-joke-api.appspot.com/random_joke"
)!

enum MyError: Error {
    case badResponseType, badStatus, noData
}

struct Joke: Decodable {
    let setup: String
    let punchline: String
}
```

This sends an HTTP GET request and parses the JSON response into a Swift object.

```
private func getJoke() async -> Joke? {
    do {
        // The data method returns a tuple.
        // The type of response is URLResponse.
        // Cast to HTTPURLResponse to get information from it.
        let (data, response) =
            try await URLSession.shared.data(from: apiURL)
        guard let response = response as? HTTPURLResponse else {
            throw MyError.badResponseType
        }
        guard response.statusCode == 200 else {
            throw MyError.badStatus
        }
        let joke = try JSONDecoder().decode(
            Joke.self,
            from: data
        )
        return joke
    } catch {
        print("getJoke error:", error)
    }
}
```

Persistence Options

- `@AppStorage` property wrapper in SwiftUI
  - simplifies storing and retrieving small amounts of basic data in `UserDefaults` on each device
- **Core Data** from Apple
  - on-device object/graph persistence framework
  - uses SQLite
  - can combine with CloudKit to store data in the cloud
- **CloudKit** from Apple
  - hosted cloud database
- **Realm** from MongoDB
  - alternative to SQLite that is similar to MongoDB
  - data can be stored on each device or in the cloud

Recommended Project Structure

- **`{AppName}App.swift`**
- **`Assets.xcassets`**
  - holds image sets, color sets, and more
- **`Extensions`** directory
  - holds extensions of existing types
- **`Models`** directory
  - holds custom structs that describe application data
- **`Screens`** directory
  - holds custom SwiftUI views that represent entire screens
- **`Views`** directory
  - holds custom SwiftUI views that are used in other views
- **`ViewModels`** directory
  - holds classes that inherit from **`ObservableObject`** and manage **`@Published`** properties

Deployment

- Once an app has been developed it can be made available to beta testers using TestFlight and then released to the App Store
  - both can be accomplished at https://appstoreconnect.apple.com/
  - requires Apple Developer Program account - currently $99/year for individuals
- *Fastlane* is a popular tool for simplifying the process
  - runs tests
  - generates screenshots
  - deploys to TestFlight and App Store
  - https://fastlane.tools/

Also consider **Bitrise** at https://bitrise.io/.

Wrap Up

- We have covered the basics of
  the Swift programming language
  and the SwiftUI framework
- There is much more to learn
  - check out my blog described on next slide
- For developing native Android apps,
  check out the **Kotlin** programming language and
  the **JetPack Compose** framework which is similar to SwiftUI

My Blog

- See Swift category at https://mvolkmann.github.io/blog/
  - provides much more information about Swift, SwiftUI, and associated frameworks and tools
  - topics include App Reviews, AppInfo, ARKit, CloudKit, Concurrency, Core Data, DocC, Face ID, Fastlane, Gauges, Grid, HealthKit, Instruments, Launch Screens, Layout Protocol, Localization, Lottie, MapKit, Navigation, Notifications, PhotosPicker, Project Structure, Reality Composer, Shortcuts, Simulator, SpriteKit, StoreKit, Swift Charts, Swift Package Manager, SwiftFormat, SwiftLint, TestFlight, UIKit, Vapor, watchOS, WeatherKit, Widgets, Xcode, and XCTest