Mobile Invader's

50 iOS Interview Questions

by KRISTIJAN KRALJ

Mobile Invader's

50 iOS Interview Questions

Kristijan Kralj

Version 1.0, April 2020

Table of Content

The Introduction

Welcome, dear colleague developer. I'll keep this introduction short since no one reads this anyway.

The purpose of this short ebook is to collect at one place all iOS interview questions that might pop up at your next interview.

The list is short for now, 50 questions. In the following months, I'll be adding more questions.

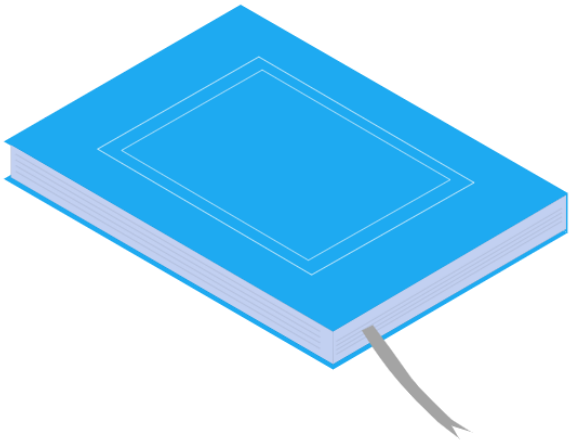
I have two favors to ask:

- If you see a wrong answer in this ebook, please let me know. I'll be happy to correct that in the next version of this book.
- If you have a suggestion on how to improve this book, or you would like to propose a question, please let me know.

You can reach me at kristijan@blueinvader.com.

Ok, enough with this introduction! Let's start with the questions.

The Basics



1. What is var and let?

   The `let` keyword is used to declare a constant variable and the `var` keyword is used to declare a variable. Variables created with these keywords are either references/pointers or values.

   The difference between them is that when you create a constant with `let`, you have to give it a value upon declaration (or within the calling scope) and you can't reassign it. In contrast, when you declare a variable with `var`, it can either be assigned right away or at a later time or not at all (i.e. be `nil`).

2. What is optional?

   An optional in Swift is a type that can hold either a value or no value. Optionals are written by appending **?** to any type:

```
let name: String?
```

3. What is nil in Swift?

   If a variable is equal to `nil`, this means that the variable doesn't have a value, or its value is set to nothing. This kind of value is necessary because not all constants and variables can be defined and set at the time of declaration.

4. What are classes?

   Classes are reference types. They increase their reference count when passed to a function or assigned to a variable or constant. They also have some extra features like an inheritance from a superclass, typecasting, and de-initializers.

5. What are structs?

   Structs are so-called value types. That means that when a struct is assigned to a variable or a constant or is passed to a function, its value is copied instead of increasing its reference count.

6. What are enums?

   An *enumeration* defines a common type for a group of related values and enables you to work with those values in a type-safe way within your code.

   You define a new enum by using an `enum` keyword:

```
enum CompassPoint {
    case north
    case south
    case east
    case west
}
```

7. What is the difference between classes and structs?

   The key thing about choosing between using a **class** or a **struct** is reference or value passing. If you need to store some primitives (i.e. **Ints**, **Floats**, **Strings**, etc.), use a **struct**. However, if you need custom behavior where passing by reference is preferable (so that you refer to the same instance everywhere), use **class**.

The other difference is that a **class** supports inheritance, while **struct** doesn't.

Generally in Swift, a **struct** is preferred over the **class.**

8. What is a protocol?

A protocol defines methods and properties which a class a struct or an enum, that adopts that protocol, must implement.

Swift also lets you provide a default implementation for your protocols with a protocol extension.

9. How to define and implement a protocol?

You define protocols in a very similar way to classes, structures, and enums:

```
protocol SomeProtocol {
    // protocol definition goes here
}

//implement the SomeProtocol
struct SomeStructure: SomeProtocol, AnotherProtocol {
    // structure definition goes here
}
```

10. What are generics?

Generic code enables you to write flexible, reusable functions and types that can work with any type. You can write code that avoids duplication and expresses its intent in a clear, abstracted manner.

Example of a generic function that swaps references to any two instances of the same type:

```
func swapTwoValues<T>(_ a: inout T, _ b: inout T) {
    let temporaryA = a
    a = b
    b = temporaryA
}
```

The generic version of the function uses a placeholder type name (called `T`, in this case) instead of an actual type name (such as `Int`, `String`, or `Double`).

In addition to generic functions, Swift enables you to define your own generic types. These are custom classes, structures, and enumerations that can work with any type, in a similar way to `Array` and `Dictionary`.

Example of such generic type:

```
struct Stack<Element> {
    var items = [Element]()
    mutating func push(_ item: Element) {
        items.append(item)
    }
    mutating func pop() -> Element {
        return items.removeLast()
    }
}
```

The type parameter is written within a pair of angle brackets (`<Element>`) immediately after the structure's name.

11. What is an extension?

Extensions add new functionality to an existing class, structure, enumeration, or protocol type. This includes the ability to extend types for which you do not have access to the original source code (known as retroactive modeling).

Declare extension with the `extension` keyword:

```
extension Person {
// new functionality to add to Person goes here
}
```

12. What are closures?

Closures are self-contained blocks of functionality that can be passed around and used in your code. This is a piece of code you can declare now and execute later.

Closures are reference types and will retain/strongly reference everything put in them unless otherwise specified. It has the following general form:

```
{ (parameters) -> return type in
    statements
}
```

13. What is an optional chaining?

Optional chaining is a process for querying and calling properties, methods, and subscripts on an optional that might currently be `nil`. If the optional contains a value, the property, method, or subscript call succeeds; if the optional is `nil`, the property, method, or subscript call returns nil.

Multiple queries can be chained together, and the entire chain fails gracefully if any link in the chain is `nil`.

You specify optional chaining by placing a question mark (?) after the optional value on which you wish to call a property, method or subscript:

```
let roomCount = john.residence?.numberOfRooms
```

14. What is UIKit?

The UIKit framework provides the required infrastructure for your iOS apps. It provides the view architecture for implementing your interface, the event handling infrastructure for delivering multi-touch and other types of input to your app, and the main run loop needed to manage interactions among the user, the system, and your app.

15. What is SwiftUI?

SwiftUI provides views, controls, and layout structures for declaring your app's user interface. The framework provides event handlers for delivering taps, gestures, and other types of input to your app, and tools to manage the flow of

data from your app's models down to the views and controls that users will see and interact with.

SwiftUI uses a declarative syntax so you can simply state what your user interface should do. For example, you can write that you want a list of items consisting of text fields, then describe alignment, font, and color for each field. Your code is simpler and easier to read, and that saves you time and maintenance.

16. What is the iOS application lifecycle - UIKit?

During running your app on an iOS device or simulator, the app can go through different stages and respond to various events. In iOS 12 and earlier, use the `UIApplicationDelegate` object to respond to life-cycle events.

- `application(_:willFinishLaunchingWithOptions:)` - Tells the delegate that the launch process has begun but that state restoration has not yet occurred.
- `application(_:didFinishLaunchingWithOptions:)` - Tells the delegate that the launch process is almost done and the app is almost ready to run.
- `applicationDidBecomeActive` - Tells the delegate that the app has become active.
- `applicationWillResignActive` - Tells the delegate that the app is about to become inactive.
- `applicationDidEnterBackground` - Tells the delegate that the app is now in the background.
- `applicationWillEnterForeground` - Tells the delegate that the app is about to enter the foreground.
- `applicationWillTerminate` - Tells the delegate when the app is about to terminate.

17. What is the iOS application lifecycle - SwiftUI?

In iOS 13 and later, use a UISceneDelegate object to respond to life-cycle events in a scene-based app.

- `scene(_:willConnectTo:options:)` - Tells the delegate about the addition of a scene to the app.
- `sceneWillEnterForeground` - Tells the delegate that the scene is about to begin running in the foreground and become visible to the user.
- `sceneDidBecomeActive` - Tells the delegate that the scene became active and is now responding to user events.
- `sceneWillResignActive` - Tells the delegate that the scene is about to resign the active state and stop responding to user events.
- `sceneDidEnterBackground` - Tells the delegate that the scene is running in the background and is no longer on screen.

18. What is ARC?

Swift uses Automatic Reference Counting (ARC) to track and manage your app's memory usage. In most cases, this means that memory management "just works" in Swift, and you do not need to think about memory management yourself. ARC automatically frees up the memory used by class instances when those instances are no longer needed.

ARC keeps track of **strong references** to instances of classes and increases or decreases their reference count accordingly when you assign or unassign instances of classes (reference types) to constants, properties, and variables. It deallocates memory used by objects which reference count got down to zero. ARC does not increase or decrease the reference count of value types because, when assigned, these are copied.

19. What is a strong reference cycle?

A strong reference cycle occurs when two instances reference each other in such a way that their reference count will never go to zero.

The common place where a strong reference cycle occurs is in closures. When you use closures within a class instance they could potentially capture `self`. If `self`, in turn, retains that closure, you'd have a mutual strong reference cycle between closure and class instance.

There are two ways to resolve a strong reference cycle: by using *weak references* or *unowned references*.

20. What are weak and unowned references?

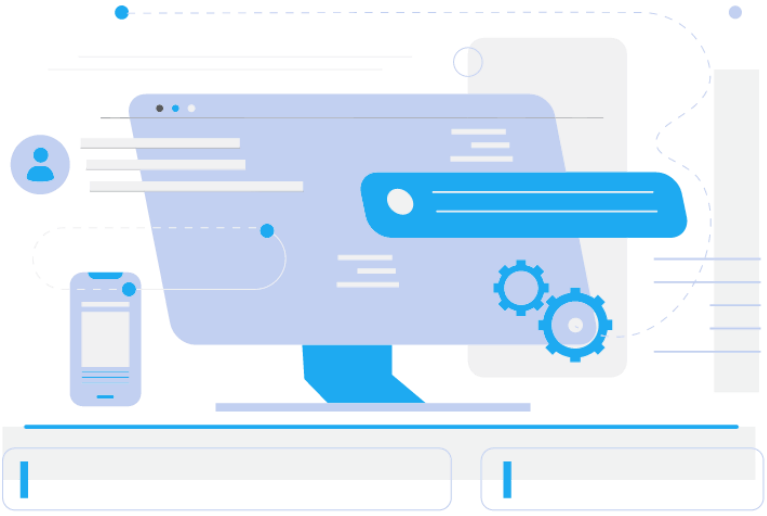
Weak reference is used when you know that a reference is allowed to become `nil`, whereas unowned reference is used when you are certain that the reference has a longer lifecycle and will never become `nil`.

Since weak references can have a value or no value at all, they must be defined as optional variables. An unowned reference has to be defined as non-optional since it is assumed to always have a value.

To define a variable as `weak` or `unowned`, use those two keywords before variable declaration:

```
weak var tenant: Person?
unowned let customer: Customer
```

Networking



21. What is REST?

REST is an API architecture that revolves around resources and HTTP verbs. Each resource is represented by a set of endpoints that can receive some of the HTTP verb requests to do various CRUD(Create, Read, Update, Delete) operations.

22. What is NSURLSession?

`NSURLSession` is a class that manages a group of related, network data-transfer tasks. It provides an API for downloading data from and uploading data to endpoints indicated by URLs. Your app can also use this API to perform background downloads when your app isn't running or, in iOS, while your app is suspended.

`NSURLSession` has a singleton `sharedSession` session (which doesn't have a configuration object) for basic requests. It's not as customizable as sessions you create, but it serves as a good starting point if you have very limited requirements.

23. What is Codable protocol?

In order to convert your data types to JSON, which is called encoding, and from JSON, which is called decoding, use Codable. This is necessary so you can send and receive data through network requests.

Codable is a typealias which combines Encodable and Decodable protocols. The class or struct you need to convert to JSON has to conform to Codable:

```
struct Landmark: Codable {
    var name: String
    var foundingYear: Int

// Landmark now supports the Codable methods init(from:) and
encode(to:), even though they aren't written as part of its
declaration.
}
```

24. How is multithreading handled in iOS networking?

The general idea with any kind of multithreading on iOS is that you don't want to block the main UI thread. That means that every HTTP or other service/networking layer request should be executed on a background thread. There are various mechanics in iOS and third-party libraries to help you with this but the most common solutions are Grand Central Dispatch (GCD) and `NSOperation`.

25. What is GCD?

Grand Central Dispatch (GCD) is a low-level API for managing operations either asynchronously or synchronously. GCD can be used to manage heavy tasks to the background so that we can improve our app's responsiveness.

26. How to download images in iOS?

For the most basic implementation of image downloading on iOS, you could use `NSURLSession` and download an image at a given `URL` as binary data, convert it from `NSData` to `UIImage`, and then display it.

There are also proven and popular libraries to use for this purpose. They can also offer advanced features. The most popular libraries are:

- AlamofireImage
- SDWebImage
- Kingfisher

27. How to cache images in iOS?

To cache an image, you could use the following approaches:

- In-memory cache - could be as simple as a `Dictionary` that keeps a reference to UIImage objects and uses the URLs they were downloaded from as unique keys. The other option is to use `NSCache` that performs something similar.
- Database cache - In this case, the image downloaded is stored in a database, for example, Core Data or Realm. This approach can work fine with smaller files.
- File storage cache - This saves downloaded images to the disk, so they can be retrieved later. Files stored like this have a unique name, so they can be easily identified and retrieved.

Local Storage

28. What is UserDefaults?

`UserDefaults` is an interface to the user's defaults database, where you store key-value pairs persistently across launches of your app.

`UserDefaults` can store primitive values as floats, doubles, integers, Boolean values, and URLs. The objects can be retrieved easily by accessing them with the key they were stored with.

Use `UserDefaults` to store user settings or preferences or tokens (although tokens really should be stored in Keychain).

29. What is Keychain?

Keychain is the best place to store small secrets, like passwords and cryptographic keys. You use the functions of the keychain services API to add, retrieve, delete, or modify keychain items.

30. What is Core Data?

Core Data is a persistence framework that helps you save objects to a database.

Use Core Data to save your application's permanent data for offline use, to cache temporary data, and to add undo functionality to your app on a single device.

Through Core Data's Data Model editor, you define your data's types and relationships and generate respective class definitions. Core Data can then manage object instances at runtime to provide the following features:

- Persistence
- Undo and redo of an individual or batched changes
- Background data tasks
- View synchronization
- Versioning and migration

31. What is NSPersistentContainer?

The `NSPersistentContainer` consists of a set of objects that facilitate saving and retrieving information from Core Data. Inside this container is an object to manage the Core Data state as a whole.

32. What is Realm?

Realm is an alternative to the Core Data database solution. It was built from the ground up to be easier to use and faster than Core Data or SQL. Advantages of Realm are that it's fast, has reactive features, is easier to use, is secure, and has an entire cloud platform for syncing and other more advanced features.

33. What is SQLite?

SQLite is a relational database that powers Core Data under the hood. It can be accessed directly on iOS and used without Core Data, but it will require implementing custom tooling for accessing, reading, and writing to it.

The main advantages of using SQLite directly are that it is going to be fast, and if you have SQL skills you can leverage them.

The main disadvantage though is that you'll have to do all the heavy lifting of setting things up and accessing and migrating the database yourself.

The Architecture

34. What is MVC?

Model–View–Controller (usually known as MVC) is a design pattern commonly used for developing user interfaces which divide the related program logic into 3 connected elements.

Components are:

- Model - Model objects encapsulate the data specific to an application and define the logic and computation that manipulate and process that data. For example, a model object might represent a character in a game or a contact in an address book.
- View - A view object is an object in an application that users can see. A view object knows how to draw itself and can respond to user actions.
  A major purpose of view objects is to display data from the application's model objects and to enable the editing of that data. Despite this, view objects are typically decoupled from model objects in an MVC application.

- Controller - A controller object connects view and model. Controller objects can also perform setup and coordinating tasks for an application and manage the life cycles of other objects.

35. What is MVVM?

Model-View-ViewModel (MVVM) is a design pattern which facilitates a separation of development of the graphical user interface from the development of the business logic or back-end logic (the data model).

The view model is responsible for exposing (converting) the data objects from the model in such a way that objects are easily managed and presented.

- Model - Same role as in the MVC.
- View - The same role as in the MVC, the view is the structure, layout, and appearance of what a user sees on the screen.
- View model - The view model is an abstraction of the view exposing public properties and commands. Instead of the controller of the MVC pattern, MVVM has some kind of binding mechanism, which automates communication between the view and its bound properties in the view model.

36. What is VIPER?

Viper is a design pattern that has five layers, doing a lot more separation of concerns. Viper abbreviates from View, Interactor, Presenter, Entity and Router.

- View - Consists of UI elements. It doesn't contain any business logic. Displays what it is told to by the Presenter and relays user input back to the Presenter.
- Interactor - Contains the business logic associated with the data. It fetches data from the model layer.
- Presenter - contains view logic for preparing content for display (as received from the Interactor) and for reacting to user inputs (by requesting new data from the Interactor).
- Entity - This is basically the model layer.
- Router - Contains navigation logic for describing which screens are shown in which order.

37. What is VIP?

VIP stands for View-Interactor-Presenter. The code in your project is structured around each of your application screens also known as *scenes.*

- View - The View has two main roles, one being as an entry point for any actions of the current scene, the other being to display the formatted information back to the user.
- Interactor - The role of the interactor is mainly the computation part of a scene. This is where you would fetch data (network or local), detect errors, make calculations, compute entries.
- Presenter - The presenter has a very precise role. Its main focus is to create a representation of parts of the data to be displayed on the screen at a specific moment in time. Once it has formatted the raw data the presenter sends it back to the controller to be displayed.

38. What is the best architecture in iOS?

This is a trick question and really there is no right answer. The best answer is: **it depends**.

When considering different architectures for your next app, it is important not to ask which is the best, but which is the best for the current situation. If the app is really simple, and it will stay like that, use MVC. On the other hand, if the app is going to be large, then use MVVM, VIP or VIPER.

39. What is the Singleton design pattern?

Singleton is a design pattern that restricts the application of just having one instance of a particular type. This is useful when exactly one object is needed to coordinate actions across the application. A singleton class returns the same instance no matter how many times an application requests it.

The most commonly used singleton used are:

- `URLSession.shared` - returns the shared session object.
- `UserDefaults.standard` - Returns the shared user defaults object.
- `UIApplication.shared` - Returns the shared app instance.

40. What is the Coordinator pattern?

In iOS applications that implement MVC or MVVM design patterns, the coordinator pattern serves to implement navigation functionality of the application. The advantage of this is that ViewController class then doesn't have to be responsible for navigation, and that's a good thing.

41. What are the SOLID principles?

S.O.L.I.D is an acronym for the 5 object-oriented design principles by Robert C. Martin (also known as Uncle Bob). If a developer follows these principles, then it should be easy for him to create applications that are easy to maintain and extend.

The principles are:

- **Single-responsibility principle** - An object should only have a single responsibility, that is, only changes if one part of the application changes.
- **Open-closed principle** - Objects should be open for extension, but closed for modification.
- **Liskov substitution** - you should be able to replace objects in an application with instances of their subtypes without changing the other code that uses that supertype.
- **Interface segregation principle** - Many smaller protocols are better than one big protocol.
- **Dependency inversion principle** - An object should depend upon protocols, not concrete types.

42. What is a Dependency injection?

Dependency injection is a technique where you give to an object the dependencies it needs. Dependencies are other objects that our object needs to fulfill its functionality.

The advantage of dependency injection is that the object who wants to call some services doesn't have to know how to construct those services. Instead, the object delegates the responsibility of providing its services to external code.

43. What is Protocol Oriented Programming (POP)?

Protocol oriented programming (POP) is a way of designing new features by expressing what the code should be doing with protocols. Then the concrete types implement these protocols. The goal is to have more smaller protocols, where each protocol only defines a narrow set of functionality.

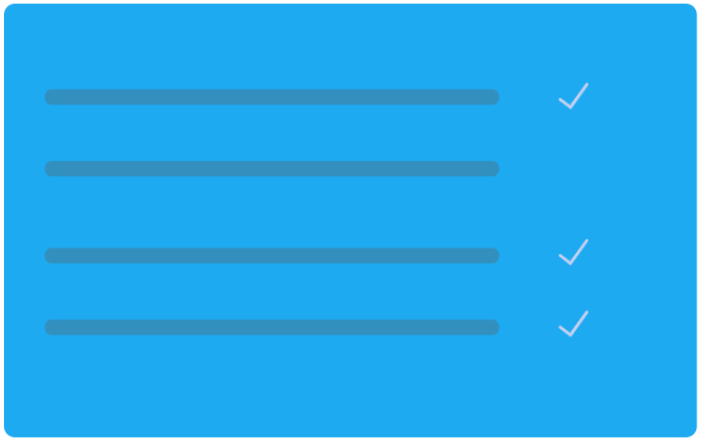
44. What access control levels exist in Swift?

Access control restricts access to parts of your code from other code. This enables you to hide the implementation details of your code, and to specify a preferred interface through which that code can be accessed and used.

Swift has the following access levels:

- **Open and public access** - enable protocols, classes, structs, and enums to be used within any other type from their defining module, and also in a type from another module that imports the defining module.
  Open access applies only to classes and class members, and it differs from public access by allowing code outside the module to subclass and override.
- **Internal access** - enables protocols, classes, structs and enums to be used within any other type from their defining module, but not in any type outside of that module.
- **File-private access** - restricts the use of a type to its own defining source file.
- **Private access** - restricts the use of a type to the enclosing declaration, and to extensions of that declaration that are in the same file.

Application Testing



45. What is unit testing?

Unit testing is a type of testing where individual components of the software are tested.

The purpose of unit testing is to confirm that each unit of the software performs as designed. A unit is a small piece in a codebase, usually a single method in a class, but it can be the class itself.

46. What is the structure of a unit test?

A unit test usually consists of three main actions, also known as *3A-s*:

- **Arrange** objects, create and set them up as necessary.
- **Act** on an object. Call the method you want to test.
- **Assert** (check) that something works as expected.

47. What is code coverage?

Code coverage is a measure used to describe the degree to which the source code of a program is executed when a particular test suite runs. An application with high test coverage, measured as a percentage, has had more of its source

code executed during testing, which suggests it has a lower chance of containing undetected software bugs compared to an application with low test coverage.

48. What is UI testing?

UI tests are tests that make sure that your app's user interface behaves correctly when expected actions are performed.

With UI testing we can find and interact with UI elements, and validate UI properties and state.

Three main classes for UI testing are:

- `XCUIApplication` - Acts as an object you can interact with. It represents an application that can be launched and terminated.
- `XCUIElement` - A UI element in an application
- `XCUIElementQuery` - A query for locating UI elements on the screen.

49. What is Test-Driven Development?

Test-driven development is a software development process where tests are written before the code, the actual implementation. The tests guide the developer to the path of the actual implementation. They drive the implementation, hence the name test-driven development (TDD).

The process of test-driven development follows the following phases:

- Quickly add a test.
- Run all tests and see the new one fail.
- Make a little change.
- Run all tests and see them all succeed.
- Refactor to remove duplication.

TDD process is also known as the Red-Green-Refactor cycle.

50. What is refactoring?

Refactoring is a change made to the internal structure of software to make it easier to understand and cheaper to modify without changing its observable behavior.

Conclusion

Congratulations! You've made it to the end! You are now one step closer to getting that iOS job you want.

Remember to take plenty of sleep before your interview.

Best of luck!