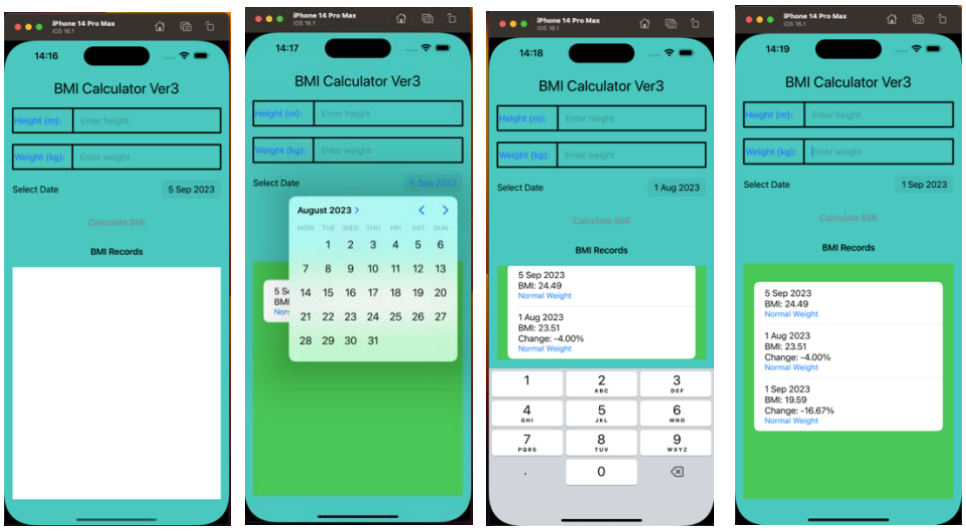
7SENG002W Week 2 SwiftUI Tutorial

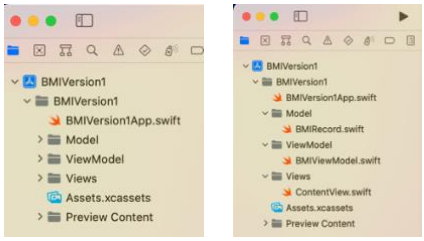
This week's tutorial will be continuation of BMI Calculator app that you started last week.

Recap Version 3:

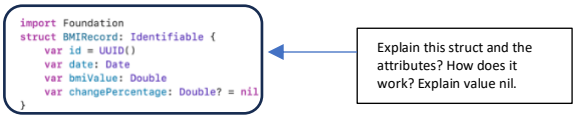
Keep a record of BMI calculations that includes date.
Show all bmi records as a list with each row showing the date, bmivalue and percentage change from last time, if it exists.
Date can be set by user but only up to current date, no future dates.
Display a percentage change in bmi value from previous time if there is a previous record.
A reminder – last year in your OOP module, you did similar work for the GUI application.
Below are the app images to help you figure out what data structures and processing you will need to do. See if you can find errors.
This version does not require persistence.



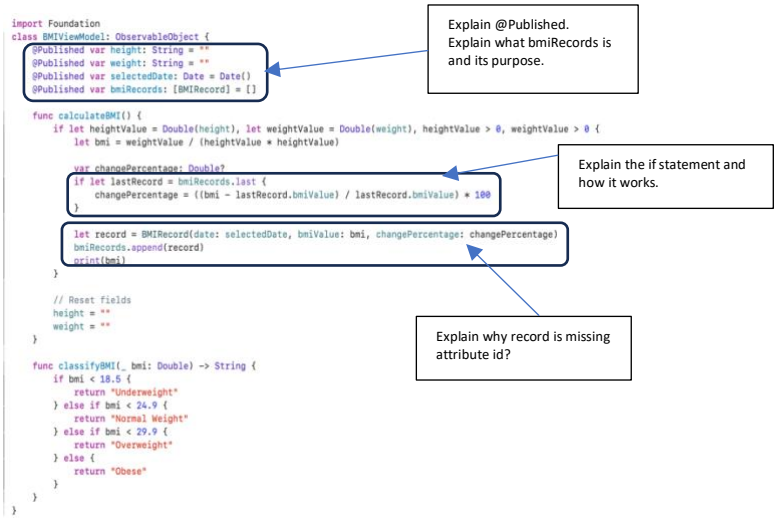
This task is ideally suited to MVVM design pattern.
Start by creating 3 groups - one for model, one for viewmodel and one for views as shown below:



Model:

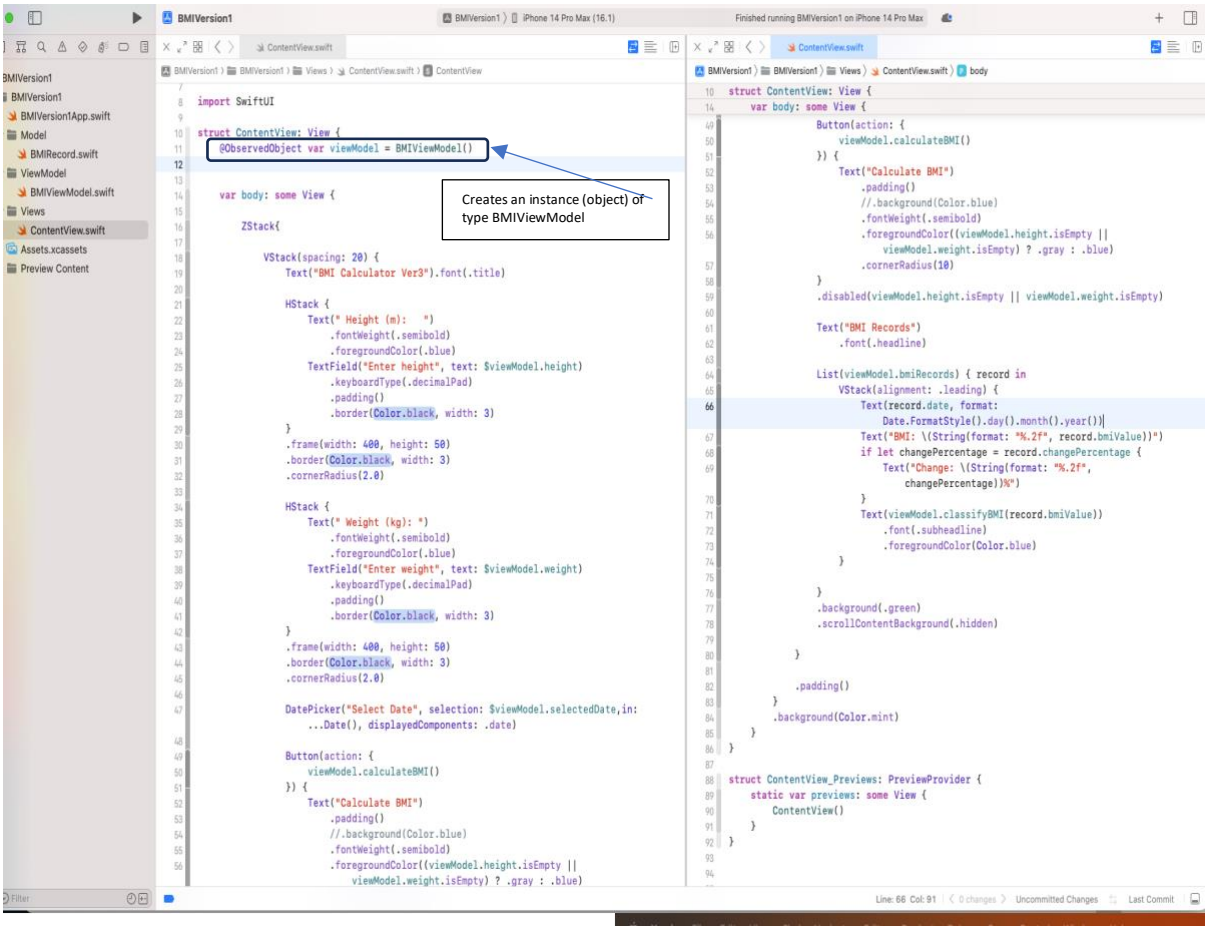


ViewModel: this uses the dataModel - BMIRecord



View:

Create an instance of ViewModel - @ObservedObject var viewModel = BMIViewModel()



Build Version3 and test it.
What issues are there and why do they happen?
This completes version3.

As before, make a copy of the whole project folder and save it in a new folder.
Rename the copied folder BMIVersion4 and use that to continue working on the app.

Version 4

Additional requirements are to display the current bmi value with date and percentage change where possible under the Calculate button as well as address issues from version3.
The issues that version3 had are:

1. Data is displayed in order of creation, rather than by date.
2. Percentage change calculation is correct, but the record used to do it is incorrect, so giving incorrect answer.
3. Calculations are not persistent – data is lost when app is switched off.
4. Device keyboard is not dismissed when the calculation takes place, obscuring the list display.

To address, the above issues, the first 3 are to do with viewmodel (app logic) and last one (keyboard) is a UI issue, so the view can deal with that.

Code changes to ViewModel:

1.Whenever a record is appended to bmiRecords array, sort this array on date attribute.

2. Process the sorted array and calculate percentage changes between consecutive records and store it back in the sorted array. Aside – what happens the first time a calculation is performed?
3. For persistence we can use userDefaults as a learning activity but note that for proper persistence we would use CoreData/SwiftData that uses sqlite database.

The process here is to take the sorted array and encode it as a json object and then with a key (a string of our own choice) pass it userDefaults.

To retrieve the stored data, reverse the above process – i.e. get data from userDefaults with our key if it exists which will come back as a json object and then decode this into an array of BMIRecord and then store it as a sorted array on date in bmiRecords.

Code changes in viewModel:

```
class BMIViewModel: ObservableObject {
    @Published var height: String = ""
    @Published var weight: String = ""
    @Published var selectedDate: Date = Date()
    @Published var bmiRecords: [BMIRecord] = []
    private let userDefaults = UserDefaults.standard
        private let bmiRecordsKey = "BMIRecordsKey"

    init(){
        loadStoredBMIRecords()
    }
func updateStoredBMIRecords() {
    print("bmiRecords... before encoding....\(bmiRecords)")
    let encoder = JSONEncoder()

    if let encoded = try? encoder.encode(bmiRecords) {
        // Convert the encoded data to a JSON string for debugging
        if let jsonString = String(data: encoded, encoding: .utf8) {
            print("JSON String representation of encoded data:
                \(jsonString)")
        }

        userDefaults.set(encoded, forKey: bmiRecordsKey)
    }
}


         func loadStoredBMIRecords() {
                if let data = userDefaults.data(forKey: bmiRecordsKey) {
                    print("bmiRecords... before decoding....\(data)")
                    let decoder = JSONDecoder()
                    if let decoded = try? decoder.decode([BMIRecord].self, from:
                        data) {
                        bmiRecords = decoded.sorted(by: { $0.date < $1.date })
                        //bmiRecords = decoded
                        print("decoded data ...\(decoded)")
                    }
                }
            }


func calculateBMI() {

    if let heightValue = Double(height), let weightValue =
        Double(weight), heightValue > 0, weightValue > 0 {
        let bmi = weightValue / (heightValue * heightValue)
        let record = BMIRecord(date: selectedDate, bmiValue: bmi)

        bmiRecords.append(record)
        bmiRecords.sort(by: { $0.date < $1.date }) // Sort the records by
            date

        for index in 1..<bmiRecords.count {
            let previousRecord = bmiRecords[index - 1]
            let changePercentage = ((bmiRecords[index].bmiValue -
                previousRecord.bmiValue) / previousRecord.bmiValue) * 100
            bmiRecords[index].changePercentage = changePercentage

        }

        updateStoredBMIRecords()
    }
```

Code change in View:

1. Add a new variable to manage Textfield focus:

```
struct ContentView: View {
    @StateObject private var viewModel = BMIViewModel()
    @FocusState private var fieldIsFocused: Bool
```

2. Add a .focussed modifier to both TextFields:

```
TextField("Enter height", text: $viewModel.height)
    .focused($fieldIsFocused)
TextField("Enter weight", text: $viewModel.weight)
    .focused($fieldIsFocused)
```

3. Modify the Button behaviour on the assumption that when the user is ready to press the button, all data has been entered and the keyboard is not needed anymore. This is achieved by changing the value of fieldIsFocussed from true to false. Can you identify a different way to do this same action?

```
Button(action: {
    viewModel.calculateBMI()
    fieldIsFocused = false
```

4. Display current bmivalue with date and percentage change where possible:

```
if let lastRecord = viewModel.bmiRecords.last {
    HStack(spacing: 10){
        Text("BMI: \(String(format: "%.2f", lastRecord.bmiValue))")
            .font(.headline)
            .padding()

        Text("Date: \(lastRecord.date, formatter: DateFormatter.shortDate)")
            .font(.subheadline)
            .padding()

        if let changePercentage = lastRecord.changePercentage {
            Text("Change: \(String(format: "%.2f", changePercentage))%")
                .foregroundColor(changePercentage >= 0 ? .white : .black)
                .font(.subheadline)
                .padding()
        }
    }
}
```

5. One additional change is to embed the list in a ScrollView:

```
ScrollView {
    List(viewModel.bmiRecords) { record in

        HStack(spacing: 10) {
            Text("\(record.date, formatter: DateFormatter.shortDate)")

            Text(String(format: "%.2f", record.bmiValue))

            if let changePercentage = record.changePercentage {
                Text(String(format: "%.2f%%", changePercentage))
                    .foregroundColor(changePercentage >= 0 ? .red : .green)
            } else { Text(" N/A ")}

        }
        Text(viewModel.classifyBMI(record.bmiValue))
            .font(.subheadline)
            .foregroundColor(Color.blue)
    }
    .frame(height: 400)
    .background(.green)
    .scrollContentBackground(.hidden)

}
```

This completes Version4.

Version5 – Independent activity

This version has very a small change in coding terms but quite significant conceptually as some of it is managed by closures (more on closures in the coming weeks).
Whenever data is entered in a TextField, action can take place as data is edited or on submit or onchange.
Implementation of anyone of these will make the button redundant.

```
HStack {
    Text(" Height (m):   ")
        .fontWeight(.semibold)
        .foregroundColor(.blue)
    TextField("Enter height", text: $viewModel.height)

        .onSubmit() {
            viewModel.calculateBMI()
        }

    //                  TextField("Enter height", text: $viewModel.height, onCommit: {
    //                      viewModel.calculateBMI()
    //                  })
    //                      .onChange(of: viewModel.height) { newHeight in
    //                                                          viewModel.calculateBMI() // Call calculate() when height changes
    //                                                  }
        .focused($fieldIsFocused)
        .keyboardType(.decimalPad)
        .padding()
        .border(Color.black, width: 3)


}
.frame(width: 400, height: 50)
.border(Color.black, width: 3)
.cornerRadius(2.0)
```

The above code shows .onSubmit modifier applied to height TextField and the commented out are the other versions that you can experiment with.

End of BMI Calculator Series.