

# ACE THE IOS INTERVIEW



ARYAMAN SHARDA

UPDATED FOR XCODE 13 & SWIFT 5

# Ace The iOS Interview

Aryaman Sharda

Published by Aryaman Sharda | Digital Bunker LLC.

Copyright © 2022 Aryaman Sharda

All rights reserved.

No portion of this book may be reproduced in any form without permission from the publisher,  
except as permitted by U.S. copyright law.

For permissions contact: [aryaman@digitalbunker.dev](mailto:aryaman@digitalbunker.dev)

<https://digitalbunker.dev/>

<https://twitter.com/aryamansharda>

# Legal Disclaimer

This eBook is presented to you for informational purposes only and is not a substitution for professional advice. The contents herein are based on the views and opinions of the author and all associated contributors.

While every effort has been made by the author and all associated contributors to present accurate and up to date information within this document, technology changes rapidly. Therefore, the author and all associated contributors reserve the right to update the contents and information provided herein as these changes progress. The author and / or all associated contributors take no responsibility for any errors or omissions if such discrepancies exist within this document.

The author and all other contributors accept no responsibility for any consequential actions taken, whether monetary, legal, or otherwise, by any and all readers of the materials provided.

Readers' results will vary based on their individual perception of the contents herein, and thus no guarantees can be made accurately. As such, no guarantees are made.

This eBook and all corresponding materials (such as source code) are provided on an as-is basis, without warranty of any kind, express or implied, including but not limited to the warranties of merchantability, fitness for a particular purpose, and noninfringement.

In no event shall the author or copyright holder be liable for any claim, damages or other liability, whether in action of contract, tort or otherwise, arising from, out of or in connection with this eBook.

All trademarks and registered trademarks appearing in this eBook are the property of their own respective owners.

# Table of Contents

<b>Legal Disclaimer</b>	<b>2</b>
<b>Table of Contents</b>	<b>3</b>
<b>About This Book</b>	<b>9</b>
<b>Onsite Interviews &amp; Take-Home Assessments</b>	<b>11</b>
How To Approach This Section	11
Breaking Down The Problem	11
Finding Practice Problems	12
Easy Wins	12
Documentation	13
Version Control	13
Writing A README	14
Conventions & Best Practices	14
<b>Assessment 1: Networking &amp; Basic Codables</b>	<b>16</b>
Managing Endpoints	18
Third Party Dependencies	21
Building The Network Layer	22
Fetching Remote Images	23
Final Implementation	25
<b>Assessment 2: UIKit &amp; User Interaction</b>	<b>26</b>
Add	28
Count	31
Undo	36
Redo	41
<b>Assessment 3: Recreating Mockups &amp; AutoLayout</b>	<b>46</b>
Mockup	47
Building The Vehicle View	49
Building The Price Display	55
<b>Assessment 4: Building A Logger</b>	<b>64</b>
Clarifying Requirements	64
Defining The Scope	65
Handling Different Severity Levels	66
Making Our Logger Context Aware	67

Creating The Logger	68
Final Implementation	69
<b>Assessment 5: Creating An Analytics Service</b>	<b>72</b>
Basic Implementation	72
Final Implementation	73
<b>Assessment 6: Efficiently Displaying The Fibonacci Sequence</b>	<b>76</b>
Picking An Approach	76
Handling Integer Overflow	76
Making Optimizations	77
Final Implementation	78
<b>Assessment 7: Visualizing Articles &amp; Advanced Codables</b>	<b>83</b>
Creating The Models	88
Parsing The JSON	89
Building The View Controller	90
Creating Custom Cells	91
Final Implementation	93
<b>How To Approach The Data Structures &amp; Algorithms Interview</b>	<b>97</b>
<b>iOS &amp; Swift Interview Q &amp; A</b>	<b>100</b>
<b>General Knowledge</b>	<b>101</b>
What does app thinning mean?	101
What is the difference between the stack and the heap?	102
Is an enum a value type or a reference type?	103
What is the difference between an escaping closure and a non-escaping closure?	104
What are trailing closures?	106
What is code coverage? How do you enable it in Xcode?	108
What common problem does the Main Thread Checker help detect?	110
What are Swift's different assertion options?	111
What are the differences between the static and class keywords?	114
What is the difference between the App ID and the Bundle ID?	115
What features of Swift do you like or dislike?	117
What are the different URLSessionConfiguration options?	118
What is the difference between static and dynamic dispatch?	119
What is the difference between the UIApplicationDelegate and SceneDelegate?	120
What is the difference between a dynamic library and a static library?	121
What are generics and what problem do they solve?	123

What is a .dSYM file?	125
What are the differences between a class and a struct?	127
What are the different execution states your application can be in?	129
How can we handle changes in our application's lifecycle state?	130
How would you debug view layout issues?	132
What is operator overloading?	136
What are the differences between a delegate and a notification?	138
What are the differences between a library and a framework?	139
How do you create interoperability between Objective-C and Swift?	140
What does the File's Owner do?	142
Are closures value or reference types?	143
What are the differences between Keychain and UserDefaults?	145
What are our options for storage and persistence?	146
When would you use a struct versus a class?	147
What do the Comparable, Equatable, and Hashable protocols do?	148
What methods are required to display data in a UITableView?	151
What does the CodingKey protocol allow you to do?	152
What is the difference between the designated and convenience initializer?	154
What is your preferred way of creating views?	155
What would happen if a struct had a reference type in it?	157
What are the stages in a UIViewController's view lifecycle?	158
How does code signing work?	160
<b>Swift Language Features</b>	<b>163</b>
What is the difference between == and ===?	163
What's the difference between Self vs self?	165
How would you limit a function or a class to a specific iOS version?	166
What does the final keyword do?	167
What does the nil coalescing operator do?	168
What does defer do?	169
What is optional chaining?	170
Can we use Swift's reserved keywords as variable or constant names?	171
What is the difference between try, try!, and try??	172
What is an inout parameter?	174
How can we limit a protocol conformance to a specific class or type?	175
Does Swift support implicit casting between data types?	177

What does the Caselterable protocol do?	178
Can you explain what the @objc keyword does?	179
What are compilation conditions?	180
What are the differences between Swift's access control modifiers?	181
What does the typealias keyword do?	182
What does copy on write mean?	184
How are Optionals implemented?	185
What are the higher order functions in Swift?	186
What is the difference between Any and AnyObject?	190
What is a raw value in an enum?	191
What are our options for unwrapping optionals in Swift?	193
What is an anonymous function?	194
What is the difference between is, as, as?, and as! ?	195
What are some of the main advantages of Swift over Objective-C?	198
How do we provide default implementations for protocol methods?	200
What does it mean to be a first class function or type?	202
Do all elements in a tuple need to be the same type?	204
What is protocol composition? How does it relate to Codable?	205
What does the mutating keyword do?	206
How do you use the Result type?	208
Is Swift a statically-typed language?	210
What is an associated value?	211
What does a deinitializer do?	212
How can you create a method with default values for its parameters?	213
What is type inference?	215
What does the rethrows keyword do?	216
What does the lazy keyword do?	218
What does typealiasaliasing do?	219
What does the associatedtype keyword do?	221
While iterating through an array, how can we get both the value and the index?	223
Your app is crashing in production. What do you do?	224
<b>AutoLayout &amp; UIKit</b>	<b>226</b>
What is a view's intrinsic content size?	226
How would you animate a view that has a constraint?	227
On what thread should all UI updates be made?	228

What is the difference between bounds and frame?	229
What is the Responder Chain?	231
What does an unwind segue do?	232
What is the difference between pushing and presenting a new view?	235
What is the difference between a point and a pixel?	236
How are Content Hugging and Content Compression Resistance different?	237
What does App Transport Security do?	238
What are layer objects?	239
What is the purpose of the reuseIdentifier?	240
What is the difference between layoutIfNeeded() and setNeedsLayout()?	241
What does layoutSubviews() do?	242
What does viewDidLayoutSubviews() do?	243
What does setNeedsDisplay() do?	244
What is a placeholder constraint?	246
What is Dynamic Type in iOS?	247
What are the differences between a .xib and a .storyboard file?	248
What is the difference between layout margins and directional layout margins?	249
<b>Concurrency</b>	<b>253</b>
What are the differences between DispatchQueue, Operation, and Threads?	254
What is the difference between a serial and a concurrent queue?	256
What is a race condition?	258
How can we prevent race conditions [the reader-writer problem]?	260
What does deadlock mean?	262
How can we group multiple asynchronous tasks together?	263
How can you cancel a running asynchronous task?	265
Can you explain the different quality of service options GCD provides?	267
What does it mean for something to be thread safe?	269
<b>Memory Management</b>	<b>270</b>
What is automatic reference counting?	270
What is the difference between strong, weak, and unowned?	272
What is a memory leak?	273
How would you avoid retain cycles in a closure?	274
<b>Testing</b>	<b>275</b>
What is a unit test?	276
What does Arrange, Act, and Assert mean with respect to unit tests?	277

What are UI tests?	278
What is Snapshot Testing?	279
What is Test Driven Development?	280
What is Behavior Driven Development?	281
What are the purposes of <code>setUp()</code> and <code>tearDown()</code> in the XCTest framework?	283
How would we test the performance of a method in our tests?	284
<b>More Questions?</b>	<b>284</b>
<b>Acing the Behavioral Interview</b>	<b>286</b>
S.T.A.R. Format	286
You're Not Out Of The Woods Yet	288
An Interviewer's Perspective	288
Before The Interview	289
Focus On The Silver Lining	291
Be Specific	291
<b>Example Behavioral Interview Questions</b>	<b>292</b>
General	292
Teamwork	293
Integrity & Ethics	294
Initiative & Innovation	295
Leadership	295
Career & Personal Growth	296
Decision Making	296
Communication	297
Adaptability	298
Time Management & Prioritization	300
<b>Engineering + Behavioral Questions</b>	<b>301</b>
General Technical Questions	302
Platform Interview Questions	302
Questions For The Technical Interviewer	303
<b>When It's Your Turn To Ask Questions</b>	<b>304</b>
Questions For The Hiring Manager	306
<b>Afterword</b>	<b>307</b>
<b>Sources</b>	<b>308</b>

# About This Book

Hey there! My name is Aryaman Sharda and I started making iOS apps way back in 2015. Since then, I've worked for a variety of companies like Porsche, Turo, and Scoop Technologies just to name a few. Over the years, I've mentored junior engineers, built developer tools, and shipped dozens of apps to the App Store.

As I was interviewing for these roles, I was always looking for easy-to-follow explanations of all the relevant iOS interview topics and iOS-specific practice problems. After years of not being able to find such a guide, I decided to write one myself.

Over the last five years, I've been consolidating every relevant question, concept, and take-home assessment I've encountered during my interviews. They are all included in the following pages, along with sample solutions, screenshots, and step-by-step instructions. This book contains *everything* you need to know for your next iOS interview. "Ace The iOS Interview" will help you take the guesswork out of your interview preparation and will hopefully serve as a practical guide for landing your dream job.

With hundreds of interviews under my belt, both as a candidate and as an interviewer, I know both what an interviewer is looking for as well as the common mistakes most candidates tend to make. We'll go over in detail about how to tailor your responses to highlight the specific criteria the interviewer is looking for, as well as strategies for the behavioral interview portion and other "do's and don'ts" along the way.

For those of you looking for your first iOS job, I know first-hand how difficult it can be. Learning iOS development has been one of the most impactful and rewarding experiences I've had. It's enabled a lucrative career and the opportunity to work on apps that impact millions of people. All of that upside is on the other side of making it through the interview and hopefully this book will help you do just that.

My sincerest hope is that you find this book helpful and that it helps you land your next iOS position. If it does, I'd love to hear about it. Thanks for reading and good luck!

I've also created a [free to join Slack workspace](#) for anyone to discuss topics related to iOS interviewing, mock interview requests, sharing interview questions, job opportunities, and much more.

*If you want to submit a question for inclusion in this book and to help other iOS developers, feel free to contact me or share it in our Slack workspace.*

# Onsite Interviews & Take-Home Assessments

In this section, we'll take a look at some of the onsite and take-home interview assessments I've encountered over the last few years. We'll cover how to break down the problem, clarify requirements, and how to work with the interviewer to solve the problem.

While all of these questions contain a sample solution, my approach may not be convincing to you and that's absolutely fine. The sample solution is primarily meant to offer you a starting point.

Feel free to modify it to match whatever conventions, design patterns, or problem-solving approach you prefer.

If you come up with an improved implementation or have any suggestions for the answers to these problems, I'd love to see it. Feel free to share them with [me on Twitter](#) or by contacting me at [aryaman@digitalbunker.dev](mailto:aryaman@digitalbunker.dev).

## How To Approach This Section

Over the following pages, I have reproduced the interview questions and take-home assessments I encountered exactly as they were presented in the interview process.

All questions are supported by screenshots and sample implementations.

I recommend spending some time thinking about how you would answer the questions, what you would ask the interviewer, and what requirements you would clarify before you peek at the sample solution.

## Breaking Down The Problem

Many onsite interviews are designed to mimic a situation where you and the interviewer are working together on a project and may need to decide on specifics as you go along. Be prepared to encounter a lot of deliberately vague questions.

Interviews of this type aim to assess your ability to think deeply about a new problem, what concerns or bottlenecks you can identify, your ability to ask clarifying questions, and the level of planning you do before you start writing code. The biggest mistake you can make here is to jump right into writing code without doing any type of due diligence or planning.

From the interviewer's perspective, they're looking to see:

- What did you do when you didn't have all of the information?
- Does your code have the flexibility to adapt to changing requirements?
- Did you spend enough time thinking about the problem before writing code?
- Did you consider how you'd handle errors?

At the end of the day, the interviewer is looking for a new teammate so demonstrating your ability to collaborate professionally with others is equally, if not more, important than solving the problem itself.

Don't be afraid to ask questions, explain your thought process out loud, and remember to be personable - it sounds obvious, but sometimes we forget when our nerves get the best of us.

## Finding Practice Problems

While interviewing, I found that one of the best places to find practice iOS interview problems was GitHub.

I would periodically search for variations of "swift take home assessment" or "iOS exercise" and would find public take-home assessments submitted by other iOS developers.

You'll have to do some digging, but you'll eventually find projects that are both novel and updated recently. With this approach, you'll have an inexhaustible supply of practice questions at your disposal.

Once you begin doing this consistently, you'll start noticing patterns in the questions, and that can give you a sense of what variations of a given problem are most popular.

If you would like to submit a problem for inclusion in this book, please feel free to contact me!

## Easy Wins

As an interviewer, it's always refreshing to review a take-home project that's well documented, follows typical Swift conventions, contains a README, and leverages version control.

Most candidates don't add this final touch to their submissions, but it can really make a difference for very little additional effort.

## Documentation

When writing comments, explaining **why** the code is written a certain way is arguably more important than explaining **what** it does. Programmers can generally understand what a given piece of code does, but they might find it harder to reason about the context in which it was written and the potential side effects of its execution.

Here are some examples of good comments as they focus on communicating why the code was written a certain way instead of what it does:

```
// TODO: When iOS 12 and lower support is dropped replace this  
// method with withTintColor(color, renderingMode:  
  
// Declaring as private to avoid accidentally being instantiated from  
// another instance.  
  
// We're disabling the "Next" button as this is an invalid state and the  
// user has no available options.
```

In contrast, these would be considered poor comments as they don't provide any additional information or insight the code doesn't already convey:

```
// Hide error label  
  
// - Parameter urlString: String representation of the URL to load the  
// image from  
  
// Presents the next view in the user flow
```

In addition, comments should:

- Be used to provide a brief summary for a longer piece of code.
- Be used to address questions or concerns the code is incapable of answering on its own.
- Clearly communicate any possible side effects that execution of the code may cause.
- Be tailored to other developers who may not have the same familiarity and history with the code as you do.

## Version Control

Every project, take-home assessment or otherwise, should use version control.

Familiarity with version control (e.g. branching, resolving conflicts, etc.) is an essential skill for your job. So, demonstrating your ability to use it correctly [with good conventions] will only help reinforce the narrative that you'll be able to hit the ground running.

## Writing A README

Your take-home assessment submission should contain a README.

At the very least, the README should provide a summary of the project, list any third-party dependencies you're using, and provide instructions about how to set up and run the application (if applicable). For example, this is where you'd tell the interviewer that they will have to run `pod install` before running the project.

You can also use the README to highlight known bugs, missing features, or any other pertinent information the interviewer should know.

It's also a good opportunity to address the following questions:

1. What are the major components of your implementation?
2. What notable technical decisions did you have to make?
3. Did you have to make any compromises to limit the scope?
4. What improvements would you make if there was more time?
5. Are there any remaining TODOs?
6. Did you implement error handling? Why or why not?
7. Why did you choose to organize your code this way? What other design and architecture patterns did you consider?
8. What third-party dependencies are you using (if any)?

## Conventions & Best Practices

Often, I'll see candidates using a mix of different coding styles and language conventions in their projects. I can't count the number of times I've seen Swift code written with Objective-C stylings or JavaScript conventions brought over to Swift.

This lack of standardization will make your code difficult to read and may make your interviewer doubt your familiarity with the language and its conventions.

For the interview, I recommend you follow generally accepted Swift best practices and conventions regardless of your personal preferences. It's a small low-effort change, but it becomes one less thing the interviewer can dock you for.

If you're looking for a good place to get started, I'd check out this well-accepted Swift style guide: <https://github.com/raywenderlich/swift-style-guide>

Apologies for any unusual formatting in the following code samples. It was done in an attempt to make this book more readable.

## Assessment 1: Networking & Basic Codables

You can find the final implementation [here](#).

**Problem Duration:** On average, 60-90 minutes.

Over years of interviewing, as a candidate and as an interviewer, this is the most common question I've encountered. If there's anything you take away from this book, I hope it's this question.

The question takes a few different forms, but generally involves consuming some JSON response (either mocked local data or retrieved as a result of an HTTP request) and presenting the fetched data in a `UITableView` or `UICollectionView`.

I've seen the duration of this question range anywhere from a one hour on-site interview to a few days when it's part of a more involved take-home assessment.

When it's assigned as part of a take-home assessment, it will often include additional UI requirements like presenting a detail view, loading images, or even creating custom UI components, but regardless of the specifics, this problem is everywhere.

It's a great question as it touches on all the foundational concepts a iOS developer should know:

### Network Layer

- Foundation
- Asynchronous Tasks
- Error Handling
- Generics
- Codables
- URL Session

### Caching & UI

- Foundation
- UIKit
- AutoLayout
- Creating Custom Cells

- Concurrency

### Everything Else

- Architecture Patterns
- Design Patterns
- Delegates & Data Sources
- Coding Conventions
- Reusable Code
- Documentation
- Version Control

On GitHub, you'll find hundreds of variations of this interview problem. I've shortlisted a few variations in case you're curious, but at their core, they all revolve around making some API calls and presenting the results in a `UITableView`.

It's not my intention to comment on the code quality or architecture design of these solutions, I'm just sharing a few examples.

- <https://github.com/ekeren/ios-interview>
- <https://github.com/raywenderlich/ios-interview/tree/master/Practical>
- <https://github.com/shayan18/Runtastic-Test-iOS>
- <https://github.com/mediassumani/TakeHome-iOS-challenge>
- <https://github.com/changjeffrey829/IOSGeniusPlaza>
- <https://github.com/brandaorafael/iOS-Faire-Take-Home-Assignment>
- <https://github.com/fila/Egeniq-assignment>
- <https://github.com/giblerw/HA-TakeHome>
- <https://github.com/ygapps/Movies/tree/master/Movies/Networking>
- <https://github.com/valizairov/AppStoreLightweight>
- <https://github.com/armaluca/clearScore>
- <https://github.com/ka-ching-as/take-home-assignment-junior-ios>

*In my experience, there have been certain variations of this question that allowed the use of third-party libraries, but it was strongly discouraged. A library, like Alamofire, would be overkill for an assignment like this.*

Let's get back to the implementation.

## Managing Endpoints

In order to build our networking layer, we'll need to come up with an elegant way of representing our application's various endpoints.

Let's start by identifying a few things that all API calls will have in common:

- HTTP Method (GET, POST, PUT, etc.)
- HTTP Scheme (HTTP & HTTPS)
- Base URL
- Path
- Query Parameters

We can model the HTTP method and the HTTP scheme with the following `enums`:

```
enum HttpMethod: String {  
    case delete = "DELETE"  
    case get = "GET"  
    case patch = "PATCH"  
    case post = "POST"  
    case put = "PUT"  
}  
  
enum HTTPScheme: String {  
    case http  
    case https  
}
```

Next, in order to represent a generic endpoint, we can create an API protocol which any endpoint we wish to support must conform to:

```

/// The API protocol allows us to separate the task of constructing a URL,
/// its parameters, and HTTP method from the act of executing the URL request
/// and parsing the response.
protocol API {
    /// .http or .https
    var scheme: HTTPScheme { get }

    // Example: "maps.googleapis.com"
    var baseURL: String { get }

    // "/maps/api/place/nearbysearch/"
    var path: String { get }

    // [URLQueryItem(name: "api_key", value: API_KEY)]
    var parameters: [URLQueryItem] { get }

    // "GET"
    var method: HTTPMethod { get }
}

```

For example, if we wanted to add support for the Google Places API, we could do so like this:

```

enum GooglePlacesAPI: API {
    case getNearbyPlaces(searchText: String?, latitude: Double,
                         longitude: Double)

    var scheme: HTTPScheme {
        switch self {
            case .getNearbyPlaces:
                return .https
        }
    }

    var baseURL: String {
        switch self {
            case .getNearbyPlaces:
                return "maps.googleapis.com"
        }
    }

    var path: String {
        switch self {
            case .getNearbyPlaces:
                return "/maps/api/place/nearbysearch/json"
        }
    }
}

```

```

        }
    }

    var parameters: [URLQueryItem] {
        switch self {
            case .getNearbyPlaces(let query, let latitude, let longitude):
                var params = [
                    URLQueryItem(name: "key", value: GooglePlacesAPI.key),
                    URLQueryItem(name: "language",
                                 value: Locale.current.languageCode),
                    URLQueryItem(name: "type", value: "restaurant"),
                    URLQueryItem(name: "radius", value: "6500"),
                    URLQueryItem(name: "location",
                                 value: "\(latitude),\n\(longitude)")
                ]
                if let query = query {
                    params.append(URLQueryItem(name: "keyword", value: query))
                }
                return params
            }
        }
    }

    var method: HTTPMethod {
        switch self {
            case .getNearbyPlaces:
                return .get
        }
    }
}

```

With this approach, not only can you represent any generic API, you can easily add support for any new API your project needs to integrate with by simply creating a new `enum`. For example, in a larger application, you could have separate `enums` to handle your authentication, booking, and checkout endpoints respectively.

This approach helps ensure the API's path, the scheme, and the required parameters are all readable, well-documented, and consolidated in one place.

As the networking demands of our application grows, the last thing we want to do is litter our codebase with multiple calls to `URLSession`. Instead, we should try and create some kind of

consolidated network service that can manage both making the request and parsing the response. As part of that implementation, we'll want our solution to be as reusable as possible, so generics would be a great language feature to leverage here.

We'll build out the networking service momentarily.

## Third Party Dependencies

Even though it's tempting to use frameworks like Alamofire and Kingfisher in your take-home assessment, I'd caution you against using them. Considering the limited scope of these assessments, we will never be able to fully utilize these tools. By using them for such a simple application, it gives off the impression that you are uncomfortable with `URLSession` which would clearly work against you in the interview. Moreover, we don't want to give the interviewer the impression that our approach for solving difficult technical problems is to introduce additional dependencies.

There's always a time and a place to add new dependencies to a project, but every additional dependency increases the codebase's maintenance costs and risks its stability.

In my experience, there have been several times when a new version of Xcode or iOS breaks one of our dependencies and we are left waiting for the developer to provide a fix before we can continue releasing our application. Not to mention, this assumes that the third-party dependency is still actively maintained!

Many times, we rely on large libraries only to utilize a small portion of their capabilities. In certain situations, it may be easier to re-create the limited functionality you need rather than introducing yet another dependency.

While bringing in a new dependency might solve your current problem, it will increase your executable size, maintenance costs, potentially block future releases, be difficult to de-integrate, and will likely be a black box of functionality. Finally, if the dependency were to no-longer be maintained, then your only options would be to replace it with an alternative (if it exists), become the maintainers yourself, or re-create the necessary behavior from scratch - clearly, none of those options are ideal.

All that being said, if you choose to use a third-party dependency in your take-home project, you should be prepared to defend that decision.

## Building The Network Layer

This network layer implementation can be easily extended to support advanced networking capabilities without having to rely on external libraries.

With this approach, our `NetworkManager` is blind to the specifics of the API being used and the contents of the response.

That is exactly the behavior we want - our `NetworkManager` simply acts as a middleman. It's sole focus is on orchestrating the HTTP request, handling errors (if any), and passing along the response to the call site.

```
import Foundation

final class NetworkManager {
    /// Builds the relevant URL components from the values specified
    /// in the API.
    private class func buildURL(endpoint: API) -> URLComponents {
        var components = URLComponents()
        components.scheme = endpoint.scheme.rawValue
        components.host = endpoint.baseURL
        components.path = endpoint.path
        components.queryItems = endpoint.parameters
        return components
    }

    /// Executes the HTTP request and will attempt to decode the JSON
    /// response into a Codable object.
    /// - Parameters:
    ///   - endpoint: the endpoint to make the HTTP request to
    ///   - completion: the JSON response converted to the provided Codable
    ///     object when successful or a failure otherwise
    class func request<T: Decodable>(endpoint: API,
                                       completion: @escaping (Result<T, Error>)
                                         -> Void) {
        let components = buildURL(endpoint: endpoint)
        guard let url = components.url else {
            Log.error("URL creation error")
            return
        }

        var urlRequest = URLRequest(url: url)
```

```

urlRequest.httpMethod = endpoint.method.rawValue

let session = URLSession(configuration: .default)
let dataTask = session.dataTask(with: urlRequest) {
    data, response, error in
    if let error = error {
        completion(.failure(error))
        Log.error("Unknown error", error)
        return
    }

    guard response != nil, let data = data else {
        return
    }

    if let responseObject = try? JSONDecoder().decode(T.self,
                                                       from: data) {
        completion(.success(responseObject))
    } else {
        let error = NSError(domain: "com.AryamanSharda",
                            code: 200,
                            userInfo: [
                                NSLocalizedDescriptionKey: "Failed"
                            ])
        completion(.failure(error))
    }
}

dataTask.resume()
}
}

```

## Fetching Remote Images

To reduce the number of HTTP requests we make, we only want to fetch images for cells that are currently visible and for images we do not already have cached. In the absence of caching, we would have to create a new HTTP request every time a cell was shown, which would result in a lot of repeated requests and would degrade the user experience significantly.

The following `extension` on `UIImageView` allows us to both fetch and cache images across our application:

```
let imageCache = NSCache<NSString, UIImage>()
```

```
extension UIImageView {
    /// Loads an image from a URL and saves it into an image cache, returns
    /// the image if already available in the cache.
    /// - Parameter urlString: String representation of the URL to load the
    /// image from
    /// - Parameter placeholder: An optional placeholder to show while the
    /// image is being fetched
    /// - Returns: A reference to the data task in order to pause, cancel,
    /// resume, etc.
    @discardableResult
    func loadImageFromURL(urlString: String,
                           placeholder: UIImage? = nil) ->
        URLSessionDataTask? {
        self.image = nil

        let key = NSString(string: urlString)
        if let cachedImage = imageCache.object(forKey: key) {
            self.image = cachedImage
            return nil
        }

        guard let url = URL(string: urlString) else {
            return nil
        }

        if let placeholder = placeholder {
            self.image = placeholder
        }

        let task = URLSession.shared.dataTask(with: url) { data, _, _ in
            DispatchQueue.main.async {
                if let data = data,
                    let downloadedImage = UIImage(data: data) {
                    imageCache.setObject(downloadedImage,
                                         forKey:
                                         NSString(string: urlString))
                    self.image = downloadedImage
                }
            }
        }
    }

    task.resume()
}
```

```
        return task
    }
}
```

## Final Implementation

Finally, we'll combine all of these calls together.

In our `UITableViewController`:

```
let endpoint = GooglePlacesAPI.getNearbyPlaces(searchText: query,
                                                latitude: latitude,
                                                longitude: longitude)

NetworkManager.request(endpoint: endpoint) { [weak self]
    (result: Result<NearbyPlacesResponse, Error>) in

    switch result {
    case .success(let response):
        self?.dataSource = response.results
        self?.tableView.reloadData()
    case .failure(let error):
        Log.error(error)
    }
}
```

And finally, in our custom `UITableViewCell`, we can leverage the extension we made on `UIImageView` earlier:

```
thumbnailImageView.loadImageFromURL(urlString: photoURL)
```

We've now created an extensible and reusable network layer, a clean way to represent APIs, and an extension for `UIImageView` that lets us fetch and maintain a cache of images without any third-party dependencies.

This implementation will be easy to reuse in any future take-home assessment.

You can find the final implementation [here](#).

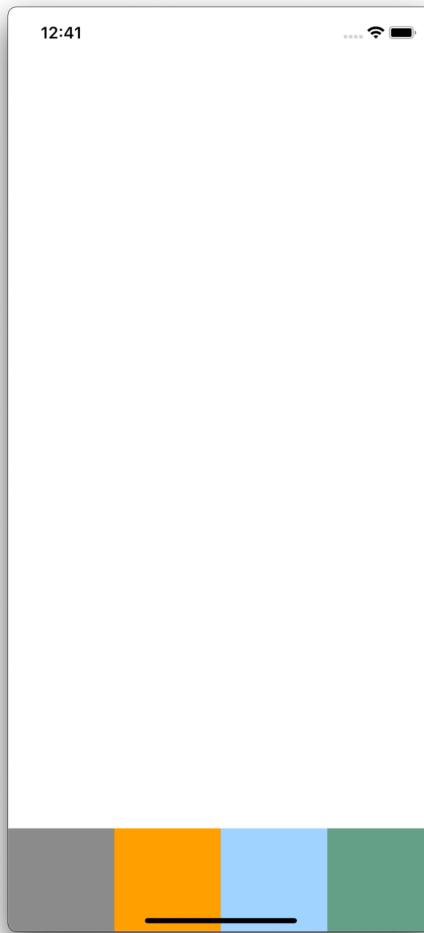
## Assessment 2: UIKit & User Interaction

**Interview Duration:** 50 minutes

You can find the starter project and the sample implementation [here](#).

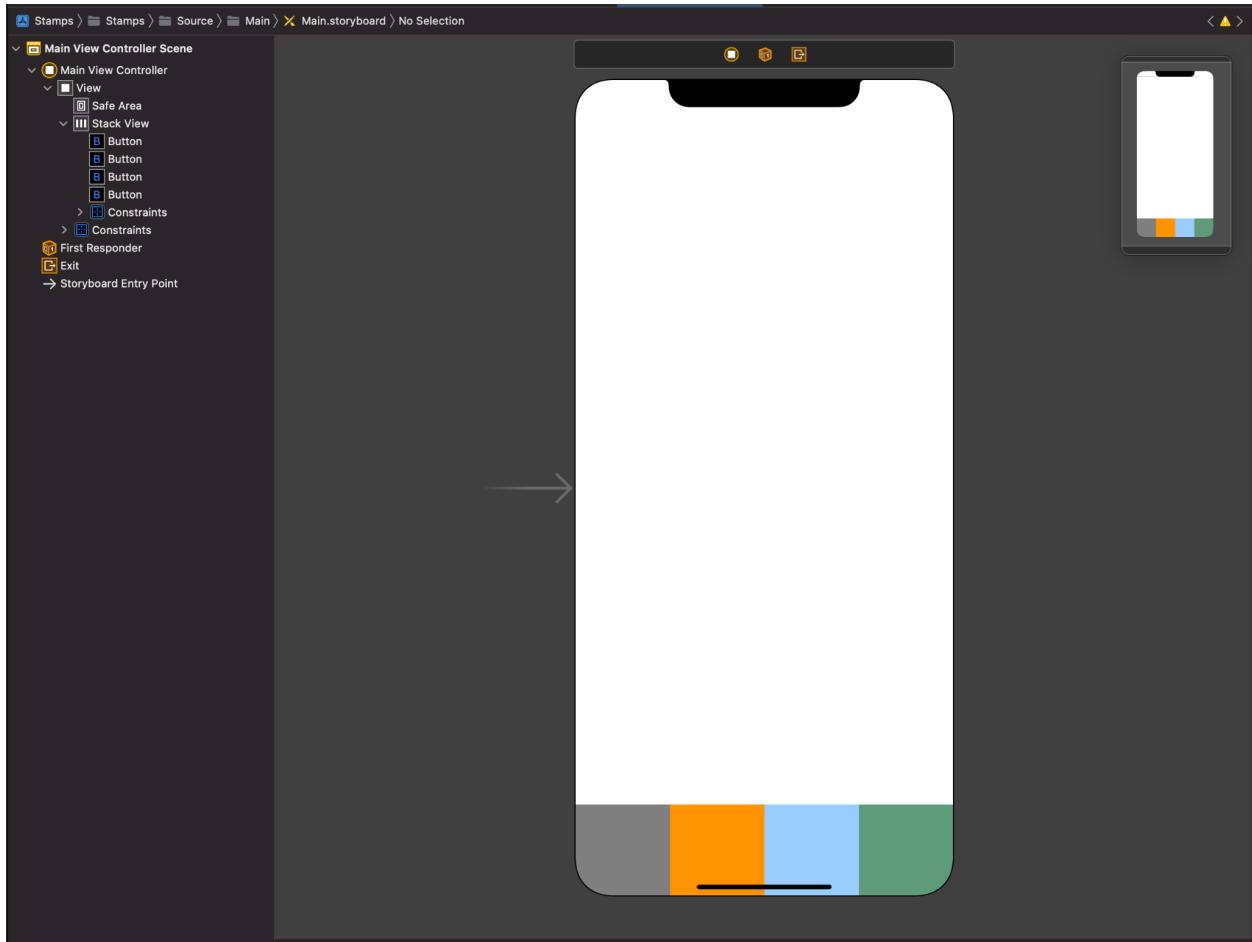
This is one of the most entertaining questions I've encountered during an iOS interview.

The starter Xcode project contains an app that looks like this:



It simply contains a `UIStackView` pinned to the bottom of the view with four equal width buttons, each with a unique background color.

Here's a look at the `.storyboard` file:



This is the starter code for the `UIViewController` shown above.

Currently, all it does is keep track of which `UIButton` has been selected and updates the `selectedColor` property to reflect this state. It also changes the alpha value of the selected `UIButton` to visually indicate that it is the current selection.

```
import UIKit

final class MainViewController: UIViewController {

    // MARK: - Object Outlets
    @IBOutlet private var stampButtons: [UIButton]!

    // MARK: - Private Objects
    private var selectedColor: UIColor = .black
}
```

```
// MARK: - Action Outlets
private extension MainViewController {
    @IBAction func stampTapped(_ sender: UIButton) {

        // Extract stamp color from the button's background color.
        guard let stampColor = sender.backgroundColor else {
            return
        }

        // Update internal values.
        selectedColor = stampColor
        updateButtonColors(selectedButton: sender)
    }
}

// MARK: - Helper Methods
private extension MainViewController {
    func updateButtonColors(selectedButton: UIButton) {

        // Deselect all the color buttons.
        stampButtons.forEach { $0.alpha = 1.0 }

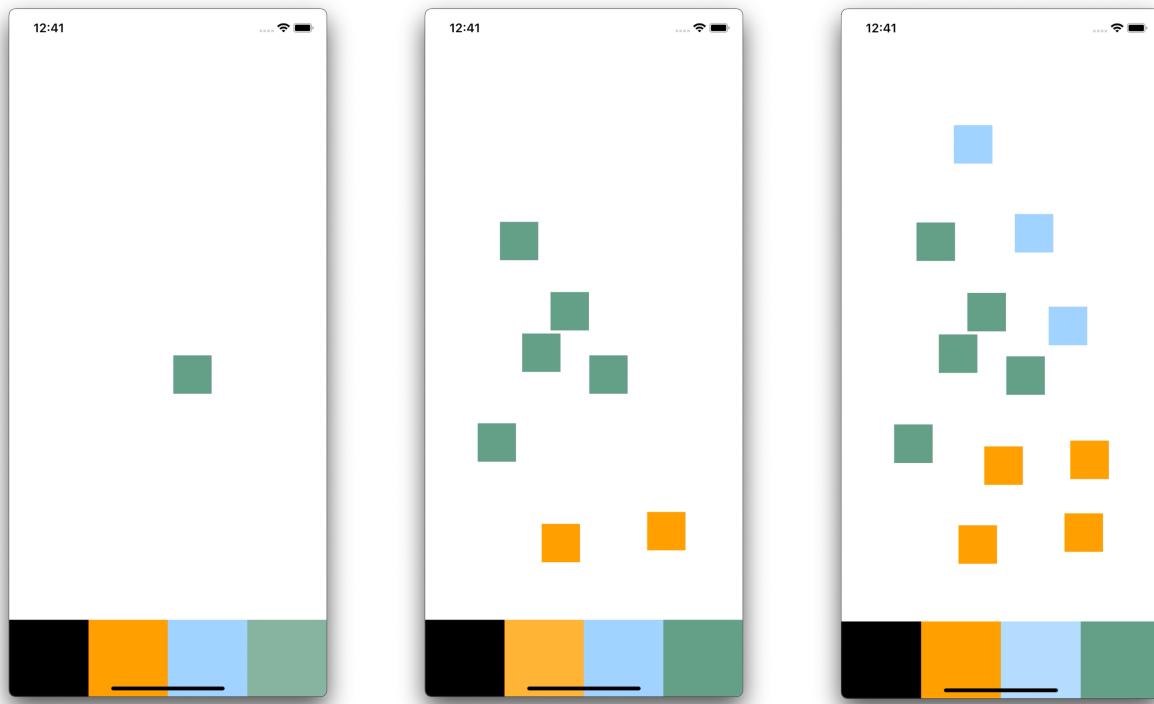
        // Select the relevant color button.
        selectedButton.alpha = 0.8
    }
}
```

Nothing crazy so far - it's just boilerplate code to set up the upcoming tasks. At this point, the interviewer informs me that they will show me a series of videos that each demonstrate functionality they want me to replicate.

If you'd like to follow a video walkthrough of this problem, you can find it on my YouTube channel here: <https://www.youtube.com/watch?v=lZHNi5PZr4E>.

## Add

The first task is to place a small 50 x 50 pixel square at the location where the user taps on the empty view.



One important question to ask the interviewer is should the new view be centered at the tap location or is that the origin point for the view. In this case, the interviewer tells me that it needs to be centered at that location, so I know I'll have to adjust the X and Y values of the new view to ensure the center matches the tap location.

Throughout this entire process, I'm talking out loud. This not only helps me demonstrate my thought process, but also gives the interviewer an opportunity to intervene if I'm going down the wrong path.

I mentioned that I was thinking about adding a `UITapGestureRecognizer` to the view and in the callback using that somehow to identify the location of the tap event and add the view at that position. At this point, the interviewer hints that `UITapGestureRecognizer` has a `location()` function that returns the `CGPoint` of the tap event.

Now, this isn't something you'd be expected to know; it's an obscure little known function. But, I'd imagine the rationale behind this is to force the candidate to talk through their approach out loud. Otherwise, without this hint, we could have likely come to the same answer by using Apple's documentation directly.

As a quick aside, I've been on the other side of this interview and asked potential candidates the same question. Oftentimes, they'll tend to overcomplicate the solution and try to manipulate `CALayer` or override `touchesBegan()`. In most cases, the initial question of an interview serves no purpose other than to introduce the problem whose constraints and functionality you'll iterate on during the course of the interview.

So, don't overcomplicate things and first focus on getting a working implementation out there - you can always optimize later. Even if the implementation is subpar, it's better to have something to show for the interview than to spend all your time trying to be clever and ending the interview with an incomplete implementation.

Alright, back to the problem.

Now that we know we're on the right track with the `UITapGestureRecognizer` and that we can get the tap location from it, it's time to add our square to the view.

The simplest method I could think of was just to create a `UIView`, set the frame to 50 x 50 pixels, assign the `backgroundColor` property, and set the `center` to be the coordinates of the tapped location.

The code for the first part looks like this:

```
override func viewDidLoad() {
    super.viewDidLoad()
    view.addGestureRecognizer(UITapGestureRecognizer(target: self,
                                                action: #selector(handleTap(_))))
}

@objc func handleTap(_ sender: UITapGestureRecognizer) {
    addSquare(with: selectedColor, at: sender.location(in: view))
}

func addSquare(with color: UIColor, at location: CGPoint) {
    let sideLength = 50.0
    let centerOffset = sideLength / 2.0
    let square = UIView(frame: CGRect(x: location.x - centerOffset,
                                       y: location.y - centerOffset,
                                       width: sideLength,
                                       height: sideLength))
```

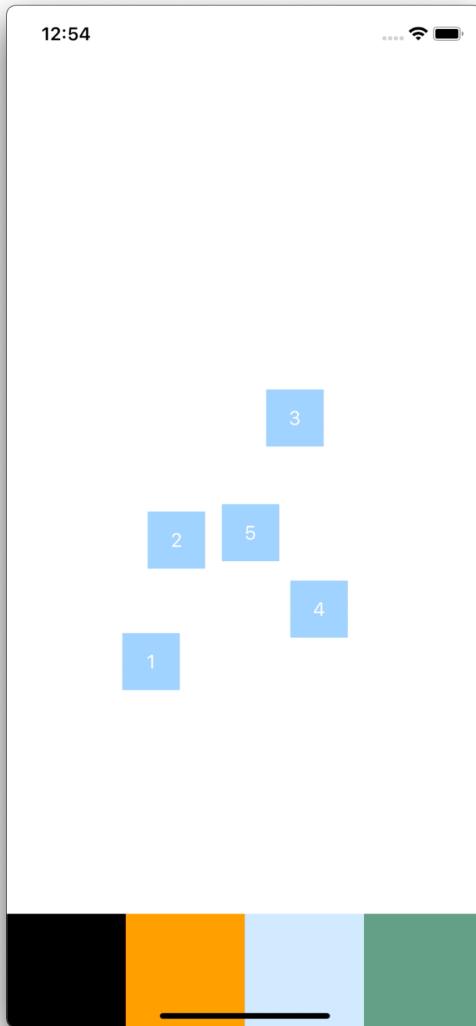
```
    square.backgroundColor = selectedColor  
    view.addSubview(square)  
}
```

Making just these changes to the starter implementation was enough to replicate the requested functionality.

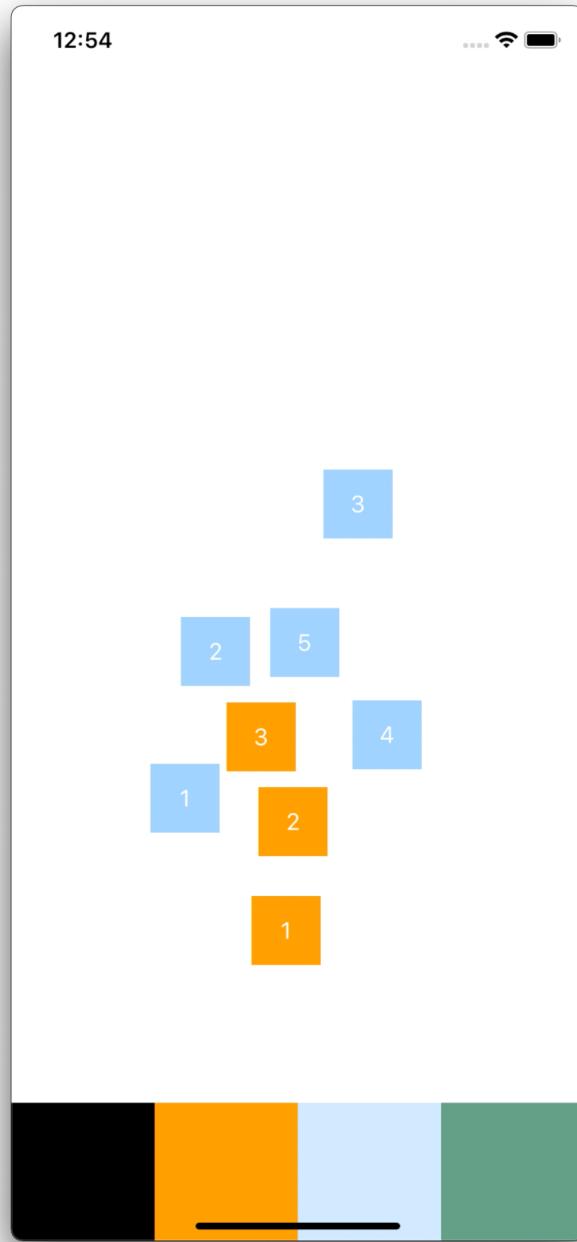
## Count

Now, the new requirement asks that we show the respective count of each particular color in the center of the square we're adding to the screen. To be clear, it's not a total count of squares, but a running count of squares of each specific color.

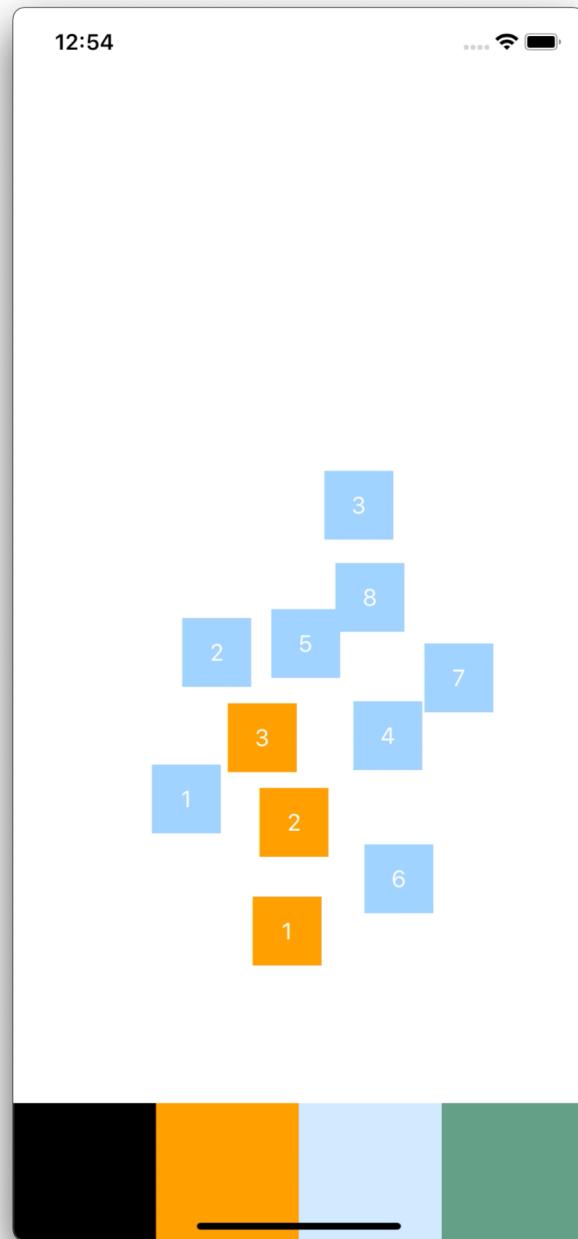
Once you select a color from the bottom tray, every time you tap on the center view, we'll continue to add the  $50 \times 50$  pixels square, but now we'll show how many squares of that color are on the screen.



Since the count is specific to each color, when we change the `selectedColor` to orange, the count should start from 1 again:



If we return to a previous color, the count should pick up where we left off:



Let's break these new requirements down.

Firstly, in order to show the count in the square, my initial thought was just to add a `UILabel` to our `UIView` with the text set to be whatever the count was at that time. But, I remembered that a `UILabel` inherits from `UIView` which means that a `UILabel` also has a `backgroundColor` property. So, instead of adding a `UILabel` to our existing `UIView`, we can just change our implementation to use a `UILabel` instead.

Next, I needed some way of keeping track of how many squares of a certain color had been added to the screen. As I continued to talk through my approach, I explained that the naive solution would be to create separate variables for each color, and use those variables to track the number of squares in that color. I acknowledged that this wasn't a scalable solution because if we tried to add additional colors, we'd need to introduce new variables for each of them and our code would get very messy very quickly.

I mentioned that ideally I'd use a `Dictionary` where the key would be the `UIColor` and the value would be the respective count, but I wasn't sure if `UIColor` was `Hashable` which would've been a prerequisite for this approach.

Since I was talking aloud the entire time, the interviewer was able to follow my approach and they hinted that I was on the right track and that before I went any further, I should try and figure out if `UIColor` was indeed `Hashable`.

So, using Xcode's "Jump to Definition" feature on `UIColor`, I saw that it inherited from `NSObject`:

```
open class UIColor : NSObject, NSSecureCoding, NSCopying {...}
```

Going one step further, I noticed that `NSObject` implemented `Hashable` which meant that `UIColor` was also `Hashable`.

```
extension NSObject : Equatable, Hashable {..}
```

This meant that my `Dictionary` approach would work and I'd be able to provide a much more scalable solution than having to maintain variables to keep track of each individual color.

```
// MARK: - Private Objects
private var colorCount: [UIColor: Int] = [:]
```

Here's our updated implementation of addSquare():

```
func addSquare(with color: UIColor, at location: CGPoint) {
    let sideLength = 50.0
    let centerOffset = sideLength / 2.0
    let square = UILabel(frame: CGRect(x: location.x - centerOffset,
                                        y: location.y - centerOffset,
                                        width: sideLength,
                                        height: sideLength))

    square.backgroundColor = selectedColor
    square.textAlignment = .center
    square.textColor = .white

    if let existingColorCount = colorCount[selectedColor] {
        let updatedCount = existingColorCount + 1
        colorCount[selectedColor] = updatedCount
        square.text = String(updatedCount)
    } else {
        square.text = "1"
        colorCount[selectedColor] = 1
    }

    view.addSubview(square)
}
```

Great! Now, we've completed the second task.

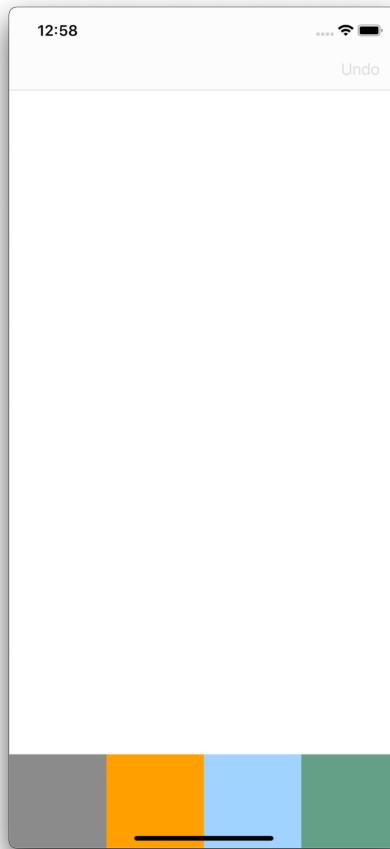
## Undo

In this step, we're tasked with adding an Undo button that will allow us to remove the most recently added square from the view.

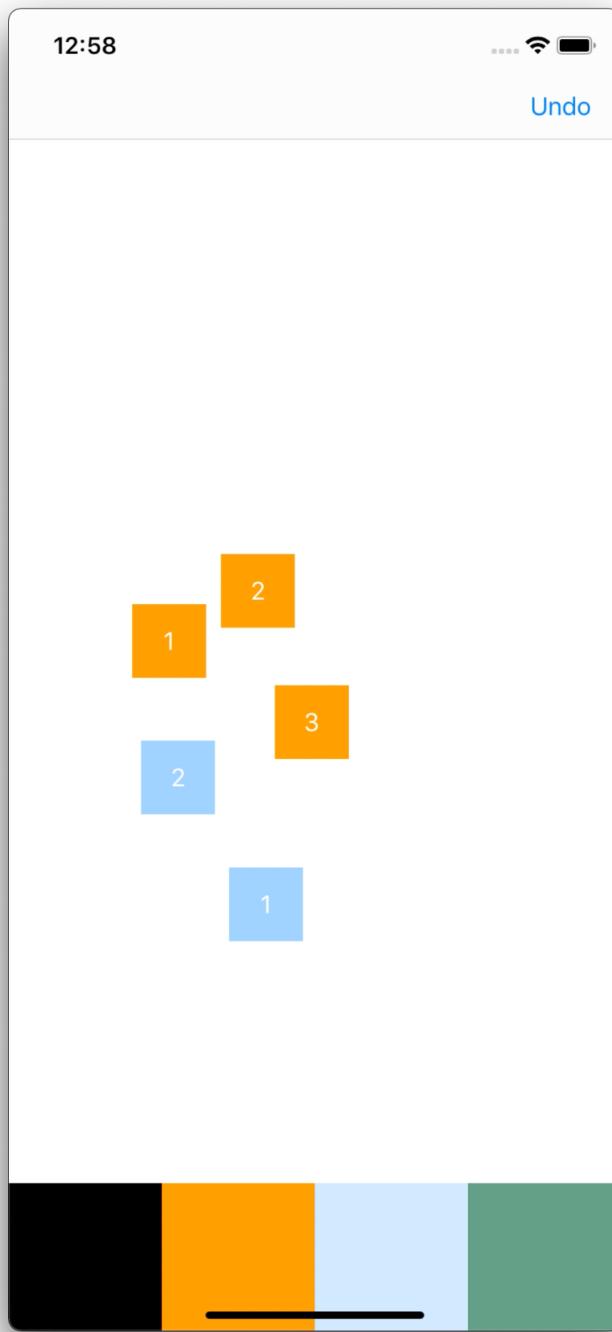
There's a few other points to mention that are only noticeable in the video:

- The Undo button should be disabled if there are no operations to revert.
- Elements should be removed in the order they were added; the most recently added square should be the first one removed and so on.
- The Undo operation isn't color specific - it can work across different colors. So, if you add a blue square and then an orange square, it should undo the orange square first and then the blue one.

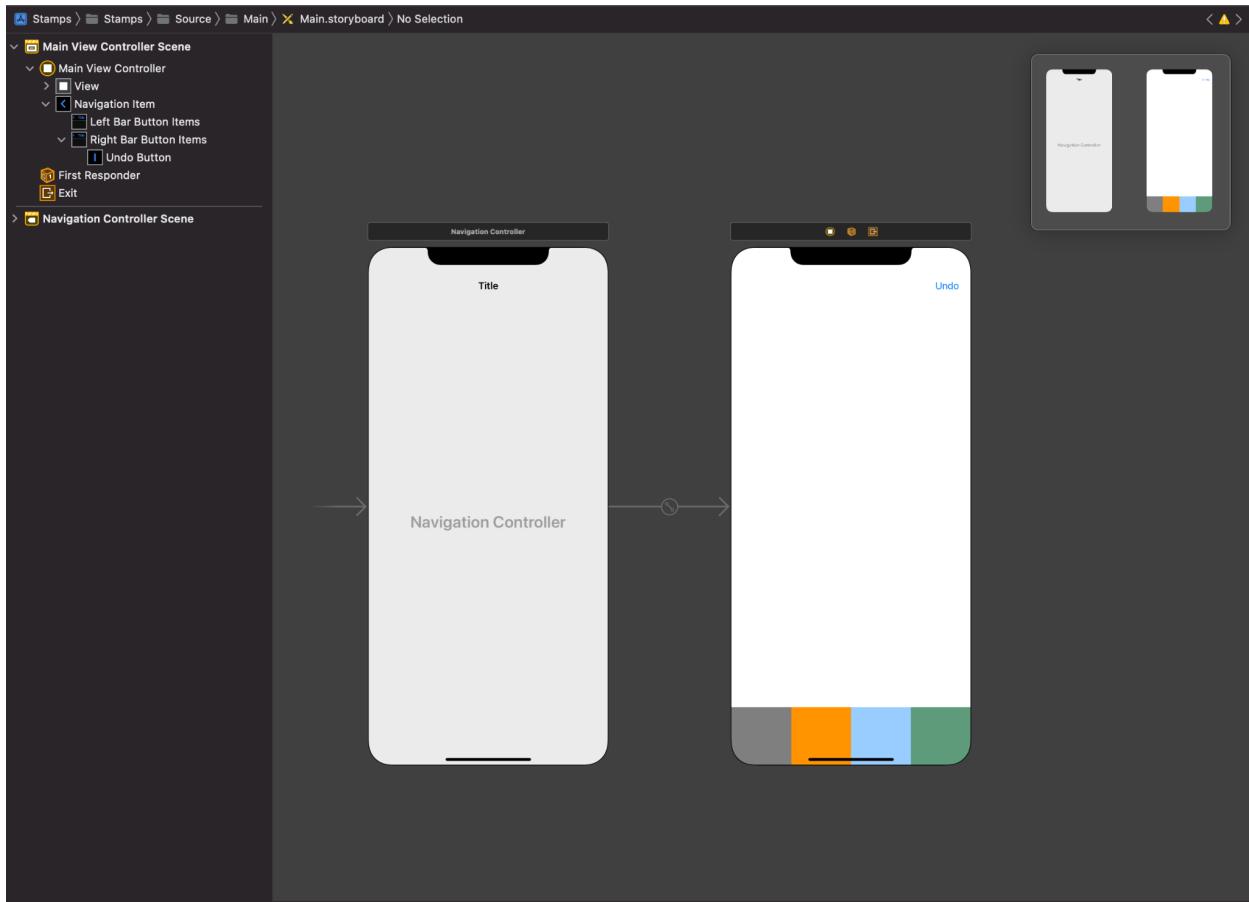
Notice that a `UINavigationBar` has been added to the view and that the Undo button is initially disabled since there are currently no operations to revert.



Now that we've added squares to our view, the Undo button is enabled.



We can easily add the Undo button and `UINavigationBar` by embedding our `UIViewController` in a `UINavigationController` and manually adding a `UIBarButtonItem`.



Next, we know that we want to remove these views in a “last in first out order”, so that should tell us we’ll need to use a stack. In Swift, we can use our arrays like a stack by taking advantage of the `append()` and `popLast()` functions.

We’ll need to create a reference to our Undo button so we can easily enable and disable it.

```
// MARK: - Object Outlets
@IBOutlet private var undoButton: UIBarButtonItem!
```

We’ll also need an array to serve as the stack. Any time we add or remove an element from the stack, we can use the `isEmpty` property to help us decide whether the `undoButton` should be enabled or disabled.

```
// MARK: - Private Objects
```

```
private var undoLabels: [UILabel] = [] {
    didSet {
        undoButton.isEnabled = !undoLabels.isEmpty
    }
}
```

On app launch, the `didSet` on `undoLabels` isn't triggered, so we have to specify the initial state of our `undoButton` explicitly.

```
override func viewDidLoad() {
    super.viewDidLoad()
    view.addGestureRecognizer(UITapGestureRecognizer(target: self,
                                                action: #selector(handleTap(_:))))
    undoButton.isEnabled = false
}
```

Next, for the actual logic behind the Undo button, we just need to remove the last element from our `undoLabels` array, if it exists, and update our respective `colorCount`.

```
@IBAction func undoPressed(_ sender: UIBarButtonItem) {
    // Grab the last label from the array (most recently added).
    guard let lastLabel = undoLabels.popLast() else {
        return
    }

    // Update color count
    if let labelColor = lastLabel.backgroundColor,
       let count = colorCount[labelColor] {
        colorCount[labelColor] = count - 1
    }

    // Remove subview
    lastLabel.removeFromSuperview()
}

func addSquare(with color: UIColor, at location: CGPoint) {
    let sideLength = 50.0
    let centerOffset = sideLength / 2.0
    let square = UILabel(frame: CGRect(x: location.x - centerOffset,
                                         y: location.y - centerOffset,
                                         width: sideLength,
```

```
height: sideLength))

square.backgroundColor = selectedColor
square.textAlignment = .center
square.textColor = .white

if let existingColorCount = colorCount[selectedColor] {
    let updatedCount = existingColorCount + 1
    colorCount[selectedColor] = updatedCount
    square.text = String(updatedCount)
} else {
    square.text = "1"
    colorCount[selectedColor] = 1
}

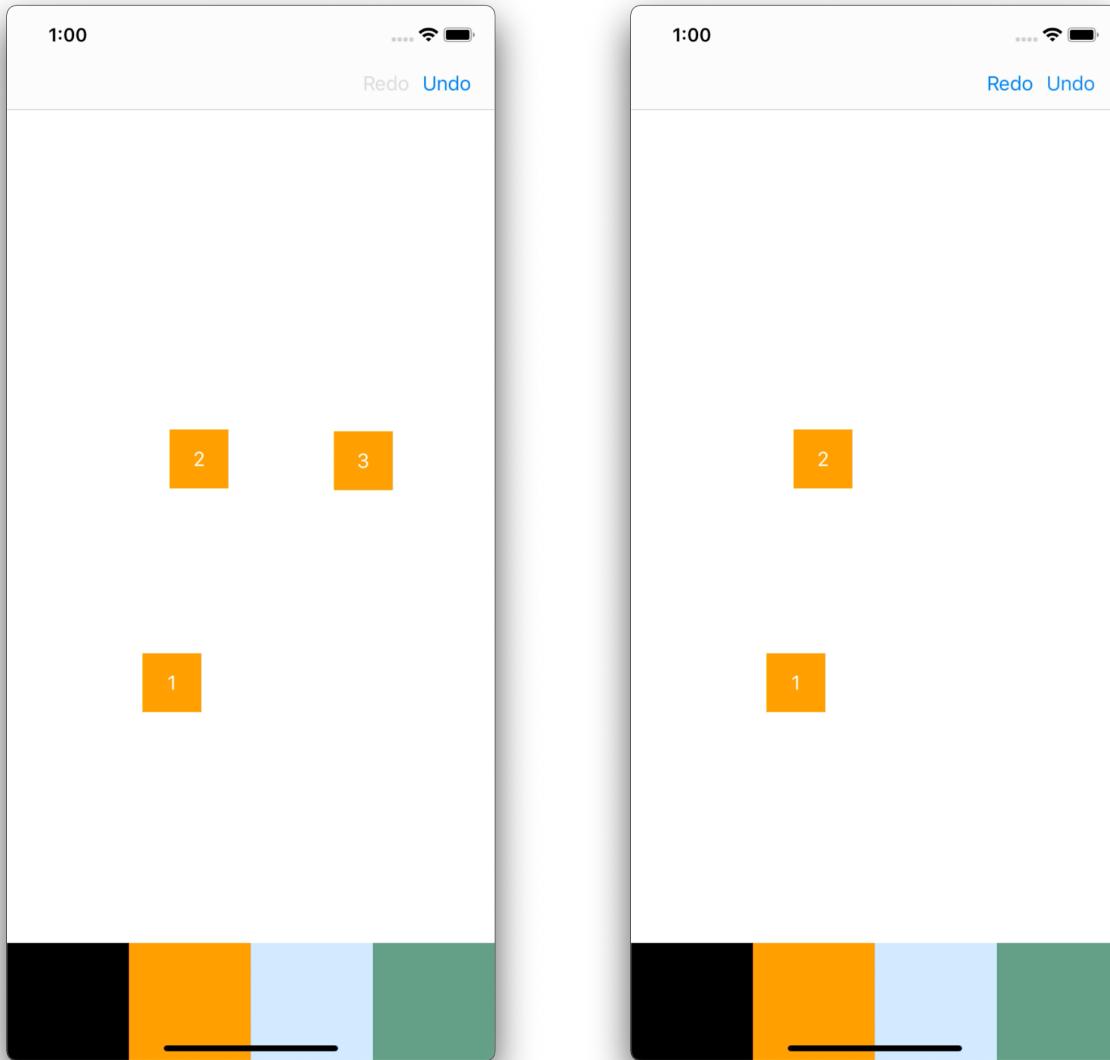
view.addSubview(square)
undoLabels.append(square)
}
```

With all of this in place, our Undo functionality works as expected.

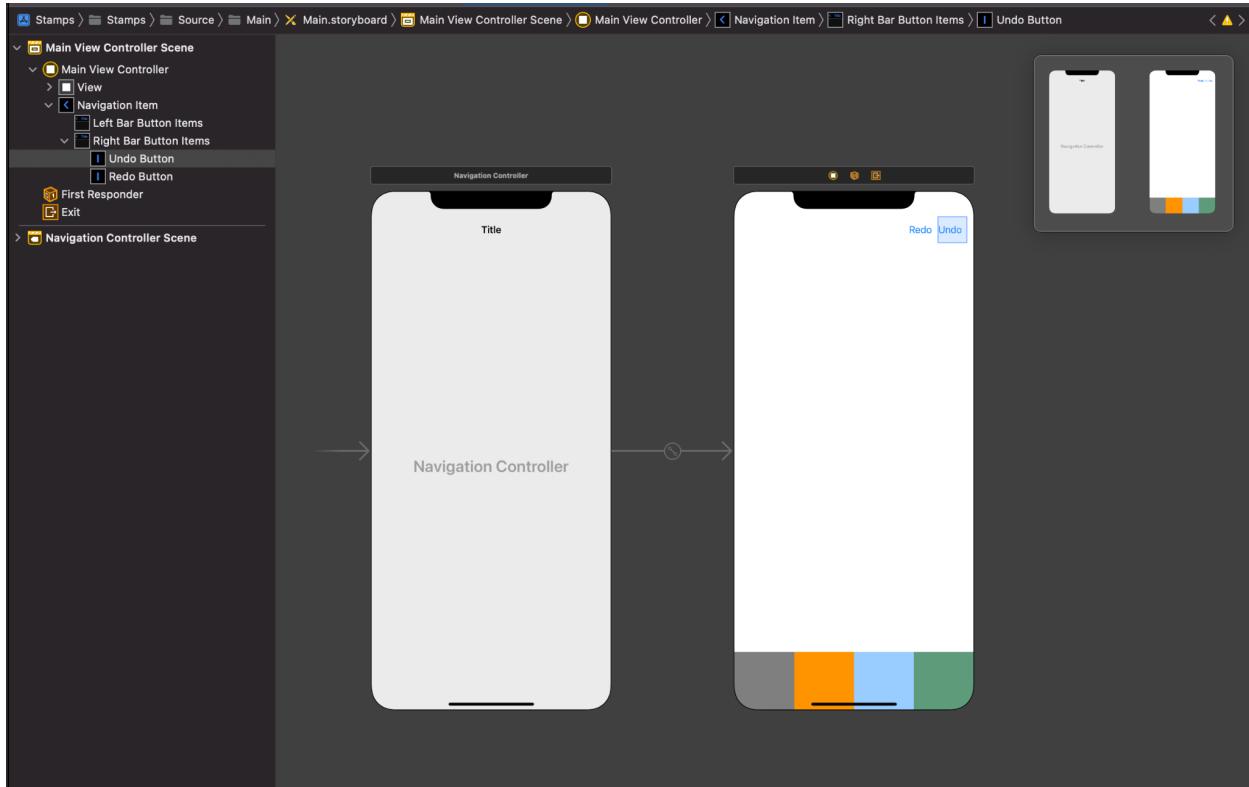
## Redo

For the final part of the interview, we're tasked with adding support for a Redo operation which will add back whatever the Undo operation removed. Similar to the Undo button, the Redo button should only be enabled if there is some operation to reverse. If the user never undoes anything, Redo should never be enabled.

In the following screenshots, clicking Redo would return the third orange square to the view in its original position.



The implementation for Redo is fairly simple. We'll start by creating an **IBOutlet** so we can easily enable and disable the Redo button:



```
// MARK: - Object Outlets
@IBOutlet private var redoButton: UIBarButtonItem!

override func viewDidLoad() {
    super.viewDidLoad()

    let tapGesture = UITapGestureRecognizer(target: self,
                                         action:
                                         #selector(handleTap(_:)))
    view.addGestureRecognizer(tapGesture)

    undoButton.isEnabled = false
    redoButton.isEnabled = false
}

// MARK: - Private Objects
private var redoLabels: [UILabel] = [] {
    didSet {
        redoButton.isEnabled = !redoLabels.isEmpty
    }
}
```

```
    }
}
```

Now, anytime we remove a view, we can take that view and add it to a new stack that manages the “redo” operations. We’ll need to update our `undoPressed()` implementation and introduce a new `redoPressed()` function:

```
@IBAction func undoPressed(_ sender: UIBarButtonItem) {
    // Grab the last label from the array.
    guard let lastLabel = undoLabels.popLast() else {
        return
    }

    redoLabels.append(lastLabel)

    // Update color count.
    if let labelColor = lastLabel.backgroundColor,
        let count = colorCount[labelColor] {
        colorCount[labelColor] = count - 1
    }

    // Remove subview.
    lastLabel.removeFromSuperview()
}

@IBAction func redoPressed(_ sender: UIBarButtonItem) {
    // Grab the last label from the array.
    guard let lastLabel = redoLabels.popLast() else {
        return
    }

    // Adds label back to the undo list of labels.
    undoLabels.append(lastLabel)

    // Update color count.
    if let labelColor = lastLabel.backgroundColor,
        let count = colorCount[labelColor] {
        colorCount[labelColor] = count + 1
    }

    // Remove subview.
    view.addSubview(lastLabel)
```

{

We now have support for both Undo and Redo operations and the view will always be added back at the same location it was removed from.

You can find the starter project and the sample implementation [here](#).

## Assessment 3: Recreating Mockups & AutoLayout

Often, onsite iOS interviews will consist of an AutoLayout focused exercise. You will be given an annotated mockup of a view and asked to implement it from scratch. In my experience, this exercise almost always involves recreating the view using .storyboards and .xibs rather than implementing the view programmatically.

While these types of questions may seem contrived, I believe they're quite reasonable given how important AutoLayout is to the iOS development process, especially when you're developing Universal apps. Fortunately though, these types of questions are fairly easy to prepare for.

In my opinion, the best way to prepare is simply by playing around with the company's app. Along the way, you'll likely find a handful of views that are visually intricate or involve complex constraints. These views are useful to practice with since the mockup that you'll encounter onsite will most likely be a variation of one of the app's existing views.

Whenever I've encountered this type of exercise, it's always involved recreating an existing view from the company's production app. This shouldn't be a surprise as companies will often use exercises like this as a litmus test of your abilities to do the day to day work the job would require.

I'd recommend recreating the five most prominent views from their app. You should continuously practice these views until you can achieve pixel-perfect recreations without any hiccups and, more importantly, without any breaking or unnecessary constraints.

During an interview, you won't have time to figure out the correct `UIStackView` alignment and distribution style, debug broken constraints, etc. The more practice you have recreating these views, the larger the time buffer you will be able to create for yourself to deal with new complications. Moreover, debugging constraint issues, especially those related to Content Hugging Priority and Content Resistance Priority, are far easier outside the high-pressure environment of an interview.

Basically, the goal of this interview is to try and determine how comfortable you are with using constraints to layout views and whether you understand basic AutoLayout features. I can't tell you how many interviews I've conducted in which candidates aimlessly click through all of the buttons and sections in Xcode looking for where to update constraints, change margins, specify

custom insets, or tweak the Content Resistance and Content Hugging Priorities. This type of behavior makes the interviewer doubt your competence with these fundamental technologies.

I know I'm repeating myself, but I strongly suggest you get reacquainted with this style of UI development. While you may not expect to use this approach on the job or prefer to make views programmatically, it may very well appear as part of the interview process and should be an easy win.

While you practice, make sure to check your implementation on small devices, different device orientations, and with "double length pseudolanguage" enabled to ensure text is not clipped and all views resize properly. Interviewers are trained to look for these edge cases and are able to pick up on situations where your implementation wouldn't handle them.

In order to make sure I didn't make the same silly mistakes over and over again, I created a checklist for myself that I use on every project: [iOS Checklist](#). You may want to maintain your own cheat sheet of common mistakes you make when it comes to UI design and testing so you don't make the same mistakes in the actual interview.

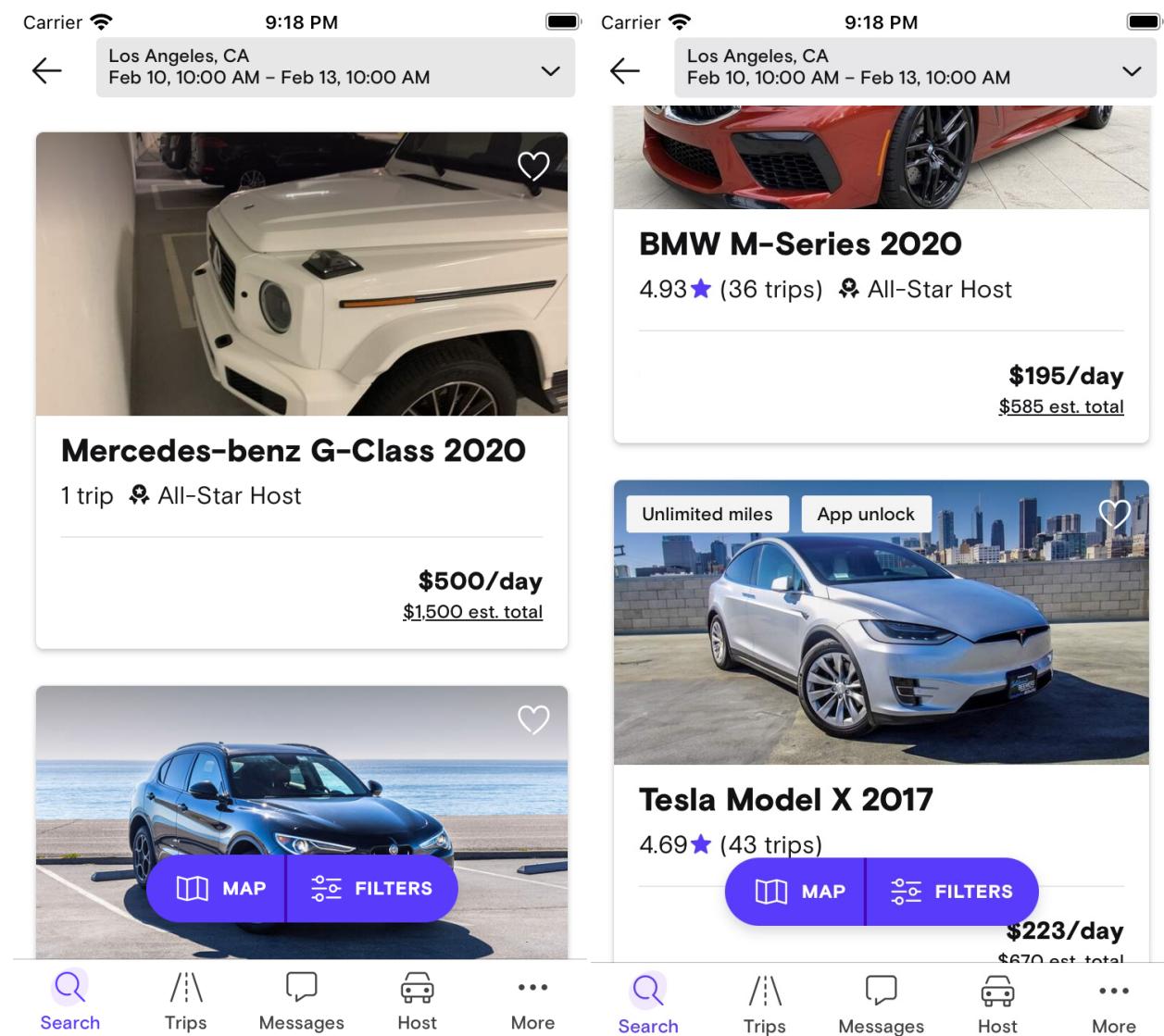
Alright, enough talking. Let's get started!

You can find the completed implementation [here](#).

## Mockup

We'll try and recreate a simplified version of the following view from Turo. As a quick aside, if you're interested in joining the iOS team at Turo, feel free to [apply here](#).

Let's ignore the custom navigation bar, tab bar, and floating buttons and instead focus on recreating a simplified version of the cell's layout. In the versions of this interview that I've experienced, the primary focus has always been on AutoLayout rather than stylistic topics like shadows, custom fonts, cornerRadius, etc. As such, I'll consider them out of scope for this walkthrough, but feel free to include it in your preparation.

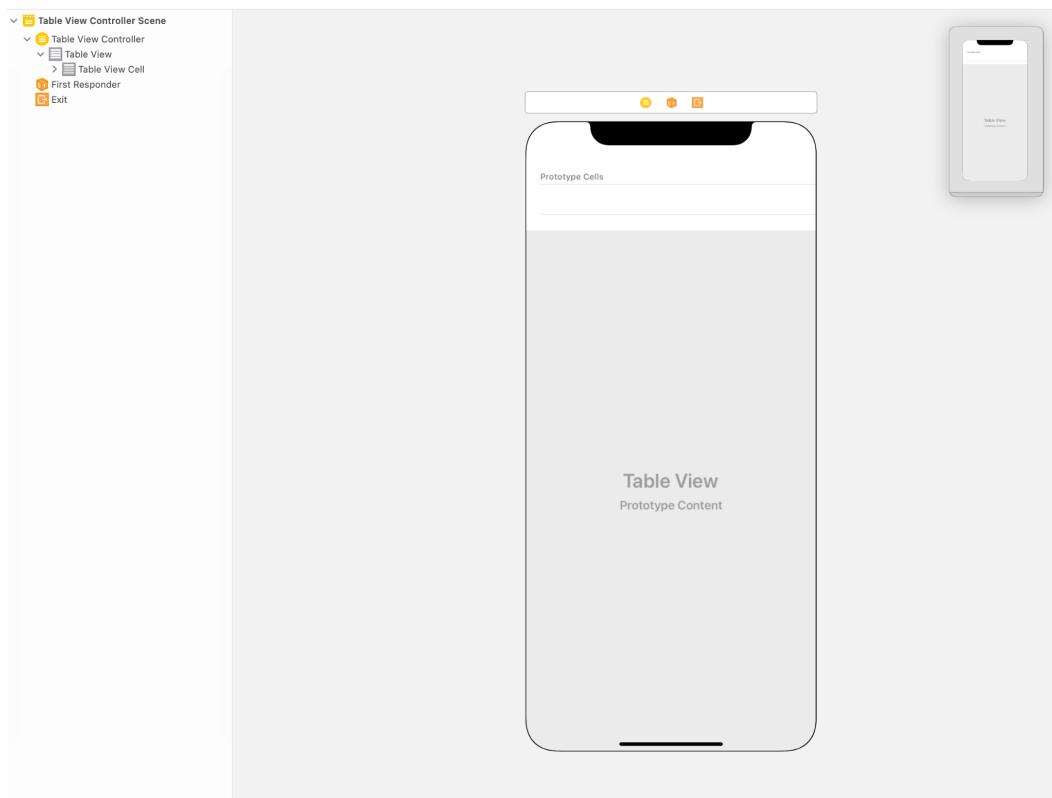


## Building The Vehicle View

I'd encourage you to try this problem yourself before you review the following sample implementation.

Typically, these types of interviews include an annotated mockup showing the dimensions of all of the subviews, but when you are practicing, it's perfectly reasonable to make educated guesses about the specifics.

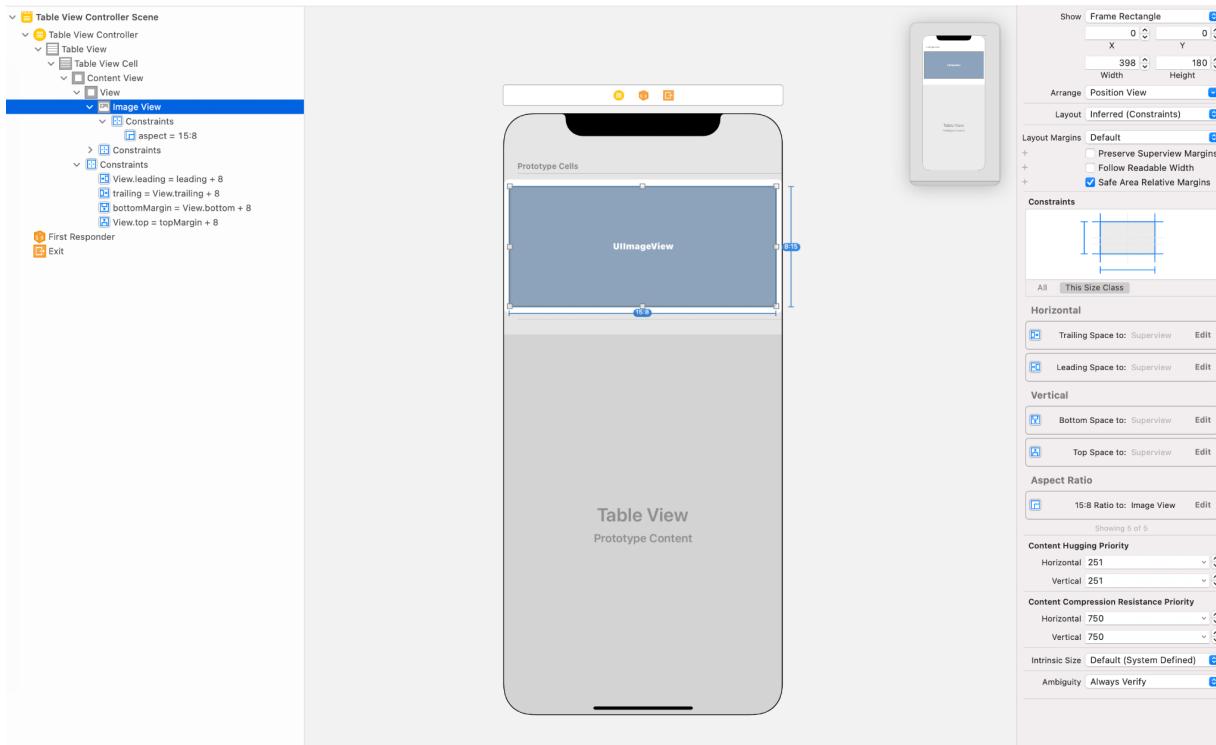
I've gone ahead and set up a `UITableView` with a single empty prototype `UITableViewCell`:



As we can see from the mockup, the final cell has rounded corners as well as a custom inset from the sides of its parent view. To make the `UITableView`'s `contentView` adhere to these custom settings will be tricky. So, we'll apply this custom styling to a new container view instead.

Additionally, the mockup shows that all of the vehicle images are the same size, so the aspect ratio / width and height must be hardcoded. In most cases, this would be specified in the mockup writeup you would be provided, but for the moment we'll have to make an educated

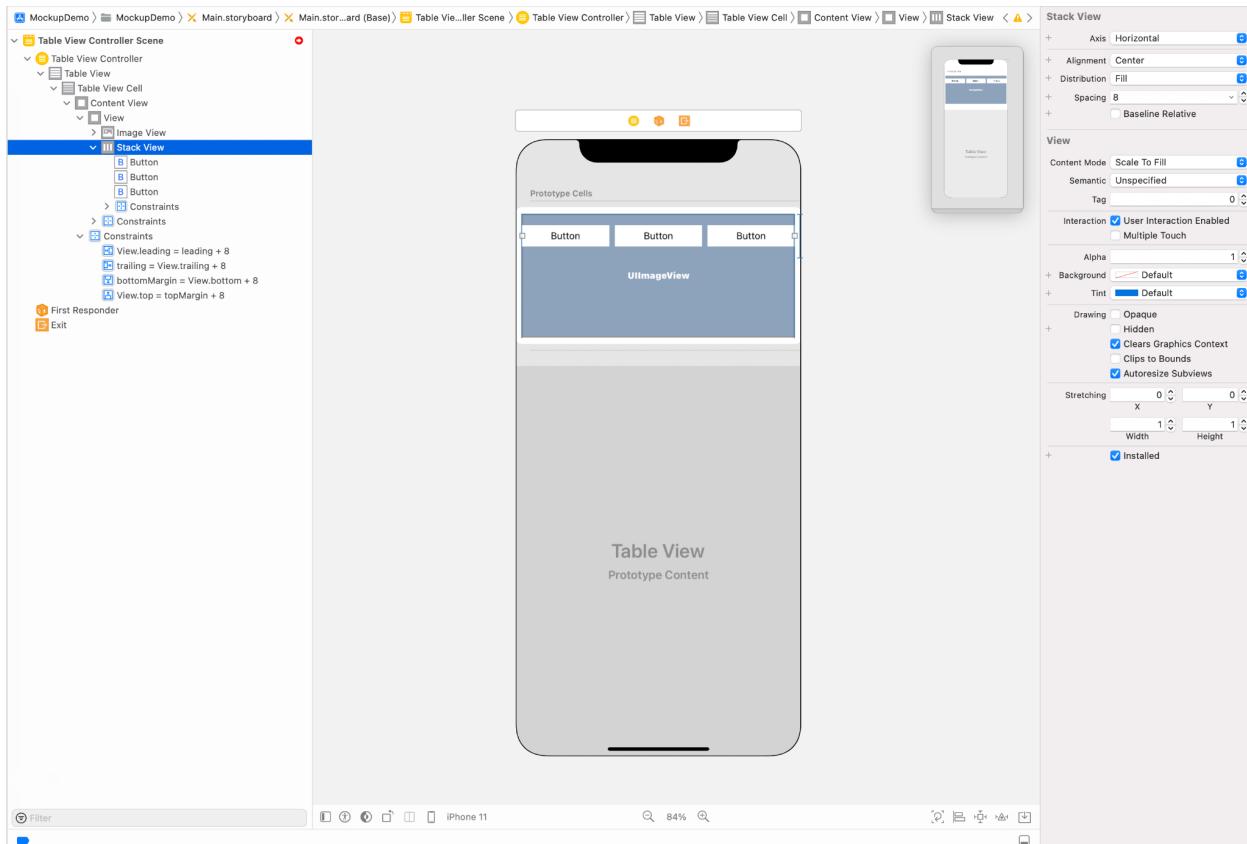
guess.



*Adding ImageView With Custom Aspect Ratio*

The mockup also shows that the top portion of the cell can display different tags (i.e. App unlock, Unlimited miles, etc.) and has a Favorites button pinned to the top-right. Interestingly though, it seems that the tags do not appear in every cell. So, we'll need to implement an easy way of hiding and showing those tags as needed.

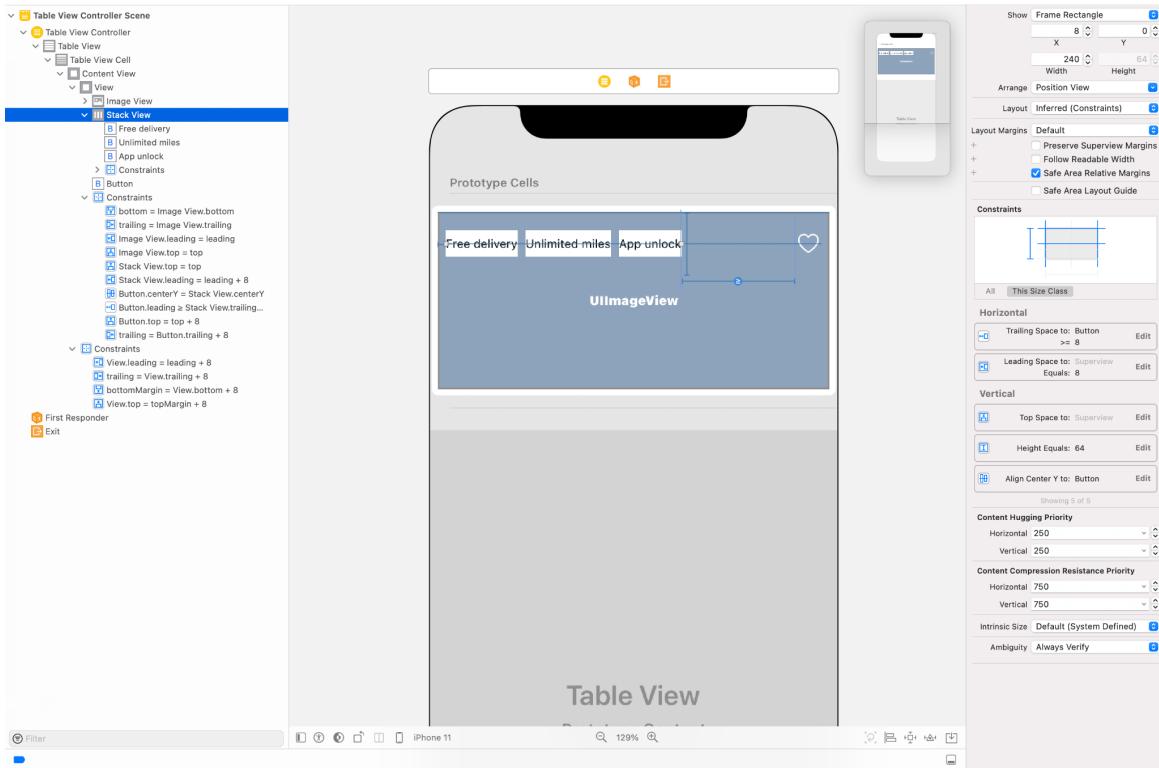
The obvious choice for this application is a `UIStackView` which will automatically adjust the constraints of all its `arrangedSubviews` when you toggle the `isHidden` property on any of the managed views. Moreover, a `UIStackView` would not only allow us to toggle the visibility of existing views without creating any additional constraints, but would also allow us to easily add and remove new views as well.



In the photo above, I've added a `UIStackView` with top, leading, and trailing constraints pinning it to its parent view and specified a fixed height. I've also added three buttons, customized the background colors, and updated the `UIStackView`'s spacing and alignment style. Next, we'll need to add the Favorites button, but this is going to trigger a series of changes.

Firstly, based off of the mockup, the `UIStackView` containing the tags needs to be able to grow as large as it needs to in order to display the text in the tags without clipping, but not so large that it encroaches on the space leading to the Favorites button. To accomplish this, we can change the trailing constraint of the `UIStackView` to be relative to the Favorites button instead of the parent view. More specifically, our updated constraint (see photo below) specifies that the trailing edge of our `UIStackView` must be at least 8px away from the leading edge of the Favorites button.

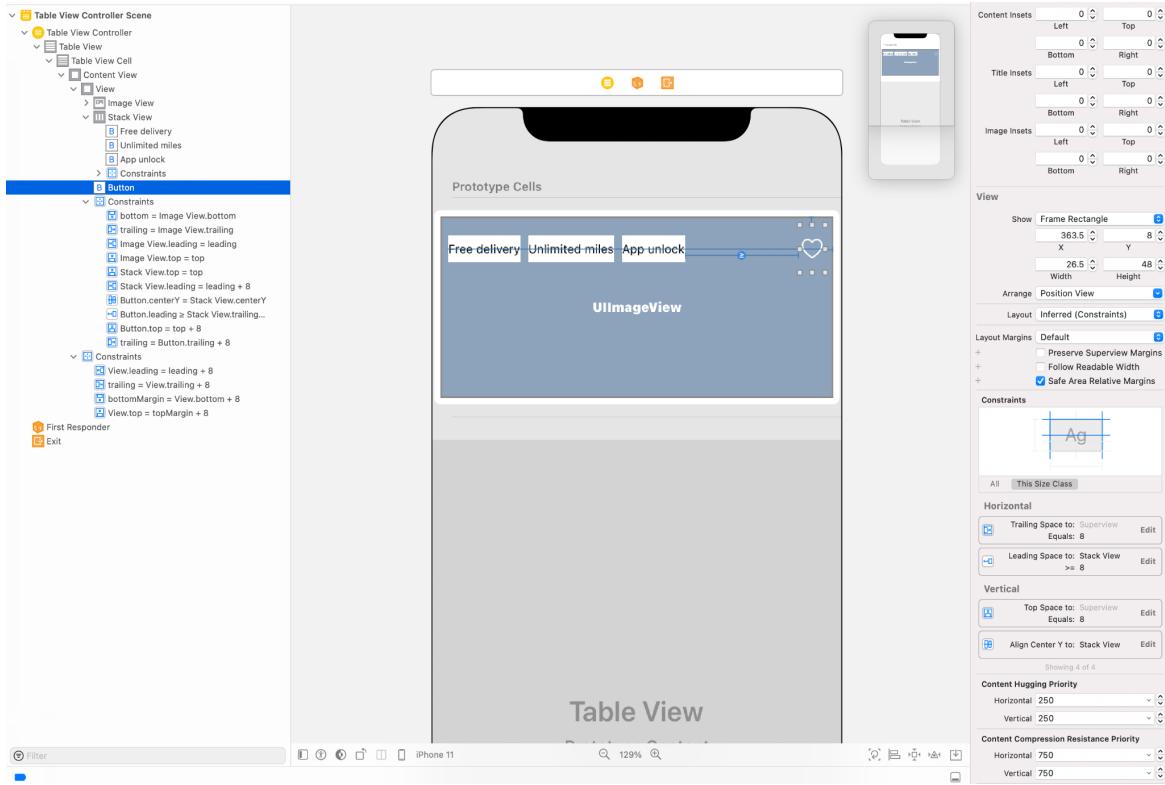
Now, our Favorites button will always be visible and unobstructed.



What if our application supported a more verbose language?

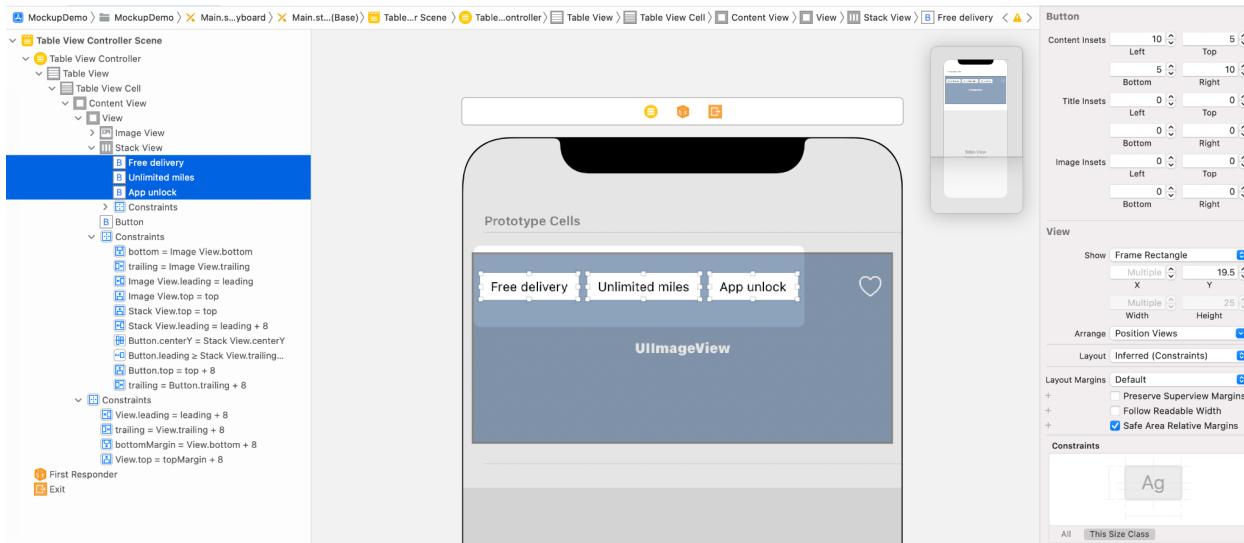
As an example, the German translation for "Unlimited miles" would be "Unbegrenzte Kilometer". If we didn't modify this constraint, our tag would likely be so large that it would block or obscure our Favorites button completely.

During the interview, you want to ensure the interviewer recognizes that you are thinking ahead and anticipating potential issues like this - all the more reason to talk out loud during the interview.



We are clearly making progress, but our next issue is that the text within the tags doesn't appear to have the correct spacing. Ideally, there would be a little additional margin between the text and the edges of the button.

While we could certainly create a custom `UIButton` to handle this, we can achieve the same result by simply changing the `contentInsets` property instead:



It never fails to surprise me how subtle and nuanced even the most seemingly straightforward views can be.

## Building The Price Display

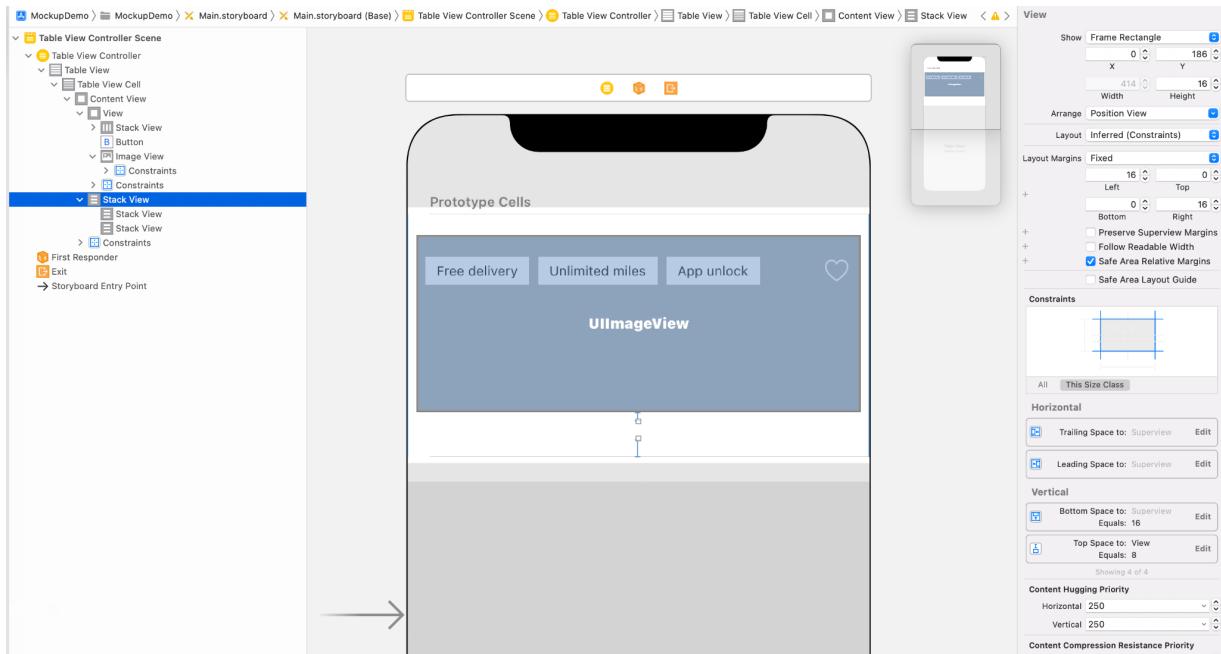
Now, we can turn our focus to the content that sits below the main `UIImageView`. There is no "right" answer here, so your implementation approach will likely differ from mine.

After looking at the mockup, I see two sections; the first section presents information about the vehicle, includes information about its rating and history, as well as some basic information about the vehicle's owner, while the second section includes information about the vehicle's price.

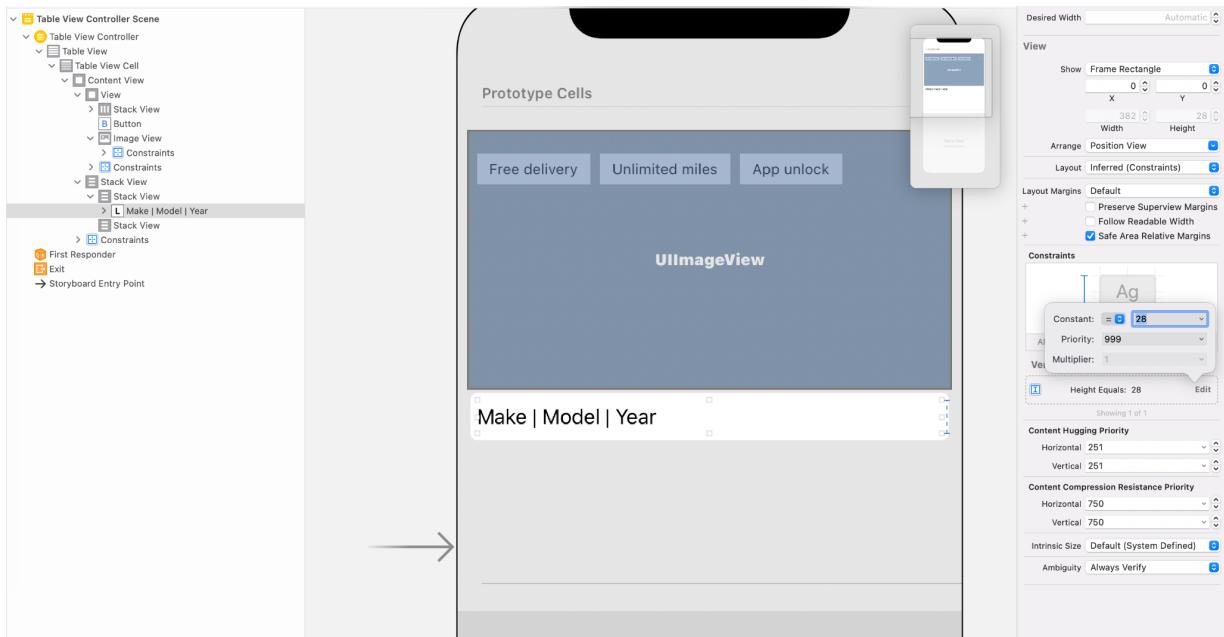
Considering that this is one of the most prominent views in the application, I would expect the content here to be influenced by possible A/B tests, future design iterations, and so forth - after all, this is the search results page. So, I would like to implement it in a way that accommodates any future changes in the easiest and most maintainable way possible. In other words, I would like whatever implementation method I choose to support easily hiding and showing any of these subviews. So, I'll add two new `UIStackViews`, one for each section.

In the absence of a `UIStackView`, we'd need to specify constraints for all permutations of views being hidden and shown. A solution like this is neither scalable nor maintainable as we would need to recreate the entire collection of constraints whenever we introduce or remove a subview.

With a `UIStackView`, we only need to specify constraints on the `UIStackView` itself, and it will automatically manage the layout of all of its visible subviews. Although `UIStackViews` have a bit of a learning curve, for this situation, they are the perfect choice.



## 1. Creating An Empty Stack View

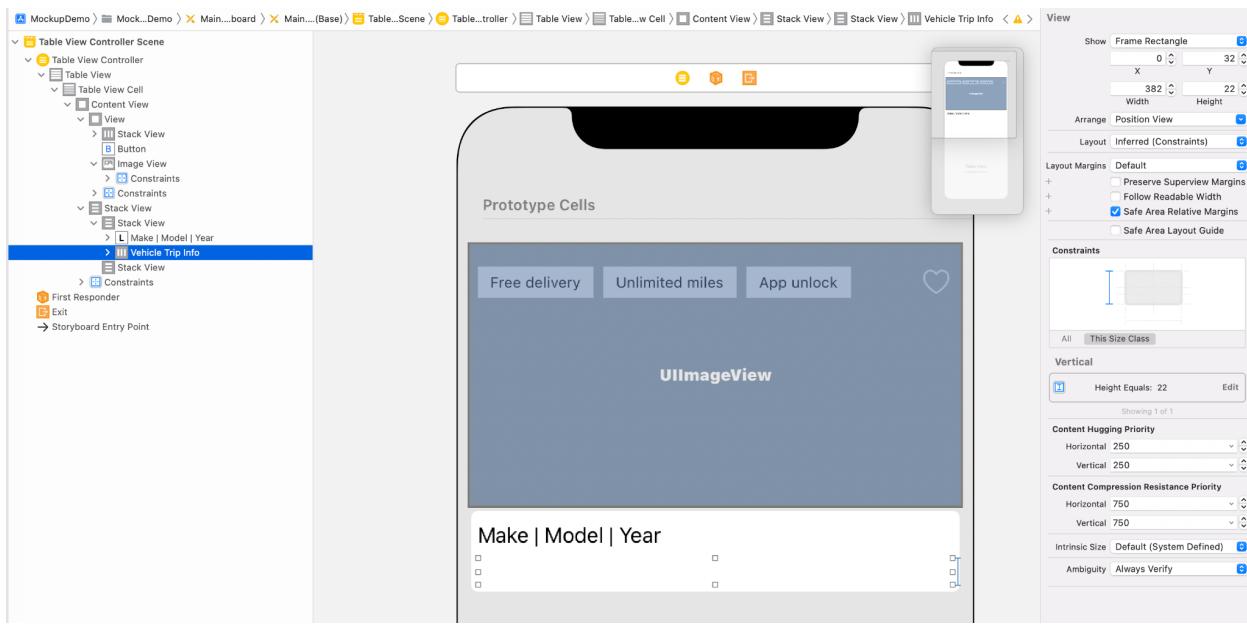


## 2. Adding Content & Tweaking Constraints

In Step 2, I added the "Make | Model | Year" label as well as specified a height of 28px.

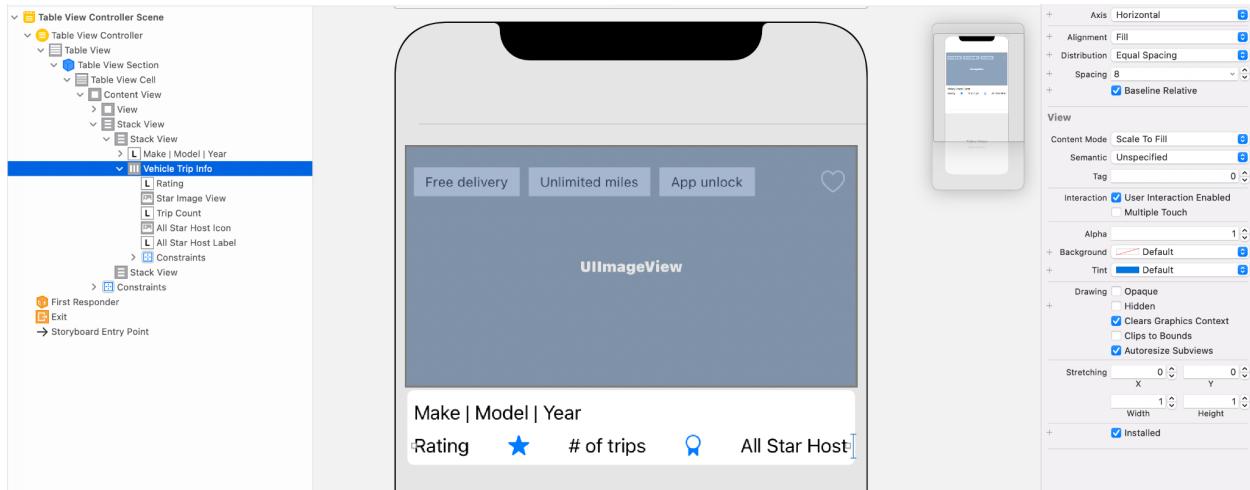
By changing the priority of the height constraint, we are able to influence the height our **UILabel** will receive from the **UIStackView**, while allowing the **UIStackView**'s constraints to take precedence if necessary.

As you may have noticed, I manually set the height of our cell, but this is only a temporary measure to make taking screenshots more convenient. As long as we specify all of our constraints correctly, we should be able to rely on our `UITableViewCell`'s `intrinsicContentSize` and `UITableView.automaticDimension` to calculate the appropriate size for our view.



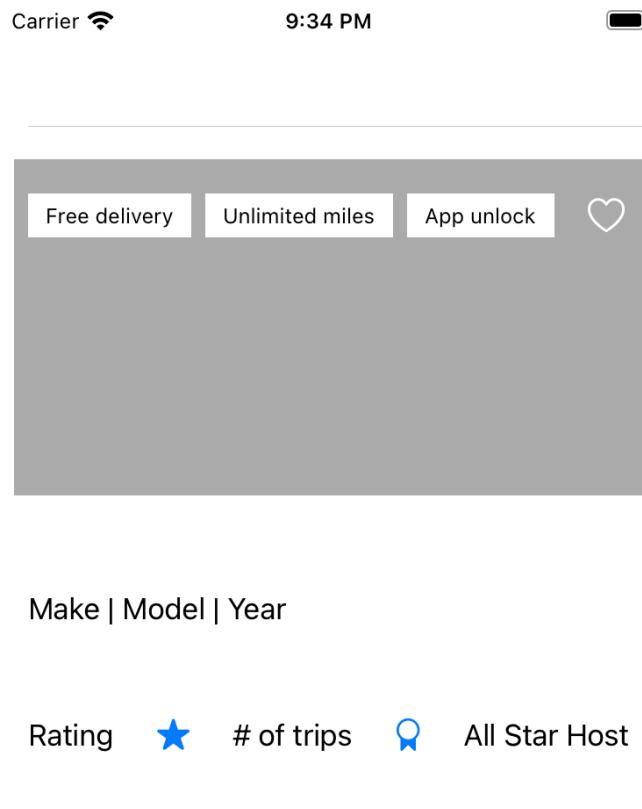
As you'll see on the following page, I've added all of the different subviews we will need to show to our “Vehicle Trip Info” `UIStackView`. Moreover, for convenience sake, I used `SFSymbols` rather than introducing custom assets into this project.

To reiterate, the goal of this walkthrough is not to build a pixel-perfect UI, but rather to honor the constraints and to recreate the mockup as closely as possible.



As a last step, I turned the `UITableView` into a static one, making testing easier by eliminating the need for a `UITableViewDataSource` or having to write any code at all.

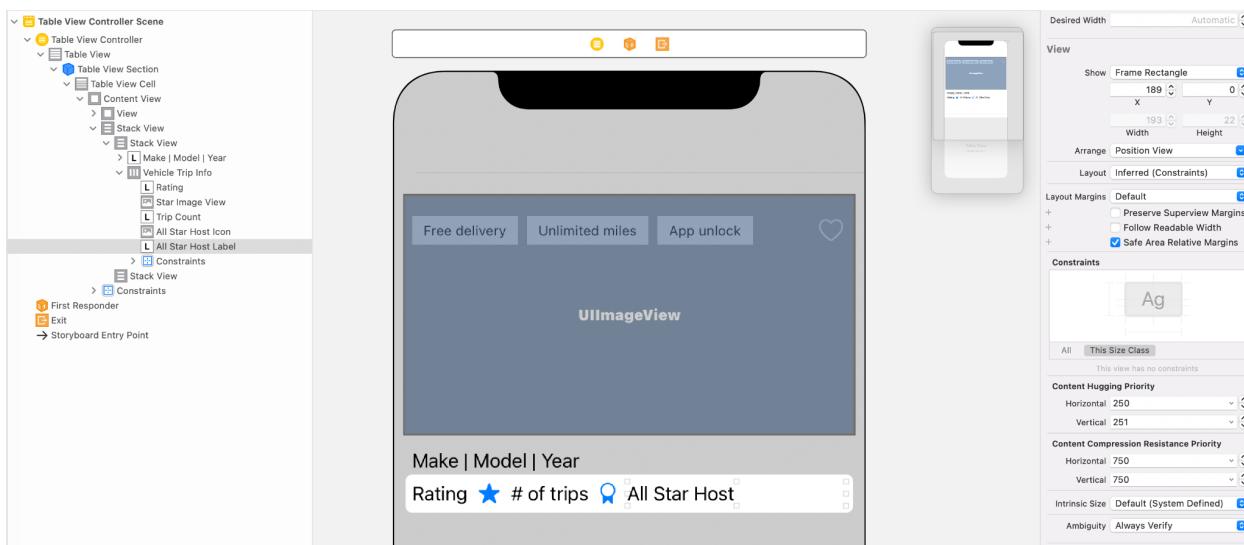
Here's how we're looking so far:



We are moving in the right direction, but as we examine the mockup, we see that the ratings and trip information are grouped more closely than in our current implementation - everything is arranged from left to right, with minimal space between adjacent elements.

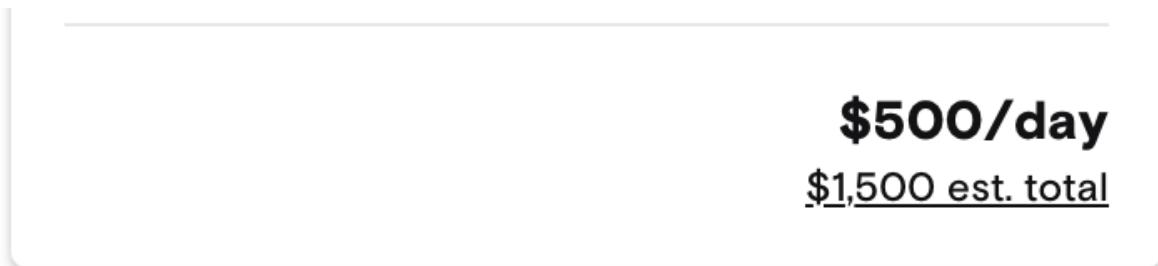
Fortunately, it's an easy fix. All we need to do is change the distribution style of our `UIStackView` to `Fill` and to decrease the horizontal Content Hugging Priority of our "All Star Host Label".

This reduction in priority informs AutoLayout that it should first attempt to change the size of the "All Star Host Label" before attempting to change the dimensions of any of the other arranged views.

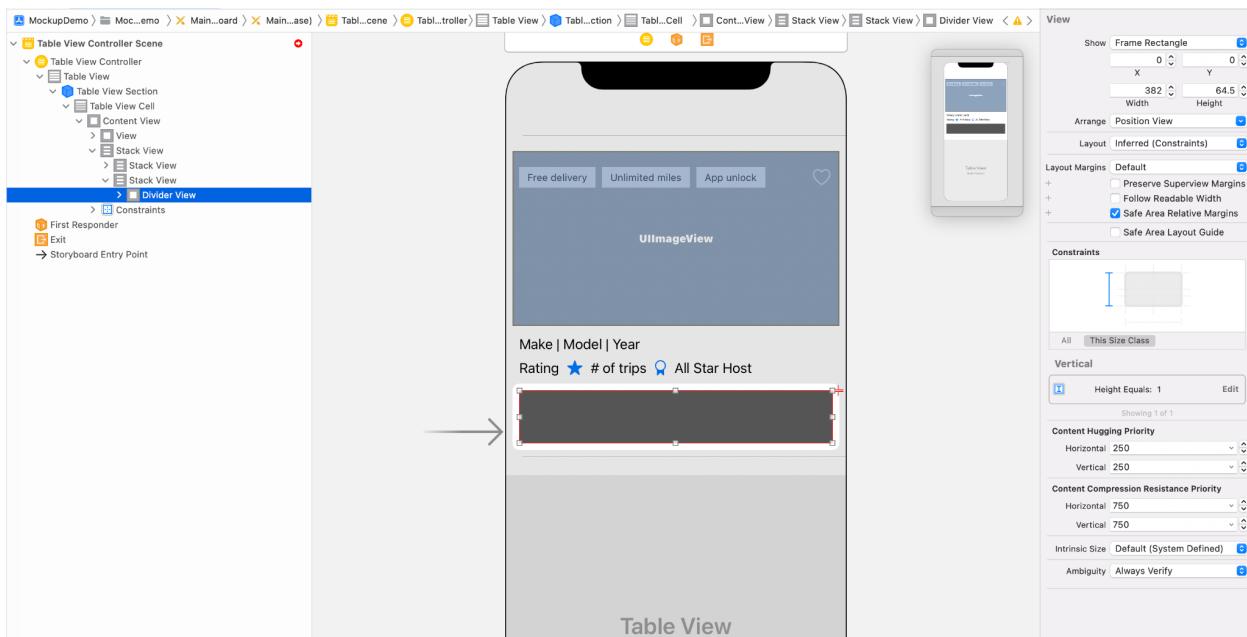


You have hopefully already noticed how even seemingly simple views can contain a lot of nuance and subtlety, and how in order to create them you might even need the use of advanced AutoLayout features.

Alright, we're onto the home stretch now:



We can see a thin dividing line before the pricing information appears. To implement this, we can simply add a `UIView` to our `UIStackView`, set its `backgroundColor`, and constrain its height to be just one pixel tall.



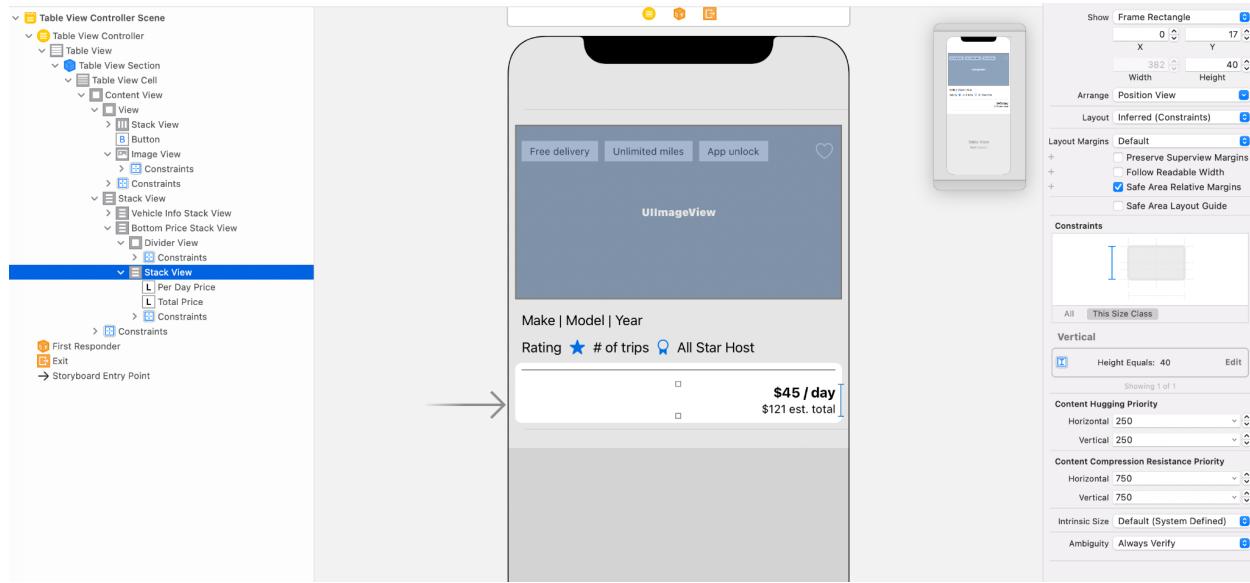
At the moment, Xcode is annoyed with us because we've specified a fixed height for both the divider view and the `UITableViewCell` and it's having difficulty honoring both requirements.

We'll fix this issue in the following steps.

After the divider view, we see two `UILabels` with pricing information stacked vertically. We will focus on matching the layout, rather than the heavily customized styling.

To house the `UILabels` containing the pricing information, I'm going to create one more vertical `UIStackView` and specify a fixed height.

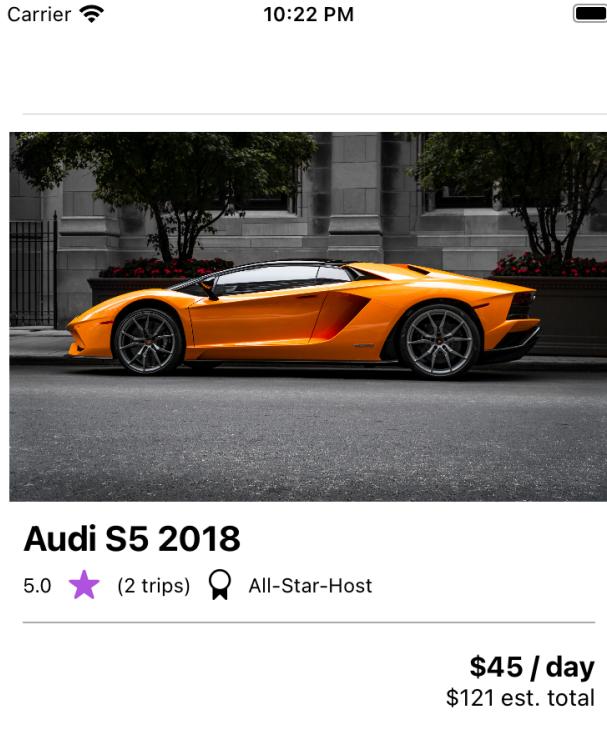
Did you notice how the addition of these additional views and constraints has resolved the layout issue from earlier?



It may seem to some that I'm overusing `UIStackViews`, but when we look at the mockup, we can clearly see that different subviews are displayed from one cell to the next. So, we know our implementation should make adding, removing, hiding, and showing subviews as easy as possible.

Given those requirements, nothing beats a `UIStackView` given it does so much of the heavy lifting for us and helps us minimize the overall number of constraints we need to specify.

We now have everything implemented! From here, after I spent a few minutes adding hardcoded text, adjusting font sizes, and making a few other basic stylistic tweaks, we've landed on our final product:



While this implementation is clearly not stylistically identical (`cornerRadius`, shadows, fonts, etc.), I'm confident you will be able to implement these topics during the interview, if necessary. Considering we didn't have exact dimensions or assets to work with, I think we did a decent job reproducing the original mockup.

Initially, I dismissed these types of problems. I figured since I used AutoLayout every day, surely I would be able to create any view I needed to. In an interview, however, with the clock ticking down, trying to debug AutoLayout issues, breaking constraints, and resolving conflicting priorities becomes nearly impossible.

I hope this walkthrough has given you some insight into how sophisticated these UI-centric interviews can be. I highly recommend trying the problem out yourself. In case you need more

practice, pick your favorite app, find an important view, and replicate it while timing yourself along the way.

As an extra precaution, I suggest that you also recreate these views programmatically, so you are prepared for whatever form the exercise takes. By practicing programmatically, you can avoid simple mistakes such as forgetting to set `isActive = true` or `translatesAutoresizingMaskIntoConstraints = false` when specifying custom constraints.

For problems like this, there is no shortage of practice problems since there are always new apps whose interface you can copy. I practiced replicating mockups repeatedly until I was able to do so without missing any steps or running into any problems - no broken, duplicate, or extra constraints, no missed steps, and remembering to speak out loud about my implementation.

As a final check, I would spend some time stress testing your implementation and ensuring that it responds appropriately to different device sizes, orientations, and potential text truncation or Dynamic Type issues.

While it may be a challenging problem, it often makes for a fun interview question, and one you can excel at with a little bit of practice.

You can find the completed implementation [here](#).

## Assessment 4: Building A Logger

You can find the final implementation [here](#).

**Interview Duration:** 60 minutes

I've somehow managed to come across this question twice in my interviewing experience. While it's certainly not my favorite question, I can understand the motivation behind it.

The task is simply to "build a logging utility for all iOS applications in our company to use".

Now, logging is something that everyone intuitively understands and it's clear the question is hinting at creating some type of framework, but there's still a lot more depth to this question than meets the eye. We could easily spend the sixty minute interview just discussing the basic requirements.

This question is less about data structures and algorithms and instead focuses on your ability to think deeply about a problem, identify requirements, define the scope, and design an intuitive and comprehensive API.

The interviewer is trying to determine if you can put yourself in the shoes of the programmer who will eventually have to use, and possibly maintain, your code. Furthermore, this question helps demonstrate whether you have worked on projects that are large enough to warrant proper logging.

You should, as always, spend some time understanding a problem's scope and the requested functionality before writing any code. Usually in these types of interview questions, if you're writing code within the first 5-10 minutes, you've jumped the gun.

### Clarifying Requirements

During my interview, the focus was less on the actual implementation specifics and more on "asking the right questions" and nailing down the high-level design of the API.

Here are a few of the clarifying questions and concerns I raised with the interviewer:

- Is multithreaded logging a concern? If so, in what order should we output the logged statements?

- What format should the logged output be in?
  - If other programmers are going to be relying on the logs, they'll need to be human-readable and would likely require custom "pretty-print" style formatting.
- How do we make sure our logging behavior is different in production vs development?
  - We need to make sure that we don't accidentally log sensitive information in production.
- Do we need to save the logs somewhere?
  - Is logging in-memory sufficient or do we need to add some type of persistence?
  - Will the logs ever leave the device?
  - How are we going to handle the space required?
  - What if we don't have any free space available?
  - Can we save the logs in plain text or should they be encrypted?
  - If we do need to persist the logs, how long do they need to last?
- Since not all loggable messages are of the same importance, should the logger provide different severity levels?
- How can we include metadata about *where* the call to our logging utility is coming from?
  - It'd be great to be able to provide contextual information about the line, function, class, etc. responsible for calling our utility.
  - Is there any other metadata we want to include?
- Since we're likely going to be calling the logger a *lot*, what performance metrics are we trying to hit? How optimized does the actual implementation need to be?

Despite its superficiality, this problem has a fair amount of depth. Obviously, you won't be able to address all of these topics in such a short interview, but the interviewer expects you to at least raise them as concerns.

From here, you and the interviewer will collaboratively shortlist the main requirements.

## Defining The Scope

Having discussed these topics, we narrowed the scope to what we could realistically accomplish in the remaining 45-50 minutes.

We agreed that:

1. An in-memory logging solution would be sufficient.
2. Different severity levels of logging should be supported.
3. Logging in production vs development should be handled differently.

4. API should be simple, clean, and easily maintained.
5. Include some metadata indicating the context in which the call to the logging utility occurred.

We discussed APIs and functionality for some time and decided that a good goal would be to create a logging utility that would work like this:

```
Log.error("This is an error message")
Log.warning("This is a warning message")
Log.info("This is an info message")
Log.info("This is an info message without context", shouldLogContext: false)

[ALERT ✗] This is an error message → ViewController.swift:226 viewDidLoad()
[WARN !] This is a warning message → ViewController.swift:227 viewDidLoad()
[INFO] This is an info message → ViewController.swift:228 viewDidLoad()
[INFO] This is an info message without context
```

We'll start from the bottom and work our way up.

## Handling Different Severity Levels

As we discussed, we'll need to be able to log messages with different severity levels. We can do this by creating an `enum` in which each case represents a distinct severity level. Additionally, we can add a computed property that will help us create level-specific custom styling.

From the example output above, you'll see that the following implementation will be used to generate the initial part of each logged message (e.g. [ALERT ✗], [WARN !], etc.).

```
enum LogLevel {
    case info
    case warning
    case error

    fileprivate var prefix: String {
        switch self {
        case .info:    return "INFO"
        case .warning: return "WARN !"
        case .error:   return "ALERT ✗"
        }
    }
}
```

And just like that we've added support for multiple severity levels and started addressing the human-readable formatting requirement.

## Making Our Logger Context Aware

Next, we know that we'll need to include some context information about where the call to the logging utility originates from, but I had absolutely no idea how to accomplish this part of the implementation.

Fortunately, since I spent the time discussing the ideal implementation with the interviewer and demonstrating that I was on the right track, they gave me a hint. The interviewer told me about the following literal expressions which would help make my task easier:

Literal	Type	Value
#file	String	The path to the file in which it appears.
#fileID	String	The name of the file and module in which it appears.
#filePath	String	The path to the file in which it appears.
#line	Int	The line number on which it appears.
#column	Int	The column number in which it begins.
#function	String	The name of the declaration in which it appears.
#dsohandle	UnsafeRawPointer	The dynamic shared object (DSO) handle in use where it appears.

A good interviewer is motivated to help you succeed, so they'll give you hints, but you first have to show that you're heading in the right direction. It's very likely that the interviewer was instructed to give this tip once the candidate began exploring this area.

With that piece of the puzzle resolved, I needed something to tie all of this information together. So, I created the following `Context` struct:

```
struct Context {
    let file: String
    let function: String
    let line: Int

    var description: String {
        "\((file as NSString).lastPathComponent):\((line)) \((function))"
    }
}
```

## Creating The Logger

Let's move on to actually logging the message.

All we're doing here is combining together the `LogLevel` styling, the actual message, and optionally the `Context` into a single `String`. Then, we check that we're running in `DEBUG` mode, and if that's the case, we'll continue on and print out the formatted message.

```
fileprivate static func handleLog(level: LogLevel, str: String,
                                  shouldLogContext: Bool, context: Context) {
    let logComponents = ["[\(level.prefix)]", str]

    var fullString = logComponents.joined(separator: " ")
    if shouldLogContext {
        fullString += " → \(context.description)"
    }

#if DEBUG
    print(fullString)
#endif
}
```

Using the `DEBUG` compiler directive ensures that we don't accidentally log sensitive information in production which would be easily accessible on a jailbroken device. Limiting logging in production also provides a small performance boost, especially when you consider how often calls will be made to our `Log` utility.

Classes that utilize our logging utility don't *need* to know how it works in order to use it. So, in the spirit of not leaking implementation details, I chose to make the `handleLog()` `fileprivate`.

Next, we'll add some helper functions to make our utility more user-friendly:

```

static func info(_ str: StaticString, shouldLogContext: Bool = true,
                file: String = #file, function: String = #function,
                line: Int = #line) {
    let context = Context(file: file, function: function, line: line)
    Log.handleLog(level: .info, str: str.description,
                  shouldLogContext: shouldLogContext, context: context)
}

static func warning(_ str: StaticString, shouldLogContext: Bool = true,
                    file: String = #file, function: String = #function,
                    line: Int = #line) {
    let context = Context(file: file, function: function, line: line)
    Log.handleLog(level: .warning, str: str.description,
                  shouldLogContext: shouldLogContext, context: context)
}

static func error(_ str: StaticString, shouldLogContext: Bool = true,
                  file: String = #file, function: String = #function,
                  line: Int = #line) {
    let context = Context(file: file, function: function, line: line)
    Log.handleLog(level: .error, str: str.description,
                  shouldLogContext: shouldLogContext, context: context)
}

```

## Final Implementation

Now, putting everything together, our final `Log` utility now looks like this:

```

enum Log {
    enum LogLevel {
        case info
        case warning
        case error

        fileprivate var prefix: String {
            switch self {
                case .info:    return "INFO"
                case .warning: return "WARN ⚠"
                case .error:   return "ALERT ✖"
            }
        }
    }
}

```

```

        }
    }
}

struct Context {
    let file: String
    let function: String
    let line: Int
    var description: String {
        "\((file as NSString).lastPathComponent):\((line) \((function))"
    }
}

static func info(_ str: StaticString, shouldLogContext: Bool = true,
                 file: String = #file, function: String = #function,
                 line: Int = #line) {
    let context = Context(file: file, function: function, line: line)
    Log.handleLog(level: .info, str: str.description,
                  shouldLogContext: shouldLogContext, context: context)
}

static func warning(_ str: StaticString, shouldLogContext: Bool = true,
                     file: String = #file, function: String = #function,
                     line: Int = #line) {
    let context = Context(file: file, function: function, line: line)
    Log.handleLog(level: .warning, str: str.description,
                  shouldLogContext: shouldLogContext,
                  context: context)
}

static func error(_ str: StaticString, shouldLogContext: Bool = true,
                  file: String = #file, function: String = #function,
                  line: Int = #line) {
    let context = Context(file: file, function: function, line: line)
    Log.handleLog(level: .error, str: str.description,
                  shouldLogContext: shouldLogContext, context: context)
}

fileprivate static func handleLog(level: LogLevel, str: String,
                                  shouldLogContext: Bool,
                                  context: Context) {
    let logComponents = ["[\(level.prefix)]", str]

```

```
var fullString = logComponents.joined(separator: " ")
if shouldLogContext {
    fullString += " ➔ \(context.description)"
}

#if DEBUG
print(fullString)
#endif
}
}
```

You can find the final implementation [here](#).

There are a few advantages to this approach:

Firstly, this lightweight implementation allows us to easily log at different severity levels without much overhead or maintenance. You could easily amend this implementation to include more granular severity levels if needed. As you can see, this approach provides much more control and scalability than independent calls to `print()` would allow.

Next, we can leverage Swift's literal expressions feature so we can easily add additional context (e.g. filename, function, line number, etc.) to our logged output. Also, we can use compiler directives to ensure we are only logging information, sensitive or otherwise, in `DEBUG` mode. Leveraging these language features greatly improves the usefulness of our utility.

Lastly, we could easily extend our implementation to include all of the other requirements we initially discussed. For example, we could introduce a configuration option and additional services by way of dependency injection that would allow us to save logs to local or cloud storage.

## Assessment 5: Creating An Analytics Service

You can find the final implementation [here](#).

**Interview Duration:** 60 minutes

This interview problem tests your ability to create a loosely coupled software architecture, clarify requirements, and build software tools with other developers in mind.

In this interview, I was asked to create an “analytics service” that all projects within the company could use to report analytics events. The question is intentionally vague in order to encourage the candidate to ask clarifying questions.

This question is also used to determine whether the candidate has experience working with analytics which is generally indicative of working in a larger company with a well-defined engineering process.

This problem also gives you a chance to demonstrate your problem-solving approach and software architecture abilities:

- Do you design your code to accommodate potential changes?
- How extensible and loosely coupled is the code you've written?
- What kind of design patterns are you comfortable with?
- How do you handle vague requirements?
- How do you approach a new problem?
- Do you consider alternative solutions before committing to an approach?

### Basic Implementation

There are several popular analytics providers like Firebase, MixPanel, and Google Analytics. So, the design of any general purpose analytics utility should account for this variability. In an ideal world, our final implementation would be compatible with any of these providers and would allow us to easily change between them.

Since it can be difficult to predict when new product requirements will trigger a change in tooling, dependencies, or third-party providers, we should strive to make our implementation as adaptable as possible.

We'll use the Firebase SDK to start.

A naive implementation would involve calling the Firebase SDK whenever we needed to log an event:

```
import FirebaseAnalytics

Analytics.logEvent("analytic_event_name", parameters: [:])
```

Note: `Analytics.logEvent()` is a part of `FirebaseAnalytics`.

With this approach, our codebase would now contain hundreds of individual references to the Firebase SDK and calls to `Analytics.logEvent()`. As a result, attempting to remove or replace the Firebase SDK would now be extremely difficult.

## Final Implementation

After spending some time creating an informal class diagram and walking the interviewer through an alternative approach, we arrived at this solution.

We both agreed that the ideal implementation would not expose the provider to the rest of the codebase. Since all analytic events have the same structure, an event name and optional metadata, we could formalize this into a `protocol`:

This should be general enough to accommodate most analytics providers.

```
protocol AnalyticsProvider {
    func sendAnalyticsEvent(named name: String, metadata: [String : Any]?)
```

Now, let's complete the implementation for an entity that conforms to `AnalyticsProvider`.

```
import FirebaseAnalytics

struct FirebaseAnalyticsProvider: AnalyticsProvider {
    func sendAnalyticsEvent(named name: String, metadata: [String : Any]?) {
        Analytics.logEvent(name, parameters: metadata)
    }
}
```

With our new implementation, this will be the only time we'll need to import the Firebase SDK! We could have just as easily created a `GoogleAnalyticsProvider` or any other variation instead.

We will now create an `AnalyticsManager` utility which will use our `AnalyticsProvider` to log events. This is just one approach to decoupling the dependency from the rest of the codebase - feel free to choose whatever approach or design patterns you prefer.

The main goal is to demonstrate to the interviewer that we understand the repercussions of littering `Analytics.logEvent()` calls throughout the codebase and that we are working to create a solution that is more generic and adaptable.

```
class AnalyticsManager {  
    private var provider: AnalyticsProvider?  
    static var shared = AnalyticsManager()  
  
    private init() {}  
  
    func configure(provider: AnalyticsProvider) {  
        self.provider = provider  
    }  
  
    func track(eventName: String, metadata: [String: Any]?) {  
        guard let provider = provider else {  
            print("Analytics provider not provided.")  
            return  
        }  
  
        provider.sendAnalyticsEvent(named: eventName, metadata: metadata)  
    }  
}  
  
// Somewhere else in the code - AppDelegate, perhaps.  
AnalyticsManager.shared.configure(provider: FirebaseAnalyticsProvider())
```

Now, this utility can be freely used throughout our codebase:

```
AnalyticsManager.shared.track(eventName: "user_clicked_forgot_password",  
                           metadata: ["userID": "aryamansharda"])
```

Ideally, the reference to `AnalyticsManager` would be passed into our `UIViewController`s via dependency injection.

Then, if we ever need to replace the analytics service, we can simply create a new class that implements the `AnalyticsProvider` protocol and pass that into the `AnalyticsManager` instead:

```
AnalyticsManager.shared.configure(provider: MixpanelAnalyticsProvider())
```

With this approach, the rest of our code is no longer aware of which analytics service we're using and replacing it becomes a straightforward task.

This setup allows us to easily add custom behavior based on the environment we're running in. For example, we may not want to record events when we're running in `DEBUG` mode as that may taint our production data; we may just want to log it to the console instead:

```
class LocalAnalyticsProvider: AnalyticsProvider {
    func sendAnalyticsEvent(named name: String, metadata: [String : Any]?) {
        print("\(name): \(metadata)")
    }
}

// AppDelegate
#if DEBUG
AnalyticsManager.shared.configure(provider: LocalAnalyticsProvider())
#else
AnalyticsManager.shared.configure(provider: FirebaseAnalyticsProvider())
#endif
```

Can you imagine how messy and difficult implementing this functionality would have been if we were making individual calls to `Analytics.logEvent()` throughout our code?

You can find the final implementation [here](#).

## Assessment 6: Efficiently Displaying The Fibonacci Sequence

You can find the final implementation [here](#).

**Interview Duration:** 60 minutes

Alright, this was another fun one.

In this interview, I was asked to create a “`UITableView` that efficiently shows the numbers in the Fibonacci sequence”.

The Fibonacci sequence is created by taking the sum of the two preceding values in the sequence, starting with 0 and 1:

Ex: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34... and so on.

### Picking An Approach

By the interviewer's use of the word "efficiently," I knew that I would have to use memoization in some capacity. It was clear that the naive method of generating Fibonacci numbers recursively was not going to scale.

Since a `UITableView` can only display a finite number of `UITableViewCellCells` at any given time, I knew I could further optimize my solution by only calculating the values necessary to populate the visible part of the `UITableView`.

Moreover, for improved performance, I could run all computation on a separate thread so the Main Thread and subsequently the UI remain responsive.

### Handling Integer Overflow

Having dealt with similar problems before, I knew that the Fibonacci sequence would quickly cause Integer Overflow issues and that we'd need to account for this edge case in our implementation.

Additionally, I knew that this would be a little trickier in Swift since it does not have native support for handling exceptionally large numbers like Java's `BigInt` class does.

As a quick aside, `BigInt` essentially represents a large number as a `String`. Then, whenever arithmetic operations need to be performed, it will convert each character back into a digit, perform the operation, and write back an updated `String`.

When I raised this concern with the interviewer, they instructed me to show as many numbers as I could accurately present. In other words, I was meant to generate the Fibonacci sequence up until the point where Integer Overflow became an issue.

Since runtime errors, like Integer Overflow, are not exceptions, keywords like `try`, `do`, and `catch` wouldn't help since there's no "exception" to handle. So, I needed to find another approach.

As I was poking around in Swift's `Integer` documentation, I came across a function that would notify me if an attempted addition operation results in an overflow:

```
func addingReportingOverflow(_ other: UInt64) ->
    (partialValue: UInt64, overflow: Bool)
```

With this in hand, it would be easy to detect the first time we exceed `UInt64`'s maximum capacity.

We'll throw an error in our implementation to handle this case.

## Making Optimizations

We only need to compute additional values in the sequence when the user is approaching the last cell in the `UITableView` [the last position of the sequence we've calculated]. If we can keep track of the visible cells in the `UITableView`, we can delay computation until it's absolutely necessary.

My initial approach was to override the `UIViewController`'s `scrollViewDidScroll` function, however I soon realized that this approach wouldn't provide enough context about where in the `UITableView` the user was when scrolling. For example, if the user was scrolling backwards up the list, there's no need to calculate the next batch of numbers in the sequence as we'd only be looking at previously computed and stored values. Knowing when the user was scrolling was not enough; I needed to know if they had moved forwards or backwards in the `UITableView` and how close they were to the end.

Eventually, I landed on `UITableView`'s `willDisplayCell:(UITableViewCell *)cell forRowAtIndexPath:(NSIndexPath *)indexPath` method. This method informs us when the `UITableView` is about to display a `UITableViewCell` for a particular row.

This would allow us to kick off the computation for the next set of values in the sequence just in the nick of time.

## Final Implementation

```
final class FibonacciCalculator {
    /// The key is the position in the sequence and the value is
    /// the actual element in the sequence
    private var sequenceCache = [Int: UInt64]()

    /// Once we've found the largest value possible, we want to stop
    /// future calculation attempts
    var maxFibonacciPositionReached = false

    private enum FibonacciError: LocalizedError {
        case overflow

        var errorDescription: String? {
            switch self {
            case .overflow:
                return "Maximum Swift UInt64 value reached."
            }
        }
    }

    /// Returns the Fibonacci number at a provided position
    /// - Parameter n: The position in the Fibonacci sequence
    /// - Returns: The Fibonacci number at that position
    func nthFibonacciNumber(_ n: Int) throws -> UInt64? {
        // We will only calculate future values if we haven't reached the
        // maximum value yet
        guard !maxFibonacciPositionReached else {
            return nil
        }

        // Handles the base cases of n = 0 and n = 1
        guard n > 1 else {
            return UInt64(n)
        }

        let previousPosition = n - 1
        let previousFibonacciNumber = sequenceCache[previousPosition] ?? 0
        let currentFibonacciNumber = UInt64(previousFibonacciNumber + sequenceCache[n - 2] ?? 0)

        sequenceCache[n] = currentFibonacciNumber
        return currentFibonacciNumber
    }
}
```

```

    }

    // Looks up the previous number in the sequence from the
    // cache or calculates it (if needed)
    let nMinusOneFibonacci = try sequenceCache[n - 1] as? UInt64 ?? nthFibonacciNumber(n - 1)
    sequenceCache[n - 1] = nMinusOneFibonacci

    let nMinusTwoFibonacci = try sequenceCache[n - 2] as? UInt64 ?? nthFibonacciNumber(n - 2)
    sequenceCache[n - 2] = nMinusTwoFibonacci

    guard let nMinusOneFibonacci = nMinusOneFibonacci,
          let nMinusTwoFibonacci = nMinusTwoFibonacci else {
        return nil
    }

    // Tries generating the next number and checks if the addition
    // triggers an overflow
    let (sum, didOverflow) =
        nMinusOneFibonacci.addingReportingOverflow(nMinusTwoFibonacci)

    // If we overflow, we can't compute any future numbers
    // in the sequence. So, we'll throw an error and update our flag
    if didOverflow {
        maxFibonacciPositionReached = true
        throw FibonacciError.overflow
    }

    // Returns the value at the n-th position in the Fibonacci sequence
    return sum
}

final class FibonacciTableViewController: UITableViewController {

    // MARK: - Properties
    private let fibonacciCalculator = FibonacciCalculator()
    private let serialQueue = DispatchQueue(label: "serial")
    private let pageAmount = 5

    private var dataSource = [UInt64]()
}

```

```

override func viewDidLoad() {
    super.viewDidLoad()

    // Compute initial "pageAmount" worth of values in the sequence
    for position in 0..<pageAmount {
        generateFibonacciNumber(at: position)
    }

    tableView.reloadData()
}

// MARK: - Private Methods
private func generateFibonacciNumber(at position: Int) {
    do {
        if let nextFibonacciNumber = try
            fibonacciCalculator.nthFibonacciNumber(position) {
            dataSource.append(nextFibonacciNumber)
        }
    } catch {
        DispatchQueue.main.async { [weak self] in
            self?.presentDefaultErrorAlert(
                initWithTitle: "Error",
                message:error.localizedDescription
            )
        }
    }
}
}

// MARK: - UITableView Data Source
extension FibonacciTableViewController {
    override func tableView(_ tableView: UITableView,
                           numberOfRowsInSection section: Int) -> Int {
        dataSource.count
    }

    override func tableView(_ tableView: UITableView,
                           cellForRowAt indexPath: IndexPath) ->
        UITableViewCell {
        let cell = tableView.dequeueReusableCell(
           (withIdentifier: "FibonacciCell", for: indexPath
        ) as UITableViewCell
    }
}

```

```

        cell.textLabel?.text = "\(dataSource[indexPath.row])"
    return cell
}

override func tableView(_ tableView: UITableView,
                      willDisplay cell: UITableViewCell,
                      forRowAt indexPath: IndexPath) {
    // If we are within one pageAmount of entries (5) from the
    // end of the UITableView and we haven't hit the maximum value
    // yet, fetch the next pageAmount worth of numbers in the
    // sequence.
    guard indexPath.row + pageAmount >= dataSource.count &&
        !fibonacciCalculator.maxFibonacciPositionReached else {
        return
    }

    serialQueue.async { [weak self] in
        guard let dataSource = self?.dataSource,
              let pageAmount = self?.pageAmount else {
            return
        }

        // Generates the next "page" worth of Fibonacci values
        for position in
            dataSource.count..

```

```
let defaultAction = UIAlertAction(title: "Ok",
                                   style: .default,
                                   handler: nil)

alertController.addAction(defaultAction)

present(alertController, animated: true, completion: nil)
}

}
```

You can find the final implementation [here](#).

## Assessment 7: Visualizing Articles & Advanced Codables

You can find the final implementation [here](#).

**Interview Duration:** 90 minutes

While the first assessment we saw is fair game for all experience levels, I've only encountered this variation during interviews for Senior iOS roles.

In this problem, the interviewer provides you with a sample JSON response which represents different sections of a news article. Your goal is to use native iOS components to render the article from the JSON response.

Here's an example of a simple article from CNN:

### 7 popular Western desserts Japanese chefs have made their own

Maggie Hiufu Wong, CNN • Updated 5th December 2021



(CNN) — Japan has long been famous internationally for its diverse and delicious eats, from sushi to ramen, with such staples appearing on menus around the world.

But in recent years word has also spread about its expertise in another culinary arena -- cakes and pastries. The country's chefs have taken many of the traditionally "Western" desserts known and loved around the world and elevated them to new heights.

Unlike Japanese sweets -- called wagashi -- Western-style confectioneries, referred to as "yogashi," are made mostly of flour and sugar. But Japanese versions are generally less sweet compared to their Western counterparts.



Part 1

Many of the classic yogashi that are popular in the West made their way to Japan centuries ago and have since been adapted, perfected and popularized. Some of the bigger dessert brands have already opened chains in other cities throughout Asia, from Bangkok to Taipei.

"In Japanese sweets making, there is a tendency to incorporate improvements ... Japan is good at using local ingredients and expressing the tastes of the season while incorporating Western techniques and combinations," says Kengo Akabame, pastry chef at [Imperial Hotel Tokyo](#).

Akabame was part of the Japanese team that competed at this year's [Coupe du Monde de la Patisserie](#), or the Pastry World Cup, and walked away with the silver prize.

"I think that the point of trying to create new things while incorporating (classic methods) leads to further evolution," says Akabame.

Hideo Kawamoto is president of [Juchheim Group](#), one of the oldest confectionery brands in Japan. He agrees that the freedom to experiment has helped the country's chefs build one successful dessert product after another.

"Japanese customers like to taste as much as they can, and know their favorites. Through these competitive markets, some chefs get to a famous position and create popular products," Kawamoto tells CNN Travel.

Given Japan's status as a leading travel destination prior to the pandemic, a successful new take on a cake would often quickly become trendy in other Asian countries.

Here are some of the popular cakes and desserts Japanese chefs have made their own.

### Strawberry shortcake



Japan's strawberry shortcake has become a popular winter cake around Asia. Here's the version at Good Good, a Hong Kong bakery.

Good Good

---

Among the most iconic yogashi in Japan is the classic strawberry shortcake, and many credit [Rin'emon Fujii](#), founder of Fujiya -- Japan's first nationwide Western-style cake shop chain -- for its popularity.

## Part 2

Generally speaking, we can see that any article will have headings with variable font sizes, multiple paragraphs, quotes, and images with captions and attributions.

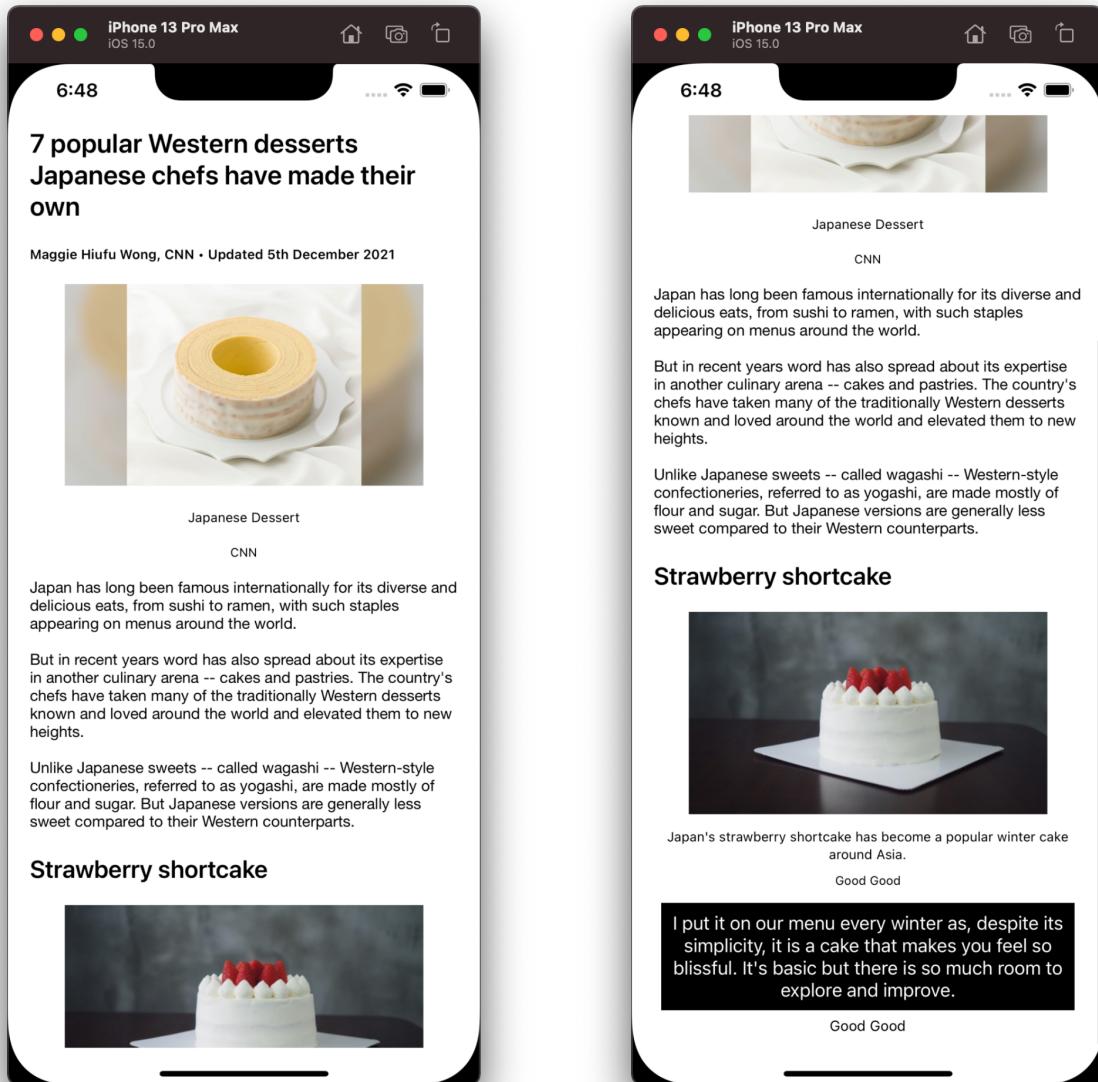
The JSON representation of this article might look something like this:

```
{
  "response": [
    {
      "heading": {
        "text": "7 popular Western desserts Japanese chefs have made their own",
        "size": 24
      }
    }
  ]
}
```

```
        },
    },
    {
        "heading": {
            "text": "Maggie Hiufu Wong, CNN • Updated 5th December 2021",
            "size": 12
        },
    },
    {
        "image": {
            "url":
"https://dynainimage.cdn.cnn.com/cnn/q_auto,w_1100,c_fill,g_auto,h_619,ar_16:9/
http%3A%2Fcdn.cnn.com%2Fcnnnext%2Fdam%2Fassets%2F211130153310-06-japan-best-cakes.jpg",
            "caption": "Japanese Dessert",
            "source": "CNN"
        }
    },
    {
        "paragraph": {
            "text": "Japan has long been famous internationally for its diverse and delicious eats, from sushi to ramen, with such staples appearing on menus around the world.\n\nBut in recent years word has also spread about its expertise in another culinary arena -- cakes and pastries. The country's chefs have taken many of the traditionally Western desserts known and loved around the world and elevated them to new heights.\n\nUnlike Japanese sweets -- called wagashi -- Western-style confectioneries, referred to as yogashi, are made mostly of flour and sugar. But Japanese versions are generally less sweet compared to their Western counterparts."
        }
    },
    {
        "heading": {
            "text": "Strawberry shortcake",
            "size": 22
        }
    },
    {
        "image": {
            "url":
"https://dynainimage.cdn.cnn.com/cnn/q_auto,w_634,c_fill,g_auto,h_357,ar_16:9/
http%3A%2Fcdn.cnn.com%2Fcnnnext%2Fdam%2Fassets%2F211130151518-05-japan-best-cakes.jpg",
            "caption": "Strawberry shortcake",
            "source": "CNN"
        }
    }
}
```

```
        "caption": "Japan's strawberry shortcake has become a popular winter  
        cake around Asia.",  
        "source": "Good Good"  
    }  
,  
{  
    "quote": {  
        "text": "I put it on our menu every winter as, despite its  
simplicity, it is a cake that makes you feel so blissful. It's basic but  
there is so much room to explore and improve.",  
        "author": "Good Good"  
    }  
}  
]  
}
```

Our goal is to recreate the article on iOS from this JSON response. In other words, we'd expect to see something like this:



Although there are clearly some stylistic differences, the general format and main content are all present.

It's important to keep in mind that since we're tasked with creating a generic solution that can represent any article, we can't assume the order of the article's sections nor how many of each section type there are.

Furthermore, since the response consists of a mix of JSON objects, we'll have to leverage some advanced **Codable** features to turn it into an array of custom models we can work with.

In summary, this problem has a few key hurdles we'll need to overcome:

1. Converting JSON into custom domain models via Swift's **Codable** protocol.
2. Decoding JSON with a heterogeneous mix of objects in the response.
3. Creating a **UITableView** that shows and manages reuse for a variety of different custom **UITableViewCell**s.
4. Creating a scalable solution that is readable, simple, and can easily be extended to add support for additional article section types like videos, advertisements, sign up form, image gallery, etc.

While this task may seem daunting, I still consider it to be a fair and reasonable assessment.

For a real-world application, consider an application like Facebook or Instagram. Your feed would have several types of custom cells coexisting with one another all powered by some variable response from the backend.

This problem tests an engineer's ability to break down a complex problem, leverage advanced Swift language features, and architect and implement extensible solutions.

## Creating The Models

Using the example JSON, we can create a simple one-to-one mapping of the fields:

```
struct Heading: Decodable {  
    let text: String  
    let size: Int  
}  
  
struct Paragraph: Decodable {  
    let text: String  
}  
  
struct Image: Decodable {
```

```
let url: String
let caption: String
let source: String
}

struct Quote: Decodable {
    let text: String
    let author: String
}
```

Since we're only concerned with converting the JSON response into an article, there is no need for `Encodable` support; conforming to the `Decodable` protocol will suffice.

## Parsing The JSON

Next, we'll need to convert our JSON response into an array of these custom models.

While there are multiple ways to solve this problem, we can still use Swift's `Codable` feature to do most of the heavy lifting by implementing our own decoding logic.

```
struct Result: Decodable {
    let response: [Response]

    enum Response: Decodable {
        case heading(Heading)
        case paragraph(Paragraph)
        case image(Image)
        case quote(Quote)

        enum DecodingError: Error {
            case wrongJSON
        }
    }

    enum CodingKeys: String, CodingKey {
        case heading
        case paragraph
        case image
        case quote
    }
}

init(from decoder: Decoder) throws {
    let container = try decoder.container(keyedBy: CodingKeys.self)
```

```
        switch container.allKeys.first {
            case .heading:
                let value = try container.decode(Heading.self,
                                                forKey: .heading)
                self = .heading(value)
            case .paragraph:
                let value = try container.decode(Paragraph.self,
                                                forKey: .paragraph)
                self = .paragraph(value)
            case .image:
                let value = try container.decode(Image.self,
                                                forKey: .image)
                self = .image(value)
            case .quote:
                let value = try container.decode(Quote.self,
                                                forKey: .quote)
                self = .quote(value)
            case .none:
                throw DecodingError.wrongJSON
        }
    }
}
```

See: [Encoding and Decoding Custom Types \[Apple\]](#)

## Building The View Controller

With all of our setup complete, we can now move on to actually converting the JSON response into a collection of our custom Swift models.

I've saved the sample JSON into a file called `Article.json` and the following code attempts to load the file from the project's bundle and decode it:

```
final class ViewController: UIViewController {
    @IBOutlet fileprivate(set) var tableView: UITableView!
    var dataSource = [Result.Response]() {
        didSet {
            tableView.reloadData()
        }
    }
}
```

```

override func viewDidLoad() {
    super.viewDidLoad()

    tableView.dataSource = self

    do {
        if let url = Bundle.main.url(forResource: "Article",
                                      withExtension: "json"),
           let data = try? Data(contentsOf: url),
           let result = try? JSONDecoder().decode(Result.self,
                                                 from: data) {
            dataSource = result.response
        }
    }
}
}
}

```

## Creating Custom Cells

Next, let's create our custom `UITableViewCell`s.

---

As a quick detour, we can use the following extension to conveniently generate reuse identifiers for our custom `UITableViewCell` subclasses:

```

extension UITableViewCell {
    static var reuseIdentifier: String {
        String(describing: self)
    }
}

```

---

Now, we can create our custom `UITableViewCell`s with their corresponding configuration functions:

```

final class HeadingCell: UITableViewCell {
    @IBOutlet private(set) var headingLabel: UILabel!

    func configure(model: Heading) {
        headingLabel.text = model.text
        headingLabel.font = UIFont.boldSystemFont(ofSize:
                                                CGFloat(model.size))
    }
}

```

```

}

final class ImageCell: UITableViewCell {
    @IBOutlet private(set) var thumbnailImageView: UIImageView!
    @IBOutlet private(set) var captionLabel: UILabel!
    @IBOutlet private(set) var sourceLabel: UILabel!

    func configure(model: Image) {
        thumbnailImageView.loadImageFromURL(urlString: model.url,
                                              placeholder: nil)
        captionLabel.text = model.caption
        sourceLabel.text = model.source
    }
}

```

Note: `ImageCell` is leveraging the `UIImageView` extension we created in Assessment 1 that allows for remotely loading and caching images.

```

final class QuoteCell: UITableViewCell {
    @IBOutlet private(set) var quoteTextLabel: UILabel!
    @IBOutlet private(set) var quoteAuthorLabel: UILabel!

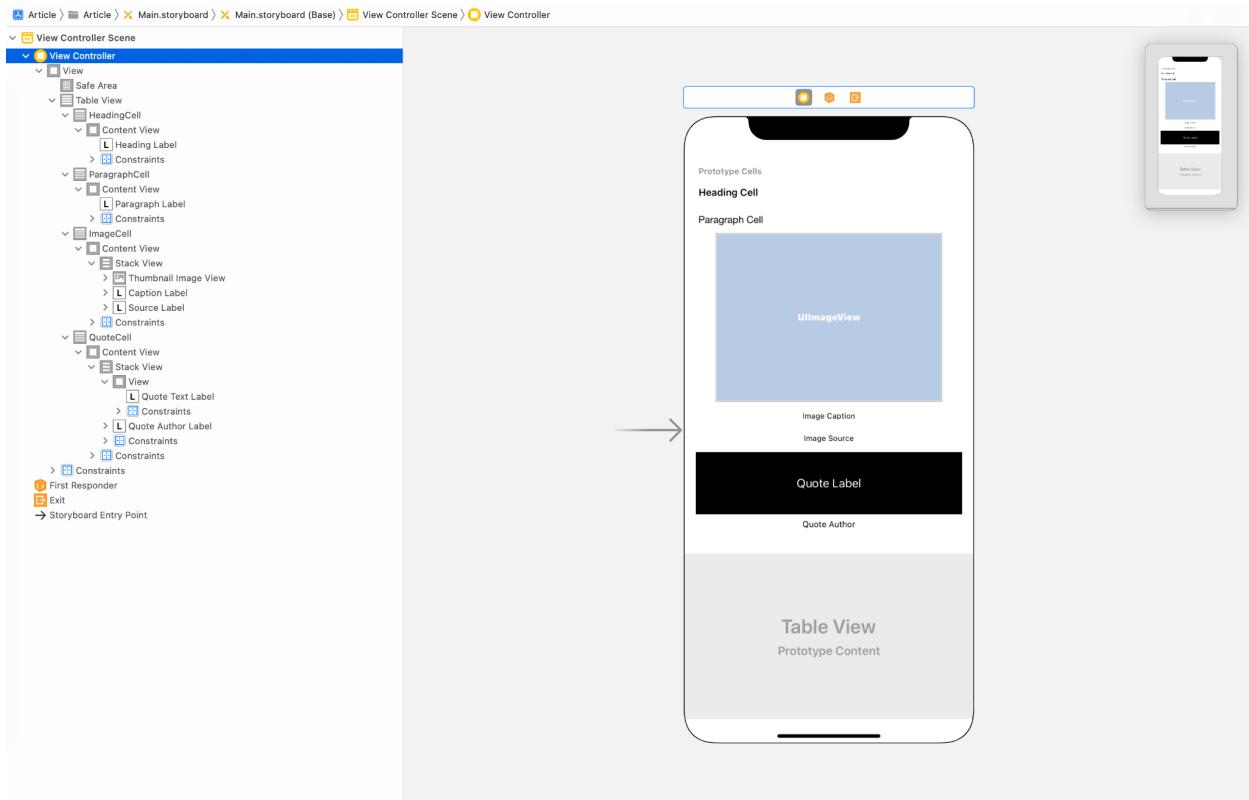
    func configure(model: Quote) {
        quoteTextLabel.text = model.text
        quoteAuthorLabel.text = model.author
    }
}

final class ParagraphCell: UITableViewCell {
    @IBOutlet private(set) var paragraphLabel: UILabel!

    func configure(model: Paragraph) {
        paragraphLabel.text = model.text
    }
}

```

Lastly, we can create Prototype Cells on the `.storyboard` corresponding to each of our custom cell types.



As you can see, I'm clearly not a designer...

Fortunately, the focus of this question is on the implementation approach rather than creating a pixel-perfect UI.

## Final Implementation

Thanks to the custom decoding logic we introduced earlier, we can easily `switch` on the type of the item in the `dataSource[]` and configure each cell as required.

The associated value of the `enum` case will provide us with the specific model we need to configure the cell.

```
extension ViewController: UITableViewDataSource {
    func tableView(_ tableView: UITableView,
                  numberOfRowsInSection section: Int) -> Int {
        dataSource.count
    }

    func tableView(_ tableView: UITableView,
```

```

        cellForRowAt indexPath: IndexPath) -> UITableViewCell {
    switch dataSource[indexPath.row] {
    case .heading(let heading):
        guard let cell = tableView.dequeueReusableCell(withIdentifier:
            HeadingCell reuseIdentifier, for: indexPath) as?
            HeadingCell else {
            return UITableViewCell()
        }

        cell.configure(model: heading)
        return cell

    case .paragraph(let paragraph):
        guard let cell = tableView.dequeueReusableCell(withIdentifier:
            ParagraphCell reuseIdentifier, for: indexPath) as?
            ParagraphCell else {
            return UITableViewCell()
        }

        cell.configure(model: paragraph)
        return cell

    case .image(let image):
        guard let cell = tableView.dequeueReusableCell(withIdentifier:
            ImageCell reuseIdentifier, for: indexPath)
            as? ImageCell else {
            return UITableViewCell()
        }

        cell.configure(model: image)
        return cell

    case .quote(let quote):
        guard let cell = tableView.dequeueReusableCell(withIdentifier:
            QuoteCell reuseIdentifier, for: indexPath)
            as? QuoteCell else {
            return UITableViewCell()
        }

        cell.configure(model: quote)
        return cell
    }
}

```

{

Now, whenever we need to support another type of article component, we can simply add a new case to `Response`, create another `UITableViewCell`, and add the appropriate configuration logic to the `switch` statement above.

With all of this in place, we're now able to convert the JSON representation of any article into its iOS counterpart.

7:08      

## 7 popular Western desserts Japanese chefs have made their own

Maggie Hiufu Wong, CNN • Updated 5th December 2021



Japanese Dessert

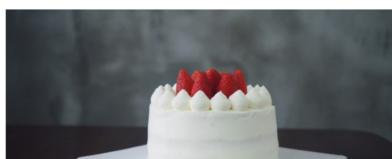
CNN

Japan has long been famous internationally for its diverse and delicious eats, from sushi to ramen, with such staples appearing on menus around the world.

But in recent years word has also spread about its expertise in another culinary arena -- cakes and pastries. The country's chefs have taken many of the traditionally Western desserts known and loved around the world and elevated them to new heights.

Unlike Japanese sweets -- called wagashi -- Western-style confectioneries, referred to as yogashi, are made mostly of flour and sugar. But Japanese versions are generally less sweet compared to their Western counterparts.

### Strawberry shortcake



You can find the final implementation [here](#).

# How To Approach The Data Structures & Algorithms Interview

Can we all agree that whiteboard interviews suck?

As prospective candidates, we're often asked questions that have no relevance to the skills we'll need on a day-to-day basis.

Invert a binary tree?

No thanks.

Unfortunately, these types of questions are inevitable. So, we should strive to find the most efficient way to prepare for them.

Certainly, spending our evenings blindly answering LeetCode questions isn't the best way to make progress. But that's what I - and I assume many of you - are advised to do.

Instead, I'd like to share my approach to preparing for the whiteboard interview. I'm not claiming this is good advice or that you should do the same - I'm just sharing my two cents.

My approach is simply to play the odds.

Glassdoor, LeetCode, and HackerRank are all excellent resources to identify commonly asked interview questions. LeetCode, for example, maintains lists of their Top Interview Questions (most frequently asked) and their Top 100 Liked Questions (community favorites). Essentially, these lists contain the questions that are the most likely to appear in an interview based on crowd-sourced reports from other engineers.

My previous interviews have almost always consisted of questions taken verbatim from these sites or minor variations of them.

So, I thought the best strategy would be to expose myself to as many of these questions as possible and hedge my bets that at least one of them would be asked in a future interview.

**Rather than spending time actually solving all of these questions, I suggest going directly to the "accepted answers" and trying to understand the approach, the runtime, edge cases, and any alternative solutions. This approach will help you efficiently build up your problem solving intuition.**

This strategy lets you learn how to solve dozens of questions quickly rather than spending hours and hours struggling to solve just a few. This strategy also allows me plenty of time to explore these problems in-depth, examine alternative implementations, and to *really* understand why the accepted solutions are optimal.

All that being said, it's still crucial to be able to solve these kinds of problems without the help of an IDE. You should try out a few problems as a litmus test of your ability to write syntactically correct code without relying on a compiler or auto-complete.

In my experience, once I was able to solve fifteen problems "freehand", I felt that my ability did not improve significantly by doing an additional fifteen. I hope you'll agree that, regardless of the specifics, the traditional approach of blindly solving programming problems has diminishing returns.

As soon as I was convinced that I could write syntactically correct Swift code unaided, I returned my focus to growing my mental question and answer bank.

To make it easier, I maintained a shared Markdown document where I would specify the questions, example inputs and outputs, a few accepted solutions, the runtime complexity, and any other notes I wanted to take. Then, I worked with friends who were also interviewing at the time to refine the document. After a few days, it contained the answers to all of the 100 Most Liked Questions on LeetCode. I would re-read this document often and try to really internalize the solutions.

As a result, when I inevitably encountered these questions in an interview, I would have some intuitive idea of how to approach the problem. At the very least, I would be able to draw on a wide range of strategies and algorithms I had seen before to help me tackle the problem.

I found that this strategy really worked for me. If not for this approach, I would have only had exposure to a small number of practice problems and I suspect the interviews would have turned out much differently.

I imagine some of you may consider this a shortcut or cheating, but I wouldn't go that far. I'm not suggesting you try and memorize the code - that would certainly be a fool's errand.

Instead, I'm suggesting a different way to quickly amass a knowledge bank of questions and answers you can rely on to solve future problems. This is the same mechanism that engineers use to advance their skills anyways; they get exposed to more problems and learn how to solve them.

**TL;DR:** Do a few practice problems to ensure you know the basic syntax and can write Swift code unaided. Next, try to absorb as many questions / answers as possible to build up your mental bank of approaches, strategies, edge cases, etc. Then, continue to periodically review these questions until you can recall the solutions with ease. From here, it's easy to apply or tweak the learnt solutions to address any questions that come your way in the interview.

# iOS & Swift Interview Q & A

In most cases, the initial screening of a candidate is done through a phone screen during which a non-technical recruiter will ask multiple choice or fill-in-the-blank style questions.

In order to help you prepare for this stage of the interview, this section will cover every single “screener” question I've been asked in the last five years. Also included here are questions colleagues have shared with me or questions submitted by other iOS developers on Twitter.

For junior and mid-level positions, these questions cover most of the topics you could reasonably expect to be asked about in your next phone screen or on-site interview.

If you're still on the hunt for more questions, I'd recommend going to the company's Glassdoor page, clicking on the Interview tab, and reading through the responses there.

## General Knowledge

What does app thinning mean?

As the name suggests, app thinning - introduced in iOS 9 - helps developers reduce the size of their app's executable. This reduction in size is accomplished through three mechanisms; slicing, bitcode, and the use of on-demand resources.

Prior to iOS 9, when you installed your app on a device, it would include assets for all device types. For example, you might be on a non-retina screen device, but the app installation would still include retina quality assets. In other words, your application's executable would include all of the resources required to run on all device types.

Unsurprisingly, this would inflate the size of the installed app with assets the user would never be able to use. So, slicing allows developers to include only the relevant assets for the device on which the app is being installed. This is one of the many conveniences asset catalogs afford us.

Typically, when you upload your app to App Store Connect, you are sending a compiled binary file. When you enable the bitcode setting in Xcode, you'll send an intermediate representation of the compiled program instead. The compilation will be completed when it's installed by the user. This delayed approach to compilation allows Apple to identify the device the app is being installed on and pick only the relevant resources and assets to bundle into the final executable alongside introducing additional device-specific optimizations.

On-demand resources are app contents that are hosted on the App Store and are separate from the related app bundle that you download. They enable smaller app bundles, faster downloads, and richer app content. The app requests sets of on-demand resources, and the operating system manages downloading and storage. The app uses the resources and then releases the request. After downloading, the resources may stay on the device through multiple launch cycles, making access even faster.

Nowadays, these settings are enabled by default.

What is the difference between the stack and the heap?

The system uses the stack to store anything on the immediate thread of execution; it is tightly managed and optimized by the CPU.

When a function creates a variable, the stack will store that variable for the lifetime of the function call. Since the stack is so strictly organized, it's very efficient and fast.

The system uses the heap to store reference types. The heap is a large pool of memory from which the system can request and dynamically allocate blocks of memory.

The lifetime of the items on the heap are flexible and dynamic as the heap doesn't automatically destroy its data like the stack does. Instead, explicit allocation and deallocation calls are needed.

This makes creating and removing data from the heap a slower process compared to creating and removing data from the stack.

Is an enum a value type or a reference type?

In Swift, an `enum` is a value type (other value types include `structs` and tuples).

In the following example, you'll see that changes made to one variable have no effect on the other.

This is because the value is simply copied from one variable to another - they don't point to the same location in memory.

```
enum CarBrands {
    case porsche
    case mercedes
}

var porsche = CarBrands.porsche
var mercedes = porsche

porsche = .mercedes

print(porsche) // Output: mercedes
print(mercedes) // Output: porsche
```

What is the difference between an escaping closure and a non-escaping closure?

If you've ever tried to pass in a closure as a parameter to a function, you might have encountered a warning from Xcode prompting you to add the `@escaping` keyword to your parameter declaration. Let's take a closer look at the difference between an escaping and non-escaping closure.

In simple terms, if the closure will be called after the function returns, we'll need to add the `@escaping` keyword. Since the closure is called at a later date the system will have to store it in memory. So, because the closure is a reference type, this will create a strong reference to all objects referenced in its body.

So, `@escaping` is used to indicate to callers that this function can potentially introduce a retain cycle and that they'll need to manage this potential risk accordingly.

```
escapingClosure {
    print("I'll execute 3 seconds after the function returns.")
}

nonEscapingClosure {
    print("This will be executed immediately.")
}

func escapingClosure(closure: @escaping () -> ()) {
    DispatchQueue.main.asyncAfter(deadline: .now() + 3) {
        closure()
    }
}

func nonEscapingClosure(closure: () -> ()) {
    // Some synchronous code...

    // Calls the closure immediately
    closure()

    // Some synchronous code...
}
```

In the case of the `nonEscapingClosure`, we can see that the closure is called immediately and will not live or exist outside the scope of our function. So, the non-escaping closure variety works for us here. This is also the default type for all closures in Swift. As an added benefit, we can use the `self` keyword here without worrying about introducing a retain cycle.

## What are trailing closures?

Trailing closures are simply syntactic sugar in Swift that allow us to implement closures without a ton of boilerplate code. When the final parameter in a call to a function is a closure, trailing closures allow you to define the closure's contents outside of the function call.

```
func sayHello(name: String, closure: () -> ()) -> Void {  
    print("Hello \(name)")  
    closure()  
}  
  
// Typical syntax  
sayHello(name: "Aryaman", closure: {  
    print("Finished saying hello.")  
})  
  
// With trailing closure syntax  
sayHello(name: "Aryaman") {  
    print("Finished saying hello.")  
}
```

While this is convenient, convention is to only use trailing closure syntax when your function only has one closure parameter.

If your function has multiple closure parameters, you should adopt the normal syntax instead.

```
func sayHello(name: String, then: () -> (), finally: () -> ()) -> Void {
    print("Hello \(name)")
    then()
    finally()
}

// With trailing closure syntax
sayHello(name: "Aryaman") {
    print("This is the then closure")
} finally: {
    print("This is the final closure")
}

// Preferred approach
sayHello(name: "Aryaman", then: {
    print("This is the then closure")
}, finally: {
    print("This is the final closure")
})
```

You can see in the example above, the first call to `sayHello()` is harder to read than the second call. When multiple closures are required, it's hard to discern which one does what.

I'd recommend you keep this in mind when completing your take-home assignments.

Additionally, if development team are using a linter like SwiftLint, it will also enforce the same convention:

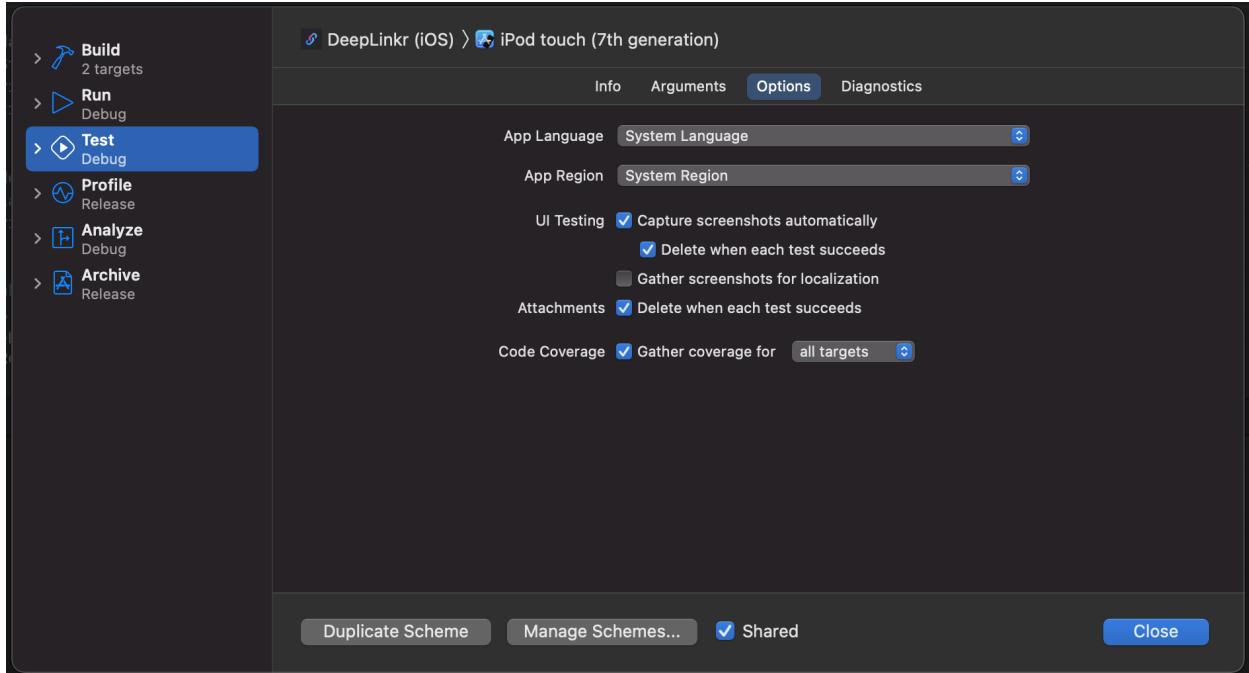
- Use trailing closure syntax when there is only one closure
- Use the long form syntax when there are multiple closures

## What is code coverage? How do you enable it in Xcode?

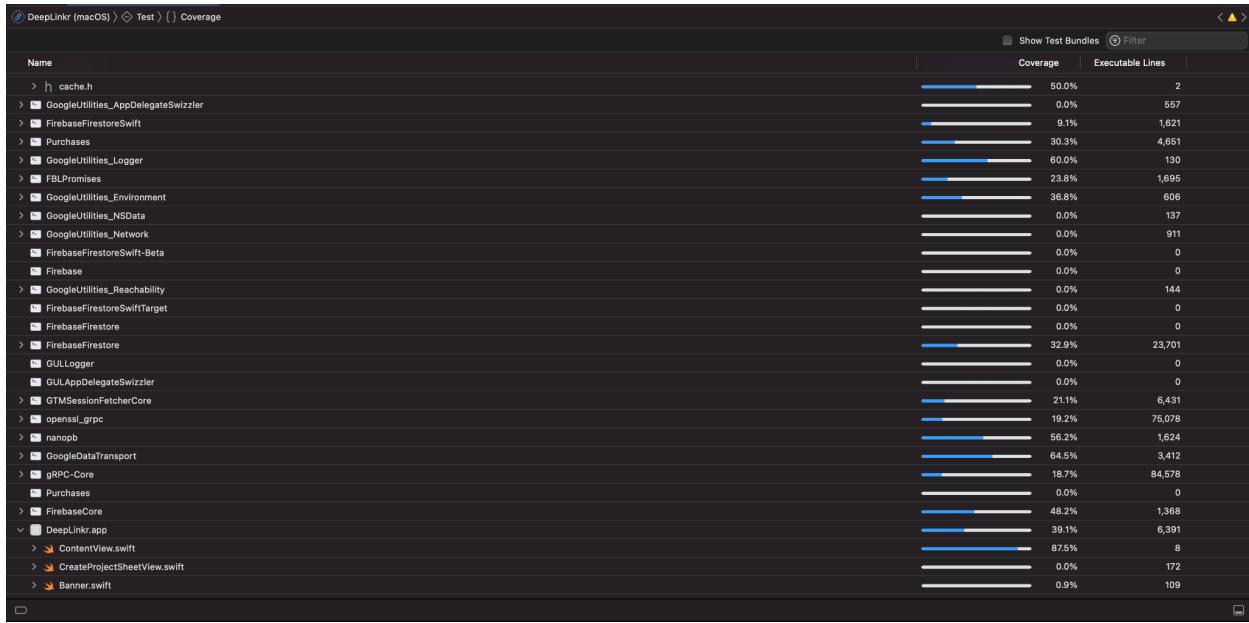
Code coverage allows you to measure what percentage of your codebase is being exercised by your unit tests. More importantly, it helps you identify what sections of your codebase your tests **don't** cover.

This option is disabled by default.

To enable it for your project, edit the target's settings and select the Code Coverage checkbox:



Now, when you run your tests, you'll see a breakdown of what areas of your codebase are covered by tests:



Clearly I've got some work to do...

Additionally, Xcode lets you see Code Coverage within the Editor itself:



The screenshot shows the Xcode editor with a Swift file named 'HollowButton.swift'. The code defines a struct 'HollowButton' that implements the 'ButtonStyle' protocol. The code is annotated with green highlights for covered lines and red highlights for uncovered lines. A sidebar menu on the right shows various editor settings, with 'Code Coverage' selected.

```

1 // HollowButton.swift
2 // HollowButton
3 // Created by Aryaman Sharda on 9/8/21.
4 //
5 import SwiftUI
6
7 struct HollowButton: ButtonStyle {
8     func makeBody(configuration: Configuration) -> some View {
9         configuration.label
10            .foregroundColor(.white)
11            .overlay(RoundedRectangle(cornerRadius: 3, style: .continuous).stroke(Color.white, lineWidth: 3))
12            .cornerRadius(3)
13    }
14 }
15
16
17
18

```

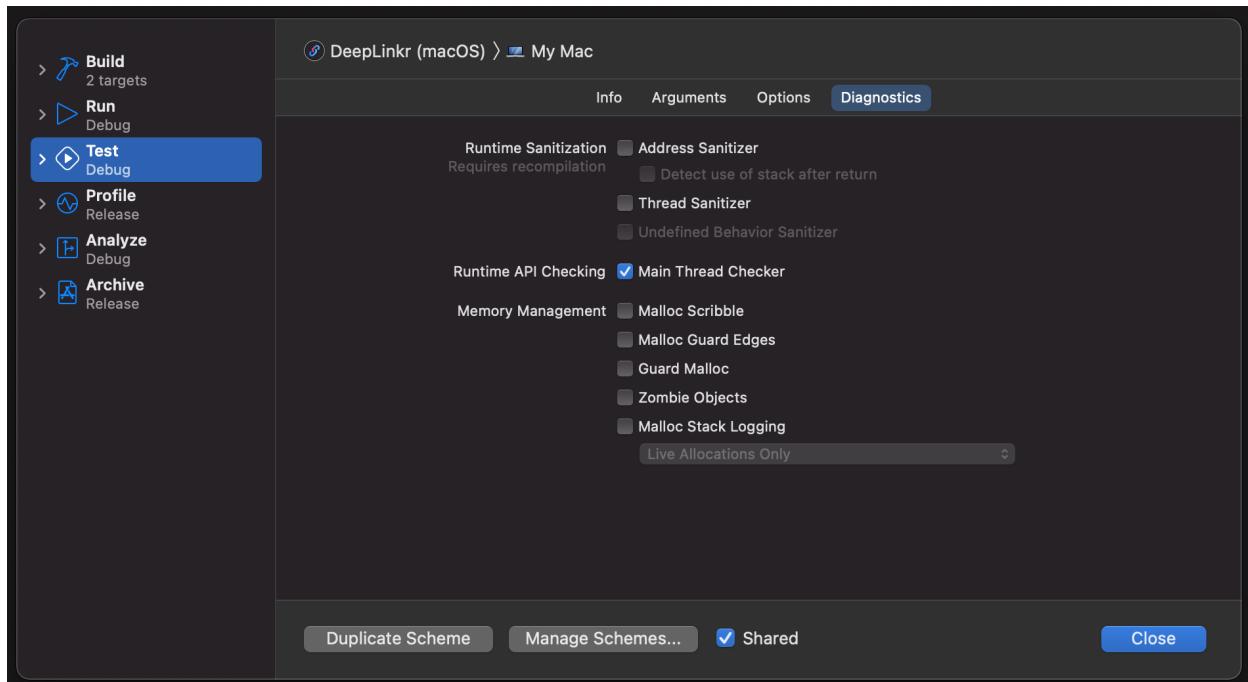
Code sections highlighted in green are covered by tests, while those highlighted in red are missing tests.

## What common problem does the Main Thread Checker help detect?

Quite simply, it's a means of identifying code that should be running on the main thread but is running on a background thread instead.

Since all UI updates should only be performed on the main thread, the Main Thread Checker is often used to help catch instances of UI updates occurring on a background thread.

To enable it, select Edit Scheme → Test → Diagnostics and toggle on the Main Thread Checker:



With this setting enabled, we'll get the following warning if we try and update the UI on a background thread:

```
class ViewController: UIViewController {
    @IBOutlet fileprivate(set) var usernameLabel: UILabel!
    override func viewDidLoad() {
        super.viewDidLoad()
        DispatchQueue.global(qos: .userInitiated).async { [weak self] in
            self?.usernameLabel.text = "@aryamansharda"
        }
    }
}
```

A purple horizontal bar highlights the line of code `self?.usernameLabel.text = "@aryamansharda"`. A small purple triangle icon with the text "UILabel.text must be used from main thread only" is displayed above the bar.

## What are Swift's different assertion options?

Assertions allow us to terminate the execution of our program if a condition is not met at runtime.

While this may seem like a drastic reaction, this feature makes the debugging process far easier. By terminating the execution of our app in this way, we give ourselves an opportunity to investigate *why* our assertion failed and to make whatever changes are necessary.

Swift provides three different assertion types:

### **assert()**

This is useful for basic debugging and allows you to assert that a condition is true before the execution of your program can continue. In the event that the condition is not satisfied, the system will terminate your application, but will allow it to remain in a debuggable state.

`assert()` is only evaluated in Debug mode and is disabled in all Release builds.

This enables you to easily sanity check your implementation during development while ensuring that your end user's experience is unaffected.

```
assert(age > 0 && age < 150, "Either an invalid age or a medical marvel.")  
assert(user.isAuthenticated, "User should be authenticated at this stage.")
```

```
guard let storageKey = model.key as? String else {  
    // assertionFailure() allows us to trigger the failure case directly  
    // without having to evaluate a conditional expression  
    assertionFailure("Cannot save model without a storage key")  
    return  
}
```

It's important to note, though, that your project's optimization settings can change the behavior of this function:

- In Playground and -Onone builds (the default for Xcode's Debug configuration), if the condition evaluates to false, the program will print the specified message and stop execution, but will remain in a debuggable state.
- In -O builds (the default for Xcode's Release configuration), the `assert()` is not evaluated.

- In -Ounchecked builds, the `assert()` is not evaluated, but the optimizer may assume that it always evaluates to true which can easily mask serious programming errors.

### **precondition()**

You can think of `precondition()` as `assert()` with support for Release builds. This assertion type will also stop your program's execution when a condition is not met.

To clarify the difference between `assert()` and `precondition()` further, `assert()` is generally used for sanity checking your code during development whereas `precondition()` is used for guarding against situations that, if they happen, mean your program cannot reasonably proceed. For example, you might use `precondition()` to validate that the arguments passed into your function match the requirements specified by your documentation.

```
precondition(state != .invalid, "App in unknown and non-recoverable state.")  
precondition(containers.count > 0, "Empty container stack.")  
preconditionFailure("Cannot cast \((type(of: objCValue)) to \((Value.self)")  
preconditionFailure("Plist file not found")
```

Similar to `assert()`, the particular compiler optimizations used in your project can influence the function's behavior:

- In Playground and -Onone builds (the default for Xcode's Debug configuration), if the `precondition()` evaluates to false, the program will print a message and stop execution, but will remain in a debuggable state.
- In -O builds (the default for Xcode's Release configuration), if the `precondition()` evaluates to false, the program's execution is stopped.
- In -Ounchecked builds, `precondition()` is not evaluated, but the optimizer may assume that it always evaluates to true which can easily mask serious programming errors.

### **fatalError()**

This assertion type is used in situations where the application has encountered a significant enough error that there is no reasonable way to proceed. If you use `fatalError()` to handle such a scenario, the application will immediately terminate in both Debug and Release builds.

```
fatalError("Unable to load image asset named \((name).")  
fatalError("init() has not been implemented")  
fatalError("ViewController is not of the expected class \((T.self).")
```

The use of this assertion type can be a little polarizing. You'll find that some developers are vehemently against using it while others see certain advantages and respect the deliberateness it offers. Regardless of which camp you find yourself in, you'll want to use this sparingly and only when your application enters a truly unexpected and unrecoverable state.

Unlike the other options we've discussed so far, `fatalError()` will work regardless of any compilation or optimization settings you've enabled on your project.

By using these different types of assertion mechanisms, we can articulate the assumptions in our code and, as a result, write code that is extremely clear about the behavior we expect.

## What are the differences between the static and class keywords?

Both `static` and `class` keywords enable us to attach methods directly to a type rather than to an instance of the type. However, they differ in their ability to support inheritance.

When we use the `static` keyword on a function declaration, that function can no longer be overridden by a subclass. However, if we were to use the `class` keyword instead, overriding this function in a subclass would still be a possibility.

As we've previously discussed, the `final` keyword attached to a `class` or function also prevents it from being subclassed or overridden. Therefore, it may be easier to remember that `static` is equivalent to `final class`.

```
class Dog {  
    class func bark() -> String {  
        return "Woof"  
    }  
  
    static func sit() -> Void {}  
}  
  
class ScoobyDoo: Dog {  
    override class func bark() -> String {  
        "Zoinks!"  
    }  
  
    // ERROR: Cannot override static method  
    override static func sit() -> Void {}  
}
```

As a final point, since functions declared with the `class` keyword can be overridden, this means that they must be dynamically dispatched. In contrast, and unsurprisingly, when you declare a function with the `static` keyword, that function call is now statically dispatched.

## What is the difference between the App ID and the Bundle ID?

During phone screens, especially for Senior iOS roles, I've been tested on my understanding of provisioning profiles, development and distribution certificates, App Store Connect, and everything related to managing an iOS release.

As part of that line of questioning, I've often been asked to clarify the difference between the App ID and the Bundle ID.

The Bundle ID is simply an identifier written in reverse DNS format that **uniquely identifies a single app**.

The following example should look pretty familiar to you:

`com.AryamanSharda.WalkingRoutes`

The Bundle ID can only contain alphanumeric characters and a period.

Since the Bundle ID is specific to an application, it's useful in helping the system distinguish between the installation of a new app or an app update. Also, because a single Xcode project can have multiple targets and therefore output multiple apps, the Bundle ID lets you uniquely identify each of your project's targets.

The App ID is a two-part string used to identify one or more apps from a **single** development team. It consists of an Apple issued Team ID and your application's Bundle ID. Additionally, the App ID is used to specify what App Services (Game Center, iCloud, In-App Purchases, Push Notifications, etc.) are available to your application.

The Team ID is created when you open a new Developer Account with Apple and is unique to your specific development team.

Here's an App ID that matches a specific application:

Explicit App ID: `A123456789.com.AryamanSharda.WalkingRoutes`

We can also have the App ID match multiple applications from the same development team:

Wildcard App ID: [A123456789.com.AryamanSharda.\\*](#)

## What features of Swift do you like or dislike?

Unfortunately, there's no one-size-fits-all answer for a question like this. These types of open-ended questions help the interviewer gauge your understanding and level of depth with a language or technology.

Is the candidate able to compare and contrast it with other languages that solve some implementation problems differently? Does the candidate follow Swift's development? Can they defend their technical opinions?

A common answer to this question is that Swift's optional chaining functionality violates the Law of Demeter.

## What are the different URLSessionConfiguration options?

`URLSession` is the main object responsible for managing HTTP & HTTPS requests in iOS.

In order to create a `URLSession`, we need to initialize it with a `URLSessionConfiguration` type:

```
URLSession(configuration: .default)
```

The default `URLSessionConfiguration` uses a persistent disk-based cache and stores credentials in the user's keychain.

Next, we have the ephemeral `URLSessionConfiguration` type.

```
URLSession(configuration: .ephemeral)
```

It's similar to the default configuration, however it doesn't maintain a cache, store credentials, or save any session-related data to disk. Hence, the name "ephemeral".

Instead, all session-related data is stored in RAM. The only time an ephemeral session writes data to disk is when you explicitly instruct it to write the contents of a URL to a file.

Finally, we have the background configuration type:

```
URLSession(configuration: .background(withIdentifier: "IDENTIFIER_NAME"))
```

This creates a `URLSessionConfiguration` object that allows HTTP and HTTPS uploads or downloads to be performed in the background. This configuration is most commonly used when transferring data files while the app runs in the background.

A `URLSession` configured with this type hands control of the transfer over to the system which then handles the file transfer in a separate process. In iOS, this configuration makes it possible for transfers to continue even when the app is suspended or terminated.

## What is the difference between static and dynamic dispatch?

Swift, like many other languages, allows a **class** to override methods and properties declared in its **superclass**. As a result, the program has to determine at **runtime** the correct version of the method or property to use.

This process is called dynamic dispatch.

Dynamic dispatch is implemented with the help of a method table (a.k.a. witness table). The witness table is used to figure out which implementation of a function to call - the **superclass**'s implementation or the **subclass**'s implementation.

Dynamic dispatch enables polymorphism which allows us to increase the expressiveness of our code, but it introduces a constant amount of runtime overhead every time we call a function. As a result, you'll typically want to avoid polymorphism in performance sensitive code.

If you don't need dynamic dispatch (you don't want the function or property to be overridden), you can improve performance by using the **final** keyword which prohibits overriding. This will allow us to use static dispatch instead which doesn't incur this performance penalty. Moreover, static dispatch allows us to leverage additional compiler optimizations.

Finally, static dispatch is supported by both value and reference types. Dynamic dispatch is only supported by reference types as it requires inheritance and value types can't support inheritance.

## What is the difference between the UIApplicationDelegate and SceneDelegate?

Prior to iOS 13, the **AppDelegate** was the main entry point for your application. This is where you would typically start the configuration of your third-party dependencies and establish the starting conditions for your application.

However, as of iOS 13, some of the responsibilities of the **AppDelegate** have been transitioned to the **SceneDelegate**. This change is due to the new multi-window support feature introduced with iPadOS.

With multi-window support, we'll still have one application, but it can have multiple windows (e.g. imagine Google Chrome or Safari). So, we'll need a separate object whose sole purpose is to manage the window(s) of the application.

The **AppDelegate** will continue to be responsible for the application lifecycle and initial setup, but the **SceneDelegate** will now be responsible for what is shown on the screen.

As part of this transition, the **SceneDelegate** will enable us to create multiple instances of our app's UI all backed by the same **AppDelegate**. This new multi-window support also means that each of these instances will appear as separate views in iOS's application switcher.

Moreover, each window is meant to work independently from one another, so now screens can independently move from the foreground to the background or vice-versa.

**AppDelegate**'s responsibilities are otherwise unchanged. It will still be responsible for setting up any data needed for the duration of the application, configuring your app's scenes, registering for external services like push notifications, and managing the application's lifecycle.

In a nutshell, the **SceneDelegate** manages the iOS app's UI lifecycle methods whereas the **AppDelegate** only handles the iOS app's application lifecycle methods.

## What is the difference between a dynamic library and a static library?

As your application matures and your application size and launch speed start to suffer, you'll likely find yourself re-evaluating how you integrate libraries into your project.

Libraries and frameworks can either be linked statically or dynamically.

Static libraries are collections of object files (the machine code output after compilation) grouped into a single containing resource. This library will then be copied into the larger executable that eventually runs on your device. If you've ever seen a file ending in .a, it's a static library.

Imagine a suitcase filled with everything you need for your vacation. A static library is similar; everything you need in order to run your application is included in the executable itself. Static libraries **cannot contain** images, sound files, media, etc. - they can only store code files.

Dynamic libraries (.dylib files) are loaded into memory when needed instead of being bundled with the executable itself. All iOS and macOS system libraries are actually dynamic.

The main advantage here is that any application that relies on these dynamic libraries will benefit from all future speed improvements and bug fixes in these libraries without having to create a new release. Additionally, dynamic libraries are shared between applications, so the system only needs to maintain one copy of the resource. Since it's shared and only loaded when needed, invoking code in a dynamic library is slower than a static one.

Let's take a detailed look at the advantages and disadvantages:

### Static Libraries

#### Pros

- Guaranteed to be present and the correct version at runtime.
- The application can run without any external dependencies. You don't need to monitor library updates since the object files are part of the executable itself. As a result, it becomes standalone and can move from platform to platform.
- Faster performance compared to calls made to a dynamic library.

#### Cons

- Makes the executable larger as it simply just contains more code.
- Your application will be slower to launch as the library needs to be loaded into memory.

- Any changes in a static library require the application to be re-compiled and re-distributed.
- You have to integrate the entire library even if you only rely on a small portion of it.

## Dynamic Libraries

### Pros

- Doesn't increase app size.
- Better application launch speech as the library is loaded only when needed.
- Only the relevant section of the library needed for the currently executing code is loaded instead of loading the library in its entirety.

### Cons

- Application may crash if the library updates are not compatible with your application (i.e. business logic / iOS version).
- Application may crash if the dynamic library cannot be loaded or found.
- Calls to dynamic library functions are slower than calls made to a static library.

There's no one size fits all answer. You'll have to make a pragmatic decision and weigh how much you value performant library calls, app size, launch time, etc. and pick the approach or a hybrid approach that best suits your needs.

## What are generics and what problem do they solve?

Generic classes and functions not only allow you to write more reusable code, but they also help you write less code overall. In simple terms, generics allow you to write the code once, but apply it to a variety of different data types.

While the syntax appears daunting at first, after some practice, generics become much more approachable.

Consider a function, `exists()`, which simply checks if some specific value exists within an array of elements:

```
func exists<T: Equatable>(item:T, elements:[T]) -> Bool
```

This function declaration states that we are going to pass something into `exists()` that conforms to `Equatable` (see `<T: Equatable>`). We don't know what entity that will be yet, but we promise it will implement the `Equatable` protocol when the time comes.

We use the variable `T` to represent this placeholder type. The method declaration then continues on to say that the remaining parameters will also expect some input matching the type of `T`.

With this method, we can use an input array of any `Equatable` type and easily check if it contains the target element - `item`. We are free to use any custom objects, structs, primitive data types, or anything else that implements `Equatable`.

Since the items in `elements` all implement the `Equatable` protocol, we can simply use `==` to check for equality:

```
// Here's our generic exists() function
func exists<T: Equatable>(item:T, elements:[T]) -> Bool {
    for element in elements {
        if item == element {
            return true
        }
    }

    return false
}

// The main takeaway here is that we've written the code once, but
```

```
// we can apply it over a wide variety of data types.

// Output: true
exists(item: "1", elements: ["1", "2", "3", "4"]))

// Output: false
exists(item: -1, elements: [1, 2, 3, 4])

// Output: true
exists(item: CGPoint(x: 0, y: 0), elements: [CGPoint(x: 0, y: 0),
                                              CGPoint(x: 0, y: 1),
                                              CGPoint(x: 0, y: 2)])
```

Here's an advanced Swift example:

```
extension UIView {
    // We're saying that `T` is eventually going to be a `UIView`.
    //
    // It could be a `UIButton`, `UIImageView`, or
    // an ordinary `UIView` - it doesn't matter.
    //
    // Now, we can load any `UIView` or a subclass from a `.Nib`.

    class func fromNib<T: UIView>() -> T {
        return Bundle(for: T.self).loadNibNamed(
            String(describing: T.self), owner: nil, options: nil
        )![0] as! T
    }
}

// Usage: let view = RestaurantView.fromNib()
// Usage: let cell = PictureCell.fromNib()
// Usage: let photoGallery = GalleryView.fromNib()
```

## What is a .dSYM file?

Whenever we upload our app to Apple, we remove some information (called symbols) from the compiled executable which specify exactly what functions and variables are being referenced.

This intentional process of removing this data from our executable can not only help reduce the size of our application's binary, but also helps in making our application more difficult to reverse engineer.

Without the inclusion of these symbols, our crash logs look like this:

```
0  libswiftCore.dylib          0x000000018f3c9380 0x18f394000 + 217984
1  libswiftCore.dylib          0x000000018f3c9380 0x18f394000 + 217984
2  libswiftCore.dylib          0x000000018f3c8844 0x18f394000 + 215108
3  libswiftCore.dylib          0x000000018f3a74e0 0x18f394000 + 79072
4  libswiftCore.dylib          0x000000018f3ab0d8 0x18f394000 + 94424
5  F49088168M                0x00000001045ac750 0x104590000 + 116560
6  F49088168M                0x00000001045b7904 0x104590000 + 162052
7  F49088168M                0x00000001045b897c 0x104590000 + 166268
8  F49088168M                0x000000010459d914 0x104590000 + 55572
9  F49088168M                0x00000001045a0e70 0x104590000 + 69232
10 F49088168M               0x00000001045a0f4c 0x104590000 + 69452
```

Clearly, it's hard to tell what's going on - all we see are memory addresses.

That's where the .dSYM file comes in.

The .dSYM file (debug symbol file) contains the information required to convert a stack-trace into a human-readable format. This file is automatically created with every release build and is used by Xcode to put the symbols back into the crash report thereby allowing you to read it properly.

Through a process known as re-symbolication, we can leverage our .dSYM file to convert our crash logs to something like this instead:

```
0  libswiftCore.dylib          0x000000018f3c9380 closure #1 in
closure #1 in closure #1 in _assertionFailure+ 217984
```

```
(___.file:line:flags:) + 452
1 libswiftCore.dylib           0x000000018f3c9380 closure #1 in
closure #1 in closure #1 in _assertionFailure+ 217984
(___.file:line:flags:) + 452
2 libswiftCore.dylib           0x000000018f3c8844 _assertionFailure+
215108 (___.file:line:flags:) + 468
3 libswiftCore.dylib           0x000000018f3a74e0
_ArrayBuffer._checkInoutAndNativeTypeCheckedBounds+ 79072
(___.wasNativeTypeChecked:) + 208
4 libswiftCore.dylib           0x000000018f3ab0d8
Array.subscript.getter + 84
5 F49088168M                 0x00000001045ac750 static
ELM327ResponseManager.getResponse(responseStr:obd2Protocol:) + 116560
(ELM327ResponseManager.swift:27)
6 F49088168M                 0x00000001045b7904
ELM327Client.dataInput(___.characteristicUuidStr:) + 162052
(ELM327Client.swift:56)
7 F49088168M                 0x00000001045b897c protocol witness for
BLEClientInputPort.dataInput(___.characteristicUuidStr:) in conformance
ELM327Client + 166268 (<compiler-generated>:0)
8 F49088168M                 0x000000010459d914
BLEConnection.peripheralDataReceived(data:characteristicUuidStr:) + 55572
(BLEConnection.swift:124)
9 F49088168M                 0x00000001045a0e70
BLEConnection.peripheral(___.didUpdateValueFor:error:) + 69232
(BLEConnection.swift:293)
10 F49088168M                0x00000001045a0f4c @objc
BLEConnection.peripheral(___.didUpdateValueFor:error:) + 69452
(<compiler-generated>:0)
```

You'll see that our crash logs now contain real method and variable names which makes debugging far easier.

Some services, like Crashlytics, will automatically re-symbolicate the crash reports for you so they're more human readable. This process allows us to ensure our crash logs are obfuscated for everyone else, but still readable and useful to us as developers.

Simply put, removing these symbols from our executable helps us ensure that our app is not only difficult to reverse engineer, but also allows us to reduce our application's binary size. Then, when needed, we can use the .dSYM file to reverse the process.

What are the differences between a class and a struct?

Here's a high-level summary of the differences:

Features	Classes	Structs
Type	Reference	Value
Implement Protocols	Yes	Yes
Define Properties	Yes	Yes
Define Methods	Yes	Yes
Define Initializers	Yes	Yes
Be Extended	Yes	Yes
Support Inheritance	Yes	No

The main takeaway here is that a **class** is a **reference-type** and a **struct** is a **value-type**.

When you pass reference-types around in your program (i.e. as a parameter), you are actually passing a direct reference to the memory location of that object. As a result, different parts of your program can easily share and modify your object since they're all referencing the exact same place in memory.

When you pass a **struct** or **enum** to a function, you're only passing along a copy of the data in them. So, even if you modify properties of the passed in value-type, the original one remains unchanged as you're effectively just modifying a duplicate. This type of behavior is called **pass-by-value semantics** and is helpful in preventing situations where one part of the code inadvertently changes values or unknowingly affects the application's state.

Both a **class** and a **struct** can implement **protocols**, define properties, methods, initializers, and be extended, but only a **class** can support inheritance and by extension polymorphism.

Since a **struct** requires less overhead to create and is safer to use due to its immutability and **pass-by-value semantics**, Apple's recommendation is to start with a **struct** when you can and change to a **class** only when needed.

However, if the entity you're working with contains a lot of data, then it's probably useful for it to be a **class** so you can share a reference to it and only incur the memory cost once.

What are the different execution states your application can be in?

There are 5 distinct states an iOS app can find itself in:

### **Not running**

This is when the app has not been launched or was previously running but has now been terminated by the system.

### **Inactive**

This occurs when the app is actively running in the foreground, but is not receiving events. This state tends to be brief as the app transitions to some other state.

### **Active**

This is the normal mode for most apps; the app is running in the foreground and receiving and responding to events.

### **Background**

This state describes an app that is running in the background, but is still executing code. Most applications tend to enter this state on the way to being suspended. Furthermore, apps that require extra execution time may remain in this stage longer.

Apps that are launched directly into the background will enter this state instead of the inactive state.

### **Suspended**

This state describes an application that is running in the background but is not executing code.

Apps that are suspended will remain in memory, but will be terminated by the system if a low-memory condition occurs.

How can we handle changes in our application's lifecycle state?

Our application's `AppDelegate` allows us to easily handle our application's lifecycle changes.

`UIApplication` will trigger one of the following delegate methods whenever our application changes its lifecycle status.

```
optional func application(_ application: UIApplication,  
                         willFinishLaunchingWithOptions launchOptions:  
                         [UIApplication.LaunchOptionsKey : Any]? = nil)  
                         -> Bool
```

This method is called at launch time and is our app's first chance to execute code.

```
optional func application(_ application: UIApplication,  
                         didFinishLaunchingWithOptions launchOptions:  
                         [UIApplication.LaunchOptionsKey : Any]? = nil)  
                         -> Bool
```

We can use this method to perform any final initialization and setup before our app is displayed to the user.

`optional func applicationWillBecomeActive(_ application: UIApplication)`

Informs the application that it is about to become the foreground app. If we have any last-minute preparation we need to do, we should do that here.

`optional func applicationWillResignActive(_ application: UIApplication)`

Informs the application that it is transitioning away from being the application in the foreground.

`optional func applicationWillEnterBackground(_ application: UIApplication)`

Notifies your app that it is now running in the background and may transition to the suspended state at any time.

`optional func applicationWillEnterForeground(_ application: UIApplication)`

Notifies your app that it is transitioning from the background state to the foreground state, but it is not yet active.

`optional func applicationWillTerminate(_ application: UIApplication)`

This method notifies your app that it is being terminated and provides you with an opportunity to do any last-minute cleanup. It's important to note that this method is only called if your application is terminated without being transitioned to background mode.

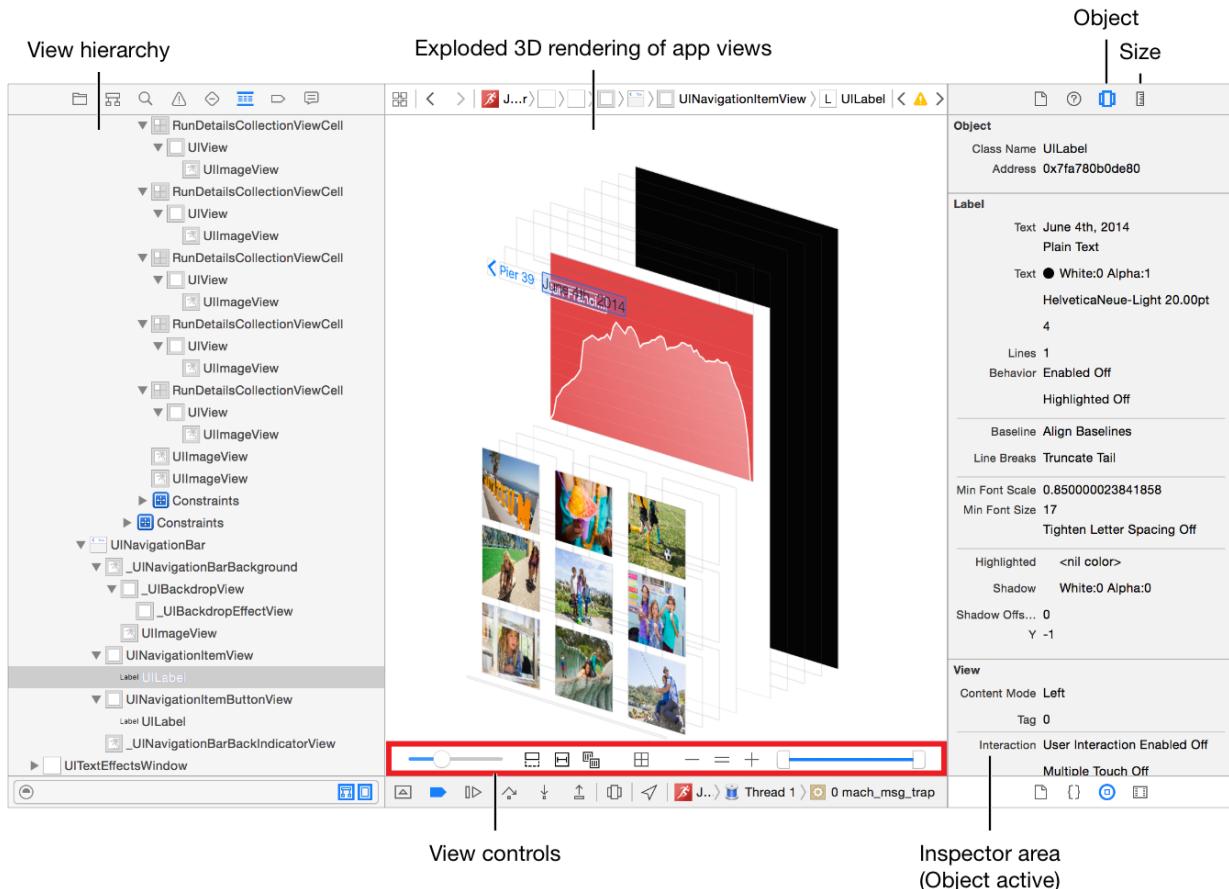
## How would you debug view layout issues?

As an iOS developer, layout problems are an inevitability. From breaking constraints to weird UI issues occurring only at runtime (text truncation, alpha value issues, broken animations, etc.), debugging layout issues in iOS can be tricky.

Therefore, it's crucial to know all of the different approaches to debugging layout issues in iOS.

This is particularly useful during an interview when time is of the essence. Moreover, it's a great way to demonstrate your familiarity with advanced Xcode features.

### Debug View Hierarchy



The Debug View Hierarchy pauses the application in its current state thereby providing the programmer an opportunity to inspect and better understand the UI hierarchy of their app.

This view will not only show you an “exploded” 3D version of your view, but it helps you understand the full hierarchy of the view from the top most view controllers all the way down to individual subviews, `UILabels`, `UIImageViews`, etc. Additionally, this tool will also highlight any `UIView`'s with runtime constraint errors.

There's a lot of functionality here, so I'd recommend spending some time playing around with it if you're unfamiliar. It can help you catch breaking constraints, clipped views, and a variety of other layout issues.

### Customizing Constraint Identifiers

Troubleshooting constraint issues is particularly challenging because the error messages are not very user-friendly.

To make things easier, we can leverage the `identifier` property on a `NSLayoutConstraint`.

This property is available to use regardless of whether the constraint is defined through a `.storyboard`, `.xib`, or programmatically.

```
var bannerWidthConstraint: NSLayoutConstraint?  
bannerWidthConstraint.identifier = "Promotional banner width"
```

A custom identifier makes it easier for you to distinguish between system-generated and user-generated constraints in Debug Logs.

By leveraging custom identifiers, the Debugger output will now contain clearer error messages which will make it much easier to track down and resolve layout issues.

#### Without Identifier:

Will attempt to recover by breaking constraint

```
<NSLayoutConstraint:0x7a87b000 H:[UILabel:0x7a8724b0'Name'(>=400)]>
```

#### With Identifier:

Will attempt to recover by breaking constraint

```
<NSLayoutConstraint:0x7b56d020 'Label Width'  
H:[UILabel:0x7b58b040'Name'(>=400)]>
```

As you can see, these identifiers allow you to quickly and easily identify specific constraints in the log output.

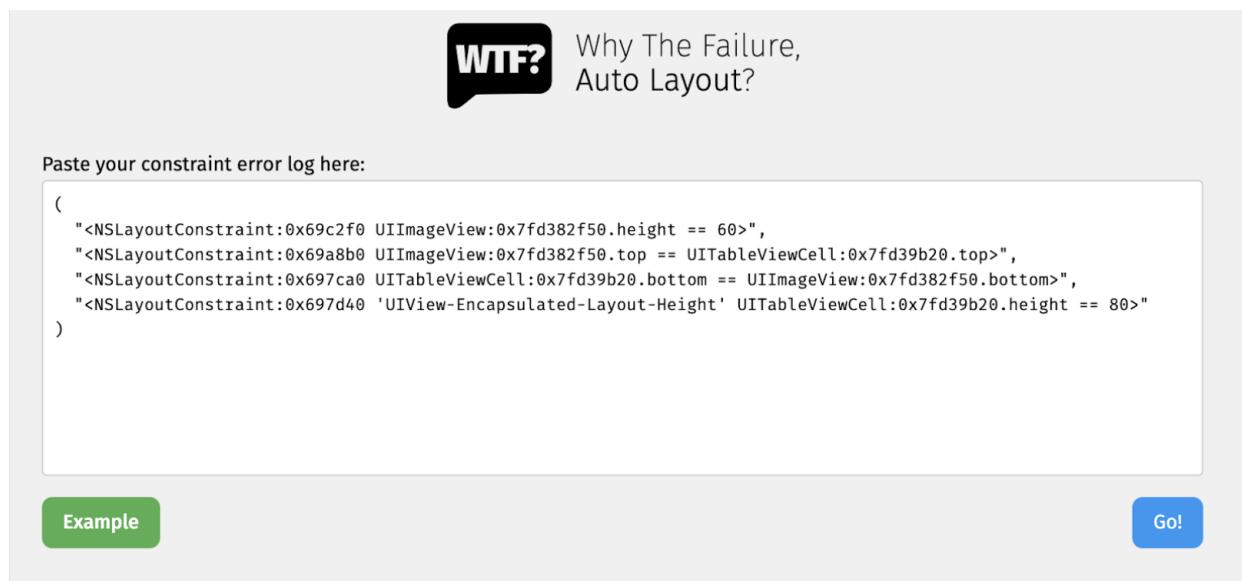
### exerciseAmbiguityInLayout()

This method randomly changes the frame of a view with an ambiguous layout between its different valid values, causing the view to move in the interface. This makes it easy to visually identify what the different valid frame configurations are and helps the developer understand what constraints need to be added to the layout to correctly and fully specify the layout of the view.

This method should only be used for debugging purposes; no application should ship with calls to this method.

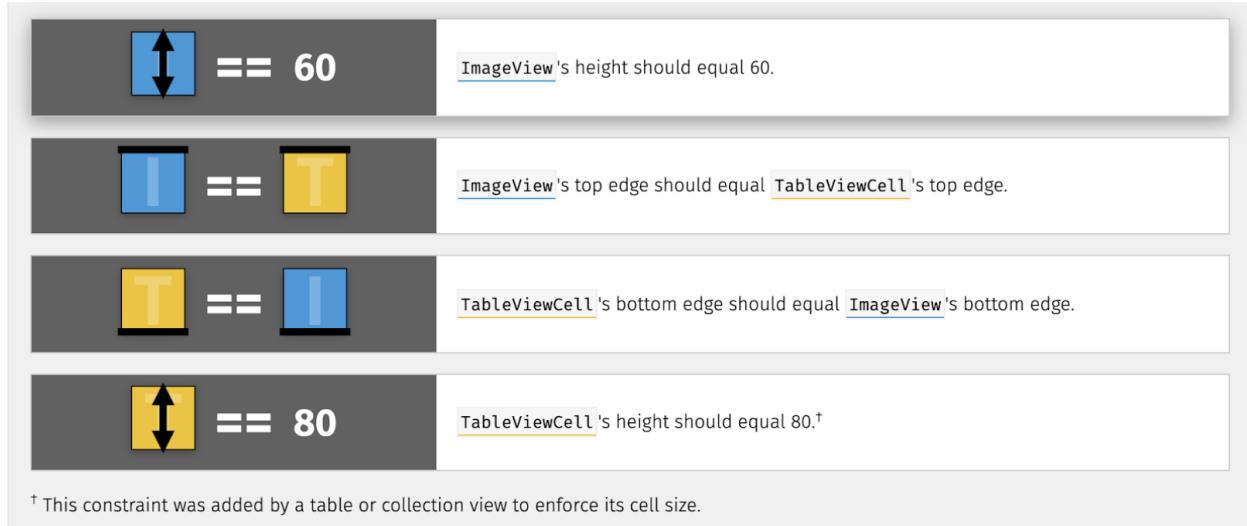
**Developer Tool:** <https://www.wtfautolayout.com/>

If you're having difficulty making sense of the Debugger's "breaking constraint" output, you can use this site to help you easily visualize the problem:



The screenshot shows the homepage of the [WTF? Auto Layout](https://www.wtfautolayout.com/) website. At the top right is a logo consisting of a speech bubble containing the letters "WTF?" in white. To the right of the logo is the text "Why The Failure, Auto Layout?". Below the logo is a text input field with the placeholder "Paste your constraint error log here:". Inside the input field is a sample of Objective-C code representing layout constraints. At the bottom left is a green button labeled "Example", and at the bottom right is a blue button labeled "Go!".

```
(<NSLayoutConstraint:0x69c2f0 UIImageView:0x7fd382f50.height == 60>,
<NSLayoutConstraint:0x69a8b0 UIImageView:0x7fd382f50.top == UITableViewCell:0x7fd39b20.top>,
<NSLayoutConstraint:0x697ca0 UITableViewCell:0x7fd39b20.bottom == UIImageView:0x7fd382f50.bottom>,
<NSLayoutConstraint:0x697d40 'UIView-Encapsulated-Layout-Height' UITableViewCell:0x7fd39b20.height == 80>)
```



The screenshot shows four constraint configurations in Interface Builder:

- Top Constraint:** An  **== 60** constraint between an  and a . The text indicates: `ImageView's height should equal 60.`
- Top Edge Constraint:** An  **==** constraint between an  and a . The text indicates: `ImageView's top edge should equal TableViewCell's top edge.`
- Bottom Edge Constraint:** An  **==** constraint between an  and an . The text indicates: `TableViewCell's bottom edge should equal ImageView's bottom edge.`
- Height Constraint:** An  **== 80** constraint between a  and an . The text indicates: `TableViewCell's height should equal 80.†`

† This constraint was added by a table or collection view to enforce its cell size.

## What is operator overloading?

Operator overloading allows you to change how existing operators interact with custom types in your codebase. Leveraging this language feature correctly can greatly improve the readability of your code.

For example, let's say we had a **struct** to represent money:

```
struct Money {  
    let value: Int  
    let currencyCode: String  
}
```

Now, imagine we're building an e-commerce application. We'd likely need a convenient way of adding up the prices of all items in our shopping cart.

With operator overloading, instead of only being able to add numeric values together, we could extend the `+` operator to support adding **Money** objects together. As part of this implementation, we could even add logic to convert one currency type into another!

When we want to create our own operator, we'll need to specify whether it's of the **prefix**, **postfix**, or **infix** variety.

**prefix** - describes an operator that comes before the value it is meant to be used with (e.x. `!isEmpty`)

**postfix** - describes an operator that comes after the value it is meant to be used with (e.x. the force-unwrapping operator - `user.firstName!`)

**infix** - describes an operator that comes in between the value it is meant to be used with and is the most common type (e.x. `+`, `-`, `*` are all **infix** operators)

In order to take advantage of this language feature, we just need to provide a custom implementation for the operator in our type's implementation:

```
struct Money {  
    let value: Int  
    let currencyCode: String  
  
    static func + (left: Money, right: Money) -> Money {  
        return Money(value: left.value + right.value,  
    }
```

```
        currencyCode: left.currencyCode)
    }
}

let shoppingCartItems = [
    Money(value: 20, currencyCode: "USD"),
    Money(value: 10, currencyCode: "USD"),
    Money(value: 30, currencyCode: "USD"),
    Money(value: 50, currencyCode: "USD"),
]

// Output: Money(value: 110, currencyCode: "USD")
print(shoppingCartItems.reduce(Money(value: 0, currencyCode: "USD"), +))
```

## What are the differences between a delegate and a notification?

Delegates and notifications are different mechanisms for informing objects about events or actions taking place in other parts of your app. Though they differ fundamentally in how they communicate, both of these options can help reduce coupling between entities in your code.

Let's look at delegates first.

A delegate is like a phone call - not only can you communicate with the person on the other end, but they can also communicate with you. In a similar way, delegates are used to establish two-way communication between objects.

There's no need for the delegating object to know specifics about the object it's talking to; the receiving object simply needs to implement the required protocol.

Typically, delegates are used when you want the object receiving the events to influence the sending object. We know, for example, that a class that implements the `UITableViewDelegate` will be notified when events occur in a `UITableView` (e.g. the selection of a cell). This class can then handle the event in whatever way it wishes (e.g. present a new view, issue a command to `UITableView`, etc.).

In contrast, a notification is more of a broadcast than a strict two-way communication. For example, whenever a user changes time zones or night mode is activated, iOS notifies all listening objects so they can adjust accordingly.

Notifications help reduce coupling between the sending and receiving objects as the sending object simply publishes a notification, but is unaware of who or if anyone is listening. Unlike with delegates, the objects receiving the notification cannot communicate with or interact with the sender as notifications are only a one-way communication.

In summary, the main difference between a delegate and a notification is that the former is used for one-to-one messaging while the latter is used for one-to-many messaging.

The appropriate choice will depend on your use case.

## What are the differences between a library and a framework?

TL;DR: The difference between a library and a framework is that you'll typically call a library, but a framework will call you.

### Library

A library is a collection of functions that each perform some work and then return control to the caller. Libraries can only contain executable code; no other assets or media.

Libraries usually contain highly tested and sophisticated code meant to address some particular problem. So, it'll often make sense to reuse code written by other developers rather than implementing it from scratch yourself. For example, many libraries exist for complicated topics like physics, audio processing, image manipulation, etc. Clearly, it would be unrealistic to implement all of that yourself.

You can think of a library like a trip to IKEA. You already have a home filled with furniture, but there's a few rooms that you need help furnishing. Instead of making it yourself, you can go to IKEA (the library) and pick and choose the relevant pieces you need. Throughout this experience, you - the programmer - are in control.

An example on iOS would be the open-source Charts library which lets you easily create bar, line, and pie graphs. It provides all of the functionality, but it's up to the programmer to decide exactly when, where, and how it should be used.

### Framework

A framework is similar to a library, but instead leaves openings for you to influence its behavior and execution. This is accomplished most commonly through subclassing, dependency injection, and delegation. Unlike libraries, frameworks can contain other media types (images, audio, etc.) - not just code.

The main distinction between a library and a framework is that there is an inversion of control. When you use a framework, you'll have a few opportunities to plug in your code, but the framework is in charge of when - and if at all - your custom code is executed. The framework has effectively inverted the control of the program; it's telling the developer what it needs and decides when to execute it.

Consider a framework like Vapor or a location tracking framework. They both provide complicated functionality, but leave little openings for you to introduce your own custom logic.

## How do you create interoperability between Objective-C and Swift?

If you're applying for an older company like YouTube, Facebook, or even Google, you'll likely have to work with both Objective-C and Swift within the same project.

As part of that workflow, it's useful to know how to expose variables, classes, and methods declared in Swift to the Objective-C runtime and vice-versa.

### Exposing Objective-C to Swift

To import a set of Objective-C files into Swift code within the same app target, you rely on an Objective-C Bridging Header file to expose those files to Swift. Xcode offers to create this header for you when you add a Swift file to an existing Objective-C app or an Objective-C file to an existing Swift app.

When you accept, Xcode creates the Bridging Header file and names it by using your product module's name followed by `"-Bridging-Header.h"` (e.x. `"Facebook-Bridging-Header.h"`).

In your newly created Bridging Header file, you can specify all of the Objective-C headers you want to expose to Swift like this:

```
#import "Properties.h"  
#import "Auth.h"
```

Now, these Objective-C entities are automatically available in any Swift file within the same target without the use of any additional `import` statements.

### Exposing Swift to Objective-C

Now, what if you wrote a cool extension in Swift that you want to have access to in Objective-C?

You can work with types declared in Swift from within the Objective-C code in your project by importing an Xcode-generated header file (e.x. `"ProductName-Swift.h"`). This file is an Objective-C header that declares the Swift interfaces in your target.

You don't need to do anything special to use the generated header—just import it in the Objective-C classes as needed:

```
#import "ProductName-Swift.h"
```

This header file is managed behind the scenes. As a result, you don't need to specify each individual class you want to expose to Objective-C like we did in the previous section.

You can think of it as an umbrella header for all of your Swift code.

By default, the generated header contains interfaces for Swift declarations marked with the `public` or `open` modifier. If your app target has an Objective-C bridging header, the generated header also includes interfaces for resources marked with the `internal` modifier. Declarations marked with the `private` or `fileprivate` modifier don't appear in the generated header and aren't exposed to the Objective-C runtime unless they are explicitly marked with a `@IBAction`, `@IBOutlet`, or `@objc` attribute.

## What does the File's Owner do?

When you load a .xib file and specify the owner property, the class responsible for loading the .xib now becomes the File's Owner.

```
open func loadNibNamed(_ name: String, owner: Any?,  
                      options: [UINib.OptionsKey : Any]? = nil) -> [Any]?  
  
// In ViewController.swift  
Bundle.main.loadNibNamed("ExampleView", owner: self, options: nil)
```

Put simply, the File's Owner is responsible for loading the .xib and facilitating communication between the code and the elements defined in the view. In the example above, ViewController initiates loading the .xib file thereby making it the File's Owner. As a result, it will now serve as the middle-man between the .xib file and our application's code.

Once the .xib file has completed loading, the File's Owner is responsible for managing the view's contents and binding all of the declared **IBOutlets** and **IBActions** in your code to the view's corresponding UI components.

When we specify the File's Owner in our .xib directly, we're effectively assigning a placeholder value that says -"This class will load me, interact with my UI, and create the necessary bindings to my various UI elements".

It's important to recognize that the File's Owner is an independent entity and not a part of, nor bound to, the .xib itself - it's just the class that assumes this middle-man responsibility.

Are closures value or reference types?

Closures are reference types.

When we use closures, we want them to be able to reference all of the variables from their surrounding context (like class and local variables). This means when the closure modifies a captured reference-type variable in its definition, we're also affecting the variable's value outside of the closure's scope.

```
var money = Money(value: 20, currencyCode: "USD")  
  
//Output: Money(value: 20, currencyCode: "USD")  
print(money)  
  
let closure = {  
    money = Money(value: 200, currencyCode: "USD")  
}  
  
closure()  
  
//Output: Money(value: 200, currencyCode: "USD")  
print(money)
```

If closures were value types, then they would only have access to a **copy** of the variables in their surrounding context instead of a direct reference to the variables themselves. So, if the value of any of these referenced variables changed, the closure would be none the wiser and would be operating on the now out-of-date values.

Sometimes, we'll want this behavior though.

We can specify a capture list in our closure's definition which will create an immutable read-only copy of the variables we've listed. This way, changes made within the closure's definition will not affect the value of the variables outside of the closure.

```
var money = Money(value: 20, currencyCode: "USD")  
  
//Output: Money(value: 20, currencyCode: "USD")  
print(money)  
  
let closure = { [money] in
```

```
    print(money)
}

money = Money(value: 200, currencyCode: "USD")

//Output: Money(value: 20, currencyCode: "USD")
closure()

//Output: Money(value: 200, currencyCode: "USD")
print(money)
```

In this example, even though we're modifying the `money` variable before we execute the closure, the closure only has access to a copy of the variable's value - not a reference to the variable itself. So, any changes made to `money` outside of the closure will not affect the value of `money` the closure operates on.

## What are the differences between Keychain and UserDefaults?

`UserDefault`s and `Keychain` are both useful in storing small key-value pairs on the user's device, but their security capabilities differ greatly.

`Keychain` is the only native option for storing data on an iOS device in an encrypted manner.

It's typically meant for storing small amounts of data like an access token, credentials, or other sensitive information. However, while it is still the most secure offering on iOS, it's important to know that `Keychain` data can be accessed on jailbroken devices.

Since `Keychain` is implemented as an SQLite database stored on the file system, it is slower than `UserDefault`s. Lastly, values stored in the `Keychain` will persist across application deletions and re-installs unless explicitly deleted.

`UserDefault`s also allows you to store key-value pairs across different invocations of your app, but it is not secure.

Values stored in `UserDefault`s are eventually written to a `.plist` file which are entirely human-readable. `UserDefault`s are usually used to store basic key-value pairs and user preferences. If you don't need the `Keychain`'s security features, `UserDefault`s are the more convenient choice. Finally, unlike in the `Keychain`, items saved in `UserDefault`s will not persist across application deletions and re-installs.

What are our options for storage and persistence?

Some of the options include:

### **User Defaults**

We can use **UserDefaults** to store simple key-value pairs in an insecure manner. Typically, you would only use **UserDefaults** to store something lightweight like a user setting.

### **.plist**

We can use a .plist to store larger data sets. It's a really flexible human-readable format.

### **Keychain**

This is the only encrypted persistent storage option available on iOS and is used for storing highly sensitive key-value pairs (primarily credentials).

### **Disk Storage**

We can serialize data, domain models, or other downloaded content and save them directly to disk.

### **Core Data / SQLite**

Useful in cases where we have larger data sets and are interested in making queries on the data.

## When would you use a struct versus a class?

Typically, you'll want to use a **struct** if any of the following conditions apply:

- Use a **struct** when encapsulating simple data types
- When you need thread safety as **structs** are passed-by-value
- You want pass-by-value semantics
- When the properties defined inside the entity are mostly value types
- You don't need inheritance
- You don't need mutability
- When you want automatic memberwise initializers

Apple's recommendation is to start with a **struct** and transition to a **class** only if you need inheritance or pass-by-reference semantics. However, if your entity is storing a lot of data then it may make sense to use a **class** so you're only incurring the memory cost once.

What do the Comparable, Equatable, and Hashable protocols do?

### Equatable

Implementing this protocol allows one instance of an object to be compared for equality with another instance of an object of the same type. You've probably leveraged this protocol without realizing it as the `Equatable` protocol is what allows us to use `==` to check the equality of two objects.

Adding `Equatable` support to your object allows your object to gain access to many convenient Swift APIs automatically. Furthermore, since `Equatable` is the base protocol for `Hashable` and `Comparable`, by adding `Equatable` conformance, we can easily extend our implementation to support creating `Sets`, sorting elements of a collection, and much more.

Let's take a look at our `Money` struct:

```
struct Money {  
    let value: Int  
    let currencyCode: String  
}
```

Currently, there's no easy way for us to check if one `Money` object is equal to another `Money` object.

We can change that by conforming to `Equatable`:

```
struct Money: Equatable {  
    let value: Int  
    let currencyCode: String  
  
    static func == (lhs: Money, rhs: Money) -> Bool {  
        lhs.currencyCode == rhs.currencyCode && lhs.value == rhs.value  
    }  
}
```

Now, our adherence to the `Equatable` protocol and our subsequent custom implementation of the `==` operator allows us to easily compare any two `Money` objects for equality.

Technically, we did some extra work though...

Since `Int` and `String` types are `Equatable` themselves, Swift can automatically synthesize the `==` implementation for us.

So, it'd be sufficient to just write:

```
struct Money: Equatable {  
    let value: Int  
    let currencyCode: String  
}
```

### Hashable

When an object implements the `Hashable` protocol it introduces a `hashValue` property which is useful in determining the equality of two objects and allows that object to be used with a `Set` or `Dictionary`.

The `hashValue` is an `Integer` representation of the object that will always be the same for any two instances that compare equally. In simple terms, this means that if we have two instances of an object - A and B - then if `A == B` it must follow that `A.hashValue == B.hashValue`.

However, the reverse isn't necessarily true. Two instances can have the same `hashValue`, but may not necessarily equal one another. This is because the process that creates the `hashValue` can occasionally create situations where two different instances generate the same `hashValue`.

The `Hashable` protocol only guarantees that two instances that are already known to be equal will also have the same `hashValue`.

It's easy to add support for the `Hashable` protocol, but note that `Hashable` requires conformance to the `Equatable` protocol as well.

From Swift 4.1 onwards, we can have the compiler automatically synthesize conformance for us just like we did with `Equatable`. This functionality is only an option when the properties within the object conform to `Hashable`.

```
struct Money: Hashable {  
    let value: Int  
    let currencyCode: String  
}
```

Otherwise, we'll have to implement the `hashValue` generation ourselves. Swift 4.2 introduced a `Hasher` type that provides a randomly seeded universal hash function for us to use in our custom implementations:

```
struct Money: Hashable {
    let value: Int
    let currencyCode: String

    func hash(into hasher: inout Hasher) {
        hasher.combine(value)
        hasher.combine(currencyCode)
    }
}
```

Conforming to the `Hashable` protocol allows you to use this object in a `Set` or as a `Dictionary` key. Many types in the standard library conform to `Hashable`: `Strings`, `Integers`, floating-point numbers, `Booleans`, and even `Set` are hashable by default.

### Comparable

The `Comparable` protocol allows us to use our custom type with the `<`, `<=`, `>=`, and `>` operators.

The implementation of this protocol is quite clever. We only have to implement the less than operator - `<` -since the implementations of all of the other comparison operators can be inferred from `<` and our `Equatable` conformance.

This conformance allows us to use handy Swift methods like `sorted()`, `min()`, and `max()` on our objects in collections.

```
struct Money: Comparable {
    let value: Int
    let currencyCode: String

    static func < (lhs: Money, rhs: Money) -> Bool {
        lhs.value < rhs.value
    }
}
```

## What methods are required to display data in a UITableView?

Every iOS interview I've done has dealt with `UITableViews` in some capacity. Most often, it will involve hitting some API and showing the response in a `UITableView` like we did in Assessment #1.

As you prepare for your interviews, you should aim to be able to create a `UITableView` with custom `UITableViewCellCells` without referring to any documentation. Ideally, this set up process should become second nature to you.

Since this topic will be a constant in all of your interviews, knowing precisely what functions are necessary and what their respective inputs, outputs, and method signatures are is crucial.

Here are the only required `UITableViewDataSource` methods:

```
// Return the number of rows for the table.  
override func tableView(_ tableView: UITableView,  
                      numberOfRowsInSection section: Int) -> Int {  
    return 0  
}  
  
// Provide a cell object for each row.  
override func tableView(_ tableView: UITableView,  
                      cellForRowAt indexPath: IndexPath) -> UITableViewCell {  
    // Fetch a cell of the appropriate type.  
    let cell = tableView.dequeueReusableCell(  
       (withIdentifier: "CellIdentifier", for: indexPath  
    )  
  
    // Configure the cell's contents.  
    cell.textLabel!.text = "Cell text"  
  
    return cell  
}
```

## What does the CodingKey protocol allow you to do?

In the event that the keys in a JSON response do not match the variable names in your `Codable` model exactly, you can use the `CodingKey protocol` to bridge specific properties that differ only in naming.

Simply put, the `CodingKey protocol` provides you with more granular control of the `Codable protocol`'s serialization and deserialization behavior. We can utilize this `protocol` with the help of a nested `enum` defined within our `Codable` model.

Here's an example JSON:

```
{  
    "url": "https://dynaimage.cdn.cnn.com/cnn/best-cakes.jpg",  
    "img_caption": "Japanese Dessert",  
    "img_source": "CNN"  
}
```

We may want to pick more Swift-y names for our variables, so our corresponding `Codable` model may look like this:

```
struct Article: Codable {  
    let url: String  
    let caption: String  
    let source: String  
}
```

Since the property names and the JSON response's keys differ, `Codable`'s default deserialization behavior will fail here.

We'll need to implement the `CodingKey protocol`:

```
struct Article: Codable {  
    let url: String  
    let caption: String  
    let source: String  
  
    enum CodingKeys: String, CodingKey {  
        case url  
        case caption = "img_caption"  
        case source = "img_source"  
    }
}
```

```
    }  
}
```

Now, we are able to influence `Codable`'s deserialization behavior and the JSON response can be converted to an `Article` without issue.

What is the difference between the designated and convenience initializer?

Every class requires a designated initializer as it's responsible for initializing stored properties and calling the superclass's `init()`.

A convenience initializer can be thought of as a wrapper around the designated initializer. They allow you to create a simpler initialization option for your `class` that either provides defaults for certain initialization parameters or helps transform some input into the exact format the designated initializer needs.

In Swift, we can create a convenience initializer by placing the keyword `convenience` before the `init`. A `class` can have any number of convenience initializers. These initializers can even call other convenience initializers in turn, but eventually they'll need to call the designated initializer:

```
class Money: NSObject {
    let value: Int
    let currencyCode: String

    init(value: Int, currencyCode: String) {
        self.value = value
        self.currencyCode = currencyCode
        super.init()
    }

    convenience init?(value: String, currencyCode: String) {
        guard let numericValue = Int(value) else {
            return nil
        }

        self.init(value: numericValue, currencyCode: currencyCode)
    }
}
```

In this example, the designated initializer expects `value` to be an `Int`. So, we can create a convenience initializer that helps us handle scenarios when we might have a `String` as input instead.

Notice that the convenience initializer is eventually calling the designated initializer.

## What is your preferred way of creating views?

There's no right or wrong answer here as the question is inherently subjective, but it's a great opportunity for you to demonstrate that you understand the challenges and limitations with each approach (`.xib` vs `.storyboard` vs programmatically).

### Storyboards

Storyboards are a great way to quickly build out new designs and are useful in encapsulating a particular user flow. By keeping all of the views relevant to an experience in your app in the same storyboard, it makes it easier to get a high-level overview of the applications' functionality and intended user experience. Storyboards can also make navigating to and from other `UIViewController`s very easy via segues.

However, storyboards can be difficult to work with on a team as they're prone to merge conflicts, long loading times, and responsiveness often suffers as the storyboard increases in size.

### XIBs

While xibs and storyboards share a lot of similarities, they trade the storyboard's navigation behavior for increased reusability. Since a xib is specific to one view, there's no provision of establishing segues from one view to the next. You'll typically use a xib when you have a single custom component that you want to re-use in multiple locations throughout the app.

Both storyboards and xibs make design changes very cumbersome to implement. For example, if you wanted to change the application's default font, colors, icons, or some other application-wide change needs to be made, you'd have to go into each storyboard and xib and manually update each view.

There are ways to mitigate this, but the "source of truth" gets a bit lost. Since you'll often apply additional styling programmatically (like shadows and rounded corners), a developer now needs to check multiple locations to get a full picture of the view's complete implementation and expected appearance. Last but not least, searching for constraints, custom styling, subviews, images, fonts, etc. is much more difficult on storyboards and xibs than on a view defined programmatically.

### Programmatically

While this option can be tedious and is often initially slower than the other options, it allows for greater control, improved searchability, and more reuse.

Any application-wide UI change can be made easily, merge conflicts are easier to manage, there's a single source of truth, and views created programmatically can be more easily tested.

In practice, you'll likely find that there's rarely ever one right approach. Often, production projects will have a mix of all three methods depending on how much reuse and development speed influences the decision making.

What would happen if a struct had a reference type in it?

In Swift, value types can contain reference types and reference types can contain value types.

In this case, the result is simply the creation of a value type with a property that has reference semantics. In other words, the reference type behaves like it always does.

Any changes made to the reference type will also be reflected in the property within the value type.

In the following example, the `User` object is a reference type. You'll see that changes made to the reference type modify the property in the `struct`.

```
class User {
    var name: String
    var age: Int

    init(name: String, age: Int) {
        self.name = name
        self.age = age
    }
}

struct Article {
    var author: User
    var id: Int
}

let user = User(name: "Aryaman", age: 26)
let article = Article(author: user, id: 123)

// Output: Aryaman
print(article.author.name)

user.name = "Aryaman Sharda"

// Output: Aryaman Sharda
print(article.author.name)
```

What are the stages in a `UIViewController`'s view lifecycle?

Here are the different stages of a `UIViewController`'s lifecycle:

`loadView()`

This function is responsible for creating the view the `UIViewController` manages. If you want to create your view manually, you'll need to override this function.

However, if you're working with .storyboard or .xib files, you can ignore this method entirely.

`loadViewIfNeeded()`

Loads the `UIViewController`'s view if it has not yet been loaded.

`viewDidLoad()`

This method is called after the `UIViewController` has loaded its view hierarchy into memory.

This method is called regardless of whether the view hierarchy was loaded from a .xib file, .storyboard, or created programmatically in the `loadView()` method. In most cases, you'll override this method in order to perform additional setup and customization.

`viewWillAppear(_ animated: Bool)`

This method is called before the `UIViewController`'s view is about to be added to a view hierarchy and before any animations are configured for showing the view. You can override this method to perform custom tasks associated with displaying the view. For example, you might use this method to change the orientation or style of the status bar to coordinate with the orientation or style of the view being presented.

According to Apple's documentation, this method "notifies the `UIViewController` that its view is about to be added to a view hierarchy", however this method gets called every time the view is about to appear regardless of whether the view has previously been added to the hierarchy.

`viewWillLayoutSubviews()`

This method is called to notify the `UIViewController` that its view is about to layout its subviews. In other words, this method is called right before `layoutSubviews()` is executed.

For example, when a view's bounds change, this function will be called as the view needs to adjust the position of its subviews and the layout will need to be recalculated. Your view

controller can override this method to make changes before the view lays out its subviews. The default implementation of this method does nothing.

#### `viewDidLayoutSubviews()`

As you'd expect, this method is called to notify the view controller that `layoutSubviews()` has finished execution.

This method being called does not indicate that the individual layouts of the view's subviews have been adjusted as each subview is responsible for adjusting its own layout.

Your `UIViewController` can override this method to make changes after the view lays out its subviews. The default implementation of this method does nothing.

#### `viewDidAppear(_ animated: Bool)`

This function notifies the `UIViewController` that its view has been added to the view hierarchy and is now visible on the screen.

You can override this method to perform additional tasks associated with presenting the view. If you override this method, you must call `super` at some point in your implementation.

#### `viewWillDisappear(_ animated: Bool)`

This method is called when the `UIViewController` is going to be removed from the view hierarchy or will no longer be visible. This method is called before the view is actually removed and before any animations are configured.

#### `viewDidDisappear(_ animated: Bool)`

This method is called when the `UIViewController` is removed from the view hierarchy.

## How does code signing work?

Code signing is a process that helps you verify the authenticity of the app or software you're installing and ensures that it hasn't been tampered with since the developer released it. The code signing process consists of a series of smaller steps: requesting a certificate, receiving a certificate, generating a provisioning profile, and finally signing the application.

Let's start with requesting a certificate.

To request a certificate, we'll need to create a certificate signing request - a .csr file. When you first set up your Apple Developer account, you will need to use Keychain Access to request a development certificate from Apple - the certificate authority.

A certificate authority is the entity responsible for issuing digital certificates. Simply put, it's a trusted organization responsible for verifying the authenticity of software and other digital goods.

Since you have most likely already experienced this certificate request flow, I won't provide a tutorial for that process here. Instead, I'd like to discuss what's happening under the hood.

When you start the process of creating your certificate signing request, Keychain Access will generate a public and private key on your local machine. The public key is eventually attached to the .csr file, but the private key never leaves your machine. After this step, Keychain Access will prompt you to provide some basic metadata like name, country, email, etc. thereby completing the creation of the request.

From here, Apple examines the request's properties and metadata to verify who is requesting the certificate. Upon successful validation, Apple will send you back a certificate, which you should store in your machine's keychain. Certificates usually last for a year before expiring and can come in several varieties; iOS App Development, iOS Distribution, Mac App Distribution, Mac Installer Distribution, etc.

Code signing is driven by asymmetric cryptography, which is what enables communication with Apple and powers the signing process. Any explanation of code signing would be incomplete without an explanation of this encryption, so let's take a quick look at how it works.

To start with, both you and Apple have your own set of public and private keys. Now, these keys are cryptographically linked meaning you can use the private key to decrypt a message encrypted with the public key, but you can't go in the other direction.

After establishing a connection with Apple, we will exchange public keys. The public keys are intended to be shared, but the private keys need to be protected.

So, when you want to send a message to Apple, you encrypt the message with Apple's public key. Then, when Apple receives the message, they'll be able to decrypt the message using their private key.

This same process happens in reverse when Apple sends the certificate to us; Apple will encrypt the message using our public key and we'll use our private key to decrypt the message and retrieve the certificate. This process ensures that our full conversation with Apple from requesting to receiving the certificate is secure.

As a next step, the provisioning profile is created, consisting of a few key components:

- **Team ID:** A unique identifier for each development team and can be found in your Apple Developer account.
- **Bundle ID:** Every iOS app has a unique bundle identifier which allows it to be uniquely identified.
- **App ID:** The combination of the Team ID and the Bundle ID.
- **Device ID:** The list of all UDIDs (Unique Device Identifier) that your iOS application is authorized to run on. This is a 40 character alphanumeric identifier.
- **Entitlements:** This specifies the permissions and capabilities of the app along with which system resources the application has permission to access. For example, services like Push Notifications, Apple Pay, App Sandbox, etc.
- **Certificate:** The certificate we received from Apple in the previous step (iOS App Development, iOS Distribution, Mac App Distribution, Mac installer Distribution, etc.)

In summary, the provisioning profile is composed of the certificate that verifies the authenticity of the software, the App ID that uniquely identifies the application, and its permissions, entitlements, and the exact list of devices that can run the application. It essentially acts as the middle-man between the end devices and the developer account.

We're finally at the last step - code signing. This step involves downloading the provisioning profile from your developer account and embedding it into your application's bundle. Next, the bundle is signed with the certificate we created earlier and can now run on your device.

Here's what's happening under the hood:

1. The certificate referenced in your provisioning profile is compared against the available certificates in your machine's keychain.
2. If a valid certificate is found, it is used to sign the executable.
3. The UDID of the device you are attempting to run the executable on is compared against the UDIDs listed in the provisioning profile.
4. The Bundle ID and Entitlements are checked against their respective counterparts in the provisioning profile.
5. If everything goes well, the app is installed on the device.

## Swift Language Features

What is the difference between `==` and `===`?

In the example below, we have a simple `Engine` class that implements the `Equatable` protocol. This allows us to provide a custom implementation for the `==` operator.

We say that two `Engines` are equal if the horsepower is the same. So, as long as the `Int` values match, `==` will return `true`.

```
class Engine: Equatable {
    var horsepower: Int

    init(horsepower: Int) {
        self.horsepower = horsepower
    }

    static func == (lhs: Engine, rhs: Engine) -> Bool {
        lhs.horsepower == rhs.horsepower
    }
}

let engine1 == Engine(horsepower: 100)
let engine2 == Engine(horsepower: 100)
let engine3 == Engine(horsepower: 200)

engine1 == engine2 // true
engine2 == engine3 // false
```

With `==` we're asking if the objects on either side of the operator point to the same reference. In other words, do they point to the same place in memory?

In the following example, when we compare `engine1` to `engineCopy` (which is also referencing the same memory location), `==` returns `true`.

```
let engine1 = Engine(horsepower: 200)
let engine2 = Engine(horsepower: 200)
let engineCopy = engine1

engine1 === engineCopy // true
engine2 === engineCopy // false
```

However, in the second check, we can see that `engine2` and `engineCopy` are pointing to entirely different objects, so even though the `horsepower` is the same, `==` returns `false`.

## What's the difference between Self vs self?

In the context of protocols and extensions, `Self` refers to the type that conforms to the protocol whereas `self` refers to the value inside that type.

Consider this extension:

```
extension Int {
    // Self here refers to the conforming type (Int)
    func square() -> Self {
        // self refers to the value of the Int itself i.e. 2
        self * self
    }
}

// self in the code above would now equal 2
let width = 2

// Output: 4
print(width.square())
```

Additionally, you can use `Self` to limit the conformance of a protocol to only specific types:

```
protocol Squareable where Self: Numeric {
    func square() -> Self
}

extension Squareable {
    func square() -> Self {
        self * self
    }
}
```

Now, our `Squareable` protocol is only available to types that conform to `Numeric`.

## How would you limit a function or a class to a specific iOS version?

We can accomplish this by using availability attributes. This language feature allows us to add support for newer APIs, methods, or classes while still maintaining backwards compatibility.

```
if #available(iOS 15, *) {
    print("Hi! I can only run on iOS 15 and up.")
} else {
    print("I'll handle all other iOS versions.")
}

// The compiler will present an error if you try and use this
// class on any version < iOS 14.0
//
// The * serves as a wildcard and will allow this class to be available
// on all other platforms.
//
// We can just as easily check for macOS, watchOS, etc.
@available(iOS 14, macOS 10.10, *)
final class HelloWorld {

}

// We can also mark a function as unavailable in a similar manner
@available(*, unavailable)
```

## What does the final keyword do?

Imagine you are working on a SDK or a particularly sensitive piece of code - something that needs to be performant, secure, etc. You'd likely want to ensure that your code is used in exactly the way you intend it to be.

That's where the `final` keyword comes in. The `final` keyword prevents a `class` from being subclassed / inherited from and indicates to other developers that this `class` isn't designed to be subclassed.

So, in the SDK example, you would likely mark relevant classes as `final` to ensure your code is used only in the way you intended.

Marking properties and functions as `final` tells the Swift compiler that the method should be called directly (static dispatch) rather than looking up a function from a method table (dynamic dispatch).

This reduces function call overhead and provides a small boost in performance.

## What does the nil coalescing operator do?

We can use the nil coalescing operator - ?? - to provide a default value in an expression involving an `Optional`. If the `Optional` resolves to `nil`, our default value will be used instead.

```
var username: String?  
  
// Output: Hello, stranger!  
print("Hello, \(username ?? "stranger")!")  
  
username = "@aryamansharda"  
  
// Output: Hello, @aryamansharda!  
print("Hello, \(username ?? "stranger")!")
```

## What does defer do?

The `defer` keyword allows us to specify code that should be executed only when we are leaving the current function's scope.

It's commonly used for releasing a shared resource, closing a connection, or performing any last-minute cleanup.

In this case, we're using it to close the connection to a database:

```
class DatabaseManager {  
    func writeLineToDatabase(entry: String) {  
        let database = Database()  
  
        defer {  
            database.disconnect()  
        }  
  
        database.connect()  
  
        do {  
            try database.write(entry: "Hello world!")  
        } catch {  
            print("An error occurred!")  
        }  
    }  
}
```

You'll see that regardless of whether the:

```
try database.write(entry: "Hello world!")
```

call succeeds or fails, we will always close out the database connection without having to duplicate the `disconnect()` call. Lastly, we can use `defer` to write setup and cleanup code next to each other, even though they need to be executed at different times.

## What is optional chaining?

Optional chaining is a convenient way of unwrapping multiple **Optional** properties sequentially. If any of the **Optional** values in the expression resolve to **nil**, the entire expression will resolve to **nil**.

Consider the following expression involving multiple **Optional** properties:

```
user?.isAdmin?.isActive
```

If **user** or **isAdmin** or **isActive** is **nil**, the entire expression becomes **nil**. Otherwise, **isActive** will return the unwrapped value.

This is much more readable than using multiple **guard** and **if let** statements to break down a sequence of **Optional** values.

Can we use Swift's reserved keywords as variable or constant names?

Yes! This makes our code much easier to read and it's accomplished through the use of backticks.

If you want to use a reserved keyword, for example as a `case` in an `enum`, you can just add backticks around the reserved keyword:

```
enum MembershipType {
    case `default`
    case premium
    case trial
}

// Free to use MembershipType.default now
```

Otherwise, without the backticks, we'd have a compilation issue.

## What is the difference between try, try!, and try???

All of these options are different ways of calling a function that can throw an error.

`try!` is the most dangerous option of the bunch and should seldomly be used as it involves force unwrapping an `Optional` value.

It's effectively saying that while the function in question *could* throw an error, this will never happen. So, we want to proceed as if the called function will always return a value:

```
func fetchData() {  
    // try! states that even though this function may return an error,  
    // we know it won't happen so we can force unwrap this optional  
    let unwrappedData = try! thisFunctionCanThrow()  
  
    // The application will crash if unwrappedData is in fact nil  
}  
  
func thisFunctionCanThrow() throws -> [String] {  
    // Imagine this function can throw an error  
    return []  
}
```

However, `try!` is sometimes used if the error is so significant and unrecoverable that no valid user flow exists - similar to the use case of `fatalError()`.

`try?` can be used to ignore any errors from the throwing function. If an error is thrown, the expression will resolve to `nil`. As a result, we'll need to unwrap the returned value in order to use it.

This variation is often used when the error thrown isn't important enough to block or change the user's experience or it's permissible to have the function call fail silently:

```
func fetchData() {  
    // The result from `thisFunctionCanThrow` will either be nil  
    // or a [String] which we'll need to unwrap to access  
    if let unwrappedData = try? thisFunctionCanThrow() {  
        print("Successfully retrieved data: \(unwrappedData)")  
    }  
}
```

```
func thisFunctionCanThrow() throws -> [String] {  
    // Imagine this function can throw an error  
    return []  
}
```

**try** is the safest option and forces us to explicitly catch and handle any issues that occur:

```
func fetchData() {  
    // This is a safe approach and allows us to decide how we  
    // handle the error  
    do {  
        let data = try thisFunctionCanThrow()  
        print("Successfully retrieved data: \(data)")  
    } catch {  
        print("An error occurred: \(error)")  
    }  
}  
  
func thisFunctionCanThrow() throws -> [String] {  
    // Imagine this function can throw an error  
    return []  
}
```

## What is an inout parameter?

Whenever you pass a value type into a function, only a copy of the value is passed along. As a result, even if you attempt to change the value of that parameter inside the function, the variable at the calling site will still maintain its original value.

```
var currentAge = 26

func updateAge(passedInAge: Int) {
    var passedInAge = passedInAge
    passedInAge = 42

    print(passedInAge) // 42
    print(currentAge) // 26
}

updateAge(passedInAge: currentAge)
```

If we want to change the value of the parameter itself instead of just working with a copy of the data, we'll need to add the `inout` keyword. This will allow us to make changes directly to the variable that was passed in even if it's a value type.

We'll need to use the & symbol when providing an `inout` parameter:

```
var currentAge = 26

func updateAgeWithInout(passedInAge: inout Int) {
    passedInAge = 42
    print(passedInAge) // 42
    print(currentAge) // 42
}

// currentAge is 26 before the call and 42 after
updateAgeWithInout(passedInAge: &currentAge)
```

The `inout` keyword is used very often in Swift and enables syntactic sugar like the `+=` operator which modifies the value of the variable on the left-hand side of the operator in place.

How can we limit a protocol conformance to a specific class or type?

There may be situations where we want to limit conformance of a **protocol** to a certain type.

For example, conformance to **Numeric** is restricted to types that also conform to **AdditiveArithmetic** and **ExpressibleByIntegerLiteral**:

```
public protocol Numeric : AdditiveArithmetic, ExpressibleByIntegerLiteral
```

We can accomplish this by using the **where** keyword or : when declaring the **protocol**:

```
protocol TestViewController: UIViewController { }
protocol TestViewControllerAlternative where Self: UIViewController { }
```

When we restrict a **protocol** to only be used by a specific type, Xcode will automatically limit its auto-complete suggestions to only suggest the **protocol** when it's applicable.

This syntax can also be applied to a **protocol** extension which allows you to provide default implementations on a case-by-case basis.

```
protocol Animal {
    func speak()
}

class Dog: Animal {}
class Cat: Animal {}

extension Animal where Self: Dog {
    func speak() {
        print("Woof!")
    }
}

extension Animal where Self: Cat {
    func speak() {
        print("Meow!")
    }
}
```

What if we wanted our `protocol` to only be supported by types that also implement some other `protocol`?

We can take care of that too:

```
protocol SecureHashable: Hashable {  
    var secureHash: String { get }  
}
```

Does Swift support implicit casting between data types?

Swift **does not** support implicit casting.

Take the following expression involving **Doubles**, **FLOATS**, and **Integers**:

```
let seconds: Double = 60
let minutes = 60 // minutes is inferred to be an Int
let hours: Float = 24.0
let daysInYear: Int = 365

// Fails with "Cannot convert value of type Float / Double to
// expected argument type Int"
let secondsInYear = seconds * minutes * hours * daysInYear

// You can see that we've had to explicitly cast all of the
// non-Integer types to Int
let secondsInYear = Int(seconds) * minutes * Int(hours) * daysInYear
print(secondsInYear) //31536000
```

When you have an expression with multiple types, Swift will not automatically convert them to a common shared type. Instead, Swift forces you to be explicit about how you want to deal with this mix of types.

Since the output can only be of one type, Swift leaves it up to the programmer to define what that should be.

## What does the Caselterable protocol do?

As the name suggests, `CaseIterable` is a protocol that provides a handy way of iterating through all of the individual cases in an `enum`.

When using an `enum` that conforms to `CaseIterable`, you can access a collection of all of the `enum`'s cases by using the `allCases` property:

```
enum CompassDirection: CaseIterable {
    case north, south, east, west
}

// "There are 4 directions."
print("There are \(CompassDirection.allCases.count) directions.")

let caseList = CompassDirection.allCases.map({ "\($0)" })
    .joined(separator: ", ")
// "north, south, east, west"
print(caseList)
```

`allCases` provides the cases in the order of their declaration.

If your `enum` does not contain any associated values or availability attributes, Swift will automatically synthesize conformance to the `CaseIterable` protocol for you.

Can you explain what the `@objc` keyword does?

If you're applying to an older company, they'll likely have a substantial part of their codebase still in Objective-C. So, you'll need to be comfortable with using both Objective-C and Swift in the same project.

This attribute is used to make your Swift code accessible to the Objective-C runtime.

Anytime you have a Swift class, property, or protocol that you want to access in Objective-C code, you'll need to prefix it with this keyword.

```
// The ViewController is accessible in an Objective-C environment
@objc class ViewController: UIViewController {

    // Visible in Objective-C
    @objc var username: String!

    // Not visible in Objective-C
    private var password: String!

    override func viewDidLoad() {
        super.viewDidLoad()
    }
}
```

`@objc`, when added to a `class` declaration, only exposes the `public init`. You will have to add it manually to all other properties and methods you want to expose.

If you want to expose all public properties and methods to Objective-C, you can use `@objcMembers` instead.

## What are compilation conditions?

Compilation conditions or compiler directives are keywords we can use to influence the compilation process itself. We can use system compiler directives like `DEBUG` or easily define our own custom flags in our project's build settings.

Here's an example of compilation conditions in action:

```
#if DEBUG
print("...")

#if targetEnvironment(simulator)
return ViewControllerA()
#else
return ViewControllerB()
#endif

#if BETA_TARGET
let image = UIImageView(image: UIImage(named: "BetaAppIcon"))
#else
let image = UIImageView(image: UIImage(named: "NormalAppIcon"))
#endif
```

Since these compiler directives are considered only at compile time, they allow us to **literally** exclude the other section of code from the compiled executable. This is useful for ensuring that any new feature you're not quite ready to release (i.e. feature flagging) or functionality that should be specific to a platform or software version isn't accidentally included in your final executable.

With this approach, you can instruct the compiler to include only select parts of your codebase in the final executable while still allowing all of the code to exist in the same project.

## What are the differences between Swift's access control modifiers?

Access controls allow us to restrict parts of our code from other source code files and modules.

This enables us to abstract away certain implementation details and instead share a preferred interface through which our code should be accessed and used.

Swift provides five different access level modifiers that can be used on classes, structs, enums, properties, methods, and initializers.

Going from the least to the most restrictive:

**open**: classes and methods marked as **open** can be subclassed and overridden outside of their defining module.

**public**: provides read / write access to an entity to all other parts of the code regardless of the defining module.

**internal**: is the default access level in Swift. Properties declared as **internal** are accessible in the scope of their defining module, but are inaccessible by parts of the code belonging to other modules.

**fileprivate**: entities marked as **fileprivate** are visible and accessible from Swift code declared in the same source file.

**private**: is the most restrictive access level and ensures that a property is only accessible inside their defining type. This means that properties and methods declared as **private** are only accessible within the class or structure in which they are defined.

## What does the typealias keyword do?

The `typealias` keyword allows us to define our own keyword to represent an existing data type in our application. This does not create a new type - it simply provides a new name for an existing type.

An excellent example is Swift's `Codable` protocol which is implemented as a `typealias`.

```
public typealias Codable = Decodable & Encodable
```

By the way, did you notice that you can combine protocols with `&` in Swift?

Now, whenever the compiler sees `Codable`, it will replace it with `Decodable & Encodable` and continue compilation.

Imagine we were trying to represent time on a clock. We could use `typealias` to make the tuple representing our clock easier to work with:

```
typealias ClockTime = (hours: Int, min: Int)

func drawHands(clockTime: ClockTime) {
    print(clockTime.hours) // 3
    print(clockTime.min) // 30
}

ClockTime(3, 30)
```

If you wish to make the `typealias` accessible throughout your codebase, declare it outside of any class or enclosing type. Otherwise, the `typealias` will be limited in scope as any other variable would be.

```
// Accessible across the codebase
typealias ClockTime = (hours: Int, min: Int)

class HelloWorld {
    // Only available within this class
    typealias Greeting = String

    func sayGreeting(greeting: Greeting) {}
}
```

It's easy to overuse `typealias` and thereby limit the discoverability of your code, so it's important to exercise a little restraint.

## What does copy on write mean?

We know that when we're working with types that use pass-by-value semantics, we're passing around a copy of the value instead of a direct memory reference.

Now, imagine that the value type we're working with is storing a lot of data - for example an `[UIImage]`.

If we aren't going to modify this array, it's pointless to duplicate it every time we pass it to a function or assign it to a new variable as this would introduce a lot of unnecessary overhead.

That's why Swift employs a resource management technique called "copy on write". This technique allows us to efficiently implement a "copy" operation on a modifiable resource.

If the resource in question is never modified, then there's no need to create a duplicate of it. Instead, we can provide shared access to this resource until such time an operation attempts to modify it. Then, and only then, will Swift duplicate the value.

This approach lets the system significantly reduce the resource consumption of pass-by-value operations in exchange for adding a small overhead to resource-modifying operations.

This functionality is available by default in arrays and dictionaries, but you'll have to explicitly add it to any new value types you introduce.

## How are Optionals implemented?

Understanding how an `Optional` is implemented relies on a solid understanding of value types and `enums`.

Here's a simplified version of Swift's `Optional` implementation:

```
public enum Optional<Wrapped>: ExpressibleByNilLiteral {  
  
    case none  
    case some(Wrapped)  
  
    public init(_ some: Wrapped) {  
        self = .some(some)  
    }  
  
    public init(nilLiteral: ()) {  
        self = .none  
    }  
}
```

Since Swift is open-source, you can see the [full `Optional` implementation here](#).

You can see that in the case of `.some` there's an associated value; this is the non-`nil` case of using an `Optional` variable.

Hopefully, you can see how simple language features like generics, enums, associated values, and value semantics enable us to build powerful and expressive language features like `Optionals` from just a few basic building blocks.

Finally, knowing that an `Optional` is simply an `enum` "under the hood" opens the door for us to create our own extensions on `Optionals` and add additional behavior and convenience methods.

## What are the higher order functions in Swift?

In mathematics and computer science, a higher-order function is a function that takes one or more functions as arguments or returns a function.

Functional programming is a programming paradigm in which we try to leverage these higher-order functions.

Let's say you wanted to create a function to double all of the numbers in an array. You might write something like this:

```
var input = [1,2,3,4,5]

for i in 0..
```

However, we've immediately run into a problem. What happens if we want to access the original values in `input`? We can no longer retrieve those values.

By changing `input`, we've used imperative programming which is a paradigm in which executed statements change the state of the program. In contrast, functional programming ensures that no changes are made to the existing state of the application and no side-effects are introduced.

You'll be able to find a more rigorous mathematical definition of functional programming elsewhere, but simply speaking the goal is:

- Avoid mutability wherever possible.
- Use functions as the building blocks of functionality. In other words, try to compose functions wherever possible.
- Create pure functions (a pure function is a function that will always produce the same result for the same input regardless of when and where it is being called from).

Simply put, in imperative programming, changing the variable's state and introducing side effects is permissible, but in functional programming it is not.

You'll notice that in all of the following examples, the `input` variable's values are never changed which allows us to satisfy the immutability requirement. Instead, all of these function calls return a completely new value.

```
.map {}
// map: Applies a function to every element in the input and returns
// a new output array.
let input = [1, 2, 3, 4, 5, 6]

// We'll take every number in the input, double it, and return it in a new
// array.
//
// Notice that the input array is never modified.
//
// Output: [2, 4, 6, 8, 10, 12]
// $0 refers to the current element in the input array `map` is operating on.
let mapOutput = input.map { $0 * 2 }

.compactMap {}
// compactMap: Works the same way as map does, but it removes nil values.
let input = ["1", "2", "3", "4.04", "aryamansharda"]

// We'll try and convert each String in `input` into a Double.
//
// Notice that the input array is never modified and the values that resolve
// to nil are not included in the output.
//
// compactMap is often used to convert one type to another.
// Output: [1.0, 2.0, 3.0, 4.04]
let compactMapOutput = input.compactMap { Double($0) }

.flatMap {}
// flatMap: Use this method to receive a flattened collection from an
// input that may have several nested structures.
let input = [[1], [2, 2], [3, 3, 3], [4, 4, 4, 4]]

// This will flatten our array of arrays into just a single array.
// Output: [1, 2, 2, 3, 3, 3, 4, 4, 4, 4]
let flatMapOutput = input.flatMap { $0 }

.reduce {}
// reduce: Allows you to produce a single value from the elements in
// a sequence.
```

```

let input = [1, 2, 3, 4]

// This will add up all of the numbers in the sequence.
// `sum` will be 10
let sum = input.reduce(0, { x, y in
    x + y
})

// You can also write this more simply as:
let sum = input.reduce(0, +)
// `sum` will be 10

.filter {}
// filter: Returns an array containing only the elements of the sequence
// that satisfies some constraint(s) while preserving order.
let input = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

// This will only return the even numbers.
// Output: [2, 4, 6, 8, 10]
let filterOutput = input.filter { $0 % 2 == 0 }

.forEach {}
// forEach: Calls the given closure on each element in the sequence in the
// same order as a for-loop.
let input = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

// This will print out the numbers in `input` just like a traditional
// for-loop.
input.forEach {
    print($0)
}

```

When you use `forEach` it is guaranteed to go through all items in sequence order, but `map` is free to process items in any order.

```

.sorted {}
// sorted: Returns the elements of the sequence, sorted using the
// custom sorting logic you specify.
let input = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

// This will create an array with the numbers sorted in descending order.
// Output: [10, 9, 8, 7, 6, 5, 4, 3, 2, 1]

```

```
let sortedInput = input.sorted(by: { $0 > $1 })
```

These functions can be applied to objects of any type - even custom ones. In addition, combining these calls can deliver impressive functionality with very little code:

```
// We'll create an array of Cars and specify some basic properties.
struct Car {
    let name: String
    let horsepower: Int
    let price: Int
}

var cars = [Car?]()
cars.append(Car(name: "Porsche 718", horsepower: 300, price: 60500))
cars.append(Car(name: "Porsche 911", horsepower: 379, price: 101200))
cars.append(nil)
cars.append(Car(name: "Porsche Taycan", horsepower: 402, price: 79900))
cars.append(Car(name: "Porsche Panamera", horsepower: 325, price: 87200))
cars.append(nil)
cars.append(nil)
cars.append(Car(name: "Porsche Macan", horsepower: 248, price: 52100))
cars.append(Car(name: "Porsche Cayenne", horsepower: 335, price: 67500))

// Let's return valid cars (not nil) that have a horsepower greater than 300
// and are sorted by descending price.
cars.compactMap { $0 }
    .filter { $0.horsepower > 300}
    .sorted { $0.price > $1.price }
    .forEach { print($0) }

// Output:
Car(name: "Porsche 911", horsepower: 379, price: 101200)
Car(name: "Porsche Panamera", horsepower: 325, price: 87200)
Car(name: "Porsche Taycan", horsepower: 402, price: 79900)
Car(name: "Porsche Cayenne", horsepower: 335, price: 67500)
```

Functional programming in iOS is becoming more and more mainstream and is critical to the SwiftUI and Combine implementations. Levering these language features and programming paradigms correctly allows you to write optimized, thread-safe, easily testable, and readable code.

## What is the difference between Any and AnyObject?

When we need to work with non-specific types, we have two options to pick from: `Any` and `AnyObject`.

`Any` is used to refer to any instance of a `class`, `struct`, function, `enum`, etc. This is particularly useful when you're working with a variety of data:

```
let items: [Any] = [1, UIColor.red, "Blue", Toggle()]
```

`AnyObject` is more restrictive as it refers to any instance of a `class` type. You'll use this when you only want to work with reference types or when you want to restrict a `protocol` to only be used with a `class` type:

```
protocol ClassOnlyProtocol: AnyObject {...}
```

In contrast, `Any` can be used with both value and reference types.

It's preferable to be as specific as possible about the type you're using, so I'd encourage you to only use `Any` or `AnyObject` when you specifically need the behavior they provide.

## What is a raw value in an enum?

Enumerations provide a common type for a group of related values thereby enabling you to work with those values type-safely within your code.

While we can use associated values to relate some required data to an `enum` case, we can use an `enum`'s `rawValue` property in instances where a hard-coded default value will suffice.

Here, `ASCIIControlCharacter` specifies that the type of its `rawValue` is going to be a `Character`.

This, in turn, requires us to provide a hard-coded `Character` for every case in our `enum`. So, we can easily link an ASCII character to its respective `enum` case.

```
enum ASCIIControlCharacter: Character {  
    case tab = "\t"  
    case lineFeed = "\n"  
    case carriageReturn = "\r"  
}
```

By default, the specified `rawValue` type can be a `String`, `Character`, `Integer`, or floating-point number, but you can add support for a custom type by adding conformance to `RawRepresentable`.

The hard-coded value you assign must be unique within the `enum`'s declaration.

If our enumeration's `rawValue` is an `Integer` or a `String`, Swift will automatically assign default values for us. However, we're still able to override the default values if need be.

When it comes to `Integers`, the implicit value for each case is one more than the last. If no value is set for the first case, the `rawValue` will start counting up from 0.

Consider this `enum` that specifies the planets and their respective position from the Sun:

```
enum Planet: Int {  
    case mercury = 1, venus, earth, mars, jupiter, saturn, uranus, neptune  
}
```

Since we've specified an explicit value of 1 for `mercury` (instead of the Swift default of 0), `Planet.mercury`'s `rawValue` will be 2, `Planet.earth` will be 3, and so on.

In the case of a `String`, Swift will set the default `rawValue` to match the `enum` case's name. In the following example, we don't need to explicitly provide any `rawValues` as the Swift defaults will work for us.

```
enum CompassPoint: String {  
    case north, south, east, west  
}
```

`CompassPoint.north.rawValue` will be "north", `CompassPoint.south.rawValue` will be "south", and so on.

If we define an enumeration with `rawValue` support, Swift will automatically add an initializer that allows us to go from the `rawValue` to the corresponding type.

For example, we could create a reference to `Planet.uranus` by simply writing:

```
let possiblePlanet = Planet(rawValue: 7)  
// possiblePlanet is of type Planet? and equals Planet.uranus
```

Since not all `rawValues` (ex. `Planet(rawValue:20)`) can be mapped to a corresponding `enum` case, using this initializer will return an `Optional`.

That's why `possiblePlanet` is a `Planet?`

What are our options for unwrapping optionals in Swift?

We have seven options for unwrapping **Optionals** in Swift with varying levels of safety:

```
var username: String?  
var user: User?  
  
// Forced unwrapping (unsafe)  
let forcedUnwrapping: String = username!  
  
// Implicitly unwrapped (often unsafe)  
// Used when a variable will start off as nil, but will have  
// a value by the time you use it.  
@IBOutlet var titleLabel: UILabel!  
  
// Optional chaining (safe)  
print(user?.emailAddress)  
  
// Optional binding (safe)  
if let value = username {  
  
}  
  
// Nil coalescing operator (safe)  
let value = username ?? "unknown"  
  
// Guard (safe)  
guard let username = username else {  
    return  
}  
  
// Optional (safe)  
if case let value? = username {  
    print(value)  
}
```

## What is an anonymous function?

An anonymous function is a function definition that isn't bound to an identifier. For example, most closures are considered anonymous functions. They can help make code more readable by allowing you to define all of the relevant logic in one place.

You will have almost certainly used anonymous functions in your own Swift code. Notice how in the following examples, functions like `{ self.view.backgroundColor = .orange }` and `{ $0 * 2 }` are defined without explicit function names attached to them.

```
func performAnimation() {
    self.view.backgroundColor = .orange
}

// Without anonymous function
UIView.animate(withDuration: 1.0, animations: performAnimation)

// With anonymous function
UIView.animate(withDuration: 1.0) {
    self.view.backgroundColor = .orange
}

// Without anonymous function
func doubleNumbers(num: Int) -> Int {
    return num * 2
}

let input = [1,2,3,4,5]

// Without anonymous function
let result = input.map(doubleNumbers(num:))

// With anonymous function
let resultAnonymous = input.map { $0 * 2 }
```

What is the difference between is, as, as?, and as! ?

Typecasting is a method of changing an entity from one data type to another.

These keywords are used to support typecasting in Swift and allow us to check the type of an instance or to treat an instance as one of the classes in its class hierarchy. This definition will hopefully make much more sense when we look at some examples.

It may be useful to understand the following terminology:

**Upcasting:** You cast an instance from a subclass to a superclass

**Downcasting:** You cast an instance from a superclass to a subclass

Casting doesn't actually modify the instance or change its values. The underlying instance remains the same; it's simply treated and accessed as an instance of the type to which it has been cast.

Here's the class hierarchy we'll use in the following examples:

```
protocol Animal {}
class Mammal: Animal {}

class Dog: Mammal {}
class Cat: Mammal {}
```

**is** (typecheck operator)

Use the typecheck operator, **is**, to check whether an instance is of a certain subclass type:

```
let dog = Dog()

// Output: true
print(dog is Animal)

// Output: true
print(dog is Mammal)

// Output: false
print(dog is Cat)
```

It's important to recognize that this keyword only returns a boolean. It does not perform any conversion - it simply checks to see if a potential type conversion *could* occur.

This keyword isn't very popular in Swift as you can always write an equivalent expression using an `if let` and `as?` instead which would have the added benefit of actually performing the conversion for you.

We'll see an example of this shortly.

### `as` (upcasting operator)

The `as` operator allows us to upcast from a subclass to superclass (i.e. `Dog` to `Animal`).

**The compiler must be able to guarantee the validity of the cast when we use this operator.**

So, we'll typically use it for conversions we know the compiler will be able to verify like `String` to `NSString`, `NSDate` to `Date`, or casting an object back to its parent class type.

```
let animal: [Animal] = [Dog() as Animal, Cat() as Animal, Mammal() as Animal]

// Output: [ExampleApp.Dog, ExampleApp.Cat, ExampleApp.Mammal]
print(animal)
```

### `as?` (conditional cast operator)

Similar to the `as` operator, `as?` also attempts to convert a class's type, but will return `nil` if the conversion fails.

Use the `as?` operator when you aren't sure if the casting operation will succeed. In the example below, the attempt to downcast `mammal` to `Dog` succeeds, but attempting to cast `mammal` to `Cat` evaluates to `nil`.

```
let mammal: Mammal = Dog()

if let dog = mammal as? Dog {
    // Valid
    print(dog)
}

// This expression will evaluate to nil
if let cat = mammal as? Cat {
```

```
    print(cat)
} else {
    print("Downcasting failed!")
}
```

**as!** (forced casting keyword)

This operator is known as the force downcasting operator and, like all other force unwrapping, will trigger a runtime error if the downcast conversion fails.

Make sure you only use this when you know the downcast will succeed!

```
let mammal: Mammal = Dog()

// Downcast succeeds
let dog = mammal as! Dog

// Triggers runtime error
let cat = mammal as! Cat
```

What are some of the main advantages of Swift over Objective-C?

### **Swift is easier to read**

Objective-C is constrained by C conventions, and arguably cannot be updated unless C itself is updated. By doing away with many legacy conventions like semicolons and parentheses for conditional expressions, Swift is simpler, easier to read, and more expressive.

### **Swift is safer**

Although a message sent to a nil object is perfectly acceptable in Objective-C and is treated as a no-op, Swift's explicitness about dealing with potentially nil values (Optionals) allows programmers to write cleaner, more transparent, and safer code. Finally, Swift's strongly typed nature is another point in its favor.

### **Swift is less verbose**

I don't think I have to explain this one. If you've used Objective-C, you'll know what I mean (no need for header files, automatically generated memberwise initializers for structs, syntax is simpler, etc).

### **Swift is faster**

Swift's performance has been greatly improved by dropping legacy C conventions. With Apple's continued support, this will likely only improve over time.

### **Swift has fewer namespace collisions**

Since Objective-C does not have formal support for namespaces, it's common to use a prefix for all class names to avoid conflict with external dependencies and modules. However, Swift provides implicit namespacing which sidesteps this problem without the need for prefixing class names.

### **Swift has greater ARC support**

With Swift, Automatic Reference Counting (ARC) is supported across all procedural and object-oriented code paths. Objective-C, on the other hand, only supports ARC within its object-oriented code and Cocoa APIs; it is not yet available for other APIs, like Core Graphics, and procedural C code.

### **Swift goes beyond iOS**

Swift is truly cross-platform - not only can you use Swift to develop applications for iOS, iPadOS, tvOS, and macOS, but you can also use Swift for server-side development too (see [Vapor](#)).

## How do we provide default implementations for protocol methods?

By leveraging Swift extensions, we can provide a default implementation for methods and properties declared in a **protocol**.

This helps reduce boilerplate and duplicated code in classes that implement the **protocol** while still allowing them to easily override the default implementation.

```
protocol Animal {
    func makeNoise()
}

extension Animal {
    func makeNoise() {
        print("Bark!")
    }
}

struct Dog: Animal {}
struct Cat: Animal {
    func makeNoise() {
        print("Meow!")
    }
}

let sparky = Dog()
sparky.makeNoise() // Bark!

let whiskers = Cat()
whiskers.makeNoise() // Meow!
```

As you can see, we've provided the default implementation for `makeNoise()` in an extension.

`Dog` is using the default implementation while `Cat` is free to provide a more specific implementation.

This same approach allows us to make certain functions in our **protocol** optional. Since we have provided a default implementation in our extension, any entity that implements the **protocol** is no longer required to implement it.

```
protocol MyProtocol {
    func doSomething()
}

extension MyProtocol {
    func doSomething() {
        /* Return a default value or just leave empty */
    }
}

struct MyStruct: MyProtocol {
    /* No compile error */
}
```

Alternatively, you can use `@objc optional` to make functions within your `protocol` optional. This would, however, restrict your `protocol` to only be implemented by `class` type objects, which would prevent your `protocol` from being used by `structs`, `enums`, etc. You'd also need to explicitly check if that optional method is implemented before you call it.

What does it mean to be a first class function or type?

A first-class citizen is an entity that can be passed as an argument, returned from a function, modified, or assigned to a variable. So, a programming language is said to have first-class functions if it treats functions as first-class citizens.

In simple terms, this means the language supports passing functions as arguments to other functions, returning functions from functions, and assigning them to variables or storing them in data structures.

As we'll see in the following examples, Swift treats functions as first-class citizens.

### Storing Functions In Variables

We can easily create a function (or in this case a closure definition) and assign it to a variable:

```
class FirstClassCitizens {
    var doubleInput:((Int) -> Int)?
    
    func setup() {
        doubleInput = { value in
            return value * 2
        }
    }
}
```

### Passing Functions As Arguments

We can also pass functions as arguments to other functions.

For example, whenever we specify an animation in Swift, the second parameter is actually accepting a function.

```
class FirstClassCitizens {
    func setup() {
        UIView.animate(withDuration: 1.0) {
            // This is a function passed as a parameter
        }

        UIView.animate(withDuration: 1.0, animations: {
            // Same as above but without trailing closure syntax
        })
    }
}
```

```
    }
}
```

## Returning Functions From Functions

We can also return a function from a function as well:

```
class FirstClassCitizens {
    func sayHello(name: String) -> (() -> String) {
        return {
            "Hello, \(name)"
        }
    }
}

let returnedFunction = FirstClassCitizens().sayHello(name: "Aryaman")
print(returnedFunction()) // Hello, Aryaman
```

Do all elements in a tuple need to be the same type?

Tuples are a very convenient way to group elements together without having to create a custom object or **struct** to encapsulate them.

To create a tuple in Swift, simply specify a comma separated list of values within a set of parentheses like this:

```
let tuple = (1, 2, 3, 123.0, "Hello, world!")
print(tuple.4) // Hello, world!
```

As you can see, a tuple doesn't have to be a homogenous set of types. It can easily be a mix of different types, but it's up to you to keep track of what data type exists at each position and interact with it accordingly.

What is protocol composition? How does it relate to Codable?

The `Codable` protocol is a prime example of Swift's protocol composition feature which allows you to easily combine existing protocols together using the `&` operator.

For example, the `Codable` protocol is actually the combination of the `Encodable` and `Decodable` protocols.

```
typealias Codable = Decodable & Encodable
```

`Decodable` allows a type to decode itself from an external representation and `Encodable` allows a type to encode itself as an external representation.

When combined together like this, the `Codable` protocol ensures that whatever object implements this protocol can both convert and be converted from some external representation.

In practice, this typically means converting a JSON response to a domain model object and vice-versa.

## What does the mutating keyword do?

In Swift, **structs** are value types which means the properties contained within are immutable by default. So, if we want to be able to modify the values within a **struct**, we'll need to use the **mutating** keyword. This keyword only applies to **value types** as reference types are not immutable in this way.

Whenever we call a function that uses this keyword and modifies the **struct**'s properties, Swift will generate a new **struct** in-place with the modifications applied and will overwrite our original **struct**.

```
struct User {  
    var firstName = "Aryaman"  
    var lastName = "Sharda"  
  
    func makeLowercase() {  
        // The following lines cause a compilation error:  
        // Cannot assign to property: 'self' is immutable  
        firstName = firstName.lowercased()  
        lastName = lastName.lowercased()  
    }  
}
```

Let's add the **mutating** keyword:

```
struct User {  
    var firstName = "Aryaman"  
    var lastName = "Sharda"  
  
    mutating func makeLowercase() {  
        firstName = firstName.lowercased()  
        lastName = lastName.lowercased()  
    }  
}
```

When working with **mutating** functions, we'll need to declare the **struct** as a variable since we're making changes to the **struct**'s properties:

```
let user = User()  
  
// Compilation Error!  
// Cannot use mutating member on immutable value: 'user' is  
// a 'let' constant  
user.makeLowercase()
```

When we make it a variable, we have no such issue:

```
// No error  
var user = User()  
user.makeLowercase()
```

## How do you use the Result type?

The `Result` type is a convenient way for us to handle both the success and failure cases of an operation while maintaining code readability.

Under the hood, the `Result` type, is an `enum` with two cases:

```
enum Result<Success, Failure> where Failure : Error {  
    /// A success, storing a `Success` value.  
    case success(Success)  
    /// A failure, storing a `Failure` value.  
    case failure(Failure)  
}
```

The success case accepts any generic `Success` type (including `Void`) and failure takes the generic `Failure` type as its associated value.

Note: Whatever you use for the `Failure`'s associated value *must* implement the `Error` protocol.

In the example below, you'll see that we're creating a `PrimeNumberError` enum with multiple cases. This approach allows us to have much greater specificity with our error messaging and has the added benefit of forcing us to handle each error case explicitly.

```
enum PrimeNumberError: Error {  
    case zero  
    case negative  
    case tooBig  
}  
  
func isPrimeNumber(num: Int) -> Result<Bool, PrimeNumberError> {  
    guard num < 0 else {  
        return .failure(.negative)  
    }  
  
    guard num > 0 else {  
        return .failure(.zero)  
    }  
  
    guard num < 1000 else {  
        return .failure(.tooBig)  
    }  
}
```

```
    return .success(primeNumberChecker(num: num))
}

switch isPrimeNumber(num: 23) {
case .success(let isPrime):
    print("The number \(isPrime ? "is" : "is not") prime")
case .failure(.tooBig):
    print("The number is too big for this function.")
case .failure(.negative):
    print("A prime number can't be negative.")
case .failure(.zero):
    print("A prime number has to be greater than 1.")
}
```

**Result** is available in Swift 5.0+.

## Is Swift a statically-typed language?

Yes, Swift is a statically-type language - a language where variable types are known at compile time.

In order for a Swift program to compile successfully, the compiler must have all of the necessary information about the types of every class, function, and property. To make things easier for the programmer and to help reduce the verbosity of our code, Swift supports type inference. This enables a compiler to deduce the type of a particular expression automatically when it compiles your code, simply by examining the values you provide.

That's why this code is totally valid even though we're not explicitly specifying the type:

```
var isApproved = true
```

In contrast to a statically typed language, we have dynamically typed languages. In a language of this variety, the types aren't determined at compile time, but are deferred until the actual point of sending a message to that object. In other words, the type is only known at runtime.

In Python, a dynamically typed language, the following code will compile, but will throw a runtime error due to the mismatched types:

```
def willThrowRuntimeError():
    s = 'acetheiosinterview' + 1
```

## What is an associated value?

An associated value is the term used to describe the value accompanying a `case` in a Swift `enum`. Associated values allow us to present more nuanced data by adding contextual information to our `cases`.

```
enum Distance {  
    case km(Int)  
    case meters(Int)  
    case miles(value: Int)  
}  
  
Distance.miles(value: 20)
```

With Swift, names can be specified for associated values in order to make their use more understandable. Additionally, each `case` can be associated with values of any type and number.

```
enum Action {  
    case tackle  
    case random  
    case kick(power: Int, speed: Float)  
    case jump(height: Int)  
    case shootLasers(useBothLasers: Bool)  
}
```

## What does a deinitializer do?

A deinitializer is a function that is called right before a `class` is deallocated. Deinitializers are only available on `class` types and each `class` can only have one deinitializer. This function does not accept any parameters.

You create a deinitializer using the `deinit` keyword:

```
deinit {
    // Perform the deinitialization
}
```

Although Swift automatically deallocates instances when they are no longer needed (the retain count becomes 0), deinitializers allow you to perform additional cleanup before your instance is deallocated.

For example, you may want to invalidate timers, terminate a socket connection, or close out a connection to a database:

```
deinit {
    database.closeConnection()
}

deinit {
    timer?.invalidate()
    timer = nil
}
```

Just before releasing an instance, the system will call the deinitializer automatically; do not call it yourself.

How can you create a method with default values for its parameters?

When we declare a function in Swift, we can specify defaults for our parameters by specifying values within the method declaration:

```
func sayHello(name: String = "reader")
```

Now, we can call this function directly without having to explicitly specify any parameters as the default values will be used instead.

If the function contains parameters that don't have a default value specified, you'll need to specify a value for that parameter as usual. Otherwise, the compiler will return a “missing argument” error.

Given this example function:

```
func sayHello(name: String = "reader") {  
    print("Hello, \(name)")  
}
```

These two function calls are equivalent:

```
sayHello()  
sayHello(name: "reader")
```

Now, let's consider this function declaration:

```
func logStatement(prettyPrint: Bool = false, includeTimestamp: Bool,  
                  enableVerboseMode: Bool = false, message: String) {}
```

As you can see, there are several parameters with default values specified, but `includeTimestamp` and `message` are explicitly required. When we create functions with a mix of parameters like this, Xcode's auto-complete will help enumerate all of the valid variations of the call to our function:



## What is type inference?

Type inference is the mechanism that enables Swift to infer the type of a variable without us having to specify it explicitly. This generally lends itself to writing cleaner and more concise code without compromising readability or type safety.

That's why we can write this:

```
var welcomeMessage = "Hello"
```

Instead of having to write (a.k.a type annotations):

```
var welcomeMessage: String = "Hello"
```

The compiler is able to *infer* that `welcomeMessage` is a `String` based off of the default value we've provided.

If we don't specify a default value, then we'll need to use type annotation to provide the compiler with the relevant information:

```
var red, green, blue: Double
```

## What does the `rethrows` keyword do?

You're likely already familiar with the `throws` keyword which is one of the simplest mechanisms for propagating an error in our code.

Swift also includes the `rethrows` keyword which indicates that a function accepts a throwing function as a parameter. More specifically, functions declared with the `rethrows` keyword **must have at least one** throwing function parameter.

Consider Swift's `map` function:

```
public func map<T>(_ transform: (Element) throws -> T) rethrows -> [T]
```

Simply speaking, we know that `map` takes in some function as input and applies that to every element in an array. If the passed in function doesn't `throw`, then we can call `map` without `try`:

```
func doubleInput(_ input: Int) -> Int {
    input * 2
}

[1,2,3,4,5].map { doubleInput($0) }
```

This is all pretty normal so far, but what if the function passed into `map` can throw an error? In that case, we'll have to call `map` with `try`:

```
func doubleInput(_ input: Int) throws -> Int {
    guard input != 0 else {
        throw Error.invalidRequirement
    }

    return input * 2
}

try [1,2,3,4,5].map { doubleInput($0) }
```

The takeaway here is that if `map` was instead declared with `throws`, in both examples we'd have to call `map` with `try` even if the passed in function didn't `throw`. This would be inelegant and would clutter our code with unnecessary `try` statements. On the other hand, if `map` were declared without `throws` or `rethrows` we wouldn't be able to pass in a throwing function to begin with.

The `rethrows` keyword allows us to handle both cases elegantly - throwing and non-throwing functions. It enables us to create functions that don't necessarily throw errors of their own, but simply forward errors from one or more of their function parameters when applicable.

## What does the `lazy` keyword do?

The `lazy` keyword allows you to defer the initialization of a variable until the first time it's used; it's similar to the concept of "lazy loading".

In the following example, the `fakeUser` variable will not be initialized until the first time the property is accessed. This allows us to prevent the slow process of creating the object until we're absolutely sure we'll need it:

```
lazy var fakeUser = try! User(dictionary:  
    JSONService.parse(filename:"FakeUserJSON"))
```

There's a few important things to note when working with `lazy` variables.

Firstly, a `lazy` property must always be declared as a variable. The `lazy` property won't have an initial value until `after` the containing object's initialization is complete. So, we'll need to be able to update the variable's value at a later point in the application's execution. Moreover, the initial value of the `lazy` property could be dependent on some outside factors which means the appropriate initialization value cannot be determined until runtime. So, for both these reasons, the `lazy` property needs to be mutable.

It's also common to create a `lazy` property for objects that are computationally expensive to initialize as there's no point creating an expensive resource if it's never used.

```
// Creating a DateFormatter is expensive.  
// lazy lets us ensure we only create it if we need it.  
private lazy var dateFormatter: DateFormatter = {  
    let dateFormatter = DateFormatter()  
    dateFormatter.dateFormat = "yyyy-MM-dd HH:mm"  
    return dateFormatter  
}()
```

It's also important to understand the distinction between a `lazy` property and a computed property. A computed property regenerates its value every time it's accessed whereas a `lazy` property creates its value once and then maintains it for the rest of the app's execution.

## What does typealiasing do?

One simple way to make your code more readable is to use Swift's `typealias` keyword which allows you to provide an alias for an existing type.

Imagine we were trying to represent time on a clock.

We could use `typealias` to make our tuple easier to work with:

```
typealias ClockTime = (hours: Int, min: Int)

ClockTime(3, 30)

func drawHands(clockTime: ClockTime) {
    print(clockTime.hours) // 3
    print(clockTime.min) // 30
}
```

An excellent example of `typealias` in action is Swift's `Codable` protocol:

```
public typealias Codable = Decodable & Encodable
```

By the way, did you notice that you can combine protocols with `&` in Swift?

Now, whenever the compiler sees `Codable` it will replace it with `Decodable & Encodable` and continue compilation.

If you wish to make the `typealias` declaration accessible throughout your codebase, declare it outside of any containing entity. Otherwise, the `typealias` will be limited in scope, as any other variable would be.

```
// Accessible across the codebase
typealias ClockTime = (hours: Int, min: Int)

class HelloWorld {
    // Only available within this class
    typealias Greeting = String

    func sayGreeting(greeting: Greeting) {}
}
```

It's easy to overuse `typealias` and thereby limit the discoverability of the code, but as long as you exercise a little restraint, you'll be fine.

What does the `associatedtype` keyword do?

Imagine we have the following `protocol`:

```
protocol Stack {
    func push(x: Int)
    func pop() -> Int?
}
```

This `protocol` would allow whatever entity conforms to it to have the functionality of a `Stack`.

But what if we wanted our `Stack` to work with `Doubles` or `Strings`? Our only option would be to duplicate the `protocol` and change the types:

```
protocol IntStack {
    func push(x: Int)
    func pop() -> Int?
}

protocol DoubleStack {
    func push(x: Double)
    func pop() -> Double?
}
```

Clearly, this approach would be a little silly and is obviously not scalable. Luckily, this is exactly the problem that `associatedtypes` can help us solve.

The `associatedtype` keyword allows us to provide a placeholder for the type of the entity that will eventually implement this `protocol`.

This allows us to generalize our `Stack`:

```
protocol Stack {
    associatedtype Element
    func push(x: Element)
    func pop() -> Element?
}
```

In the example above, we've declared this new type - `Element` - which doesn't exist anywhere else in our code. Now, whenever we implement this `protocol`, `Element` will be replaced with the type of the entity implementing `Stack`.

```
class IntStack: Stack {
```

```
func push(x: Int) {  
}  
  
func pop() -> Int? {  
}  
}  
  
class StringStack: Stack {  
    func push(x: Int) {  
}  
  
    func pop() -> Int? {  
}  
}
```

The compiler is able to infer that `Element` should be an `Int` and a `String` respectively based on our implementations of `push()` and `pop()` and the type of the parameters we've specified.

You can always explicitly specify what the substituted type should be if you prefer:

```
class IntStack: Stack {  
    typealias Element = Int  
  
    func push(x: Int) {  
}  
  
    func pop() -> Int? {  
}  
}
```

Now, our `protocol` is extremely extensible without us having to duplicate any code.

While iterating through an array, how can we get both the value and the index?

We can accomplish this easily with the help of the `enumerated()` function.

```
let title = ["Ace", "The", "iOS", "Interview"]

for (index, value) in title.enumerated() {
    print("Index: \(index), Value: \(value)")
}

// Index: 0, Value: Ace
// Index: 1, Value: The
// Index: 2, Value: iOS
// Index: 3, Value: Interview
```

The `enumerated()` function returns a sequence of pairs composed of the index and the value of each item in the array. Then, we can use tuple destructuring and a `for` loop to go through every element in the sequence.

A common mistake is to apply this function on a `Dictionary`. If you do this, the `Dictionary` will be treated as an array of tuples and your output will look like this:

```
let userInfo: [String: Any] = ["age": 25, "gender": "male"]
for (index, value) in userInfo.enumerated() {
    print("Index: \(index), Value: \(value)")
}

// Index: 0, Value: (key: "age", value: 25)
// Index: 1, Value: (key: "gender", value: "male")
```

Your app is crashing in production. What do you do?

Unfortunately, there's no one size fits all answer here, but the following explanation might provide a starting point.

The main focus of your answer should be on discussing an approach for isolating the bug, understanding how to replicate it consistently, how you would go about resolving the issue, creating a new release, and finally what preventative measures you would introduce as a result.

You can start off your answer by mentioning that you'll try and identify the iOS version, app version, and device type of the affected users. Then, if you have access to the device logs, you'll use them to help you reproduce the crash consistently. Knowing the exact steps that precipitated the crash will allow us to write better tests.

You could also mention that as part of the exploration into the issues, you'd use Exception Breakpoints, Debugger output, crash logs, etc. to help isolate the crash.

Or, if the issue is harder to replicate, you can mention that you might use the Debugger to manually set the application into all of its potential states. For example, if there was a crash during the sign in flow, you might manually put your app into the following states for testing purposes: logged in user, blocked user, account recovery mode, etc.

Subsequently, you might mention that once you've identified the offending area of code, you'll look at recent pull requests that introduced changes in that area and work backwards from there.

**You want to demonstrate that you'll be systematic and methodical in your debugging approach.**

Eventually, once you've identified a fix, you could talk about how you'll test the fix thoroughly and ensure that it works across different device types and app configurations (e.g. user account types, languages, regions, etc.).

It may also be prudent to discuss that you'd write additional tests to cover this area of code to ensure the issue doesn't happen again and to protect against future regressions.

With the fix in place, you can notify your team about the new hotfix and request a code review. You could also mention that if this crash is critical, you would request an expedited review from Apple.

Once the fix has been released, you can discuss how you'll monitor the new release and ensure that the reported crash numbers do in fact reduce and that your fix was successful.

With the crisis averted, your answer can now focus on how you'd prevent issues like this in the future.

You could bring up leveraging App Store Connect's phased release feature which will slowly release the new version of your application. This will allow you to prevent untested code from becoming immediately available to all users. With a phased release, if you detect stability issues, you can easily pull the release before it affects a larger percentage of your user base.

Your final point could be about starting a discussion with your team to better understand how this issue made it past your tests and the code review process.

The main objective here is to demonstrate a methodical approach that covers identifying the issue, resolving it, and how you would prevent similar issues moving forward.

## AutoLayout & UIKit

What is a view's intrinsic content size?

All `UIViews` have an `.intrinsicContentSize` property that specifies the amount of space the `UIView` needs to show its content in an ideal manner.

If you've ever used `UITableView.automaticDimension`, this is deferring to the `UITableViewCell`'s intrinsic content size to figure out the appropriate height for the cell.

As another example, if you had a `UILabel` with a custom font and word wrapping enabled, the intrinsic content size would be the size needed to show all of the text without any truncation.

How would you animate a view that has a constraint?

With AutoLayout, you can simply change a view's constraints and let the system figure out how to resize that view and all of its neighbors.

In interviews, I often see candidates mix AutoLayout constraint changes with manual changes to a view's frame. This combination of two different layout paradigms really complicates the logic and is often error-prone. I'd recommend you stick with AutoLayout whenever possible.

Now, assuming you have a reference to the constraint you want to manipulate, you can simply animate a change by updating the constraint's value like so:

```
imageViewHeightConstraint.constant = 80  
  
UIView.animate(withDuration: 0.5) {  
    self.view.layoutIfNeeded()  
}
```

Make sure you update the constraint outside of the animation block!

Remember to call `layoutIfNeeded()` on `self.view` and not on the view you are attempting to animate. Otherwise, the changes in layout will be applied without animation.

Additionally, Apple recommends calling `layoutIfNeeded()` once before the animation block to ensure all pending layout operations are completed.

It's very important that we call `layoutIfNeeded()` on `self.view` and not just on the view we are trying to animate. By calling this function on `self.view`, the animation and layout changes will "trickle down" through all of the other subviews.

Remember, you want to animate changes to the neighboring view's layouts as well - not just changes to any one particular subview.

On what thread should all UI updates be made?

All UI updates should be performed on the Main Thread.

When your application launches, the `UIApplication` is set up on the Main Thread. From here, all of the views in your app ultimately descend from this `UIApplication` instance.

This means that any interaction with a `UIButton` or changes to the text in a `UILabel` have to be performed on the Main Thread as the application itself is configured to run on the Main Thread.

Furthermore, pending changes to the UI are not applied immediately. They are drawn at the end of the thread's run loop. So, to ensure all of the UI updates are applied together (and are applied without any flickering or visual side effects), it's important to have them all queued on the same thread.

If we had multiple threads of varying priorities and computational abilities performing UI updates, then the UI would flicker or be unpredictable as each thread would be changing the UI independently of the state reflected in another thread.

Moreover, it's important to remember that the Main Thread is far more performant than all other threads. If we were to update our UI on the background thread (which has a lower priority and limited resources), the task would first be added to a queue of other potentially long-running tasks (i.e. downloading a file). Now, our UI update will not be applied until all of the other higher priority and previously queued tasks finish executing.

To avoid that poor user experience and to keep things simple, we delegate all of the “real-time” activity to the Main Thread. Since the Main Thread has so much more computational ability, delegating UI-centric tasks to it helps ensure the system applies these UI updates as fast as possible.

## What is the difference between bounds and frame?

This question is asked frequently during interviews, particularly as part of an initial phone screen or online assessment.

The bounds of a `UIView` is the rectangle expressed as a location (`x,y`) and size (`width, height`) relative to its own coordinate system `(0,0)`.

The frame of a `UIView` is the rectangle expressed as a location (`x,y`) and size (`width, height`) relative to the superview it is contained within.

Here's an example of the difference:

```
let sampleView = UIView(frame: CGRect(x: 20, y: 420, width: 100,
                                         height: 100))
sampleView.backgroundColor = .red
view.addSubview(sampleView)

// (20.0, 420.0, 100.0, 100.0)
print("Frame: ", sampleView.frame)
// (0.0, 0.0, 100.0, 100.0)
print("Bounds: ", sampleView.bounds)

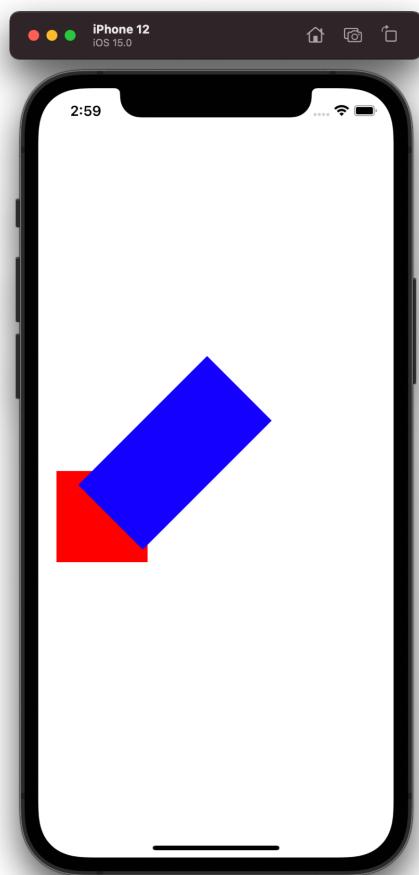
let rotatedView = UIView(frame: CGRect(x: 100, y: 300, width: 100,
                                         height: 200))

// Rotating the view a bit
let transform = CGAffineTransform(rotationAngle: CGFloat.pi / 4)
rotatedView.transform = transform
rotatedView.backgroundColor = .blue
view.addSubview(rotatedView)

// (43.933982822017896, 293.93398282201787,
// 212.13203435596424, 212.13203435596427)
print("Frame: ", rotatedView.frame)

// (0.0, 0.0, 100.0, 200.0)
print("Bounds: ", rotatedView.bounds)
```

The image below should help clarify the difference between the frame and the bounds:



## What is the Responder Chain?

iOS handles all user interaction - touch, press, shake, etc. - through something called the Responder Chain. This is essentially the hierarchy of all objects that have an opportunity to respond to user input.

If a particular object can't handle the event, it passes it up to the next item in the chain. This creates a hierarchy of objects that are equipped to handle user interaction of all types.

At the top of this hierarchy, you have the `UIApplicationDelegate`.

If you've ever placed your finger in a `UITextField` on iOS, you'll notice that the keyboard pops up immediately. From here, all subsequent user interaction events are sent to the `UITextField` to handle. This is because the `UITextField` is now the first responder - it's the first object in the hierarchy that has a chance to respond to user interaction.

That's why when you want to dismiss the keyboard, you have to write `textField.resignFirstResponder()` which is the `UITextField`'s way of saying that it's giving up control and wants to revert back to the previous Responder Chain hierarchy.

## What does an unwind segue do?

You can use an unwind segue to jump to any **UIViewController** further up your **UIViewController** hierarchy while simultaneously destroying all other **UIViewController**s in between. More specifically, you can use it to navigate back through one or more push, modal, or popover segues.

Let's say you're navigating from **ViewControllerA** → **ViewControllerB** → **ViewControllerC** and you want to go from **ViewControllerC** back to **ViewControllerA**. If you had to rely on just the back button alone, you'd have to go through **ViewControllerB** first.

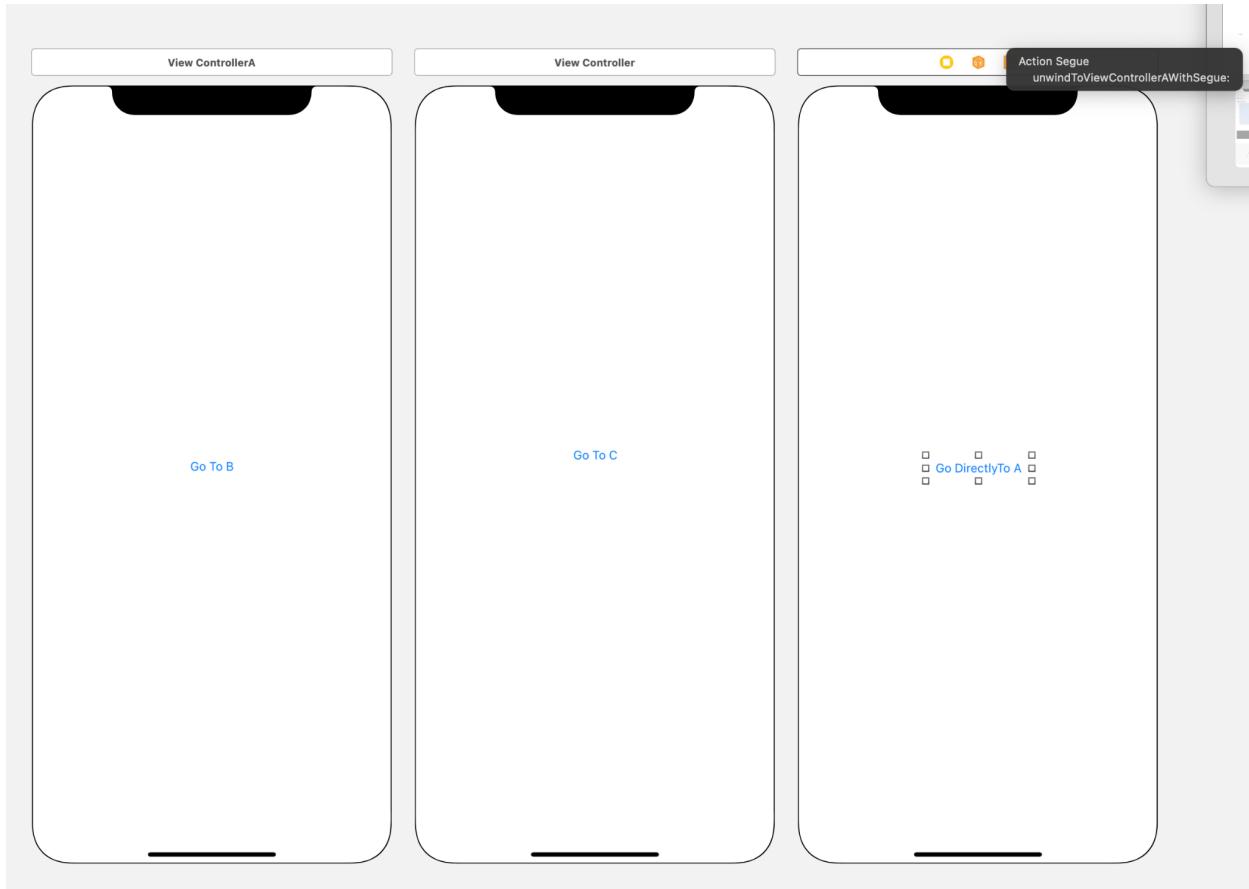
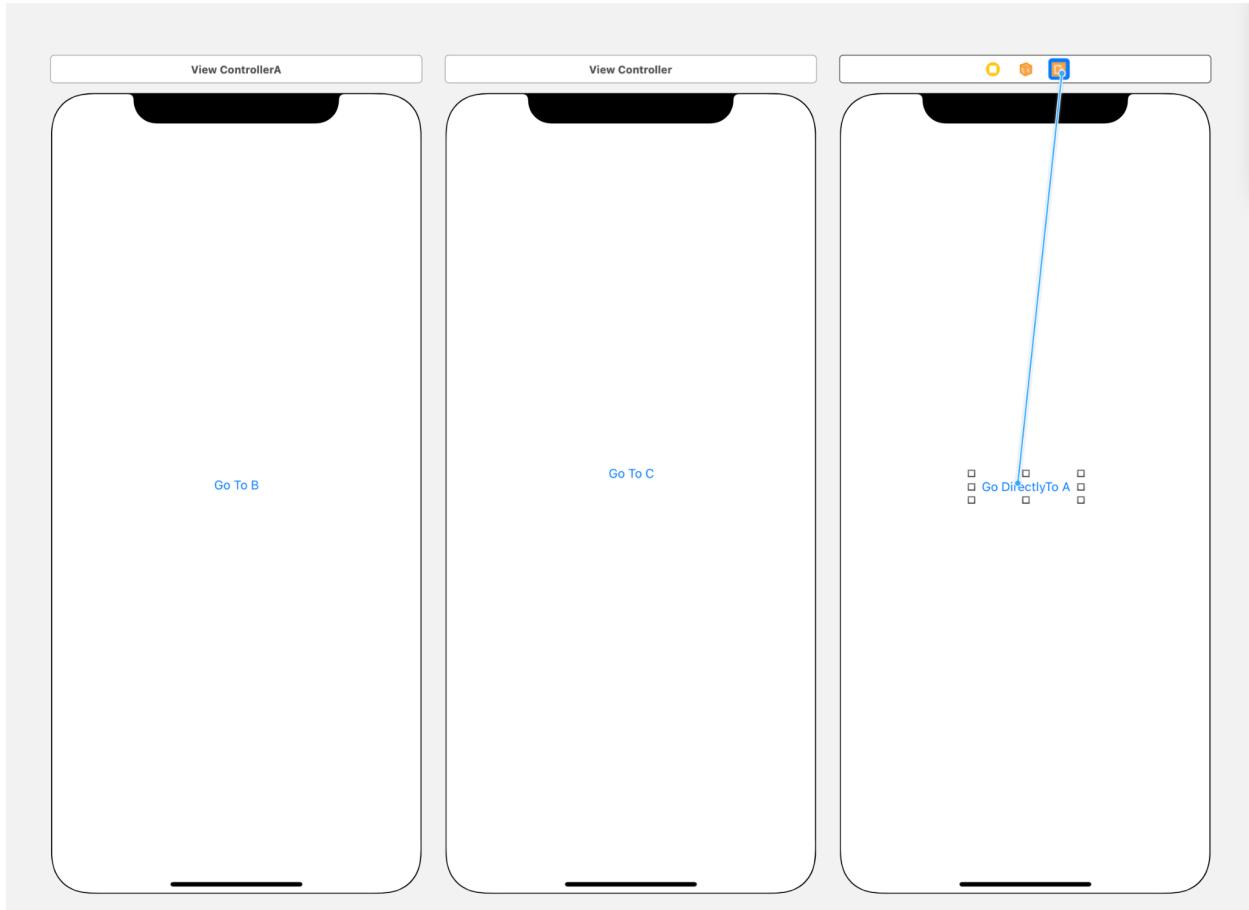
With an unwind segue, we can jump directly to **ViewControllerA** and destroy **ViewControllerB** along the way.

In **ViewControllerA**, add:

```
@IBAction func unwindToViewControllerA(segue: UIStoryboardSegue) { }
```

You can leave the method body empty.

Then, in the **UIViewController** you're departing from - **ViewControllerC** - you can drag from the body of the **UIViewController** to the Exit Icon and select the function we've created.



Now, when we press the “Go Directly to A” button, the application will navigate directly from **ViewControllerC** to **ViewControllerA** skipping over and destroying **ViewControllerB** along the way.

## What is the difference between pushing and presenting a new view?

Both pushing and presenting presentation styles have their own default behavior and conventions, so it's important to understand the differences between the two.

A push transition will add another `UIViewController` to a `UINavigationController`'s view hierarchy. The `UIViewController` that originates the push should belong to the same `UINavigationController` as the `UIViewController` that is being added to the stack.

With a push transition, you will automatically get a back button from the new `UIViewController` to the previous one. Additionally, you'll also get the ability to swipe to the right to pop the new `UIViewController` from the `UINavigationController`'s view hierarchy without writing any additional code.

Push transitions are only available to `UIViewControllers` that are embedded in a `UINavigationController` instance.

Now, turning to presenting a `UIViewController` (i.e. modal transition).

This is simply the case of one `UIViewController` presenting another `UIViewController` vertically over itself - neither of these `UIViewControllers` have to be embedded in a `UINavigationController`.

The modally presented `UIViewController` will typically appear without a `UINavigationBar` or `UITabBar` unless specified otherwise. Remember, though, that different versions of iOS have different default styling for modally presented views.

Finally, the presenting `UIViewController` is generally responsible for dismissing any modally presented `UIViewController` it presents.

What is the difference between a point and a pixel?

When you're working with designers daily, it's important to understand the difference between a point and a pixel.

As you probably know, a pixel is the smallest addressable element of your display or device.

A point is actually a measurement of length and defined as 1 / 72 of an inch. So, on a 72 PPI (pixels per inch) display one point will equal exactly one pixel.

Typically, it's used to measure the height of a font, but can be used to measure any length.

## How are Content Hugging and Content Compression Resistance different?

At its core, AutoLayout is a constraint solver. It will take some number of views and their constraints and try to formulate an arrangement that satisfies all of them.

Sometimes in this process, AutoLayout will make a view smaller than you'd like or may make it larger than you intend it to be.

In situations like this, we can leverage Content Hugging and Content Compression Resistance for more granular control over how AutoLayout resizes our views.

In order to understand how they work, we need to understand intrinsic content size. This is the minimum size views want to be to show all of their content. For example, views like `UIImageViews`, `UIButton`s, and `UILabel`s all know what their size should be in order to accommodate the content they're meant to show.

Content Hugging Resistance represents how hard a view is going to fight against being made larger than its intrinsic content size. The higher the priority, the harder it's going to resist growing.

Conversely, Content Compression Resistance represents how hard a view is going to fight against being made smaller than its intrinsic content size. The higher the priority, the harder it's going to resist shrinking.

So, when AutoLayout is trying to resolve constraints, it's going to look at these properties and their respective priorities to figure out which views it can make larger and which views it can make smaller.

## What does App Transport Security do?

App Transport Security (a.k.a ATS) is Apple's mechanism to ensure developer's use HTTPS in their app's communications.

As I'm sure you know, HTTPS is the secure variation of HTTP. In HTTPS, the same fundamentals of HTTP apply, but with the addition of using TLS (Transport Layer Security) to protect the privacy and integrity of the exchanged data from unauthorized parties and Man in the Middle attacks.

TLS relies on encrypting the communication both ways - from the client to the server and the server to the client. So, with App Transport Security, Apple is ensuring that developers use HTTPS (and as a result TLS) in order to ensure privacy and security for all apps and customers.

You can set `NSAllowsArbitraryLoads` to `true` in your application's `.plist` which will disable the ATS requirement, but your app will likely be rejected unless you can provide a compelling justification to App Review.

## What are layer objects?

Every `UIView` has a `CALayer` property which is responsible for the actual rendering of visual content and animations.

That's why we'll often write code like this:

```
layer.shadowOpacity = 0.3  
layer.shadowRadius = 2  
layer.shadowOffset = CGSize(width: 0, height: 2)  
layer.borderWidth = 0
```

We're giving instructions directly to the rendering component of the `UIView`.

There is an important distinction between the `UIView` and the `CALayer`. The `UIView` takes care of its own layout and placement, but it is the `CALayer` that is responsible for rendering its actual contents, including borders, shadows, corner radius, complex animations, and other visual effects.

## What is the purpose of the reuseIdentifier?

One of the performance optimizations `UITableViews` and `UICollectionViews` make is to only initialize enough cells to fill the user's screen. Then, whenever the user scrolls, instead of instantiating a new cell, it can just replace the contents of an existing previously allocated cell [the cell that is about to be scrolled off the screen]. This approach is not only very performant, but also utilizes less memory.

Imagine a `UITableView` with multiple custom `UITableViewCell`s. In order to perform the optimization mentioned above, the `UITableView` needs to be able to quickly find a cell that differs only in content, but shares the same layout.

This is exactly the problem that reuse identifiers solve.

`UITableViews` use reuse identifiers to understand what rows (if any) differ only in content and not in layout. These cells then become candidates for reuse.

That's why if you have a `UITableView` with multiple `UITableViewCell` types, you'll need to register multiple cell reuse identifiers.

```
tableView.registerClass(MyCustomCell.self,  
                      forCellReuseIdentifier: "MyCustomCell")
```

What is the difference between `layoutIfNeeded()` and `setNeedsLayout()`?

Building off the previous answer, let's take a closer look at the differences between `layoutIfNeeded()` and `setNeedsLayout()`.

### **setNeedsLayout()**

Calling this function tells the system that you want to invalidate the current layout of the view and trigger a layout update in the next update cycle. This function should only be called on the Main Thread.

This function will return immediately as it simply queues this task onto the Main Thread and then returns. Since the actual work is only done on the next update cycle, you can use this function to invalidate the layout of multiple views at once. Consolidating all of your layout updates to one update cycle is far better for performance.

In simple terms, this function will set a flag in the `UIView` that will indicate to the system that the view's layout needs to be updated. This, in turn, will schedule a call to `layoutIfNeeded()` which will check the status of this flag before proceeding.

Then, assuming the view's layout does in fact need to be updated, `layoutIfNeeded()` will call `layoutSubviews()` to update the view prior to the next update cycle.

### **layoutIfNeeded()**

Unlike `setNeedsLayout()`, `layoutIfNeeded()` tells the system that we want to apply the view's pending layout changes immediately and we **do not** want to wait for the next update cycle.

So, whenever you need to apply layout changes immediately, use `layoutIfNeeded()` not `setNeedsLayout()`.

When using Auto Layout, the layout engine updates the position of views as needed to satisfy changes in constraints. Using the view that receives the `layoutIfNeeded()` message as the root view, this method lays out the view subtree starting at the root. If no layout updates are pending, this method exits without modifying the layout or calling any layout-related callbacks.

What does `layoutSubviews()` do?

The default implementation of this function uses any constraints you have set to determine the size and position of all of the view's subviews. Overriding this method allows you to perform a

more precise layout of a view's subviews by setting the frame rectangles of your subviews directly.

Typically, you would only override this function if AutoLayout wasn't offering the behavior you wanted.

According to Apple's documentation:

You should not call this method directly. If you want to force a layout update, call the `setNeedsLayout()` method instead to do so prior to the next drawing update. If you want to update the layout of your views immediately, call the `layoutIfNeeded()` method.

## What does `viewDidLayoutSubviews()` do?

Simply put, `viewDidLayoutSubviews()` allows you to make customizations to views after they've been positioned by AutoLayout, but before they are visible to the user.

Whenever the bounds change for a `UIViewController`'s view (i.e. device rotation), it's likely that the position and size of all the subviews will need to be updated as well. So, the system will call `layoutSubviews()` to perform this change.

Then, once your `UIViewController` has finished laying out all of its subviews (all of your subviews are in their correct location and their frames honor whatever AutoLayout constraints you've specified), the system will call `viewDidLayoutSubviews()`.

From here, if you need to further customize or override any changes prior to the view being visible on screen, `viewDidLayoutSubviews()` gives you an opportunity to do so.

If you're wondering why we can't make these changes in `viewDidLoad()` or `viewWillAppear()`, it's because the frames of the subviews aren't finalized by the time those functions are called. They're only considered finalized once `layoutSubviews()` finishes running, so our only option to make customizations is in `viewDidLayoutSubviews()`.

The order of the `UIViewController` lifecycle is as follows:

1. `loadView()`
2. `viewDidLoad()`
3. `viewWillAppear()`
4. `viewWillLayoutSubviews()`
5. `viewDidLayoutSubviews()`
6. `viewDidAppear()`
7. `viewWillDisappear()`
8. `viewDidDisappear()`

## What does setNeedsDisplay() do?

Just like `setNeedsLayout()` tells the system the layout needs to be updated, `setNeedsDisplay()` informs the system that the view's content needs to be redrawn. This function queues the redrawing task and returns immediately. The view is only redrawn at the next drawing cycle at which point any and all views are updated.

Typically, you will only need to call `setNeedsDisplay()` if you are also overriding `drawRect()` in your implementation. As would be the case if you were working on a custom `UIControl` or if your view contained some custom shapes or effects.

`drawRect()` is responsible for the actual rendering of the view and is called by the system whenever drawing is required. You should never call this function explicitly; letting the system manage the calls to this function helps avoid multiple redraws if one has already been queued.

Let's say we have a `DrawLineView` which draws a line between 2 provided points. It might look something like this:

```
class DrawLineView: UIView {
    var point1, point2: CGPoint!

    // Only override drawRect() if you perform custom drawing.
    // An empty implementation adversely affects performance
    // during animation.
    override func drawRect(rect: CGRect) {
        // Draws a line from point1 to point2
        let context = UIGraphicsGetCurrentContext()

        CGContextMoveToPoint(context, point1.x, point1.y)
        CGContextAddLineToPoint(context, point2.x, point2.y)
        CGContextStrokePath(context)
    }
}
```

If we wanted to update the location and the length of the line, simply updating the values of `point1` and `point2` will not suffice. Changing these properties will **not** automatically redraw the line with updated starting and ending points. So far all we've done is change the underlying data, but we still haven't forced an update of the UI [forced a call to `drawRect()`].

As a result, we'll need to call `setNeedsDisplay()` after updating `point1` and `point2`:

```
drawLineView.point1 = CGPointMake(startDot.center.x, startDot.center.y);
drawLineView.point2 = CGPointMake(endDot.center.x, endDot.center.y);

drawLineView.setNeedsDisplay()
```

This call to `setNeedsDisplay()` will in turn call `drawRect()` which will actually redraw the line to reflect its new starting and ending locations.

Here's an example courtesy of [fujianjin6471 on GitHub](#). It may help to check out this project and play around with it to better understand `setNeedsDisplay()`'s role.

## What is a placeholder constraint?

There may be cases where a constraint can only meaningfully be set at runtime. Perhaps you don't know which subview to constrain a constraint too or maybe you don't know what the constant value of the constraint should be.

If you choose to omit this constraint at compile time, AutoLayout might complain about ambiguous constraints or an unsatisfiable layout. By using placeholder constraints, we can sidestep this issue.

Simply put, a placeholder constraint is a constraint that exists only at design time. They are not included in the layout when the app runs.

You typically add placeholder constraints when you plan to dynamically add constraints at runtime. By temporarily adding the constraints needed to create a non-ambiguous, satisfiable layout, you clear out any warnings or errors in Interface Builder.

## What is Dynamic Type in iOS?

Introduced in iOS 10, Dynamic Type allows developers to automatically scale their application's font size up or down to accommodate users with accessibility issues or users that need increased visibility. It can also accommodate those who can read smaller text allowing more information to appear on the screen.

Developers can choose to support Dynamic Text on a view-by-view basis. If you choose to add support for Dynamic Text, you can use `traitCollection.preferredContentSizeCategory` to retrieve the user's preferred content size and modify your UI styling accordingly.

You could also implement `traitCollectionDidChange()` which will notify you when the user's preferred content size setting changes. Then, you can make whatever additional UI changes you need to make based off of the new value in `traitCollection.preferredContentSizeCategory`:

```
override func traitCollectionDidChange(  
    _ previousTraitCollection: UITraitCollection?) {  
    super.traitCollectionDidChange(previousTraitCollection)  
    // Use this property to update your application's text styling  
    traitCollection.preferredContentSizeCategory  
}
```

If you override this function, make sure you always call `super.traitCollectionDidChange(previousTraitCollection)` first.

What are the differences between a .xib and a .storyboard file?

Both a .xib and .storyboard are stored as XML files and are both converted to binary files - .nibs - at compile time.

A .xib file usually specifies one **UIViewController** or standalone view whereas a .storyboard specifies a set of **UIViewControllers** and the navigation behavior between them.

What is the difference between layout margins and directional layout margins?

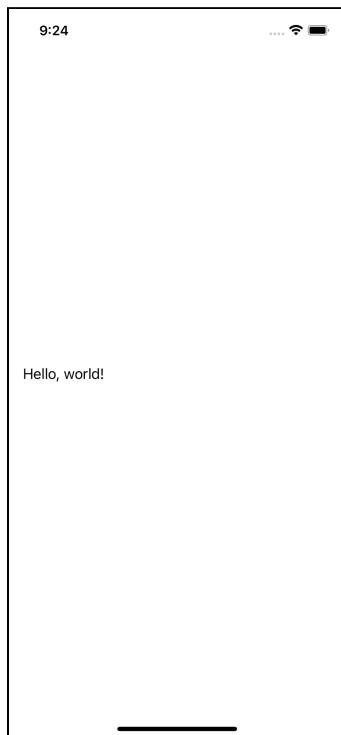
`layoutMargins` is a property of a `UIView` that allows the developer to specify the `top`, `left`, `bottom`, and `right` insets for a view's margin. The system defaults a `UIView` to an inset of 16 pixels on all edges.

```
override func viewDidLoad() {
    super.viewDidLoad()

    greetingLabel.translatesAutoresizingMaskIntoConstraints = false

    greetingLabel.centerYAnchor.constraint(
        equalTo: view.centerYAnchor).isActive = true
    greetingLabel.leadingAnchor.constraint(
        equalTo: view.layoutMarginsGuide.leadingAnchor).isActive = true
    greetingLabel.trailingAnchor.constraint(
        equalTo: view.layoutMarginsGuide.trailingAnchor).isActive = true
}
```

As you can see, the `UIView` is inset 16 pixels from the left:



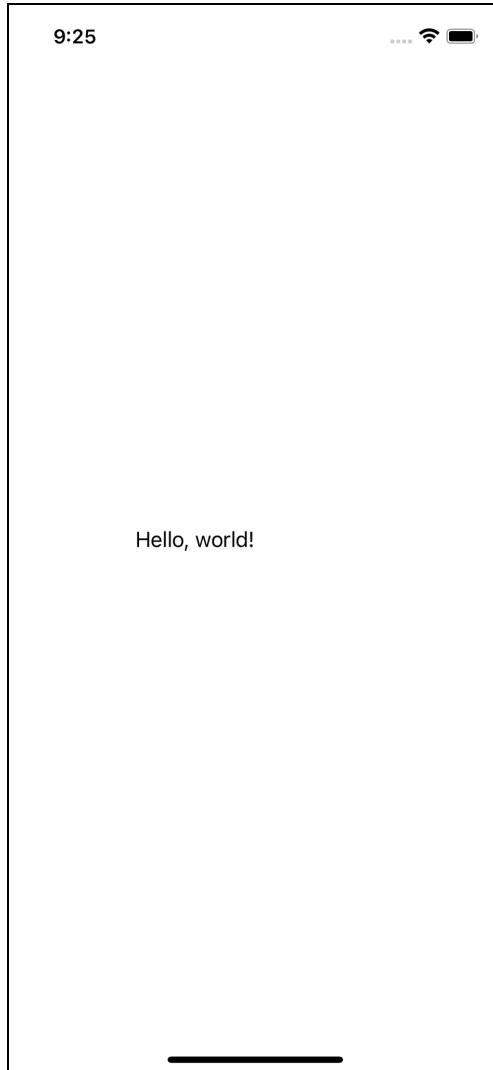
We can easily customize it with our own values:

```
override func viewDidLoad() {
    super.viewDidLoad()

    view.layoutMargins = UIEdgeInsets(top: 0, left: 100,
                                    bottom: 0, right: 0)

    greetingLabel.translatesAutoresizingMaskIntoConstraints = false
    greetingLabel.centerYAnchor.constraint(
        equalTo: view.centerYAnchor).isActive = true
    greetingLabel.leadingAnchor.constraint(
        equalTo: view.layoutMarginsGuide.leadingAnchor).isActive = true
    greetingLabel.trailingAnchor.constraint(
        equalTo: view.layoutMarginsGuide.trailingAnchor).isActive = true
}
```

All constraints relative to the `layoutMargins` will now honor the custom insets we specified above:



But, there's a silent issue here. What happens if our device uses a language that lays out right to left like Hebrew or Farsi? In that case, we'd want our `left` edge inset to start from the right-hand side and vice-versa.

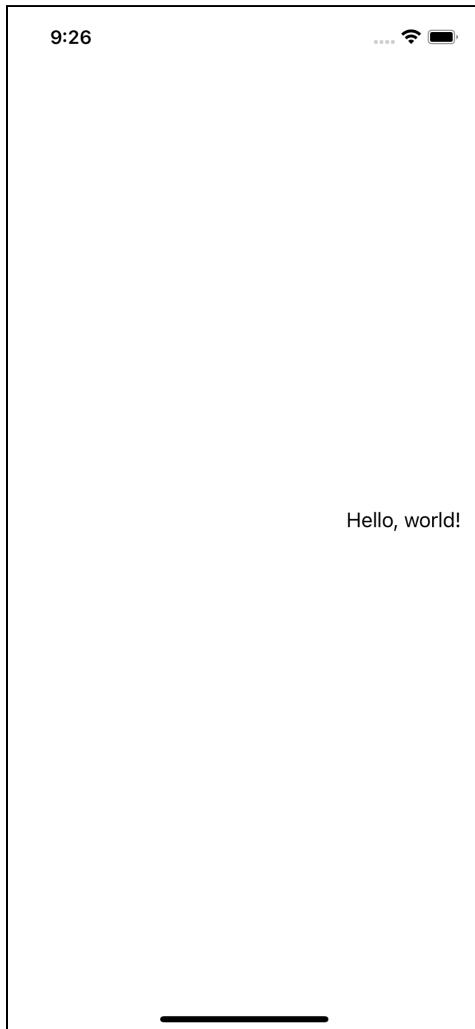
We can use `directionalLayoutMargins` to fix this. This property was introduced in iOS 11 and should always be used in place of `layoutMargins`.

It allows us to specify constraints and custom insets on a `UIView` while taking into account the current language's direction:

```
view.directionalLayoutMargins =  
    NSDirectionEdgeInsets(top: 0, leading: 100, bottom: 0, trailing: 0)  
  
greetingLabel.translatesAutoresizingMaskIntoConstraints = false  
greetingLabel.centerYAnchor  
    .constraint(equalTo: view.centerYAnchor).isActive = true  
greetingLabel.leadingAnchor  
    .constraint(equalTo: view.layoutMarginsGuide.leadingAnchor)  
    .isActive = true  
greetingLabel.trailingAnchor  
    .constraint(equalTo: view.layoutMarginsGuide.trailingAnchor)  
    .isActive = true
```

Notice the change in type - `directionalLayoutMargins` are of the `NSDirectionEdgeInsets` type instead and the `left` and `right` parameters are now replaced with `leading` and `trailing`.

When we update our implementation to use `directionalLayoutMargins`, we can see that our margins now honor the layout direction of the device's primary language:



The system keeps the `layoutMargins` property of the root `UIView` in sync with the `directionalLayoutMargins`. So, the `left` inset will automatically take the value of the `leading` or `trailing` margin depending on the layout direction.

## Concurrency

What are the differences between DispatchQueue, Operation, and Threads?

The differences between DispatchQueues, Operations, and Threads can be difficult to distinguish, so let's take a moment to review their similarities and differences.

### Threads

Simply speaking, threads allow a process to split itself up into multiple simultaneously running tasks.

In iOS, you have what Apple calls a "pool of threads." This is a collection of threads, other than the Main Thread, that the system can assign work to. The operating system will delegate work to one or more of these threads depending on the number of other competing tasks, the load on the processor, and a variety of other environmental factors.

The threads in iOS are so heavily optimized that if a developer were to interact with or dispatch work to them directly, they would almost certainly degrade the thread's performance or introduce issues related to reusability or locking. As a result, Apple provides abstractions over these threads like DispatchQueue and OperationQueue for developers to use rather than tasking developers with implementing their own custom thread management logic.

When we use these services, we lose visibility into exactly what threads are assigned to execute our work, but this deliberate abstraction simplifies our work tremendously. It allows us to put our trust in iOS to look at all of the relevant factors and dispatch the work intelligently on our behalf.

The main takeaway here is that you should always delegate your work to a DispatchQueue or an OperationQueue instead of trying to interact with the threads directly.

### DispatchQueue / Grand Central Dispatch

A DispatchQueue, also known as Grand Central Dispatch (GCD), allows you to execute tasks either serially or concurrently on your app's main and / or background threads. As a developer, you simply assign work to GCD and it will delegate it to one or more threads from the system's thread pool, but makes no promises or guarantees about which threads will be used.

DispatchQueues support both serial and concurrent execution. Serial queues synchronize access to a resource, whereas concurrent queues execute one or more tasks concurrently. In the case of a concurrent queue, tasks are completed according to their complexity and not their order in the queue.

In the simplest terms, DispatchQueue provides a layer of convenience over the system's thread pool. The developer simply needs to define the task - a DispatchWorkItem - and ship it over to a DispatchQueue which will handle all of the heavy lifting and thread management.

### **OperationQueue**

You can think of OperationQueue as a better, more powerful version of DispatchQueue.

Grand Central Dispatch (GCD) is a low-level C API that interacts directly with the Unix level of the system whereas OperationQueue is built on top of GCD and provides an Objective-C interface instead.

As we will discuss in a moment, OperationQueue comes with many advantages, but it also introduces some additional overhead. This is due to the fact that OperationQueue instances need to be allocated and deallocated whenever they are used. Although this process is heavily optimized, this additional overhead makes it slower than GCD.

So, what's the advantage of OperationQueue if it's slower than GCD? OperationQueue allows you to manage scheduled operations with much greater control than GCD provides.

OperationQueues make it easy to specify dependencies between individual operations, to cancel and suspend enqueued operations, or to re-use operations, none of which are possible with GCD.

Apple's official recommendation is to always use the highest level of abstraction available, but if you only need to dispatch a block of code to execute in the background and you don't necessarily need OperationQueues' additional features, I think DispatchQueue is a reasonable choice. If you need more functionality, you can always upgrade to an OperationQueue.

What is the difference between a serial and a concurrent queue?

We can create both types of queues with the help of `DispatchQueue` which is built on top of Grand Central Dispatch.

In the case of a serial queue, the tasks will complete in the order they are added to the queue (FIFO); the first task in the queue will complete before the next task begins.

Serial queues are often used to provide synchronized access to a shared resource in order to prevent race conditions.

```
let serialQueue = DispatchQueue(label: "aryamansharda")

serialQueue.async {
    print("This happens first!")
}

serialQueue.async {
    print("This happens second!")
}
```

As you'd expect, the output will be "This happens first!" and then "This happens second!".

On the other hand, a concurrent queue allows us to start multiple tasks at the same time. However, while we're starting all of these tasks at the same time, a concurrent queue makes **no guarantee** about the order in which the tasks will finish as these tasks are run in parallel.

If you need to establish dependencies between operations or need to ensure a completion order, consider using an `OperationQueue` or a serial queue instead.

```
let concurrentQueue = DispatchQueue(label: "aryamansharda",
                                    attributes: .concurrent)

concurrentQueue.async {
    print("I'm added to the queue first!")
}

concurrentQueue.async {
    print("I'm added to the queue second!")
}
```

Even though we're adding tasks sequentially, it's plausible for the second task to finish before the first one completes. It's simply a matter of how the system dispatches the work to the underlying threads and their respective workloads.

## What is a race condition?

A race condition occurs when there are multiple threads accessing the same memory concurrently and at least one of the threads is trying to update the information stored there (a.k.a a write task).

Imagine we had 3 threads, A, B, and C all accessing the same memory. In this example, A & B are simply reading the value at this location while C is trying to update the value stored there.

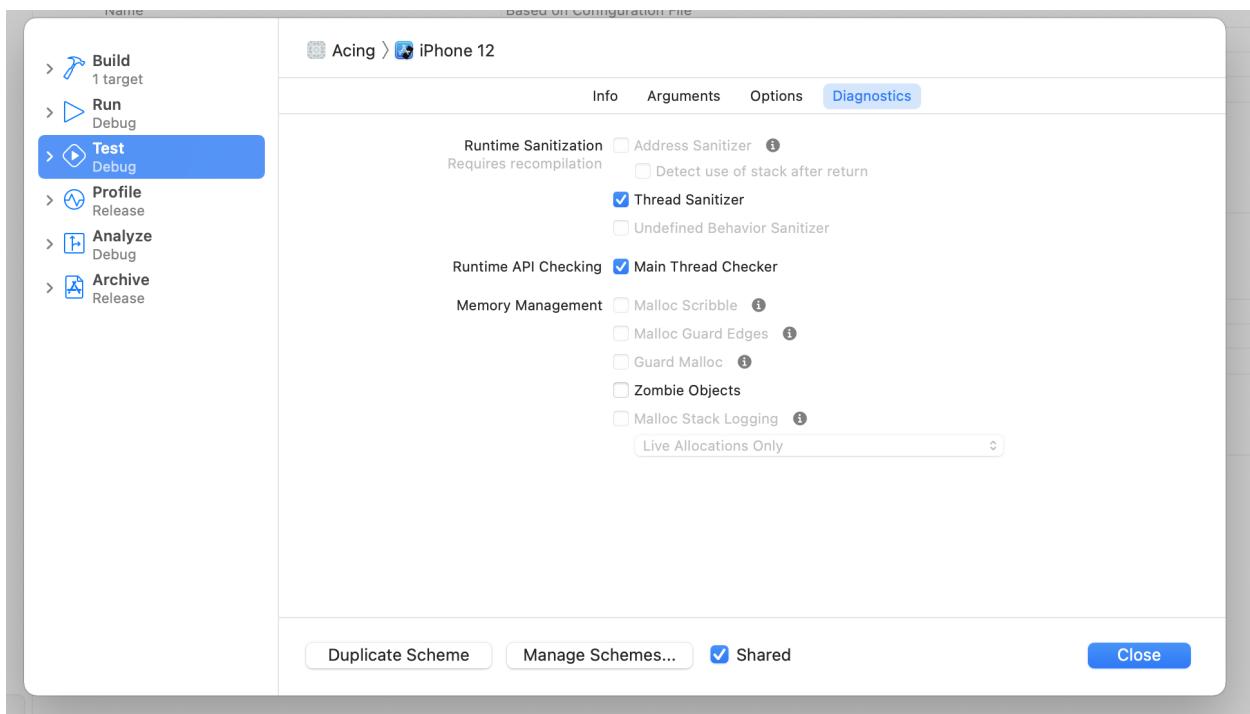
The order in which each thread completes their task can greatly influence the results of the other threads.

For example, if the task running on C finishes prior to the tasks on A & B, then A & B will both be seeing the updated value. Otherwise, if C finishes afterwards, A & B will have completed their tasks on what is now out of date information.

The order in which these threads complete their task is entirely dependent on the operating system's available resources, the respective thread's priority level, and in how the operating system's scheduler schedules these operations.

As you can imagine, this would introduce some variability with our code's execution and we now have a race condition on our hands; whichever thread *happens* to finish first can wildly and unpredictably influence the state of our application.

If all 3 threads were just reading the value, we'd have no issues.



Fortunately, Xcode provides us with the Thread Sanitizer tool to help us debug issues involving race conditions. This tool can be enabled by editing your target's scheme and will detect race conditions at runtime.

How can we prevent race conditions [the reader-writer problem]?

Continuing on from the previous question, let's see how we can prevent race conditions.

The solution comes down to ensuring synchronized write access to the shared resource. This would allow us to maintain the performance benefit of having multiple threads read from a shared resource simultaneously, but if any thread(s) wants to update the resource, we'll need to force that operation to happen in a synchronized manner.

The obvious solution would be to use a serial queue which ensures synchronized access to a resource by default.

However, if we want to use a concurrent queue, then we'll need to use a `DispatchBarrier` to make access to our resource thread-safe. `DispatchBarriers` help ensure that no writing occurs while reading and no reading occurs while writing.

When you add a `DispatchBarrier` to a concurrent `DispatchQueue`, the queue delays the execution of the `DispatchBarrier` block (and any tasks submitted after the `DispatchBarrier`) until all previously submitted tasks finish executing. You can think of the `DispatchBarrier` like a dam.

Once the tasks preceding the `DispatchBarrier` in the queue finish executing, the queue executes the `DispatchBarrier` block by itself. Then, once the block finishes, the queue resumes its normal execution behavior.

Effectively, by surrounding the write operation with a `DispatchBarrier`, we're ensuring that anytime we try to update this value all reads on other threads are blocked until the write is finished. It's in essence making a concurrent queue briefly serial until the write operation is finished.

```
class VisitorCount {  
    let queue = DispatchQueue(label: "aryamansharda.visitor.count",  
                             attributes: .concurrent)  
  
    private var visitorCount = 0  
  
    func getVisitorCount() -> Int {  
        queue.sync {  
            return visitorCount  
        }  
    }  
}
```

```
    }
}

func updateVisitorCount() {
    queue.sync(flags: .barrier) {
        visitorCount += 1
    }
}
```

In this example, you'll see that by using the `DispatchBarrier` we've ensured that only one thread can update the `visitorCount` at a time. Moreover, any subsequent calls to `getVisitorCount()` will have to wait until the `DispatchBarrier` is executed and the queue's normal concurrent behavior is restored.

## What does deadlock mean?

Deadlocks occur when threads sharing the same resource are waiting on one another to release access to the resource in order for them to complete their respective tasks.

Interviewer: "Explain deadlock and I'll give you the job."

Candidate: "Give me the job and I'll explain deadlock to you."

Simply put, deadlock describes a situation in which you have two or more entities waiting on one another and, as a result, no progress is made. A deadlock can occur between any two entities. For instance, your code could be waiting for a file system operation to complete and the file system could be waiting for your code to complete execution.

Here's a simple example of a deadlock:

```
func deadLock() {  
    let queue = DispatchQueue(label: "deadlock-demo")  
  
    queue.async { // A  
        queue.sync { // B  
            print("B: Waiting on A to finish.")  
        }  
        print("A: Waiting on B to finish.")  
    }  
}
```

Our queue is a serial queue so it's going to run these blocks synchronously - one after the other.

In this case, the inner closure (B) can't run until the outer closure (A) finishes. This is because A is holding the control of the current thread (remember it's a serial queue). A can never finish its execution because it's now waiting for B to finish running.

A depends on B and B depends on A and the thread stalls. One solution would be to change the queue type from serial to concurrent. This would allow the inner closure, B, to start without waiting for A to finish.

Be careful when working with serial queues as they can easily lead to deadlocks.

How can we group multiple asynchronous tasks together?

Swift's `DispatchGroup` allows us to group together and monitor the completion status of multiple asynchronous tasks.

With `DispatchGroup`, we attach multiple work items to a group and schedule them for asynchronous execution on the same queue or on different queues. Then, when all of the work items are finished executing, the `DispatchGroup` executes its `notify` completion handler.

There are several situations where you might use a `DispatchGroup`. For example, if you need to make several independent web requests to fetch information before the user can proceed, you can have each of the individual requests happen asynchronously, but group all of them together. This will allow each request to finish on its own accord, but will wait till all requests are finished before proceeding.

Or, if you were uploading a collection of photos, you might have several asynchronous photo upload tasks as this would allow each photo to be uploaded independently of the others. Then, when all photo uploads are complete, you could use the `notify` completion handler to show the user a success message.

With a `DispatchGroup`, you enter the group when you start your task and you leave when you finish. When the count of group members goes to zero - every `enter()` has a matching `leave()` - the `DispatchGroup` knows that it can go ahead and call its `notify` completion handler.

```
class DispatchGroupDemo {
    func uploadImages(images: [UIImage]) {

        let group = DispatchGroup()

        for image in images {
            group.enter()
            ImageUploader.upload(image) {
                // Successfully uploaded photo
                group.leave()
            }
        }

        group.notify(queue: .main) {
            // TODO: Show user success message
        }
    }
}
```

```
    }  
}  
}
```

Each `image` in `images` is uploaded independently and only once all of the upload tasks are complete will our `notify` closure be executed.

If you're using `OperationQueues` instead, you can simply add dependencies between tasks so they complete sequentially, but are essentially treated as a group of related tasks.

How can you cancel a running asynchronous task?

Let's look at canceling a task on a `DispatchQueue` first. In order to do this, we'll just need to maintain a reference to the `DispatchWorkItem`:

```
let task = DispatchWorkItem { [weak self] in
    print("Performing a task...")
}
```

```
DispatchQueue.main.async(execute: task)
task.cancel()
```

Things are a little more nuanced when we're working with an `OperationQueue`.

When we add an `Operation` to the `OperationQueue`, we're relinquishing control over to the queue which is now responsible for managing and scheduling those operations.

With the `OperationQueue` handling the heavy lifting, our only available actions are to resume / suspend the `OperationQueue` in its entirety, cancel all of the queued tasks, or call `cancel()` on a specific `Operation`.

When we call `cancel()` on an `Operation`, it does not force our `Operation`'s code to stop executing. Instead, it simply updates the `Operation`'s `isCancelled` property to reflect this change in state.

Our `Operation` could be in one of the following states when `cancel()` is called: finished executing, queued but not yet running, and currently executing.

If the `Operation` has already finished executing, this method has no effect.

Canceling an `Operation` that is currently in an `OperationQueue`, but not yet executing, makes it possible to remove the `Operation` from the queue earlier. When the `OperationQueue` arrives at our canceled `Operation`, the default implementation of the `start()` function will first check to see whether `isCancelled` is `true` and if so will exit immediately.

```
let operationQueue = OperationQueue()

let op1 = BlockOperation { print("First") }
let op2 = BlockOperation { print("Second") }
```

```
let op3 = BlockOperation { print("Third") }

operationQueue.addOperations([op1, op2, op3], waitUntilFinished: false)

// Ex: Pause / Resuming Operation Queue
operationQueue.isSuspended = true
if operationQueue.isSuspended {
    operationQueue.isSuspended = false
}

// Cancels a single Operation
op2.cancel()

// Output: true
print(op2.isCancelled)
```

You can also cancel all operations scheduled in an `OperationQueue` by calling `cancelAllOperations()`.

It may seem odd that all `cancel()` accomplishes is changing a boolean value, but that's about the only change we can safely make. What if there's cleanup that needs to take place? If we terminate an executing `Operation`, should we throw an exception? If the `Operation` stops now, will the data be corrupted?

Given the complications of canceling an `Operation`, you can see why our only option is to update this flag and handle the cancellation behavior on a case-by-case basis. If you want to support canceling a running `Operation`, then you'll need to add checks for `isCancelled` throughout the `Operation`'s code and modify your logic accordingly:

```
final class ExampleOperation: Operation {
    override func main() {
        guard !isCancelled else { return }

        // Do something....
    }
}
```

Can you explain the different quality of service options GCD provides?

The quality-of-service (QoS) options available on `DispatchQueue` allow us to categorize the importance of the work we're scheduling. The system will then intelligently prioritize tasks with higher quality-of-service designations.

Since higher priority work is performed more quickly and with more computational resources than lower priority work, it typically requires more energy than lower priority work. So, by accurately specifying appropriate QoS classes for the work your app performs, you can help ensure that your app is responsive and energy efficient.

There are 4 QoS options we'll look at in decreasing order of priority and performance:

#### **.userInteractive**

This designation should be used for work that is interacting with the user (i.e. refreshing the user interface or performing animations). In other words, it should be used for work that is of such high importance that if it doesn't happen quickly the application may appear frozen.

Any work queued with this QoS designation happens nearly instantaneously.

#### **.userInitiated**

This is used for work that the user has initiated and requires immediate results. Actions like retrieving information from an API, opening or modifying a file, or generally any work that needs to be completed in order to continue with the user flow.

Any work queued with this QoS designation is generally completed within a few seconds or less.

#### **.utility**

This is used for work that may take some time to complete and doesn't require an immediate result - actions like downloading or importing data.

Generally, tasks with this designation will also show some type of progress indicator to the user (like you see when you download a podcast or a Netflix episode). This QoS provides a balance between responsiveness, performance, and energy efficiency.

Any work queued with this QoS designation is generally completed within a few seconds to a few minutes.

### .background

This final designation has the lowest priority and is used for tasks that are not visible to the user like indexing, synchronizing, backups, saving data, or any general purpose maintenance work.

Any work queued with this QoS designation can take considerable time to complete; anywhere from a few minutes to a few hours as the computational resources and priority given to tasks with this designation are minimal.

By categorizing the tasks you send to `DispatchQueue`, you enable the system to optimize the completion of those tasks by reallocating resources away from lower priority work and redirecting it to the higher priority tasks instead.

What does it mean for something to be thread safe?

Code is considered thread-safe when it functions correctly during simultaneous execution by one or more threads. More specifically, thread safety ensures that some shared resource can only be modified by **one** thread at any given time.

Consider a scenario where your application has multiple threads all trying to access the same shared resource. Some threads may simply be attempting to read from the resource while other threads are trying to modify it.

Now, the resulting state of our application is entirely dependent on whichever thread *happens* to finish execution first. Remember, in software development, we strive for determinism wherever possible.

Imagine a situation where two threads are attempting to modify the same document. In such a scenario, it's possible for one thread to overwrite the changes made by the other. So, in order to make this operation thread safe, you have to ensure synchronized access to the resource. In other words, you'll need to make all of the other threads wait until the current thread modifying the resource finishes execution.

Keep in mind that the ordering of the threads' access to some shared resource is determined by the operating system's scheduler. Since the scheduler dispatches tasks based on the current resource availability, task priority, etc. it may order operations differently from moment to moment. So, there's no guarantee that the threads will always execute their tasks in a particular order, hence the "race condition" problem.

Thread safety is generally discussed in relation to modifying a resource. This is due to the fact that multiple threads can read from a resource at the same time since this operation is not changing the resource in any way.

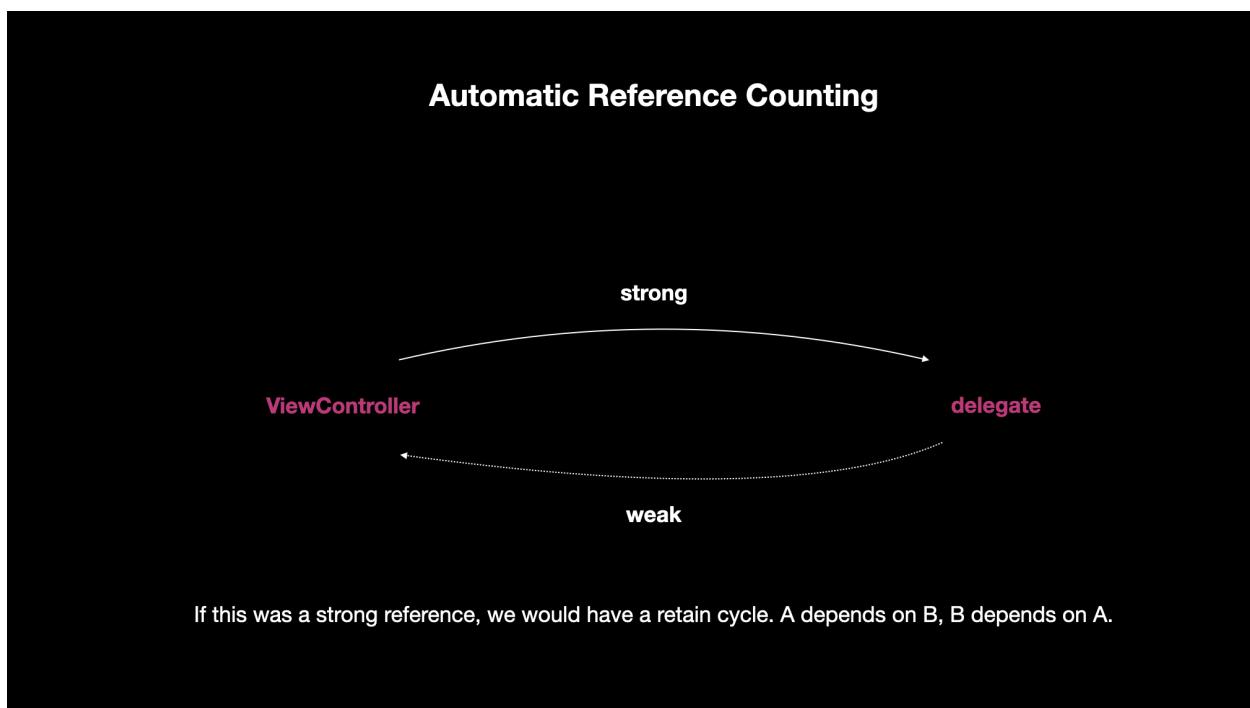
## Memory Management

What is automatic reference counting?

In simple terms, ARC is a compile time feature that helps us manage memory on iOS. ARC simply counts how many **strong** references there are to an object and when the count is zero, that object can be freed from memory.

Remember only a **strong** reference increases the retain count; a **weak** or **unowned** reference has no effect on the object's retain count. In iOS, a **strong** reference is the default.

Imagine that there is some **UIViewController** that implements a delegate. Since we're using a **strong** reference, we know the **UIViewController** is intentionally increasing the delegate's retain count to prevent it from being cleared from memory. In turn, the delegate has a **weak** reference back to the **UIViewController** so there's no change to the **UIViewController**'s retain count.



Instead, if we had a **strong** reference from the delegate back to the **UIViewController**, that would increment the retain count of the **UIViewController**.

Now, both items would have a retain count of one and neither object could ever be freed; the `UIViewController` depends on the delegate and the delegate depends on the `UIViewController`.

This is what we call a retain cycle.

We can prevent this retain cycle by making the delegate have a `weak` reference to the implementing object. Oftentimes, whenever a child object has a reference to its parent object, we'll make it a `weak` reference in order to avoid this exact issue.

That's why you'll commonly see delegates declared with the `weak` keyword like this:

```
class LocationManager {  
    weak var delegate: LocationManagerDelegate?  
}
```

Note that the `delegate` is an `Optional` as anything with a `weak` reference *can be nil* during its execution.

What is the difference between strong, weak, and unowned?

Please see the previous answer's explanation of Automatic Reference Counting.

All of these keywords are different ways of describing how one object maintains a reference to another object.

**strong** is the default keyword in iOS and will increment the reference count of whatever object it's referring to.

**weak** does not increment the reference count and the object it references can be **nil**. This is commonly used when working with delegates.

**unowned** does not increment the reference count either, but promises that the value it references will not be **nil** during its lifetime.

## What is a memory leak?

Memory leaks occur when a program incorrectly manages memory allocations such that memory that is no longer needed is not released. Additionally, it is also possible for a memory leak to occur when an object is in memory, but cannot be accessed by the running application.

In iOS, most memory leaks are a result of retain cycles.

This occurs when two entities keep a **strong** reference to one another. Since these entities' respective retain counts would be non-zero, ARC (automatic reference counting) would be unable to release either one.

The advantage of keywords like **weak** or **unowned** is that they allow us to create references to other objects without affecting their retain count. As a result, most memory leaks can be mitigated by making the offending reference **weak** or **unowned**.

If we need to debug a memory leak, we can use Xcode's Memory Graph Tool or the Leaks profiling template within Instruments.

How would you avoid retain cycles in a closure?

This will build off our understanding of ARC from the previous questions.

Oftentimes, closures will introduce retain cycles. Since a closure is a reference type it maintains a **strong** reference to all of the objects referenced in the body of the closure thereby increasing their retain count.

To manage this, we can use a capture list. This allows us to explicitly specify which objects we want to maintain a reference to, but more importantly whether we want those references to be **weak**, **strong**, or **unowned**.

We can pick **weak** or **unowned** (where applicable) to ensure that we can still reference all of the objects the closure needs, but we don't inadvertently increase their retain count and introduce a retain cycle.

The following example has a retain cycle as the body of the closure creates a **strong** reference to `isUserActive` and `isUserOnlineView`.

```
class RetainCycleDemo {
    @IBOutlet var isUserOnlineView: UIView!
    var isUserActive = false

    func setUserActivityStatusView() {
        userService.checkUserOnlineStatus { [isOnline] in
            self.isUserActive = isOnline

            if isOnline {
                self.isUserOnlineView.backgroundColor = .green
            } else {
                self.isUserOnlineView.backgroundColor = .red
            }
        }
    }
}
```

Fortunately, we can fix this by simply using a `weak` reference to `self` instead. And, voila - no more retain cycles!

```
class RetainCycleDemo {
    @IBOutlet var isUserOnlineView: UIView!

    var isUserActive = false

    func setUserActivityStatusView() {
        userService.checkUserOnlineStatus { [weak self] isOnline in
            self?.isUserActive = isOnline

            if isOnline {
                self?.isUserOnlineView.backgroundColor = .green
            } else {
                self?.isUserOnlineView.backgroundColor = .red
            }
        }
    }
}
```

# Testing

## What is a unit test?

A unit test is a type of automated test used to validate the correctness of a piece of code by providing an exhaustive list of inputs and ensuring that the expected outputs are returned.

Xcode provides easy support for writing unit tests by way of the `XCTest` framework. In our app's testing target, we could write a simple test like this and assert that the output from the function call matches our expected output:

```
func testOnlyEvenNumbersFilter() {  
    let input = [2,3,4,5,6,7]  
    let onlyEvens = Math.onlyEvens(input)  
    XCTAssertEqual([2,4,6], onlyEvens)  
}
```

Note: When you're creating tests in your testing target, the first word of the function name needs to be `test` in order for Xcode to register it as a unit test.

Unit testing can be a great way to protect our codebase against regressions and is far more scalable than relying on manual testing alone.

What does Arrange, Act, and Assert mean with respect to unit tests?

Arrange, Act, and Assert describe the ideal structure for a unit test.

Firstly, you have to **arrange** all the necessary inputs and preconditions. Secondly, you perform some **action** or operation on the object to be tested. And finally, you **assert** that the expected outcome has occurred.

Here's an example:

```
class EmailValidationTests: XCTestCase {
    func testValidEmail() {
        // Arrange
        let testEmail = "aryaman@digitalbunker.dev"

        // Act
        let isValidEmail = EmailValidation.validate(testEmail)

        // Assert
        XCTAssertEqual(isValidEmail)
    }

    func testInvalidEmail() {
        // Arrange
        let testEmail = "aryaman@digitalbunker"

        // Act
        let isValidEmail = EmailValidation.validate(testEmail)

        // Assert
        XCTAssertFalse(isValidEmail)
    }
}
```

This approach helps improve the readability of your tests and makes the expected behavior obvious.

## What are UI tests?

User interface tests allow us to test our application's behavior from the user's perspective. It helps us to ensure that our application's UI interactions, animations, and various user flows continue to work correctly as our application evolves.

While unit testing focuses on the inputs and outputs of the functions within our codebase, UI Testing aims to verify user-facing behavior.

To implement UI tests, we will continue to use the `XCTest` framework. `XCTest` will then leverage the iOS Accessibility System to interact with the various UI components specified by the test.

For example, a simple UI test that ensures the correctness of a login form might look like this:

```
import XCTest
class LoginViewUITest: XCTestCase {
    func testUserLogin() {
        // Launch app
        let app = XCUIApplication()
        app.launch()

        // Simulate button press
        app.buttons["loginButton"].tap()

        // Asserts that the Login form is shown correctly
        XCTAssertTrue(app.textFields["usernameTextField"].exists)
        XCTAssertTrue(app.textFields["passwordTextField"].exists)
    }
}
```

Xcode can also record and automatically create UI tests based on your interactions with the application. For most UI tests, you can just hit Record in Xcode, walk through some flow in your application, and then add the automatically generated test to your test suite.

## What is Snapshot Testing?

Snapshot Testing involves comparing the UI of an application against a set of reference images to ensure correctness.

If the difference between the actual UI and the reference image exceeds some custom threshold, the test fails. This is a really convenient way to ensure that there are no unexpected changes or regressions made to the UI of your application.

While UI tests allow us to test the functionality of our application, snapshot tests focus more on verifying that the implementation matches the agreed upon designs. Snapshot Testing is often easier to implement and update than writing traditional UI tests. Plus, it lets you easily verify the application's appearance across a variety of device sizes. Lastly, if you change the UI of your application, you'll be forced to update its corresponding snapshot test which helps ensure your testing suite remains up to date.

Currently, Snapshot Testing is not natively supported through Xcode. However, there are several open-source iOS libraries that enable you to add Snapshot Testing support to your app.

## What is Test Driven Development?

Test-Driven Development (TDD) is a software development process relying on software requirements first being converted to test cases prior to writing any production code. Then, the correctness of all subsequent development is measured against those tests.

A typical TDD workflow would look like this:

1. Write a single test for the functionality you intend to build or amend.
2. Run the test and ensure that it fails (which it should as the functionality is not yet built).
3. Write just enough code to ensure the test passes.
4. Refactor the code and perform whatever clean up is necessary (remove duplication, Single Responsibility Principle, etc.)
5. Rinse and repeat all the while building up new features and tests simultaneously.

This approach is useful in helping reduce the frequency of regressions in your application and in increasing your project's code coverage.

Writing these tests early on helps provide documentation about how the app is expected to behave. TDD promotes writing modular and testable code as it forces the developer to think in small units of functionality that can be easily tested.

In discussions of TDD, you may often see it broken down into 3 stages - Red, Green, and Refactor.

### **Red**

Create a unit test that fails.

### **Green**

Write just enough production code to make your test pass.

### **Refactor**

Once your tests are passing, you're free to make any changes to your code. This is your opportunity to clean up your implementation and refine your approach.

TDD only focuses on unit tests and doesn't cover UI behavior or integration tests, so it's often paired with additional testing paradigms.

## What is Behavior Driven Development?

Behavior Driven Development (BDD) is a software development methodology that aims to document and design an application based around the behavior the end user is expected to experience. In other words, it is behavior and results focused instead of implementation focused.

For example, when you're writing traditional unit tests, you'll often write tests for all of the various inputs a particular function could expect to receive and assert that the result matches what you'd expect.

This approach though is entirely focused on the implementation details of your application. As a result, you spend more time testing particulars of the implementation over the actual business logic of your application. BDD strives to tip the balance in the other direction by focusing on testing *what* your application does instead of *how* it does it.

In BDD, when writing tests, you'll start with a user story and model your tests around the expected behavior for the end user. While Swift doesn't support writing BDD style tests natively, popular frameworks like Quick & Nimble allow us to add BDD style testing to the iOS ecosystem.

Here's an example test case written in the BDD style - notice that we're focusing on testing the end result instead of how we get there:

```
describe("the favorite button") {
    it("is selected if the participant is already a favorite") {
        favorite = TestUtils.fakeFavorite()
        createFavoritesViewController()
        expect(viewController.favoriteButton.isFavorited).to(beTrue())
    }

    it("is not selected if the participant is not already a favorite") {
        favorite = nil
        createFavoritesViewController()
        expect(viewController.favoriteButton.isFavorited).to(beFalse())
    }
}

describe("Email/Password Authentication") {
    context("when user presses sign in") {
```

```
it("shows login view") {
    homeScreenViewController.signInButton
        .sendActions(for: .touchUpInside)

    expect(navigationController.topViewController)
        .toEventually(beAKindOf(LoginViewController.self))
}

context("when user presses sign up") {
    it("shows sign up and interactor not called") {
        homeScreenViewController.registerButton
            .sendActions(for: .touchUpInside)

        expect(navigationController.topViewController)
            .toEventually(beAKindOf(RegisterViewController.self))
    }
}
}
```

BDD tests follow the Given, When, and Then format.

If we consider the examples above, we start with the `describe` keyword which clarifies the action or behavior we're looking to test and establishes any preconditions for the test [given].

Next, the `context` block describes the conditions in which this behavior should be expected [when].

Finally, the `it` block specifies the expected behavior and validates the results [then].

Typically, you'd start with the user story and work backwards to fill out these sections.

This style of writing tests very closely models written language and as a result BDD style tests are often more readable and easily maintained than their unit testing counterparts. This style of testing also makes the expected behavior extremely clear and is reasonably self-documenting.

What are the purposes of `setUp()` and `tearDown()` in the `XCTest` framework?

Whenever we run tests, we'll often want each test to run independently. More specifically, we want to ensure that each test is executed under identical conditions and the end state of one test doesn't inadvertently influence the start state of another test.

The `XCTest` framework provides two functions we can use to ensure these conditions are met - `setUp()` and `tearDown()`. These functions allow us to prepare and clean up all allocations and dependencies before the next test is run.

`setUp()`: This method is called before each test is invoked and provides us an opportunity to reset state.

`tearDown()`: This method is called after every test method has finished execution and is used to perform any necessary clean up or releasing of resources.

These functions allow us to reuse data more easily in our tests.

For example, we can create instance variables to represent mock data in our `XCTestCase` class and then utilize the `setUp()` method to reset the initial state of these variables for each test and the `tearDown()` method to perform any necessary cleanup after the test is completed.

How would we test the performance of a method in our tests?

The `XCTest` framework includes a `measure()` function for profiling code execution times. We can call this method within our testing target to measure the performance of a block of code:

```
func testExample() throws {
    measure {
        sayHelloWorld()
    }
}
```

By default, this method counts the number of seconds it takes to execute a block of code.

To customize the metrics `measure()` function tracks, we can override `defaultPerformanceMetrics[]` and specify our own metrics to track.

The `measure()` function runs the code within the closure ten times and reports the average execution time and standard deviation. These numbers will now serve as the benchmarks for all future runs of this test.

On a subsequent run, if the execution time is within 10% of the benchmark value, the test will pass. Otherwise, the test will fail. This can help us catch performance problems early on in the development process.

If we were to improve the performance of the code we're profiling, we can simply update the baseline measurement, and now all subsequent executions will be compared against this new value.

Xcode saves benchmark numbers into source control which allows them to be standardized across teams.

Finally, the benchmark numbers are device-specific. This ensures that Xcode will not fail the test when the iPhone 6 performs worse than the iPhone 13.

## More Questions?

If there's an interview question you want to share with other readers or a mistake in one of the answers, [please message me on Twitter](#) or [email me at aryaman@digitalbunker.dev](mailto:aryaman@digitalbunker.dev).

# Acing the Behavioral Interview

I'd attribute my success in behavioral interviews to the use of the S.T.A.R format.

Truth be told, I was skeptical about the notion that a specialized answer format would fix all of my behavioral interview woes, but - generally speaking - it's done just that.

With just a little bit of practice, I noticed a marked improvement in the clarity and impactfulness of my responses. Ever since I've adopted this approach, my success rate and comfortability with behavioral interviews has improved significantly.

I'd reserve judgment until you've tried it out.

Even if you don't use this format during the interview, practicing with it is a great way of crafting concise and effective answers to common interview questions.

## S.T.A.R. Format

The S.T.A.R format simply states that you answer a behavioral question by first specifying the situation, then the task at hand, the action you took, and then finally the result.

**Situation:** Set the scene and provide the necessary context for the example you intend to share.

**Task:** Describe what your responsibility was in that situation. What needed to be done? Why?

**Action:** Explain the exact steps you took to address the problem.

**Result:** Share what outcomes your actions achieved. Quantify it when possible.

Despite the slightly contrived nature of the following example, I hope it helps demonstrate the usefulness of the S.T.A.R format as a mental model.

You can read more about the format [here](#).

**Question:** Can you tell me about a time when you had to manage competing priorities?

### Situation

In my current role, we have a very small iOS team. A few months ago, one of my colleagues left the company which meant the rest of us had to pick up their workload.

This additional workload was on top of the already full schedule we had managing releases, maintenance work, and feature requests from executives and product managers.

### Task

We knew we wouldn't be able to do everything, so we needed to find a better way to balance managing our technical debt and maintenance work with improving and building new features for our customers.

### Action

With our company's "customer first" philosophy in mind, I re-prioritized tasks based on what would be most impactful for our end-users. For tasks of equal priority, I would consider their potential benefit to shareholders and their ease of implementation to help break the tie. I also tried to schedule in a bit of time for tech-debt tasks, especially if the task seemed manageable at the time and would increase the team's future velocity.

Then, I shared this updated priority list with the rest of the team to ensure we were all on the same page. My team and I worked on this list in priority order and kept one another informed of any major developments or blockers.

### Result

Despite now being a smaller engineering team, we were still able to continue shipping important features. By placing an emphasis on developing the most important features for our customers, we increased our application's rating by half a point on the App Store.

And lastly, by proactively prioritizing tasks, getting everyone's buy-in, and increased inter-team communication, we have been able to absorb our colleague's work without increasing stress or affecting team morale.

## You're Not Out Of The Woods Yet

Having conducted my fair share of behavioral interviews, I would often get the impression that candidates treated this part of the interview as less important than the technical portions.

This couldn't be further from the truth.

In my experience, there have been several instances where the hiring committee has gone for the less technically qualified candidate purely because they were a better culture fit, more personable, or showed more genuine interest in the company and the team. While there is no shortage of technically skilled engineers, finding individuals that are sociable, team players, and great communicators can be challenging.

Performing well in your behavioral interview is especially important if you're interviewing for a Senior iOS role.

As a Senior or Staff engineer, you'd be expected to mentor Junior engineers, manage projects with tight deadlines, and collaborate with other teams. As a result, the characteristics - and more importantly, the limitations - of your personality and operating style become more consequential.

I've also noticed that candidates occasionally become too relaxed at this point and lose all formality and professionalism.

People seem to be under the false impression that if they've made it this far in the interview process and they've already defended their technical skills that the rest is just a formality.

If it was only that easy...

## An Interviewer's Perspective

Let's take a moment to consider the interviewer's perspective.

What are they looking for? How would their ideal candidate respond to their questions?

**When you put yourself in the interviewer's shoes, it'll be easier to craft answers that "check all of their boxes".**

Most interviews, especially those with the hiring manager, are interested in seeing how you work under pressure, your ability to work in a team, and how you communicate with other engineers and non-technical colleagues.

With every behavioral interview question, I'll try to craft my response to "move the needle" forward on one or more of these topics:

- Responsibility for one's actions
- Honesty and trustworthiness
- Being self-motivated
- Drive and ambition
- Reliability
- Adaptability
- Positivity
- Problem-solving ability
- Teamwork and collaboration skills
- Professionalism
- Time management skills
- Analytical skills
- Communication abilities
- Goal orientated
- Focused on personal growth and career development

## Before The Interview

The one saving grace of behavioral interviews is that you'll generally know what to expect going into them.

You'll likely start by introducing yourself, then take a deeper dive into your resume and background. Next, you'll go over specific questions about your previous projects, what interests you about this company, and finally an opportunity to ask the interviewer questions.

Regardless of this predictability, it's worth writing out your responses to a few common questions, conducting mock interviews with friends, and videotaping yourself so you can self-critique your responses and delivery.

And finally, it's important that you practice a lot - ideally out loud.

I'd also suggest spending some time crafting an elevator pitch for yourself. It should be concise and to the point, but provide a high-level overview of your education, past experience, projects, and why the company and position suit you. The interviewer is using this time to identify topics they want to delve deeper into.

Your elevator pitch should be tailored to highlight your alignment with the company's values and the company's overall mission.

One of the reasons behavioral interviews are formulaic is that interviewers need a standardized way of comparing multiple candidates. So, having solid answers, especially for the standard questions, can give you a leg up.

The following questions are low-hanging fruit that you can use to win some points in your favor early on in the interview. So, even if you stumble on an unconventional or difficult question later on, the overall interview will still be trending in the right direction.

Prior to the interview, you should look through the job description and try to come up with examples and anecdotes for each soft skill or job requirement it mentions. By providing examples that relate directly to the job description, it helps build your case as the ideal candidate.

It may be beneficial to spend some time reviewing your major projects and prior experiences working on a team, managing competing priorities, and all major professional achievements so you can work them into your answers.

If your examples or answers cite specific technologies, don't over-embellish your competency; be prepared to answer further questions about it. You've effectively opened the door for follow up questions and most interviewers will take you up on that opportunity to try and gauge how deep your experience *actually* is.

Generally speaking, in these types of interviews, you'll provide the range of topics and experiences, but the interviewer picks the depth.

Lastly, make sure you research the company and come up with some meaningful questions to ask the interviewer. Sometimes, I find myself viewing the candidate unfavorably when they ask a question that could easily be answered with a Google Search or a few minutes spent on the company's website.

## Focus On The Silver Lining

It's important to craft your answers so they end on a positive note.

Oftentimes, questions will try to bait you to blame your previous team or say something disparaging about your current employer. So, it's important that you're able to identify this line of questioning.

It's essential that you give an honest, but restrained response here.

For example, you may be asked to describe a failure or a missed deadline. While you should certainly acknowledge your shortcomings and provide some initial context, the main bulk of your response here should utilize the S.T.A.R format to convey how you solved the problem (be specific about the action you took) and the long-term results of your solution.

While it's important to acknowledge a shortcoming, make sure you don't spend too much time dwelling on it or emphasizing how large of a failure it was. I've seen this mistake in many candidates I've interviewed.

When possible, your description of the results should include some quantifiable metric or other means of making the results more tangible. The main goal here is to provide just enough context to the interviewer so that they can understand how clever and impactful your solution was and report back to the hiring committee about how enterprising you are.

## Be Specific

During the initial portion of the interview, the interviewer is looking for topics that they'll want to delve deeper into. So, you want to keep your answers succinct and just a few minutes in length.

It'll be clear when the interviewer wants to take a deeper dive into a topic. More information, especially at the beginning, is not always better.

In the early stages of the interview, the interviewer is still attempting to understand your background and the breadth of your past experiences. When you provide a lengthy answer to every question, it can often prevent the interviewer from arriving at this "big picture" understanding of who you are as a candidate and all of the prior experience and knowledge you're bringing to the table.

Your answers should focus on providing relevant context, discussing your decision making approach, technologies you leveraged, and concrete examples of how you've applied your skill set.

For example, when asked - "What are some of the most significant projects you've taken on in your current role?"

You might say:

"There have been 3 key projects that I've been a part of. At a high level, I helped redesign our search experience to increase booking conversions, improved our code coverage from 70% to 85% by adding a new testing utility to our development workflow, and built internal tooling for the iOS team to help manage our release process. Happy to go into more detail."

Feel free to continue using the S.T.A.R format here.

## Example Behavioral Interview Questions

Once you've been through a few rounds of the interview cycle, you'll start to notice that most behavioral questions tend to be pretty similar. They're all just different ways for the interviewer to learn about your personality, your working style, how you handle pressure and deadlines, etc.

The following is a comprehensive list of potential questions.

All of these questions do not require prepared responses, but - at a minimum - it would be prudent to prepare some responses for the high-level topics. And remember, it's important your answers end on a positive note and help demonstrate how you meet the requirements specified in the job listing.

Below are several questions I have asked or answered myself as well as questions that other iOS developers have shared with me.

### General

- What does your ideal working environment look like?
- How would you describe your communication and working style?

- What are some of your strengths and weaknesses?
- Tell me about yourself.
- Can you tell me about how you're making a difference in your current role?
- What appeals to you about working here? Have you used our product(s)?
- What's the most demanding project you've helped lead? What made it so difficult?
- How have you contributed to the culture of teams, companies, or groups in the past?
- What are you looking for in your new role?
- Why should we hire you?
- How would your current coworkers describe you?
- What career accomplishment are you the most proud of?
- What about the company's culture appeals to you the most? Do you think you'd fit in well?
- Why are you looking to leave your current role?
- If you were hired, what would make this position perfect for you?
- Which characteristics of your current job do you hope will continue into your next one?
- Could you tell me about one of your favorite projects? What made it a memorable experience?

## Teamwork

Typically, teamwork-centric questions are looking for examples of your ability to work with others under challenging circumstances (i.e. conflicts between team members, challenging project constraints, or clashing personalities).

- Tell us about a time when you had to work with someone whose personality was very different from your own.
- Can you tell me about a time you wished you handled a disagreement differently with a coworker?
- Did you ever have to routinely work with a difficult coworker? If so, how did you handle that interaction?
- Could you give an example of a time when a coworker wasn't responsive when you needed information? What did you do?
- Can you tell me about your most recent experience working on a team? What was your role and how did you contribute?
- Can you tell me about a time where your team members disagreed with you? How did you deal with it? Were you able to convince them?
- Have you been part of a team where the members weren't getting along? What did you do?

- Have you ever encountered a coworker who didn't do their work on a difficult project? How did you react?
- Could you describe an occasion when you and your team members had to compromise?
- What's normally your role on a team? Are you generally leading the projects or functioning more as an independent contributor?
- Do you ever delegate work to the members of your team? How do you choose the people to complete each task?
- Have you ever let your team down? Is there anything you would do differently next time?
- Can you tell me about a time when you had to collaborate with other engineers, designers, and product managers on a project? What was your role and how did the collaboration go?
- In your previous roles, what was the average team size? Have you ever been the team lead on a project?
- How would your current coworkers describe your working style?
- Have you ever had to compete against another co-worker (i.e. promotion)? If so, what were the circumstances? What was the outcome?
- Tell me about a time when you were in conflict with a peer and how the situation was resolved.
- Describe a team experience you found disappointing. What would you have done differently to prevent this?
- Can you tell me about a time where you felt like a good leader?
- Can you tell me about a high performing team you've been a part of? What were they doing right that helped enable their success?
- What about past teams would you like to keep in your new one? What would you like to change?
- Do you prefer working alone or as part of a team?
- Can you describe your ideal teammate? What characteristics of this individual are most important to you?

## Integrity & Ethics

- Has a teammate or manager ever questioned your honesty? What happened?
- Can you tell me about a time where it was difficult to be honest or own up to a mistake?
- If you noticed that your coworker was doing something incorrectly, would you intervene? If so, what would you do?
- Have you ever been in a position where you were dishonest? What happened?

- Can you describe a time when a colleague made a mistake and you caught it? How did you handle it?
- Can you describe a situation where you were faced with a major professional challenge? How did you handle it?
- Can you tell me about a mistake you made and what you did to correct it?
- Can you recall a time when your contribution was overlooked and a member of your team got credit for it instead? How did you deal with it?
- What would you do if you disagreed with one of your manager's decisions?

## Initiative & Innovation

- Tell me about a time when you came up with a unique solution to a problem at work.
- Have you ever changed a process or workflow at work? What process did you change? How did your coworkers react?
- Have you introduced any new initiatives or projects? Was it easy to get your team members' buy-in? What was the outcome?
- Can you tell me about a project that succeeded due to your contributions?
- Have you been bored at work and wished you had more to do? Can you tell me what you did to fix the issue?
- Describe a time when you went above and beyond what was expected of you. Did anyone acknowledge your efforts? If so, how did that make you feel?
- Tell me about a time when you noticed a process or project that needed improvement. What did you do?
- Have you ever had a creative idea fail? Tell me about it. What would you have done differently?
- How do you verify that your work is accurate?
- Tell me about a time when you noticed a small issue before it became a major one. Did you take action to resolve it? How did you prevent it from happening again?

## Leadership

- Can you tell me about a time where a project didn't have a leader, but you took charge? What exactly did you do?
- Have you ever had to lead a difficult group? What made it challenging? How did you manage them?
- What do you find most challenging about being a leader?
- Could you describe a project or task that challenged you as a leader? Did the challenge make you a better leader?

- When it comes to being a leader, what has been your greatest accomplishment?
- Can you recount a time when you helped motivate your team? If so, how?
- Have you ever disagreed with your manager's leadership style or the company's culture? If so, how did you respond?
- Tell me about a time when a coworker approached you with concerns about a project, workflow, or deadline. How did you handle the situation?
- What would you do if one of your teammates was not meeting expectations?
- How do you keep your team engaged?
- Could you tell me about a time when you helped a coworker reduce stress?

## Career & Personal Growth

- How do you set personal goals? How do you make sure they're both achievable and ambitious?
- Tell me about a goal you set and how you achieved it. What steps did you take to ensure you would meet your objective?
- Could you tell me about a goal you set, but didn't achieve? What prevented you from achieving it? What was your reaction? What did you learn from the experience?
- How do you set goals for your team? Were you able to keep everyone on track to achieve their goals?
- What would you say is the proudest moment of your career? Why was it important to you?
- Could you give me an example of a time when you felt unsatisfied with your work?
- Describe a time when your superior was not around and you faced a difficult situation. What steps did you take to resolve the problem? How did it turn out?
- When was the last time you asked your manager for direct feedback?
- What was your most recent career achievement?
- Have you ever been passed over for a promotion? Do you think it was fair?
- From your current job, what is the biggest lesson you'll carry with you?

## Decision Making

- Can you tell us about the biggest problem you've encountered on one of your projects? What made it difficult to solve? How did it turn out? What would you do differently now?
- What's your thought process prior to making a decision?
- Have you ever had to make a decision based on limited information? How did you decide what to do?

- Would you mind sharing a time when you were forced to make a quick decision (without all of the necessary information)?
- Have you ever felt pressured while making a decision? How did it affect your decision making approach?
- Could you describe a decision you made that affected your coworkers? How did you make your choice?
- Were there any decisions you made that weren't popular? How did you handle them?
- Did you ever regret one of your decisions? If so, why?
- What's your approach to solving difficult problems?
- Is it more important to finish a job on time or to do it right?
- Can you tell me about a time when you had to push back on some product or design requirements and what actions you took?
- Give me a recent example of a stressful situation on the job. What happened? How did you handle it?
- Think about a situation when you made a poor decision or did something that just didn't turn out right. What happened?
- Give me an example of a project that completely failed. Why do you think it was a failure? Could anything have been done differently in order to turn it into a success?
- Tell me about a time where you felt defeated (e.g. your project was falling apart, you were unable to meet your timeline goals, your idea was dismissed, etc). How did you respond to the adversity?

## Communication

Communication related questions will likely make up the majority of most behavioral interviews. The key here is to emphasize your thought process, your preparation, and how you adapt your communication style to cater to the present circumstances and the individuals involved.

- Can you tell me about a time you succeeded in convincing someone / your team to see things your way?
- Can you tell me about a time you had to explain a technically involved or complicated issue to a non-technical team member? How did you handle this situation?
- Can you tell me about a time when you had to say "no" to a feature request or scope change?
- How would you build rapport with someone that has a different personality and communication style than you do?
- Can you tell me about a time where you were unsuccessful in communicating your point effectively? How has this changed your communication style?
- Have you ever had to deliver bad news about a project, feature, or a missed deadline? What did you do to prepare and what was the outcome?
- Have you ever miscommunicated with a coworker or a manager? How did you rectify the situation?
- What was the communication style like with your previous bosses?
- Have you ever disagreed with your manager's decisions? How did you handle that?
- Can you give me an example of a time when you disagreed with your co-worker or another programmer? What actions did you take in that situation?
- Tell me about a situation where you had to speak up and be assertive in order to get a point across that was important to you.
- Tell us about a time your manager was unreasonable and how you handled it.
- Tell me about a time when you presented a great idea to management, but couldn't get their support. How did you respond? Were you able to change their minds?

## Adaptability

Sometimes even the best-laid plans go awry. This is especially true when it comes to software development where scope creep and design changes are inevitable. So, in order to succeed in such a role, it's important to demonstrate your resolve and your ability to adapt to changes.

Try and recall a work crisis you helped your team navigate. Your answers should emphasize the lesson you learnt or changes you made as a result of this experience.

Remember to always end your answers on a positive note.

- Can you tell me about a time when you had to be flexible and adapt to changing requirements?
- Tell me about a time when you had to think outside the box to solve a problem.
- Can you describe an experience where you had to learn something new quickly?
- How do you work under pressure?
- Can you tell us about a time when changes occurred outside of your control? How did you handle the situation?
- Can you tell me about a time where your project or your work environment were changing rapidly? How did you adjust?
- Do you remember a time when you changed the course of a project? How did you tell your team about the change? Are there any things you would have done differently?
- Have you ever had to adapt to a coworker's work style to accomplish a task? If so, how?
- Have you ever felt uncomfortable with a change? How did you deal with it?
- Has there ever been a time that, despite your best efforts, you were unable to achieve your goal? What happened and why?
- Can you tell me about a time you learnt from a previous mistake or missed opportunity? Have you changed your approach as a result?
- Have you ever had to adopt a new process, technology, or way of doing things that was a significant departure from the previous approach? What was your experience like adapting to the change?
- Can you tell me about a time when you were tasked with something outside your normal job duties? How did you handle it?
- Can you recall a time when you adapted to a change and your colleagues resisted it?
- Tell me about a project that had a major obstacle. What steps did you take to overcome the obstacle?
- What would you do if the requirements from the product team or stakeholders were vague?
- If you can, tell us about a time when your team experienced a significant restructuring. How did it affect you and your colleagues?
- What would you do if you were getting a lot of negative feedback during an app demonstration or testing session?

- If you made a strong recommendation in a meeting, but your team decided against it, how would you feel? What would you do?
- Do you remember a time when you found new information that affected a decision you had already made? How did you respond?
- Describe a situation in which you felt you had not communicated well enough. What were the takeaways?
- Tell me about a time when you had a disagreement with another programmer. How did you handle the situation? Were you able to reach a mutually beneficial resolution to the conflict? What could you have done to either prevent or resolve the conflict more easily?

## Time Management & Prioritization

The interviewer is looking for a demonstration of your abilities to manage multiple responsibilities, handle stress and deadlines, and your general organization and prioritization abilities.

In your answers, make sure to address your ability to delegate tasks and manage competing priorities.

- Consider a time when you had to prioritize certain tasks over others. How do you decide which tasks deserve higher priority?
- Have you ever been the engineering lead on a long-term project? How did you ensure the project remained on track?
- How do you manage stress?
- What would you do if you missed a deadline?
- How would you estimate a reasonable time to complete a task?
- Can you tell me about a time you had to manage multiple responsibilities? How did you make sure you didn't get overwhelmed?
- Have you managed a project before? How did you break down, assign, and track the tasks?
- Once you break down a project into tasks, how do you decide which to do first?
- What is the most stressful situation you've faced? How did you handle it?
- Could you tell me about a time when you experienced a situation that could have become stressful? What did you do to avoid it?
- How do you prioritize multiple competing requests from co-workers, managers, customers, etc.?
- How do you decide whether to delegate a task or complete it yourself?

- Sometimes it's impossible to accomplish everything on your to-do list. Can you recall a time when you felt overwhelmed by your responsibilities? How did you handle the situation?
- Are you better under pressure or with time to plan and organize?
- In your opinion, what is the one obstacle or issue you can foresee that could prevent you from meeting a deadline?
- How do you manage scope creep?
- Have you ever been stressed over a project delivery in the past? Did it affect your work-life balance? How did you deal with it?
- Tell me about a time when you had a problem working under pressure. How did you handle that situation? Did you decide to ask for support? Did you delegate some of your work? How and when did you ask for help?
- Can you describe an instance where your supervisor or manager gave you too much work with not enough time to complete it? What did you do?
- Describe a situation when you worked effectively under pressure. How did you feel? What was going on and how did you get through it?

To reiterate, the main goal for the behavioral interview is to demonstrate that you would be a great addition to the company and that you already embody the company's values and align with their mission.

Consider spending some time preparing answers to a few of these questions and try to make each answer showcase one or more of the company's values or requirements from the job listing.

## Engineering + Behavioral Questions

Most interviews will include time for platform specific questions. They're not quite "technical" in the whiteboarding sense, but more so a discussion of your depth of interest in the field and prior experience.

These questions provide an opportunity for the interviewer to learn more about the engineering processes you've used in your previous roles and your current development workflow.

The following are not only questions you might be asked, but they're also fair game to ask the interviewer in return:

## General Technical Questions

- What's your process for troubleshooting a crashing application?
- What was the last thing you read in a book or blog that you found helpful for your work?
- What publications and resources do you follow to keep up with industry trends?
- Describe a time when you helped optimize your application's performance.
- What are some of your favorite aspects of your career in technology?
- What are your least favorite aspects of your tech career?
- Do you have any personal or professional projects that you're particularly proud of?
- What testing methodologies are you familiar with?
- Is code review a part of your current development process?
- What developer tools do you use? What do you like or dislike about them?
- Have you ever been on-call before? What was your experience like?
- Are you involved in any open-source projects? If so, in what capacity?
- What team size are you used to working in?
- Have you ever been the tech lead for a project?
- What are some of the production projects you've been involved with so far? What technologies did you use?
- What's been the hardest bug you've ever had to resolve?
- Do you have experience working with and mentoring more junior engineers?
- Let's say your current project is on hold and no new project has been assigned. How would you spend that downtime?
- Imagine that there is a regression in the production version of our app. How do you mitigate it? How will you prevent new bugs from reaching customers?

## Platform Interview Questions

- What got you interested in iOS?
- How often do you attend iOS-related meetups or conferences?
- Do you contribute to any iOS open source projects?
- Describe your general testing practices when building an iOS app.
- If you could have Apple add or improve one API, what would it be?
- Have you ever filed bugs with Apple?
- How do you manage third-party dependencies? Do you prefer Carthage, CocoaPods, or Swift Package Manager? Could you explain how they're different?
- How do you manage iOS releases? What do you do if there's a critical bug in production?

- Who are some of your favorite independent Mac or iOS developers?
- What is your favorite iOS application?
- Do you build iOS apps outside of work? Can you describe your personal development workflow?
- What's your area of expertise (within iOS) and what area would you like to learn more about?
- If you could change anything about Xcode, what would you change?
- If you could change one thing about our iOS app, what would it be?
- Do you prefer designing views programmatically or using .storyboards / .xib? Why?
- What design and architecture patterns do you generally use in your apps?
- Which of the applications or features you've helped build are you most proud of?
- How do you stay up to date about developments in the iOS community? Any blogs, podcasts, etc. you follow?
- Do you have any experience with releasing an application to the App Store? Have you ever had any issues with Apple App Review?
- Of all of the projects you've been involved in, which one has had the biggest impact on you?
- What types of frameworks or tools do you think would be beneficial when starting a new long-term production project from scratch?
  - This could include topics like third-party dependency management (Cocoapods / Carthage / SPM), linters, formatting and documentation utilities, and testing frameworks.
- How would you avoid the Massive View Controller problem?
- Are you familiar with dependency injection? When would you use it?
- Why are singletons bad for testing?
- What is MVC?
- Have you used the Coordinator pattern?
- What are your thoughts on VIPER or Clean Architecture?
- Have you used MVVM before? Can you explain it to me?
- What do you look at when evaluating an architecture for a project or a feature?

## Questions For The Technical Interviewer

- How does engineering work get assigned?
- How are technical decisions made and communicated?
- How do you balance maintenance work and feature development?
- Can you give me an example of someone who's been in a technical role at your company for a long time and how their responsibilities and role have changed?

- How are disagreements resolved (technical, product, or interpersonal)? What happens when personalities clash?
- Is there a written roadmap all developers can see? How far into the future does it extend? How closely is it followed?
- What's the approach to managing technical debt?
- Have any of your employees spoken at conferences about their work?
- Do you have an engineering blog like Yelp, Twitter, etc.?
- Who is responsible for deployment? How often do you deploy?
- When something goes wrong, how do you handle it? What are the repercussions for a developer that introduces a breaking change or a bug?
- Can you tell me about your QA process? How do you ensure the stability of your application and protect against regressions?
- What happens when a critical bug makes it to production?
- What is the testing infrastructure like? Unit tests, integration tests, UI tests, snapshot testing, TDD, BDD?
- Do you have a QA team? If so, what is the QA process like?
- How do you implement regression testing?
- Is there automated testing in place?
- Are you using any architectural patterns to make development easier and more consistent?
- Can you describe your technology stack and the development tools you're currently using?

## When It's Your Turn To Ask Questions

Don't forget the interview goes both ways! You're interviewing the interviewer and the company too. You'll likely be at this company for some time, so you want to be absolutely sure you'll enjoy your role, the team, and are excited by and aligned with the company's direction.

You should take this time to not only ask any open questions you have, but to also demonstrate to the interviewer that you have a sincere interest in being a part of this organization. I've been on hiring committees where we've hired the technically weaker candidate in large part because they asked more insightful questions and conveyed more sincere interest than the more qualified candidate. You should use this time to demonstrate how much prior research you've done into the company and the market it operates in.

Prior to the interview, you should download the company's apps and spend some time playing with it. Ideally, you'd enter the interview with specific questions related to the app or potentially even ideas for future features and improvements.

Next, you should read through the company's website and engineering blog (if applicable) to ensure you're not asking questions that can be easily answered with a quick search.

At the very least you should take this opportunity to find out more about:

- Company culture
- Team size
- Career growth & development
- Your potential team members
- QA processes & workflows
- Staffing challenges
- Company morale
- Company's financial position and funding status
- Tech stack (for your prospective team and the company at large)
- General design & development workflow
- Expectations for employees
- Promotion and feedback cycle
- Employee onboarding process

You certainly don't need to ask all of these questions during the interview, but hopefully this gives you a shortlist to pick from and provides inspiration for other questions you're interested in having answered.

## Questions For The Hiring Manager

Your goal here should be to get a sense of their management style, their personality, what motivates them, and what it would be like to work with them on a daily basis.

Do they seem like someone who would be sincerely interested in your career growth? Would they give you opportunities to prove yourself? Could you see yourself having hard discussions with them? Would you feel comfortable disagreeing with their opinions? Are they laid back or demanding? Do they describe the role, the company, and the team favorably?

Here's a short list of potential questions to ask the hiring manager:

- Can you tell me more about the other members of the team I would be working with?
- How do you measure an employee's success in this role?
- What do you think is the biggest challenge facing the team / company right now?
- What is the engineering team's project planning process like?
- What are the opportunities for growth in this position?
- Does the company offer a learning stipend (or reimburse courses, certifications, workshops, etc.) ?
- Can you tell me about the current employee retention rate?
- How does this role fit within the current structure of the team?
- What is the team's process for prioritizing, planning, and assigning projects and tasks?
- How are design decisions made within your team?
- What's the onboarding process like for new engineers?
- Do you like working here?
- How would you describe your management style?
- Can you tell me about an interesting project or opportunity that the team worked on?
- What is a typical career path for someone in this position?
- Can you tell me more about the company culture? What about the culture of the engineering team specifically?
- What type of personalities generally work well within your team?
- Can you tell me more about the day-to-day responsibilities of this role?
- How often do engineers work together / participate in pair programming? Would inexperienced engineers get one-on-one time with more experienced engineers?
- Is it possible to change teams within the company? What's that process like?
- Do co-workers tend to hang out together outside of work?
- Can you tell me more about the team structure?

- Can you tell me a little bit about the company's financial health? Is the company currently profitable? Have there been any layoffs recently?
- What made you decide to join the company?
- Is there anything you wish you had known when you joined the company?
- How much input does engineering have on product development and decision making?
- What is the company / team's diversity and inclusion strategy?
- What do you enjoy most about your job? Is there anything you would like to change?
- What excites you most about the future of the company?
- What are your expectations for the company in the next five years?

# Afterword

Congrats on making it to the end!

I hope by now you're feeling more confident about your upcoming iOS interview - I know you're going to ace it. Writing this book has been one of the most difficult projects I've undertaken, but it has without a doubt made me a better engineer. In my opinion, teaching others is one of the best ways to improve as a developer. It forces you to delve deeper into topics than you would otherwise, which is excellent for filling in any knowledge gaps you may have.

Whether you land your dream job or learnt something new along the way, please consider leaving a review! I need a sign that I didn't waste 6 months writing this 😊

Also, if you come across something interesting or educational that would be useful to other iOS developers, please send it my way. I love to share quality educational content on my social media. Finally, if there's anything I've missed or if you would like to propose a topic or question for inclusion in this book, let me know that as well. You can email me at [aryaman@digitalbunker.dev](mailto:aryaman@digitalbunker.dev) or message me on [Twitter](#).

I will continue to update this book with new content, without charge, in the coming months.

It's time for a shameless plug, if you'll indulge me::

- Want to work with me at Turo? Apply here [**Top Workplaces USA 2022**):  
<https://grnh.se/22f48f191us>
- If you're interested in more of my content, check out my [Twitter](#), [YouTube](#), or [my personal website](#).
- To hire me for freelancing work, email me at [aryaman@digitalbunker.dev](mailto:aryaman@digitalbunker.dev) or direct message me on [Twitter](#).
- If you're in need of a deep link testing tool, check out my most recent developer tool [DeepLinkr](#).

I've got a few more books to come 😊

I've also created a [free to join Slack workspace](#) for anyone to discuss topics related to iOS interviewing, mock interview requests, sharing interview questions, job opportunities, and much more.

# Sources

- <https://hedgehoglab.com/blog/what-is-app-thinning>
- <https://www.swiftbysundell.com/tips/gathering-test-coverage-in-xcode/>
- <https://www.hackingwithswift.com/example-code/language/what-is-trailing-closure-syntax>
- <https://stackoverflow.com/questions/25156377/what-is-the-difference-between-static-func-and-class-func-in-swift>
- <https://betterprogramming.pub/demeters-law-don-t-talk-to-strangers-87bb4af11694>
- <https://developer.apple.com/swift/blog/?id=27>
- <https://www.avanderlee.com/swift/defer-usage-swift/>
- <https://sarunw.com/posts/how-to-declare-swift-protocol-for-specific-class/>
- <https://www.raywenderlich.com/books/swift-apprentice/v6.0/chapters/2-types-operatings>
- <https://developer.apple.com/documentation/swift/caseIterable>
- <https://mycodetips.com/swift-ios/what-is-the-objc-attribute-and-when-to-use-objc-in-swift-code-2369.html>
- <https://github.com/dashvlas/awesome-ios-interview/blob/master/Resources/English.md>
- <https://docs.swift.org/swift-book/LanguageGuide/AccessControl.html>
- <https://stackoverflow.com/questions/3656391/whats-the-dsym-and-how-to-use-it-ios-sdk>
- <https://www.hackingwithswift.com/example-code/language/what-is-copy-on-write>
- <https://holyswift.app/copy-on-write-in-swift>
- <https://www.swiftbysundell.com/basics/optionals/>
- <https://stackoverflow.com/questions/12622424/how-do-i-animate-constraint-changes>
- <https://medium.com/@duwei199714/ios-why-the-ui-need-to-be-updated-on-main-thread-fd0fef070e7f>
- <https://medium.com/flawless-app-stories/unwind-sequences-in-swift-5-e392134c65fd>
- <https://stackoverflow.com/questions/12561735/what-are-unwind-sequences-for-and-how-do-you-use-them>
- <https://stackoverflow.com/questions/12561735/what-are-unwind-sequences-for-and-how-do-you-use-them>
- <https://www.quora.com/What-is-the-distinction-between-pixels-and-points>
- [https://developer.apple.com/library/archive/documentation/WindowsViews/Conceptual/ViewPG\\_iPhoneOS/Introduction/Introduction.html#/apple\\_ref/doc/uid/TP40009503](https://developer.apple.com/library/archive/documentation/WindowsViews/Conceptual/ViewPG_iPhoneOS/Introduction/Introduction.html#/apple_ref/doc/uid/TP40009503)
- <https://www.hackingwithswift.com/example-code/language/whats-the-difference-between-any-and-anyobject>

- [https://docs.swift.org/swift-book/LanguageGuide/TypeCasting.html#/apple\\_ref/doc/uid/TP40014097-CH22-ID342](https://docs.swift.org/swift-book/LanguageGuide/TypeCasting.html#/apple_ref/doc/uid/TP40014097-CH22-ID342)
- <https://docs.swift.org/swift-book/LanguageGuide/Enumerations.html>
- <https://medium.com/@MarcStevenCoder/various-ways-to-unwrap-an-optional-in-swift-dcfe8188225>
- <https://docs.swift.org/swift-book/LanguageGuide/TypeCasting.html>
- <https://learnappmaking.com/type-casting-swift-how-to/>
- <https://learnappmaking.com/scene-delegate-app-delegate-xcode-11-ios-13/>
- <https://stackoverflow.com/questions/56498099/difference-between-scenedelegate-and-appdelegate>
- <https://stackoverflow.com/questions/9392744/what-is-the-difference-between-modal-and-push-segue-in-storyboards>
- <http://blog.fujianjin6471.com/2015/06/11/An-example-of-when-should-setNeedsDisplay-be-called.html>
- <https://bmnotes.com/2018/11/11/what-is-the-difference-between-setneedslayout-layutifneeded-and-layoutsubviews-quick-notes/>
- <https://developer.apple.com/documentation/uikit/uiview/1622601-setneedslayout>
- <https://developer.apple.com/documentation/uikit/uiview/1622507-layoutifneeded>
- <https://github.com/fujianjin6471/DemosForBlog>
- <https://www.infoworld.com/article/2920333/swift-vs-objective-c-10-reasons-the-future-favors-swift.html?page=2>
- <https://theswiftdev.com/5-reasons-to-choose-swift-over-objective-c/>
- <https://stackoverflow.com/questions/24032754/how-to-define-optional-methods-in-swift-protocol>
- [https://en.wikipedia.org/wiki/First-class\\_function](https://en.wikipedia.org/wiki/First-class_function)
- <https://cocoacasts.com/swift-fundamentals-what-are-first-class-functions-in-swift>
- <https://medium.com/the-andela-way/swift-understanding-mutating-functions-in-two-minutes-d9e363904e3a>
- <https://www.hackingwithswift.com/sixty/7/5/mutating-methods>
- <https://www.knowledgehut.com/interview-questions/ios>
- <https://www.raywenderlich.com/2271-operator-overloading-in-swift-tutorial#toc-anchor-004>
- <https://docs.swift.org/swift-book/LanguageGuide/AdvancedOperators.html>
- <https://www.hackingwithswift.com/example-code/language/what-is-a-selector>
- <https://www.quora.com/What-are-Swift-selectors-in-technical-terms-and-how-are-they-implemented>
- <https://useyourloaf.com/blog/delegation-or-notification/>

- <https://stackoverflow.com/a/3057818/2756409>
- [https://developer.apple.com/documentation/swift/imported\\_c\\_and\\_objective-c\\_apis/importing\\_objective-c\\_into\\_swift](https://developer.apple.com/documentation/swift/imported_c_and_objective-c_apis/importing_objective-c_into_swift)
- [https://developer.apple.com/documentation/swift/imported\\_c\\_and\\_objective-c\\_apis/importing\\_swift\\_into\\_objective-c](https://developer.apple.com/documentation/swift/imported_c_and_objective-c_apis/importing_swift_into_objective-c)
- <https://www.avanderlee.com/swift/concurrent-serial-dispatchqueue/>
- [https://developer.apple.com/documentation/dispatch/dispatch\\_barrier](https://developer.apple.com/documentation/dispatch/dispatch_barrier)
- [https://developer.apple.com/library/archive/documentation/ToolsLanguages/Conceptual/Xcode\\_Overview/UsingInterfaceBuilder.html#:~:text=xib%20.,and%20the%20transitions%20between%20them.](https://developer.apple.com/library/archive/documentation/ToolsLanguages/Conceptual/Xcode_Overview/UsingInterfaceBuilder.html#:~:text=xib%20.,and%20the%20transitions%20between%20them.)
- <https://mobikul.com/hashable-equatable-and-comparable-swift/>
- <https://nshipster.com/equatable-and-comparable/>
- [https://en.wikipedia.org/wiki/Test-driven\\_development](https://en.wikipedia.org/wiki/Test-driven_development)
- <https://www.agilealliance.org/glossary/tdd/>
- <https://useyourloaf.com/blog/changing-root-view-layout-margins/>
- <https://docs.swift.org/swift-book/LanguageGuide/Deinitialization.html>
- <https://developer.apple.com/documentation/dispatch/dispatchgroup>
- <https://stackoverflow.com/questions/43305051/what-are-the-differences-between-throws-and-rethrows-in-swift>
- <https://docs.swift.org/swift-book/LanguageGuide/Properties.html>
- [https://github.com/tspike/ios\\_fibonacci\\_tableview/tree/master/FibonacciTable](https://github.com/tspike/ios_fibonacci_tableview/tree/master/FibonacciTable)
- <https://stackoverflow.com/questions/59695137/how-to-decode-json-array-with-different-objects-with-codable-in-swift>
- <https://medium.com/chili-labs/configuring-multiple-cells-with-generics-in-swift-dcd5e209ba16>
- <https://www.avanderlee.com/swift/associated-types-protocols/#:~:text=An%20associated%20type%20can%20be.by%20a%20simple%20code%20example.>
- <https://www.donnywals.com/understanding-how-dispatchqueue-sync-can-cause-deadlocks/>
- <https://useyourloaf.com/blog/making-space-for-dynamic-type/>
- <https://www.hackingwithswift.com/read/39/5/measure-how-to-optimize-our-slow-code-and-adjust-the-baseline>
- <https://www.programiz.com/swift-programming/associated-value-enum>
- <https://www.donnywals.com/what-is-escaping-in-swift/>
- <https://www.marcosantadev.com/capturing-values-swift-closures/>
- <https://stackoverflow.com/questions/35065127/associated-types-in-swift>

- <https://github.com/dashvlas/awesome-ios-interview/blob/master/Resources/English.md#please-explain-arrange-act-assert>
- <https://cocoacasts.com/what-are-app-ids-and-bundle-identifiers/>
- <https://blog.cleancoder.com/uncle-bob/2014/12/17/TheCyclesOfTDD.html>
- <https://cocoacasts.com/choosing-between-nsoperation-and-grand-central-dispatch/>