Week 2 Tutorial Building the Parcel App Calculator

🎯 Learning Outcomes

By the end of this tutorial, you will be able to:

- Build a SwiftUI interface with Labels, TextFields, a Toggle, and a Button.
- Implement input validation and display user-friendly error messages.
- Write and call separate functions for different calculation logic.
- Use a `Toggle` switch to dynamically change an app's behaviour.

You will build the **Parcel App Calculator**, an app that calculates shipping costs based on a parcel's dimensions and weight.
You will create two versions:

- **Version 1:** A basic calculator with simple rules.
- **Version 2:** An enhanced version that adds a toggle for advanced pricing rules.

NB: This app will be the basis for assessment one and fundamental in building an iOS app.

Version 1: Basic Calculator

Step 1: Build the User Interface

Your main ContentView should be structured with a VStack containing the following SwiftUI views:

1. A Text view for the app title (e.g., "Parcel Cost Calculator").
2. Four HStack views, each containing:
    a. A Label view without the icon (e.g., "Weight (kg):")
    b. A TextField for user input (bound to @State variables for weight, length, width, and depth).
3. A Button view labeled "Calculate Cost".
4. A Text view to display the result (the calculated cost or an error message).

Postage Cost Rules for basic version

- **Error check** – show an error if any input is empty or ≤ 0.
- **Base cost** – every parcel starts at £3.00.
- **Weight charge** – add £0.50 per kilogram.
- **Volume charge** –
    - Calculate volume: length × width × height (in cm).

- Convert to litres (1000 cm^3 = 1 litre).
- Add £0.10 per litre.

- **Minimum charge** – total must be at least £4.00.

A useful approach is to build the basic UI first and then code the functionality later, so start with a title for the app followed by labels and TextFields to capture the parcel data and a button that will simply function initially to confirm the parcel data by printing it in the console.

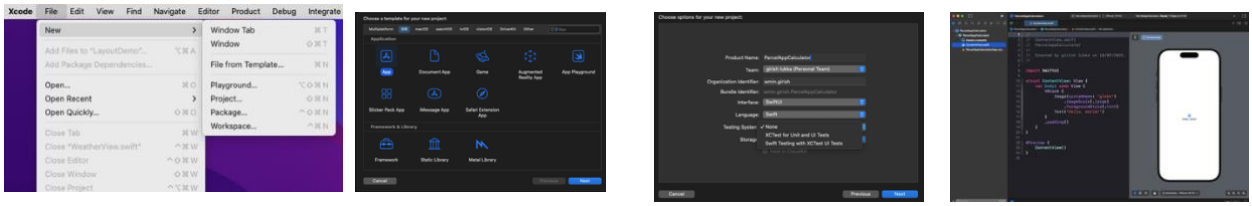
The UI for this is shown in Figure 1 – essentially it is a title, a label and TextField for each parcel attribute – weight, height, length, width, a button and cost if it exists with a "£" symbol, formatted to 2 dps.

Version 1:

Getting Started

1. Open **Xcode 16**.
2. Select **iOS App** template.

Name your app **ParcelAppCalculator**.



Edit ContentView.swift and replace its code with the code shown in image below.

Do type the code from the image and use Xcode completion features and read the error messages.

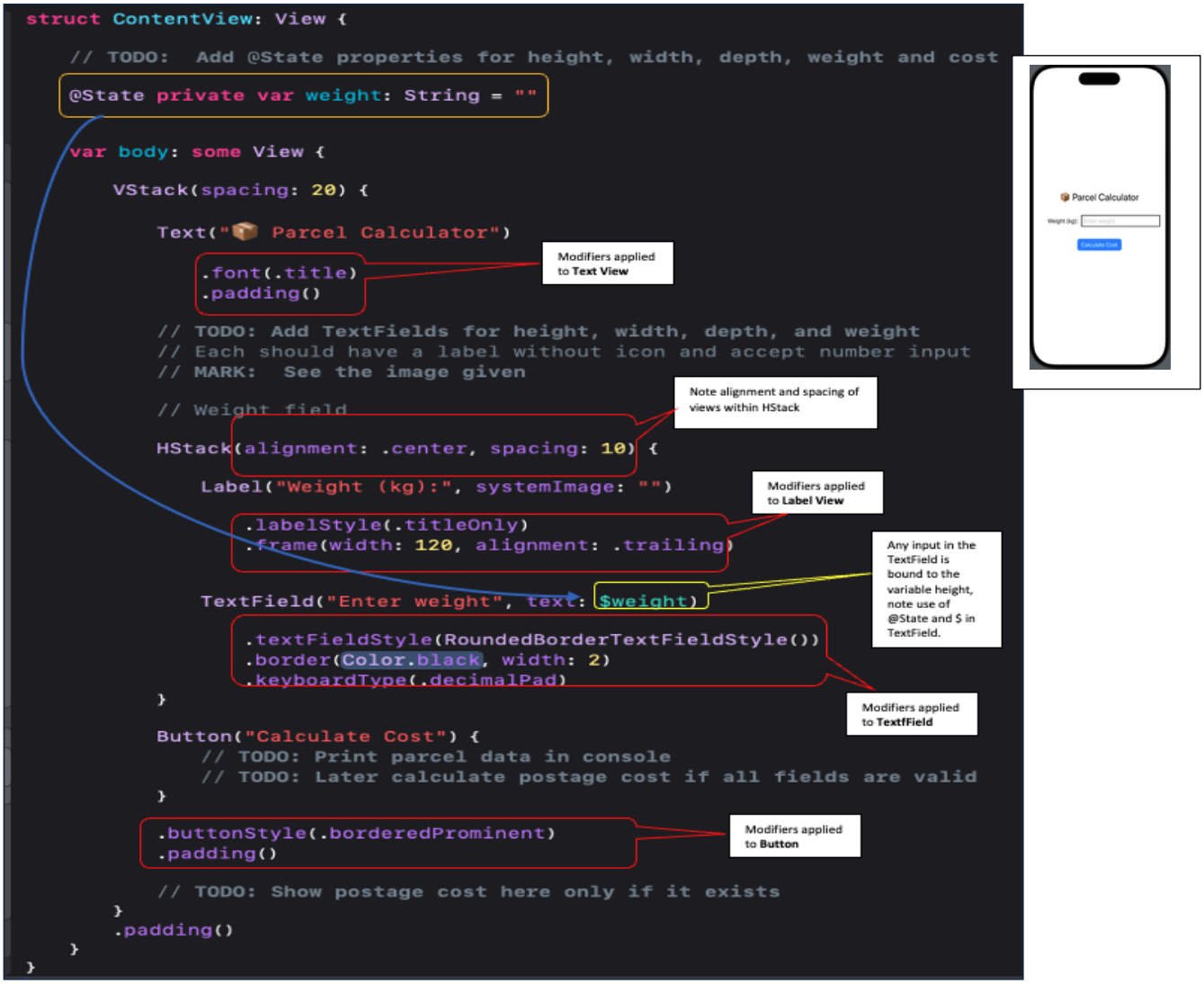


Figure 1: Edited – new ContentView

Code Explanation

TextField("Enter Weight (kg)",text: $weight)
The first argument to the TextField view is the placeholder that provides hints to the user on how to enter input. In this field, the user must enter weight in kilograms e.g., 5 (meaning 5 kg).

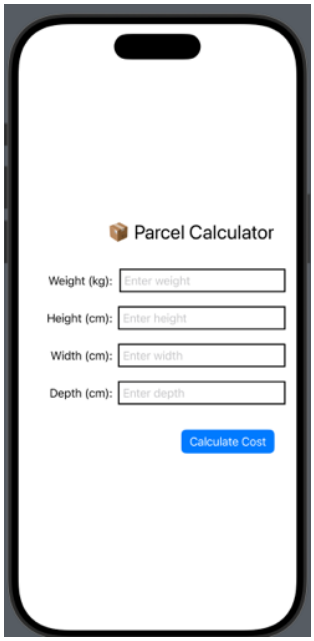
Note use of $ prefix to bind the state variable to the text value in TextField view, as shown in Figure 1.
Preview updates dynamically - the view within XCode so there is no need to use a simulator.

Complete UI Build (repeated from step1)

Use one main VStack view to stack everything vertically.
Each input (Height, Width, Depth, Weight) will sit in its own HStack view, containing:

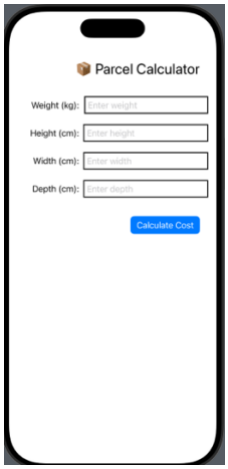
1. A Label view on the left.
2. A TextField view on the right.

Complete UI:



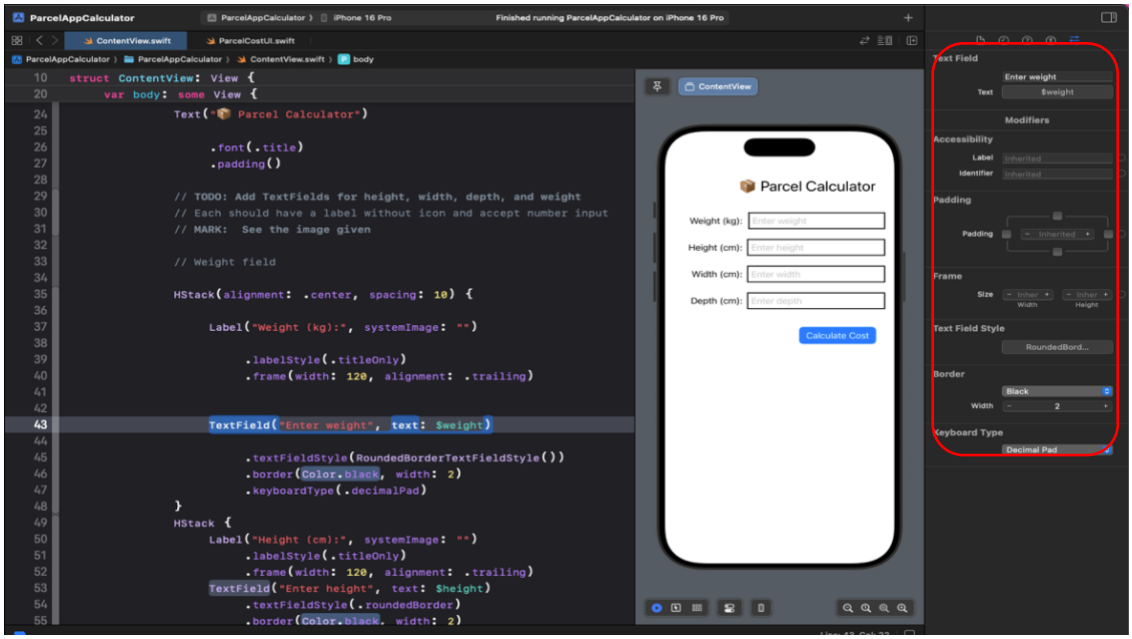
Note that the the TextFields and labels are all in the centre.

Use a frame around VStack to better align the labels, TextField and button.



Examine the effect of commenting out the TextField modifier relating KeyboardType.
What other KeyboardType modifiers are available?

Highlight the weight TextField by hovering the mouse over and see the information in the properties panel.



Click the “+” button and use Add Modifier tab to see other modifier options to add. Experiment with some of them to enhance the Textfield.
This completes the UI part, quite basic but sufficient for our purpose.

Version 1 Functionality:

Initially, user enters data via phone keyboard and Calculate button should be disabled until all TextFields have data and it should then print the user data in console using string interpolation.

Disabling Button

There are many ways to do this but for this app use a computed variable that will be either true or false based on if there is data in all text fields or missing in anyone.

```
var isDisabled: Bool {
    weight.isEmpty || length.isEmpty || width.isEmpty || height.isEmpty
}
```

Add a disabled modifier to button code.
Enhance it by using a ternary operator to set background color to grey or blue based on value of var isDisabled.

Question – why is *var isDisabled*ed not annotated with @State?

```
// Add this modifier to your Button
Button("Calculate Cost") {
    calculateCost()
}
.disabled(isDisabled)
.padding()
.background(isDisabled ? Color.gray : Color.blue)
.foregroundColor(.white)
.cornerRadius(8)
```

Next, edit button code so that when it is clicked, user entered data is printed to console using string interpolation and a calculation based on these rules is performed:

Error **message**
Displays error message if any input is not > 0
Check all parcel data exists and is greater than 0, otherwise display error message.
Calculation should only happen if all data is numeric and >0.

Base **Cost**
Every parcel starts with a fixed base cost of **£3.00.**
Weight **Charge**
Add **£0.50 for every kilogram** of the parcel's weight.
Volume **Charge**
Calculate the **volume:**
volume = length × width × height
(where dimensions are entered in **centimetres**)
Convert this to **litres** (since 1000 cm^3 = 1 litre).
Add **£0.10 for every litre of volume.**
Minimum **Charge**
The total postage cost must be at least **£4.00,** even if the above calculation is lower.

Button code:

```
Button("Calculate Cost") {

        print("Parcel Data:")
        print("Height: \(height)")
        print("Width: \(width)")
        print("Height: \(height)")
        print("Weight: \(weight)")

        if let weightValue = Double(weight),
           let depthValue = Double(depth),
           let widthValue = Double(width),
           let heightValue = Double(height),
           weightValue > 0, depthValue > 0, widthValue > 0, heightValue > 0 {
            let volume = depthValue * widthValue * heightValue
            var totalCost = 3.00 // Base cost
            // Weight charge
            totalCost += weightValue * 0.50
            // Volume charge
            totalCost += (volume / 1000) * 0.10
            // Ensure minimum charge
            totalCost =  max(totalCost, 4.00)
            cost = String(format: "%.2f", totalCost)
            print("£"+cost)
        }
        else{
            cost = "Error: Please enter a valid numeric amount"
        }
}
```

Code to display cost calculation or error

```
if !cost.isEmpty {
    if let costValue = Double(cost), costValue > 0.00 {
        Text("Total cost is £\(cost)")
    } else {
        Text(cost)
            .foregroundColor(.red)
    }
}
```

Question – why is cost cast as a Double?

Using code folding, get an overview of ContentView:

Re-factor Button code:

Instead of writing the code inside Button's action parameter, it is better to use a function that is within ContentView but outside of body and the button action would invoke function call:

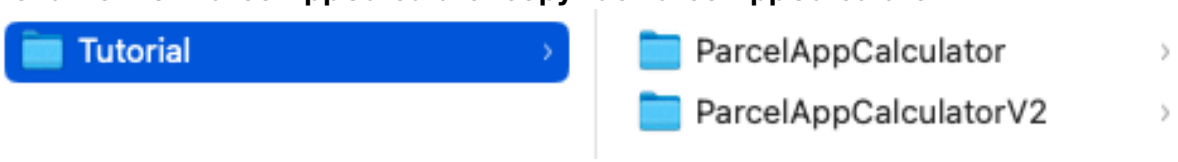
```
import SwiftUI

struct ContentView: View {
    @State private var weight: String = ""
    @State private var length: String = ""
    @State private var width: String = ""
    @State private var height: String = ""
    @State private var cost: String = ""

    var isDisabled: Bool { ... }

    var body: some View { ... }

    private func calculateCost() { ... }

}
#Preview {
    ContentView()
}
```

This completes the first version of the app – create a personal summary of how this has been built and importantly the SwiftUI views that have been used, configured and enhanced through modifiers and finally Swift language constructs to manipulate the data.

Version 2: Advanced Calculator & Toggle

Make a copy of **version1 and save it in the same folder:**

Rename the "ParcelAppCalculator copy" as ParcelAppCalculatorV2:



You should now use the second version to continue adding features and functionality to first version.

Additional New requirements:

User must be able to use a toggle switch to choose between a basic calculator or an advanced calculator. Basic calculator criteria have already been implemented in the first version.

Advanced Postage Cost Rules:

1. **Input Validation**
    - All inputs (weight, length, width, height) must be numbers greater than 0.
    - If weight is more than **30kg**, show error: *"Max weight: 30kg"*.
    - If any single dimension (length, width, or height) is more than **150cm**, show error: *"Max dimension: 150cm"*.
2. **Calculate Volume and Weight**
    - **Volume** = length × width × height (in cm^3).
    - **Dimensional weight** = volume ÷ 5000.
    - **Chargeable weight** = greater of actual weight or dimensional weight.
3. **Base Cost**
    - Start with **£2.50**.
    - Add **£1.50 per kg** (using chargeable weight).
    - Add **£0.75 per litre** (1000 cm^3 = 1 litre).
4. **Weight-based Surcharges**

- If weight > 20kg → multiply total cost by **1.50**.
- Else if weight > 10kg → multiply by **1.25**.

5. **Minimum Charge**

    - The final postage cost must be at least **£5.00**.

Step 1: Add a Toggle Switch

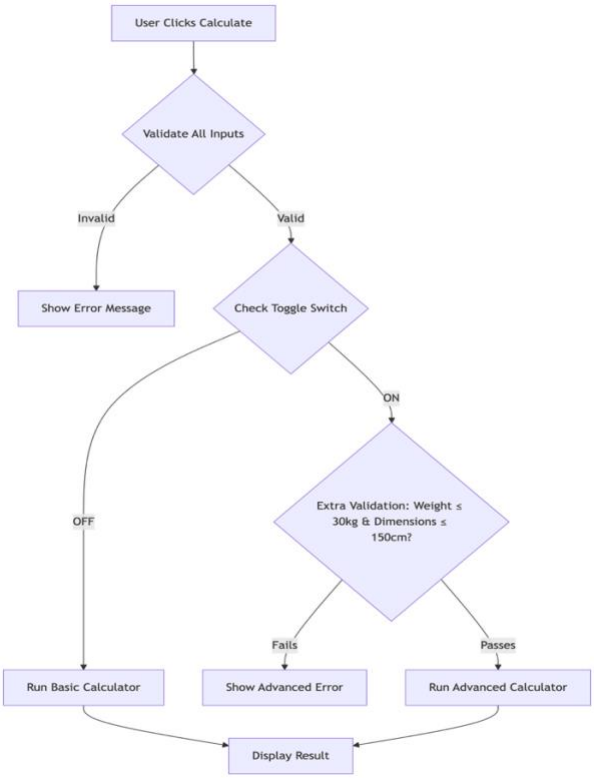
Add a `@State` variable and a `Toggle` view to your `ContentView` to switch between basic and advanced pricing

Step 2 – Understand the Advanced Pricing Rules

Advanced Rules:

1. **Validation:** Weight must be ≤ 30kg. Each dimension (L, W, D) must be ≤ 150cm.
2. **Dimensional Weight:** Chargeable weight is the *greater* of:
    - Real weight
    - Dimensional weight (Volume in cm^3 / 5000)
3. **Base Cost:** £2.50 + (£1.50 × chargeable weight) + (£0.75 × volume in litres)
4. **Surcharges:**
    - Weight > 20kg: Multiply total by 1.5.
    - Weight > 10kg: Multiply total by 1.25.
5. **Minimum Charge:** £5.00.

Step 3: Refactor the calculateCost() Function – Refer to Flowchart below:




```
private func calculateCost() {
    cost = ""
    errorMessage = ""

    // 1. Convert inputs to Double and validate they are > 0, use guard let:
    guard let weightValue = Double(weight), weightValue > 0,
          let lengthValue = Double(length), lengthValue > 0,
          let widthValue = Double(width), widthValue > 0,
          let heightValue = Double(height), heightValue > 0 else {
        errorMessage = "Please enter valid numbers greater than 0."
        return
    }

    // 2. Check the toggle state and route to the correct calculator
    if useAdvancedPricing {
        // 3. Perform advanced validation
        if weightValue > 30 {
            errorMessage = "Error: Max weight is 30kg."
            return
        }
        if lengthValue > 150 || widthValue > 150 || heightValue > 150 {
            errorMessage = "Error: Max dimension is 150cm."
            return
        }
        // 4. If validation passes, call the advanced function
        let totalCost = calculateAdvancedCost(weight: weightValue, length: lengthValue, width: widthValue, height: heightValue)
    } else {
        // Use the basic function
        cost = calculateBasicCost(weight: weightValue, length: lengthValue, width: widthValue, height: heightValue)
    }
}
```

An important change from Version 1 is to use a variable for error message when initial validation fails or when advanced calculator validation fails.

```
@State private var errorMessage: String = "" // is this correct?
```

Write a **function** that performs a basic calculation that uses input data cast as double as arguments to perform a calculation and return the cost. This was done in Version 1

```
private func calculateBasicCost(weight: Double, length: Double, width: Double, height:
    Double) -> Double {
    // Todo write your code here:


    return max(totalCost, 4.00)
}
```

Write a **function** that performs advanced calculation that uses input data cast as double as arguments to perform a calculation and return the cost.

```
private func calculateAdvancedCost(weight: Double, length: Double, width: Double, height:
    Double) -> Double {
    // 1. Calculate Volume and Weight
    // Todo write your code here

    // 2. Base Cost
    // Todo write your code here

    // 3. Weight-based Surcharges
    // Todo write your code here

    // 4. Minimum Charge
    return max(totalCost, 5.00)
}
```

Test your Version2 app and confirm that it works as expected.