HACKING WITH SWIFT

PRO SWIFTUI

Unleash SwiftUI's full potential and build smarter, faster a

FREE SAMPLE

Paul Hudson

Chapter 1

Layout and Identity

Parents and children

At the core of SwiftUI is its three-step layout process:

1. A parent view proposes a size for its child.
2. Based on that information, the child then chooses its own size and the parent view must respect that choice.
3. The parent view then positions the child in its coordinate space.

It sounds so trivial, and you're probably wondering why I'm starting out by mentioning something that is so straightforward, but the simple truth is that this simple process unlocks a huge amount of power and the more you really understand it the more you'll be able to get SwiftUI doing exactly what you want.

The key to the power is answering a simple question: what is the "parent view" in that process? When you're learning SwiftUI, the answer seems obvious. For example:

```
VStack {
  Text("Hello, world!")
    .frame(width: 300, height: 100)

  Image(systemName: "person")
}
```

If I asked a learner what the parent of the text view was, they would probably answer "the **VStack**." And honestly that's a perfectly fine answer, and I wouldn't correct a beginner who said it – it feels natural, and it fits the hierarchy we can see when the code runs. However, it's also *wrong*, and once you have sufficient experience with SwiftUI it's important you understand why the answer is wrong, and more importantly what is *right*.

Let's start simple: when we use any modifier in SwiftUI, we are most of the time creating a new view that wraps the original view to add some extra behavior or styling. For example, this is just one view:

```
Text("Hello, world!")
```

Whereas this is two views:

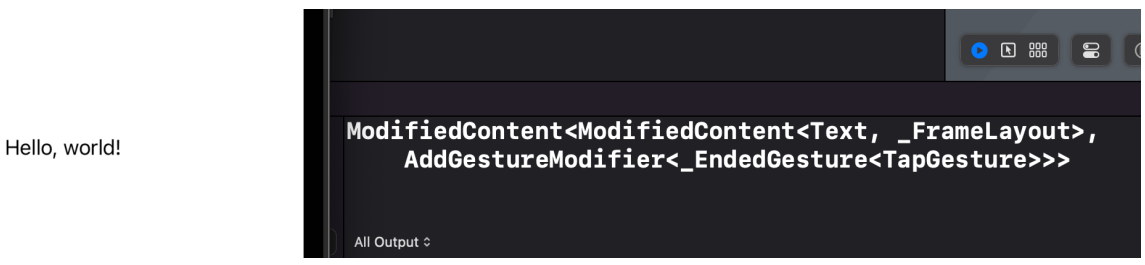
```
Text("Hello, world!")
  .frame(width: 300, height: 100)
```

This makes sense if you break it down and keep the three-step layout process in mind: the text view *can't* position itself, because that's the job of the parent. So, the only way for "Hello, world!" to be aligned center in a 300x100 container is for the parent – the frame – to be that 300x100 container. This is also why we can stack many modifiers to create more complex effects: we aren't modifying the original view again and again, but instead modifying a new view that modifies the original.

Once you understand this process of creating new views using modifiers, so much of the rest of SwiftUI makes sense. This is why I encourage repeatedly encourage folks to print out the underlying types of their views, like this:

```
Text("Hello, world!")
  .frame(width: 300, height: 100)
  .onTapGesture {
    print(type(of: self.body))
  }
```

When you do that, you'll see the **ModifiedContent** type appear a lot – not exactly *once* for every modifier, because again not all modifiers create new views. **ModifiedContent** is itself a struct that conforms to the **View** protocol, and I'd like you to look it up using Open Quickly.



If you aren't familiar with it, Open Quickly is an Xcode feature that lets you type to search through your code and also Apple's own APIs; activate it using Shift+Cmd+O, then type ModifiedContent and press return. This will open Xcode's generated interface file for SwiftUI, and you should be able to find this in there:

```
@frozen public struct ModifiedContent<Content, Modifier>
```

A little further down you'll also see its **View** conformance:

```
extension ModifiedContent : View where Content : View,
Modifier : ViewModifier {
```

These things aren't magic, and they aren't *secret* – you can use **ModifiedContent** directly if you want, because it's public API. For example, back in ContentView.swift we could create a custom a modifier like this one:

```
struct CustomFont: ViewModifier {
  func body(content: Content) -> some View {
    content.font(.largeTitle)
  }
}
```

We could apply that to some text using **ModifiedContent**, like so:

```
struct ContentView: View {
  var body: some View {
    ModifiedContent(content: Text("Hello"), modifier:
CustomFont())
  }
}
```

That's obviously a lot more wordy than the normal SwiftUI code we'd write, but what I want

you to understand is that the end result is *identical* to what we'd get by using **modifier()** like this:

```
Text("Hello")
  .modifier(CustomFont())
  .onTapGesture {
    print(type(of: self.body))
  }
```

This is what SwiftUI's result builder does for us: it repeatedly transforms our modifiers into **ModifiedContent** views, nesting them again and again to get exactly the right result. This is all done at compile time rather than run time: the actual underlying type of these two views are identical, rather than just the finished, rendering layouts.

So, when we write code like this:

```
Text("Hello, world!")
  .frame(width: 300, height: 100)
```

That creates the original text view, plus a new **ModifiedContent** around it that happens to contain the fixed frame instructions. The text still has its original frame – the original size it wants to work with – but now we've added a second frame around it.

You can see the original frame is alive and kicking by passing in a custom alignment for the outer frame:

```
Text("Hello, world!")
  .frame(width: 300, height: 100, alignment: .bottomTrailing)
```

If you think about it, the only way the text can be aligned to the bottom trailing edge is if it knows its original frame. So, our text view has its own default frame that exactly matches the natural size for its text, and *no amount of futzing from us can ever force that text to extend its bounds beyond the natural width and height of its lines*.

However, by applying the **frame()** modifier we're creating a new **ModifiedContent** view around the text that takes up more space – it's not strictly a "frame" view in its own right, but I think it's helpful to talk about it like that.

Fixing view sizes

Let's look at this simple code again:

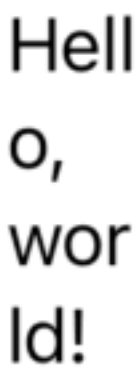
```
Text("Hello, world!")
  .frame(width: 300, height: 100)
```

Like I said, despite attaching a **frame()** modifier, no amount of futzing from us can ever force that text to extend its bounds beyond the natural width and height of its lines – that's just not how SwiftUI works.

Of course, the real question here is this: what *is* the natural size for the text? Well, the answer is that text likes to live on one long line, and that's exactly what it will do unless you ask for something else. For example, we might say that our frame had a width of only 30 rather than 300, like this:

```
Text("Hello, world!")
  .frame(width: 30, height: 100)
```

Now there isn't enough space for the text, it's forced to wrap across several lines to fit into the tiny box we've allocated to it.



On the surface this sounds like it breaks the second rule of Swift's layout system: "the child chooses its own size and the parent view must respect that choice." In this case the child is the text view and the "frame" view is its parent, so how come the text is being forced to accept the size of its parent, the frame?

What's really happening here is that all views use six values to decide how much space they want to use for layouts, and understanding how they interact is key to getting the most out of SwiftUI's layout system.

The six are:

- Minimum width and minimum height, which store the least space this view accepts. Anything lower than these values will be ignored, causing the view to "leak" out of the space that was proposed to it.
- Maximum width and maximum height, which store the *most* space this view accepts. Anything greater than these values will be ignored, meaning that the parent must position

the view somehow inside the remaining space.
- Ideal width and ideal height, which store the *preferred* space that this view wants. It's okay to provide values outside these, as long as they still lie in the range of minimum through maximum. (If you're coming from a UIKit background, think of this as being like the intrinsic content size of your view.)

It's that last one that stores the natural size for our text: the text has ideal width and height suitable to store its characters on one single line, but it has no minimum size – it doesn't care if we try to squeeze the text into a limited width, because it will automatically wrap its letters around to multiple lines.

This is where the **fixedSize()** modifier comes in, which has the job of promoting a view's ideal size to be its minimum and maximum size. It's used like this:

```
Text("Hello, world!")
  .fixedSize()
  .frame(width: 30, height: 100)
```

When that code runs, our text will appear at its original width, despite us trying to override it. Take a moment to think about it: what do you think is actually happening internally with that code?

Tip: I know it's really tempting to skip ahead and read my discussion for this question, but I promise you'll learn more if you just pause for a moment and think of your own answer to the question: how will SwiftUI interpret that code, and in what order? Remember, the **fixedSize()** modifier create a parent view around the text, then in turn the **frame()** modifier creates parent view around the **fixedSize()**.

Still here?

Okay, what we end up with is this:

- We have three views in total: two **ModifiedContent** views and a **Text** view.
- In terms of parent-child relationships, our frame is the overall parent, and it has a fixed size

view for its child, which in turn has a text view for its child.

- When we create a 30x100 frame, it will offer that full space to **fixedSize()** child.
- The view created through **fixedSize()** proposes that same size to its **Text** view.
- The text has no idea it's going to be placed in a 30x100 frame, so it says, "well, my *ideal* size is 95x20, but I'm happy to take up less space if needed."
- The **fixedSize()** modifier then uses that same information, except now it turns the *ideal* size into a *fixed* size – it effectively returns the equivalent of **self.frame(width: text.idealWidth, height: text.idealHeight)**.
- So now the frame gets told it has to position a child much bigger than the size it proposed, and does so – it doesn't have a choice.

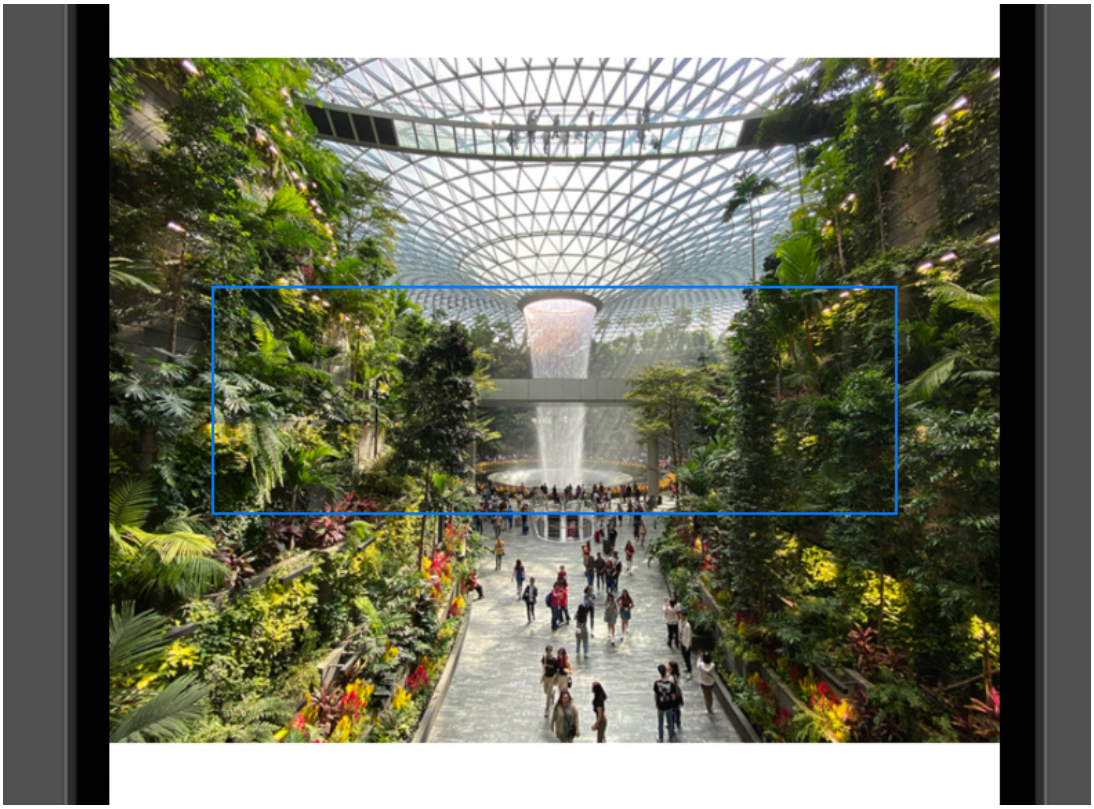
So, **fixedSize()** is how we promote *ideal* size up to be *fixed size*. You can use **fixedSize()** with no parameters to get both axes fixed at the same time, or provide Boolean parameters to fix one specific axis if you prefer.

In the case of text views, remember that fixing its horizontal size will default to it going over one line no matter how long its text is. If that's what you want, great! If not, you might find that fixing only its vertical axis is more useful, because it allows the text to be squashed horizontally while still growing as tall as needed to handle its lines wrapping.

But what will other view types do? Consider code like this:

```
Image("singapore")
  .frame(width: 300, height: 100)
```

On its surface that appears to request a 300x100 image, but any SwiftUI veteran will know it doesn't work like that – the image will be its original size, happily overflow the 300x100 frame we allocated for it. If you're using Xcode's preview canvas in selection mode you'll be able to see the thin outline of the frame right there.



You can see it more clearly if you use the **clipped()** modifier to see what's really happening:

```
Image("singapore")
  .frame(width: 300, height: 100)
  .clipped()
```

Perhaps now you have a better idea of what's happening here: image views get their ideal width and height directly from the image data you load into them, and just like text views no amount of futzing from us can override that.

"But Paul," I hear you say, "surely we can override the ideal size by making the image resizable?" Nope! Again, no amount of futzing from us can override the ideal size of an image – you can see it for yourself with code like this:

```
Image("singapore")
```

```
  .resizable()
  .fixedSize()
  .frame(width: 300, height: 100)
```

That makes the image resizable, but then promotes the ideal size into a fixed size – lo and behold, the image is back to its original size again.

Earlier I said, "when we use any modifier in SwiftUI, we are most of the time creating a new view that wraps the original view to add some extra behavior or styling."

Well, **resizable()** is one of the modifiers that *doesn't* create a new view: it sends back an image with the resizing behavior in place, but that isn't wrapped in some kind of "resizable" modifier – all we did was tell it to have a flexible width and height, but the underlying ideal size is still there.

The key here is to remember that whatever frame you try to apply to the image will automatically inherit values from the image that you don't specifically override.

For example, a common problem SwiftUI learners hit is when they use a very wide image alongside some text, like this:

```
VStack(alignment: .leading) {
  Image("wide-image")
  Text("Hello, World! This is a layout test.")
}
```

If that image is very large compared to the device you're using, e.g. 2000x1000, then it will stretch the width of the **VStack** beyond the edges of the screen, which will cause the text to be placed off screen too. This is rarely what you want – how would you go about fixing it?

To fix this without squashing the image, the simplest thing to do is wrap it in a frame with a completely flexible width, like this:

```
VStack(alignment: .leading) {
  Image("wide-image")
    .frame(minWidth: 0, maxWidth: .infinity)
  Text("Hello, World! This is a layout test.")
}
```

Critically, if you remove the **minWidth** parameter there, the frame will get its minimum width from the image, which again wants to show its entire picture at its natural size. And even *with* both minimum and maximum width provided, adding **fixedSize()** afterwards shows that the ideal width and height of the image is still there!

Another common problem beginners face is making two views the same width or height depending on their content. For example, they might have a layout like this:

```
HStack {
  Text("Forecast")
    .padding()
    .background(.yellow)
  Text("The rain in Spain falls mainly on the Spaniards")
    .padding()
    .background(.cyan)
}
```

In that layout, the **HStack** proposes the available space to its children, then splits up the space based on what they sent back. In practice, the larger text view will need significantly more

space than the smaller one, and when space is restricted the text will wrap – how can we make them the same size?

Well, the **background()** modifiers will create frames using whatever size they receive from the text they wrap, but if we add a custom frame to them then the backgrounds become free to expand to fill more space:

```
HStack {
  Text("Forecast")
    .padding()
    .frame(maxHeight: .infinity)
    .background(.yellow)
  Text("The rain in Spain falls mainly on the Spaniards")
    .padding()
    .frame(maxHeight: .infinity)
    .background(.cyan)
}
```

Now, remember: even though we've told the background it has a flexible maximum height, we *haven't* overridden its *ideal height* – that still comes through from the text it contains. So, each piece of text has its own ideal height that exactly fits its content, the background inherits that ideal height, and the **HStack** around it calculates its own ideal height to be the maximum of

the ideal heights of the two views it contains. As the text views have infinite maximum heights and will therefore expand to fill all the available height, the **HStack** will also expand to fill all the available height.

As a result, if we make the **HStack** use **fixedSize()**, we can make our two text views have the same height with very little code:

```
HStack {
  Text("Forecast")
    .padding()
    .frame(maxHeight: .infinity)
    .background(.yellow)
  Text("The rain in Spain falls mainly on the Spaniards")
    .padding()
    .frame(maxHeight: .infinity)
    .background(.cyan)
}
.fixedSize(horizontal: false, vertical: true)
```

Telling the **HStack** to fix its size is different from telling each of the text views to fix their size: we want them to resize upwards to some upper limit, which in this case is the ideal size of their container.

In their documentation, Apple describes **fixedSize()** as “the creation of a counter proposal to

the view size proposed to a view by its parent," which is quite apt once you understand what's happening internally.

Layout neutrality

Not all views have a meaningful ideal size, and in fact some views have very little sizing preference at all – they will happily adapt their own size based on the way we use them alongside other views. This is called being *layout neutral*, and a view can be layout neutral for any combination of its six dimensions.

In its purest form, layout neutrality it looks like this:

```
struct ContentView: View {
  var body: some View {
    Color.red
  }
}
```

That will fill the whole space with red, because the color will occupy whatever space is available. On the other hand, we could use the color as a background:

```
Text("Hello, World!")
  .background(.red)
```

Because the color doesn't actually care how much space it takes up, it will simply read the ideal and maximum sizes from its child, the text, and use that for itself. It *doesn't* read the minimum size because like I said earlier the text view is itself layout neutral for its minimum width and height – the text doesn't mind being squeezed smaller, so the background color doesn't mind either.

If we wanted to describe this behavior accurately, we'd say that the background color has an ideal width, ideal height, maximum width, and maximum height, but is layout neutral for its minimum width and minimum height. In practice this means it will fit snugly around the text it wraps rather than expanding to fill all the available space, but it's also happy to be squeezed downwards if needed.

Now take a look at this code:

```
struct ContentView: View {
  var body: some View {
    Text("Hello, World!")
      .frame(idealWidth: 300, idealHeight: 200)
      .background(.red)
  }
}
```

What do you think that might do, and why? Remember to work backwards – the layout starts with **background(.red)** as the parent, and works inwards from there.

Take a moment to pause and think about it before continuing.

If you break it down, the flow works like this:

- The background has the whole screen to work with, and **Color.red** is completely layout neutral so it's happen to occupy whatever is available. If this were the entirety of our layout, the color would expand to fill the screen.
- The background passes on the size proposal it received (the whole screen) to its child, **frame()**, which is layout neutral for minimum width, minimum height, maximum width, and maximum height.
- The frame proposes the whole screen to its child, the text, which is layout neutral for minimum width and minimum height, but cares very much about its ideal width, ideal height, maximum width, and maximum height.
- The text sends back to the frame the four values it cares about, but because the frame has provided its own ideal width and height those two are effectively ignored – the frame will use its own ideal width and height to override whatever the text asked for. However, because the frame is layout neutral for its maximum width and height, it will inherit those from the text.
- The frame then sends its final size up to its parent view, the background: it has the 300x200 ideal size we set, but a maximum size of whatever the text says it needs, e.g.

95x20.

- That 95x20 space then gets filled with the red background.

So, it really is important to think about all six sizing values when working with layout – they combine together in really interesting ways that may not necessarily make sense at first, but if you break it all down into a sort of blow-by-blow conversation then hopefully the exact behavior will become clear.

Helpfully, all six of these sizing values are *optional*, which means you can switch between layout neutrality and a fixed value by using either a number or **nil**. This is most commonly done using a ternary conditional operator, like this:

```
struct ContentView: View {
  @State private var usesFixedSize = false

  var body: some View {
    VStack {
      Text("Hello, World!")
        .frame(width: usesFixedSize ? 300 : nil)
        .background(.red)

      Toggle("Fixed sizes", isOn: $usesFixedSize.animation())
    }
  }
}
```

What that code does at runtime depends on the value of **usesFixedSize**:

- When it's true, the frame will propose 300 points of width to the text
- When it's false, the frame will propose to the text whatever size it receives from the **VStack** – it effectively does nothing at all.

So, if **nil** forces a view to be layout neutral for one particular dimension, what happens if we

have a view that's layout neutral for *every* dimension? We can certainly do this with the **frame()** modifier, but at some point every view has some kind of size data, even if that's just a nominal ideal size in order to keep our layouts from exploding.

To see an example of what I mean, try code like this:

```
ScrollView {
  Color.red
}
```

What size can the scroll view propose to **Color.red**?

Usually **Color.red** is happy to fill all the available space, but it wouldn't make any sense here because it would lead to an infinitely sized scroll view. In this situation, the red color will get a nominal 10-point height – enough that we can see it's being placed, but it won't expand beyond that.

You can get a better idea of what's happening if you try attaching a frame to the color, like this:

```
ScrollView {
  Color.red
    .frame(minWidth: nil, idealWidth: nil, maxWidth: nil,
minHeight: nil, idealHeight: nil, maxHeight: 400)
  }
```

```
}
```

We're now explicitly giving the color a maximum height to work with, but it won't matter – it will still stay 10 points high, because our frame is layout neutral for its ideal height. This isn't just that the color remains small while the frame grows empty around it, which you can see if you try coloring the frame blue:

```
ScrollView {
  Color.red
    .frame(minWidth: nil, idealWidth: nil, maxWidth: nil,
minHeight: nil, idealHeight: nil, maxHeight: 400)
    .background(.blue)
  }
}
```

You won't see any background there, because the frame still has an ideal height matching the color it contains. In order to get the color to expand, we need to override its ideal height in its frame parent, like this:

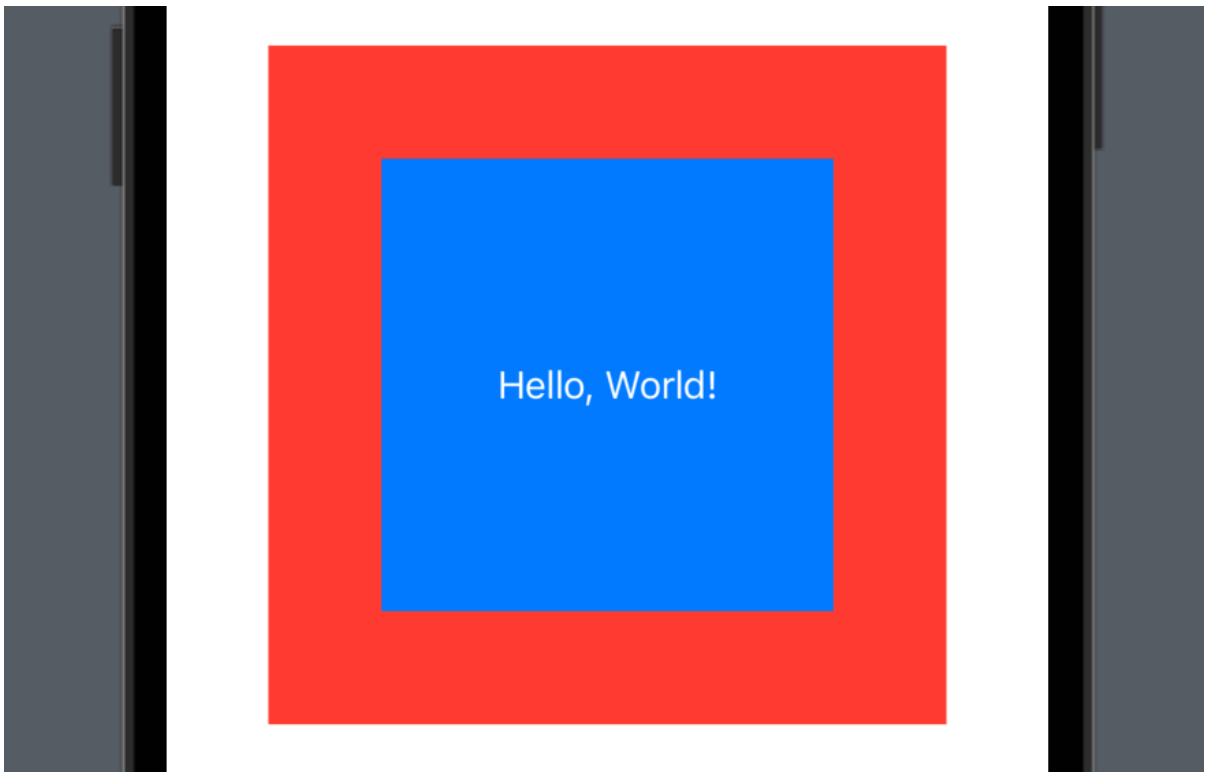
```
ScrollView {
  Color.red
    .frame(minWidth: nil, idealWidth: nil, maxWidth: nil,
minHeight: nil, idealHeight: 400, maxHeight: 400)
  }
}
```

Now the red color will expand to be 400 points high – internally the color has an infinite maximum height but gets capped to the 400 points it is offered by the frame, but now the frame *won't* adopt the 10-point ideal height of the color and will instead use 400 points so that the color can grow freely.

Multiple frames

Many SwiftUI modifiers can be stacked to create interesting effects, such as perhaps adding multiple backgrounds to some text:

```
Text("Hello, World!")
  .frame(width: 200, height: 200)
  .background(.blue)
  .frame(width: 300, height: 300)
  .background(.red)
  .foregroundColor(.white)
```



However, it's possible and indeed common to apply a **frame()** modifier twice back to back – with no other modifiers in between. This is because SwiftUI separates the concepts of fixed frames and flexible frames: a single view can have a fixed width or height, *or* it can have