Assignment II: CodeBreaker

Objective

The goal of this assignment is to continue to recreate the demonstrations given through lecture 4 (now that we've added game logic) and then make some bigger enhancements. It is important that you understand what you are doing with each step of recreating the demo from lecture so that you are prepared to do those enhancements.

This continues to be about experiencing the creation of a project in Xcode and typing code in from scratch. **Do not copy/paste any of the code from anywhere.** Type it in and watch what Xcode does as you do so.

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.

Due

This assignment is due before the start of lecture 6.

Materials

Xcode (as explained in assignment 1).

Your solution to Assignment 1.

Required Tasks

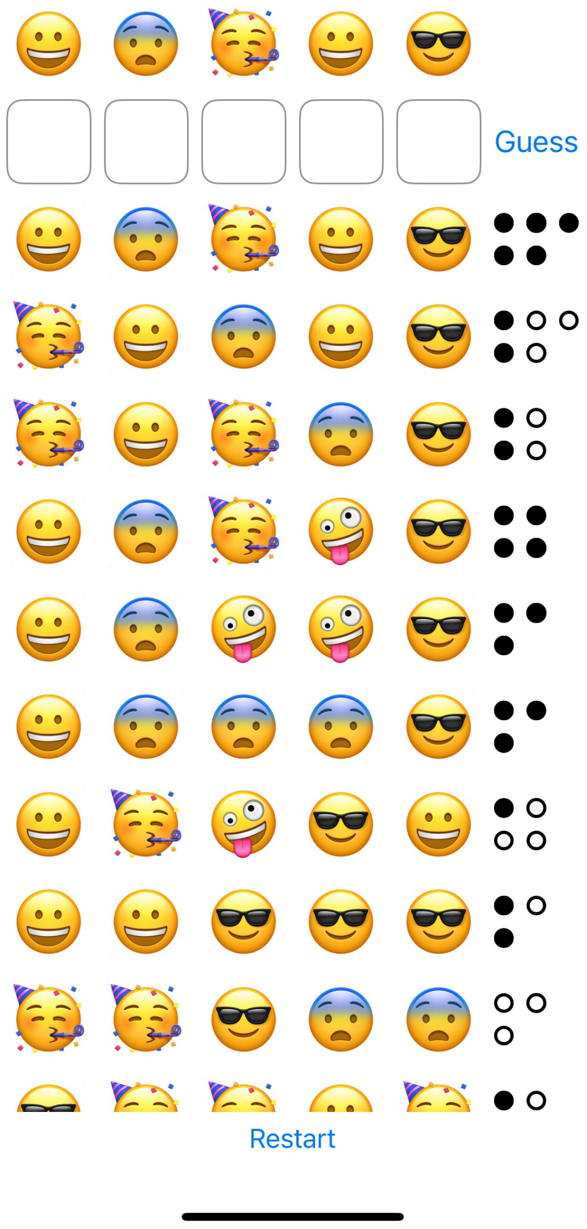
1. Get the CodeBreaker game working as demonstrated in lecture (thru lecture 4). Type in all the code. Do not copy/paste from anywhere.
2. Ignore attempts by the user that they've already tried before or which have no pegs chosen at all.
3. Add a "Restart Game" button to your UI (anywhere you think is best) which resets the UI and starts a brand new game.
4. Make the number of pegs in a CodeBreaker game be configurable (between 3 and 6 pegs) instead of being fixed at 4. You've already fixed `MatchMarkers` to do this in Assignment 1, now make it work with the rest of the code.
5. Now that you support 3 to 6 pegs, Restart Game should choose a random number of pegs between 3 and 6.
6. Enhance CodeBreaker further to play two different kinds of games: games where the pegs are colored circles (current version) or games where each peg is an emoji (i.e. a `String`).
7. The emoji version of your CodeBreaker game can choose from any set of emojis you choose. Faces are fun, but sports or animals or vehicles are all good ideas too. The size of the set of emojis can be linked to the number of pegs (3, 4, 5 or 6) or not. It's up to you.
8. In addition to choosing a random number of pegs as per Required Task 6, Restart Game must also pick one of the two different kinds of games (emoji or colored circles) at random.
9. Use at least one `static var` or `func` somewhere in your solution.

Hints

1. Your initial version of Required Task 3 can probably be implemented in four lines of code or so.
2. Surprisingly, Required Task 4 requires almost no changes to your UI (other than changing the way in which CodeBreaker games are created since you need to parameterize how many pegs a given game should have). Some of this is because of the work you did in Assignment 1 and some of it just due to the power of `ForEach`.
3. Your `init` functions in your Model will probably now need to take the number of pegs in the game.
4. You'll need to generate a random number between 3 and 6 to implement Required Task 5. You can do this with `Int.random(in: 3...6)`.
5. A `Peg` is currently `typealiased` to be a `Color`. That is not going to work now that `Pegs` can be either an emoji (a `String`) or a `Color`. A simple solution? Make `Peg` a `String` and specify colors by name (e.g. `"red"` or `"blue"`). If a `Peg` (`String`) isn't one of your supported `Colors`, then assume it's an emoji.
6. If `Peg` is no longer a `Color`, you can fall back to `import Foundation` in your Model instead of the `import SwiftUICore` (or `SwiftUI`) we were forced to use when `Peg` was a `Color`. Since our Model is supposed to be UI-independent, not `importing` any `UI` module feels much more comfortable and makes us less likely to put what should be UI code into our Model.
7. You'll likely want the `Text` that displays your emoji to be flexible in size (just like the colored circle pegs are). A simple way to do this is to make the `Text` be an `.overlay` of a `Circle` (or a `Rectangle`) and apply the modifiers `.font(.system(size: 120))` and `.minimumScaleFactor(9/120)` to it. This will make the `Text` choose a font for itself that is no larger than 120 point but no smaller than 9 point. And it will be flexible in size because, since it is an `.overlay`, it will be inheriting its layout entirely from its flexible-sized parent (e.g. a `Circle`).
8. To pick randomly between your two types of games (emoji or colored circles), you probably want a random `Bool` value. `Bool.random()` will give you one.
9. You can feel free to only support the colors we've been using so far (`"blue"`, `"green"`, `"red"`, `"yellow"`) but a slightly more expansive solution is to limit yourself to the built-in colors in the `Color struct`. Check out Challenge Mode below for how you might do this.

Screenshot

Screenshots are NOT Required Tasks (they're more like Hints). Your UI does NOT have to look like this. This is only provided in case the text description above is leaving you baffled as to what's being asked of you. This screen shot is the "emoji" version of the game with 5 pegs. Obviously that's only one part of the solution. You need to support 3, 4 and 6 peg versions and of course you must also support the original colored pegs version. The Restart button should choose some combination of all of that randomly.



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.).
- 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 and knows how the CodeBreaker game code from the lectures works, but should <u>not</u> assume that they already know your (or any) solution to the assignment.

Challenge Mode

Challenge Mode is an opportunity to expand on what you've learned this week and push yourself with more difficult exercises. Attempting one or more of these each week is highly recommended to get the most out of this course.

1. 2 pts. Enlist the help of AI to expand the colors your game can support (beyond `"blue"`, `"red"`, `"green"`, and `"yellow"`) to (at least) include all the built-in colors in the `Color struct` (e.g. `Color.brown`, `Color.pink`, etc.).

   Unfortunately, the `Color struct` does not support accessing its built-ins by name, but you can ask an AI of your choice to write you some code to get a `Color` from its name (and vice-versa). Something like: "create a failable initializer for `Color` which takes a color name and translates it into one of `Color`'s built-in `static` colors". Hopefully your AI will give you code for an `extension` to the `Color struct` that has an `init?` that will do the trick!

   You could also follow-up by asking it "now create an `Optional var` called `name` which does the inverse" (though you wouldn't absolutely need such a thing to do this Challenge Mode, it could come in handy in certain solutions).

   Be sure to check your AI's work! Sometimes it hallucinates and makes up colors that are not, in fact, `static vars` in `Color`. Also, some AI's are smarter than others, so if you don't get what you want from one of them, try one of the others.

   If your AI does succeed in doing this, it will hopefully make the `init` (and the `name var`) return `Optionals` (since it needs to return `nil` when there's no such `Color` for a given name or vice-versa). This is a good opportunity to make sure you understand how an `Optional` works.

   If all goes well, and depending what your AI comes up with, you should be able to write some code that looks something like this …

   ```
   if let color = Color(name: peg) { /*color pegs*/ } else { /*emoji pegs*/ }.
   ```

   or maybe this? `let color = Color(name: peg) ?? .clear`

   Test out your code by using different colors on game restart. If you can do that, you are well on your way to accomplishing this next Challenge Mode …

2. 1 pt. When you restart a game, choose from various different sets of emojis (or colors) to use (we'll call this a "theme" for the game). For example, one time you restart you might use a faces theme (😀🤪🥳😨), then the next time earth tone colors, then the next time vehicle emojis (🚗🚲🛬⛵), then the next time the standard colors, etc.

3. 1 pt. *Name* the various themes you added for Challenge Mode 2 above (e.g. "Faces" or "Earth Tones") and *show this name somewhere on screen* as a title for the game.

   For this one, you might find you want to build a data structure using both `Array` and `Dictionary`, but you'll need to understand `Optional`s to use `Dictionary` since looking something up in a `Dictionary` returns an `Optional` (which makes sense because sometimes what you're trying to extract from a `Dictionary` might not be in there).

   This Challenge Mode should probably only require a few lines of code (not including the lines of code that load up your various themes of emojis/colors).

   You'll probably have to hardwire one of your themes to be the first one that appears when your app starts up (unless you want to be truly bold and try to figure out how to use `.onAppear` to pick one at random the first time your game appears on screen).

   An additional simplifying assumption you can make here is that there's at least one well-known theme that is guaranteed to always be in there (i.e. the one that appears when you first start the game). With those assumptions, you can use "force unwrapping" (i.e. `!`) more often and not have to be checking whether the return value from doing a `Dictionary` lookup is `nil` or not. For example, you could then assume that `themes.keys.randomElement()` never returns `nil` (if `themes` were a `Dictionary` of your themes) and that `themes["Mastermind"]!` would never crash. Otherwise, you might be in a bit of a pickle when your game first appears and there is no theme to use!