

# Stanford CS193p

Developing Applications for iOS

Spring 2023

Lecture 6



CS193p  
Spring 2023



# 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

## 👁 Container Views

The "stacks" (`HStack`, `VStack`) divide up the space offered to them amongst their subviews

`ForEach` defers to its container to lay out the Views inside of it

Modifiers (e.g. `.padding()`) essentially "contain" the View they modify. Some do layout.





# 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 the 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.

Rinse and repeat.

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





# 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".

```
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.

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

## 👁️ Modifiers

Remember that View modifier functions (like `.padding`) themselves return a View. That View, for all intents and purposes, “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 it’s 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 you’ve already used (hopefully): `.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!) It then offers the space it chose to the View it is modifying (again, acting 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)
```

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 ...  
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.

In your homework you were asked to do this but it was a very poor solution.

We'll fix that in our demo 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 // not showing content: parameter label  
        . . .  
    }  
}
```

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.

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
```





# Layout

## 👁 Container

How exactly do container Views “offer” space to the Views they contain?

With the modifier `.frame(...)`.

This `.frame` modifier has a lot of arguments you can check out in the documentation.

Once a View chooses the size it wants from what’s offered, the container must then position it.

It does this with the `.position(CGPoint)` modifier.

The `position` is the center of the subview in the container’s coordinate space.

Stacks would use their alignment and spacing to figure this CGPoint out for each View.

You can also offset a View from where its container put it using `.offset(CGSize)` modifier.

For Memorize, we’re going to use frame/position to create our own “stack”-like Grid container.





# Back to the Demo

## 👁 Layout

Make CardView proportion itself to the size its “offered” by its container.

How do we deal with “magic numbers” in our code in Swift?

Next lecture we’ll lay out our cards in a grid instead of using an HStack.

