iOS: Staff's Choice

Evil Hangman*

due by noon ET on Wed 4/18

Ingredients.

- Cocoa Touch
- MVC
- Evil

Help.

Help is available throughout the week at `http://help.cs76.net/`! We'll do our best to respond within 24 hours. Be sure, though, to take advantage of lectures and sections as well as videos thereof!

* Inspired by Keith Schwarz of Stanford.

Academic Honesty

All work that you do toward fulfillment of this course's expectations must be your own unless collaboration is explicitly allowed by some project. Viewing, requesting, or copying another individual's work or lifting material from a book, magazine, website, or other source—even in part—and presenting it as your own constitutes academic dishonesty, as does showing or giving your work, even in part, to another student.

Similarly is dual submission academic dishonesty: you may not submit the same or similar work to this course that you have submitted or will submit to another. Nor may you provide or make available your or other students' solutions to projects to individuals who take or may take this course (or CSCI S-76) in the future.

You are welcome to discuss the course's material with others in order to better understand it. You may even discuss problem sets with classmates, but you may not share code. You may also turn to the Web for instruction beyond the course's lectures and sections, for references, and for solutions to technical difficulties, but not for outright solutions to problems on projects. However, failure to cite (as with comments) the origin of any code or technique that you do discover outside of the course's lectures and sections (even while respecting these constraints) and then integrate into your own work may be considered academic dishonesty.

If in doubt as to the appropriateness of some discussion or action, contact the staff.

All forms of academic dishonesty are dealt with harshly.

Grades.

Your work on this project will be evaluated along four primary axes.

Correctness. To what extent is your code consistent with our specifications and free of bugs?
Design. To what extent is your code written well (*i.e.*, clearly, efficiently, elegantly, and/or logically)?
Scope. To what extent does your code implement the features required by our specification?
Style. To what extent is your code readable (*i.e.*, commented and indented with variables aptly named)?

Hangman.

☐ Well hello! I am thinking of a seven-letter word:

```
_______
```

Guess a letter, and I'll tell you where, if anywhere, it appears in the word. You can guess incorrectly up to, oh, six times. If you can figure out the word, you win; if you can't (and you're out of guesses), I win. Ready? Okay, guess a letter.

`A`, you say? You are so smart. That letter appears twice:

```
-A---A-
```

Recognize the word now? No? Not to worry, you still have six chances left, since your first guess was a good one. Guess again.

`E`, you say? Nope, sorry, not so smart after all. You've now used up one of your six chances, so you have five chances left. Guess again.

`I`, you say? Nope! You have four chances left. Guess again.

`Z`, you say? Nope! You now have three chances left. Guess again.

`N`, you say? Nice! That letter also appears twice:

```
-AN--AN
```

You still have three chances left. Guess again.

`E`, you say? I'll pretend I didn't hear that. You guessed that already! You still have three chances left. Guess again.

`H`, you say? Nice! That letter appears once:

```
HAN--AN
```

Recognize the word yet? `G`, you say? `M`, you say? That's right! It's `HANGMAN`!

☐ So that is how the age-old game of Hangman is played. But for this project, you will not just implement Hangman. (Hangman is boring.) You will implement….

Wait for it…

Evil Hangman.

Evil Hangman.

☐ Evil Hangman is quite like Hangman, except that the computer cheats. Rather than pick, say, a seven-letter word at the game's start, the computer instead starts off with a mental list (well, maybe a set or array) of all possible seven-letter English words. Each time the human guesses a letter, the computer checks whether there are more words in its list with the letter than without and then whittles the list down to the largest such subset. If there were more words with the letter than without, the computer congratulates the user and reveals the letter's location(s). If there were more words without the letter than with, the computer just laughs.

Put more simply, the computer constantly tries to dodge the user's guesses by changing the word it's (supposedly) thinking of. Pretty evil, huh? The funny thing is most humans wouldn't catch on to this scheme. They'd instead conclude they're pretty bad at Hangman. Mwah hah hah.

Suffice it to say that the challenge ahead if you is to implement…

Wait for it…

Evil Hangman.

☐ But what is the algorithm for evil? Well, let's consider what the computer needs to do anytime the user guesses a letter. Suppose that the user needs to guess a four-letter word and that there are only a few such words in the whole English language: `BEAR`, `BOAR`, `DEER`, `DUCK`, and `HARE`. And so the computer starts off with a universe (*i.e.*, list) of five words. Next suppose that the user guesses `E`. A few of those words contain `E`, so the computer had best decide how to dodge this guess best. Let's partition those words into "equivalence classes" (a la CSCI E-207) based on whether and where they contain `E`. It turns out there are four such classes in this case:

`----`, which contains `BOAR` and `DUCK`
`-E--`, which contains `BEAR`
`-EE-`, which contains `DEER`
`---E`, which contains `HARE`

Note that `HARE` is not in the same equivalence class as `BEAR`. Even though both have just one `E`, each's `E` is in a different location. If the computer is forced to admit that the word that it's "thinking of" contains an `E` (because it just so happens to have discarded all words that lack `E` in response to the user's past guesses), it will be forced to announce (and thus commit to) that letter's location.

So, back to our story: the user has guessed `E`. What should the computer now do? Well, it could certainly declare that the word it's "thinking of" does not contain `E`, the implication of which is that the list of five words becomes two (`BOAR` and `DUCK`). That does feel optimal: armed with two words, the computer might still have a chance to "change its mind" yet again later. Then again, `DUCK` seems pretty easy to guess, whereas most people might not even think of `HARE`. On the other hand, I don't remember the last time I used `BOAR` in a sentence. But let's keep it simple: you

may assume that the computer will always react to a user's guess by whittling its list down to the largest equivalence class, with pseudorandomness breaking any ties.

Realize, though, that the largest equivalence class might not always correspond to `----`. For instance, suppose that the user had guessed `R` instead of `E`. The equivalence classes would thus be as follows:

`----`, which contains `DUCK`
`---R`, which contains `BEAR`, `BOAR`, and `DEER`
`--R-`, which contains `HARE`

In this case, the computer should go ahead and admit that the word that it's "thinking of" contains `R` at its end, since that leaves three possible words (`BEAR`, `BOAR`, and `DEER`) and thus the maximum amount of maneuverability down the road in reaction to subsequent guesses. Of course, if the user has never heard of a `DUCK`, then `----` could very well be a superior strategy. But, again, lest you drive yourself nuts with overanalysis, you may assume that the largest equivalence class is optimal, even though it might not be in reality.

Indeed, that strategy might sometimes backfire, at least in a sense. Suppose that, in a new version of the story at hand, there are only three four-letter words in the English language: `BOAR`, `DEER`, and `HARE`. Suppose now that the user guesses `E`. In this case, our equivalence classes are:

`----`, which contains `BOAR`
`-EE-`, which contains `DEER`
`---E`, which contains `HARE`

Because each has one word, these classes are, of course, of the same size. But if we happen to select `-EE-` pseudorandomly, thereby whittling our list down to just `DEER`, we'll have revealed to the user two of the word's letters, whereas we could have revealed zero (had we selected `----` instead) or one (had we selected `---E` instead). But, again, that's okay. You are welcome, but not required, to implement a more sophisticated algorithm than that prescribed here; just make clear in some comments how yours happens to work.

As for how to divide a list of words into equivalence classes programmatically, well, we'll leave that as a challenge for you!

Getting Started.

☐ Alright, here we go!

Launch Xcode and select **File > New > Project…**. When prompted to choose a template, select **Application** under **iOS** (if not selected already), then select **Utility Application**. Then click **Next**.

On the screen that appears: input **Hangman** for **Product Name**; input **edu.harvard.extension** for **Company Identifier**; leave **Class Prefix** blank (it's okay if you see a placeholder of **XYZ** in gray); select **iPhone** next to **Device Family**; leave **Use Storyboards** unchecked (unless you'd like to use them); leave **Use Core Data** unchecked (unless you'd like to use it); check **Use Automatic Reference Counting**; check **Include Unit Tests**; then click **Next**. Choose a location for the project when prompted, check the box to create a local git repository if you'd like, then click **Create**.

☐ Let's take a tour of the code Xcode just generated for you.

With **Hangman.xcodeproj** open in Xcode, go ahead and expand each of the triangles at left (except for **UIKit.framework**, **Foundation.framework**, **CoreGraphics.framework**, and **SenTesting.framework**), and you should see `AppDelegate.{h,m}`, `MainViewController.{h,m,xib}`, and `FlipsideViewController.{h,m,xib}` among their contents. We'll take a tour through each of those files, but first a quick demo. Ensure that **Hangman > iPhone 5.1 Simulator** is selected in the drop-down to the right of the **Run** button in Xcode's top-left corner. Then click **Run** (or hit ⌘-R on your keyboard). If all goes well, the iOS Simulator should launch with this app. How fun!

Okay, it doesn't do all that much yet, but do click that ⓘ button in the app's bottom-right corner. Notice how the app flips around to its flipside. Well that's kind of neat. Click **Done** in the flipside's top-left corner, and you should be returned to the front side. It's okay if you'd like to do that again.

Anyhow, once ready for that tour of the code, quit iOS Simulator, return to Xcode, and open up `AppDelegate.h`. Notice how this file declares a class called `AppDelegate`, which implements a protocol called `UIApplicationDelegate`. That's good, because it's to an object of that class that `main` (in `main.m`) will ultimately delegate control. That class is implemented in `AppDelegate.m`, most of whose methods are just stubs with comments, except for `application:didFinishLaunchWithOptions:`.

Now open up `MainViewController.xib`. Ah, there's part of the app's UI. Indeed, there's that ⓘ button. If you see two cubes and one square to the left of the graph paper in Xcode's middle, click the little arrow in the circle in the graph paper's bottom-left corner. Then click **File's Owner** under **Placeholders**. Can you guess who the owner of this nib will be? Open Xcode's **Identity Inspector** to confirm or deny your prediction, as by clicking the fourth icon from the right below **View** in Xcode's top-right corner. Yup, this nib belongs to an instantiation of the `MainViewController` class.

Now click **View** under **Objects** (which itself is below **Placeholders**). Notice that this object's class is **UIView**, and it looks like it will fill an iPhone's whole screen. To confirm as much, open Xcode's **Attributes Inspector**, as by clicking the third icon from the right below **View** in Xcode's top-right corner, and change **Background** to something other than gray. (You may need to click the graph paper to see the change.)

Now open up `MainViewController.h`, which declares the class that owns the nib you just closed. Notice how `MainViewController` descends from `UIViewController`. That's handy, because `UIViewController` comes with a whole lot of features, per the UIViewController Class Reference:

http://developer.apple.com/library/ios/documentation/uikit/reference/UIViewController_Class/Reference/Reference.html

Notice, too, that `MainViewController` implements a protocol called `FlipsideViewControllerDelegate`. (It turns out that protocol is defined in `FlipsideViewController.h`, but more on that in a moment.) Notice that `MainViewController` has one instance method (`showInfo:`), but no instance variables.

Now open up `MainViewController.m`. Hm, lots more in this file. Let's look at each method in turn. Defined first are `viewDidLoad`, `viewDidUnload`, and `shouldAutorotateToInterfaceOrientation:`, all of which are documented in that UIViewController Class Reference. Also defined in `MainViewController.m` is `flipsideViewControllerDidFinish:`, which belongs to that protocol called `FlipsideViewController`, but more on that in just another moment. Last up is `showInfo:`, whose declaration was in `MainViewController.h`. Notice how this method allocates a `FlipsideViewController`, thereafter initializing it with `FlipsideViewController.xib`. It then defines itself as the delegate for the flipside's controller, sets the transition from front side to flipside, and presents that flipside.

Alright, almost done with the tour. Open up `FlipsideViewController.xib`. Ah, there's the flipside's UI. If you ctrl- or right-click **File's Owner**, you'll see how this nib's owner (a `FlipsideViewController`) is wired up to a view. You'll also see how that **Done** button is wired to a method (*i.e.*, `UIAction`) called `done:`. If you expand the triangle next to **View** (and, in turn, everything below), you'll see how the flipside's navigation bar and title relate to that button. In fact, if you ctrl- or right-click **Bar Button Item – Done**, you'll see that same wiring in reverse.

Okay, now open up `FlipsideViewController.h`. Not only does this file declare a class called `FlipsideViewController` (which descends from `UIViewController`), it declares a property called `delegate`, which must apparently point to an object (implied by `id`) that implements the protocol called `FlipsideViewControllerDelegate`. It also declares that instance method called `done:`. And, as promised, it declares that protocol, which apparently prescribes a method called `flipsideViewControllerDidFinish:`. The implication, of course, is that any class that implements `FlipsideViewControllerDelegate` (*e.g.*, `MainViewController`) should implement a method called `flipsideViewControllerDidFinish:`. Just as we've seen!

Finally, open up `FlipsideViewController.m`. In here are some stubs, just like those we saw in `MainViewController.m`. But this file also implements `done:`, whose sole line of code

apparently informs the flipside's delegate (*i.e.*, a `MainViewController` object) when the flipside is done flipping back.

Phew, time for a break.

☐ Welcome back. Feel free now to poke around the files in the **Supporting Files** group, but odds are you won't need to touch any of them for this project. But let's add one additional file to that group, an array of 234,371 English words![†] Download the property list (*i.e.*, XML file) at

`http://cdn.cs76.net/2012/spring/projects/ios-staff/words.plist?download`

and then drag `words.plist` into into that **Supporting Files** group within Xcode. When prompted to choose options for adding the file, check **Copy items into destination group's folder (if needed)**, ensure that **Hangman** is checked to the right of **Add to targets**, and then click **Finish**. (Since `words.plist` isn't a folder, it doesn't matter what's selected next to **Folders** in that window.) You should now see `words.plist` among your app's **Supporting Files**. Click it, and you'll see a huge array of English words.[‡]

Alright, that's it! It's time to delegate control of this project to you! (Get it?)

Specification.

☐ Your challenge for this project is to implement Evil Hangman as a native iOS app. The overall design and aesthetics of this app are ultimately up to you, but we require that your app meet some requirements. **All other details are left to your own creativity and interpretation.**

Features.

- ☐ Immediately upon launch, gameplay must start (unless the app was simply backgrounded, in which case gameplay, if in progress prior to backgrounding, should resume).
- ☐ Your app's front side must display placeholders (*e.g.*, hyphens) for yet-unguessed letters that make clear the word's length.
- ☐ Your app's front side must inform the user (either numerically or graphically) how many incorrect guesses he or she can still make before losing.
- ☐ Your app's front side must somehow indicate to the user which letters he or she has (or, if you prefer, hasn't) guessed yet.
- ☐ The user must be able to input guesses via an on-screen keyboard.
- ☐ Your app must only accept as valid input single alphabetical characters (case-insensitively). Invalid input (*e.g.*, multiple characters, no characters, characters already inputted, punctuation, *etc*.) should be ignored (silently or with some sort of alert) but not penalized.

[†] The words are taken from `/usr/share/dict/words` on Mac OS 10.7.3, with all forced to uppercase and duplicates removed.
[‡] Even though Xcode presents the contents of `words.plist` with three columns (**Key**, **Type**, and **Value**) as though the file contained a dictionary, it indeed contains an array (implemented in XML with an `array` element, which you can see if you open `words.plist` with a text editor).

- ☐ Your app's front side must have a title (*e.g.*, **Hangman**) or logo as well as two buttons: one that flips the UI around to the app's flipside, the other of which starts a new game.
- ☐ If the user guesses every letter in some word before running out of chances, he or she should be somehow congratulated, and gameplay should end (*i.e.*, the game should ignore any subsequent keyboard input). If the user fails to guess every letter in some word before running out of chances, he or she should be somehow consoled, and gameplay should end. The front side's two buttons should continue to operate.
- ☐ On your app's flipside, a user must be able to configure two settings: the length of words to be guessed (the allowed range for which must be [1, *n*], where *n* is the length of the longest word in `words.plist`); and the maximum number of incorrect guesses allowed (the allowed range for which must be [1, 26]).
- ☐ When settings are changed, they should only take effect for new games, not one already in progress, if any.

Implementation Details.

- ☐ Your app's UI should be sized for an iPhone or iPod touch (*i.e.*, 320 × 480 points) with support for, at least, `UIInterfaceOrientationPortrait`. However, if you or your partner owns an iPad and would prefer to optimize your app for it (*i.e.*, 768 × 1024 points), you may, so long as you inform your TF prior to this project's deadline.
- ☐ You must use the contents of `words.plist` as your universe of possible words. You're welcome, but not required, to transform it into some other format (*e.g.*, SQLite).
- ☐ Your app must come with default values for the flipside's two settings; those defaults should be set in `NSUserDefaults` with `registerDefaults:`. Anytime the user changes those settings, the new values should be stored immediately in `NSUserDefaults` (so that changes are not lost if the application is terminated).
- ☐ You must implement each of the flipside's numeric settings with a `UISlider`. Each slider should be accompanied by at least one `UILabel` that reports its current value (as an integer).
- ☐ You are welcome to implement your UI with Xcode's interface builder in `MainViewController.xib` and `FlipsideViewController.xib`, or you may implement your UI in code in `MainViewController.{h,m}` and `FlipsideViewController.{h,m}`.
- ☐ You must obtain a user's guesses via a `UITextField` (and the on-screen keyboard that accompanies it). For the sake of aesthetics, you are welcome, but not required, to keep that `UITextField` hidden (so long as the on-screen keyboard works). You are also welcome, but not required, to respond to user's keypresses instantly, without waiting for them to hit **return** or the like, in which case `textField:shouldChangeCharactersInRange:replacementString` in the `UITextFieldDelegate` protocol might be of some interest.
- ☐ Your app must use Automatic Reference Counting (ARC).
- ☐ Your app must tolerate low-memory warnings (as by reloading views when needed).
- ☐ Your app must work within the iPhone 5.1 Simulator; you need not test it on actual hardware. However, if you or your partner owns an iPad, iPhone, or iPod touch, and you'd like to install your app on it, see `https://manual.cs50.net/iOS` for instructions.

How to Submit.

☐ First, open up your project (*i.e.*, `Hangman`) in Xcode, select **Clean** from Xcode's **Product** menu, and then close them again.

Then create a ZIP file containing the project's folder and name the ZIP `########.zip`, where `#######` is your 8-digit Harvard ID (HUID), the same credential that you use to log into `help.cs76.net`.

Then head to `https://www.cs76.net/submit`, click the **login** link at top-right, click the link to your TF's dropboxes at top-left, click this project's own folder, click **Upload File**, and upload your ZIP file as prompted; no need to give it a title. Be sure not to click the wrong project's folder. You may re-submit in this same manner as many times as you'd like. Just take care to delete any prior submissions.

Be sure not to submit or re-submit after this project's deadline.