Assignment II: More Memorize

Objective

The goal of this assignment is to continue to recreate the demonstrations given in the first four lectures and then make some bigger 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.

This continues to be 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. It is also due before you watch Lecture 5. Don't click on the link for the video for Lecture 5 until after you've submitted this assignment!

Materials

- Xcode (as explained in assignment 1).
- You must watch lectures 1, 2, 3 & 4 before doing this assignment since Required Task 1 requires you to duplicate what is done there.

Required Tasks

1. Get the Memorize game working as demonstrated in lectures 1 through 4. Type in all the code. Do not copy/paste from anywhere.
2. If you're starting with your assignment 1 code, remove your theme-choosing buttons and (optionally) the title of your game.
3. Add the formal concept of a "Theme" to your Model. A Theme consists of a name for the theme, a set of emoji to use, a number of pairs of cards to show, and an appropriate color to use to draw the cards.
4. At least one Theme in your game should show *fewer* pairs of cards than the number of emoji available in that theme.
5. If the number of pairs of emoji to show in a Theme is fewer than the number of emojis that are available in that theme, then it should not just always use the first few emoji in the theme. It must use *any* of the emoji in the theme. In other words, do not have any "dead emoji" in your code that can never appear in a game.
6. Never allow more than one pair of cards in a game to have the same emoji on it.
7. If a Theme mistakenly specifies to show more pairs of cards than there are emoji available, then automatically reduce the count of cards to show to match the count of available emoji.
8. Support at least 6 different themes in your game.
9. A new theme should be able to be added to your game with a single line of code.
10. Add a "New Game" button to your UI (anywhere you think is best) which begins a brand new game.
11. A new game should use a *randomly chosen theme* and touching the New Game button should repeatedly keep choosing a new random theme.
12. The cards in a new game should all start face down.
13. The cards in a new game should be fully shuffled. This means that they are not in any predictable order, that they are selected from *any* of the emojis in the theme (i.e. Required Task 5), and also that the matching pairs are not all side-by-side like they were in lecture (though they can *accidentally* still appear side-by-side at random).
14. Show the theme's name in your UI. You can do this in whatever way you think looks best.
15. Keep score in your game by penalizing 1 point for every *previously seen* card that is involved in a *mismatch* and giving 2 points for every *match* (whether or not the cards involved have been "previously seen"). See Hints below for a more detailed explanation. The score is allowed to be negative if the user is bad at Memorize.
16. Display the score in your UI. You can do this in whatever way you think looks best.

Hints

1. Economy is still (and is always) valuable in coding.
2. Your ViewModel's connection to its Model can consist of more than a single `var model`. It can be any number of `var`s. The "Model" is a *conceptual* entity, not a single `struct`.
3. A Theme is a completely separate thing from a `MemoryGame` (even though both are part of your application's Model). You should not need to modify a single line of code in `MemoryGame.swift` to support themes!
4. Since a Theme is now part of your Model, it must be UI-independent. Representing a color in a UI-independent way is surprisingly nuanced (not just in Swift, but in general). We recommend, therefore, that you represent a color in your Theme as a simple `String` which is the name of the color, e.g. "orange". Then let your ViewModel do one of its most important jobs which is to "interpret" the Model for the View. It can provide the View access to the current theme's color in a UI-*dependent* representation (like SwiftUI's `Color struct`, for example).
5. You don't have to support every named color in the world (a dozen or so is fine), but be sure to do something sensible if your Model contains a color (e.g. "fuchsia") that the ViewModel does not know how to interpret.
6. We'll learn a better (though still not perfect) way to represent a color in a UI independent fashion later in the quarter.
7. Required Task 6 means that, for example, a Halloween game should never have four 🎃 cards.
8. Required Task 7 means that, for example, if a theme's emojis are `["👻","🎃","🕷"]` and the number of pairs to show in the theme is 47, you must automatically reduce that 47 to 3.
9. You might find `Array`'s `randomElement()` function useful in this assignment (though note that this function (understandably) returns an Optional, so be prepared for that!). This is just a Hint, not a Required Task.
10. There is no requirement to use an Optional in this assignment (though you are welcome to do so if you think it would be part of a good solution).
11. You'll very likely want to keep the `static func createMemoryGame()` from lecture to create a new memory game. But that function needs a little bit more information to do its job now, so you will almost certainly have to add an argument to it.
12. On the other hand, you obviously *won't* need the `static let emojis` from last week's lecture anymore because emojis are now obtained from whatever the current Theme is.

13. It's quite likely that you will need to add an `init()` to your ViewModel. That's because you'll probably have one `var` whose initialization depends on another `var`. You can resolve this kind of catch-22 in an `init()` because, in an `init()`, you can control the *order* in which `var`s get initialized (whereas, when you use property initializers to initialize `var`s, the order is undetermined, which is why property initializers are not allowed to reference other `var`s).
14. The code in your ViewModel's `init()` might look very, very similar to the code involved with your new game mechanism since you obviously want to start a new game in both of these places. Don't worry if you end up with some code duplication here (you probably don't quite know enough Swift yet to factor this code out).
15. You might well have to shuffle two *different* `Array`s in this assignment. This is just a Hint, not a Required Task.
16. An amazing thing about "in-line functions" (actually called "closures") in Swift is that if you declare a local variable in the scope that contains a closure, *the closure can use that variable*! For example, if `foo` below is a function that takes a function `() -> Void` as an argument, then …

```
let greetings = ["Hello","Howdy","Heya"].shuffled()
foo {
    print(greetings) // this is legal! greetings is usable here!
}
```

    This might come in handy for Required Task 5.
17. Make sure you think carefully about where all of your code lives (i.e. is it in the View, or in the ViewModel, or in the Model?). This assignment is mostly about MVVM, so this is very important to get right.
18. We're not making this a Required Task (yet), but try to put the keyword `private` or `private(set)` on any variables where you think it would be appropriate.

19. A card has "already been seen" only if it has, at some point, been face up *and then is turned back face down*. So tracking "seen" cards is probably something you'll want to do when you turn a card that is face up to be face *down*.

20. If you flipped over a 🐠 + 👻, then flipped over a ✏️ + 🏀, then flipped over two 👻s, your score would be 2 because you'd have scored a match (and no penalty would be incurred for the flips involving 🐠, ✏️ or 🏀 because they have not (yet) been involved in a *mismatch*, nor was the 👻 ever involved in a mismatch). If you then flipped over the 🐠 again + 🐼, then flipped 🏀 + 🐠 once more, your score would drop 3 full points down to -1 overall because that 🐠 card had already been seen (on the very first flip) and subsequently was involved in two separate mismatches (scoring -1 for each mismatch) and the 🏀 was mismatched after already having been seen (-1). If you then flip 🐠 + the other 🐠 that you finally found, you'd get 2 points for a match and be back up to 1 total point.

21. The "already been seen" concept is about specific *cards* that have already been seen, not emoji that have been seen.

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. MVVM
2. Intent functions
3. `init` functions
4. Type Variables (i.e. `static`)
5. Access Control (i.e. `private`)
6. `Array`
7. Closures

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 iOS API and knows how the Memorize game code from the lectures works, but should not assume that they already know your (or any) solution to the assignment.

Extra Credit

We try to make Extra Credit be opportunities to expand on what you've learned this week. Attempting at least some of these each week is highly recommended to get the most out of this course.

If you choose to tackle an Extra Credit item, mark it in your code with comments so your grader can find it.

1. When your code creates a Theme, allow it to default to use all the emoji available in the theme if the code that creates the Theme doesn't want to explicitly specify how many pairs to use. This will require adding an `init` or two to your Theme `struct`.
2. Allow the creation of some Themes where the number of pairs of cards to show is not a specific number but is, instead, *a random number*. We're not saying that *every* Theme now shows a random number of cards, just that *some* Themes can now be created to show a random number of cards (while others still are created to show a specific, pre-determined number of cards).
3. Support a gradient as the "color" for a theme. Hint: `fill()` can take a `Gradient` as its argument rather than a `Color`. This is a "learning to look things up in the documentation" exercise.
4. Modify the scoring system to give more points for choosing cards more quickly. For example, maybe you get max(10 - (number of seconds since last card was chosen), 1) x (the number of points you would have otherwise earned or been penalized with). (This is just an example, be creative!). You will definitely want to familiarize yourself with the `Date struct`.