ATLS 4320: Advanced Mobile Application Development
Week 2: Tab Bar Controllers and Pickers

Tab Bar Controllers

https://developer.apple.com/design/human-interface-guidelines/ios/bars/tab-bars/
The tab bar controller provides navigation for different sections of an app. Provides access to several peer information categories in a flat hierarchy.
Tab bars should be used for navigation, not to perform actions (those are tool bars)
Guideline is to have 3-5 tabs on an iPhone (5 are shown in horizontal compact), a few more are acceptable on iPad.

- Too few tabs make the interface feel disconnected
- When there are more tabs than can be shown the tab bar controller will automatically display a "More" tab with the rest of the items.
    - This requires additional taps, is a poor use of space, and should be avoided.

The tab bar controller is the root view controller in an app.
Use when you want to present different perspectives of related data (music, clock, phone apps)
Data is not passed between views in a tab bar, it is just a way to organize categories of information.
UITabBarController class https://developer.apple.com/documentation/uikit/uitabbarcontroller

A delegate that adopts the UITabBarControllerDelegate protocol handles tab bar interactions.

Each tab bar item is associated with a view controller which controls a view.
UITabBarItem class https://developer.apple.com/documentation/uikit/uitabbaritem

Pickers

https://developer.apple.com/design/human-interface-guidelines/ios/controls/pickers/
A picker is a scrollable list of distinct values (replaces a select list on the web)

- slot-machine looking wheels that are used when you have a list of values
- can have multiple components (columns) that are independent or dependent

UIPickerView class https://developer.apple.com/documentation/uikit/uipickerview
Use a picker for medium length lists of items, ideally when users are familiar with the entire set of values. Because many of the values are hidden when the wheel is stationary, it's best when users can predict what the values are.
For short lists of choices, iOS14 added menus that can be accessed from any button.
For longer lists with values that aren't well known to your users, use a table view.
Pickers don't hold any data themselves. They call methods on their data source and delegate to get the data they need to display.

The UIPickerViewDataSource protocol handles the data for the picker view and must be adopted
https://developer.apple.com/documentation/uikit/uipickerviewdatasource

The UIPickerViewDelegate protocol handles the view content and must be adopted
https://developer.apple.com/documentation/uikit/uipickerviewdelegate

iOS14 introduces a new Date Picker and Color picker.

<u>Date Picker</u>

The new date picker lets the user select a date from a calendar view. Time is entered using the number keypad.
The date picker can be displayed in 4 different styles:

- inline – an editable field that fits in small spaces, expands to editing view
- compact – a label that expands to editing view
- wheels - like the standard picker
- automatic – system uses the platform and mode to determine the display style

Color Picker
https://developer.apple.com/design/human-interface-guidelines/ios/controls/color-wells/
The color well displays a color picker with an interface that provides 4 ways for a user to select a color. They can use a grid or spectrum, select RGB values, or select a color from the screen.

Music
Create a new project and make sure iOS is selected at the top.
Select App (many of the previous templates such as Single View App are no longer in Xcode 12)
Product Name: music
Team: None
Org identifier: ATLAS (can be anything, will be used in the bundle identifier)
Interface: Storyboard
Life Cycle: UIKit App Delegate
Language: Swift
Uncheck core data and include tests.
Uncheck create local git repo

Many of the changes in Xcode 12 are around Apple's effort for multiplatform apps (ios, MacOS, etc).

If you go into the storyboard you'll notice a "minimap" at the top right. It will provide an overview of everything in your canvas, which is helpful when there are many views on your canvas.
You can turn it off/on using Editor | Canvas |Minimap or through the editor options button.
You'll also notice that some of the inspector icons have changed (attributes, size, and connections).

Tab Bar Controller
The easiest way to add a tab bar controller is to select the view controller created and then Embed In Tab Bar Controller (through Embed In icon or the Editor menu).
This creates a tab bar controller and makes the view controller the first tab.
Click on the tab bar controller. In the identity inspector you'll see that the class is UITabBarController.
In the attributes inspector you'll see that Is Initial View Controller is checked. And in the Connections inspector you'll see that segue is already set up to the other view controllers.
Click on the view controller and look at the identity inspector. It's already set up to use the class ViewController and the swift file was created for us.
In the Connections inspector note that a relationship segue has already been set up from the tab bar controller.
You'll also notice that the tab bar item is present with a title of Item. For the time being change it to "First". We'll add an image later.
Just for testing, add a label that says "First" just so we can test the tab bar controller.

Now let's add a second tab to our tab bar.
From the Library (+) drag a View Controller onto the canvas.

To connect the tab bar controller to the new view controller cntrl-click and drag from the tab bar controller to the new view controller and choose Relationship segue view controllers. This will add a second tab bar button that is hooked up to the new view controller.
Change the title of the tab bar item to "Second".
Just for testing, add a label that says "Second".

Now we need a new class to control this view.
File | New | File
iOS Source | Cocoa Touch class
Call it SecondViewController and make sure it's a subclass of UIViewController.
Uncheck Also create xib file and make sure the language is Swift.
Make sure it's being saved into your project folder and the music target is checked
This should create SecondViewController.swift

In the Storyboard click on the second ViewController and in the identity inspector change its class to SecondViewController.

Go ahead and run the app and you'll see that the tab bar is working and load the views for each tab.

Single component Picker
In our first view we're going to add a single component picker where you choose a genre of music.
Add a picker view from the library (ignore the default data it shows)
Move the label above the picker and update the text to Music Genre.
Then add another label below the picker that we'll use for output.

Create outlet connections for the picker and the 2nd label called musicPicker and choiceLabel. Make sure you're connecting to ViewController.swift.

With the picker selected go into the connections inspector and connect the dataSource and delegate to the view controller by clicking in the circle and dragging the connection to View Controller (white square in a yellow circle icon on top of the view). Now this picker knows that the ViewController class is its data source and delegate, and will ask it to supply the data to be displayed. (can also do this programmatically in viewDidLoad).

Go into ViewController .swift and add the protocol we're going to implement
```
class ViewController: UIViewController, UIPickerViewDelegate,
UIPickerViewDataSource
```
You will have an error that ViewController does not conform to UIPickerViewDataSource until we implement the required methods for the protocol. You can click Fix for the stub methods to be added.
Look at the delegate classes.

Define an array that will hold the music genres.
```
let genre = ["Country", "Pop", "Rock", "Classical", "Alternative", "Hip
Hop", "Jazz"]
```

Now let's implement the methods needed to implement the picker.
```
//Methods to implement the picker
//Required for the UIPickerViewDataSource protocol
func numberOfComponents(in pickerView: UIPickerView) -> Int {
        return 1 //number of components
```

```
}

//Required for the UIPickerViewDataSource protocol
func pickerView(_ pickerView: UIPickerView, numberOfRowsInComponent
component: Int) -> Int {
        return genre.count //number of rows of data
}
```

The delegate methods are optional, but you need to implement at least one and it's often this one.

```
//Picker Delegate methods
//Returns the title for a given row
func pickerView(_ pickerView: UIPickerView, titleForRow row: Int,
forComponent component: Int) -> String? {
    return genre[row]
}

//Called when a row is selected
func pickerView(_ pickerView: UIPickerView, didSelectRow row: Int,
inComponent component: Int) {
    choiceLabel.text="You like \(genre[row])" //writes the string with the
row's content to the label
}
```

Run it in the simulator
Click and drag your mouse in the simulator to move the picker.
Use auto layout so the view looks good in portrait and landscape.
Use autoresizing feature or add constraints (can't use both in a view).
(I embedded all the views in a stack view with spacing of 50. Then I then aligned the stack view horizontally and added a top constraint.)
Assistant editor – use the jump bar to access preview mode.

Multi-component picker
We'll use the second tab for a 2-component independent picker.

In Main.storyboard add a picker view from the library.
Move the label above the picker and update the text to Decades of Music.
Then add another label below the picker that we'll use for output.

Create the same outlet connections for the picker and the 2nd label called musicPicker and choiceLabel.
Make sure you're connecting to SecondViewController.swift.
The assistant editor will automatically open the swift file with the class assigned in the identity inspector to the scene selected in the storyboard. It will also only let you make connections to its associated class.

With the picker selected go into the connections inspector and connect the dataSource and delegate to the view controller by clicking in the circle and dragging the connection to View Controller.

Add needed constraints.

Go into SecondViewController .swift and add the protocol we're going to implement

```
class SecondViewController: UIViewController, UIPickerViewDelegate,
UIPickerViewDataSource
```

Add the required protocol stub methods.

We're going to add a 2nd component that lists the decade for music.

Add the data arrays.

```
let genre = ["Country", "Pop", "Rock", "Classical", "Alternative", "Hip
Hop", "Jazz"]
let decade = ["1950s", "1960s", "1970s", "1980s", "1990s", "2000s", "2010s",
"2020s"]
```

Genre will be component 0
Decade will be component 1

Most of the changes will be in the data source methods.
This time we're going to return 2 in `pickerView(_:numberOfRowsInComponent:)` since we have 2 components.

And now we have to check with component is picked before we can return the row count.

```
func pickerView(_ pickerView: UIPickerView, numberOfRowsInComponent
component: Int) -> Int {
        if component==0 {
            return genre.count
        } else {
            return decade.count
        }
}
```

Then in our delegate method we also have to check which component was picked before we can return the value.

```
func pickerView(_ pickerView: UIPickerView, numberOfRowsInComponent
component: Int) -> Int {
        if component==0 {
            return genre[row]
        }
        else {
            return decade[row]
        }
}
```

Don't forget the delegate method to draw the title for each row

```
    //Picker Delegate methods
    //returns the title for the row
    func pickerView(_ pickerView: UIPickerView, titleForRow row: Int,
forComponent component: Int) -> String? {
        if component == 0 {
            return genre[row]
        } else {
            return decade[row]
        }
    }
```

Now we have to use both components when printing out our results.

```
func pickerView(_ pickerView: UIPickerView, didSelectRow row: Int,
inComponent component: Int) {
    let genrerow = pickerView.selectedRow(inComponent: 0) //gets the
selected row for the genre
    let decaderow = pickerView.selectedRow(inComponent: 1) //gets the
selected row for the decade
    choiceLabel.text="You like \(genre[genrerow]) from the
\(decade[decaderow])"
}
```

If you hover over code such as an if statement and click the command key the editor will show you the code block. A click then gives you code options.

Property Lists

Few applications use arrays with data hard coded so this time we're going to use the data stored in a property list. A property list is a simple data file in XML that stores item types and values. They use a key to retrieve the value, just as a Dictionary does. Property lists can have Boolean, Data, Date, Number, and String node types to hold individual pieces of data, as well as Arrays or Dictionaries to store collections of nodes.

All of our apps have an Info.plist file in Supporting Files.

- Use the Bundle (previously NSBundle) class to access the plist
    - A Bundle object represents a location in the file system that groups code and resources that can be used in a program
    - We can use a bundle is to get access to the files in our app
- Property lists can be created using a property list editor, directly in Xcode, or in any text editor
- You can use the PropertyListDecoder class to decode data from a plist and assign the data to an array or dictionaries (depending on plist structure).

<u>Dependent multi-component picker</u>

Now let's add a third tab to our tab bar. In this tab we're going to have a 2 component picker but this time the components are going to be dependent on each other.

Go back into Main.Storyboard and drag a View Controller from the object library onto the canvas.

Make the connection from the tab bar controller to the new view controller by cntrl-click and drag from the tab bar controller to the new view controller and choose Relationship segue view controllers. This will add a third tab bar button that is hooked up to the new view controller.

Update the tab bar item title to be Third.

If you ever want to switch the order of the tabs you can just drag them to change their order. The connections remain intact so that's not affected.

Now we need a new class to control this view.

File | New | File

iOS Source | Cocoa Touch class

Call it ThirdViewController and make sure it's a subclass of UIViewController.

Uncheck Also create xib file and make sure the language is Swift.

Make sure it's being saved into your project folder and the music target is checked

This should create ThirdViewController.swift

In the Storyboard click on the third ViewController and in the identity inspector change its class to ThirdViewController.

Go into the Storyboard and create a similar interface as before - a picker and two labels (one that says Artist Picker and another label that's empty that we'll use for output).
Create outlet connections for the picker and the 2nd label called artistPicker and choiceLabel. Make sure you're making the connections to ThirdViewController.
With the picker selected go into the connections inspector and connect the dataSource and delegate to the View Controller icon.
Added needed constraints.

Grab the artistalbums2 plist from my github repo and drag it into your project.
Make sure you choose Copy items if needed and that the project target is checked.
Look at the plist. Right click Open As | Source Code shows the xml.
Note that it's an array of dictionaries, with:

- key name, value string
- key albums, value array of strings.

Now let's add a model class to represent this data.

File | New | File | Swift File
ArtistAlbums
Make sure the music target is checked.

We'll create a struct to represent this data model.
Classes and structures in Swift have many things in common. Both can:

- Define properties to store values
- Define methods to provide functionality
- Define subscripts to provide access to their values using subscript syntax
- Define initializers to set up their initial state
- Be extended to expand their functionality beyond a default implementation
- Conform to protocols to provide standard functionality of a certain kind

Do you remember one thing that classes can do that structs can not?

Classes have additional capabilities that structures do not:

- Inheritance enables one class to inherit the characteristics of another.
- Type casting enables you to check and interpret the type of a class instance at runtime.
- Deinitializers enable an instance of a class to free up any resources it has assigned.
- Reference counting allows more than one reference to a class instance.

```
struct ArtistAlbums: Decodable {
    let name : String
    let albums : [String]
}
```

Codable
https://developer.apple.com/documentation/foundation/archives_and_serialization/encoding_and_decoding_custom_types
Starting in Swift 4.1 the ability to encode and decode custom types became much easier through the Encodable and Decodable protocols (and the Codable type alias for both). By adopting these protocols

for your custom types you can implement the Encoder and Decoder protocols which will encode or decode your data to and from an external representation such as JSON or property list.
The property names in your custom type MUST match the property names in your data. In the case of a plist the names of the keys must be the same as the names of the properties in your struct for the decoding to work automatically. (you can use codingkeys if the names don't match).

Swift 4 made it much easier to convert external data such as JSON and plists into internal representations of the data using the Encodable and Decodable protocols. These protocols will let you automatically encode and decode data into class or struct instances of your custom types. The property names in your custom type MUST match the property names in your data. In the case of a plist the names of the keys must be the same as the names of the properties in your struct for the decoding to work automatically. (you can use codingkeys if the names don't match).

To support encoding and decoding you need to adopt the Encodable and Decodable protocols, or the Codable typealias, which combines the Encodable and Decodable protocols. This process is known as making your types codable.

- Types that are Codable include standard library types like String, Int, and Double; and Foundation types like Date, Data, and URL. Array, Dictionary, and Optional also conform to Codable when they contain codable types.
- Adopt Codable to the inheritance list of your class/struct (: Codable)
- The class/struct property names MUST match the key names of the items in the property list/JSON file
- Use the PropertyListEncoder and PropertyListDecoder for plists
    - decode(_:from:) will decode the data from the file. It must match the structure of the type
    - encode(_:) creates a property list of the value you pass it

It's best practice to keep anything that is not directly related to controlling the view out of the view controller so we're going to create a class that will handle loading the data from the plist. This also decouples the data from the view controller, which is good practice. We could change this class so it's loading JSON data and it wouldn't change the view controller code at all. It also makes it easier to test.

File | New | File | Swift File
DataLoader
Make sure the music target is checked.

We'll create a class with an array of ArtistAlbums to load our plist into and a method to load the data from our plist.

```
class DataLoader {
    var allData = [ArtistAlbums]()

    func loadData(){
        if let pathURL = Bundle.main.url(forResource: fileName,
withExtension: "plist"){
            //initialize a property list decoder object
            let plistdecoder = PropertyListDecoder()
            do {
                let data = try Data(contentsOf: pathURL)
                //decodes the property list
                allData = try plistdecoder.decode([ArtistAlbums].self, from:
```

```
data)
            } catch {
                // handle error
                print("Cannot load data")
            }
        }
    }
}
```

The decode(_:from:) method is a key step. It uses the structure of the ArtistAlbums struct to decode the plist. The property names of the struct must match the keys in the dictionary in the data file for this to work using the property list decoder.
https://developer.apple.com/documentation/foundation/propertylistdecoder/2895397-decode

We'll also want to be able to ask for all the artists for the first picker component and for all the albums for a given artist. So we'll create two methods that return this data.

```
    func getArtists() -> [String]{
        var artists = [String]()
        for artist in allData {
            artists.append(artist.name)
        }
        return artists
    }

    func getAlbums(index:Int) -> [String] {
        return allData[index].albums
    }
```

Go into ThirdViewController.swift and add the protocols and the required methods like last time.

```
class ThirdViewController: UIViewController, UIPickerViewDataSource,
UIPickerViewDelegate
```

Now we're going to define the following at the class level: an instance of DataLoder that we'll use to access our data, 2 arrays to hold the artists names and album names, 2 constants for the component numbers since we'll be using them a lot. and a constant with the filename.

```
    var musicData = DataLoader()
     var artists = [String]()
    var albums = [String]()
    let artistComponent = 0
    let albumComponent = 1
    let filename = "artistalbums2"
```

In viewDidLoad we'll use our instance of ArtistAlbumsController to load the data, get the array of artists, and get the first artists list of albums.

```
    override func viewDidLoad() {
        super.viewDidLoad()
        musicData.loadData(filename: filename))
        artists= musicData.getArtists()
        albums= musicData.getAlbums(index: 0)
    }
```

The next 3 methods haven't changed much, we're just using the constants we defined.

```
//Required for the UIPickerViewDataSource protocol
    func numberOfComponents(in pickerView: UIPickerView) -> Int {
        return 2
    }

    func pickerView(_ pickerView: UIPickerView, numberOfRowsInComponent
component: Int) -> Int {
        if component == artistComponent {
        return artists.count
    } else {
        return albums.count
        }
    }


//Picker Delegate methods
//Returns the title for a given row
    func pickerView(_ pickerView: UIPickerView, titleForRow row: Int,
forComponent component: Int) -> String? {
        //checks which component was picked and returns the value for the
requested component
        if component==artistComponent {
            return artists[row]
        }
        else {
            return albums[row]
        }
    }
```

We need to change this method so when the artist is changed the list of albums changes as well.

```
//Called when a row is selected
    func pickerView(_ pickerView: UIPickerView, didSelectRow row: Int,
inComponent component: Int) {
        //checks which component was picked
        if component == artistComponent {
            albums = musicData.getAlbums(index: row) //gets the albums for the
selected artist
            artistPicker.reloadComponent(albumComponent) //reload the album
component
            artistPicker.selectRow(0, inComponent: albumComponent, animated:
true) //set the album component back to 0
        }
        let artistrow = pickerView.selectedRow(inComponent: artistComponent)
//gets the selected row for the artist
        let albumrow = pickerView.selectedRow(inComponent: albumComponent)
//gets the selected row for the album
        choiceLabel.text = "You like \(albums[albumrow]) by
\(artists[artistrow])"
    }
```

If you find your text being cut off in the label, set the attribute Autoshrink to Minimum font scale or size so it will automatically adjust.

Some suggest further modularization and separating out the datasource and delegate methods since those handle the data and don't control the view but for simplicity I left them in the view controller.

Icons

In the tab bar controller if you select the tab bar you'll see it has many properties in the attributes inspector. The bar tint default seems to be yellow. I just changed it to the system background color.
For each tab you can customize the text and image by selecting the tab bar item in each view controller.
You can use a standard image or add a custom one

- About 25x25 pixels for 1x (max 48x32)
- png format
- Colors are ignored, alpha values from 0 (completely invisible) to 1 (completely visible) are used.
- Different versions for unselected and selected (filled in)

Let's set the tab bar name and image. Drag the png files into Assets.xcassets that you want to use. Then in the Storyboard click on the tab bar item for the first scene and in the attributes inspector you can change its title and image, or remove the title if you just want an image. (31-ipod, 65-note, 66-microphone, 120-headphones, 194-note-2) (you can download glyphish icons to use)
Ideally the images you use should be descriptive, otherwise use a title as well.
For app icons you need 120x120 for the 2x and 180x180 for 3x.
You should also set up a launch screen and its constraints.

App access

Now let's add a fourth tab to our tab bar. In this tab we're going to access other apps.
Add a new view controller into your storyboard and connect the tab bar controller with a relationship segue.
Then add a new Cocoa Touch class for your new view controller called FourthViewController and make sure it's a subclass of UIViewController.
This should create FourthViewController.swift
In the Storyboard click on the fourth ViewController and in the identity inspector change its class to FourthViewController.

(these steps are the same as what we did when we added the third view controller)

Now let's get our fourth view set up.
Let's make this a view where a button will take the user to Spotify, the iTunes music library or if that's not on their device (simulator) then it opens itunes in Safari.
And add a button that says Listen and connect it as an action to a method called gotomusic.
Since this is the fourth scene we must connect to FourthViewController.swift.
Add Missing Constraints for all views in the fourth view controller and fix as needed.
Change the tab bar button image (ipod)

Now let's go into FourthViewController.swift and implement gotomusic().

```
    @IBAction func gotomusic(_ sender: UIButton) {
        //check to see if there's an app installed to handle this URL scheme
        if(UIApplication.shared.canOpenURL(URL(string: "spotify://")!)){
            //open the app with this URL scheme
            UIApplication.shared.open(URL(string: "spotify://")!, options:
[:], completionHandler: nil)
        }else {
```

```
            if(UIApplication.shared.canOpenURL(URL(string: "music://")!)){
                UIApplication.shared.open(URL(string: "music://")!, options:
[:], completionHandler: nil)
            } else {
                UIApplication.shared.open(URL(string:
"http://www.apple.com/music/")!, options: [:], completionHandler: nil)
            }
        }
    }
```

For a Swift app, iOS creates a UIApplication object to set up the app's runtime environment - its use of the display, its ability to handle touches and rotation events, etc. This object is also how we can interact with the rest of the iOS system. We're using the shared.canOpenURL(_:) method to see if there's an app available to handle that string (returns a Boolean). I'm not passing any options so I'm passing an empty dictionary [:].

UIApplication also has the method shared.open(_: options: completionHandler:), which will launch other applications and have them deal with the URL.
mailto:, facetime:, and tel: open the Mail, FaceTime, and Phone apps, respectively.

UIApplication also has a UIApplicationDelegate object, which is informed of major life-cycle events that affect the app. This is the AppDelegate class that Xcode gave us to start with. When all the app setup is done, the application(_: didFinishLaunchingWithOptions:) method gets called. From our point of view, this is where the app "starts," although a bunch of stuff has already been done for us by this point.

Starting in iOS9 you have to declare any URL schemes of non-Apple apps you want your app to be able to call canOpenURL(_:). This does not affect open(_: options: completionHandler:).
If you forget this step you'll get the error "`canOpenURL: failed for URL: "spotify://" - error: "This app is not allowed to query for scheme spotify`"
(you will also get this error in the simulator without spotify)

Go into Info.plist and add LSApplicationQueriesSchemes as an Array. Click on the arrow to the left of LSApplicationQueriesSchemes to open it and then hit the + to add an item to the array called "spotify".

Audio
The iOS SDK has multiple multimedia frameworks to access iOS's audio capabilities.
Frameworks for audio from easiest to use to harder

- System Sound Services - plays user-interface sound effects, or to invoke vibration
    - supports caf, aif, or wav formats and must be less than 30 secs.
- Media Player framework - plays songs, audio books, or audio podcasts from a user's iPod library.
- AVFoundation framework - plays and records audio
    - We'll be using this to record and play back sounds in our app
- Audio Toolbox framework - plays audio with synchronization capabilities, access packets of incoming audio, parse audio streams, convert audio formats, and record audio with access to individual packets.
- Audio Unit Framework - connect to and use audio processing plug-ins
- OpenAL framework - provides positional audio playback and lets you mix sounds

- Best choice for games
- OpenAL gives you more control of audio but is more complicated.

The **AVFoundation** framework
The **AVAudioSession** class acts as an intermediary between your app and the system's media services
https://developer.apple.com/documentation/avfoundation/avaudiosession

- Configure your audio session
  - Configure audio settings such as sample rate, I/O buffer duration, and number of channels
  - Handle audio route changes
    - Events such as a phone call
    - Audio use by another app
  - Audio session category
    - how your audio session interacts with others

The **AVAudioPlayer** class provides playback of audio data from a file or memory
https://developer.apple.com/documentation/avfoundation/avaudioplayer

- Play sounds of any duration from files or memory
- Configure and control playback
- Manage audio level metering

The **AVAudioPlayerDelegate** protocol has optional methods that are called when the audio file finishes playing, if there are interruptions or if there's an error.
https://developer.apple.com/documentation/avfoundation/avaudioplayerdelegate
The **AVAudioRecorder** class provides recording of audio data
https://developer.apple.com/documentation/avfoundation/avaudiorecorder

- Record until the user stops the recording
- Record for a specified duration
- Pause and resume a recording
- Obtain input audio-level data for level metering
- In iOS, the audio being recorded comes from the device connected by the user such as the built-in microphone or a headset microphone.
- Configurable setttings include bit depth, bit rate, and sample rate conversion quality https://developer.apple.com/documentation/avfoundation/avaudioplayer/1389359-settings

The **AVAudioRecorderDelegate** protocol has optional methods that are called when the recording completes, if there are interruptions or if there's an error
https://developer.apple.com/documentation/avfoundation/avaudiorecorderdelegate

Since we'll be recording and saving some audio we need to understand how and where our app can do that.

Sandbox
Your app sees the iOS file system like a normal UNIX file system
Every app gets its own /Documents directory which is referred to as its sandbox
Your app can only read and write from that directory for the following reasons:

- Security (so no one else can damage your application)
- Privacy (so no other applications can view your application's data)
- Cleanup (when you delete an application, everything its ever written goes with it)

To find your sandbox in the Finder go into your home directory and go to
Library/Developer/CoreSimulator/Devices/*Device UDID*/data/Containers/Data/Application

(The Library option is hidden so if you don't see the Library folder in your home directory Go | Go to Folder | Library hold down the alt key, or hold the Option key Go | Library)
Each app has it's own folder (names are the globally unique identifiers (GUIDs) generated by xcode
Each app has subdirectories

- Documents-app sandbox to store its data
- Library-user preferences settings
- Tmp-temp files (not backed up into iTunes)

The same file structure exists on devices. You can connect your device to your computer and in Window | Devices and Simulators select your device and you will see Installed Apps. Select an app and click the gear and Show Container to see the sandbox contents of that app.

The FileManager class is part of the Foundation framework and provides and interface to the file system. It enables you to perform many generic file-system operations such as locating, creating, copying, and moving files and directories.

- You can use URL or String for file location but URL is preferred
- We will use the urls(for:in:) method to locate the document directory. This method returns an array but on iOS there's only one document directory so we can just use the first item in the array

Add a fifth tab as before, along with a FifthViewController class for the new view controller and set the class for the new view in Interface Builder.
(same steps as before)

In the storyboard add 3 buttons for Record, Play, and Stop. (add needed constraints)
Connect these as outlets called recordButton, playButton, and stopButton. We need these so we can enable and disable these buttons as needed.
Also connect these as actions called recordAudio, playAudio, and stopAudio.

Now we need to add the AVFoundation.framework to our app.
Click on the Target and go into the Build Phases tab.
Open Link Binary with Libraries and click the + to add the AVFoundation.framework under iOS 14.3.

You must request permission before your app can access the microphone.
In info.plist you must add the entry "Privacy - Microphone Usage Description" and give it a value that will be used in the permission request to the user.

In FifthViewController.swift we need to import the AVFoundation framework and adopt the AVAudioPlayerDelegate and AVAudioRecorderDelegate.

```
import UIKit
import AVFoundation

class FifthViewController: UIViewController, AVAudioPlayerDelegate,
AVAudioRecorderDelegate
```

These protocols don't have required methods so you shouldn't get any errors. We'll implement their optional methods later.

Create an instance variable for our audioplayer and audiorecorder

```
    var audioPlayer: AVAudioPlayer?
    var audioRecorder: AVAudioRecorder?
```

Create a constant for the name of the file where the audio will be saved

```
     let fileName = "audio.m4a"
```

Then we'll create a method where we'll do our setup and initialization

```
    func setupAudio() {
    //disable buttons since no audio has been recorded
        playButton.isEnabled = false
        stopButton.isEnabled = false

        //get path for the audio file
        let dirPath = FileManager.default.urls(for: .documentDirectory, in:
.userDomainMask)
        let docDir = dirPath[0] //document directory
        let audioFileURL = docDir.appendingPathComponent(fileName)

     //the shared audio session instance
        let audioSession = AVAudioSession.sharedInstance()
        do {
           //sets the category for recording and playback of audio
           try
audioSession.setCategory(AVAudioSession.Category.playAndRecord, mode:
.default, options: .init(rawValue: 1))
        } catch {
            print("audio session error: \(error.localizedDescription)")
        }

        //recorder settings
https://developer.apple.com/documentation/avfoundation/avaudiorecorder/13883
86-init
        let settings = [
            AVFormatIDKey: Int(kAudioFormatMPEG4AAC), //specifies audio
codec
            AVSampleRateKey: 12000, //sample rate in hertz
            AVNumberOfChannelsKey: 1, //number of channels
            AVEncoderAudioQualityKey: AVAudioQuality.high.rawValue //audio
bit rate
        ]

        do {
            //create the AVAudioRecorder instance
            audioRecorder = try AVAudioRecorder(url: audioFileURL, settings:
settings)
            audioRecorder?.prepareToRecord()
            print("audio recorder ready")
        } catch {
            print("audio recorder error: \(error.localizedDescription)")
        }
    }
```

viewDidLoad is a good place to call our setup method.

```
    override func viewDidLoad() {
        super.viewDidLoad()
         setupAudio()
    }
```

Then we'll implement our methods to record, stop, and play.

```
    @IBAction func recordAudio(_ sender: UIButton) {
        //if not already recording, start recording
        if audioRecorder?.isRecording == false{
            playButton.isEnabled = false
            stopButton.isEnabled = true
            audioRecorder?.delegate = self
            audioRecorder?.record()
        }
    }


    @IBAction func playAudio(_ sender: UIButton) {
        //if not recording play audio file
        if audioRecorder?.isRecording == false{
            stopButton.isEnabled = true
            recordButton.isEnabled = false

            do {
                try audioPlayer = AVAudioPlayer(contentsOf:
                    audioRecorder!.url)
                audioPlayer!.delegate = self
                audioPlayer!.prepareToPlay()//prepares the audio player for
playback by preloading its buffers
                audioPlayer!.play() //plays audio file
            } catch let error {
                print("audioPlayer error: \(error.localizedDescription)")
            }
        }
    }

    @IBAction func stopAudio(_ sender: UIButton) {
        stopButton.isEnabled = false
        playButton.isEnabled = true
        recordButton.isEnabled = true
        //stop recording or playing
        if audioRecorder?.isRecording == true {
           audioRecorder!.stop()
        } else {
           audioPlayer!.stop()
        }
    }
```

Both delegate protocols have optional methods. Since you don't tap a button when the audio playing ends, let's implement that one to change the buttons as needed.

```
    //AVAudioPlayerDelegate method
    //Called when a recording is stopped or has finished due to reaching
its time limit
    func audioPlayerDidFinishPlaying(_ player: AVAudioPlayer, successfully
flag: Bool) {
        recordButton.isEnabled = true
        stopButton.isEnabled = false
    }
```