Architecture

- **MVVM**
  Design paradigm

- **Varieties of Types**
  - `struct`
  - `class`
  - `protocol`
  - “Don’t Care” type (aka generics)
  - `enum`
  - `functions`
MVVM

Model-View-ViewModel

A “code organizing” architectural design paradigm.
Works in concert with the concept of “reactive” user-interfaces.
Must be adhered to for SwiftUI to work.
It is different from MVC (Model View Controller) that UIKit (old-style iOS) uses.
MVVM

Model
UI Independent
Data + Logic
“The Truth”

View
MVVM

Model
UI Independent
Data + Logic
“The Truth”

View
Reflects the Model
Stateless
Declared
Reactive

data flows this way (i.e. read-only)
MVVM

ViewModel

Binds View to Model
Interpreter

Model

UI Independent
Data + Logic
“The Truth”

View

Reflects the Model
Stateless
Declared
Reactive
MVVM

ViewModel

Binds View to Model
Interpreter

Model

UI Independent
Data + Logic
“The Truth”

View

Reflects the Model
Stateless
Declared
Reactive

notices changes
MVVM

Model
UI Independent
Data + Logic
“The Truth”

ViewModel
Binds View to Model
Interpreter
might “interpret”
notices changes

View
Reflects the Model
Stateless
Declared
Reactive
MVVM

ViewModel

might “interpret”
publishes “something changed”

notices changes
Binds View to Model
Interpreter

View

Reflects the Model
Stateless
Declared
Reactive

Model

UI Independent
Data + Logic
“The Truth”
MVVM

Model

UI Independent
Data + Logic
"The Truth"

ViewModel

Binds View to Model
Interpreter

View

Reflects the Model
Stateless
Declared
Reactive

ObservableObject
@Published
objectWillChange.send()
.environmentObject()

might "interpret"

publishes "something changed"

notices changes

automatically observes publications, pulls data and rebuilds

@ObservedObject
@Binding
.onReceive
@EnvironmentObject

Reacting
MVVM

Model

ViewModel
  Binds View to Model
  Interpreter

View

What about the other direction?

UI Independent
Data + Logic
“The Truth”

Reflects the Model
Stateless
Declared
Reactive
MVVM

ViewModel

Binds View to Model
Interpreter
Processes Intent

Model

UI Independent
Data + Logic
“The Truth”

View

Reflects the Model
Stateless
Declared
Reactive

What about the other direction?
MVVM

ViewModel
Binds View to Model
Interpreter
Processes Intent

Model
UI Independent
Data + Logic
“The Truth”

View
Reflects the Model
Stateless
Declared
Reactive

What about the other direction?
MVVM

ViewModel
- Binds View to Model
- Interpreter
- Processes Intent

What about the other direction?

Model
- UI Independent
- Data + Logic
- "The Truth"

View
- Reflects the Model
- Stateless
- Declared
- Reactive

MVVM
- Processes Intent function
- Calls Intent function
- Modifies the Model
- Binds View to Model
MVVM

ViewModel

Binds View to Model
Interpreter
Processes Intent

might "interpret"
publishes "something changed"
calls Intent function
automatically observes publications, pulls data and rebuilds

Model

UI Independent
Data + Logic
"The Truth"

View

Reflects the Model
Stateless
Declared
Reactive

views changes
modifies the Model

interprets
MVVM

ViewModel
Binds View to Model
Interpreter
Processes Intent

Model
UI Independent
Data + Logic
"The Truth"

View
Reflects the Model
Stateless
Declared
Reactive

ObservableObject
@Published
objectWillChange.send()
.environmentObject()

might "interpret"

publishes "something changed"

calls Intent function

automatically observes publications, pulls data and rebuilds

@ObservedObject
@Binding
.onReceive
@EnvironmentObject

environmentObject()
Architecture

**MVVM**
- Design paradigm

**Varieties of Types**
- `struct`
- `class`
- `protocol`
- “Dont’ Care” type (aka generics)
- `enum`
- `functions`
Both `struct` and `class` have ...

... pretty much exactly the same syntax.

stored vars (the kind you are used to, i.e., stored in memory)

```swift
var isFaceUp: Bool
```
Both **struct** and **class** have ...

... pretty much exactly the same syntax.

- stored *vars* (the kind you are used to, i.e., stored in memory)
- computed *vars* (i.e. those whose value is the result of evaluating some code)

```swift
var body: some View {
    return Text("Hello World")
}
```
Both **struct** and **class** have ...

... pretty much exactly the same syntax.

- **stored vars** (the kind you are used to, i.e., stored in memory)
- **computed vars** (i.e. those whose value is the result of evaluating some code)
- **constant lets** (i.e. **vars** whose values never change)

```swift
let defaultColor = Color.orange
```

```swift
CardView().foregroundColor(defaultColor)
```

---

**struct** and **class**
Both **struct** and **class** have ...

... pretty much exactly the same syntax.

stored vars (the kind you are used to, i.e., stored in memory)

computed vars (i.e. those whose value is the result of evaluating some code)

constant lets (i.e. vars whose values never change)

functions

```swift
func multiply(_ operand: Int, by otherOperand: Int) -> Int {
    return operand * otherOperand
}
```

```swift
multiply(5, by: 6)
```

```swift
func multiply(operand: Int, by: Int) -> Int {
    return operand * by
}
```

```swift
multiply(operand: 5, by: 6)
```

```swift
func multiply(_ operand: Int, by otherOperand: Int) -> Int {
    return operand * otherOperand
}
```

```swift
multiply(5, by: 6)
```
Both **struct** and **class** have ...

... pretty much exactly the same syntax.

- stored **vars** (the kind you are used to, i.e., stored in memory)
- computed **vars** (i.e. those whose value is the result of evaluating some code)
- constant **lets** (i.e. **vars** whose values never change)
- **functions**
- **initializers** (i.e. special functions that are called when creating a **struct** or **class**)  

```swift
struct MemoryGame {  
    init(numberOfPairsOfCards: Int) {  
        // create a game with that many pairs of cards  
    }  
}
```
Both `struct` and `class` have ...

... pretty much exactly the same syntax.

- stored `vars` (the kind you are used to, i.e., stored in memory)
- computed `vars` (i.e. those whose value is the result of evaluating some code)
- constant `lets` (i.e. `vars` whose values never change)
- `functions`
- `initializers` (i.e. special functions that are called when creating a `struct` or `class`)

So what's the difference between `struct` and `class`?
**struct and class**

**struct**
- Value type
- Copied when passed or assigned
- Copy on write
- Functional programming
- No inheritance
- "Free" `init` initializes ALL vars
- Mutability must be explicitly stated
- Your "go to" data structure

Everything you've seen so far is a struct (except View which is a protocol)

**class**
- Reference type
- Passed around via pointers
- Automatically reference counted
- Object-oriented programming
- Inheritance (single)
- "Free" `init` initializes NO vars
- Always mutable
- Used in specific circumstances

The ViewModel in MVVM is always a class (also, UIKit (old style iOS) is class-based)
Generics

Sometimes we just don’t care

We may want to manipulate data structures that we are “type agnostic” about. In other words, we don’t know what type something is and we don’t care. But Swift is a strongly-typed language, so we don’t use variables and such that are “untyped.” So how do we specify the type of something when we don’t care what type it is? We use a “don’t care” type (we call this feature “generics”) ...
Generics

Example of a user of a “don’t care” type: Array

Awesome example of generics: Array.
An Array contains a bunch of things and it doesn’t care at all what type they are!
But inside Array’s code, it has to have variables for the things it contains. They need types.
And it needs types for the arguments to Array functions that do things like adding items to it.
Enter … GENERICS.
Generics

How Array uses a “don’t care” type

Array’s declaration looks something like this …

```swift
struct Array<Element> {
    ...
    func append(_ element: Element) { ... }
}
```

The type of the argument to `append` is `Element`. A “don’t care” type. Array’s implementation of `append` knows nothing about that argument and it does not care. `Element` is not any known struct or class or protocol, it’s just a placeholder for a type.

The code for using an Array looks something like this …

```swift
var a = Array<Int>()
a.append(5)
a.append(22)
```

When someone uses Array, that’s when `Element` gets determined (by `Array<Int>`).
Generics

How Array uses a “don’t care” type

Array’s declaration looks something like this …

```swift
struct Array<Element> {
    ...
    func append(_ element: Element) { ... }
}
```

Note that Array has to let the world know the names of all of its “don’t care” types in its API. It does this with the `< >` notation on its struct declaration `Array<Element>` above. That’s how users of Array know that they have to say what type Element actually is.

```swift
var a = Array<Int>()
```

It is perfectly legal to have multiple “don’t care” types in the above (e.g. `<Element, Foo>`)

Generics

Type Parameter

I will often refer to these types like Element in Array as a “don’t care” type. But its actual name is Type Parameter.
Other languages most of you may know (e.g. Java) have a similar feature. However, Swift combines this with protocols to take it all to the next level. We’ll talk about that next week!
Functions as Types

Functions are people* too! (* er, types)

You can declare a variable (or parameter to a func or whatever) to be of type “function”. The syntax for this includes the types of the arguments and return value.

You can do this anywhere any other type is allowed.

Examples …

(Int, Int) -> Bool // takes two Ints and returns a Bool
(Double) -> Void // takes a Double and returns nothing
() -> Array<String> // takes no arguments and returns an Array of Strings
() -> Void // takes no arguments and returns nothing (this is a common one)

All of the above a just types. No different than Bool or View or Array<Int>. All are types.

var foo: (Double) -> Void // foo’s type: “function that takes a Double, returns nothing”

func doSomething(what: () -> Bool) // what’s type: “function, takes nothing, returns Bool”
Functions as Types

Functions are people* too! (* er, types)

Example …

```swift
var operation: (Double) -> Double
This is a var called operation.
It is of type “function that takes a Double and returns a Double”.

Here’s a simple function that takes a Double and returns a Double …
```func square(operand: Double) -> Double {
    return operand * operand
}
```

operation = square // just assigning a value to the operation var, nothing more
let result1 = operation(4) // result1 would equal 16
Note that we don’t use argument labels (e.g. operand:) when executing function types.
operation = sqrt // sqrt is a built-in function which happens to take and return a Double
let result2 = operation(4) // result2 would be 2
We’ll soon see an example of using a function type for a parameter to a function in our demo.
Functions as Types

Closures

It’s so common to pass functions around that we are very often “inlining” them. We call such an inlined function a “closure” and there’s special language support for it. We’ll cover this in the demo and again later in the quarter.

Remember that we are mostly doing “functional programming” in SwiftUI. As the very name implies, “functions as types” is a very important concept in Swift. Very.
MVVM and Types in Action

Now that we know about MVVM, let’s implement it in our Memorize application. In doing so, we’ll see a lot of what we just talked about ...

We’re going to use the special `init` function (in both our Model and our ViewModel).

We’re going to use generics in our implementation of our Model.

We’re going to use a function as a type in our Model.

We’re going to see a class for the first time (our ViewModel will be a class).

We’re going to implement an “Intent” in our MVVM.

And finally, we will make our UI “reactive” through our MVVM design.

Whew! Let’s get started ...