CSCI S-65 Summer 2015

Assignment 2
Due: July 8th at 6:30PM
Total: 125 pts.

READINGS

A. **Additional reading to get a sense of App structure in general (not strictly required for this assignment, but start getting this in now so you can re-read it later.) (Ignore bits about Objective C.)**
https://developer.apple.com/library/ios/documentation/iPhone/Conceptual/iPhoneOSProgrammingGuide/TheAppLifeCycle/TheAppLifeCycle.html

B. **Delegate concepts (very short)**
https://developer.apple.com/library/ios/documentation/General/Conceptual/DevPedia-CocoaCore/Delegation.html

C. **Delegate technical discussion in more depth:**
https://developer.apple.com/library/ios/documentation/General/Conceptual/CocoaEncyclopedia/DelegatesandDataSources/DelegatesandDataSources.html

D. **Application Delegate in particular (In particular "Managing State Transitions".) (For now you can ignore the parts about opening URLs on startup.)**
https://developer.apple.com/library/ios/documentation/UIKit/Reference/UIApplicationDelegate_Protocol/

E. **Memory management (you should have found this already for the ARC question in assignment 1, but in case you didn't) (In particular closures)**
https://developer.apple.com/library/ios/documentation/Swift/Conceptual/Swift_Programming_Language/AutomaticReferenceCounting.html

F. **Computed properties (look down for this subheading) (should be mostly review)**
https://developer.apple.com/library/ios/documentation/Swift/Conceptual/Swift_Programming_Language/Properties.html

G. **Extensions (should be mostly review)**
https://developer.apple.com/library/prerelease/ios/documentation/Swift/Conceptual/Swift_Programming_Language/Extensions.html

PROBLEMS

All of the problems are much shorter this week. You do not write any Apps from scratch in this assignment, although you will be getting very familiar with the Game of Life simulation code.

1. (5 pts) How is the Application Delegate different from all other delegates?

2. (10 pts) For closures that are:

   a. Stored as a property in a class instance;
   b. Therefore, automatically capture **`self`** of that class;

   why is it important that they explicitly capture **`self`** as **`weak`** or **`unowned`**?

3. (12 pts) Consider a typical arrangement of a **`UIPicker`** within a **`UIViewController`**. The **`UIPicker`** is an Outlet (specified with the **`@IBOutlet`** attribute) which StoryBoard declares for you automatically as a **`weak`** reference. Now assume the **`UIPicker`**'s stored **`delegate`** property has been set to the **`self`** of the **`ViewController`** instance – assume this happened at a typical time such as in **`viewDidLoad()`** or in the **`didSet`** observer of the **`UIPicker`** reference. Draw a box-and-pointer diagram showing the reference cycle that's avoided by declaring the reference to the **`UIPicker`** as **`weak`**.

4. (4 pts) Why is it that adding **`UIPickerDataSource`** to the protocol list of a class cause an immediate compiler error, whereas adding **`UIPickerViewDelegate`** does not? For the sake of the question, assume the class starts off with no function members.

5. (6 pts) Suppose there was a generic (non-iOS) Swift class where the value of a property **`p`** was completely determined by the values of 3 other properties. One way of keeping **`p`** current is to make 3 **`didSet`** property observers: one for each dependent property, and recalculate **`p`** in each case. However this quickly becomes tedious. What's a better solution? Assume that the calculations used to compute **`p`** are quick and lightweight in memory usage.

6. (8 pts) Research the web to show how to convert a number to its hexadecimal notation as a **`String`**. As always show your citation. Do the same for the reverse: parse a **`String`** representation of a hexadecimal number. Wrap the code you find/write in an **`extension`** to the **`Int`** class and call the method **`fromHexString`**. What **`String`** represents the largest hexadecimal value you can parse and store correctly, assuming the result must be stored in an **`Int`** (signed, 64-bit integer)? Show your function

working with a few test cases. You may assume you are not parsing any '-' (negative) signs or non-integral values when parsing the String.

Problems 7-17 refer to the Game of Life simulation framework project. Most questions are very short and require no programming. Only the bottom-most (most complex) UIViewController and its helper classes are being referred to herein: the `LiveCellGridViewController`, the `CellGridModel`, and the `ModelBasedCellGridView`.

7. (5 pts) Does the call to **`setNeedsDisplay`** directly invoke **`drawRect`** in its call chain, or is it scheduled for later? When?

8. (5 pts) What happens when the ":" is left out of the selector string in the NSTimer initialization? What does the ":" indicate?

9. (6 pts) The given code has a notification sent in response to the **`ApplicationWillResignActive`** event. When the **`ViewController`** receives this event, it responds by canceling the repeating timer via **`invalidate`**(). Why is this unnecessary?

10. (10 pts) It would have been convenient to attach **`UIColors`** as **`rawValues`** to the **`enum`** in **`CellGridModel`**. (If we did so there's a particular property in a particular class that could be dispensed with – point it out). Why did we scrupulously avoid doing so?

11. (3 pts) In **`ModelBasedGridView`**, why must we prefix **`cellDim`** and **`drawRect`** with **`override`**?

12. (2 pts) What happens if you change **`cellPath.fill()`** to **`cellPath.stroke()`**?

13. (6 pts) Change the squares to circles. Use the **`UIBezierPath(arcCenter: …)`** constructor. Hints: See the **`UIBezierPath`** reference in the Apple Docs. As the center, use the center of the **`CGRect`** that is calculated currently; the radius should be easy to calculate; angles are measured in radians, so that 2*pi = 360 degrees; since you're going all the way around, it doesn't really matter at what angle you start as long as you end there, and drawing direction doesn't matter.

14. (15 pts) Add a slider directly beneath that controls the time interval between calculations. Lay out the slider directly below the grid. Let it range from 0.1 to 5 seconds. The visual update pacing should track the slider as it is dragged, not just when it is released ('Touch Up').

15. (10 pts) Add a text label just to the right of the slider that shows the time

interval currently selected. It should also update as the slider is dragged.

16. (5 pts) Add a text label below the slider that shows what generation the simulation is on. It should in sync with the actual generation shown. (Technically it's impossible to do this with super-high precision since it's up to iOS to actually schedule the redraws of different elements; however the *requests* to make the visual updates should be together in the code.)

17. (8 pts) Add an 'Active' toggle (`UISwitch`) that controls whether the animation is active or paused. It's alright if it doesn't take effect until the next scheduled timer has gone off.

What's coming ahead: We'll be enhancing the GoL simulation a bit more to showcase other important iOS features:

- Fully demonstrating and emphasizing MVC best practices
- Abstracting the model into a protocol to allow for alternate data model implementations
- Adding gesture recognizers that allow the user to "paint" new cells
- Embedding the grid in a scroll view to allow a larger playing area than what fits on-screen, and perhaps adding a pinch-zoom gesture
- Storing the current grid state in persistent storage, for later recall
- Putting the computation in a separate thread from the UI "main" thread, and using closures to communicate