Assignment II: Set

Objective

In this assignment, you will enhance your solution from Assignment 1 to add a second card game, Set, to your card matching game.

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

Materials

- You will have to have succesfully completed Assignment 1 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 tab bar controller to your application. One tab will be the game you built last week in Assignment 1. The other tab will be a new game, Set. Set is still a card game, so a good solution to this assignment will use object-oriented programming techniques to share a lot of code.
2. Don't violate any of the Required Tasks from Assignment 1 in the playing card game tab (in other words, don't break any non-extra-credit features from last week). The only exception is that your playing card game is required to be a 2-card-match-only game this week, so you can remove the switch or segmented control you added for Required Task #5 in Assignment 1. Your Set game is a 3-card matching game.
3. 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 your other card game (except that it is a 3 card (instead of 2 card) match with different kinds of cards).
4. Choose reasonable amounts to award the user for successfully finding a set (or incorrectly picking cards which are not a set).
5. Your Set game should have 24 cards.
6. Instead of drawing the 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. Your Set game should have a Deal button, Score label and Flips label just like your playing card matching game from Assignment 1 does.
8. Your Set game should also report (mis)matches like Required Task #3 in Assignment 1, but you'll have to enhance this feature (to use `NSAttributedString`) to make it work for displaying Set card matches.

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."
2. It might help to think of Set cards as having the symbols on *both* sides of the card, but the front side having a different background color or something than the back side (so you can visually see when it has been chosen). Having a Set game where you can't see the symbols until you flip the card over (like in our playing card game) might make it prohibitively difficult (plus that's not the way Set is played and you are supposed to have investigated how Set is played as part of this assignment).
3. You might not want your Set game's cards to have rounded rects around them (i.e. they could have no border if you want, just the symbols on the card). To get rid of the rounded rect around a button, change the button's type to the Custom type (just choose this in the Attributes Inspector for a `UIButton` instead of the Rounded Rect type) for your Set game's cards.
4. Setting the background color of a Rounded Rect button doesn't really make a lot of sense (it's not the interior area of the rounded rect, it's the *exterior* area). But setting the background color of a Custom button makes a lot of sense.
5. Once a set is found, you can use alpha to fade the button out (like we did in Assignment 1) or just clear out the set completely (i.e. cards in the Set game can be blank after they've been matched). It's hard enough for the user to get their minds around which cards match without the distraction of already-matched cards still visible (even if faded out).
6. The bonus for matches and penalty for mismatches were hardwired into your Controller in Assignment 1, but you'll want to make them parameterizable for Assignment 2 since you have 2 different games with different scoring.
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. Just use `NSAttributedString` to draw beautiful strings in your UI.
8. If you violated MVC in your solution to Required Task 3 of Assignment 1, 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 3 of Assignment 1 with better MVC separation before doing Required Task 8 in this assignment.

9. Speaking of Object-Oriented Programming, you'll want to use it to good effect in this assignment. You have two Controllers which share a lot of functionality. What do we do in OOP when we want to share functionality?

10. If you subclass a subclass of `UIViewController`, you can wire up to the superclass's outlets and actions simply by 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. If you want to send messages to the outlet in the subclass's *code*, though, you would obviously have to make the outlet public (it is good object-oriented design, however, to try to keep as much private as possible and only make public what is *essential* to the use or subclassing of a class--it is quite possible to implement this entire assignment without making a single outlet or action public).

11. There is no concept like "protected" in Objective-C. Unfortunately, if a subclass wants to send messages to its superclass *in code* (again, not with ctrl-drag), those methods (including properties) will have to be made public. Again, a good object-oriented design usually keeps publication of internal implementation to a minimum!

12. 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.

13. 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), all the outlets and actions will, of course, be preserved. 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.

14. 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 by 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.

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:

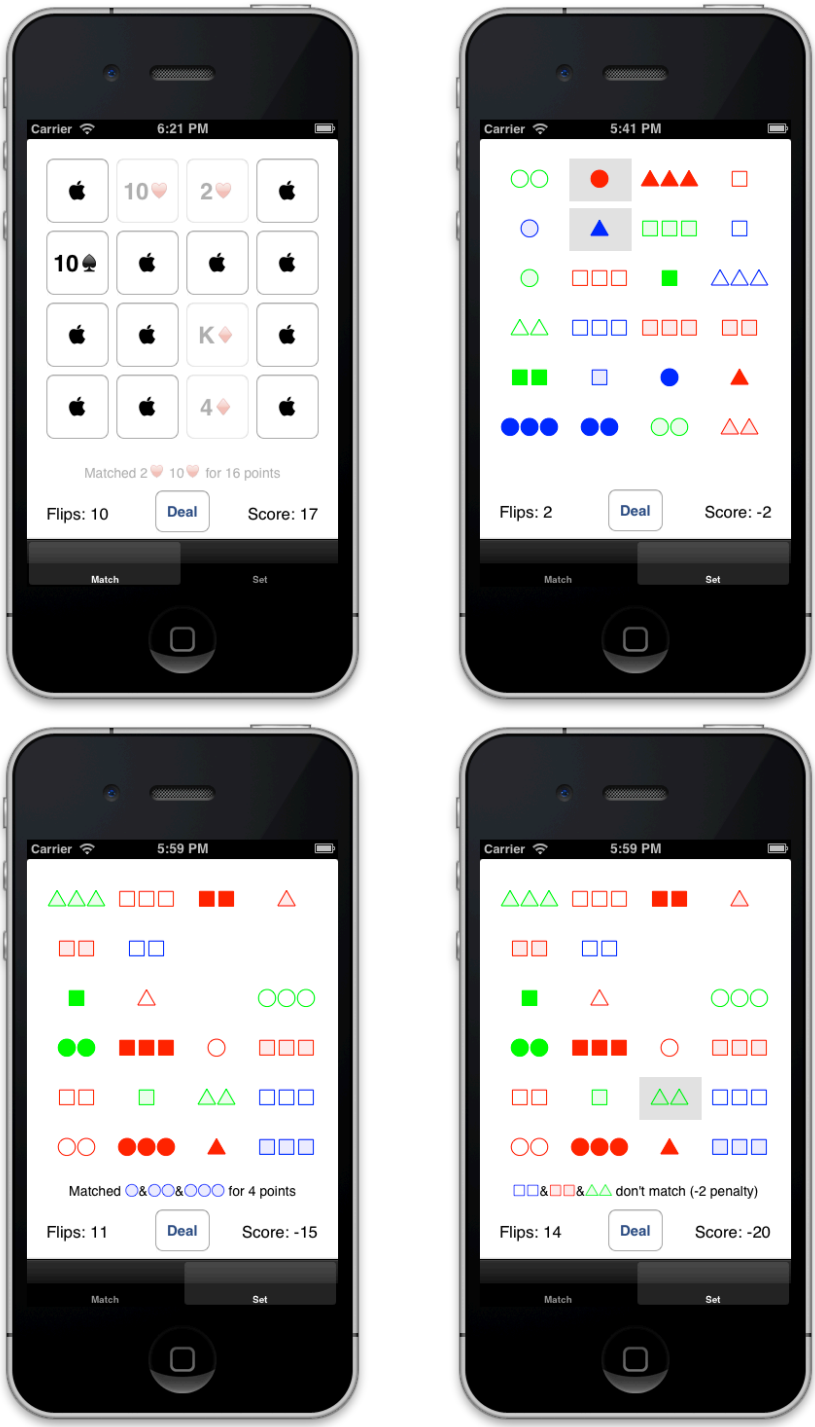
- Project does not build.
- Project does not build without warnings.
- One or more items in the **Required Tasks** section was not satisfied.
- A fundamental concept was not understood.
- Code is sloppy and hard to read (e.g. indentation is not consistent, etc.).
- Your solution is difficult (or impossible)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

Screen Shots

These screen shots are for example purposes only. Note carefully that *this* section of the assignment writeup is **not** under the Required Tasks section. In fact, screen shots like this are included in assignment write-ups only at the request of past students and over objections by the teaching staff. Do not let screen shots like this stifle your creativity!

The screen shots below have not implemented all Required Tasks yet so beware!



Extra Credit

Here are a couple of 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" through which the blue gradient shines through). Search the documentation for more on how to create icons like that and set them.

2. Add third tab to track the user's scores. It should be clear which scores were playing card match games and which scores were Set card match games.

3. Add another tab for some "settings" in the game.