1. Before we get started:
    a. You need a mac
    b. You need an iPhone 6 or better to run this on your phone
    c. Go over to github to grab the 3d image asset we are working with and put it somewhere where you won't lose it
    d. Disclaimer, circumstances beyond my control, ninjas, etc.
2. Intro
    a. Who am I?
    b. Who are you?
    c. Why "second mobile app"
    d. What are our goals here
3. Let's get started:
    a. Open Xcode
    b. New Project -> Elf On A Shelf
    c. Open Main.storyboard
    d. Put a label on the screen -> North Pole
    e. Put another label on the screen
    f. Make a NorthPoleVC class, make a label outlet
    g. Wire up the label
    h. Run it, did it work?
    i. Make a messy network call to get the weather api data for North Pole, AK
        i. https://api.wunderground.com/api/b193c8afeeecdbb2/conditions/q/AK/North_Pole.json
    j. But this is gross and garbage, don't do this in vdl, don't even do this in the view controller
4. Let's talk Protocols
    a. What they are
    b. Where you may have seen them before
        i. Other languages
        ii. TableView stuff
    c. Move all this out to a protocol & extension
5. Temperature Displaying
    a. New file: Temperature Fetching, fetchTemp function
    b. Extract fetchTemp out of vdl - not a big win yet, but let's try this
    c. Extend TemperatureFetching, move everything into the extension
    d. New View Controller with a label "Santa Claus, IN" and temp label
    e. New file: SantaClausVC, delete cruft, IBOutlet for temp label, extend to be tempFetching
    f. Add segues back and forth, name them
    g. Add swipe gestures back and forth
    h. Add iboutlet to each VC, wire up
    i. Show off, see the two
    j. Run it--but wait, SCVC is getting the North Pole VC info! Oh noes!

    k. Add city & state vars to protocol - you need to tell the protocol if its getting, setting, or both these properties
    l. then define them in each vc
6. Let's talk Codable!
    a. What its replacing
    b. Encodable + Decodable
    c. Walk through implementation
    d. Walk through updating key names
    e. Walk through returning an actual object and updating that label on the main queue
7. Swipable
    a. New file, swift file, swipable, protocol
    b. Explain the where Self: A thing (further restrict the nature of the thing)
    c. Walk through the code
8. Recap & Break
    a. Stretches
    b. Go get coffee, meet back in five
9. Let's talk ARKit
    a. A bit of a history of it
    b. Limitations of ARKit right now
10. Let's get ARKit Busy
    a. Drag an ARKitSceneKitView onto the NPVC & resize it (this is where the magic will happen)
    b. Over in NPVC.swift, import ARKit
    c. Walk through setting up world tracking configuration
        i. This is what gives us the tracking of orientation and position as well as detects real world surfaces
    d. Pause the session in viewWillDisappear
        i. So the camera stops tracking when we leave the viewcontroller
    e. Now we have to ask the user for permission to use the camera, this is a requirement since iOS 10, and it's always best to ask for permission at the moment you need it
    f. Open info.plist, click the + in the last row and select "Privacy - Camera Use Permission" and set the value to "For Augmented Reality"
    g. A Word About Plists
    h. Run it! On a device! Hooray!
    i. Sort out everybodies' shitshow
11. Detect a plane
    a. Now we need to detect a horizontal plane in this view
    b. Let's start by declraring our NPVC as an ARSCNViewDelegate (do this in an extension)
        i. A word here, if you are familiar with ObjC style delegation and protocol conformance, I'm following the RW swift style guide, and I really like this

way of grouping protocols in individual extensions at the bottom of the file. It keeps everything in a consistent place, and keeps protocol conformance discreet

c. Up in vdl, let's make ourselves the sceneView delegate and create a SceneKit scene and assign it to our sceneView
d. This making me itch, so let's see if we can move this out into a setupARScene method
e. Now, we're going to need to create a sceneKit node that represents our virtual plane, and we are going to need tht node to have an ANCHOR, which represents the position and orientation for placing objects in a scene as well as the PLANEGEOMETRY: new file -> cocoaclass -> VirtualPlane: SCNNode
    i. Let's also give it this required init, which is required by subclasses of SCNNode
f. We have three different situations that are in response to various delegate calls that the the ARSCNViewDelegate makes:
    i. When new nodes are detected and added to the scene
    ii. When the camera moves or an object in view moves and the node is updated
    iii. When a node is deleted from a scene
g. Let's take these one at a time, starting with adding a new node to the scene. When that delegate method is called, it wants to initialize. So, let's give it and anchor, some geometry, and some color (or material)
    i. (paste and explain) Remember that the plane geometry is using the z AR axis as the Y SceneKit axis
    ii. SCNMaterial is the "material" of the object, the shading attributes that define the appearance of its surface
    iii. "Diffuse" is one of those attributes, and relates to the amount and color of light reflected off the surface. By default, it's a white color, and here we're making it kind of transparent by reducing the alpha. Because the "contents" type is "Any" we need to explicitly state that it's a UIColor
    iv. Once we have it set, we assign it to the materials property of the planeGeometry (which is an array)
    v. Let's pull this into its own method, for cleanliness
    vi. And now, let's make a sceneKit node to show this geometry that we just made by creating a node (paste and expalain)
    vii. This SCNMatrix4MakeRotation method with these attributes handles that rotation we need to do to convert the ARKit Z to SceneKit Y axis
    viii. Then we need to adjust that material we created to make sure it has the correct dimensions, and let's go ahead and start with that as a new function called updatePlaneMaterialDimensions() since, spoilers, we may want to use it again

    ix. At this point, we know that the only item in that materials array is the one we want, we can go ahead and grab the first item. We may want this to be something more sophisticated if add to that materials array.
    x. We get our width and height, and that SCNMatrix4MakeScale handles the rotation again for us
    xi. All that's left here is to call our update method and add planeNode as a child node to our scene
h. So now we need to go back to our NPViewController and handle those three delegate methods: when a node is added, when a node is updated, and when a node is deleted
    i. First things first, let's add a property on the vc for the UUID of planes. We want to keep a list of the virtual planes that we have and make sure that, when the didAdd protocol method is called, we can determine if it's a new plane being added or an existing one being updated.
    ii. Down in the protocol conformance, just start typing renderer...and find the didAddNode method stub (paste & explain)
    iii. We're going to unwrap a planeAnchor, create a virtualPlane with that anchor, we're going to add it to the node
    iv. But, as the camera moves, it's going to update the plane, and make that second of the three delegate calls, the one for updating Nodes, and we need to handle that. Let's start by going back to our virtualPlane (paste and explain)
    v. Again, we are going to set the width and we are going to set the height to the z axis, and the anchor position because this may have changed when the camera moved
    vi. We're going to update the material dimensions again
    vii. And then we have something we can call from the viewController when when that updateNode method is changed.
    viii. And here's what that looks like (past and explain)
    ix. And what we are doing here, is unwrapping the anchor, checking its ID to make sure it matches something in our existing list of anchors, and calling that update method we just wrote on it
    x. ALMOST THERE EVERYBODY I PROMISE we just need to implement the final, delete method. Check this out: (paste and explain)
    xi. Here we unwrap that anchor again, and this time we try and find the item in our planes list that matches the id of the anchor we are removing, and removing it.
i. LAST THING: add this debug line to your vdl. It will place some dots as the app is sensing a plane, and will give you this multicolored spindle thingy that shows the "world's origin" point and for me, it's a sanity check that things are working.
j. Run the app! Warning: it will take a disturbingly long time for an actual plane to appear, then it will be a plane party. (recount story of chasing the dog around and swearing)

12. So now, let's put something on this plane! That's what we are here for, right?
    a. First, let's add the assets. They are pinned in a .zip in the Slack channel for this class. Download them, then add them to Xcode in an assets folder.
    b. We can really only add an elf when we have a plane to put him on, right? So let's first add a variable for tracking status of the AR engine. In the HIG for ARKit business, apple wants us to make sure we keep the user informed on the status of an ARSession, and c'mon, it was only a few minutes ago that we were freaking out about planes. So.
        i. In NPVC, create an enum for ARElfSessionState
        ii. FYI: CustomStringConvertible is another protocol, and you conform to it by providing a description
        iii. This derived property pattern is pretty common for enums, and is very handy
        iv. Create an iboutlet for statuslabel, then place the status label on the screen in the storyboard, give it a little bit of a background
        v. Now we want to create a property on our NPVC for currentElfStatus, which is going to be an ARElfStessionState (paste and explain)
        vi. What this does, is--whenever that status changes, didSet gets called, and when that happens, we will pop over to the main thread and update the label. If the status is updated to failed, we'll go through and loop over all the child nodes and remove them--basically, wipe the slate and start over.
        vii. So now that we have this status, let's make sure it gets updated when appropriate. First, over in our planes array, if we didSet another plane in there, update the status. Also, over in vdl, if we have a plane right away, make sure we set our status to ready. In viewWillDisappear, let's just make sure we are sending the right message by changing the status to .tempunavail. And finally, let's round out the delegate methods in ARSCNViewDelegate by handling didFailWithError, wasInterupted, and interuptionEnded. Finally, run the app for a sanity check.
    c. So now we have some planes, and some notification of when those planes are ready for an elf, let's make an elf. Start by creating a scenekitnode in the viewController (do this)
        i. And a function to initialize the node (paste and explain)
        ii. So here I have some bad news. The first bit of bad news is that Christmas is over, and all the elves have run back home to the north pole to work on next Christmas, leaving behind only clues that they have been here. The second bit of bad news is that 3d models of christmas elves are freakin' expensive and I wasn't about to spend $150 on one. So all we can find at this time, is a dropped elf hat.
        iii. Notice that we need to add the whole scene that the contains the elf poop. Also, that recursive flag--if the node we are adding has its own sub-nodes, those are added recursively. Since we only have one node here, we can set that to false.

    iv. Now, we have to call this initialize method in vdl--once we have one initialized, we'll eventually be able to clone it when we drop one one a plane
d. Alright, we've got planes, we've got elves, now we need to detect touches so we can place these guys. To do that, we are going to harness the TOUCHESBEGAN method on the view controller, extract the touching point, do a hit test to make sure the touch point is within a plane, and drop some elf droppings if we can.
e. THIS IS THE HOME STRETCH PEOPLE
f. So we're going to start by overriding touches began--if you start typing that, you'll get the autocomplete
    i. First test is to make sure we have one valid touch, and that we are only looking at the first one of them
    ii. Second test is that our currentElfStatus is ready for placing, and bail out if we aren't
    iii. Finally, we grab the touchpoint and...this is where we need to make sure that this touchpoint is within a plane, so let's create a method that does that
g. Paste and explain virtualPlaneProperlySet
    i. First, we're going to need a property to store our possibly valid virtual plane in, so let's make one of those and make it optional--we might not have one, after all
    ii. "existingPlaneUsingExtent" is a plane anchor that is already in the scene, with that plane's limited size
    iii. So now we have a plane, (put the guard in the touchesBegan method) and we just need to drop some elf--or an elf hat--on that plane
h. addElfToPlane(plane: plane, at: touchPoint)
    i. We're going to get a hit from the hit test again
    ii. And make sure that we have a hit and grab that first one
    iii. Then we are going to clone that elfNode we initialized in vdl
    iv. Then we take our cloned elfNode and add it to the screen
i. RUN THE APP DROP SOME HATS WOOHOO WE DID IT

13. Tap to place santa claus
14. Protocol out the above?
15. Create a factory that randomly dispenses an elf