This is CS50x

OpenCourseWare

David J. Malan (https://cs.harvard.edu/malan/)
malan@harvard.edu
(https://www.facebook.com/dmalan) (https://github.com/dmalan) (https://www.instagram.com/davidjmalan/)
(https://www.linkedin.com/in/malan/) (https://www.quora.com/profile/David-J-Malan) (https://twitter.com/davidjmalan)

iOS

iOS development requires that you have (access to) a Mac running a recent version of macOS.

What to Do

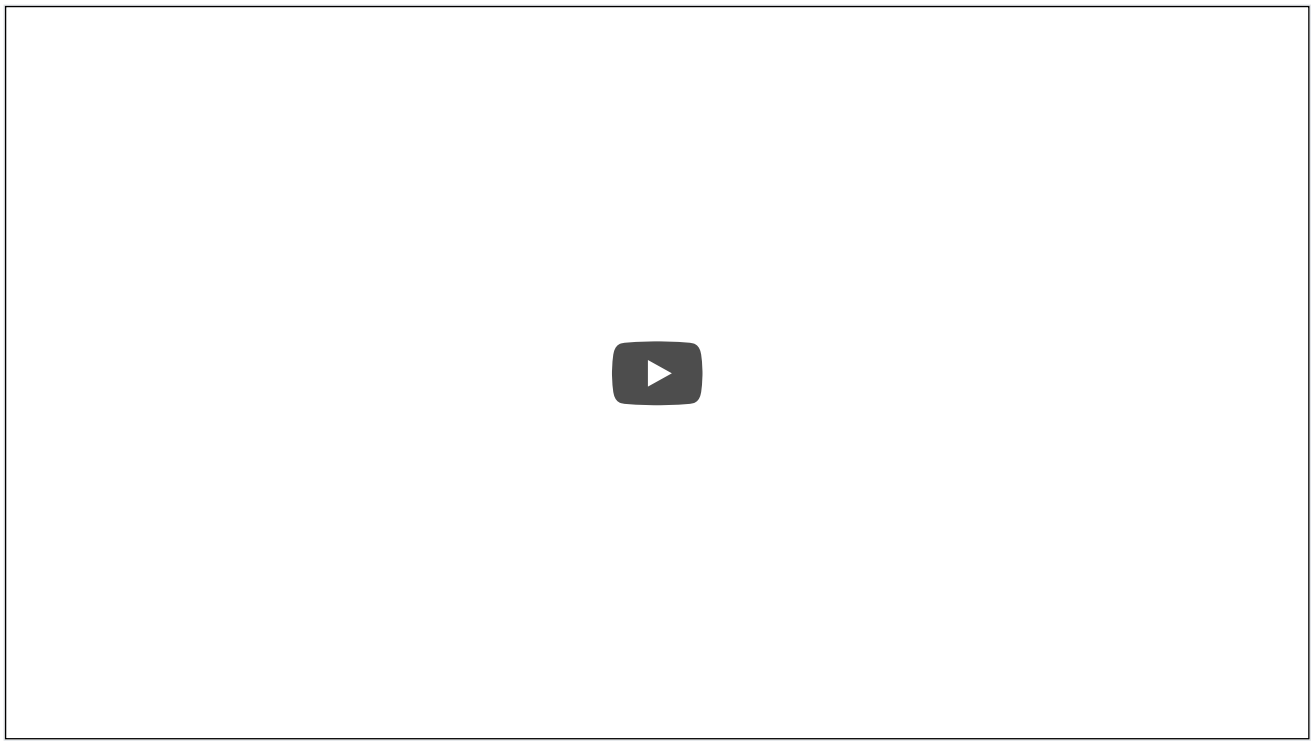
1. After watching Lessons 1, 2, and 3, submit Pokédex.
2. After watching Lesson 4, submit Fiftygram.
3. After watching Lesson 5, submit Notes.

When to Do It

By 11:59pm on 31 December 2020.

How to Do It

▼ **Introduction**



- In this track, we'll be covering how to build apps for Apple's iOS platform, using the programming language Swift. We'll learn by example from three apps, one that reads data from the internet using an API, one that implements image filters, and one that lets you take notes on your phone.

▼ **Lesson 1**

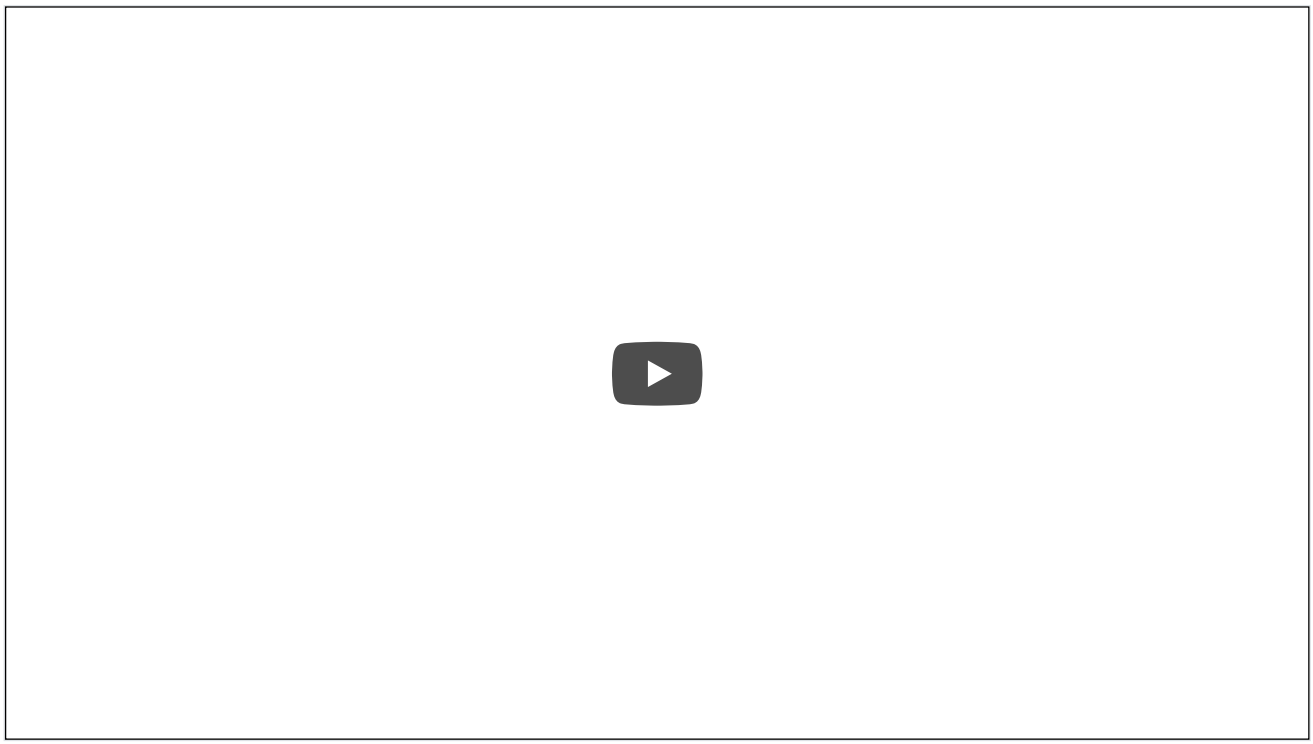
- Xcode is the IDE made by Apple that we'll use to write our apps, and it includes all the tools we'll need, though it is only available for macOS.
- Newer iOS apps are generally written in a language called Swift, though some use an older language, Objective-C.
- [0:37] Swift is similar to C, but there are some differences.
- When declaring variables, we can set them to be constant (immutable) with `let`, or mutable with `var`. This will help protect us from accidentally changing values that should remain constant.
- Data types include:
  - `Array`
  - `Bool`
  - `Dictionary`
  - `Int`
  - `Double`, `Float`
  - `Set`
  - `String`
- We take a look at a few snippets of syntax for conditions, arrays, dictionaries, and string interpolation, or substituting variables in strings.
- [7:23] We see function declarations, like `func sayHello(name: String) -> String { ... }`, where we use the `func` keyword to declare the function name, and then the parameter(s) and the return value. When we call the function, we also need to include the name of the parameter we're specifying, with `sayHello(name: "Tommy")`.
- Structs also exist, acting as a container for data, with multiple fields of different types that we can specify.
- Classes are like structs, but can also have functions. For example, we might have a `class Person` that has a special function, or method, called `init`, to create one instance of the `Person` class. The instances are commonly referred to as an object.
- We can add another method to our class, like `sayHello()`, and after we create an instance of it with `let person = Person(name: "Tommy")`, we can call it with `person.sayHello()`. This time, our function doesn't need a parameter for `name`, since our instance can use the `name` we passed in when we created it.
- [13:34] With inheritance, we can create a subclass of a given class, which will by default include the methods of the class it inherits from. But we can override any of those methods to do something different with `override func`.
- [16:01] Protocols, or interfaces in other languages, are like a list of methods that a class must implement. In a protocol, we don't specify how the function is implemented, but just that it needs to exist on any class using it. This helps our compiler help us be sure that our classes can be used as we would expect.
- [17:53] Optionals, which we can declare with something like `var name: String?`, allow us to declare variables that might have the value of nil, the equivalent of a null value. To use this variable, we need to then check that it's not nil, with `if let n = name` (which we can think of as `if (name != null)`). Then, inside the condition, we'll know that we can use `n` as a string. Again, this helps the compiler help us avoid mistakes. (Though, we can bypass that with using the variable as `name!`, by adding an exclamation point.)
- [21:14] We can also use a guard clause, with the syntax `guard let n = name else`, which sets the variable `n` to `name` if `name` is not nil,

and continues outside of the condition. If `name` *is* nil, then the `else` branch will run, perhaps returning with a special message.

- [23:01] We can open Xcode, and click on "Create a new Xcode project". We'll go to the macOS tab, and choose the "Command Line Tool" template to get started with Swift. We'll have a few details about our project to fill out, and we can use "CS50" as the Organization Name and "edu.harvard.cs50" as the Organization Identifier (the domain name backwards, by convention). (If we wanted to actually publish an app to the App Store, we'll need to get our own account from Apple.) And we'll be sure Swift is the selected language.
- The main screen will be our code editor, and the left side will show the files in our project. We can also change some settings on the right side.
- We can open the file `main.swift`, and see that there's already some generated code for us. In the top left, we can compile and run our code with the triangular play button. In the bottom we have the output of our program, as though we ran it in the terminal.

- [27:34] We can write an example program that sorts students into tracks, with a class, a constructor for the class called `init`, and an array of instances of the class.
- Xcode has autocomplete, so as we type it'll suggest types and highlight anything we might be doing incorrectly.
- We create an array of student names, and a dictionary where the keys are the names of the students (strings) and the values are tracks.
- Then, we iterate through our student array, pick a random track, and store it in our dictionary. Finally, we iterate over our dictionary and print out each student and their assignment.

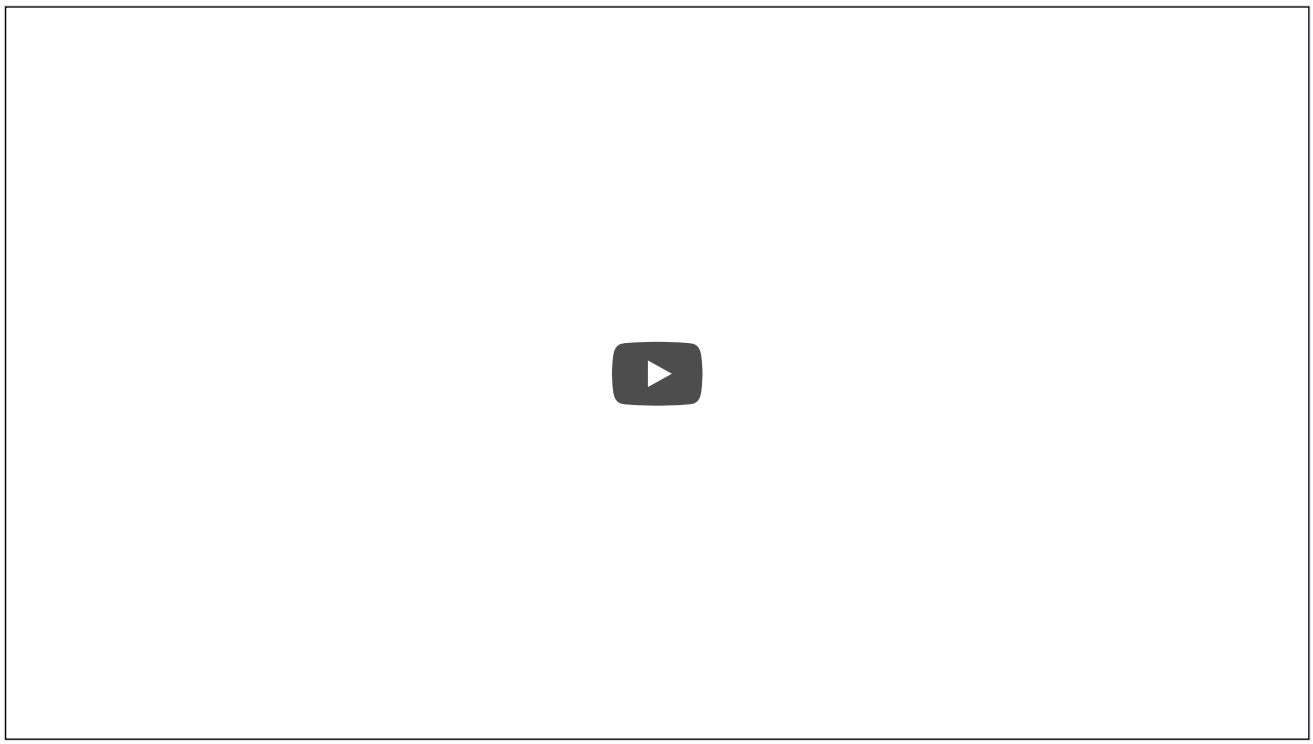
▼ **Lesson 2**



- To write our iOS app, we'll be using a framework called UIKit, made by Apple, with lots of functions to create UI, or user interface, elements on the screen, like tables, images, and buttons.
- We'll follow the MVC pattern, or model-view-controller, where we have models, or code that handles storing our data; views that handle displaying data; and controller that contains our logic for bridging our model and our view. In iOS, they're called view controllers, and decide what gets shown when.
- [2:38] We take a look at table views, which shows a list of contents. Xcode allows us to have visual containers, or storyboards, with which we can create views and indicate how our app moves between them.
- IBOutler and IBAction types will be used for connecting data to views. Segues will allow us to map actions in the app to changes to our views or data.
- [4:35] We open Xcode again, and now we create a project with the "Single View App" template under the iOS tab. We don't need Core Data, Unit Tests, or UI Tests yet, so we can leave them all unchecked.
- We'll build a simple Pokédex app, which will show us a list of Pokémon and more details. We have some generated files:
  - `AppDelegate.swift`, which we can think of as the `main` function for our app, which will decide what happens when the app is started or closed, but we probably won't need to override any of the default behaviors.
  - `ViewController.swift`, which will contain the code for what our app does when a view for our app is loaded.
  - `Main.storyboard`, which shows a preview of an iPhone screen and what our view will look like.
  - `Assets.xcassets` is a directory into which we can put assets like images and sounds.

    - `LaunchScreen.storyboard`, the view that our app will show while it starts.
    - `Info.plist`, a generated configuration file with everything we need for now.
- [12:25] We can start by opening our main storyboard, deleting the blank view controller we see, and adding a "Table View Controller" with the home button icon on the toolbar. We can drag it onto our storyboard, and we'll also need to change our `ViewController.swift` file to match and connect to the controller.
- We make sure that our view controller "Is Initial View Controller" in one of the panels on the right, and we can start by defining our model.
- We'll create a new Swift file and call it `Pokemon.swift`. Inside, we'll declare a `struct Pokemon` with two fields, and in `ViewController` we can create an array of elements of the `Pokemon` type.
- It turns out that we can override methods in our Table View, like `numberOfSections`, `numberOfRowsInSection`, and `cellForRowAt` to display our data in the table.
- [20:50] In the storyboard view, we can find the tab for the Attributes inspector, and apply basic styles.
- Finally, in `cellForRowAt` we get a reusable cell, set the text label, and return it for the table to display.
- [29:20] We can also add a title bar to show where we are, by using the storyboard to place our table view inside a Navigation Controller. Now, our table view can have a title.
- [31:30] We'll add another view by searching for a regular View Controller and dragging it into our storyboard. We can add labels and use the UI to move and resize them.
- [34:20] We'll now need to make another Swift file for this controller, and call it `PokemonViewController.swift`. We'll use an IBOutlet in order to connect labels to variables in our class. First, in our storyboard we need to set the view controller's class to our new class, and use the UI to set each label to the right outlet.
- [38:20] We'll have our table view pass in the `Pokemon` to the `PokemonViewController`, and we can override the `viewDidLoad` function to set the value of the labels based on the Pokemon passed in.
- We'll need to go back to the storyboard and create a Segue for our table view to transition to the new view controller. We'll pick "Show" as the Segue type, and set an identifier we can use from our view controller.
- We'll override `prepare(for segue)` and use the `if let destination = segue.destination as? PokemonViewController` syntax to make sure that the view we're going to is a `PokemonViewController`. Then, we can get the right Pokemon for the row, and set it on the view directly.
- [47:30] We can run our app and see the views as we expect now, and to format the number of the Pokemon we can use `String(format: "#%03d", pokemon.number)`.

▼ **Lesson 3**



- We can use an API, application programming interface, to load data from the internet in our app. An API is like a set of code that someone else has written, designed for you to use too.
- In this case, we'll be making requests to a website and getting data back in a format called JSON, JavaScript Object Notation.
- An object in JSON might look like a dictionary of key-value pairs:

```
{
```

```
  "course": "cs50",
  "tracks": ["mobile", "web", "games"],
  "year": 2019,
}
```

- The values can be a string, an array, or a number as we see here, or some other data types.

- [2:12] We'll use a new pattern in Swift called try, catch, where any exception, or error, can be "caught":

```
do {
    let result = try something()
}
catch let error {
    print (error)
}
```

- In this case, the `something()` function might cause an exception, so we want to `try` running it, and `catch` any error that does occur.

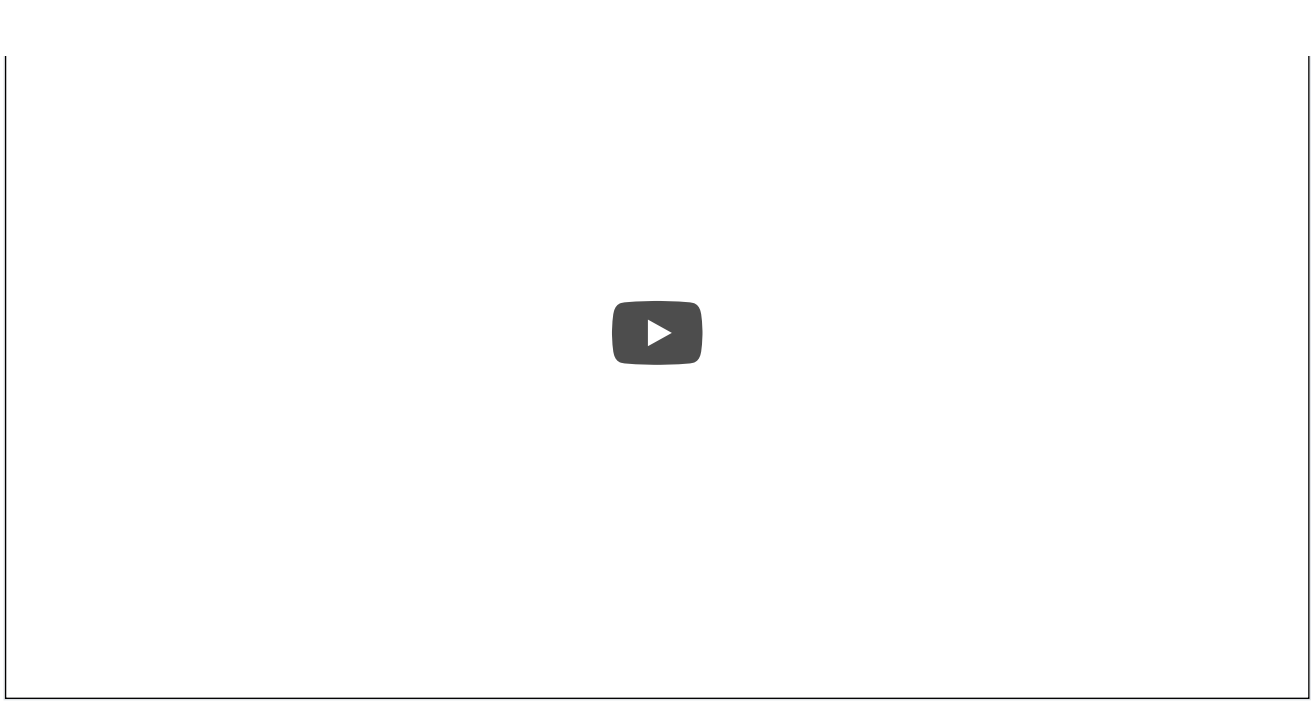
- [3:15] A closure, or anonymous function, will also be helpful:

```
let reversed = names.sorted(by: {(
    s1: String,
    s2: String
) -> Bool in
    return s1 > s2
})
```

- We pass in a closure to the `sorted` function on the `names` array, which is a function that takes in two strings and returns a Boolean, acting as a comparison function between them.

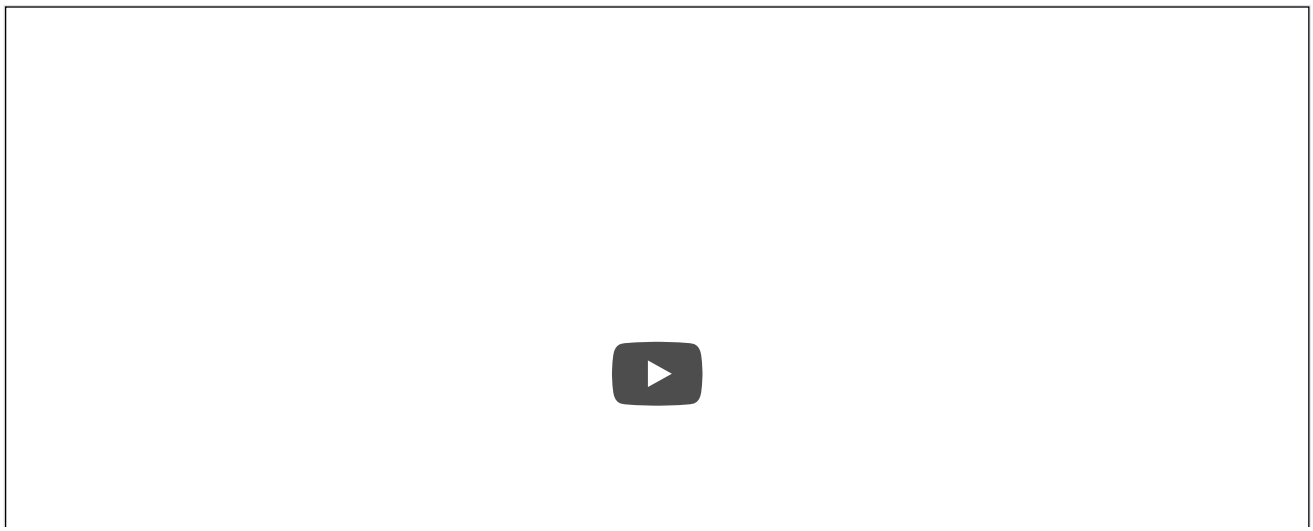
- [5:22] We'll also see a pattern called delegates, where we can attach event handlers, so a change to one object can "notify" another one to respond; the logic to handle a change is "delegated" to somewhere else.
- [6:01] We'll use https://pokeapi.co to get our data about Pokemon, in JSON format.
- In Xcode, we'll populate our array of `pokemon` in the `viewDidLoad` function of our `ViewController`.
- We can read the documentation of the API and figure out the URL format we need to use. Then, we can use the `URL` and `URlSession` class in Swift to actually get the data.
- [13:10] We'll use a closure to handle the data (or error) we get back from our request, along with guards to make sure the types aren't optional.
- [15:18] We'll also need a struct to represent the data we get back, like `PokemonList`. With the JSONDecoder class, we can decode the data into a struct we define. To make our struct decodable, we also have to specify it is codable (convertable to and from JSON), so we use `struct PokemonList: Codable` in our definition. We also need to wrap this decoding function with a try, catch, since we're not guaranteed that the data we get back will successfully be decoded.
- [19:05] We try to run our app, and see an error about a key not found. It turns out the API doesn't return a `number` field, that's required by our `Pokemon` struct.
- [20:55] To display the data we get back, we can set our table's data to the results of our decoded data, using the keyword `self` in a closure to be clear which variable we're setting.
- [22:50] And now that we've changed the data for the table, we have to tell the table to reload the view with `self.tableView.reloadData()`, but we actually have to ask the main thread (the foreground of the app) to reload the data, since our URL request is in a background thread.
- [25:40] We can add a `capitalize` function to change the name of the Pokemon before we display it in our table.
- [28:23] Let's use the API in our second view, for each individual Pokemon. We can explore the API to see that a lot of information is returned. It turns out that one field, `types`, is an array with one or more `type` objects. We'll create a struct to represent this data as a whole, `PokemonData`, and add the fields we want to decode, with `PokemonTypeEntry` and `PokemonType` structs as needed.
- [32:20] Now we can make a second API call in our `PokemonViewController`, changing the URL based on the Pokemon passed in, decoding the data, and setting the values of the labels in the view. We also need to be careful to clear placeholder data in our labels, since we might not have values to set them to each time.

▼ Lesson 4



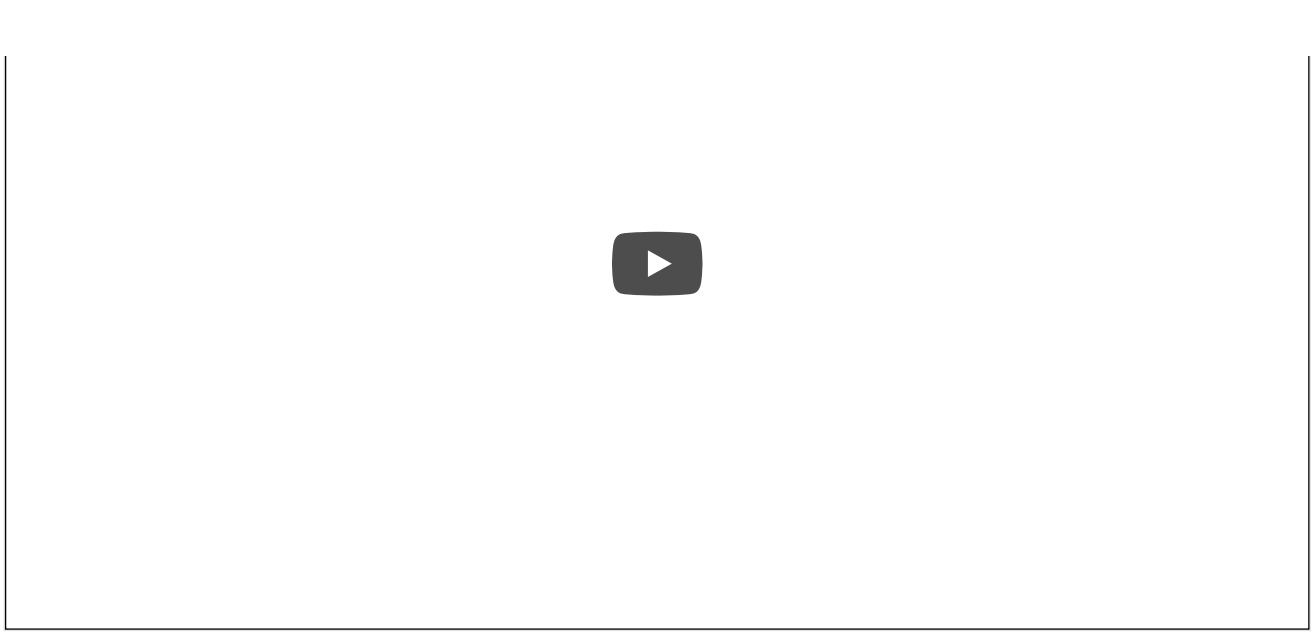
- We'll make a different app this time, one that can apply filters to photos.
- We'll create a new Single View App project in Xcode, and add some basic components to our main storyboard. We'll add:
  - a navigation controller so our app has a title
  - a Bar Item that looks like a button in the navigation bar, labeled "Choose Photo"
  - an Image View to show an image
  - buttons for each of the filters that we'll want
- [5:25] First, we want to be able to pick a photo and show it. We'll need an `IBAction` to link to "Choose Photo", an `IBOutlet` for the Image View, and connect both using the storyboard UI in Xcode by clicking and dragging.
- [7:35] To show the user's photo gallery for them to select a photo, we'll use the `UIImagePickerController` class and make sure that our class will be the delegate, or the class that will be responding to the user selecting a photo in that controller class.
- [9:55] To display the image picker, we'll need to use the Navigation Controller to `present` it, and use the `didFinish` function that the ImagePickerController will call for us. By looking through some documentation or examples, we'll need to get a `UIImage` from what's passed in to the function, and set it on the Image View. We also need to dismiss the picker after a photo is selected.
- [15:10] Now we can write some of the code to apply filters to the image after it's been selected. We can look at the documentation for Core Image Filter from Apple, and notice that we have many built-in options like `CISepiaTone` .
- [16:20] We'll create another `IBAction` , connect it in the storyboard, and use the `CIFilter` class to create an object we can use to filter our image. We need to convert the classes of images we have with a little extra code to guard the types.
- [22:30] We can save the original image so the filter doesn't keep getting reapplied, and be careful to add a guard to make sure it exists before we try applying a filter.
- [24:20] We'll add a few more buttons for other filters, and create and connect `IBAction` s for them too. Since we're now repeating some of this code, we'll factor the common lines out into a helper method. Now, all three of our filters work.

▼ Lesson 5



- SQLite is a simple SQL database where our Swift code can write queries and save data.
- We'll need queries like:
  - `CREATE TABLE`
  - `INSERT INTO`
  - `SELECT ... FROM`
  - `UPDATE ... SET`
- [3:50] We'll make a note-taking app as our example, and start by creating a Single View App in Xcode as before. We'll want a Table View Controller to show a list of notes.
- [6:20] Now let's define our model with a struct `Note`, and show a list of those in our table by overriding the functions on our ViewController.
- [10:00] We'll define a new class, `NoteManager`, to work with the database. With SQLite, we can save and open our database like a file, so we'll use the built-in `FileManager` to open or create a file called `notes.sqlite3` in a function called `connect`.
- [13:15] We need a variable of type `OpaquePointer`, which is just a pointer to our database file. This way, `sqlite3_open` can keep the same connection to the database.
- [15:40] We'll also create our table if it doesn't exists, as we open the database file.
- [17:20] To create a new note, we'll have a `create` function that first tries to `connect` (in case we haven't already) and then `sqlite3_prepare_v2` to prepare a statement. We'll then use `sqlite3_step` to run the next step of the statement, and `sqlite3_finalize` to finish executing it.
- [22:00] We'll have our `create` function also return the ID of the row we just created, in case we want to do something else with it right after.
- [22:55] Next, we'll write `getAllNotes` to return a list of the `Note`s we have saved, by preparing a `SELECT` statement. Since we want to run this query for each row we can get back, we'll have to use a `while` loop to run each step of our statement until there are no more rows. Inside our loop, we'll use `sqlite3_column_int` or `sqlite3_column_text` to get the values of each column from the last row from our statement, and build a `result` array of `Note`s to return.
- [28:05] We'll add a Bar Item button and link it to an action to create a new note. In our view controller, we'll create a `NoteManager`. We'll add a `main` reference in `NoteManager` to an instance of itself (a singleton, since there will ever be one of them that we need), and make sure that `init` is marked as `private` so we won't have multiple `NoteManager`s.
- [31:15] Now, our action can create a new note in the database, but we'll need a new method to reload data from the database and refresh our table view.
- [33:40] When we try to run our app, we couldn't create the database, and it turns out we need to use `documentDirectory` as the directory for our database file. But after creating a note and stopping and starting our app again, we don't see the note saved. We need to load the notes when the table view is loaded for the first time, too, not just when we add a new note.
- [36:10] We also need a view to show and edit individual notes, so we'll add a second View Controller and a segue from our table view controller to it. We'll add a text view, and it turns out that our storyboard has little buttons that allow us to add constraints, or rules for where the text view can expand to, on the screen.
- [39:20] Now we'll need our model to save any changes to the note, so we'll write a `save` method. We'll prepare a statement with `UPDATE`, and to pass data into our query we'll use `?`s as placeholders, and bind data to it with `sqlite3_bind_text` or `sqlite3_bind_int`. This will help protect us from SQL injection attacks.
- [43:40] In our table view controller, we'll use `prepare(for segue...` to pass a selected `Note` to our note view controller. We'll create an outlet for the text view (to display the contents for the note) and save the note automatically when the user leaves the view, by overriding `viewWillDisappear`. We'll also need to reload the data in our table view after we come back to it, so we'll override `viewWillAppear` in it.
- [48:20] We don't see our note being updated in the app after we change it, so we look at our note view controller. It turns out, that even though we save the `Note` correctly, we're not updating that based on the text view. So we'll set the struct's `contents` to the text in the text view.

▼ Conclusion



- Apple's iOS developer documentation will have lots of topics and examples that we can use to build apps on top of what we've learned so far.