Mobile Applications Development Lecture 3

Girish Lukka

Topics to be covered

- Functions – closures
- Swift variable – getters and setters
- Design patterns MVC, MVM
- SwiftUI data wrappers
- Code demos

Closures

- Closures are essentially anonymous functions or functions without a name and they're essentially a self-contained package of functionality that we can pass around and use.
- A normal function:

```
3   import Foundation
4
5   func functionName (parameter: parameterType) -> returnType {
6
7       //code to do something
8       return output
9   }
```

```
4
5
6   func getDrink (money: Int) -> Int {
7       let change  = money - 5
8       return change
9   }
10
11  var myChange = getDrink(money: 10)
12  print(myChange)
13
```

```
5
5

5
"5\n"
```

Passing a function as an argument:

```swift
// pass a function as an argument

func calculator (n1: Int, n2: Int, operation: (Int,Int) -> Int) -> Int {
    return operation(n1,n2)
}

func times(no1: Int, no2: Int) -> Int {
    return no1 * no2
    // signature of add is (Int,Int) -> Int
}

//call functimes as a stand-alone function
times(no1: 5, no2: 4)


// pass times as argument to calculator

calculator(n1: 4, n2: 7, operation: times)
```

Variable setters and getters example

```
5   //getter and setter
6
7   var width: Float = 7                                          7
8   var height: Float = 2.7                                       2.7
9
10  var tinsOfPaint: Int {
11
12      get {
13          let area = width * height                             (2 times)
14          let areaPerTin: Float = 1.5                           (2 times)
15          let numberofTins = area / areaPerTin                  (2 times)
16          let roundedTins = ceilf(numberofTins)                 (2 times)
17          return Int(roundedTins)                               (2 times)
18      }
19
20      set {
21          let areaCanCover = Double(newValue) * 1.5             19.5
22          print("area painted can be  \(areaCanCover)")         "area painted...
23      }
24  }
25  print(tinsOfPaint)                                            "13\n"
26  tinsOfPaint = tinsOfPaint                                     13
```

Line: 13 Col: 20

```
13
area painted can be  19.5
```

Convert a function to a closure

```
461  func times(no1: Int, no2: Int) -> Int {
462      return no1 * no2
463      // signature of times is (Int,Int) -> Int
464  }
465
466   convert times function to a closure as follows:
467
468   remove func and name and rewrite it in a { } with reserved word in:
469
470  { (no1: Int, no2: Int) -> Int in
471      return no1 * no2
472  }
473   // this can now be passed directly to calculator function::
474  calculator(n1: 5, n2: 7, operation: { (no1: Int, no2: Int) -> Int in
475      return no1 * no2
476  })
```

```
478  // This can be further shortened when type in closures can be clearly infered, only a
         single value is returned and it is the last argument - a trailing clousre
479
480  calculator(n1: 5, n2: 7) { $0 * $1}
481
```

Closure example with events

```
229   // .onEdit
230
231   @State private var text = ""
232
233   var body: some View {
234       TextField("Enter text", text: $text, onEdit: { isEditing in
235           if isEditing {
236               print("Editing started")
237           } else {
238               print("Editing ended")
239           }
240       })
241   }
```

Closure example with events

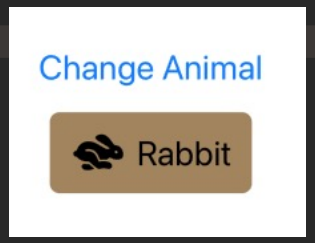
```
219  // .onCommit
220
221  @State private var text = ""
222
223  var body: some View {
224      TextField("Enter text", text: $text, onCommit: {
225          print("Text committed: \(self.text)")
226      })
227  }
```

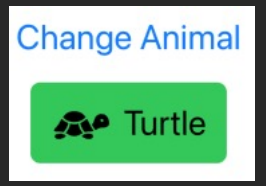
Closure example with events

```
209  // .onChange
210
211  @State private var text = ""
212
213  var body: some View {
214      TextField("Enter text", text: $text, onChange: { newValue in
215          print("Text changed to: \(newValue)")
216      })
217  }
```

.onChange closure example

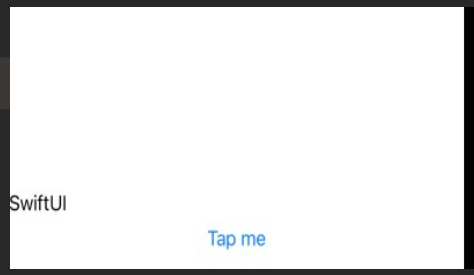
```swift
//.onChange Closure
import SwiftUI
struct OnChangeView: View {
    @State private var name = "Rabbit"
    @State private var icon = "hare.fill"
    @State private var color = Color.brown

    var body: some View {
        VStack(spacing: 20) {

            Button("Change Animal") {
                if name == "Rabbit" {
                    name = "Turtle"
                } else {
                    name = "Rabbit"
                }
            }

            Label(name, systemImage: icon)
                .padding()
                .background(color, in: RoundedRectangle(cornerRadius: 8))
        }
        .font(.title)
        .onChange(of: name) { newValue in
            if newValue == "Turtle" {
                icon = "tortoise.fill"
                color = Color.green
            } else {
                icon = "hare.fill"
                color = Color.brown
            }
        }
    }
}
```

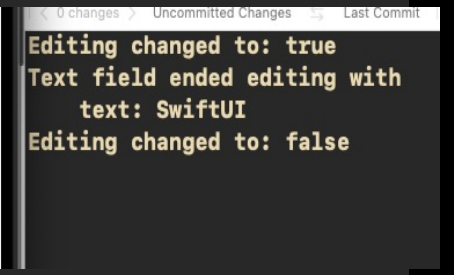


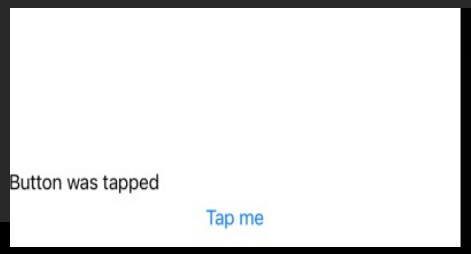


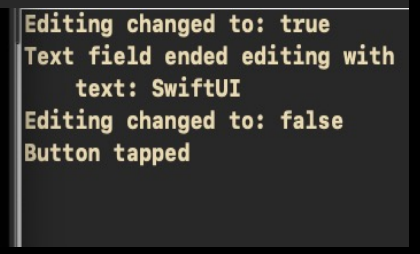
Nested Closure example with events

```
413  struct ContentView: View {
414      @State private var text = ""
415
416      var body: some View {
417          VStack {
418              TextField("Enter text", text: $text, onEditingChanged: { isEditing in
419                  print("Editing changed to: \(isEditing)")
420              }) {
421                  print("Text field ended editing with text: \(self.text)")
422              }
423
424              Button(action: {
425                  print("Button tapped")
426                  self.text = "Button was tapped"
427              }) {
428                  Text("Tap me")
429              }
430          }
431      }
432  }
```









Design Pattern– Model View Controller

- Model-View-Controller (MVC) is a design pattern for building user interfaces. In MVC, the model represents the data, the view represents the user interface, and the controller mediates between the two.
- For example, in a BMI calculator app, the model could store the user's height and weight. The view would display the user interface and allow the user to enter their height and weight. The controller would use the user's height and weight to calculate the user's BMI and update the view to display the result.

Design Pattern– Model-View-ViewModel

- Model-View-ViewModel (MVVM) is similar to MVC, but with a few key differences. In MVVM, the view model acts as a bridge between the model and the view. The view model contains the logic to transform the model into a format that can be displayed by the view.
- For example, in a BMI calculator app using MVVM, the model would still store the user's height and weight. The view model would calculate the user's BMI and provide it to the view to display. The view would then only be responsible for displaying the information provided by the view model.
- In the BMI calculator example, the view model might have properties for height, weight, and BMI, and methods to calculate the BMI based on the height and weight. The view would bind to these properties and display the values. When the user enters their height and weight, the view would update the view model, which would then recalculate the BMI and update the view.

Design Pattern– BMI Code

```
// MVC
struct Model {
    var height: Double
    var weight: Double
}

class ViewController {
    var model = Model(height: 0, weight: 0)
    var bmi: Double {
        return model.weight / (model.height * model.height)
    }

    func updateBMI() {
        // Update the view with the new BMI
    }
}

class View {
    var viewController: ViewController

    func heightChanged(_ height: Double) {
        viewController.model.height = height
        viewController.updateBMI()
    }

    func weightChanged(_ weight: Double) {
        viewController.model.weight = weight
        viewController.updateBMI()
    }
}
```

```
// MVVM
struct Model {
    var height: Double
    var weight: Double
}

class ViewModel {
    var model: Model
    var bmi: Double {
        return model.weight / (model.height * model.height)
    }

    init(model: Model) {
        self.model = model
    }
}

class View {
    @ObservedObject var viewModel: ViewModel

    func heightChanged(_ height: Double) {
        viewModel.model.height = height
    }

    func weightChanged(_ weight: Double) {
        viewModel.model.weight = weight
    }
}
```

MVVM – Book Example

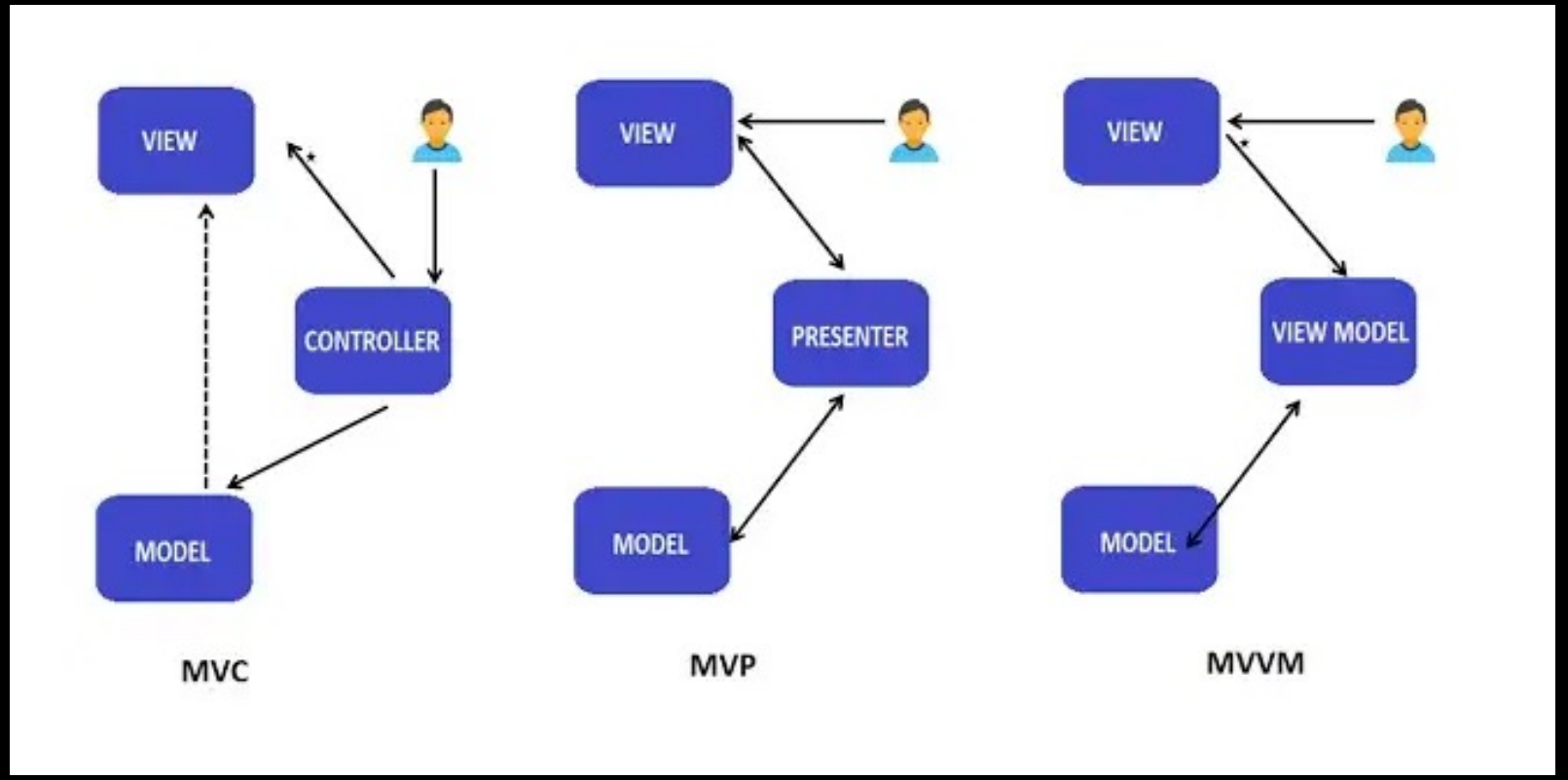
- In this example, Book is a struct that represents a book with a title and author.
- The BookListViewModel is an ObservableObject that holds an array of Book instances.
- The BookListView is a SwiftUI view that displays the books in a navigation view with a list.
- The view model is passed to the view as an environment object, making it available to all child views.
- The BookApp struct is a SwiftUI app that sets up the window group and initializes the environment object with an instance of BookListViewModel.

Design Pattern– Book Example

```swift
struct BookListView: View {
    @EnvironmentObject var viewModel: BookListViewModel

    var body: some View {
        NavigationView {
            List(viewModel.books) { book in
                VStack(alignment: .leading) {
                    Text(book.title)
                        .font(.headline)
                    Text(book.author)
                        .font(.subheadline)
                }
            }
            .navigationBarTitle("Books")
        }
    }
}

struct BookApp: App {
    var body: some Scene {
        WindowGroup {
            BookListView().environmentObject(BookListViewModel())
        }
    }
}
```

```swift
// MVVM BOOK
import SwiftUI

struct Book {
    let title: String
    let author: String
}

class BookListViewModel: ObservableObject {
    @Published var books: [Book] = [
        Book(title: "To Kill a Mockingbird", author: "Harper Lee"),
        Book(title: "Pride and Prejudice", author: "Jane Austen"),
        Book(title: "The Great Gatsby", author: "F. Scott Fitzgerald"),
        Book(title: "Moby-Dick", author: "Herman Melville"),
    ]
}
```

Visual: MVC MVP MVVM



The main objective of using a design pattern is to separate the ‘front-end’ and ’back-end’ and how to manage the flow of information.
This should lead to better maintainability and re-usability

MVVM - Book

Model

```
struct BookModel: Identifiable {
    var id = UUID()
    var name = ""
}
```

View Model

```
class BookViewModel: ObservableObject {
    @Published var books = [BookModel]()

    func fetch() {
        books =
            [BookModel(name: "SwiftUI Views"),
             BookModel(name: "SwiftUI Animations"),
             BookModel(name: "Data in SwiftUI"),
             BookModel(name: "Combine Reference")]
    }
}
```

View

```
struct MvvmExample: View {
    @StateObject var vm = BookViewModel()

    var body: some View {
        List(vm.books) { book in
            HStack {
                Image(systemName: "book")
                Text(book.name)
            }
        }
        .onAppear {
            vm.fetch()
        }
    }
}
```