

Today

- Animation

 - How does it work?

- ViewModifier

 - What exactly are functions like `foregroundColor`, `font`, `padding`, etc. doing?

- Animation

 - Implicit vs. Explicit Animation

 - Animating Views (via their ViewModifiers which can implement the Animatable protocol)

 - Transitions (animating the appearance/disappearance of Views by specifying ViewModifiers)

 - Matched Geometry Effect

 - Animating Shapes (via the Animatable protocol)



Animation

- Animation

Animation is very important in a mobile UI.
That's why SwiftUI makes it so easy to do.



Animation

• Animation

Animation is very important in a mobile UI.

That's why SwiftUI makes it so easy to do.

One way to do animation is by animating a **Shape**.

We'll show you this a bit later when we get our Pie moving.



Animation

• Animation

Animation is very important in a mobile UI.

That's why SwiftUI makes it so easy to do.

One way to do animation is by animating a **Shape**.

We'll show you this a bit later when we get our Pie moving.

The other way to do animation is to animate Views via their **ViewModifiers**.

So what's a **ViewModifier**?



ViewModifier

- ViewModifier

You know all those little functions that modified our Views (like `aspectRatio` and `padding`)? They are (likely) turning right around and calling a function in View called `modifier`.



ViewModifier

• ViewModifier

You know all those little functions that modified our Views (like `aspectRatio` and `padding`)? They are (likely) turning right around and calling a function in View called `modifier`.

e.g. `.aspectRatio(2/3)` is likely something like `.modifier(AspectModifier(2/3))`

`AspectModifier` can be anything that conforms to the `ViewModifier` protocol ...



ViewModifier

- ViewModifier

The `ViewModifier` protocol has one function in it.

This function's only job is to create a new View based on the thing passed to it.



ViewModifier

• ViewModifier

The `ViewModifier` protocol has one function in it.

This function's only job is to create a new View based on the thing passed to it.

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  
    }  
}
```

ViewModifier

• ViewModifier

The `ViewModifier` protocol has one function in it.

This function's only job is to create a new View based on the thing passed to it.

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  
    }  
}
```

When we call `.modifier` on a View, the `Content` passed to this function is `that View`.



ViewModifier

• ViewModifier

The `ViewModifier` protocol has one function in it.

This function's only job is to create a new View based on the thing passed to it.

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  
    }  
}
```

When we call `.modifier` on a View, the `Content` passed to this function is `that View`.

`ViewModifier` code looks a lot like View code (`func body(content:)` instead of `var body`).

That's because `ViewModifiers` are Views.

And writing the code for one is almost identical.



ViewModifier

• ViewModifier

The `ViewModifier` protocol has one function in it.

This function's only job is to create a new View based on the thing passed to it.

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  
    }  
}
```

When we call `.modifier` on a View, the `Content` passed to this function is `that View`.

`ViewModifier` code looks a lot like View code (`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.



ViewModifier

- ViewModifier

Probably best learned by example.



ViewModifier

• ViewModifier

Probably best learned by example.

Let's say we wanted to create a modifier that would "card-ify" another View.

In other words, it would take that View and put it on a card like in our Memorize game.

It would work with any View whatsoever (not just our `Text("🃏")`).

What would such a modifier look like?



ViewModifier

• Cardify ViewModifier

```
Text("X").modifier(Cardify(isFaceUp: true)) // eventually .cardify(isFaceUp: true)
```

```
struct Cardify: ViewModifier {  
    var isFaceUp: Bool  
    func body(content: Content) -> some View {  
        ZStack {  
            if isFaceUp {  
                RoundedRectangle(cornerRadius: 10).fill(Color.white)  
                RoundedRectangle(cornerRadius: 10).stroke()  
                content  
            } else {  
                RoundedRectangle(cornerRadius: 10)  
            }  
        }  
    }  
}
```



ViewModifier

• Cardify ViewModifier

```
Text("🀄").modifier(Cardify(isFaceUp: true)) // eventually .cardify(isFaceUp: true)
```

```
struct Cardify: ViewModifier {  
    var isFaceUp: Bool  
    func body(content: Content) -> some View {  
        ZStack {  
            if isFaceUp {  
                RoundedRectangle(cornerRadius: 10).fill(Color.white)  
                RoundedRectangle(cornerRadius: 10).stroke()  
                content  
            } else {  
                RoundedRectangle(cornerRadius: 10)  
            }  
        }  
    }  
}
```



ViewModifier

• Cardify ViewModifier

```
Text("X").modifier(Cardify(isFaceUp: true)) // eventually .cardify(isFaceUp: true)
```

```
struct Cardify: ViewModifier {  
  var isFaceUp: Bool  
  func body(content: Content) -> some View {  
    ZStack {  
      if isFaceUp {  
        RoundedRectangle(cornerRadius: 10).fill(Color.white)  
        RoundedRectangle(cornerRadius: 10).stroke()  
        content  
      } else {  
        RoundedRectangle(cornerRadius: 10)  
      }  
    }  
  }  
}
```



ViewModifier

• Cardify ViewModifier

```
Text("X").modifier(Cardify(isFaceUp: true)) // eventually .cardify(isFaceUp: true)
```

```
struct Cardify: ViewModifier {
```

```
    var isFaceUp: Bool
```

```
    func body(content: Content) -> some View {
```

```
        ZStack {
```

```
            if isFaceUp {
```

```
                RoundedRectangle(cornerRadius: 10).fill(Color.white)
```

```
                RoundedRectangle(cornerRadius: 10).stroke()
```

```
                content
```

```
            } else {
```

```
                RoundedRectangle(cornerRadius: 10)
```

```
            }
```

```
        }
```

```
    }
```

```
}
```

.modifier() returns a View
that displays **this**

ViewModifier

👁 ViewModifier

How do we get from ...

```
Text("🀄").modifier(Cardify(isFaceUp: true))
```

... to ...

```
Text("🀄").cardify(isFaceUp: true)
```

?



ViewModifier

• ViewModifier

How do we get from ...

```
Text("🀄").modifier(Cardify(isFaceUp: true))
```

... to ...

```
Text("🀄").cardify(isFaceUp: true)
```

?

Easy ...

```
extension View {  
    func cardify(isFaceUp: Bool) -> some View {  
        return self.modifier(Cardify(isFaceUp: isFaceUp))  
    }  
}
```



Demo

- Cardify

Let's go implement this in our Memorize application!



Animation

• Important takeaways about Animation

Only changes can be animated. Changes to what?

ViewModifier arguments

Shapes

The “existence” (or not) of a View in the UI



Animation

• Important takeaways about Animation

Only changes can be animated. Changes to what?

ViewModifier arguments

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).



Animation

• Important takeaways about Animation

Only changes can be animated. Changes to what?

ViewModifier arguments

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).

ViewModifiers are the primary “change agents” in the UI.



Animation

• Important takeaways about Animation

Only changes can be animated. Changes to what?

- ViewModifier arguments

- 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).

ViewModifiers are the primary “change agents” in the UI.

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 arguments are animatable (e.g. .font’s are not), but most are.



Animation

• Important takeaways about Animation

Only changes can be animated. Changes to what?

ViewModifier arguments

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).

ViewModifiers are the primary “change agents” in the UI.

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 arguments are animatable (e.g. .font’s are 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’s joining a container that is already in the UI.

A View going off-screen is only animated if it’s leaving a container that is staying in the UI.

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



Animation

- How do you make an animation “go”?

Three ways ...



Animation

- How do you make an animation “go”?

Three ways ...

Implicitly (automatically), by using the view modifier `.animation(Animation)`.



Animation

- How do you make an animation “go”?

Three ways ...

Implicitly (automatically), by using the view modifier `.animation(Animation)`.

Explicitly, by wrapping `withAnimation(Animation) { }` around code that might change things.



Animation

- How do you make an animation “go”?

Three ways ...

Implicitly (automatically), by using the view modifier `.animation(Animation)`.

Explicitly, by wrapping `withAnimation(Animation) { }` around code that might change things.

By making Views be included or excluded from the UI.



Animation

• How do you make an animation “go”?

Three ways ...

Implicitly (automatically), by using the view modifier `.animation(Animation)`.

Explicitly, by wrapping `withAnimation(Animation) { }` around code that might change things.

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)



Animation

• 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 (next slide).



Animation

👁 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 (next slide).

Simply 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.

Without **.animation()**, the changes to opacity/rotation would appear instantly on screen.



Animation

• 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 (next slide).

Simply 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.

Without `.animation()`, the changes to opacity/rotation would appear instantly on screen.

Warning! The `.animation` modifier does not work how you might think 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`.



Animation

- Animation

The argument to `.animation()` is an **Animation** struct.
It lets you control things about an animation ...



Animation

• Animation

The argument to `.animation()` is an `Animation` struct.

It lets you control things about an animation ...

Its `duration`.



Animation

• Animation

The argument to `.animation()` is an `Animation` struct.

It lets you control things about an animation ...

Its `duration`.

Whether to `delay` a little bit before starting it.



Animation

• Animation

The argument to `.animation()` is an `Animation` struct.

It lets you control things about an animation ...

Its `duration`.

Whether to `delay` a little bit before starting it.

Whether it should `repeat` (a certain number of times or even `repeatForever`).



Animation

• Animation

The argument to `.animation()` is an `Animation` struct.

It lets you control things about an animation ...

Its `duration`.

Whether to `delay` a little bit before starting it.

Whether it should `repeat` (a certain number of times or even `repeatForever`).

Its "curve" ...

• Animation Curve

The kind of animation controls the rate at which the animation "plays out" (it's "curve") ...

`.linear` This means exactly what it sounds like: consistent rate throughout.

`.easeInOut` Starts out the animation slowly, picks up speed, then slows at the end.

`.spring` Provides "soft landing" (a "bounce") for the end of the animation.



Animation

- **Implicit vs. Explicit Animation**

These “automatic” implicit animations are usually not the primary source of animation behavior.

They are mostly used on “leaf” (i.e. non-container, aka “Lego brick”) Views.

Or, more generally, on Views that are typically working independently of other Views.



Animation

• Implicit vs. Explicit Animation

These “automatic” implicit animations are usually not the primary source of animation behavior.

They are mostly used on “leaf” (i.e. non-container, aka “Lego brick”) Views.

Or, more generally, on Views that are typically working independently of other Views.

A likely more common cause of animations is a change in your 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 ...



Animation

- **Explicit Animation**

Explicit animations create an animation transaction during which ...

All eligible changes made as a result of executing a block of code will be animated together.



Animation

• Explicit Animation

Explicit animations create an animation transaction during which ...

All eligible changes made as a result of executing a block of code will be animated together.

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  
}
```



Animation

• Explicit Animation

Explicit animations create an animation transaction during which ...

All eligible changes made as a result of executing a block of code will be animated together.

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 the UI like "entering editing mode."

It's fairly rare for code that handles a user gesture to not be wrapped in a withAnimation.



Animation

• Explicit Animation

Explicit animations create an animation transaction during which ...

All eligible changes made as a result of executing a block of code will be animated together.

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 the UI like "entering editing mode."

It's fairly rare for code that handles a user gesture to not be wrapped in a withAnimation.

Explicit animations do not override an implicit animation.



Animation

• Transitions

Transitions specify how to animate the arrival/departure of Views.

Only works for Views that are inside **CTAOS**.

(**C**ontainers **T**hat **A**re **A**lready **O**n-**S**creen.)



Animation

• Transitions

Transitions specify how to animate the arrival/departure of Views.

Only works for Views that are inside **CTAOS**.

(**C**ontainers **T**hat **A**re **A**lready **O**n-**S**creen.)

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 that’s on the move.

Thus a transition is just a version of a “changes in arguments to ViewModifiers” animation.



Animation

• Transitions

Transitions specify how to animate the arrival/departure of Views.

Only works for Views that are inside **CTAAOS**.

(**C**ontainers **T**hat **A**re **A**lready **O**n-**S**creen.)

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 that’s on the move.

Thus a transition is just a version of a “changes in arguments to ViewModifiers” animation.

An asymmetric transition has 2 pairs of ViewModifiers.

One pair for when the View appears (insertion)

And another pair for when the View disappears (removal)



Animation

• Transitions

Transitions specify how to animate the arrival/departure of Views.

Only works for Views that are inside **CTAOS**.

(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 that’s on the move.

Thus a transition is just a version of a “changes in arguments to ViewModifiers” animation.

An asymmetric 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 vars/funcs on the **AnyTransition** struct.



Animation

• Transitions

All the transition API is “type erased”.

We use the struct `AnyTransition` which erases type info for the underlying `ViewModifiers`.

This makes it a lot easier to work with transitions.



Animation

• Transitions

All the transition API is “type erased”.

We use the struct `AnyTransition` which erases type info for the underlying `ViewModifiers`. This makes it a lot easier to work with transitions.

For example, here are 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)



Animation

- Transitions

How do we specify which kind of transition to use when a View arrives/departs?



Animation

• Transitions

How do we specify which kind of transition to use when a View arrives/departs?

Using `.transition()`. Example using two built-in transitions, `.scale` and `.identity` ...

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

If `isFaceUp` 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 RR would fade in.



Animation

• Transitions

How do we specify which kind of transition to use when a View arrives/departs?

Using `.transition()`. Example using two built-in transitions, `.scale` and `.identity` ...

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

If `isFaceUp` 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 RR 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.)



Animation

• Transitions

`.transition()` is just specifying what the ViewModifiers are.

It doesn't 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.



Animation

• Transitions

`.transition()` is just specifying what the ViewModifiers are.

It doesn't 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.

• Setting Animation Details for a Transition

You can set an animation (curve/duration/etc.) to use for a transition.

AnyTransition 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)))`

Animation

• Matched Geometry Effect

Sometimes you want a View to move from one place on 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 arguments.

(`.position` is what HStack, LazyVGrid, etc., use to position the Views inside them.)

This kind of thing happens automatically when you explicitly animate.



Animation

• Matched Geometry Effect

Sometimes you want a View to move from one place on 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 arguments.

(`.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.



Animation

• Matched Geometry Effect

Sometimes you want a View to move from one place on 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 arguments.

(`.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 arrives.

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.



Animation

• Matched Geometry Effect

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?



Animation

• Matched Geometry Effect

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 “don’t care”: Hashable
```



Animation

• Matched Geometry Effect

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 “don’t care”: Hashable
```

Declare the Namespace as a private var in your View like this ..

```
@Namespace private var myNamespace
```



Animation

• Matched Geometry Effect

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 “don’t care”: Hashable
```

Declare the Namespace as a private var in your View like this ..

```
@Namespace private var myNamespace
```

Now write your code so that only one of the two is ever included in the UI at the same time.

You 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.



Animation

- `.onAppear`

Remember that animations only work on Views that are in **CTAOS**.
(**C**ontainers **T**hat **A**re **A**lready **O**n-**S**creen.)

How can you kick off an animation as soon as a View's **C**ontainer arrives **O**n-**S**creen?



Animation

• .onAppear

Remember that animations only work on Views that are in **CTAOS**.
(**C**ontainers **T**hat **A**re **A**lready **O**n-**S**creen.)

How can you kick off an animation as soon as a View's **C**ontainer arrives **O**n-**S**creen?

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

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



Animation

• .onAppear

Remember that animations only work on Views that are in **CTAOS**.
(Containers That Are Already On-Screen.)

How can you kick off an animation as soon 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 { }**).

Use **.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, it is a **CTAOS**, so any animations for its children that are appearing can fire.

Of course, you'd need to use **withAnimation** inside **.onAppear { }**.



Animation

• .onAppear

Remember that animations only work on Views that are in **CTAAOS**.
(Containers That Are Already On-Screen.)

How can you kick off an animation as soon 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 { }**).

Use **.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, it is a **CTAAOS**, so any animations for its children that are appearing can fire.

Of course, you'd need to use **withAnimation** inside **.onAppear { }**.

We'll use **.onAppear { }** to kick off a couple of animations in demo this week.

Especially ones that only make sense when a certain View is visible (e.g. our Pie).



Animation

- Shape 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?



Animation

- Shape 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?

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.)



Animation

- Shape 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?

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.)

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.)



Animation

• Shape 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?

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.)

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.)

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.



Animation

- Shape and ViewModifier Animation

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
```



Animation

- Shape and ViewModifier Animation

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 don't care.

Well ... it's a "care a little bit."

Type 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.



Animation

- Shape and ViewModifier Animation

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 don't care.

Well ... it's a "care a little bit."

Type 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's 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.



Animation

- Shape and ViewModifier Animation

Because 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.



Animation

• Shape and ViewModifier Animation

Because 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 have to be).

We might well not want to use the name "animatableData" in our Shape/VM code

(we want to use 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)

(essentially exposing them to the animation system with a different name).

In the demo, we'll see doing this both for our Pie Shape and our Cardify ViewModifier.



Demo

• Implicit Animation

We'll cover only implicit animation in today's demo.

This is one of the less important kinds of animation (even though we're covering it first).

We'll make our emoji "spin" on matching cards.

And see what it's like to try to diagnose the problem when an animation fails to fire.

• Next Lecture

We'll do a ton of animation stuff next time ...

Explicit animations of our ViewModel Intent (et. al.) functions.

Many geometry effects (rotation, scale, even 3D rotation).

Dealing cards off of a deck of cards (matchedGeometryEffect).

Delaying animations.

Using @State to control animations.

Transition matched cards out by making them shrink down to nothingness.

Kicking off animations when a container appears (onAppear).

Animate Cardify and Pie.

