Power Up Your Animations Tasks

DIY #1

You've seen just how handy EasyAnimation is when you want to build animation sequences and groups. So far you got the login button animating repeatedly and the next step would be to simulate the authentication going trough and moving on to the next screen of the app.

As you know with UIKit there is no way to stop a repeating animation since you don't have any object to reference the animation. With Easy Animation on the other hand you do have an object you can call methods on - only in the cases when you create animation sequences but still - something that UIKit lacks.

Scroll to the first line of the code that animates the login button:

```
UIView.animateAndChainWithDuration(0.25, delay: 0.0,
```

The call to `animateAndChainWithDuration` and all chained animation calls return a chain animation object of type `EADelayedAnimation`, which you can use to stop the animation. Just assign the returned object to a constant like so:

```
let loginBtnAnimation = UIView.animateAndChainWithDuration(0.25, delay: 0.0,
```

Now you can simulate a bit of delay and then add an animation to show to the user that they successfully logged in.

At the bottom of `actionLogin` append:

```
delay(seconds: 5.0, {
  //successfull login

})
```

You give your login animation 5 seconds to animate and then you will stop it. `EAAnimationDelayed` has a handy method that cancels the animation sequence so add the code to do that just under `//successfull login`:

```
loginBtnAnimation.cancelAnimationChain(completion: {
    //reset button
})
```

Note that this method does not *stop* the animation but waits until the current leg of the sequence completes and than cancels the sequence and releases all animations. Then it calls your `completion` code.

Now add some extra animations in the `completion` block:

- Create an animation sequence by using `animateAndChainWithDuration` like you did previously (no need to catch the animation object in a constant this time).
- In the first 0.33 sec animation transform down the button to a 0.01 scale (you can lookup how to create a scale transform in the earlier code)
- In the second 0.25 sec animation move the `fldUsername` field left by `self.view.bounds.width` points (out of the left screen edge)
- In the third 0.25 sec animation move `fldPassword` by the same amount of points out of the left edge of the screen.
- When your animation works fine add this code in the completion of the last animation of the sequence to send the user to the next screen:

```
actionShowOnboarding()
```

NB: Once you visit the next screen the app will begin launching directly to that second screen. If you want to go back to the login screen just tap on "Log out" in the top left corner.

DIY #2

In this exercise you will make the content of the "Listen everywhere" view strip interactive. Whenever the user expands this strip you will create a button, add it to the view hierarchy and animate it on screen.

This will require you to create manually all constraints to position correctly the button so don't waste any time this will be code intense.

First add an instance variable to keep hold of your button:

```
var startButton: UIButton!
```

and a method to create and show the button. This code is a normal UIKit that doesn't have to do with animations so you can just copy and paste the whole method:

```
func showStartButton() {
  startButton = UIButton.buttonWithType(.Custom) as! UIButton
  startButton.backgroundColor = UIColor(red: 1.0, green: 1.0, blue: 1.0,
alpha: 0.9)
  startButton.setTitleColor(UIColor.blackColor(), forState: .Normal)
  startButton.setTitle("Start", forState: .Normal)
  startButton.titleLabel?.font = UIFont.systemFontOfSize(22.0)
  startButton.layer.cornerRadius = listenView.bounds.size.width/6
  startButton.layer.masksToBounds = true
    startButton.setTranslatesAutoresizingMaskIntoConstraints(false)

  startButton.addTarget(self, action: "actionStartListening:",
forControlEvents: .TouchUpInside)

  listenView.addSubview(startButton)
}
```

`showStartButton` creates a `UIButton` sets the colors of the background and text and adds a text title "Start". Finally it adds a tap action that calls `actionStartListening:`. actionStartListening is already included in the stater code and just invokes the segue to the next screen.

Add also a simple method to hide the button when the strip is deselected:

```
func hideStartButton() {
    startButton.removeFromSuperview()
}
```

Now just as for the "Sign Up" strip you are going to add the custom code in toggleListen. Append at the bottom:

```
showStartButton()

deselectCurrentView = {
    self.hideStartButton()
}
```

Now you are ready to add the constraints and animations in `showStartButton`. You will add the button centered horizontally and located just under the botton of the screen. Then you will animate it up and into the visible area of the "Listen everywhere" strip.

Append to `showStartButton` the constraint to center the button horizontally within `listenView`:

```
let conX = NSLayoutConstraint(item: startButton, attribute: .CenterX,
relatedBy: .Equal, toItem: listenView, attribute: .CenterX, multiplier: 1.0,
constant: 0.0)
```

Then add the vertical alignment constraint:

```
let conY = NSLayoutConstraint(item: startButton, attribute: .Bottom,
relatedBy: .Equal, toItem: listenView, attribute: .Bottom, multiplier: 0.67,
constant: view.frame.size.height * 0.33)
```

The code aligns the button towards the upper half of the strip by using the 0.67 multiplier and then offsets the button down by adding a constant to the constraint equation. You will animate the button by adjusting the constant later on.

Finally add the constraints to fix the button's width and height:

```
let conWidth = NSLayoutConstraint(item: startButton, attribute: .Width,
relatedBy: .Equal, toItem: listenView, attribute: .Width, multiplier: 0.33,
constant: 0.0)
let conHeight = NSLayoutConstraint(item: startButton, attribute: .Height,
relatedBy: .Equal, toItem: listenView, attribute: .Width, multiplier: 0.33,
constant: 0.0)
```

Activate the constraints:

```
NSLayoutConstraint.activateConstraints([conX, conY, conWidth, conHeight])
```

Before you create the animation is very important to trigger a layout pass - right now all constraints are added in place but the button's `position` and `size` aren't update just yet. They will be after you call `layoutIfNeeded`. Add:

```
startButton.layoutIfNeeded()
```

You can finally add the animation to your button:

```
UIView.animateWithDuration(1.33, delay: 0.1, usingSpringWithDamping: 0.7,
initialSpringVelocity: 0.0, options: .BeginFromCurrentState |
.AllowUserInteraction, animations: {
    conY.constant = 0.0
    self.startButton.layoutIfNeeded()
}, completion: nil)
```

This will animate the button up whenever the user expands the "Listen everywhere" strip.

DIY #3

This is a freestyle DIY - so far you have the custom transition going on, now you have a chance to customize it in any way you like.

If you're short on ideas while you're on the spot try some of these:

- instead of flying right to left, have `toVC` appear centered and invisible and then fade in
- have fromVC scale up to 10.0 times size, then have `toVC` start at 10.0 scale and scale to its normal size to fit on the screen

DIY #4

This is a quick exercise mainly to let my voice rest for just a bit. You have the left channel meter working - now in couple of steps you will add also the right channel.

Scroll to `setupMeter` and add the code to create a new meter view:

```
//right channel
var rightMeter = createMeter()
rightMeter.frame.origin.x = displayView.bounds.size.width * 0.51
displayView.layer.addSublayer(rightMeter)
```

The code is very similar to the one that creates the left channel meter view but this time you offset the position of the view to the right half of the meter display (thus the width * 0.51)

Okay - this ought to create the second meter view - you can run the app to check if you see it.

Next - add the code to animate the right channel meter. You need to add your code inside the setupMeter method inside the `levelsHandler` closure.

Copy over the code that animates the left channel and adjust it to work for the right channel levels value and to animate the right channel meter view.

DIY #5

In **VisualizerViewController.swift** you will have a similar setup to the replicator layer display you created for your SongsViewController.swift.

This time your replication will progress horizontally from left to right so that it resembles a timeline of sorts. (Unlike in the previous case where your replication progressed from bottom to top)

First you need to create a method that will create a replicator layer to your liking (just like you did before for the channel meter). Add the initial code of that factory method:

```
func createDisplay(first: CALayer) -> CAReplicatorLayer {
  var display = CAReplicatorLayer()
  display.addSublayer(first)

  display.instanceTransform = CATransform3DMakeTranslation(12, 0, 0)
  display.instanceCount = Int(view.frame.size.width / 12)+1

  return display
}
```

In screen's animation you will use bars of 10pt width so you set the `instanceTransform` to 12pt (this will give you a tiny bit of space between the replications) and for `instanceCount` you just take the width of the view controller's view divide it by the space needed per replication and end up with how many copies you need to fill the screen width. (You add one so that you have one copy being cut-off the screen rather than having a little gap at the screen edge)

The starter code includes two instance variables `leftBar` and `rightBar`. Those are the two layers you are going to replicate and animate. In this exercise you will work with `leftBar`.

Scroll to `viewDidLoad` and append:

```
leftBar.cornerRadius = 2.0
leftBar.backgroundColor = UIColor(red: 0.98, green: 0.69, blue: 0.37, alpha:
1.0).CGColor
leftBar.frame = CGRect(x: 0, y: view.bounds.size.height/2, width: 5, height:
2)
```

You round up the layer's corners a bit and give it a reddish color. Finally you center the layer within the screen and give it a 5 by 2 points size. This will be the initial position and shape of the layer being replicated.

Next - add `leftBar` to a replicator:

```
var leftDisplay = createDisplay(leftBar)
leftDisplay.frame = view.bounds
view.layer.addSublayer(leftDisplay)
```

Just as you did before - you create a replicator, set its frame and add some content to it.

Now you are ready for action!

You remember that `SongsViewController` plays the song and passes the values of the left and right channel to `updateLevels` in `VisualizerViewController`.

This is the place where you need to create your animations.

First - scale the layer according the channel level (I tried some number and the formula below produces a nice levels-to-layer-scale ratio):

```
let leftScaleFactor = max(0.2, CGFloat(left) + 40) * 3
```

This scale will let the animation take up around 1/3 of the height of the screen at most times and leave enough empty space around.

Now add a scale animation to the layer:

```
let leftScale = CABasicAnimation(keyPath: "transform.scale.y")
leftScale.toValue = leftScaleFactor
leftScale.duration = 1.0
leftBar.addAnimation(leftScale, forKey: nil)
```

This is a bit of a hack so let's have a look at what's going on.

1 You create a scale animation.

2 You set the `toValue` - the current scale you want to have. You do not specify a `fromValue` as you are probably used to - this will make the animation start from its current scale.

3 You set a 1.0 second duration - this will give CoreAnimation enough time to have all replications of this animation complete before removing it. You will experiment with this in a bit to better understand why you need this 1.0 sec duration.

4 Finally you add the animation to `leftBar`.

If you run the app right now you will see all replications obediently animate in sync. This is interesting but not what you need. You want to create a kind of timeline and therefore you need all the copies to animate with a bit of delay between each other.

Scroll to `createDisplay` and insert just before the `return` statement:

```
display.instanceDelay = 0.02
```

This will make all replicated animations run with a 0.02 delay from copy to copy.

Cool - you have your audio wave going on!

Now - if you are still curious about the 1.0 duration, change it to 0.5 or 0.25 to see the effect. When the animation has finished and CoreAnimation removes it - it’s removed also from all replications and that destroys the effect you are looking for.