# Property Lists

- **Property List is really just a definition for AnyObject which is known to be a collection of objects which are ONLY one of: NSString, NSArray, NSDictionary, NSNumber, NSData, NSDate**
  - e.g. NSDictionary whose keys were NSString and values were NSArray of NSDate
  - In Swift, the definition of Property List is exactly the same and the bridging all works
  - Handling Property Lists usually requires a fair amount of casting (i.e. is and as)
  - That's because plist are composed of AnyObject and you must figure out if it's what you expect
- **Property Lists are used to pass around data "blindly"**
  - grammatical semantics of the contents of a Property List are known only to its creator
  - Property Lists are also used as a "generic data structure"
- **Can be passed to API that reads/writes generic data …**
  - Only good for small amounts of data e.g. "preference s & settings"
  - You would never want your application's actual "data" stored here.
  - Can be stored permanently: Use writeToURL(NSURL, atomically: Bool) and AnyObject(contentsOfURL:NSURL)
- **Also three formats for storing in files or reading from internet via a URL:**
  - XML, Binary, NSPropertyListSerialization converts Property Lists to/from NSData

# Persistence – UserDefaults

- **A storage mechanism for Property List data**
  - tiny database to store Property List data
  - persists between launchings of your application!
  - great for things like "preferences & settings"
  - do not use it for anything big!
- **It can store/retrieve entire Property Lists by name (keys) …**

```
setObject(AnyObject, forKey: String) // the AnyObject must be a Property List
objectForKey(String) -> AnyObject?
arrayForKey(String) -> Array<AnyObject>? // returns nil if value is not set or not an array
```

- **It can also store/retrieve little pieces of data ...**

```
setDouble(Double, forKey: String)
doubleForKey(String) -> Double // not an optional, returns 0 if no such key
```

# Persistence – UserDefaults

- **Using UserDefaults**
  - Get the defaults reader/writer ...

```
let defaults = UserDefaults.standard
```

  - Then read and write ...

```
let plist: AnyObject = defaults.objectForKey(String)
defaults.setObject(AnyObject, forKey: String) // AnyObject must be a PropertyList
```

  - Your changes will be automatically saved.
  - But you can be sure they are saved at any time by synchronizing …

```
if !defaults.synchronize() { /* failed! not much you can do about it */ }
```

  - it's not "free" to synchronize, but it's not that expensive either

# Demo

*RPN Calculator*

# Demo

- **Adding persistence using UserDefaults**

# Views

- **A view represents a rectangular area**
  - UIView subclass
  - Defines a coordinate space
  - Draws and handles events in that rectangle
- **Hierarchical**
  - Only one superview `var superview: UIView?`
  - Can have many (or none) subviews `var subviews: [UIView]`
  - (subviews is actually [AnyObject] but array contains list of UIView)
  - Subview order (in that array) matters: those later in the array are on top of those earlier
  - A view can clip its subviews to its own bounds or not (the default is not to)
- **UIWindow:**
  - The UIView at the very top of the view hierarchy (also includes status bar)
  - iOS applications have only one UIWindow (generally)
  - iOS is all about views, not windows

# Views

- **The hierarchy is most often constructed in Xcode using the built-in Interface Builder**
  - Even custom views are added to the view hierarchy using Xcode
- **But it can be done in code as well:**

```
addSubview(aView: UIView) // sent to aView's (soon to be) superview
removeFromSuperview() // this is sent to the view you want to remove (not its superview)
```

- **Where does the view hierarchy start?**
  - The top of the (useable) view hierarchy is the Controller's var view: UIView
  - This simple property is a very important thing to understand!
  - This view is the one whose bounds will change on rotation, for example.
  - This view is likely the one you will programmatically add subviews to (if you ever do that).
  - All of your MVC's View's UIViews will have this view as an ancestor.
  - It's automatically hooked up for you when you create an MVC in Xcode.
- **Managing the order of subviews (not very common)**

```
insertSubview(view: UIView, atIndex: Int)
insertSubview(view: UIView, aboveSubview: UIView)
insertSubview(view: UIView, belowSubview: UIView)
```

# Initialising a UIView

- **As always, try to avoid an initializer if possible. But having one in UIView is slightly more common than having a UIViewController initializer**
- **A UIView's initializer is different if it comes out of a storyboard**

```
init(frame: CGRect) // initializer if the UIView is created in code
init?(coder: NSCoder) // initializer if the UIView comes out of a storyboard
```

- **If you need an initializer, implement them both …**

```
func setup() { ... }

override init(frame: CGRect) { // designed initializer
    super.init(frame: frame)
    setup()
}

required init?(coder: NSCoder) { // required initializer
    super.init(coder: aDecoder)
    setup()
}
```

- **Another alternative to initializers in UIView …**

```
awakeFromNib() // this is only called if the UIView came out of a storyboard
```

  - This is not an initializer (it's called immediately after initialization is complete)
  - All objects that inherit from NSObject in a storyboard are sent this (if they implement it)
  - Order is not guaranteed, so you cannot message any other objects in the storyboard here

# Coordinates

- **CGFloat**
  - Always use this instead of Double or Float for anything to do with a UIView's coordinate system
  - You can convert from a Double or Float using `let cgf = CGFloat(aDouble)`
- **CGPoint**
  - Simply a struct with two CGFloats in it: x and y

```
var point = CGPoint(x: 128.0, y: 280.0)
point.y += 20.0
point.x -= 28.0
```

- **CGSize**
  - Also a struct with two CGFloats in it: width and height.

```
var size = CGSize(width: 100.0, height: 50.0)
size.width += 28.0
size.height -= 20.0
```

# Coordinates

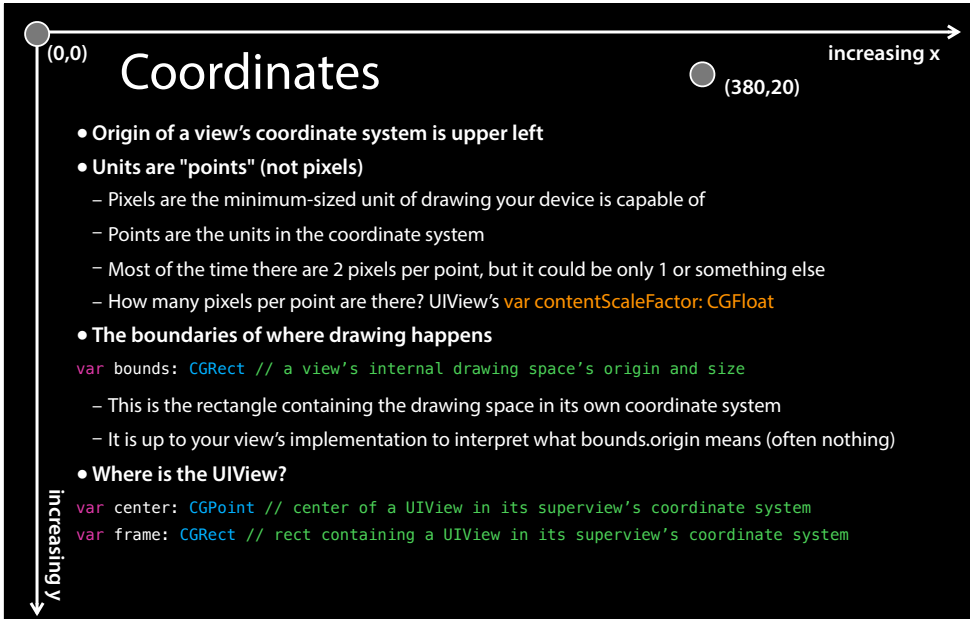
- **CGRect**
  - A struct with a CGPoint and a CGSize in it ...

```
struct CGRect {
    var origin: CGPoint
    var size: CGSize
}
let rect = CGRect(origin: CGPoint, size: CGSize) // there are other inits as well
```

  - Lots of convenient properties and functions on CGRect like …

```
var minX: CGFloat // left edge
var midY: CGFloat // midpoint vertically
intersection(CGRect) -> Bool // does this CGRect intersect this other one?
intersect(CGRect) // clip the CGRect to the intersection with the other one
contains(CGPoint)->Bool // does the CGRect contain the given CGPoint?
```

  - and many more (make yourself a `CGRect` and type `"."` after it to see more)




# Coordinates

- **Origin of a view's coordinate system is upper left**
- **Units are "points" (not pixels)**
  - Pixels are the minimum-sized unit of drawing your device is capable of
  - Points are the units in the coordinate system
  - Most of the time there are 2 pixels per point, but it could be only 1 or something else
  - How many pixels per point are there? UIView's var contentScaleFactor: CGFloat
- **The boundaries of where drawing happens**

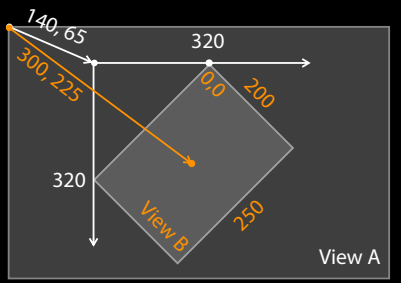
```
var bounds: CGRect // a view's internal drawing space's origin and size
```

  - This is the rectangle containing the drawing space in its own coordinate system
  - It is up to your view's implementation to interpret what bounds.origin means (often nothing)
- **Where is the UIView?**

```
var center: CGPoint // center of a UIView in its superview's coordinate system
var frame: CGRect // rect containing a UIView in its superview's coordinate system
```

# Coordinates

- **Use frame and center to position the view in the hierarchy**
  - They are used by superviews, never inside your UIView subclass's implementation.
  - You might think frame.size is always equal to bounds.size, but you'd be wrong ...
  - Because views can be rotated (and scaled and translated too).
- **Rotated View B**
  - `bounds = ((0,0),(200,250))`
  - `frame = ((140,65),(320,320))`
  - `center = (300,225)`
  - middle in its own coordinate space is:
    `(bounds.midX, bounds.midY) = (100,125)`
    in this case.
- **Views are rarely rotated, but don't misuse frame or center by assuming that.**



# Creating Views

- **Most often your views are created via your storyboard**
  - Xcode's Object Palette has a generic UIView you can drag out
    After you do that, you must use the Identity Inspector to changes its class to your subclass
- **On rare occasion, you will create a UIView via code**
  - You can use the frame initializer …

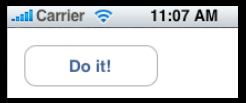
```
let newView = UIView(frame: myViewFrame)
```

  - Or you can just use

```
let newView = UIView()  // frame will be CGRectZero
```

- **Example:**

```
// assuming this code is in a UIViewController
let buttonRect = CGRect(x: 20, y: 20, width: 100, height: 50)
let button = UIButton(frame: buttonRect)
button.titleLabel!.text = "Do it!"
view.addSubview(button)
```




# Custom Views

- **When would I want to create my own UIView subclass?**
  - I want to do some custom drawing on screen.
  - I need to handle touch events in a special way (i.e. different than a button or slider does)
  - We'll talk about handling touch events next week. This week is drawing.
- **Drawing is easy: create a UIView subclass and override one single method**

```
override func draw(_ rect: CGRect)
```

  - You can optimize by not drawing outside of aRect if you want (but not required)
  - The rect is purely an optimization
- **Never call draw, Ever!**
  - Instead, let iOS know that your view's drawing is out of date with one of these UIView methods:

```
setNeedsDisplay()
setNeedsDisplayInRect(regionThatNeedsToBeRedrawn: CGRect)
```

  - It will then set everything up and call draw for you at an appropriate time
  - Obviously, the second version will call your draw with only rectangles that need updates

# Custom Views

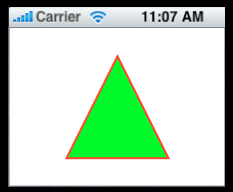
- **So how do I implement my draw()?**
  - Use a C-like (non object-oriented) API called Core Graphics
  - Or use the object-oriented UIBezierPath class
- **Core Graphics Concepts**
  - You get a context to draw into (could be printing context, drawing context, etc.)
  - The function UIGraphicsGetCurrentContext() gives a context you can use in drawRect
  - Create paths (out of lines, arcs, etc.)
  - Set drawing attributes like colors, fonts, textures, linewidths, linecaps, etc.
  - Stroke or fill the above-created paths with the given attributes
  - The API is C (not object-oriented)
- **UIBezierPath**
  - Same as above, but captures all the drawing with a UIBezierPath instance
  - UIBezierPath automatically draws in the "current" context (drawRect sets this up for you)
  - Methods for adding to the UIBezierPath (lineto, arcs, etc.) and setting linewidth, etc.
  - Methods to stroke or fill the UIBezierPath

# Core Graphics Context

- **The context determines where your drawing goes**
  - Screen (the only one we're going to talk about today)
  - Offscreen Bitmap
  - PDF
  - Printer
- **For normal drawing, UIKit sets up the current context for you**
  - But it is only valid during that particular call to drawRect
  - A new one is set up for you each time drawRect is called
  - So never cache the current graphics context in drawRect to use later!
- **How to get this magic context?**
  - Call the following C function inside your drawRect method to get the current graphics context:

```
let context = UIGraphicsGetCurrentContext()
```

# Defining a Path in Core Graphics




- **Begin the path**

```
CGContextBeginPath(context)
```

- **Move around, add lines or arcs to the path**

```
CGContextMoveToPoint(context, 85, 10)
CGContextAddLineToPoint(context, 10, 150)
CGContextAddLineToPoint(context, 160, 150)
```

- **Close the path (connects the last point back to the first)**

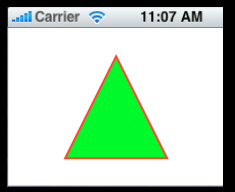
```
CGContextClosePath(context) // not strictly required
```

- **Set any graphics state (more later), then stroke/fill the path**

```
UIColor.greenColor().setFill() // object-oriented convenience method
UIColor.redColor()setStroke()
CGContextDrawPath(context, kCGPathFillStroke) // kCGPathFillStroke is a constant
```

- **It is also possible to save a path and reuse it**
  - Similar functions to the previous slide, but starting with CGPath instead of CGContext
  - Feel free to look them up in the doc

# Defining a Path with UIKit




- **Create a UIBezierPath**

```
let path = UIBezierPath()
```

- **Move around, add lines or arcs to the path**

```
path.move(to: CGPoint(85, 10)) // assume screen is 160x250
path.addLine(to: CGPoint(10, 150))
path.addLine(to: CGPoint(160, 150))
```

- **Close the path (if you want)**

```
path.close()
```

- **Now that you have a path, set attributes and stroke/fill**

```
UIColor.green.setFill() // note this is a method in UIColor, not UIBezierPath
UIColor.red.setStroke() // note this is a method in UIColor, not UIBezierPath
path.linewidth = 3.0 // note this is a property in UIBezierPath, not UIColor
path.fill()
path.stroke()
```

# Drawing with UIKit

- **You can also draw common shapes with UIBezierPath**

```
let roundRect = UIBezierPath(roundedRect: CGRect, cornerRadius: CGFloat)
let oval = UIBezierPath(ovalIn: CGRect)
```

  - ... and others
- **Clipping your drawing to a UIBezierPath's path**
  - addClip()
  - For example, you could clip to a rounded rect to enforce the edges of a playing card
- **Hit detection**

```
func contains(CGPoint) -> Bool // returns whether the point is inside the path
```

  - The path must be closed. The winding rule can be set with usesEvenOddFillRule property.
- **Lots of other stuff, check out docs**

# UIColor

- **Colors are set using UIColor**
  - There are type methods for standard colors, e.g.

```
let green = UIColor.green
```

  - You can also create them from RGB, HSB or even a pattern (using UIImage)
- **Background color of a UIView**

```
var backgroundColor: UIColor
```

- **Colors can have alpha (transparency)**

```
let transparentYellow = UIColor.yellow.withAlphaComponent(0.5)
```

  - Alpha is between 0.0 (fully transparent) and 1.0 (fully opaque)
- **If you want to draw in your view with transparency …**
  - You must let the system know by setting the UIView var opaque = false
- **You can make your entire UIView transparent …**

```
var alpha: CGFloat
```

# View Transparency

- **What happens when views overlap and have transparency?**
  - As mentioned before, subviews list order determines who is in front
  - Lower ones (earlier in the array) can "show through" transparent views on top of them
  - Transparency is not cheap, by the way, so use it wisely
- **Completely hiding a view without removing it from hierarchy**

```
var hidden: Bool
```

  - A hidden view will draw nothing on screen and get no events either
  - Not as uncommon as you might think to temporarily hide a view

# Drawing Text

- **Usually we use a UILabel to put text on screen**
- **But there are certainly occasions where we want to draw text in our drawRect**
- **To draw in drawRect, use NSAttributedString**

```
let text = NSAttributedString("Hello UCD")
text.draw(CGPoint)
let textSize: CGSize = text.size // how much space the string will take up
```

- **Mutability is done with NSMutableAttributedString**
  - It is not like String (i.e. where let means immutable and var means mutable)
  - You use a different class if you want to make a mutable attributed string …

```
let mutableText = NSMutableAttributedString("some string")
```

- **NSAttributedString is not a String, nor an NSString**
  - You can get its contents as an NSString with its String or MutableString property

# Attributed String

- **Setting attributes on an attributed string**

```
func setAttributes(attributes: Dictionary, range: NSRange)
func addAttributes(attributes: Dictionary, range: NSRange)
```

  - Warning! This is a pre-Swift API. NSRange is not a Range.
  - And indexing into the string is using old-style indexing (not String.Index).
- **Attributes**

```
NSForegroundColorAttributeName : UIColor
NSStrokeWidthAttributeName : CGFloat
NSFontAttributeName : UIFont
```

  - See the documentation under NSAttributedString(NSStringDrawing) for (many) more.

# Fonts

- **Since iOS 7, Fonts are fundamental to look and feel of UI and important to get right**
- **The absolutely best way to get a font in code:**
  - Get preferred font for a given text style (e.g. body, etc.) using this UIFont type method …

```
class func preferredFontForTextStyle(UIFontTextStyle) -> UIFont
```

  - Some of the styles (see UIFontDescriptor documentation for more) …

```
UIFontTextStyle.Headline
UIFontTextStyle.Body
UIFontTextStyle.Footnote
```

- **There are also "system fonts"**
  - These appear usually on things like buttons

```
class func systemFontOfSize(pointSize: CGFloat) -> UIFont
class func boldSystemFontOfSize(pointSize: CGFloat) -> UIFont
```

  - Don't use these for your user's content. Use preferred fonts for that.
- **Other ways to get fonts**
  - Check out UIFont and UIFontDescriptor for more, but you should not need that very often

# Drawing Images

- **There is a UILabel-equivalent for images**
  - UIImageView
  - But, again, you might want to draw the image inside your drawRect …
- **Creating a UIImage object**

```
let image: UIImage? = UIImage(named: "UCD_logo") // note that its optional
```

  - You add foo.jpg to your project in the Images.xcassets file (we've ignored this so far)
  - Images will have different resolutions for different devices (all managed in Images.xcassets)
- **You can also create one from files in the file system**
  - (But we haven't talked about getting at files in the file system ... anyway ...)

```
let image: UIImage? = UIImage(contentsOfFile: String)
let image: UIImage? = UIImage(data: NSData) // raw jpg, png, tiff, etc. image data
```

- **You can even create one by drawing with Core Graphics**
  - See documentation for UIGraphicsBeginImageContext(CGSize)
- **Once you have a UIImage, you can blast its bits on screen**

```
let image: UIImage = …
image.drawAtPoint(CGPoint) // the upper left corner of the image put at CGPoint
image.drawInRect(CGRect) // scales the image to fit aCGRect
image.drawAsPatternInRect(CGRect) // tiles the image into aCGRect
```

# Redraw on bounds change

- **By default, when a UIView's bounds changes, there is no redraw**
- **Instead, the 'bits' of the existing image are scaled to the new bounds size**
- **This is often not what you want ...**
  - Luckily, there is a UIView property to control this!
  - It can be set in Xcode too.
  - var contentMode: UIViewContentMode
- **UIViewContentMode**
  - Don't scale the view, just place it somewhere …

```
.Left/.Right/.Top/.Bottom/.TopRight/.TopLeft/.BottomRight/.BottomLeft/.Center
```

  - Scale the "bits" of the view …

```
.ScaleToFill/.ScaleAspectFill/.ScaleAspectFit // .ScaleToFill is the default
```

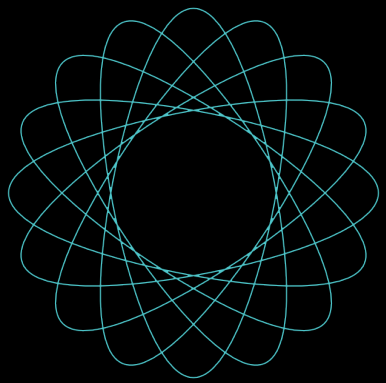
  - Redraw by calling drawRect again
    .Redraw

# Demo

*Spirograph*

# Demo

- **Creating a custom UIView**



# Swift – Protocols

- **A way to express an API minimally**
  - Instead of forcing the caller to pass a class/struct, we can ask for specifically what we want
  - We just specify the properties and methods needed
- **A protocol is a TYPE just like any other type, however:**
  - It has no storage or implementation associated with it
  - Any storage or implementation required to implement the protocol is in an implementing type
  - An implementing type can be any class, struct or enum
  - Otherwise, a protocol can be used as a type to declare variables, as a function parameter, etc.
- **There are three aspects to a protocol**
  1. the protocol declaration (what properties and methods are in the protocol)
  2. the declaration where a class, struct or enum says that it implements a protocol
  3. the actual implementation of the protocol in said class, struct or enum

# Swift – Protocol, Declaration

- **Declaration of the protocol itself**

```
protocol SomeProtocol : InheritedProtocol1, InheritedProtocol2 {
    var someProperty: Int { get set }
    func aMethod(arg1: Double, anotherArgument: String) -> SomeType
    mutating func changeIt()
    init(arg: Type)
}
```

  - Anyone that implements SomeProtocol must also implement InheritedProtocol1 and 2
  - You must specify whether a property is get only or both get and set
  - Any functions that are expected to mutate the receiver should be marked mutating (unless you are going to restrict your protocol to class implementers only with class keyword)
  - You can even specify that implementers must implement a given initializer
  - No implementation there

# Swift – Protocol, Conforming

- **How an implementer says "I implement that protocol"**

```
class SomeClass : SuperclassOfSomeClass, SomeProtocol, AnotherProtocol {
    // implementation of SomeClass here
    // which must include all the properties and methods in SomeProtocol & AnotherProtocol
}
```

  - Claims of conformance to protocols are listed after the superclass for a class

# Swift – Protocol, Conforming

- **How an implementer says "I implement that protocol"**

```
enum SomeEnum : SomeProtocol, AnotherProtocol {
    // implementation of SomeEnum here
    // which must include all the properties and methods in SomeProtocol & AnotherProtocol
}
```

  - Claims of conformance to protocols are listed after the superclass for a class
  - Obviously, enums and structs would not have the superclass part

# Swift – Protocol, Conforming

• **How an implementer says "I implement that protocol"**

```
struct SomeStruct : SomeProtocol, AnotherProtocol {
    // implementation of SomeStruct here
    // which must include all the properties and methods in SomeProtocol & AnotherProtocol
}
```

- Claims of conformance to protocols are listed after the superclass for a class
- Obviously, enums and structs would not have the superclass part
- Any number of protocols can be implemented by a given class, struct or enum

# Swift – Protocol, Conforming

• **How an implementer says "I implement that protocol"**

```
class SomeClass : SuperclassOfSomeClass, SomeProtocol, AnotherProtocol {
    // implementation of SomeClass here, including ...
    required init(...)
}
```

- Claims of conformance to protocols are listed after the superclass for a class
- Obviously, enums and structs would not have the superclass part
- Any number of protocols can be implemented by a given class, struct or enum
- In a class, inits must be marked required (or otherwise a subclass might not conform)

# Swift – Protocol, Conforming

• **How an implementer says "I implement that protocol"**

```
extension Something : SomeProtocol {
    // implementation of SomeProtocol here
    // no stored properties though
}
```

- Claims of conformance to protocols are listed after the superclass for a class
- Obviously, enums and structs would not have the superclass part
- Any number of protocols can be implemented by a given class, struct or enum
- In a class, inits must be marked required (or otherwise a subclass might not conform)
- You are allowed to add protocol conformance via an extension
- Since Swift >1, you are also allowed to extend an existing protocol

# Swift – Protocol Example

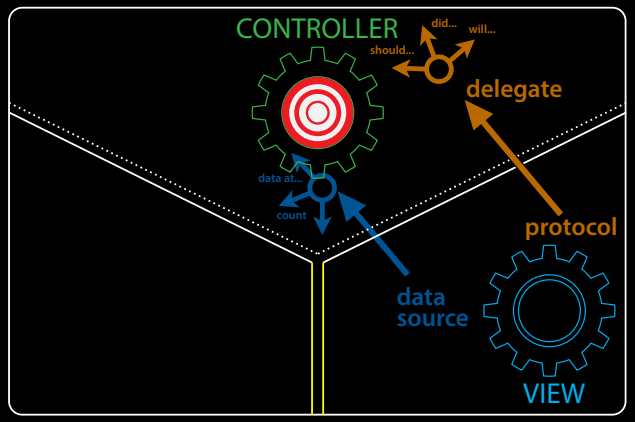
• **Using protocols like the type that they are**

```
protocol Moveable {
    mutating func moveTo(p:CGPoint)
}

class Vehicle : Moveable {
    func moveTo(p: CGPoint) { ... }
    func checkEmission()
}

struct Shape : Moveable {
    mutating func moveTo(p: CGPoint) { ... }
    func draw()
}

let vw: Vehicle = Vehicle()
let circle: Shape = Shape()
```

```
var thingToMove: Moveable = vw
thingToMove.moveTo(...)

thingToMove.checkEmission()
thingToMove = circle

let thingsToMove: [Moveable] = [square, vw]

func slide(_ slider: Moveable) {
    let positionToSlideTo = // ...
    slider.moveTo(positionToSlideTo)
}

slide(circle)
slide(vw)

func slipAndSlide(_ x: protocol<Slippery,
Moveable>)

slipAndSlide(vw)
```

# Swift – Protocol & Delegation

- **A very important use of protocols**
  - It's how we can implement "blind communication" between a View and its Controller




# Swift – Protocol & Delegation

- **A very important use of protocols**
  - It's how we can implement "blind communication" between a View and its Controller
- **How it plays out ...**
  1. Create a delegation protocol (defines what the View wants the Controller to take care of)
  2. Create a delegate property in the View whose type is that delegation protocol
  3. Use the delegate property in the View to get/do things it can't own or control
  4. Controller declares that it implements the protocol
  5. Controller sets self as the delegate of the View by setting the property in #2 above
  6. Implement the protocol in the Controller
- **Now the View is hooked up to the Controller**
  - But the View still has no idea what the Controller is, so the View remains generic/reusable

# Demo

*Spirograph*

# Demo

- **Adding delegation to custom view**
- **Keep view away from model and controller**
- **Set delegate (controller) as data source**

# Gestures

- **We've seen how to draw in a UIView, how do we get touches?**
  - We can get notified of the raw touch events (touch down, moved, up, etc.)
  - Or we can react to certain, predefined "gestures." The latter is the way to go!
- **Gestures are recognized by instances of UIGestureRecognizer**
  - The base class is "abstract." We only actually use concrete subclasses to recognize.
- **There are two sides to using a gesture recogniser**
  1. Adding a gesture recognizer to a UIView (asking the UIView to "recognize" that gesture)
  - Usually the first is done by a Controller
  - Occasionally a UIView will do this itself if the gesture is integral to its existence
  2. Providing a method to "handle" that gesture (not necessarily handled by the UIView)
  - provided either by the UIView or a Controller

# Gestures

- **Adding a gesture recognizer to a UIView**
  - Imagine we wanted a UIView in our Controller's View to recognize a pan gesture ...

```
@IBOutlet weak var pannableView: UIView {
    didSet {
       let recognizer = UIPanGestureRecognizer(target: self,
                                               action: #selector(self.pan(_:)))
        pannableView.addGestureRecognizer(recognizer)
    }
}
```

  - This is just a normal outlet to the UIView we want to recognize the gesture
  - We use its property observer to get involved when the outlet gets hooked up by iOS
  - Here we are creating an instance of a concrete subclass of UIGestureRecognizer (for pans)
  - The target gets notified when the gesture is recognized (in this case, the Controller itself)
  - The action is the method invoked on recognition (the : means it has an argument)
  - Here we ask the UIView to actually start trying to recognize this gesture in its bounds
  - Let's talk about how we implement the handler…

# Gestures

- **A handler for a gesture needs gesture-specific information**
  - So each concrete subclass provides special methods for handling that type of gesture
- **For example, UIPanGestureRecognizer provides 3 methods**

```
func translation(in: UIView?) -> CGPoint // cumulative since start of recognition
func velocity(in: UIView?) -> CGPoint // how fast the finger is moving (points/s)
func setTranslation(CGPoint, in: UIView?)
```

  - This last one is interesting because it allows you to reset the translation so far
  - By resetting the translation to zero all the time, you end up getting "incremental" translation
- **The abstract superclass also provides state information**

```
var state: UIGestureRecognizerState { get  set }
```

  - This sits around in .possible until recognition starts
  - For a discrete gesture (e.g. a Swipe), it changes to .recognized (Tap is not a normal discrete)
  - For a continues gesture (e.g. a Pan), it moves from .began to repeatedly .changed and .ended
  - It can go to .failed or .cancelled too, so watch out for those!

# Gestures

- **So, given this information, what would the pan handler look like?**

```
func pan(gesture: UIPanGestureRecognizer) {
    switch gesture.state {
    case .changed: fallthrough
    case .ended:
        let translation = gesture.translation(in: pannableView)
        // update anything that depends on the pan gesture using translation.x and .y
        gesture.setTranslation(CGPointZero, in: pannableView)
    default: break
    }
}
```

  - Remember that the action was "pan:" (if no colon, we would not get the gesture argument)
  - We are only going to do anything when the finger moves or lifts up off the device's surface
  - fallthrough means "execute the code for the next case down"
  - Here we get the location of the pan in the pannableView's coordinate system
  - Now we do whatever we want with that information
  - By resetting the translation, the next one we get will be how much it moved since this one

# Gestures

- **UIPinchGestureRecognizer**

```
var scale: CGFloat // not read-only (can reset)
var velocity: CGFloat { get } // scale factor per second
```

- **UIRotationGestureRecognizer**

```
var rotation: CGFloat // not read-only (can reset); in radians
var velocity: CGFloat { get } // radians per second
```

- **UISwipeGestureRecognizer**
  - Set up the direction and number of fingers you want, then look for .Recognized

```
var direction: UISwipeGestureRecoginzerDirection // which swipes you want
var numberOfTouchesRequired: Int // finger count
```

- **UITapGestureRecognizer**
  - Set up the number of taps and fingers you want, then look for .Ended

```
var numberOfTapsRequired: Int // single tap, double tap, etc.
var numberOfTouchesRequired: Int // finger count
```

# Demo

*Spirograph*

# Demo

- **Add a gesture recognizer (pinch) to the SpirographView to control spirograph scale**
- **Add a gesture recognizer (pan) to control the parameter range in the Controller**