

## Table of Contents

- Table of Contents
- Swift UI
  - Set up your first iOS app
  - Xcode
    - \* Navigator
    - \* Inspector
    - \* Main editing window
  - Assets
  - Info.plist
  - AppnameApp.swift
  - ContentView.swift : Your first view
    - \* Some
    - \* Text
    - \* ContentView\_Previews
    - \* Padding
- Animation
  - So what is a ViewModifier?
  - How to apply a ViewModifier?
  - Animation
    - \* Implicit Animation
    - \* Implicit vs Explicit Animation
    - \* Explicit Animation
  - Transitions
    - \* How to apply transitions
    - \* Setting Animation Details for a Transition
  - Matched Geometry Effect
    - \* .onAppear
    - \* Shapes and ViewModifier Animation

## Swift UI

A declarative user interface for developing apps on every Apple platform. SwiftUI provides views, controls, and layout structures for declaring your app user interface. The framework provides event handlers for delivering taps, gestures, and other types of input, and tools to manage the flow of data from app's models down to the views and controls. [Apple Official Documentation](#)

Swift UI uses the Swift programming language. To program on SwiftUI download the Apple Developer Tool (no Apple Developer Program needed) Xcode. From iPad OS 15 you can use Swift Playgrounds to build and publish an iOS app using SwiftUI.

### Set up your first iOS app

1. Launch Xcode
2. Create a new Xcode project
3. Choose which template you need (e.g. iOS app, macOS app, watchOS app, multiplatform app)
4. Inside iOS app we can build more than one type of app (e.g. App, Document App, Game App, Augmented Reality App, Sticker Pack App, iMessage App)
5. Select App
6. Add info required to build the app
  - (a) Add product name (app name)
  - (b) Development Team (create a development team and add your apple id)
  - (c) Organization Identifier: a string that identify your company or person as individual (it is written in reverse dns notation)
  - (d) Interface
    - i. SwiftUI
    - ii. Storyboard (old way of building UI)
  - (e) Lifecycle
    - i. SwiftUI app
    - ii. UIKit
  - (f) Language
    - i. Swift (only choice if you choose SwiftUI)
    - ii. Objective C

- (g) Core Data (Object oriented database ORM)
- (h) Tests (to test UI)
- 7. Choose a folder, better if in the home directory in a folder named Directory
- 8. Remember to create a Git repository under (Source Control)

## **Xcode**

### **Navigator**

Left panel is called Navigator. It lets you navigate inside your navigation. There are several buttons on top. Each one lets you navigate in a different way: - by file (default) - by searching - by debugging - ....

### **Inspector**

Right panel is called Inspector and lets you inspect elements and their properties dynamically

### **Main editing window**

Is in the center. Can be split up with the Previewer that previews our UI without running in the simulator or with another file. On top of our main window we can find a breadcrumb navigator that lets us know where we are and what we have selected.

### **Assets**

Assets are accessible from the folder Assets.xcassets. Usually you drag files inside assets and then give a name like a variable. AppIcon is a special asset that requires more formats of images.

### **Info.plist**

Info is essentially the settings of our App. Although we probably never gonna use it since its interface is a little bit raw and cumbersome. Instead clicking on top of our project (with the blue app store icon) we got a nice UI for editing the same settings.

### **AppnameApp.swift**

It is the main file. It does not do much other than launching the app calling the main view.

## ContentView.swift : Your first view

Swift UI does a great job separating the interface from the logic. A View and most of the types we are gonna use in Swift UI are Struct. Struct are commons in other language programs are an abbreviation for Data Structure. A struct can be seen as a collections of variables, however we'll see they are powerful types on which most of the Swift UI elements are built up. Struct have variables but can also have functions. They are similar to classes but we'll see they lack inheritance. Struct are not object oriented things. Swift support also classes and supports both Object Oriented Programming or Functional Programming. We'll use OOP to hook our models to the UI, while our logic and UI will basically a set of structs.

A basic Swift UI view can be:

```
struct MyFirstView: View {  
    var body: some View {  
        Text("Hello World")  
    }  
}
```

In functional programming the behavior of things is important. In a Struct we define how things works with function. It also describe the storage but it does not encapsulate the storate. It says what data there is in the struct, it describes them but it does not care of values and how these variables are stored. In this case the struct MyFirstView “behaves” like a View. Behaving like something in functional programming usually mean we inherit some work already done for us but also means we may need to provide something to conform to the type. In the case of View, we'll see that View is a Swift protocol. It is possible to create custom views by declaring types that conform to the View protocol. To conform to the protocol we need to implement the required body computed property to provide the content for our custom view. To declare a variable inside a struct the syntax is simple:

```
var body: some View {  
    //some view in here  
}
```

Colon some View indicates the type. It is more complicated than just declaring a variable in swift:

```
var i: Int  
var s: String
```

### Some

There is also a weird some keyword. Why is the body some instead of the type? It is literal, the variable body behaves something like a View. We don't know

what is gonna return, but we know that the body variable return one or a set of more sub-views that are encapsulated together (lego example). In this case we wanna tell to the compiler to check if what the var body returns behaves like some View (it could be text, image, combined views etc.) without knowing what are we gonna return. If we replace “some View” with “Text” it is still gonna work, but when we are going to change the body and things get more complicated, it may break.

After some view we have something that looks like a function. Indeed it is a function. In functional programming we can drop functions everywhere. This function does not take arguments, does not need a name, it does not need one. This function return a Text. The return statement is implicit (you can add it if you want to try, it still works).

How come we have a function right after a variable? The var body is not actually stored in memory, it is a variable that is calculated executing this function. Everytime someone asks the content of this struct, Swift UI executes the function and return the calculated value.

## **Text**

What is a Text? Text is another struct that behaves like a View. From Apple docs the definition is the following: A view that displays one or more lines of read-only text.

## **ContentView\_Previews**

While ContentView is what we see in the main view of the app, ContentView\_Previews is not part of the Application. It is there to initialize the Preview.

## **Padding**

Padding is a function that can be applied to every struct that behaves like a View. It is a function that return a new View modified (in this case with added padding to the original view). It is actually a ViewModifier. So, when we add padding to a Text the result is not gonna be a Text but some kind of View.

## **Animation**

Animation is very important in a mobile UI. Swift makes very easy to do. There are basically two ways to do animation: 1. by animating a Shape 2. by animating Views via their ViewModifiers

## So what is a ViewModifier?

View modifiers are all those little functions that modified our Views (like aspectRatio, padding etc) They are (likely) turning right around and calling a function in View called Modifier. e.g. .aspectRatio(2/3) is likely something like .modifier(AspectRatio(2/3)) AspectModifier can be anything that conforms to the ViewModifier protocol.

Apple Documentation - View Modifier A modifier that you apply to a view or another view modifier, producing a different version of the original value.

ViewModifier is a protocol that lets us create a reusable modifier that can be applied to any view.

Conceptually, this protocol is sort of like this...

```
protocol ViewModifier{
    typealias Content //the type of the View passed to body(content:)
    func body(content: Content) -> some View {
        //return some View that almost
        //certainly contains the View content
    }
}
```

An example of view modifier is:

```
struct BorderdLabel: ViewModifier{
    var isSet: Bool
    func body(content: Content) -> some View {
        content
        .font(.caption2)
        .padding(10)
        .overlay(RoundedRectangle(cornerRadius:10))
    }.foregroundColor(Color.blue)
}
```

Where Content is the View that we are actually going to modify. The code is similar to the View code, but we have func body(content:) instead of var body. That's because ViewModifiers are Views, and writing the code for one is almost identical. There is also a special ViewModifier, GeometryEffect, for building geometry modifiers (scaling, rotation etc..).

## How to apply a ViewModifier?

We have two ways to apply a view modifier: 1. Apply the modifier directly to the view (e.g. Text("Some Text").modifier(BorderedLabel())) 2. Create an extension that use a modifier to the View itself and return the View modified

```
extension View{
```

```

    func borderedLabel(isSet: Bool) -> some View {
        return self.modifier(BorderedLabel(isSet: isSet))
    }
}

```

It is important to highlight a couple of things on how this works. 1. The content argument is just the Text view we pass 2. We pass the arguments like in the View 3. We could have an init (it is allowed)

These arguments are crucial in ViewModifiers since when this changes it will kick off an animation. ViewModifiers always return a View (some View) not multiple views or something else. They take a view and return a new one (remember they are structs, they are read only in memory so we return always a new one).

P.S. (ViewModifiers do not have var body so you do not need the SwiftUI Template when implementing on a single new file)

## Animation

Basics: \* Important takeaways about Animation - Only changes can be animated. Changes to what? - arguments to ViewModifiers

- arguments to the creation of Shapes - the existence (or not) of a View in the UI

\* Animation is showing the user changes that have already happened (i.e. the recent past) - Our code does something, makes a change, and only then SwiftUI triggers the animation

\* ViewModifiers are the primary change agents in the UI

- It is important to understand that: - A change to a ViewModifier's arguments has to happen after the View is initially put in the UI - In other words: only changes in a ViewModifier's arguments since it joined the UI are animated. -

Not all ViewModifier's arguments are animatable (e.g. .font is not), but most are. - When a View arrives or departs, the entire thing is animated as a unit:

- A View coming on-screen is only animated if it is joining a container that is already in the UI. - A View going off-screen is only animated if it is leaving a container that is staying in the UI. -

ForEach and if-else in ViewBuilders are common ways to make Views come and go

How do we make an animation "go"? 1. Implicitly (automatically), by using the view modifier .animation(Animation) 2. Explicitly, by wrapping with Animation(Animation) { } around code that might change things 3. Independently, By making Views be included or excluded from the UI

Again, all of the above only cause animations to "go" if the View is already part of the UI (or if the View is joining a container that is already part of the UI)

## Implicit Animation

- Automatic animation, essentially marks a View so that ...

- All ViewModifier arguments that precede the animation modifier will always be animated
- the changes are animated with the duration and “curve” you specify

To create a simple implicit animation add a `.animation(Animation)` view modifier to the View you want to auto-animate.

```
Text(" ")
    .opacity(scary ? 1 : 0)
    .rotationEffect(Angle.degrees(upsideDown ? 180 : 0))
    .animation(Animation.easeInOut)
```

Now whenever `scary` or `upsideDown` changes, the opacity/rotation will be animated. All changes to arguments to animatable view modifiers preceding `.animation` are animated. If we put something after `.animation()` those will not be animated (since animation is a view modifier) Without `.animation()`, the changes to opacity/rotation would appear instantly on screen.

Warning! The `.animation` modifier does not work well on a container. A container just propagates the `.animation` modifier to all the Views it contains. In other words, `.animation` does not work not like `.padding`, it works more like `.font`. It is good for single view like `Text`, `Image` etc.

```
//
// ImplicitAnimation.swift
// Memorize
//
// Created by nick88msn on 08/06/21.
//

import SwiftUI

struct AnimationView: View {
    @State var isScary: Bool = true
    @State var isUpsideDown: Bool = false

    var body: some View {
        VStack{
            Spacer()
            Text(" ")
                .font(.largeTitle)
                .opacity(isScary ? 1 : 0)
                .rotationEffect(Angle.degrees(isUpsideDown ? 180 : 0))
                .animation(.easeInOut)
            Button(action: {
                isScary.toggle()
                isUpsideDown.toggle()
            }, label: {
```



```

        Text(isScary ? "Press me" : "Bring me back")
    })
    .padding()
    .font(.title)
    .foregroundColor(isScary ? .primary : .secondary)
    Spacer()
}
}
}

```

```

struct AnimationView_Previews: PreviewProvider {
    static var previews: some View {
        return AnimationView()
            .previewDevice("iPhone 12 Pro")
            .preferredColorScheme(.dark)
    }
}

```

The argument of the `.animation()` ViewModifier is an Animation struct. It lets you control things about an animation such as: - its duration - its delay - whether the animation repeat a bunch of times or even loops endlessly in `repeatForever` - its curve to control the rate at which the animation plays out: - `.linear`, with a consistent rate throughout - `.easeInOut`, starts out the animation slowly, picks up speed, then slows at the end - `.spring`, provides soft landing (bounce) for the end of the animation

### Implicit vs Explicit Animation

Implicit animation are usually not the primary source of animation behavior. They are mostly used on “leaf” (i.e. non container like Text, Image etc) Views. Or, more generally, on Views that are typically working independently of other Views.

A more common cause of animation is a change in our Model. Or more generally, changes in response to some user action. For these changes, we want a whole bunch of Views to animate together. For that, we use Explicit Animation.

### Explicit Animation

Explicit Animation create an animation transaction during which all eligible changes made as a result of executing a block of code will be animated together. It is very easy to create one, you supply the Animation (duration, curve etc) to use and the block of code.

```

withAnimation(.linear(duration:2)){
    // do something that will cause ViewModifier/Shape arguments to change somewhere
}

```

Explicit animations are almost always wrapped around calls to ViewModel Intent functions. But they are also wrapped around things that only change then UI like “entering editing mode.”. It is fairly rare for code that handles a user gesture to not be wrapped in a withAnimation.

Explicit Animations do not override an implicit animation. Those implicit animation are kind of independent and are not affected by an explicit animation.

## Transitions

Transitions specify how to animate the arrival/departure of Views. It only works for Views that are inside Containers that are already On-Screen. Under the covers, a transition is nothing more than a pair of ViewModifiers, One of the modifiers is the “before” modification of the View that’s on the move. The other modifier is the “after” modification of the View on the move. Thus a transition is just a version of a “changes in arguments to ViewModifiers” animation.

It is possible to have an asymmetric transition (two different transition for in and out). An asym transition has 2 pairs of ViewModifiers. One pair for when the View appears (insertion), and another pair for when the View disappears (removal).

Example: A view fades in when it appears, but then flies across the screen when it disappears. Mostly we use “pre-canned” transitions (opacity, scaling, moving across the screen). They are static var/funs on the AnyTransition struct.

All the Transition API is ‘type erased’. We use the struct AnyTransition which erases type info for the underlying ViewModifiers. This makes a lot easier to work with transitions.

Some of the built-in transitions: - AnyTransition.opacity (uses .opacity modifier to fade the View in and out) - AnyTransition.scale (uses .frame modifier to expand/shrink the View as it comes and goes) - AnyTransition.offset(CGSize) (use .offset modifier to move the View as it comes and goes) - AnyTransition.modifier(active:identity:) (you provide the two ViewModifiers to use)

## How to apply transitions

How do we specify which kind of transition to use when a View arrive/departs? Using .transition(). Example using two built-in transitions, .scale and .identity:

```
ZStack {
    if isFacedUp{
        RoundedRectangle(cornerRadius:10).stroke()
        Text(" ").transition(AnyTransition.scale)
    } else {
        RoundedRectangle(cornerRadius: 10).transition(AnyTransition.identity)
    }
}
```

If `isFacedUp` changed from `false` to `true` (and `ZStack` was already on screen and we were explicitly animating) the back would disappear instantly, Text would grow in from nothing, front `RondedRectangle` would fade in.

Unlike `.animation()`, `.transition()` does not get redistributed to a container's content Views. So putting `.transition()` on the `ZStack` above only works if the entire `ZStack` came/went (`Group` and `ForEach` do distribute `.transition()` to their content Views, however).

`.transition()` is just specifying what the `ViewModifiers` are. It does not cause any animation to occur. In other words, think of the word `transition` as a noun here, not a verb. You are declaring what transition to use, not causing the transition to occur. Most probably you are going to trigger the transition with an explicit animation.

### Setting Animation Details for a Transition

You can set an animation (curve/duration/etc.) to use for a transition. Any-`Transition` structs have a `.animation(Animation)` of their own you can call. This sets the `Animation` parameters to use to animate the transition. e.g. `.transition(AnyTransition.opacity.animation(.linear(duration: 20)))`

### Matched Geometry Effect

Sometimes you want a View to move from one place on the screen to another (and possibly resize along the way). If the View is moving to a new place in its same container, this is no problem. “Moving” like this is just animating the `.position` `ViewModifier`'s (`.position` is what `HStack`, `LazyVGrid`, etc., use to position the Views inside them). This kind of thing happens automatically when you explicitly animate.

But what if the View is “moving” from one container to a different container? This is not really possible. Instead, you need a View in the “Source” position and a different one in the “destination” position. And then, you must “match” their geometries up as one leaves the UI and the other arriver. So this is similar to `.transition` in that it is animating Views' coming and going in the UI. It's just that it's particular to the case where a pair of Views' arrivals/departures are synced.

In the memorize game a great example of this would be “dealing cards off of a deck”. The “deck” might well be its own View off to the side. When a card is “dealt” from the deck, it needs to fly from there to the game. But the deck and game's main View are not in the same `LazyVGrid` or anything. How do we handle this? We mark both Views using this `ViewModifier`: - `.matchedGeometryEffect(id: ID, in: Namespace)` // ID type is a generics: Hashable - ID is the Identifier for our view - Namespace is a token, we create this token in our view with: `Swift @Namespace private var myNamespace` - We use namespace when we have more than one geometry effect on the same id

To write all up, we need that only one of the two is ever included in the UI at the same time. We can do this with if-else in a `ViewBuilder` or maybe via `ForEach`. Now, when one of the pair leaves and the other arrives at the same time, their size and position will be synced up and animated.

### **.onAppear**

Since that animations only work on Views that are in Containers that are already on screen, how can we kick off an animation as asoon as a View's Container arrives on screen?

View has a nice function called `.onAppear {}`

It executes a closure any time a View appears on screen (there's also `.onDisappear {}`)

User `.onAppear {}` on your container view to cause a change (usually in Model/ViewModel) that results in the appearance/animation of the View you want to be animated. Since by definition, your container is on-screen when its own `.onAppear {}` is happening, so any animations for its children that are appearing can fire. Of course, we need to use `withAnimation` inside `.onAppear {}`.

### **Shapes and ViewModifier Animation**

All actual animation happens in Shapes and ViewModifiers (even transitions and `matchedGeometryEffects` are just paired ViewModifiers). So how do they actually do their animation?

1. Essentially, the animation system divides the animation's duration up into little pieces (along whatever "curve" the animation uses, e.g. `.linear`, `.easeInOut`, `.spring`, etc.)
2. A shape or ViewModifier lets the animation system know what information it wants piece-ified (e.g. our Pie Shape is going to want to divide the Angles of the pie up into pieces)
3. During animation, the system tells the Shape/ViewModifier the current piece it should show. The Shape/ViewModifier makes sure its body draws appropriately at any "piece" value.
4. The communication with the animation system happens (both ways) with a single var. This var is the only thing in the `Animatable` protocol. Shapes and ViewModifiers that want to be animatable must implement this protocol.

**var animatableData: Type**

Type is a generics that has to implement the protocol `VectorArithmetic`. That's because it has to be able to be broken up into little pieces on an animation curve.

Type is very often a floating point number (FLoat, Double, CGFloat). But there is another struct that implements VectorArithmetic called AnimatablePair. AnimatablePair combines two VectorArithmetics into one VectorArithmetic.

Of course you can have AnimatablePairs of AnimatablePairs, so you can animate all you want. Beacuse it's communicating both ways, this animatableData is a read-write var. - The setting of this var is the animation system telling the Shape/VM which "piece" to draw - The getting of this var is the animation system getting the start/end points of an animation.

Usually this is a computed var (though it does not ahve to be). We might well not want to use the name "animatableData" in our Shape/VM code (using variable names that are more descriptive of what that data is to us). So the get/set very often just gets/sets some other var(s) exposing them to the animation system with a different name.