

Assignment I:

Memorize

Objective

The goal of this assignment is to recreate the demonstration given in the first two lectures and then make some small enhancements. It is important that you understand what you are doing with each step of recreating the demo from lecture so that you are prepared to do those enhancements.

Mostly this is about experiencing the creation of a project in Xcode and typing code in from scratch. **Do not copy/paste any of the code from anywhere.** Type it in and watch what Xcode does as you do so.

Be sure to review the Hints section below!

Also, check out the latest in the Evaluation section to make sure you understand what you are going to be evaluated on with this assignment.

Due

This assignment is due in one week. You should finish this assignment before you start watching Lecture 3.

Materials

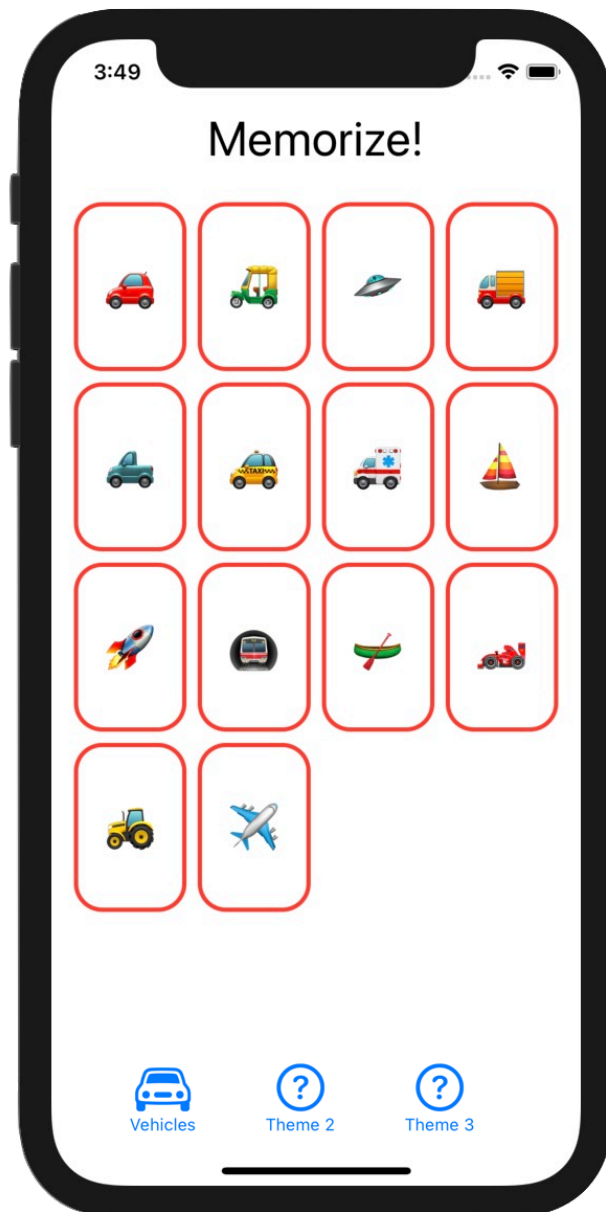
- You will need to install the (free) program Xcode 12 using the App Store on your Mac (previous versions of Xcode will not work). It is highly recommended that you do this immediately so that if you have any problems getting Xcode to work, you have time to get help from Piazza.
- In order to recreate the demo, you will certainly need to watch the first two lectures (see Piazza for all course materials).
- You will also need the SF Symbols application which can be downloaded from <https://developer.apple.com/sf-symbols>.

Required Tasks

1. ~~Get the Memorize game working as demonstrated in lectures 1 and 2. Type in all the code. Do not copy/paste from anywhere.~~
2. ~~You can remove the ⊖ and ⊕ buttons at the bottom of the screen.~~
3. ~~Add a title “Memorize!” to the top of the screen.~~
4. ~~Add at least 3 “theme choosing” buttons to your UI, each of which causes all of the cards to be replaced with new cards that contain emoji that match the chosen theme. You can use Vehicles from lecture as one of the 3 themes if you want to, but you are welcome to create 3 (or more) completely new themes.~~
5. ~~The number of cards in each of your 3 themes should be different, but in no case fewer than 8.~~
6. The cards that appear when a theme button is touched should be in an unpredictable (i.e. random) order. In other words, the cards should be shuffled each time a theme button is chosen.
7. ~~The theme-choosing buttons must include an image representing the theme and text describing the theme stacked on top of each other vertically.~~
8. ~~The image portion of each of the theme-choosing buttons must be created using an SF Symbol which evokes the idea of the theme it chooses (like the car symbol and the Vehicles theme shown in the Screenshot section below).~~
9. ~~The text description of the theme-choosing buttons must use a noticeably smaller font than the font we chose for the emoji on the cards.~~
10. ~~Your UI should work in portrait or landscape on any iPhone. This probably will not require any work on your part (that’s part of the power of SwiftUI), but be sure to experiment with running on different simulators in Xcode to be sure.~~





Screenshot

1. Screenshots are only provided in this course to help if you are having trouble visualizing what the Required Tasks are asking you to do. Screenshots are **not** part of the Required Tasks themselves (i.e. your UI does **not** have to look exactly like what you see below).



Hints

1. Economy is valuable in coding. The easiest way to ensure a bug-free line of code is not to write that line of code at all.
2. You will almost certainly want to make the `emojis var` in your `ContentView` be `@State` (since you're going to be changing the contents of this `Array` as you choose different themes and shuffle cards).
3. You might have to pick your themes based on what symbols you're able to find in SF Symbols! There is a much wider variety of emoji to choose from in the universe than there are SF Symbols to choose from.
4. Shuffling the cards might be easier than you think. Be sure to familiarize yourself with the documentation for `Array`. Note that there are some seemingly identical functions in `Array`, one of which is a verb and other is an adjective that is the past-tense of that verb. Try to figure out the difference (though you can use either one). In Swift, we generally prefer using the "past tense verb form" version. You'll find out why next week.
5. Other than reviewing the documentation for `Array`, you are not expected to use any aspect of Swift/SwiftUI that was not shown in lecture (though you are welcome to try to if you want!). You'll be doing *exactly* the same sorts of things we did in lecture.
6. Required Task 6 only says the cards need to appear in random order *when a theme button is pressed*. It is perfectly fine if your application *launches* with exactly the same theme and cards in exactly the same order each time. We haven't learned yet how to set up the way a `View` looks when it first appears, so you can hardwire that as needed for this assignment.
7. You can control the size of your SF Symbol images using `.font()`. SF Symbols are often interleaved with surrounding `Text` and so `Image(systemName:)` conveniently adjusts the size of the `Image` depending on the `.font()` it is modified with. It's probably a good idea to use `.largeTitle` for these (but not for the text captions underneath them since Required Task 9 prohibits that).
8. Don't forget that you can add `View` modifiers (like `.font()`, for example) either in the Inspector on the right-hand side of Xcode's screen or by directly typing in the code to call the function.
9. Give some thought to how your theme-choosing buttons and their associated text are aligned relative to each other, especially if the SF Symbols you choose are of varying heights, for example. This is not a Required Task, but a good solution will consider this. In lecture, we did briefly see how to align things that are stacked together.

10. We're not looking for super-clean Swift code in this assignment (because you barely know anything about the language!). So, for example, if you end up having numerous array literals like [", ", ", ""] peppered about your code (even if you end up duplicating the same one in two different places), that's okay. We'll learn how to handle constants like this next week.
 11. A great way to help verify Required Task 10 is to add more iPhone devices to your Preview pane (just like we did for dark mode in the demo). Give it a try!
 12. If you really want to test yourself this week, check out the “Extra Credit” below!
-

Things to Learn

Here is a partial list of concepts this assignment is intended to let you gain practice with or otherwise demonstrate your knowledge of.

1. Xcode 12
 2. Swift 5.4
 3. Writing code in the in-line function that supplies the value of a `View`'s `body` `var`
 4. Syntax for passing closures (aka in-line functions) (i.e. code in `{ }`) as arguments
 5. Understanding the syntax of a `ViewBuilder` (e.g. “bag of Lego”) function
 6. Using basic building block `Views` like `Text`, `Button`, `Spacer`, etc.
 7. Putting `Views` together using `VStack`, `HStack`, etc.
 8. Modifying `Views` (using `.font()`, etc.)
 9. Using `@State` (we'll learn much more about this construct later, by the way)
 10. Very simple use of `Array`
 11. Using a `Range` (e.g. `0..emojiCount`) as a subscript to an `Array`
 12. The SF Symbols application
 13. Putting system images into your UI using `Image(systemName:)`
 14. Looking things up in the documentation (`Array` and possibly `Font`)
 15. `Int.random(in:)` (Extra Credit)
 16. Running your application in different simulators
-

Evaluation

In all of the assignments this quarter, writing quality code that builds without warnings or errors, and then testing the resulting application and iterating until it functions properly is the goal.

Here are the most common reasons assignments are marked down:

- Project does not build.
- One or more items in the Required Tasks section was not satisfied.
- A fundamental concept was not understood.
- Project does not build without warnings.
- Code is visually sloppy and hard to read (e.g. indentation is not consistent, etc.).
- Your solution is difficult (or impossible) for someone reading the code to understand due to lack of comments, poor variable/method names, poor solution structure, long methods, etc.

Often students ask “how much commenting of my code do I need to do?” The answer is that your code must be easily and completely understandable by anyone reading it. You can assume that the reader knows the SwiftUI API and knows how the Memorize game code from lectures 1 and 2 works, but should not assume that they already know your (or any) solution to the assignment.

Extra Credit

Here are some additional ways to challenge yourself ...

1. Make a random number of cards appear each time a theme button is chosen. The function `Int.random(in: Range<Int>)` can generate a random integer in any range, for example, `let random = Int.random(in: 15...75)` would generate a random integer between 15 and 75 (inclusive). Always show at least 4 cards though.
2. Try to come up with some sort of equation that relates the *number of cards* in the game to the width you pass when you create your `LazyVGrid`'s `GridItem(.adaptive(minimum:maximum:))` such that each time a theme button is chosen, the `LazyVGrid` makes the cards as big as possible without having to scroll.

For example, if 8 cards are shown, the cards should be pretty big, but if 24 cards are shown, they should be smaller. The cards should still have our 2/3 aspect ratio.

It doesn't have to be perfect either (i.e. if there are a few extreme combinations of device size (e.g. iPod touch for example) and number of cards, punting to scrolling is okay). The goal is to make it noticeably better than always using 65 is.

It's probably impossible to pick a width that makes the cards fit just right in both Portrait and Landscape, so optimize for Portrait and just let your `ScrollView` kick in if the user switches to Landscape.

Your "equation" can include some `if-else`'s if you want (i.e. it doesn't have to be a single purely mathematical expression) but you don't want to be special-casing every single number from 4 to 24 cards or some such. Try to keep your "equation" code *efficient* (i.e. not a lot of lines of code, but still works pretty well in the vast majority of situations).

The type of the arguments to `GridItem(.adaptive(minimum:maximum:))` is a `CGFloat`. It's just a normal floating point number that we use for drawing. You know what kind of results 65 gives you, so you're going to have to experiment with other numbers up and down from there.

We haven't covered `func`tions yet, but you likely would want to put your calculation in a `func`. If so, you'd have to figure that out on your own. Your reading assignment covers `func` syntax in detail of course, but you probably just want something like this: `func widthThatBestFits(cardCount: Int) -> CGFloat`.

When you do this, it becomes even more obvious that we really want the font we use to draw the emoji to scale with the size of the cards. We'll learn to do that next week or the week after, so there's nothing to do on that front this week.

Finally, what you'll really come to understand is that the "equation" we need is actually dependent on *the size of the area we have to draw the cards in*. That's also something we'll find out more about in lecture in the next week or so.