Don’t copy-paste from this document. It’s better to write things down yourself.

The format of the sections starts vague and then gets very specific. If you’re comfortable, try it yourself! Lean on the code written as much as you need to, or ask for more help if the document isn’t as clear as it tries to be.

At the start of each section, you should be able to run and build the project. If you are unable to do so, raise your hand and ask for help.

Because of this, the order in which we write things may be slightly nonsensical. Keep this in mind while coding. We’re not going to be able to test what we’re doing as we’re doing it, but in the real world we’re not as strapped for time.

One last thing: When calling upon functions that have already been written for you, go explore them. Find out what they do, why they do it, and how they do it.

Grab the project off github

Use the following URL to grab the incomplete project from github. Either clone it or download it and extract it to your computer.

https://github.com/MichaelWTA/doggo-game-incomplete

Then open XCode and select the project file to open.

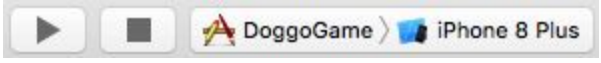
Learn how to run the program and use the debugger (TODO: Debug)

Navigate yourself to “// TODO: Debug” using the search functionality in the left pane or by hitting Command+Shift+F. Let’s learn how to set breakpoints. Go ahead and add a line that we can attach our breakpoint to, for example:

```
print("here")
```

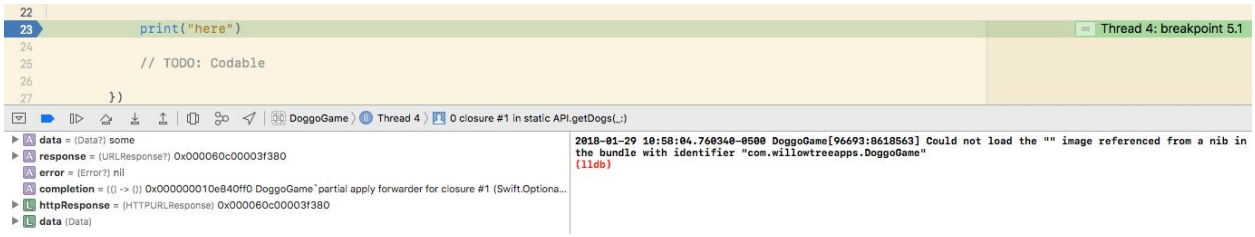
Now click on the line number to the left to add a breakpoint:

And run the program using cmd-r or by clicking the play button in the top left after selecting a simulator:



You should have the simulator open up and then after a second XCode will come back to the foreground because we’ve hit the breakpoint. We want to focus on the bottom portion of our IDE

now:



The play/pause button will continue our program til we hit the next breakpoint (if there is one). The next button to the right will 'step forward', executing a line of code. The next will 'step into', showing what is going on inside the line of code we're executing. And the other is 'step out of', the opposite of 'step into'.

Under that is a pane that shows the variables currently in our scope, You can expand these to see details on them. To the right of that is a pane that shows our debugging output. You can type on the line that says "(lldb)". Try to type "`po response!`" and see what happens. "po" means "print out" and can be used to analyze what the current state of your program is. Use it often.

Fix codable (TODO: Codable)

Right now our API just grabs raw JSON data. We can't really do much with that until we make it consumable. We want to edit our data models to be codable, which allows us to easily convert our raw JSON into the models we designed.

Codable is already implemented for `Headshot.swift`, so look at the `getDogPicture()` method for help if need be.

First, make our `Profile` model conform to Protocol `Codable`. Open `Profile.swift` and add this conformance to the class declaration.

```
struct Profile: Codable {
```

Now we need to take advantage of what the "codable" protocol does. It allows us to easily and painlessly convert JSON into structs that we have defined, and vice versa.
Let's apply this in `Api.swift`. We'll create a `do catch` block to try and parse the JSON data, or catch any errors that occur while we're attempting to do so.

```
do {

} catch let error as NSError {

}
```

In this `do` we are going to create a `JSONDecoder` and try to use it to `decode` the `Profile` struct from our `data`. If it works, we'll call a `completion` with that data.

```
do {
    let decoder = JSONDecoder()
    let jsondata = try decoder.decode(Profile.self, from: data)
    completion(jsonData, nil)
}
```

If we fail to `decode` this data then that means that our JSON didn't conform to the `struct` we created. We got unexpected data. That's why in our `catch` we'll `NSLog` the `error` and then return it via our `completion`.

```
} catch let error as NSError {
    NSLog("JSON Parsing error: \(error)")
    completion(nil, error)
}
```

And that should be it for handling the data from the API!

Give our DoggoButton state (TODO: AnswerState, showAnswer)

We will add state to our DoggoButtons in order to more easily shade them green and red, depending on their state. And show or hide the names associated with the buttons. When state gets set it will call upon a function to handle all of this.

In our `enum AnswerState` add the case for `.correct` and a case for `.incorrect`.

```
enum AnswerState {
    case correct
    case incorrect
    case unknown
}
```

Next we will use the `switch` statement on `didSet` of our `answerState` variable. This will be called whenever `answerState` gets set to a value. Add a case for `.correct` and `.incorrect` which will `showAnswer` and tint the button the desired color.

```
switch answerState {
    case .correct:
        showAnswer(withColor: .green)
    case .incorrect:
        showAnswer(withColor: .red)
```

```
    case .unknown:
        hideAnswer()
}
```

Unfortunately, `showAnswer` doesn't exist yet. Let's fix that. We'll create a `private func` that takes a `UIColor`. Swift lets us give two names to the variable so that we can call upon with one (as we did before, `withColor`) but then refer to it with another (`color`). Like so:

```
private func showAnswer(withColor color: UIColor) {

}
```

This func won't be called from any other classes, so we make it `private`. It's good practice.

The class we're working in right now is a `DoggoButton`. It's a button represented with a picture of a dog, and then some other UI elements layered on top of that. One of them is our `tintView`, whose `backgoundColor` we would like to set. We'd also like to make sure the `titleLabel` isn't below anything else, it will show the name of the dog on the button.

```
private func showAnswer(withColor color: UIColor) {
    tintView.backgroundColor = color
    if let titleLabel = titleLabel {
        bringSubview(toFront: titleLabel)
    }
}
```

The strangest part of this code is "`if let titleLabel = titleLabel`". What's going on here? In Swift there's a concept of an `optional type`. For example, an `Int` has to be what we consider a number. But if it could be a number or it could be `nil` (nothing) then it has to be declared as an `Int?`, an `optional Int`. This allows our code to be safe because we'll always be prepared for the possibility that a value is `nil`. When we use "`if let`" we're saying "if there is a non-nil value here, use it".

Sorry, that was a mouthful. Call for help if you want some assistance in understanding. Otherwise, let's continue creating this function. Right now the `alpha` of both of these elements (`tintView` and `titleLabel`) is `0`, they are entirely see-through. Let's give them an `alpha`, and actually `animate` it over a period of time so that they fade into view.

```
private func showAnswer(withColor color: UIColor) {
    tintView.backgroundColor = color
    if let titleLabel = titleLabel {
        bringSubview(toFront: titleLabel)
    }

    UIView.animate(withDuration: 0.5, animations: {
```

```
        self.tintView?.alpha = 0.3
        self.titleLabel?.alpha = 1.0
    })
}
```

Now we've completed our `showAnswer` function! When the state of our button changes from `.unknown` to `.correct` or `.incorrect` the name will fade into view, and it will fade into red or green depending on whether the answer is correct or not.

Write some logic (TODO: checkAnswer)

Check to see if the selected answer is correct. If it is, create and start a new round. If the answer is incorrect, decrement the number of guesses we have left. If we have no more guesses left, create and start a new round.

In our `DoggoGame.swift` file we will write a `func` called `checkAnswer`. It will take an `Int` as an argument (the `index` of the answer we selected), and return a `Bool` (whether or not the answer was correct).

```
func checkAnswer(index: Int) -> Bool {

}
```

In order to determine whether the answer is the correct one, grab the selected profile from `currentChoices[ ]` and compare it to `correctAnswer`, then return whether the answer was correct or not.

```
func checkAnswer(index: Int) -> Bool {
    let profile = currentChoices[index]
    if correctAnswer == profile {
        return true
    } else {
        return false
    }
}
```

If the user has selected the correct answer, we should create and start a new round using `createNewRound()` and our `delegate`, `startNextRound()`. It would be nice to wait for a second before doing so as well so they can see their correct answer, which we can do with `DispatchQueue.main.asyncAfter()`.

```
if correctAnswer == profile {
    DispatchQueue.main.asyncAfter(deadline: .now() + 1) {
        self.createNewRound()
        self.delegate?.startNextRound()
    }
    return true
}
```

If the user has selected an incorrect answer, we should decrement their `guessesLeft`. If they have no more guesses, we should create and start a new round. In the same way we did before.

```
else {
    guessesLeft -= 1
    if guessesLeft <= 0 {
        createNewRound()
        self.delegate?.startNextRound()
    }
    return false
}
```

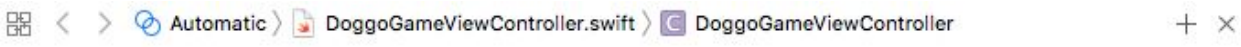
That should be it for this function!

Link to storyboard (TODO: doggoTapped)

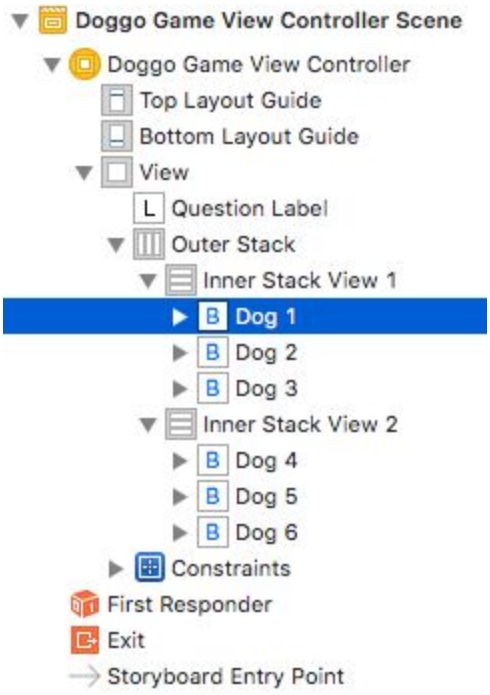
Link our buttons to a function that will check their answer when a user taps a button. If the user answered correctly, change the state of the button to correct and disable user interaction with all buttons while we set up the next round. If it is incorrect, set the state to incorrect and disable the button that we pressed so it cannot be pressed again.

Open our `Main.storyboard` and tap on the picture of two circles in the top right of XCode.

This will open the editor and the storyboard side by side. Navigate the editor to `DoggoGameViewController.swift` if it is not there by default.

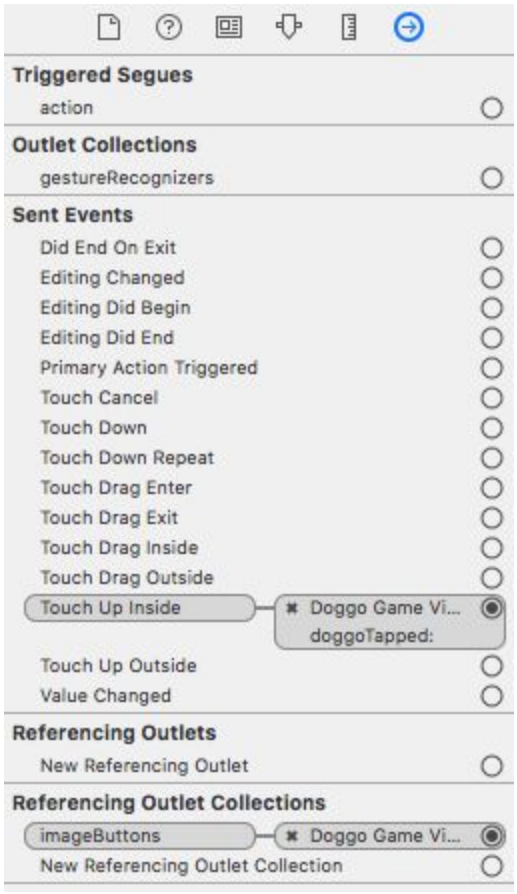


Explore the storyboard navigation bar, expanding the elements until you can see the buttons labeled Dog 1' to Dog 6'.



Ctrl-click and drag one of the buttons to the text editor to create an action outlet. Be sure that the `Connection` is an `Action`, your function has a name, and its `Type` is `DoggoButton`.

When selecting this button, you should now be able to see the new `Touch Up Inside` event connected to it in the right side panel.



Ctrl-click and drag each other button onto the function declaration you just created, or from the function to the buttons. The function itself should highlight as you do so, and each button should be given the same event in its navigation panel. Now each button will call upon this function when interacted with! Unfortunately, this function doesn't do anything! Yet. Let's fix that.

As we said before, on tap we want to check our answer and then set the state of the button appropriately. Also, if the user was incorrect we should disable user interaction on that button. If they were correct, totally disable user interaction while we set up the next round.

Let's `checkAnswer` using the `sender.id` in our `doggoTapped` function.

```
@IBAction private func doggoTapped(_ sender: DoggoButton) {
    if doggoGame.checkAnswer(index: sender.id) {

    } else {

    }
}
```

Now that we know whether the answer is correct or incorrect, set the `sender.answerState` and `setInteractionEnabled` to false for all buttons, or just for the `sender` appropriately.

```
if doggoGame.checkAnswer(index: sender.id) {
    sender.answerState = .correct
    setInteractionEnabled(false)
} else {
    sender.answerState = .incorrect
    sender.isUserInteractionEnabled = false
}
```

Now you should be able to tap on a picture and have it turn green or red depending on whether your answer was correct or not. Also, once red you can no longer tap on that picture, so the user can't accidentally choose the same one twice.

BONUS

This app is obviously in need for improvement. Make it so. Ask us for help implementing your ideas.
Scoring
Launch screen
Configurations