

Property Observers

• Property Observers

Remember that Swift is able to “detect” when a struct changes?

Well, you can take action when it notices this if you like.

Property observers are essentially a way to “watch” a var and execute code when it changes.



Property Observers

• Property Observers

Remember that Swift is able to “detect” when a struct changes?

Well, you can take action when it notices this if you like.

Property observers are essentially a way to “watch” a var and execute code when it changes.

The syntax can look a lot like a computed var, but it is completely unrelated to that.

```
var isFaceUp: Bool {  
    willSet {  
        if newValue {  
            startUsingBonusTime()  
        } else {  
            stopUsingBonusTime()  
        }  
    }  
}
```

Inside here, `newValue` is a special variable (the value it's going to get set to).



Property Observers

• Property Observers

Remember that Swift is able to “detect” when a struct changes?

Well, you can take action when it notices this if you like.

Property observers are essentially a way to “watch” a var and execute code when it changes.

The syntax can look a lot like a computed var, but it is completely unrelated to that.

```
var isFaceUp: Bool {  
    willSet {  
        if newValue {  
            startUsingBonusTime()  
        } else {  
            stopUsingBonusTime()  
        }  
    }  
}
```

Inside here, `newValue` is a special variable (the value it's going to get set to).

There's also a `didSet` (inside that one, `oldValue` is what the value used to be).



Layout

- How is the space on-screen apportioned to the Views?



Layout

- How is the space on-screen apportioned to the Views?

It's amazingly simple ...

1. Container Views "offer" space to the Views inside them



Layout

- How is the space on-screen apportioned to the Views?

It's amazingly simple ...

1. Container Views "offer" space to the Views inside them
2. Views then choose what size they want to be
3. Container Views then position the Views inside of them



Layout

- How is the space on-screen apportioned to the Views?

It's amazingly simple ...

1. Container Views "offer" space to the Views inside them
2. Views then choose what size they want to be
3. Container Views then position the Views inside of them
4. (and based on that, Container Views choose their own size as per #2 above)



Layout

- HStack and VStack

Stacks divide up the space that is offered to them and then offer that to the Views inside. It offers space to its “least flexible” (with respect to sizing) subviews first ...



Layout

- HStack and VStack

Stacks divide up the space that is offered to them and then offer that to the Views inside. It offers space to its “least flexible” (with respect to sizing) subviews first ...

Example of an “inflexible” View: **Image** (it wants to be a fixed size).

Another example (slightly more flexible): **Text** (always wants to size to exactly fit its text).

Example of a very flexible View: **RoundedRectangle** (always uses any space offered).



Layout

- HStack and VStack

Stacks divide up the space that is offered to them and then offer that to the Views inside. It offers space to its “least flexible” (with respect to sizing) subviews first ...

Example of an “inflexible” View: **Image** (it wants to be a fixed size).

Another example (slightly more flexible): **Text** (always wants to size to exactly fit its text).

Example of a very flexible View: **RoundedRectangle** (always uses any space offered).

After an offered View(s) takes what it wants, its size is removed from the space available.

Then the stack moves on to the next “least flexible” Views.

Very flexible views (i.e. those that will take all offered space) will share evenly (mostly).

Rinse and repeat.



Layout

- HStack and VStack

Stacks divide up the space that is offered to them and then offer that to the Views inside. It offers space to its “least flexible” (with respect to sizing) subviews first ...

Example of an “inflexible” View: **Image** (it wants to be a fixed size).

Another example (slightly more flexible): **Text** (always wants to size to exactly fit its text).

Example of a very flexible View: **RoundedRectangle** (always uses any space offered).

After an offered View(s) takes what it wants, its size is removed from the space available.

Then the stack moves on to the next “least flexible” Views.

Very flexible views (i.e. those that will take all offered space) will share evenly (mostly).

Rinse and repeat.

After the Views inside the stack choose their own sizes, the stack sizes itself to fit them.

If any of the Views in the stack are “very flexible”, then the stack will also be “very flexible.”



Layout

- HStack and VStack

There are a couple of really valuable Views for layout that are commonly put in stacks ...



Layout

- HStack and VStack

There are a couple of really valuable Views for layout that are commonly put in stacks ...

`Spacer(minLength: CGFloat)`

Always takes all the space offered to it.

Draws nothing.

The `minLength` defaults to the most likely spacing you'd want on a given platform.



Layout

- HStack and VStack

There are a couple of really valuable Views for layout that are commonly put in stacks ...

Spacer(minLength: CGFloat)

Always takes all the space offered to it.

Draws nothing.

The minLength defaults to the most likely spacing you'd want on a given platform.

Divider()

Draws a dividing line cross-wise to the way the stack is laying out.

For example, in an HStack, **Divider** draws a vertical line.

Takes the minimum space needed to fit the line in the direction the stack is going.



Layout

- HStack and VStack

Stack's choice of who to offer space to next can be overridden with `.layoutPriority(Double)`.
In other words, `layoutPriority` trumps "least flexible".



Layout

- HStack and VStack

Stack's choice of who to offer space to next can be overridden with `.layoutPriority(Double)`. In other words, `layoutPriority` trumps "least flexible".

```
HStack {  
    Text("Important").layoutPriority(100) // any floating point number is okay  
    Image(systemName: "arrow.up") // the default layout priority is 0  
    Text("Unimportant")  
}
```

The Important Text above will get the space it wants first. Then the Image would get its space (since it's less flexible than the Unimportant Text). Finally, Unimportant would have to try to fit itself into any remaining space. If a Text doesn't get enough space, it will elide (e.g. "Swift is..." instead of "Swift is great!").



Layout

- HStack and VStack

Another crucial aspect of the way stacks lay out the Views they contain is alignment.



Layout

- HStack and VStack

Another crucial aspect of the way stacks lay out the Views they contain is alignment.

When a VStack lays Views out in a column, what if the Views are not all the same width?
Does it “left align” them? Or center them? Or what?



Layout

- HStack and VStack

Another crucial aspect of the way stacks lay out the Views they contain is alignment.

When a VStack lays Views out in a column, what if the Views are not all the same width? Does it “left align” them? Or center them? Or what?

This is specified via an argument to the stack ...

```
VStack(alignment: .leading) { . . . }
```



Layout

- HStack and VStack

Another crucial aspect of the way stacks lay out the Views they contain is alignment.

When a VStack lays Views out in a column, what if the Views are not all the same width? Does it “left align” them? Or center them? Or what?

This is specified via an argument to the stack ...

```
VStack(alignment: .leading) { . . . }
```

Why .leading instead of .left?

Stacks automatically adjust for environments where text is right-to-left (e.g. Arabic or Hebrew). The .leading alignment lines the things in the VStack up to the edge where text starts from.



Layout

- HStack and VStack

Another crucial aspect of the way stacks lay out the Views they contain is alignment.

When a VStack lays Views out in a column, what if the Views are not all the same width? Does it “left align” them? Or center them? Or what?

This is specified via an argument to the stack ...

```
VStack(alignment: .leading) { . . . }
```

Why .leading instead of .left?

Stacks automatically adjust for environments where text is right-to-left (e.g. Arabic or Hebrew). The .leading alignment lines the things in the VStack up to the edge where text starts from.

Text baselines can also be used to align (e.g. `HStack(alignment: .firstTextBaseline) { }`).



Layout

- HStack and VStack

Another crucial aspect of the way stacks lay out the Views they contain is alignment.

When a VStack lays Views out in a column, what if the Views are not all the same width? Does it “left align” them? Or center them? Or what?

This is specified via an argument to the stack ...

```
VStack(alignment: .leading) { . . . }
```

Why .leading instead of .left?

Stacks automatically adjust for environments where text is right-to-left (e.g. Arabic or Hebrew). The .leading alignment lines the things in the VStack up to the edge where text starts from.

Text baselines can also be used to align (e.g. `HStack(alignment: .firstTextBaseline) { }`).

You can even define your own “things to line up” alignment guides.

But that’s a bit beyond the scope of this course.

So we’re just going to use the built-ins (text baselines, .center, .top, .trailing, etc.).



Layout

- **LazyHStack and LazyVStack**

These “lazy” versions of the stack don’t build any of their Views that are not visible.

They also size themselves to fit their Views.

So they don’t take up all the space offered to them even if they have flexible views inside.

You’d use these when you have a stack that is in a ScrollView.



Layout

- LazyHStack and LazyVStack

These “lazy” versions of the stack don’t build any of their Views that are not visible.

They also size themselves to fit their Views.

So they don’t take up all the space offered to them even if they have flexible views inside.

You’d use these when you have a stack that is in a ScrollView.

- ScrollView

ScrollView takes all the space offered to it.

The views inside it are sized to fit along the axis your scrolling on.



Layout

- **LazyHStack and LazyVStack**

These “lazy” versions of the stack don’t build any of their Views that are not visible.

They also size themselves to fit their Views.

So they don’t take up all the space offered to them even if they have flexible views inside.

You’d use these when you have a stack that is in a ScrollView.

- **ScrollView**

ScrollView takes all the space offered to it.

The views inside it are sized to fit along the axis your scrolling on.

- **LazyHGrid and LazyVGrid**

We’ve already seen how these lay out their Views.



Layout

- **LazyHStack and LazyVStack**

These “lazy” versions of the stack don’t build any of their Views that are not visible.

They also size themselves to fit their Views.

So they don’t take up all the space offered to them even if they have flexible views inside.

You’d use these when you have a stack that is in a ScrollView.

- **ScrollView**

ScrollView takes all the space offered to it.

The views inside it are sized to fit along the axis your scrolling on.

- **LazyHGrid and LazyVGrid**

We’ve already seen how these lay out their Views.

- **List and Form and OutlineGroup**

These are sort of like “really smart VStacks”.

We’ll talk about them later in the quarter.



Layout

- ZStack

ZStack sizes itself to fit its children.

If even one of its children is fully flexible size, then the ZStack will be too.



Layout

- ZStack

ZStack sizes itself to fit its children.

If even one of its children is fully flexible size, then the ZStack will be too.

- .background modifier

```
Text("hello").background(Rectangle().foregroundColor(.red))
```

This is similar to making a ZStack of this Text and Rectangle (with the Text in front).

However, there's a big difference in layout between this and using a ZStack to stack them.

In this case, the resultant View will be **sized to the Text** (the Rectangle is not involved).

In other words, the Text solely determines the layout of this "mini-ZStack of two things".



Layout

- ZStack

ZStack sizes itself to fit its children.

If even one of its children is fully flexible size, then the ZStack will be too.

- .background modifier

```
Text("hello").background(Rectangle().foregroundColor(.red))
```

This is similar to making a ZStack of this Text and Rectangle (with the Text in front).

However, there's a big difference in layout between this and using a ZStack to stack them.

In this case, the resultant View will be **sized to the Text** (the Rectangle is not involved).

In other words, the Text solely determines the layout of this "mini-ZStack of two things".

- .overlay modifier

Same layout rules as .background, but stacked the other way around.

```
Circle().overlay(Text("Hello"), alignment: .center)
```

This will be **sized to the Circle** (i.e. it will be fully-flexible sized).

The Text will be stacked on top of the Circle (with the specified alignment inside the Circle).

Layout

• Modifiers

Remember that View modifier functions (like `.padding`) themselves return a View. That View, conceptually anyway, “contains” the View it’s modifying.



Layout

• Modifiers

Remember that View modifier functions (like `.padding`) themselves return a View.

That View, conceptually anyway, “contains” the View it’s modifying.

Many of them just pass the size offered to them along (like `.font` or `.foregroundColor`).

But it is possible for a modifier to be involved in the layout process itself.



Layout

• Modifiers

Remember that View modifier functions (like `.padding`) themselves return a View.

That View, conceptually anyway, “contains” the View it’s modifying.

Many of them just pass the size offered to them along (like `.font` or `.foregroundColor`).

But it is possible for a modifier to be involved in the layout process itself.

For example the View returned by `.padding(10)` will offer the View that it is modifying a space that is the same size as it was offered, but reduced by 10 points on each side.



Layout

• Modifiers

Remember that View modifier functions (like `.padding`) themselves return a View. That View, conceptually anyway, “contains” the View it’s modifying.

Many of them just pass the size offered to them along (like `.font` or `.foregroundColor`). But it is possible for a modifier to be involved in the layout process itself.

For example the View returned by `.padding(10)` will offer the View that it is modifying a space that is the same size as it was offered, but reduced by 10 points on each side. The View returned by `.padding(10)` would then choose a size for itself which is 10 points larger on all sides than the View it is modifying ended up choosing.



Layout

• Modifiers

Remember that View modifier functions (like `.padding`) themselves return a View. That View, conceptually anyway, “contains” the View it’s modifying.

Many of them just pass the size offered to them along (like `.font` or `.foregroundColor`). But it is possible for a modifier to be involved in the layout process itself.

For example the View returned by `.padding(10)` will offer the View that it is modifying a space that is the same size as it was offered, but reduced by 10 points on each side. The View returned by `.padding(10)` would then choose a size for itself which is 10 points larger on all sides than the View it is modifying ended up choosing.

Another example is a modifier we’ve already used: `.aspectRatio`.

The View returned by the `.aspectRatio` modifier takes the space offered to it and picks a size for itself that is either smaller (`.fit`) to respect the ratio or bigger (`.fill`) to use all the offered space (and more, potentially) and respect the ratio. (yes, a View is allowed to choose a size for itself that is larger than the space it was offered!) `.aspectRatio` then offers the space it chose to the View it is modifying (as its “container”).



Layout

• Example

```
HStack { // aside: the default alignment here is .center (not .top, for example)
  ForEach(viewModel.cards) { card in
    CardView(card: card).aspectRatio(2/3, contentMode: .fit)
  }
}

.foregroundColor(Color.orange)
.padding(10)
```



Layout

• Example

```
HStack { // aside: the default alignment here is .center (not .top, for example)
  ForEach(viewModel.cards) { card in
    CardView(card: card).aspectRatio(2/3, contentMode: .fit)
  }
}
.foregroundColor(Color.orange)
.padding(10)
```

The first View to be offered space here will be the View made by `.padding(10)`



Layout

• Example

```
HStack { // aside: the default alignment here is .center (not .top, for example)
  ForEach(viewModel.cards) { card in
    CardView(card: card).aspectRatio(2/3, contentMode: .fit)
  }
}
.foregroundColor(Color.orange)
.padding(10)
```

The first View to be offered space here will be the View made by `.padding(10)`
Which will offer what it was offered minus 10 on all sides to the View from `.foregroundColor`



Layout

• Example

```
HStack { // aside: the default alignment here is .center (not .top, for example)
  ForEach(viewModel.cards) { card in
    CardView(card: card).aspectRatio(2/3, contentMode: .fit)
  }
}
.foregroundColor(Color.orange)
.padding(10)
```

The first View to be offered space here will be the View made by `.padding(10)`
Which will offer what it was offered minus 10 on all sides to the View from `.foregroundColor`
Which will in turn offer all of that space to the HStack



Layout

• Example

```
HStack { // aside: the default alignment here is .center (not .top, for example)
  ForEach(viewModel.cards) { card in
    CardView(card: card).aspectRatio(2/3, contentMode: .fit)
  }
}
.foregroundColor(Color.orange)
.padding(10)
```

The first View to be offered space here will be the View made by `.padding(10)`

Which will offer what it was offered minus 10 on all sides to the View from `.foregroundColor`

Which will in turn offer all of that space to the HStack

Which will then divide its space equally among the `.aspectRatio` Views in the ForEach.



Layout

• Example

```
HStack { // aside: the default alignment here is .center (not .top, for example)
  ForEach(viewModel.cards) { card in
    CardView(card: card).aspectRatio(2/3, contentMode: .fit)
  }
}
.foregroundColor(Color.orange)
.padding(10)
```

The first View to be offered space here will be the View made by `.padding(10)`
Which will offer what it was offered minus 10 on all sides to the View from `.foregroundColor`
Which will in turn offer all of that space to the HStack
Which will then divide its space equally among the `.aspectRatio` Views in the ForEach.
Each `.aspectRatio` View will set its width to be its share of the HStack's width
and pick a height for itself that respects the requested 2/3 aspect ratio.
Or it might be forced to take all of the offered height and choose its width using the ratio.
(Whichever fits.)



Layout

• Example

```
HStack { // aside: the default alignment here is .center (not .top, for example)
  ForEach(viewModel.cards) { card in
    CardView(card: card).aspectRatio(2/3, contentMode: .fit)
  }
}

.foregroundColor(Color.orange)
.padding(10)
```

The first View to be offered space here will be the View made by `.padding(10)`

Which will offer what it was offered minus 10 on all sides to the View from `.foregroundColor`

Which will in turn offer all of that space to the HStack

Which will then divide its space equally among the `.aspectRatio` Views in the ForEach.

Each `.aspectRatio` View will set its width to be its share of the HStack's width

and pick a height for itself that respects the requested 2/3 aspect ratio.

Or it might be forced to take all of the offered height and choose its width using the ratio.

(Whichever fits.)

The `.aspectRatio` then offers all of its chosen size to its CardView, which will use it all.



Layout

• Example

```
HStack { // aside: the default alignment here is .center (not .top, for example)
  ForEach(viewModel.cards) { card in
    CardView(card: card).aspectRatio(2/3, contentMode: .fit)
  }
}
.foregroundColor(Color.orange)
.padding(10)
```

The size this whole View (i.e. the one returned from `.padding(10)`) chooses for itself will be ...



Layout

• Example

```
HStack { // aside: the default alignment here is .center (not .top, for example)
  ForEach(viewModel.cards) { card in
    CardView(card: card).aspectRatio(2/3, contentMode: .fit)
  }
}
.foregroundColor(Color.orange)
.padding(10)
```

The size this whole View (i.e. the one returned from `.padding(10)`) chooses for itself will be ...
The result of the `HStack` sizing itself to fit those `.aspectRatio` Views + 10 points on all sides.



Layout

- Views that take all the space offered to them

Most Views simply size themselves to take up all the space offered to them.

For example, Shapes usually draw themselves to fit (like `RoundedRectangle`).

Custom Views (like `CardView`) should do this too whenever sensible.

But they really should adapt themselves to any space offered to look as good as possible.

For example, `CardView` would want to pick a font size that makes its emoji fill the space.

We'll be doing that in the demo later today.



Layout

• Views that take all the space offered to them

Most Views simply size themselves to take up all the space offered to them.

For example, Shapes usually draw themselves to fit (like `RoundedRectangle`).

Custom Views (like `CardView`) should do this too whenever sensible.

But they really should adapt themselves to any space offered to look as good as possible.

For example, `CardView` would want to pick a font size that makes its emoji fill the space.

We'll be doing that in the demo later today.

So how does a View know what space was offered to it so it can try to adapt?

Using a special View called a `GeometryReader` ...



Layout

• GeometryReader

You wrap this `GeometryReader` View around what would normally appear in your View's body ...

```
var body: View {  
    GeometryReader { geometry in // using trailing closure syntax for content: parameter  
        . . .  
    }  
}
```



Layout

• GeometryReader

You wrap this `GeometryReader` View around what would normally appear in your View's body ...

```
var body: View {  
    GeometryReader { geometry in // using trailing closure syntax for content: parameter  
        . . .  
    }  
}
```

The `geometry` parameter is a `GeometryProxy`.

```
struct GeometryProxy {  
    var size: CGSize  
    func frame(in: CoordinateSpace) -> CGRect  
    var safeAreaInsets: EdgeInsets  
}
```

The `size` var is the amount of space that is being "offered" to us by our container.

Now we can, for example, pick a font size appropriate to that sized space.

`GeometryReader` itself (it's just a View) **always accepts all the space offered to it.**



Layout

- **Safe Area**

Generally, when a View is offered space, that space does not include “safe areas”.

The most obvious “safe area” is the notch on an iPhone X.

Surrounding Views might also introduce “safe areas” that Views inside shouldn’t draw in.



Layout

• Safe Area

Generally, when a View is offered space, that space does not include “safe areas”.

The most obvious “safe area” is the notch on an iPhone X.

Surrounding Views might also introduce “safe areas” that Views inside shouldn’t draw in.

But it is possible to ignore this and draw in those areas anyway on specified edges ...

```
ZStack { ... }.edgesIgnoringSafeArea([.top]) // draw in “safe area” on top edge
```

