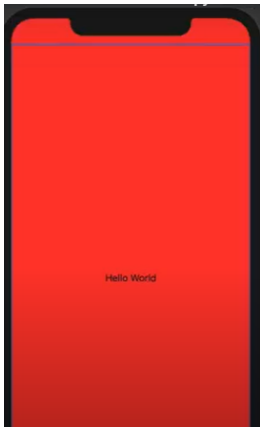
Instructor: Paul Hudson
Source: [100 Days of SwiftUI – Day 23 – Hacking with Swift]
[100 Days of SwiftUI – Day 24 – Hacking with Swift]

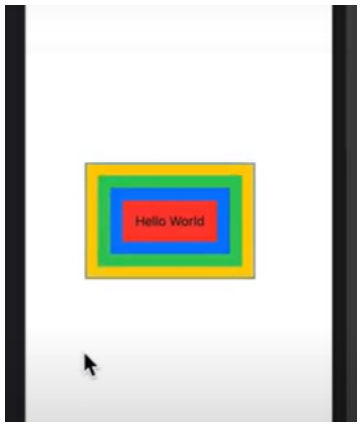
Frame

```
Text("Hello World")
    .frame(maxWidth: .infinity, maxHeight: .infinity)
    .background(Color.red)
    .edgesIgnoringSafeArea(.all)
```



Padding

```
import SwiftUI

struct ContentView: View {
    var body: some View {
        Text("Hello World")
            .padding()
            .background(Color.red)
            .padding()
            .background(Color.blue)
            .padding()
            .background(Color.green)
            .padding()
            .background(Color.yellow)
    }
}
```




```
struct ContentView: View {
    var body: some View {
        Button("Hello World") {
            print(type(of: self.body))
        }
        .frame(width: 200, height: 200)
        .background(Color.red)
    }
}
```

Conditional modifiers:

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

    var body: some View {
        Button("Hello World") {
            self.useRedText.toggle()
        }
        .foregroundColor(useRedText ? .red : .blue)
    }
}
```

self.useRedText.toggle() // toggle bw “true” and “false”
If useRedText is true ⇒ .foregroundColor.red; else ⇒ .foregroundColor.blue

Invalid return

```
var body: some View {  Function declares an opaque return type,
    if self.useRedText {
        return Text("Hello World")
    } else {
        return Text("Hello World")
            .background(Color.red)
    }
```

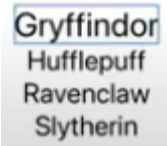
- Return Text != return Text.background
- The return type must be specific

Environment modifiers:

```
var body: some View {
    VStack {
        Text("Gryffindor")
        Text("Hufflepuff")
        Text("Ravenclaw")
        Text("Slytherin")
    }
    .font(.title)
}
```

- .font(.title) // put this after all the texts will make all the text get the “title” font

```
var body: some View {
    VStack {
        Text("Gryffindor")
            .font(.largeTitle)
        Text("Hufflepuff")
        Text("Ravenclaw")
        Text("Slytherin")
    }
    .font(.title)
}
```



- .font(.largeTitle) // put this after one of the text will make that specific text bigger

Regular modifiers:

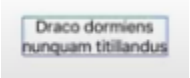
- `blur()` is a regular modifier, so any blurs applied to child views are *added* to the `VStack` blur rather than replacing it

```
VStack {
    Text("Gryffindor")
        .blur(radius: 0)
    Text("Hufflepuff")
    Text("Ravenclaw")
    Text("Slytherin")
}
.blur(radius: 5)
```

-

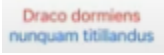
Views as property:

```
struct ContentView: View {
    let motto1 = Text("Draco dormiens")
    let motto2 = Text("nunquam titillandus")

    var body: some View {
        VStack {
            motto1
            motto2
        }
    }
```



^^^ writing like this still gives some output

```
var body: some View {
    VStack {
        motto1
            .foregroundColor(.red)
        motto2
            .foregroundColor(.blue)
    }
```



^^^ get the result individually

```
var motto1: some View { Text("Draco dormiens") }
let motto2 = Text("nunquam titillandus")
```

^^^ creating the computed property here is also allowable

View composition:

```
struct ContentView: View {
    var body: some View {
        VStack(spacing: 10) {
            Text("First")
                .font(.largeTitle)
                .padding()
                .foregroundColor(.white)
                .background(Color.blue)
                .clipShape(Capsule())

            Text("Second")
                .font(.largeTitle)
                .padding()
                .foregroundColor(.white)
                .background(Color.blue)
                .clipShape(Capsule())
        }
```

Another way to do the above

```
struct CapsuleText: View {
    var text: String

    var body: some View {
        Text(text)
            .font(.largeTitle)
            .padding()
            .foregroundColor(.white)
            .background(Color.blue)
            .clipShape(Capsule())
    }
}
```

- Create a struct CapsuleText right before the struct ContentView
- CapsuleText struct conforms View protocol
- Write everything similar to the body above
    - But the real text is replaced by the var text

```swift
struct ContentView: View {
    var body: some View {
        VStack(spacing: 10) {
            CapsuleText(text: "First")
            CapsuleText(text: "Second")
        }
    }
}
```

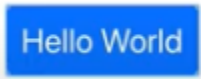
- Then call the struct in the original body

```swift
struct CapsuleText: View {
    var text: String

    var body: some View {
        Text(text)
            .font(.largeTitle)
            .padding()
            .background(Color.blue)
            .clipShape(Capsule())
    }
}

struct ContentView: View {
    var body: some View {
        VStack(spacing: 10) {
            CapsuleText(text: "First")
                .foregroundColor(.white)
            CapsuleText(text: "Second")
                .foregroundColor(.yellow)
```

- If we remove .foregroundColor from the CapsuleText struct, we can apply .foregroundColor individually for each text

A struct that conforms ViewModifier protocol:

```swift
struct Title: ViewModifier {
    func body(content: Content) -> some View {
        content
            .font(.largeTitle)
            .foregroundColor(.white)
            .padding()
            .background(Color.blue)
            .clipShape(RoundedRectangle(cornerRadius: 10))
    }
}

struct ContentView: View {
    var body: some View {
        Text("Hello World")
            .modifier(Title())
```



- A struct conforming ViewModifier is written before being used inside the ContentView
- Inside that struct, a function returns some View

- Then inside the ContentView, apply this modifier to Text
    - Text(“text”).modifier(Title())

```
extension View {
    func titleStyle() -> some View {
        self.modifier(Title())
    }
}

struct ContentView: View {
    var body: some View {
        Text("Hello World")
            .titleStyle()
    }
}
```

- We can put an extension bw the Title struct and the ContentView
- “Extensions allow you to add methods to existing types, to make them do things they weren’t originally designed to do.”
    - Create an extension of the View protocol
    - Add a new title modifier
    - Then use it. The result is same as above

Create watermarks

- Create an extension of the View protocol
- Add watermarked function
- Call it inside the body

```
extension View {
    func watermarked(with text: String) -> some View {
        self.modifier(Watermark(text: text))
    }
}

struct ContentView: View {
    var body: some View {
        Color.blue
            .frame(width: 300, height: 300)
            .watermarked(with: "Hacking with Swift")
    }
}
```

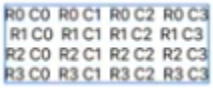
Custom containers: struct GridStack<Content: View> conforms the View protocol

```swift
struct GridStack<Content: View>: View {
    let rows: Int
    let columns: Int
    let content: (Int, Int) -> Content

    var body: some View {
        // more to come
    }
}
```

```swift
struct GridStack<Content: View>: View {
    let rows: Int
    let columns: Int
    let content: (Int, Int) -> Content

    var body: some View {
        VStack {
            ForEach(0 ..< rows) { row in
                HStack {
                    ForEach(0 ..< self.columns) { column in
                        self.content(row, column)
                    }
                }
            }
        }
    }
}
```

- For each row in rows
    - For each column in columns
        - Print content(row, column)

```swift
struct ContentView: View {
    var body: some View {
        GridStack(rows: 4, columns: 4) { row, col in
            Text("R\(row) C\(col)")
        }
    }
}
```



- Call the GridStack inside the body of the ContentView

Add a circle ...

```swift
struct ContentView: View {
    var body: some View {
        GridStack(rows: 4, columns: 4) { row, col in
            HStack {
                Image(systemName: "\(row * 4 + col).circle")
                Text("R\(row) C\(col)")
            }
        }
    }
}
```

Or ..
Create an init inside the GridStack struct

```
init(rows: Int, columns: Int, @ViewBuilder content: @escaping
    (Int, Int) -> Content) {
     self.rows = rows
     self.columns = columns
     self.content = content
}
```

- `@escaping` attribute, which allows us to store closures away to be used later on

```
struct ContentView: View {
    var body: some View {
        GridStack(rows: 4, columns: 4) { row, col in
            Image(systemName: "\(row * 4 + col).circle")
            Text("R\(row) C\(col)")
        }
    }
}
```

- If a `VStack` has a foreground color and some text inside also has a foreground color, the text's foreground color is used.
  Local modifiers always override environment modifiers from the parent.
- We can return a specific View instead of some View in a SwiftUI body, but this is not recommended
- SwiftUI views should contain structs (not classes. If it contains classes, it might not compile or run)
- Colors are views