# 2: Multiplication Exercises II

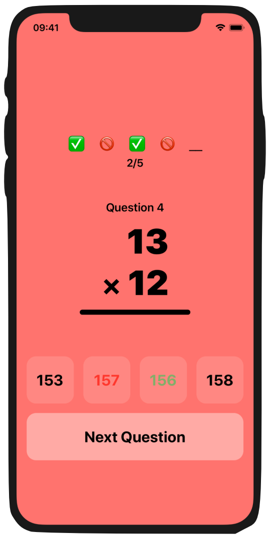
Due: Noon, 25 January

## Description

In this assignment you will add functionality to the views you created in Assignment 1. You will use the MVVM (Model-View-View Model) pattern using SwiftUI property wrappers for data binding. *This is a challenging assignment. Start early!*

Your app should have the following functionality:

- Upon pressing a button labeled "Start" a multiplication problem is presented by giving a multiplicand and a multiplier (each of which is a random number between 10 and 20, inclusive).
- Four buttons below the problem should display 4 possible answers, one of which is correct. The other three should be within 5 of the correct answer with no duplicates.
- Upon the user selecting an answer, the app should indicate the correct answer, indicate whether the user's answer was correct, and update a running total of correct answers.
- Upon pressing the button (now labeled "Next"), another problem should be presented and labels appropriately updated.
- The game continues until 5 problems have been answered, at which point the button should be labeled "Reset". Pressing the button at this point restarts the game.



## Walk Through

The most challenging part of this assignment could be deciding how to organize your code. The goal of the MVVM pattern is to separate functionality, reduce dependencies, and give each part of your code specific responsibilities. We have three parts to consider: the View, the View Model, and the Model. Although you can start with any of them I find it convenient for this application to start with the model since it is not dependent upon any other parts. However, your design will certainly be iterative. You'll come back to the model and make changes as you develop the other parts.

There are many ways this project can be organized using the MVVM principle. We're going to outline a reasonable one, but certainly not "the correct" one.

1. **Create your application's model.** Your model contains the data for your app. The data that your views will ultimately use to display information. The heart of this app is a math problem. Think about everything that is needed to describe a problem to display it (or to keep track of its result). The model should not import SwiftUI, just Foundation.

   1.1. Define a struct to represent a Math problem. Give it all the properties it needs to represent a problem. An enum for the answer status might be a good idea.

   1.2. Give it a custom initializer so an instance is ready to use (display). Consider what parameters to pass into it. Keep the model as generic as possible. For example, the range of possible values and the total number of possible answers should be provided via the initializer.

2. **Create the view model.** Your view model will create math problems, keep track of the game state, e.g., the current problem, record of results. It contains the logic of your game.

Both the model & views should be pretty dumb, just do their jobs. The view model contains the logic. I recommend keeping track of a game state with an enum. Your view model will need to import SwiftUI.

2.1. Create a new Swift file for the view model. Give it a name that describes its purpose. "GameManager" might be reasonable.

2.2. Define your view model as a class that conforms to the `ObservableObject` protocol. This will allow views that contain the view model as an `@ObservedObject` (or @StateObject, @EnvironmentObject) property to update when the view model changes.

- The view model should contain `@Published` properties for every piece of data for which the view needs to respond/update. These will always be value types (structs, arrays, scalars) that will tell SwiftUI that the view model changes whenever any of these change. Think about what the view will need. You'll figure this out as you revise your views to interact with the view model.
- It should also contain a game state (highly recommended). Think about the different states that your UI could be in and define an enum for them.
- The view model may also contain data related to the state of the game, e.g., current problem number, number answered correctly, etc. This kind of data could be stored together in its own struct (part of the model), but I think it makes more sense to keep it in the view model as individual components.

2.3. Define functions that the view will call in response to user interactions (e.g., pressing buttons).

- A function handling the "start" button should manage most of the game state. In fact, it might essentially be a switch statement involving the game state.
- A function handling the user's guess needs to update the view indicating whether the answer was correct, updating the display of answer results, etc. It will also update the game state.
- Mark any functions that are used internally (and that should not/will not be directly used by views) as `private`.

3. **Update your views.** Your views need to respond to changes in the view model and also need to send the view model requests (or intents or updates). Be sure you've created an instance of your view model using the @StateObject wrapper and added it as an environmentObject to the main view.

3.1. Your views' buttons should call functions in the view model (the "intents")

3.2. Views that should update in response to changes in the view model should get their content from *Published* properties in the view model. E.g., the current math problem (a struct value) should likely be a published property. The views displaying the current problem can access the data contained in this value.

3.3. Buttons should be enabled/disabled according to information provided by the view model, perhaps published Boolean properties.

4. **Clean up your project.**

4.1. Declare all constant values (other than 0 or 1) as let-constants in an appropriate place. Constants used in only one place (e.g., in a class) can simply be declared locally.

Constants used across data structures can be declared as static constants in their own struct. Constants used in the view (e.g., corner radii, fonts and colors) can be declared as constants, perhaps in a struct called `ViewConstants`.

4.2. Put your views in different files, unless they are very small and it is convenient to have them in the same file where they are used.

4.3. Organize your files into groups. You can create a group by right clicking a file in the file navigator and clicking “New Group.” Create groups titled “Models”, “ViewModels”, and “Views”. Drag your files into the appropriate group in the file navigator. Ignore the existing files (AppDelegate, SceneDelegate, etc.)

## Testing

Testing of the application is imperative. Be sure that all the required features have been implemented.

1. Your project should build without errors or warnings.
2. The app should have a clean, consistent and intuitive user interface. Buttons and other control elements should only be enabled/displayed when expecting user interaction. E.g., the *Next* button should be disabled/hidden after a problem has been displayed and the user has not yet selected an answer. Text should always be showing something appropriate (which might mean they are blank). Fonts and colors should be appropriately chosen.
3. Each of the requirements in the Description section above and each of the steps in the walkthrough will be considered to verify that you’ve completed the assignment correctly.

## Hints

Most assignments will have hints sprinkled throughout the assignment or in a separate section like this.

1. Use the documentation to look up primitive operations on common data types. No sense writing code when Swift already provides an answer. The Array type has many useful functions. (You don’t need to write your own code to shuffle array elements.)
2. You can use a closed range (`ClosedRange<Int>`) to specify a range of values for answers.
3. Swift’s `repeat-while` loop is useful for generating non-duplicated wrong answers.
4. Make sure that any stored properties in your view model that should trigger a rebuilding of your views are marked as `@Published`.
5. Keep your logic in the view model. The view should simply ask the view model (via functions or published properties) for anything it needs. Your model doesn’t need any details about the game, just what a problem is.
6. Remember to declare your view model as an @EnvironmentObject in all subviews of your main view.

7. Customize your app's appearance! The default button and fonts are quite plain. A background image or color might also contribute to a custom look. Use the `.background` and `.foregroundColor` modifiers.
8. Use special symbols in strings (like a ✅or ✘). Use ^⌘Space to bring up the special character viewer. Use string interpolation to use expressions in strings
9. Use the `.opacity` view modifier (perhaps with a ternary operator) with values 0.0 and 1.0 to conditionally toggle the visibility of an element without changing the layout of rest of the view hierarchy.
10. Use the .disabled view modifier on buttons to enable/disable them.

## Troubleshooting

Assignments may also have a section for troubleshooting to call out common pitfalls or areas to watch out for.

1. Error messages are typically pretty good, but sometimes the compiler gets very confused with Views that are not properly formed. You'll get an obscure error about the body variable not conforming to the protocol. I've encountered cases where it was due to a misspelling.
2. If Xcode tells you it can't display the preview in the Canvas try restarting Xcode.

## Submission

Prepare your project for submission:

1. Remove all breakpoints, print statements and any other cruft.
2. Make sure your project compiles cleanly from scratch: choose **Clean** from the Product menu in Xcode, then Build and Run your program. Be sure to remove all warnings.
3. Check that all resources were copied into your project folder.
4. Ensure that you have merged your work back onto the master branch**. Commit and push your project**
5. If you are able, use the continuous integration process to verify your project on the server builds.