Stanford CS193p

Developing Applications for iOS
Winter 2015

Today

- Objective-C Compatibility
  - Bridging
- Property List
  - `NSUserDefaults`
  - Demo: `var program` in `CalculatorBrain`
- Views
  - Custom Drawing
- Demo
  - FaceView

Objective-C Compatibility

- Bridging

  iOS was developed in a language called Objective-C.
  Virtually all of the iOS API is accessible seamlessly from Swift.
  A few special data types are handled specially (and powerfully) via bridging.
  Bridging means that you can use the interchangeably.

  `NSString` is bridged to `String`

  `NSArray` is bridged to `Array<AnyObject>`

  `NSDictionary` is bridged to `Dictionary<NSObject, AnyObject>`

  `Int, Float, Double, Bool` are all bridged to `NSNumber` (but not vice-versa)
  To get from `NSNumber` to these types use `doubleValue`, `intValue`, etc.
  `Int`, `Float`, `Double` also bridged to the C types `int`, `float`, `double`

Objective-C Compatibility

- Casting to/from bridged types

  You can also "cast" (reliably, i.e. no need for `as?`) to/from a bridged type

  `let length = (aString as NSString).length` // `length` is an `NSString` method (be careful!)

  `(anArray as NSArray).componentsJoinedByString(NSString)` // `cjbs` is a `NSArray` method

  `String`, `Array` and `Dictionary` are `structs`, not objects (`classes`)

  But they can still be an `AnyObject`

  That's because they are bridged to these NS versions which are objects

  The keys in an `NSDictionary` are `NSObject` subclasses (`Dictionary<NSObject, AnyObject>`)

  But that does not prevent a Swift `String` from being a key in an `NSDictionary`

  That's because it is bridged to `NSString` which does inherit from `NSObject`

  You won't even notice this bridging 99% of the time

Property List

- What is a Property List?

  Property List is really just the definition of a term
  It means an `AnyObject` which is known to be a collection of objects which are ONLY one of …
  `NSString`, `NSArray`, `NSDictionary`, `NSNumber`, `NSData`, `NSDate`
  e.g. an `NSDictionary` whose keys were `NSString` and values were `NSArray` of `NSDate` is one

  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 it's an `AnyObject`, so you have to figure out if it's what you expect

  Property Lists are used to pass around data "blindly"
  The semantics of the contents of a Property List are known only to its creator

  Property Lists are also used as a "generic data structure"
  And so can be passed to API that reads/writes generic data …

NSUserDefaults

- A storage mechanism for Property List data

  It's essentially a very tiny database that stores Propery List data.
  It persists between launchings of your application!
  Great for things like "settings" and such.
  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
```

NSUserDefaults

- Using NSUserDefaults

Get the defaults reader/writer …

```
let defaults = NSUserDefaults.standardUserDefaults()
```

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

- Add a public property, `program`, to `CalculatorBrain`
  - It sets or gets the `CalculatorBrain`'s "program".
  - It will be a Property List (i.e. `AnyObject`, but built from known pieces).
  - Callers won't have any idea what kind of object it is!
  - The value is only good for saving/resetting the `CalculatorBrain`'s program.

Views

- A view (i.e. `UIView` subclass) represents a rectangular area
  - Defines a coordinate space
  - For drawing
  - And for handling touch events
- Hierarchical
  - A view has only one superview … `var superview: UIView?`
  - But it can have many (or zero) subviews … `var subviews: [UIView]` *
    - * aside: it's actually `[AnyObject]`, but on slides, I'll note what is actually in an array
  - The order in the `subviews` 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, very top of the view hierarchy (even includes status bar)
  - Usually only one `UIWindow` in an entire iOS application … it's all about views, not windows

Views

- The hierarchy is most often constructed in Xcode graphically
  - Even custom views are usually 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 `UIView`s will have this view as an ancestor.
  - It's automatically hooked up for you when you create an MVC in Xcode.

Initializing 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) {                // a designed initializer
     super.init(frame: frame)
     setup()
}
required init(coder aDecoder: NSCoder) {  // a required initializer
     super.init(coder: aDecoder)
     setup()
}
```

Initializing a UIView

- 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

Coordinate System Data Structures

- 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 cfg = CGFloat(aDouble)`

- CGPoint

  Simply a struct with two CGFloats in it: x and y.

```
var point = CGPoint(x: 37.0, y: 55.2)
point.y -= 30
point.x += 20.0
```

- CGSize

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

```
var size = CGSize(width: 100.0, height: 50.0)
size.width += 42.5
size.height += 75
```

Coordinate System Data Structures

- CGRect

  A struct with a CGPoint and a CGSize in it …

```
struct CGRect {
     var origin: CGPoint
     var size: CGSize
}
let rect = CGRect(origin: aCGPoint, size: aCGSize) // 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
intersects(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)

(0,0)

increasing x

View Coordinate System

(500, 35)

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

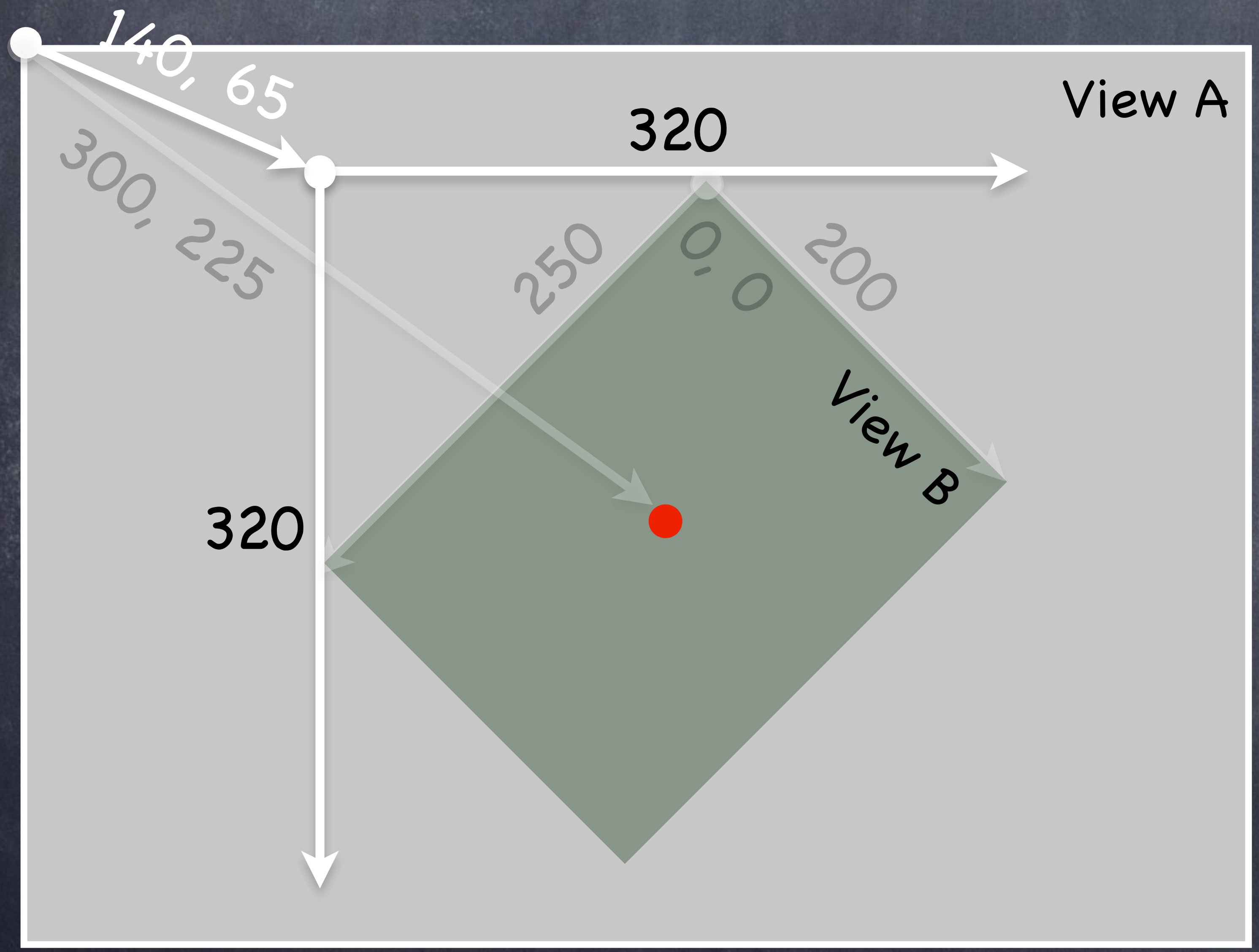
increasing y

bounds vs frame

- Use frame and/or center to position a UIView

  These are never used to draw inside a view's coordinate system (i.e. for positioning only)

  You might think frame.size is always equal to bounds.size, but you'd be wrong …



Views can be rotated (and scaled and translated)

View B's bounds = ((0,0),(200,250))
View B's frame = ((140,65),(320,320))
View B's center = (300,225)

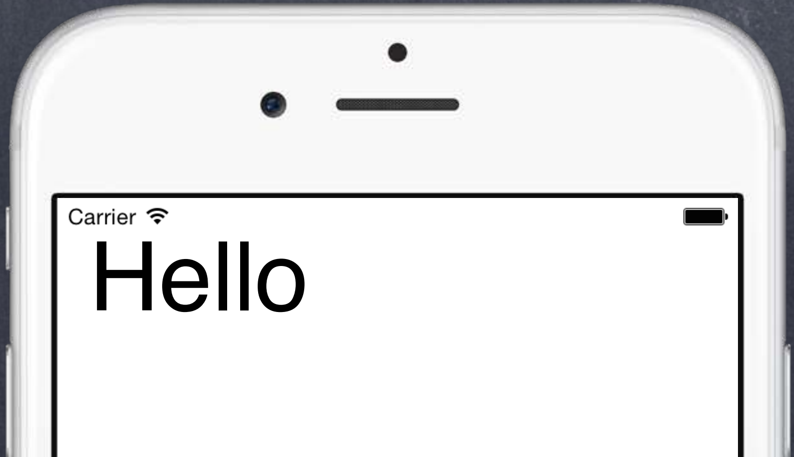
View B's middle in its own coordinates is …
(bounds.midX, bounds.midY) = (100, 125)

Views are rarely rotated, but don't misuse frame or center anyway 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 labelRect = CGRect(x: 20, y: 20, width: 100, height: 50)
let label = UILabel(frame: labelRect) // UILabel is a subclass of UIView
label.text = "Hello"
view.addSubview(label)
```



Custom Views

- When would I 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 in a bit. First we'll focus on drawing.
- To draw, just create a `UIView` subclass and override `drawRect:`
  - `override func drawRect(regionThatNeedsToBeDrawn: CGRect)`
  - You can draw outside the `regionThatNeedsToBeDrawn`, but it's never required
  - The `regionThatNeedsToBeDrawn` is purely an optimization
- NEVER call drawRect!! EVER! Or else!
  - Instead, if you view needs to be redrawn, let the system know that by calling …
  - `setNeedsDisplay()`
  - `setNeedsDisplayInRect(regionThatNeedsToBeRedrawn: CGRect)`
  - iOS will then call your `drawRect` at an appropriate time

Custom Views

- So how do I implement my `drawRect`?
  - You can use a C-like (non object-oriented) API called Core Graphics
  - Or you can use the object-oriented `UIBezierPath` class (which is how we'll do it)
- 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
- `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`

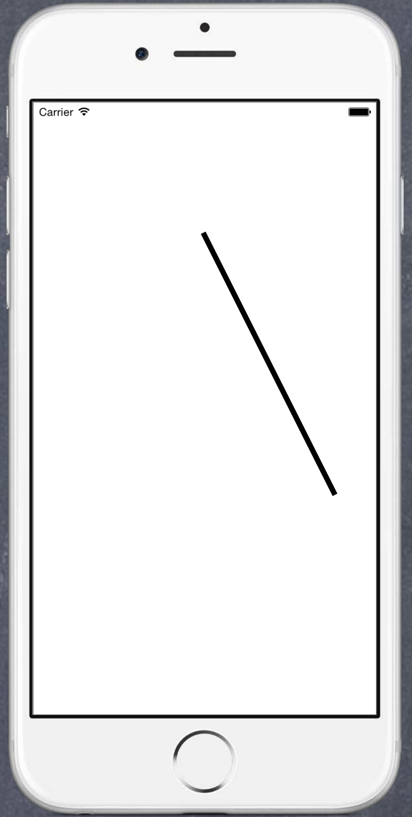
Defining a Path

- Create a UIBezierPath

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

- Move around, add lines or arcs to the path

```
path.moveToPoint(CGPoint(80, 50)) // assume screen is 160x250
path.addLineToPoint(CGPoint(140, 150))
```



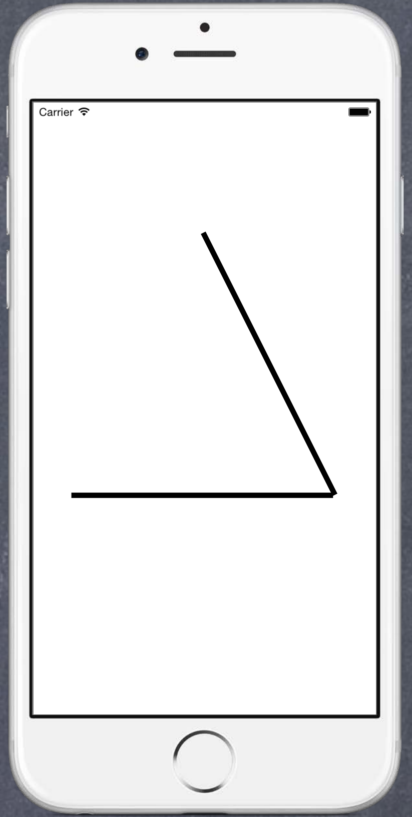
Defining a Path

- Create a UIBezierPath

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

- Move around, add lines or arcs to the path

```
path.moveToPoint(CGPoint(80, 50)) // assume screen is 160x250
path.addLineToPoint(CGPoint(140, 150))
path.addLineToPoint(CGPoint(10, 150))
```



Defining a Path

- Create a UIBezierPath

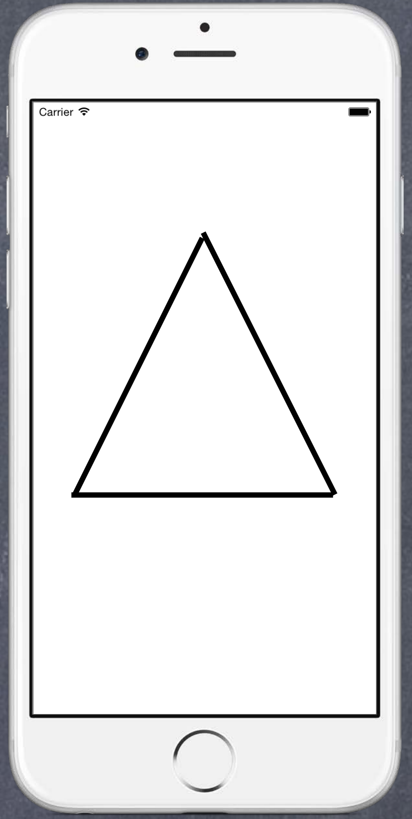
```
let path = UIBezierPath()
```

- Move around, add lines or arcs to the path

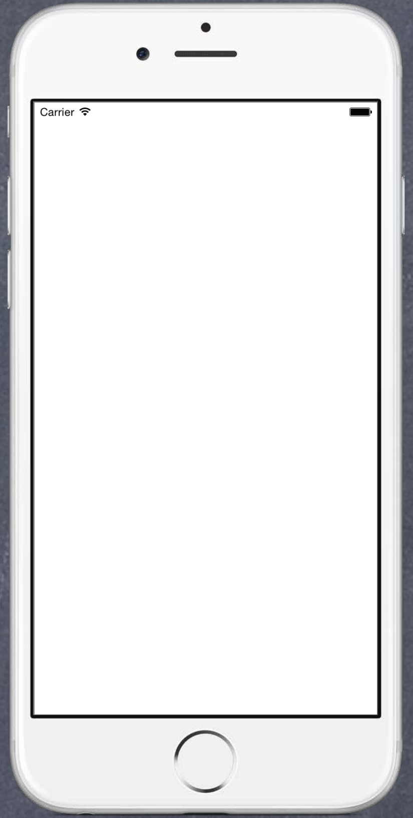
```
path.moveToPoint(CGPoint(80, 50)) // assume screen is 160x250
path.addLineToPoint(CGPoint(140, 150))
path.addLineToPoint(CGPoint(10, 150))
```

- Close the path (if you want)

```
path.closePath()
```



Defining a Path



- Create a UIBezierPath

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

- Move around, add lines or arcs to the path

```
path.moveToPoint(CGPoint(80, 50)) // assume screen is 160x250
path.addLineToPoint(CGPoint(140, 150))
path.addLineToPoint(CGPoint(10, 150))
```

- Close the path (if you want)

```
path.closePath()
```

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

```
UIColor.greenColor().setFill()  // note this is a method in UIColor, not UIBezierPath
UIColor.redColor().setStroke()  // note this is a method in UIColor, not UIBezierPath
path.linewidth = 3.0            // note this is a property in UIBezierPath, not UIColor
```

Defining a Path

- Create a UIBezierPath

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

- Move around, add lines or arcs to the path

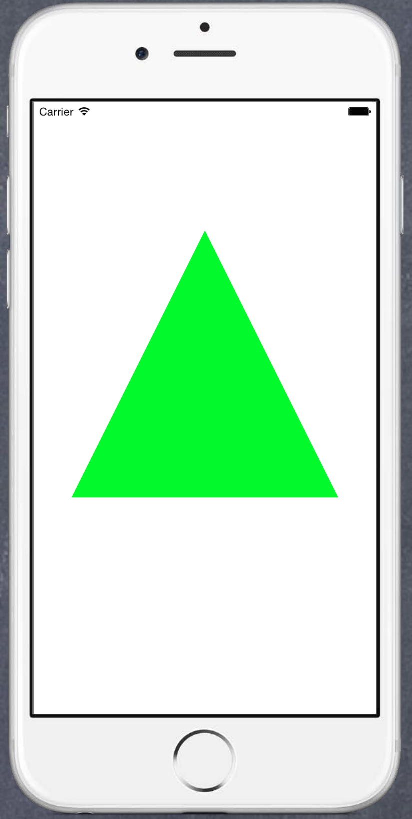
```
path.moveToPoint(CGPoint(80, 50)) // assume screen is 160x250
path.addLineToPoint(CGPoint(140, 150))
path.addLineToPoint(CGPoint(10, 150))
```

- Close the path (if you want)

```
path.closePath()
```

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

```
UIColor.greenColor().setFill()  // note this is a method in UIColor, not UIBezierPath
UIColor.redColor().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()
```



Defining a Path

- Create a UIBezierPath

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

- Move around, add lines or arcs to the path

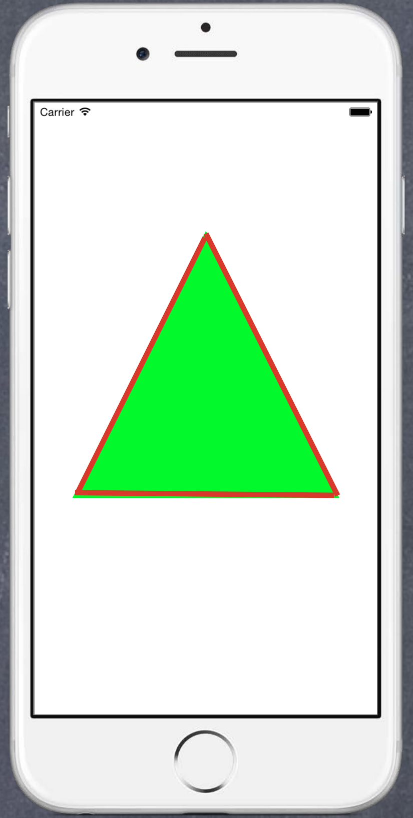
```
path.moveToPoint(CGPoint(80, 50)) // assume screen is 160x250
path.addLineToPoint(CGPoint(140, 150))
path.addLineToPoint(CGPoint(10, 150))
```

- Close the path (if you want)

```
path.closePath()
```

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

```
UIColor.greenColor().setFill()   // note this is a method in UIColor, not UIBezierPath
UIColor.redColor().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

- You can also draw common shapes with UIBezierPath

```
let roundRect = UIBezierPath(roundedRect: aCGRect, cornerRadius: aCGFloat)
let oval = UIBezierPath(ovalInRect: aCGRect)
```

  … 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 containsPoint(CGPoint) -> Bool // returns whether the point is inside the path
```

  The path must be closed. The winding rule can be set with `usesEvenOddFillRule` property.

- Etc.

  Lots of other stuff. Check out the documentation.

UIColor

- Colors are set using UIColor
  - There are type methods for standard colors, e.g. `let green = UIColor.greenColor()`
  - 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.yellowColor().colorWithAlphaComponent(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")
  text.drawAtPoint(aCGPoint)
  let textSize: CGSize = text.size // how much space the string will take up
  ```

- Mutability is done with `NSMutableAttributedString`

  It is <u>not</u> 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`).

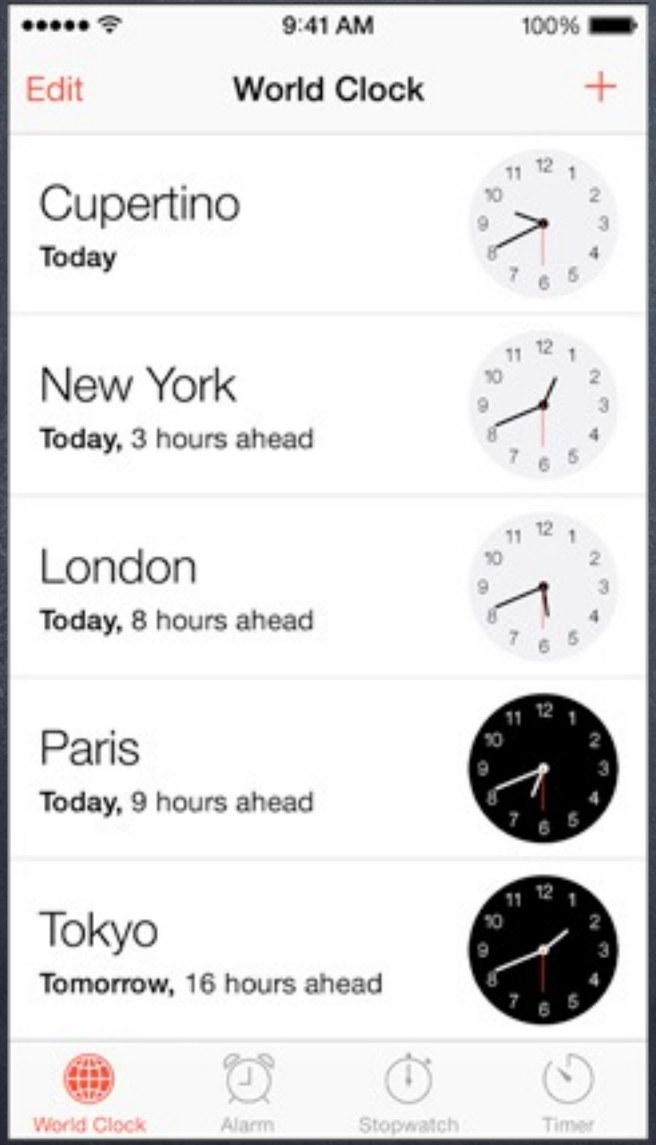
- Attributes

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

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

Fonts

- Fonts in iOS 7 and later are very important to get right
  - They are fundamental to the look and feel of the UI



9:41 AM 100%

Edit World Clock +

Cupertino
Today

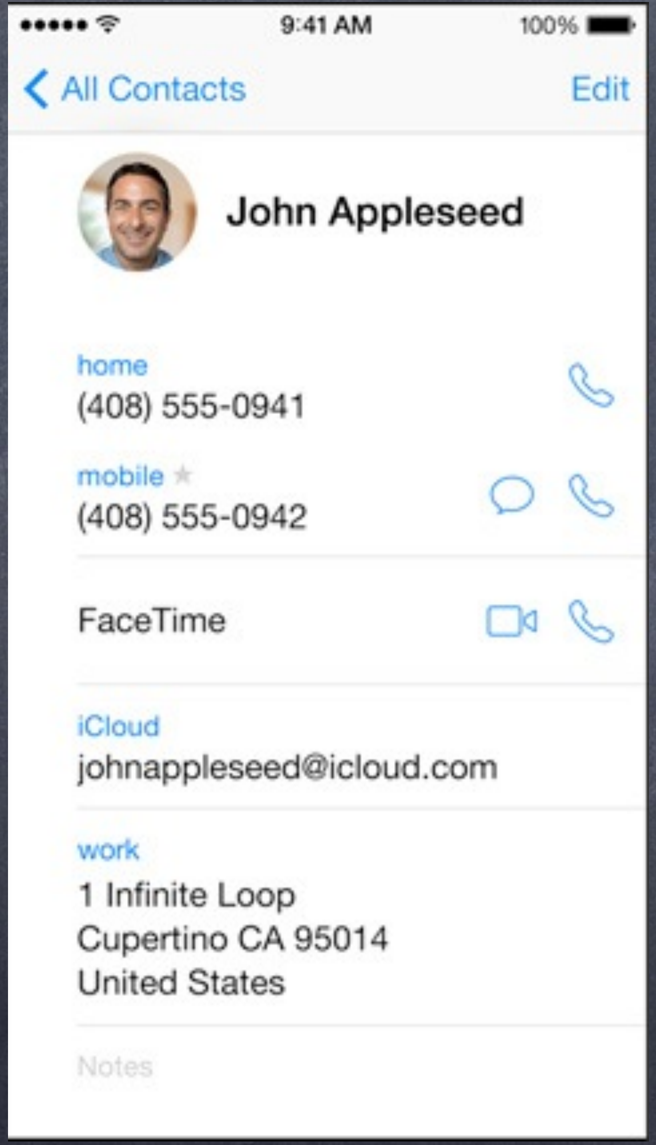
New York
Today, 3 hours ahead

London
Today, 8 hours ahead

Paris
Today, 9 hours ahead

Tokyo
Tomorrow, 16 hours ahead

World Clock Alarm Stopwatch Timer



9:41 AM 100%

All Contacts Edit

John Appleseed

home
(408) 555-0941

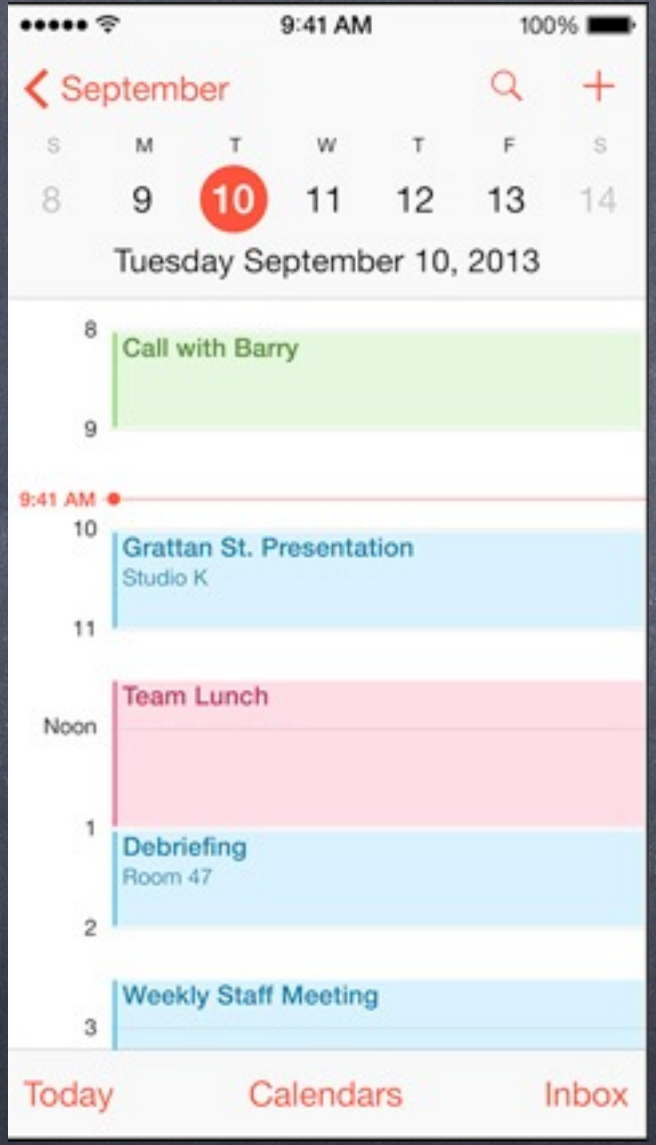
mobile
(408) 555-0942

FaceTime

iCloud
johnappleseed@icloud.com

work
1 Infinite Loop
Cupertino CA 95014
United States

Notes



9:41 AM 100%

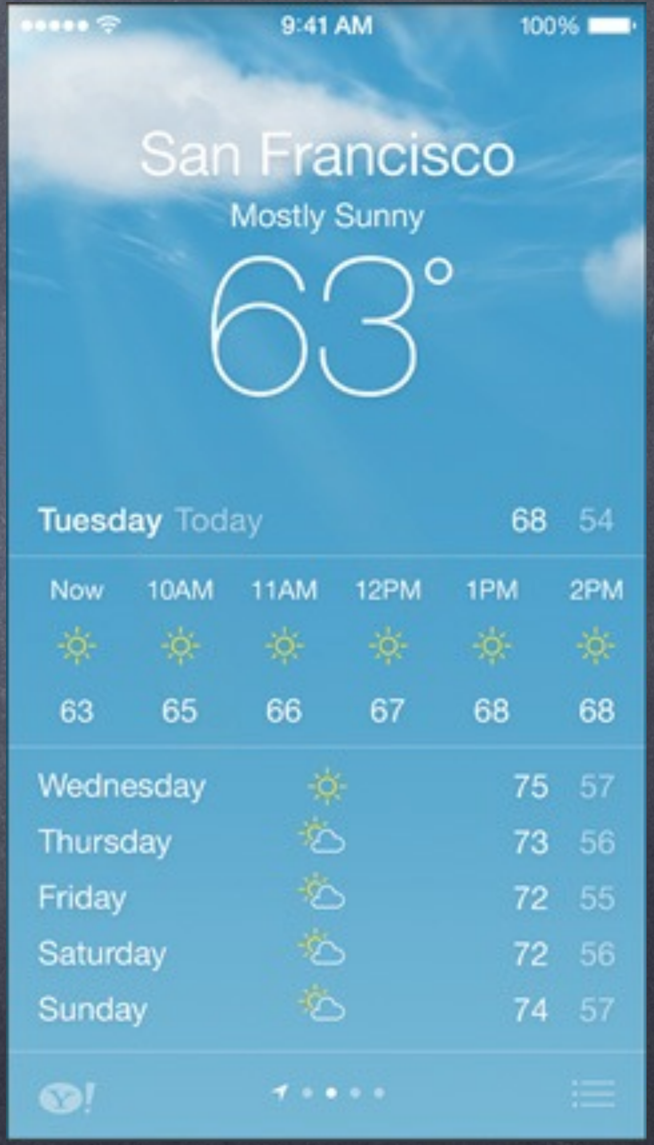
September

S	M	T	W	T	F	S
8	9	10	11	12	13	14

Tuesday September 10, 2013

8 Call with Barry
9
9:41 AM
10 Grattan St. Presentation
Studio K
11
Noon Team Lunch
1 Debriefing
Room 47
2
3 Weekly Staff Meeting

Today Calendars Inbox



9:41 AM 100%

San Francisco
Mostly Sunny
63°

Tuesday Today 68 54

Now	10AM	11AM	12PM	1PM	2PM
63	65	66	67	68	68

Wednesday	75	57
Thursday	73	56
Friday	72	55
Saturday	72	56
Sunday	74	57

Fonts

- 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: “foo”)` // 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: aString)`

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

- You can even create one by drawing with Core Graphics

  See documentation for UIGraphicsBeginImageContext(CGSize)

Drawing Images

- Once you have a UIImage, you can blast its bits on screen

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

- FaceView

  Create a custom `UIView` subclass to draw a face with a specified amount of “smiliness”