Assignment II:

Set

Objective

The goal of this assignment is to give you an opportunity to create your first app completely from scratch by yourself. It is similar enough to assignment 1 that you should be able to find your bearings, but different enough to give you the full experience!

Since the goal here is to create an application from scratch, **do not start with your assignment 1 code, start with New → Project in Xcode**.

This assignment must be submitted using the submit script described here by the start of lecture next Wednesday (i.e. before lecture 6). You may submit it multiple times if you wish. Only the last submission before the deadline will be counted.

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.

Materials

• You will want to review the rules to the game of Set.

Required Tasks

1. Implement a game of solo (i.e. one player) Set.

2. Have room on the screen for at least 24 Set cards. All cards are always face up in Set.

3. Deal 12 cards only to start. They can appear anywhere on screen (i.e. they don't have to be aligned at the top or bottom or anything; they can be scattered to start if you want), but should not overlap.

4. You will also need a "Deal 3 More Cards" button (as per the rules of Set).

5. Allow the user to select cards to try to match as a Set by touching on the cards. It is up to you how you want to show "selection" in your UI. See Hints below for some ideas. Also support "deselection" (but when only 1 or 2 (not 3) cards are currently selected).

6. After 3 cards have been selected, you must indicate whether those 3 cards are a match or a mismatch (per Set rules). You can do this with coloration or however you choose, but it should be clear to the user whether the 3 cards they selected match or not.

7. When any card is chosen and there are already 3 **non-matching** Set cards selected, deselect those 3 non-matching cards and then select the chosen card.

8. As per the rules of Set, when any card is chosen and there are already 3 **matching** Set cards selected, replace those 3 matching Set cards with new ones from the deck of 81 Set cards (again, see Set rules for what's in a Set deck). If the deck is empty then matched cards can't be replaced, but they should be hidden in the UI. If the card that was chosen was one of the 3 matching cards, then no card should be selected (since the selected card was either replaced or is no longer visible in the UI).

9. When the Deal 3 More Cards button is pressed either a) replace the selected cards if they are a match or b) add 3 cards to the game.

10. The Deal 3 More Cards button should be disabled if there are a) no more cards in the Set deck or b) no more room in the UI to fit 3 more cards (note that there is always room for 3 more cards if the 3 currently-selected cards are a match since you replace them).

11. Instead of drawing the Set cards in the classic form (we'll do that next week), we'll use these three characters ▲ ● ■ and use attributes in `NSAttributedString` to draw them appropriately (i.e. colors and shading). That way your cards can just be `UIButton`s. See the Hints for some suggestions for how to show the various Set cards.

12. Use a method that takes a closure as an argument as a meaningful part of your solution. You cannot use one that was shown in lecture.

13. Use an `enum` as a meaningful part of your solution.

14. Add a sensible `extension` to some data structure as a meaningful part of your solution. You cannot use one that was shown in lecture.

15. Your UI should be nicely laid out and look good (at least in portrait mode, preferably in landscape as well, though not required) on any iPhone 7 or later device. This means you'll need to do some simple Autolayout with stack views.

16. Like you did for Concentration, you must have a New Game button and show the Score in the UI. It is up to you how you want to score your Set game. For example, you could give 3 points for a match and -5 for a mismatch and maybe even -1 for a deselection. Perhaps fewer points are scored depending on how many cards are on the table (i.e. how many times Deal 3 More Cards has been touched). Whatever you think best evaluates how well the player is playing.

Hints

1. You can use the same UI layout mechanism we used in Concentration (i.e. stack views). Early in the game, some of the buttons will start out not showing a card (since there are only 12 to start and you must have enough room to show 24) and late in the game, there will be buttons representing matched cards that couldn't be replaced. Treat all of those just like a "matched and removed" card from Concentration is treated (i.e. the button is there, but is invisible to the user).

2. Note that you are not required to align the cards when there are fewer than the max showing, so the random positioning of the elements of an Outlet Collection is not a problem this week. We'll be fixing this next week with a much better UI architecture.

3. A couple of really great methods in `Array` are `index(of:)` and `contains()`. But they only work for `Array`s of things that implement the `Equatable` protocol (like `Int` and `String` do). If you have a data type of your own that you want to put in an `Array` and use `index(of:)` and `contains()` on, just make your data type implements `Equatable`.

4. We kept track of the face up and matched states in Concentration in our `Card`s. While this was great for demonstrating how mutability works in a value type, it might not have been the best architecture. Having data structures that are completely immutable (i.e. have have no `var`s, only `let`s) can make for very clean code. For example, in your Set implementation, it'd probably be just as easy to keep a list of all the selected cards (or all the already-matched cards) as it would be to have a `Bool` in your Set Card data structure. And you might find the code is much simpler too.

5. If you use the approach in Hint 4, you'll almost certainly want to pay attention to Hint 3, though.

6. You can show selection using the `UIButton`'s `backgroundColor` if you want, but `UIKit` also knows how to put a *border* around any `UIView` (including a `UIButton`) with code like this (which would draw a 3 points wide border in blue, for example):

    ```
    button.layer.borderWidth = 3.0
    button.layer.borderColor = UIColor.blue.cgColor
    ```

7. You can also round the corners of your button if you want using a similar mechanism:

    ```
    button.layer.cornerRadius = 8.0
    ```

8. If you want a character in an `NSAttributedString` to be filled in, you specify an `NSAttributedStringKey.strokeWidth` which is a negative number.

9. For the "striped" look of a Set card, just use an `NSAttributedStringKey.foregroundColor` of about 15% alpha (created with the `UIColor` method `withAlphaComponent`). A 100% alpha `foregroundColor` can be used for the "filled" look and a positive `strokeWidth` for the "outline" look.

10. Other than the two `NSAttributedStringKey`s above, you will probably only need `NSAttributedStringKey.strokeColor`.

11. You can use whatever colors you want for your UI (i.e. you don't have to use the "standard" Set colors).

12. Be careful which font you choose for your Set card buttons. Some fonts have those three shapes (▲●■) be different sizes. The `systemFont` seems to have them all the same size.

13. Be sure to use exactly these three Unicode characters: ▲●■. Some other, similar shape characters don't fill or stroke quite right for our purposes.

14. Note that a `UIButton`'s `title` and its `attributedTitle` can both be set separately (`attributedTitle` takes precedence if set) so, for example, if you wanted a `UIButton` to have no title at all, you'd want to set **both** of those to `nil`.

15. `NSAttributedStringKey`s like `foregroundColor` are not appropriate for use in a Model (note that a color is a `UIColor`, which means it's a UI thing). Do not use `NSAttributedString`s in your Model.

16. It'd probably good MVC design **not** to hardwire specific color names or shape names (like diamond or oval or green or striped) into the property names in your Model code. As you can see with this assignment (where we're using ▲●■ instead of the standard shapes and shading instead of striping, etc.), the colors, shapes, etc., are really a **UI concept** and have nothing to do with the **Model**.

17. Next week we won't be using attributed strings at all but if you design your Model correctly this week, **your Model should not have to change by even a single line of code**. Give some thought to making your Model have just the right API to communicate what is going on in the game, but not make any assumptions about how the game is being presented to the user.

18. A Set Game has a list of cards that are being played, it has some selected cards, it knows whether the currently selected cards are a match or not, it has a deck of cards from which it is dealing, and it probably wants to keep track of which cards have already been matched. That's pretty much it. Your Model's API should present those concepts cleanly. The only actual functionality your Model has is selecting cards to try to match and dealing three new cards on demand (because those are the fundamental concepts of the Set game).

19. Preventing adding three more cards when the UI is "full" is a UI thing (not a Model thing). Your test to see if the Deal 3 More Cards button is enabled should be completely in the UI. The Model has no concept of "no more cards fit in the UI" since it knows nothing about the UI. In other words, Required Task 9 is a Model task and Required Task 10 is a Controller task.

20. Be careful to test your "end game." When the deck of Set cards runs out, successfully matched cards can no longer be replaced with new cards. Those un-replaced matched cards can't appear in the UI (otherwise users might try to match them

against other cards!). For this reason, your Model's API will have to reveal which cards have already been successfully matched.

21. Remember that your Model doesn't actually note successfully matched cards as matched until the next card selection happens or "Deal 3 More Cards" happens (which is fine, you don't want to hide matched cards from the UI until after the user has had a chance to see that he or she has made a successful match anyway).

22. For testing your end-game, you'll probably want to temporarily trick your Model into thinking that **any** 3 cards are a match so you can get to the end-game quickly (unless you're really, really good at Set!).

23. In terms of scope, you should certainly be able to implement your Model in under 100 lines of code (not counting comments or a curly brace on a line by itself). In fact, it can be accomplished in significantly fewer than that. Your UI (i.e. your `ViewController`) is probably of similar scope. If you start to need much more than 200 lines of code total to implement the Required Tasks in this assignment, you might have taken a wrong turn somewhere.

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. All the things from Assignment 1, but from scratch this time.
2. Closures
3. `extension`
4. Using `struct` to declare constants
5. `Equatable`
6. `enum`

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.).

- Violates MVC.

- UI is a mess. Things should be lined up and appropriately spaced to "look nice."

- Improper object-oriented design including proper use of value types versus reference types.

- Improper access control (i.e. `private` not used appropriately).

- 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, but should <u>not</u> 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. How much Extra Credit you earn depends on the scope of the item in question.

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

1. Factor "speed of play" into the user's score. You can find out what time it is (to a fraction of a second) using the `Date` `struct`. The faster the user finds Sets, the higher his or her score should be. The penalty for mismatches probably needs to be high to discourage too much guessing.

2. Penalize pressing Deal 3 More Cards if there is a Set available in the visible cards. You'll have to write an algorithm to determine whether a pile of Set cards does, in fact, include at least one Set.

3. If you do write an algorithm to detect Sets, you could also add a "cheat" button that a struggling user could use to find a Set!

4. Or you could even have a "play against the iPhone" mode. Each time a set is found, start a random-length timer (you'll have to learn how to use `Timer.scheduledTimer(withTimeInterval:repeats:) { }` for this which uses a closure!) after which the iPhone picks a set if the user does not pick one first. See who can get the most sets! Maybe represent the iPhone with an emoji somewhere on the screen (🤔 while the timer is running, then maybe 😁 a couple of seconds before the iPhone makes a turn and 😂 if the iPhone wins or 😢 if not)? As always, be sure to give careful thought to your MVC architecture if you tackle this one.