Good Programming Practices Curriculum

MVVM

Model - View - ViewModel

View

User interface itself, defining layout and appearances

ViewModel

Exposes data from the Model in an easily consumable format for the View and contains presentation logic; doesn't directly reference the view

Model

Manages the data and business logic, often fetching data from repositories or services

How it Works

1. View displays data from ViewModel
2. User actions in View trigger commands in ViewModel
3. The ViewModel interacts with Model to fetch or update data
4. Model returns data to the ViewModel which then updates its exposed properties
5. Data binding automatically updates the View when the ViewModel's properties change.

- Separation of Concerns: Keeps UI code separate from business logic.
- Improved Testability: ViewModels can be tested without needing the UI.
- Enhanced Maintainability: Easier to modify UI or logic independently.
- Better Scalability: Provides a structured approach for growing applications.

Naming Conventions

Camel Case vs Snake Case

Swift is a CamelCase language.

UpperCamelCase (PascalCase):

Used for Types (Classes, Structs, Enums, Protocols)

lowerCamelCase

Used for Instances, variables, constants, and functions

Function Naming Conventions

Goal: be functionally descriptive
Good: `view.addSubview(button)`

- More descriptive; Can give other devs a sense of what the function does without reading it
  Bad: `view.add(button)`
- The `add` function is non-descriptive; It would be difficult to understand what the code is doing without reading what `add` does
  Bad: `view.removeElement(at: index)`
- Its obvious that we are removing an element

Common Naming Conventions

Get vs. Fetch

Get: When accessing data already in memory
Fetch: When retrieving data from database (Firebase)

Remove vs. Delete

Remove: When removing data from a collection; Removing data from local computer
Delete: Permanently destroying item from database (Firebase)

Other Commonly Used Terms

Create, Update, is
`createUser()`
`updateProfile()`
`isLoggedIn()`

If the term is `sort()`, `reverse()`, `append()` it modifies object in place
If the term is `sorted()`, `reversed()`, `appending()`, it returns a new copy

Code Documentation

Main Goal: Have code written in a way where someone can easily read the code and understand what is happening.
Comments should be primarily used to explain why we are doing something rather than what we are doing

Bad Practice:

```
// Iterate through the array of users
for user in users {
      // Check if user is 18 or older
      if user.age >= 18 {
            // Add to adults array
            adults.append(user)
      }
}
```

You don't need to explain every line of code to us. Other devs should be able to read the code and understand what it is doing

Better Practice:

```
let isLegalAdult = user.age >= 18
```

```
if isLegalAdult {
      adults.append(user)
}
```

Slightly better than the previous example. Devs unfamiliar with the code can easily see that if the user is a legal adult then they are included in the adults list.

Bad Practice:

```
func calculateTip(bill: Double) -> Double {
    if bill > 20 {
        return bill * 0.15
    } else {
        return 5.0
    }
}
```

We don't know why we are doing any checks or comparisons

Better Practice:

```
func calculateTip(bill: Double) -> Double {
      // For orders under $20, we enforce a $5 minimum tip to account for driver gas cots.
      let minimumTipThreshold = 20.0

      if bill > minimumTipThreshold {
            return bill * 0.15 // Standard 15% rate
      } else {
            return 5.0
      }
}
```

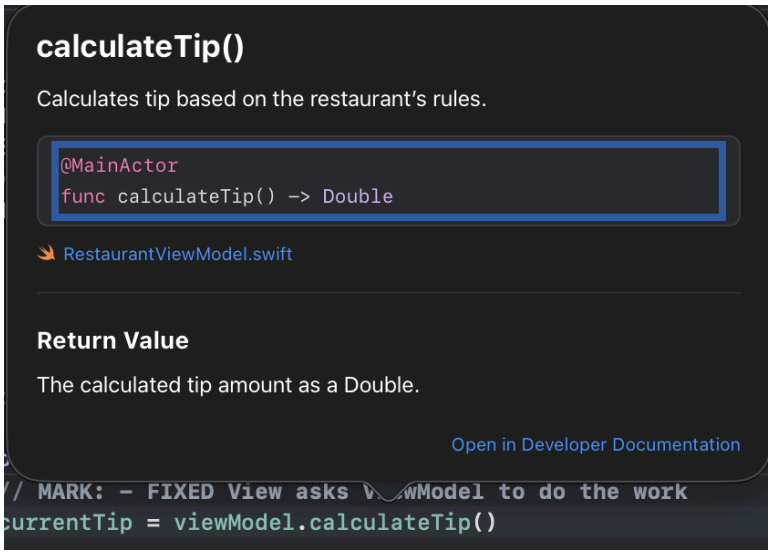
Function Documentation

Use `///` for documenting functions and properties.
Really try to do this **especially if other people will be using your function.** Can use

`option` + `left click` to easily see the description of the function and `command` + `left click` to jump to the function

```
/// Calculates the total cost including tax.
/// - Parameter subtotal: The raw price of items.
/// - Returns: The final price formatted as a Double.
func calculateTotal(for subtotal: Double) -> Double { ... }
```

Format:
1st Line: Description of the function
Parameters: Description of arguments passed in
Returns: Description of return value



Mark Comments

Use `// MARK: - Section Name` to organize file. It creates a separator line in Xcode minimap and a jump bar, making navigation easier.

Visibility Modifiers

Add `**private**` to everything.

- reduces "surface area" of your code

- Makes it easier to understand what is being used by other classes versus what is safe to change.

`**private**`: make everything private when you first create it
`**fileprivate**`: limits access to the entire source file.
`**internal**`: the default. Used for main function of your app that is accessed by ViewModels
`**public**`: Anywhere, including modules that import this one.

Make private and utilize “get-ers” and “set-ers” for everything

Other Practices

- Try to limit Force Unwrapping (!): Using `!` can cause unexpected app crashes. Instead use `if let` or `guard let` to unwrap this optionals. This allows an informative error statement to be printed rather than causing a crash.
- Avoid hardcoding numbers: This makes it more difficult to understand code and perform updates
  Bad: `if password.count > 8`
  Good: `let minPasswordLength = 8; if password.count > minPasswordLength`

1 Page Recap / Summary (Can put this in Readme)

1. MVVM Architecture (The Structure)

Goal: Separation of Concerns. The UI shouldn't know about API calls; the Logic shouldn't know about UI.

- **View (The Face):**
  - *Role:* Layout and appearance only.
  - *Action:* Listens to the ViewModel. Displays data.
- **ViewModel (The Brain):**
  - *Role:* Holds the state (e.g., `isLoading`, `usersList`). Formats data for the View.
  - *Action:* Fetches data from Model, processes it, and publishes changes. **Never import UIKit/SwiftUI.**
- **Model (The Data):**
  - *Role:* Raw data structures and business logic.
  - *Action:* Interacts with databases/services.

---

2. Naming Conventions

Swift is a **CamelCase** language.

- **UpperCamelCase:** Classes, Structs, Enums, Protocols (`UserProfile`)
- **lowerCamelCase:** Variables, Functions, Constants (`userProfile`)

The Verb Dictionary (Function Naming)

- Working with Database: **Fetch, Delete**
- Working with on-app data: **Get, Remove**
- Other common terms: **Is, Update, Create**

---

3. Documentation Practices

Comments explain **WHY**, Code explains **WHAT**.

- **Avoid:** Explaining syntax or obvious flow
- **Preferred:** Self-documenting variable names.
  - *Bad:* `if user.age > 18 { ... }`
  - *Good:* `let isLegalAdult = user.age >= 18; if isLegalAdult {...}`
- **Preferred:** Contextual comments (`// Minimum tip is $5 to cover gas costs`).

Formal Documentation (SwiftDoc) Use `///` for shared functions.

Swift

```
/// Calculates total cost.
/// - Parameter subtotal: The raw price.
/// - Returns: Price with tax included.
func calculateTotal(for subtotal: Double) -> Double { ... }
```

Navigation: Use `// MARK: - Section Name` to organize large files.

---

4. Safety & Cleanliness

- **No Force Unwrapping (`!`):** Avoid using `label.text!`. It causes crashes.
  - *Fix:* Use `if let` or `guard let` to handle nils gracefully.
- **No Magic Numbers:** Don't hardcode numbers in logic.
  - *Bad:* `if password.count > 8`
  - *Good:* `let minPasswordLength = 8; if password.count > minPasswordLength`