Assignment III: Set

Objective

In this assignment, you will enhance your solution from Assignment 2 to add a second card game, Set, to your card matching game. You will also use a navigation controller to show the history of matches and mismatches for each game.

Be sure to check out the **Hints** section below!

Materials

- You will have to have successfully completed Assignment 2 and use it as a code base for this assignment.
- Check out how to play Set. You will not have to implement the full rules, but you should understand the conditions under which 3 cards make a Set and what a deck of Set cards consists of.

Required Tasks

1. Add a new MVC to your Matchismo solution from last week which plays a simple version of the card matching game Set. A good solution to this assignment will use object-oriented programming techniques to share a lot of code with your Playing Card-based matching game.
2. The Set game only needs to allow users to pick sets and get points for doing so (e.g. it does not redeal new cards when sets are found). In other words, *it works just like the Playing Card matching game*. The only differences are that it is a 3-card matching game and uses different cards (deal your Set cards out of a complete Set deck).
3. Your Playing Card game must continue to work as required from last week except that it only needs to work as a 2-card matching game (you can remove the switch or segmented control from your UI, but keep your 3-card matching infrastructure because Set is a 3-card matching game).
4. Use a `UITabBarController` to present the two games in your UI in separate tabs.
5. Just to show that you know how to do this, your Set game should have a different number of cards on the table than your Playing Card game (either game can have however many cards of whatever aspect ratio you think is best for your UI).
6. 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).
7. Both games must show the score somewhere and allow the user to re-deal.
8. Your Set game should also report (mis)matches like Required Task #5 in the last assignment, but you'll have to enhance this feature (to use `NSAttributedString`) to make it capable of working for both the Set and Playing Card games.
9. Last week, there was an Extra Credit task to use a `UISlider` to show the history of the currently-being-played game. This week we're going to make showing the history a Required Task, but instead of using a slider, you must invent yet another new MVC which displays the history in a `UITextView` and which is segued to inside a `UINavigationController`. It should show the cards involved in every match or mismatch as well as how many points were earned or lost as a result (you are already showing this information in a `UILabel` for each card choosing, so this should be rather straightforward to implement). Add a bar button item "History" on the right side of the navigation bar which performs the push segue. This feature must work for both the Playing Card game and the Set game.

Hints

These hints are not required tasks. They are completely optional. Following them may make the assignment a little easier (no guarantees though!).

1. Don't forget that you can specify alpha when you create a text color for a character in an `NSAttributedString`. That's a good way to do "shading." Don't forget that you can also set the stroke width of a character appropriately to "outline" it if necessary.
2. Set cards are unlike Playing Cards in that, when they are not chosen, they are not face down. It is up to you to pick a good "look" for a Set card when it is chosen versus not chosen. A simple way to do that would be with different button background images.
3. A Set card is too complicated to be represented by its `contents` `@property`. The `contents` for a Set card will probably always just be `nil`.
4. The three shapes we are using this week are circle, triangle and square, but we won't be using those shapes next week. In fact, you could imagine playing Set with any three distinct shapes. So maybe think about that in your API design? Ditto colors.
5. Certain fonts are designed with the circle, triangle and square being different sizes (why? no idea). While this doesn't look that great, it's okay for this week because we will be moving on to drawing the Set cards properly next week.
6. You can use the same UI elements as last week to re-deal and show the score if you want, but there are probably even better ways to do it given the new features you've added this week.
7. Your Model is UI independent, so it cannot have `NSAttributedString`s with UI attributes anywhere in its interface or implementation. Any attribute defined in UIKit is a UI attribute (obvious ones are those whose values are, for example, a `UIColor` or a `UIFont`). All the attributes discussed in lecture were UI attributes. While it would theoretically be legal to have an `NSAttributedString` *without* UI attributes in your Model, it is recommended you **not** do that for this assignment. Use `NSAttributedString` only in your Controller camp, not your Model camp.
8. If you violated MVC in your solution to Required Task 5 of Assignment 2, then Required Task 8 in this assignment will be more difficult (that's why you shouldn't have violated MVC!) and you'll probably want to go back and redo Required Task 5 of Assignment 2 with better MVC separation before doing Required Task 8 in this assignment.
9. If you subclass a subclass of `UIViewController`, you can wire up to the superclass's outlets and actions simply by manually opening up the superclass's code in the Assistant Editor in Xcode (side-by-side with the storyboard) and ctrl-dragging to it as you normally would. In other words, you are not required to make a superclass's outlets and actions public (by putting them in its header file) just to wire up to them

with ctrl-drag (it is quite possible to implement this entire assignment without making a single outlet or action public).

10. There is no concept like "protected" in Objective-C. Unfortunately, if a subclass wants to send messages to its superclass *in code* (not with ctrl-drag), those methods (including properties) will have to be made public. A good object-oriented design usually keeps publication of internal implementation to a minimum!
11. All methods (including properties) are inherited by subclasses regardless of whether they are public or private. And if you implement a method in a subclass, you will be overriding your superclass's implementation (if there is one) regardless of whether the method is public or private. As you can imagine, this could result in some unintentional overrides, but rarely does in practice.
12. If you copy and paste an *entire MVC scene* in your storyboard (not the components of it piece-by-piece, but the entire thing at once), then all the outlets and actions will, of course, be preserved (this can be quite convenient). Even if you then change the class of the Controller in one of the scenes, as long as the new class implements those outlets and actions (for example, by inheritance), the outlets and actions will continue to be preserved.
13. As you start working with multiple MVCs in a storyboard, you might get yourself into trouble by accidentally changing the name of an action or outlet or making a typo or otherwise causing your View to send messages to your Controller that your Controller does not understand. Remember from the first walkthrough of this course that you can *right click* on any object in your storyboard to see what it is connected to (i.e. what outlets point to it and what actions it sends) and you can also *disconnect* outlets and actions from there (by clicking the little X's next to the outlets and actions). If you are getting crashes that complain of messages being sent to objects that don't respond to that message (sometimes a method is referred to by the term "selector" by the way), this might be something to check.
14. There is no reason that the history-reporting MVC needs to be different for the two games. In fact, if you decide to use a different history-reporting MVC for each, you will want to justify your reasons for doing so in comments in your code.
15. Remember that every time you segue to a new MVC in a `UINavigationController`, it is an entirely *new instance* of that MVC. Also remember that that MVC is considered part of the pushing MVC's View (can only talk back to the Controller of the pushing MVC in a blind, structured way--luckily, there's no need to do that in this assignment).
16. The History MVC required task is mostly about creating a new MVC and how to segue to it (and only a tiny bit about using a `UITextView` to display text). Don't overthink the part of this which is actually displaying the attributed strings in the `UITextView`. Just use the code you already have in your `CardGameViewController` to generate these attributed strings. Focus on the transitioning between MVCs (`UINavigationController`, segueing, View Controller Lifecycle).

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. Using `NSAttributedString`.
2. Subclassing with Controllers.
3. Making code generic and reusable using OOP versus copying/pasting it.
4. Creating, loading and using images in Xcode 5.
5. Multiple MVCs in the same application.
6. Wiring up MVCs in a storyboard (`UINavigationController`/`UITabBarController`).
7. Segueing (`prepareForSegue:sender:`).
8. Using methods in the View Controller lifecycle.
9. Using `UITextView`.
10. Yet more experience with Objective-C and Xcode 5.

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.
- Project does not build without warnings.
- One or more items in the **Required Tasks** section was not satisfied.
- A fundamental concept was not understood.
- Code is 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, etc.
- Solution violates MVC.
- UI is a mess. Things should be lined up and appropriately spaced to "look nice." Xcode gives you those dashed blue guidelines so there should be no excuse for things not being lined up, etc. Get in the habit of building aesthetically balanced UIs from the start of this course.
- Assignment was turned in late (you get 3 late days per quarter, so use them wisely).
- Incorrect or poor use of object-oriented design principles. For example, code should not be duplicated if it can be reused via inheritance or other object-oriented design methodologies.

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 SDK, but should not assume that they already know the (or a) solution to the problem.

Extra Credit

Here are a few ideas for some things you could do to get some more experience with the SDK at this point in the game.

1. Create appropriate icons for your two tabs. The icons are 30x30 and are pure alpha channels (i.e. they are a "cutout"). Search the documentation for more on how to create icons like that and set them.

2. Add another tab to track the user's high scores. You will want to use `NSUserDefaults` to store the high scores permanently. The tab might want to show information like the time the game was played and the game's duration. It must also be clear which scores were Playing Card matching games and which scores were Set card matching games. Use attributes to highlight certain information (shortest game, highest score, etc.).

3. Include the ability to sort the scores shown in the Extra Credit above by last played, score or game duration.

4. Add yet another tab for some "settings" in the game (match bonuses, etc.).