Happiness Walkthrough

Create a new project

Delete the main storyboard (not the file, the actual storyboard)

Delete the view controller.

Drag out a new view controller

Create a new file for it, Cocoa Touch class, subclass of UIViewController, call it HappinessViewController

Change the identity of the view controller that was dragged out to HappinessViewController

Now drag a view into the ViewController

Drag it to be the whole screen, then reset to suggested constraints

Now make a new file, also CocoaTouch, that is a subclass of UIView, call it FaceView

Now go to the storyboard and pick the view (ctrl-shift-click) and change it to FaceView

Go to the FaceView class.

Discuss how drawing works in iOS. There are two modes, one that uses Core Tools, the other that is more object oriented. Since Swift is OO, we will use the second way.

```
override func drawRect(rect: CGRect) {
}
```

Start typing:

```
let facePath = UIBezierPath(
```

at this point you can see all of the different kinds of bezier paths that you can draw. We actually want an oval, but we're going to draw it using an arc path just to show some stuff off.

```
let facePath = UIBezierPath(arcCenter: faceCenter, radius: faceRadius, startAngle: 0, endAngle: 2*M_PI, clockwise: true)
```

Let's make some of those properties.

```
var faceCenter: CGPoint {
    return convertPoint(center, fromView: superview) // center is in my superview's coords
}

var faceRadius: CGFloat {
    return min(bounds.size.width, bounds.size.height) / 2
}
```

This still leaves us with an error since 2*M_PI is a double, and we want an CGFloat.

```
        let facePath = UIBezierPath(arcCenter: faceCenter, radius: faceRadius, startAngle: 0,
endAngle: CGFloat(2*M_PI), clockwise: true)
```

Let's add some other configurable options:

First

```
  facePath.lineWidth = 3
```

But we'd like to be able to configure this

```
    var lineWidth: CGFloat = 3
```

And update to

```
  facePath.lineWidth = lineWidth
```

Now, when the line width changes I need to redraw. We can easily handle this with a property observer.

```
    var lineWidth: CGFloat = 3 { didSet { setNeedsDisplay() } }
```

similarly

```
    var color: UIColor = UIColor.blueColor() { didSet { setNeedsDisplay() } }
```

Back in drawRect:

```
        color.set()
        facePath.stroke()
```

Run it. Nothing happens, except an error message. We didn't specify where to start.

Fix it with the inspector (click on Is Initial View Controller). Show how the arrow can move.

Now the result is decent, but doesn't look that great.

Change the mode on the View (in the inspector) from ScaleToFill to Redraw. The reason to do this is to force the new call to redraw rather than the default of just scaling bits that were already drawn.

The head is still too big, so let's scale it down some.

```
    var scale: CGFloat = 0.90 {didSet { setNeedsDisplay() }}
```

Then update the faceRadius accordingly

```
    var faceRadius: CGFloat {
        return min(bounds.size.width, bounds.size.height) / 2 * scale
    }
```

Run that.

Xcode added a very cool feature in version 6. Before the class statement in FaceView add the command:

@IBDesignable

This is awesome because the view will draw right in the storyboard.

To get the rest of the face I'm going to need a bunch of constants to manage the placement of the eyes, the size of the mouth, etc. In Swift we do something like this:

```
private struct Scaling {
    static let FaceRadiusToEyeRadiusRatio: CGFloat = 10
    static let FaceRadiusToEyeOffsetRatio: CGFloat = 3
    static let FaceRadiusToEyeSeparationRatio: CGFloat = 1.5
    static let FaceRadiusToMouthWidthRatio: CGFloat = 1
    static let FaceRadiusToMouthHeightRatio: CGFloat = 3
    static let FaceRadiusToMouthOffsetRatio: CGFloat = 3
}
```

Paste in the code for Eye and Smile. There isn't much to say here, but quickly walk through the code.

```
private enum Eye { case Left, Right }

private func bezierPathForEye(whichEye: Eye) -> UIBezierPath
{
    let eyeRadius = faceRadius / Scaling.FaceRadiusToEyeRadiusRatio
    let eyeVerticalOffset = faceRadius / Scaling.FaceRadiusToEyeOffsetRatio
    let eyeHorizontalSeparation = faceRadius / Scaling.FaceRadiusToEyeSeparationRatio

    var eyeCenter = faceCenter
    eyeCenter.y -= eyeVerticalOffset
    switch whichEye {
    case .Left: eyeCenter.x -= eyeHorizontalSeparation / 2
    case .Right: eyeCenter.x += eyeHorizontalSeparation / 2
    }

    let path = UIBezierPath(
        arcCenter: eyeCenter,
        radius: eyeRadius,
        startAngle: 0,
        endAngle: CGFloat(2*M_PI),
        clockwise: true
    )
    path.lineWidth = lineWidth
    return path
}
```

Back in drawRect

```
bezierPathForEye(.Left).stroke()
bezierPathForEye(.Right).stroke()
```

Play around with it a bit. E.g. change the end angle to be just M_PI

```
private func bezierPathForSmile(fractionOfMaxSmile: Double) -> UIBezierPath
{
    let mouthWidth = faceRadius / Scaling.FaceRadiusToMouthWidthRatio
    let mouthHeight = faceRadius / Scaling.FaceRadiusToMouthHeightRatio
    let mouthVerticalOffset = faceRadius / Scaling.FaceRadiusToMouthOffsetRatio

    let smileHeight = CGFloat(max(min(fractionOfMaxSmile, 1), -1)) * mouthHeight

    let start = CGPoint(x: faceCenter.x - mouthWidth / 2, y: faceCenter.y + mouthVerticalOffset)
    let end = CGPoint(x: start.x + mouthWidth, y: start.y)
    let cp1 = CGPoint(x: start.x + mouthWidth / 3, y: start.y + smileHeight)
    let cp2 = CGPoint(x: end.x - mouthWidth / 3, y: cp1.y)

    let path = UIBezierPath()
    path.moveToPoint(start)
    path.addCurveToPoint(end, controlPoint1: cp1, controlPoint2: cp2)
    path.lineWidth = lineWidth
    return path
}
```

Then finish off drawRect.

```
let smilePercent = 0.75
let smilePath = bezierPathForSmile(smilePercent)
smilePath.stroke()
```

Run it. Change the smallness to a small negative value and run again.

Turns out we can edit the attributes of the FaceView as well. Before each var just add
@IBInspectable
Do this for lineWidth, color, scale

We have a view, we have a ViewController (which is basically empty so far).

Take a break to do the slides for this.

Let's fill in the controller and add a model.

```
    var happiness: Int = 50 { // 0 = Sad, 100 = joyous
        didSet {
            happiness = min(max(happiness, 0), 100)
            print("Happiness = \(happiness)")
            updateUI()
        }
    }

   func updateUI() {
   }
```

Notes: different version of values than in View (on purpose of course). Uses didSet to make sure that we are getting legal values. UpdateUI is a function that we will write, that will simply tell the View to do its thing.

Now we need to hook the view up to the view controller. This is probably the single most important thing you need to learn this term. You cannot do MVC without understanding this step.

The FaceView's data is the amount of smile. We need a way to get that from our controller. We'll do that by setting up a Protocol.

The first thing we do is set up the protocol (At the top of FaceView before the designable):

```
protocol FaceViewDataSource {
    func smilinessForFaceView(sender: FaceView) ->Double?
}
```

Then we need a variable to be our delegate. The thing that we actually use to call for our data.

```
    var dataSource: FaceViewDataSource?
```

Now, when I have a controller that wants to be my data source, it will just set itself as the delegate/datasource and then when I ask for the data it will be the place that sends it.

Now let's make that variable "weak". Why? Well because of reference counting.

This will add an error, which we can fix by adding : class to our protocol definition. This forces anything that implements our protocol be a class.

```
        let smilePercent = dataSource?.smilinessForFaceView(self) ?? 0.0
```

?? means if the thing on the left is nil, then use this alternative value. This will work for either of the two possible nils in our expression (the dataSource and what the function returns)

Now to the view controller. Let Swift know that it implements the protocol.

class HappinessViewController: UIViewController, FaceViewDataSource

then

```
    func smilinessForFaceView(sender: FaceView) -> Double? {
        return Double(happiness - 50)/50
    }
```

We also need to set ourselves as the datasource.

We need a pointer to the View, so we get that as an outlet.

Control drag over from the storyboard and make an outlet called faceView.

```
    @IBOutlet weak var faceView: FaceView! {
        didSet {
            faceView.dataSource = self
        }
    }
```

Now we can finish updateUI()

```
    func updateUI() {
        faceView.setNeedsDisplay()
    }
```

Run it with different levels of happiness.

Stop and do the Gestures lecture

Now let's add some gestures

In FaceView add:

```
    func scale(gesture: UIPinchGestureRecognizer) {
        if gesture.state == .Changed {
            scale *= gesture.scale
            gesture.scale = 1
        }
    }
```

Then in the controller we update:

```
    @IBOutlet weak var faceView: FaceView! {
        didSet {
            faceView.dataSource = self
```

```
            faceView.addGestureRecognizer(UIPinchGestureRecognizer(target: faceView, action:
"scale:"))
        }
    }
```

Run it. Hold down the option key to get a pinch gesture.

Now let's use a gesture to handle the amount of happiness. We could do this exactly the same way if we wanted. Instead will do it right in the storyboard.

Just drag a Pan Gesture recognizer right onto the View. This adds a little icon at the top of the controller. From it we can control drag over to the view controller as an action.

```
    @IBAction func changeHappiness(sender: UIPanGestureRecognizer) {
    }
```

We're going to read the amount of swipe and translate that into a happinesschange. To do that we'll have to figure out how to scale the change in pixels to a change in value. Let's make a constant to help:

```
    private struct Constants {
        static let HappinessGestureScale: CGFloat = 4
    }
```

Then

```
    @IBAction func changeHappiness(gesture: UIPanGestureRecognizer) {
        switch gesture.state {
        case .Ended: fallthrough
        case .Changed:
            let translation = gesture.translationInView(faceView)
            let happinessChange = -Int(translation.y / Constants.HappinessGestureScale)
            if happinessChange != 0 {
                happiness += happinessChange
                gesture.setTranslation(CGPointZero, inView: faceView)
            }
        default:
            break
        }
    }
```