Stanford CS193p

Developing Applications for iOS
Fall 2017-18

Today

- Miscellaneous
  - Error Handling
  - Any
  - Other Interesting Classes
- Views
  - Custom Drawing
- Demo: Draw a Playing Card
  - enum

Thrown Errors

- In Swift, methods can throw errors

  You will always know that can `throw` because they'll have the keyword `throws` on the end.

  ```
  func save() throws
  ```

  You must put calls to functions like this in a `do { }` block and use the word `try` to call them.

  ```
  do {
      try context.save()
  } catch let error {
     // error will be something that implements the Error protocol, e.g., NSError
     // usually these are enums that have associated values to get error details
     throw error // this would re-throw the error (only ok if the method we are in throws)
  }
  ```

  If you are certain a call will not `throw`, you can force `try` with `try!` …

  ```
  try! context.save() // will crash your program if save() actually throws an error
  ```

  Or you can conditionally `try`, turning the return into an Optional (which will be `nil` if fail) …

  ```
  let x = try? errorProneFunctionThatReturnsAnInt() // x will be Int?
  ```

Any & AnyObject

- `Any` & `AnyObject` are special types
  - These types used to be commonly used for compatibility with old Objective-C APIs
  - But not so much anymore in iOS 11 since those old Objective-C APIs have been updated (though we did see it with `NSAttributedString`)
  - Variables of type `Any` can hold something of any type (`AnyObject` holds `class`es only).
  - Swift is a strongly typed language, though, so you can't invoke a method on an `Any`.
  - You have to convert it into a concrete type first.
  - One of the beauties of Swift is its strong typing, so generally you want to avoid `Any`.

Any & AnyObject

- Where will you see it in iOS?

  We already saw it in NSAttributedString.

  `let attributes: [NSAttributedStringKey:Any] = …`

  The attributes of an NSAttributedString can be different things (UIColor, UIFont, etc.).

  So the type Any was used as the type of the values in the attributes Dictionary.

  Sometimes (rarely) Any will be the type of a function's argument.

  Here's a UIViewController method that includes a sender (which can be of any type).

  `func prepare(for segue: UIStoryboardSegue, sender: Any?)`

  The sender is the thing that caused this "segue" (i.e., a move to another MVC) to occur.

  The sender might be a UIButton or a UITableViewCell or some custom thing in your code.

  It's an Optional because it's okay for a segue to happen without a sender being specified.

  These are old Objective-C API.

  In Swift we would probably would have used an enum with associated data or a protocol.

Any & AnyObject

- Don't use `Any` in this course
  - You can, of course, call iOS API that uses it.
  - But don't use it for any of your own internal data structure work.

Any & AnyObject

- How do we use a variable of type Any?

  We can't use it directly (since we don't know what type it really is).
  Instead, we must convert it to another, known type.

  Conversion is done with the as? keyword in Swift.
  This conversion might not be possible, so the conversion generates an Optional.
  You can also check to see if something can be converted with the is keyword (true/false).

  We almost always use as? it with if let ...

```
let unknown: Any = … // we can't send unknown a message because it's "typeless"
if let foo = unknown as? MyType {
    // foo is of type MyType in here
    // so we can invoke MyType methods or access MyType vars in foo
    // if unknown was not of type MyType, then we'll never get here
}
```

Casting

- By the way, casting with `as?` is not just for `Any` & `AnyObject`

  You can cast any type with `as?` into any other type that makes sense.
  Mostly this would be casting an object from one of its superclasses down to a subclass.
  But it could also be used to cast any type to a `protocol` it implements (more on this later).

  Example of "downcasting" from a superclass down to a subclass …

```
let vc: UIViewController = ConcentrationViewController()
```

  The type of `vc` is `UIViewController` (because we explicitly typed it to be).
  And the assignment is legal because a `ConcentrationViewController` is a `UIViewController`.
  But we can't say, for example, `vc.flipCard(…)`, since `vc` is typed as a `UIViewController`.

  However, if we cast `vc` to be a `ConcentrationViewController`, then we can use it …

```
if let cvc = vc as? ConcentrationViewController {
    cvc.flipCard(…) // this is okay
}
```

Other Interesting Classes

- NSObject
  - Base class for all Objective-C classes
  - Some advanced features will require you to subclass from `NSObject` (and it can't hurt to do so)
- NSNumber
  - Generic number-holding `class` (i.e., reference type)
  - `let n = NSNumber(35.5)` or `let n: NSNumber = 35.5`
  - `let intified: Int = n.intValue` // also `doubleValue`, `boolValue`, etc.
- Date
  - Value type used to find out the date and time right now or to store past or future dates
  - See also `Calendar`, `DateFormatter`, `DateComponents`
  - If you are displaying a date in your UI, there are localization ramifications, so check these out!
- Data
  - A value type "bag o' bits". Used to save/restore/transmit raw data throughout the iOS SDK.

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]`
  - 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

```
func addSubview(_ view: UIView)// sent to view's (soon to be) superview
func removeFromSuperview()     // 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.

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 designated initializer
    super.init(frame: frame)
    setup()                                     // might have to be before super.init
}
required init?(coder aDecoder: NSCoder) {   // a required, failable 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

  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 to/from a Double or Float using initializers, e.g., `let cgf = 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

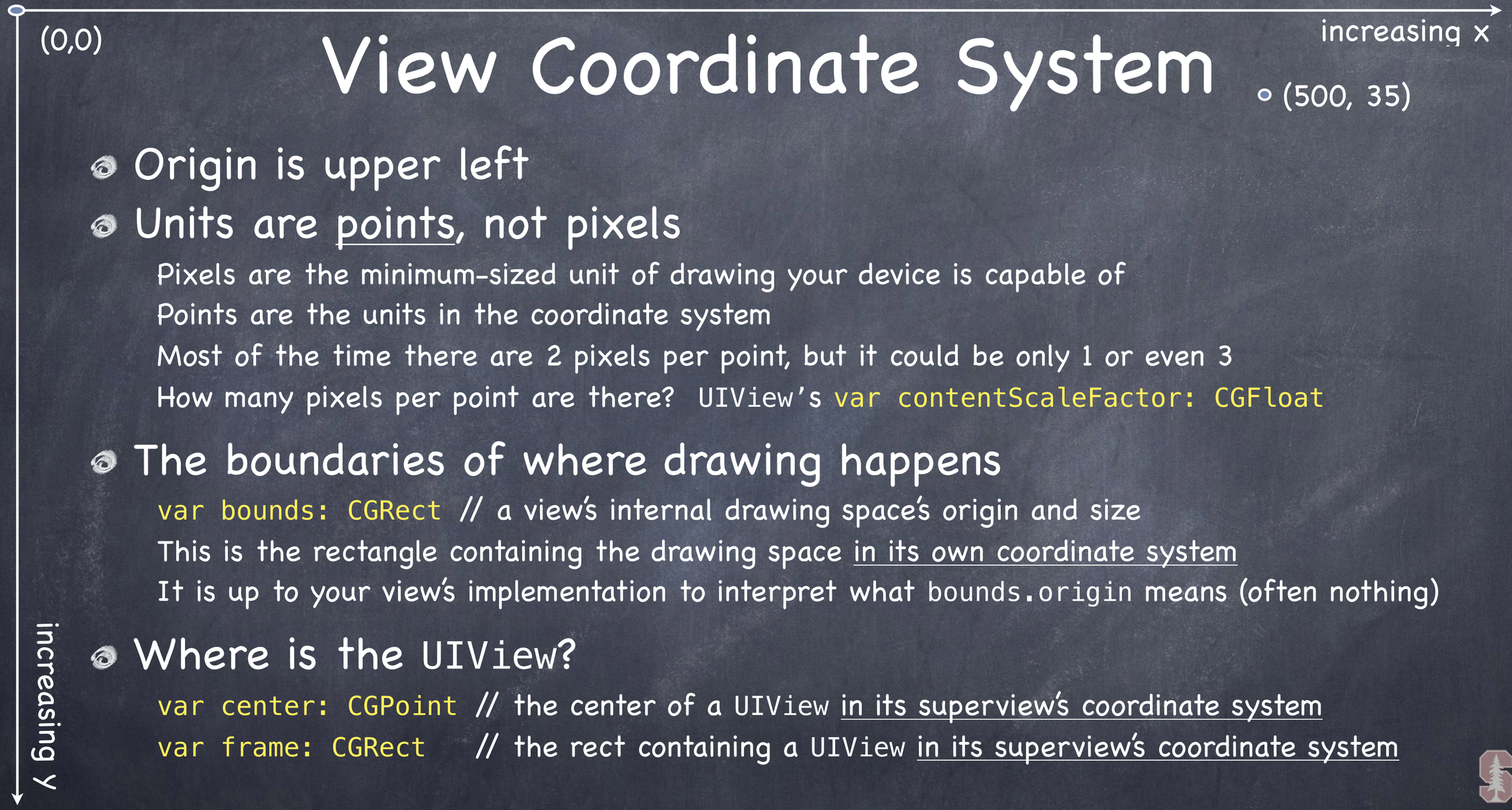
  A struct with a CGPoint and a CGSize in it …

```
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)

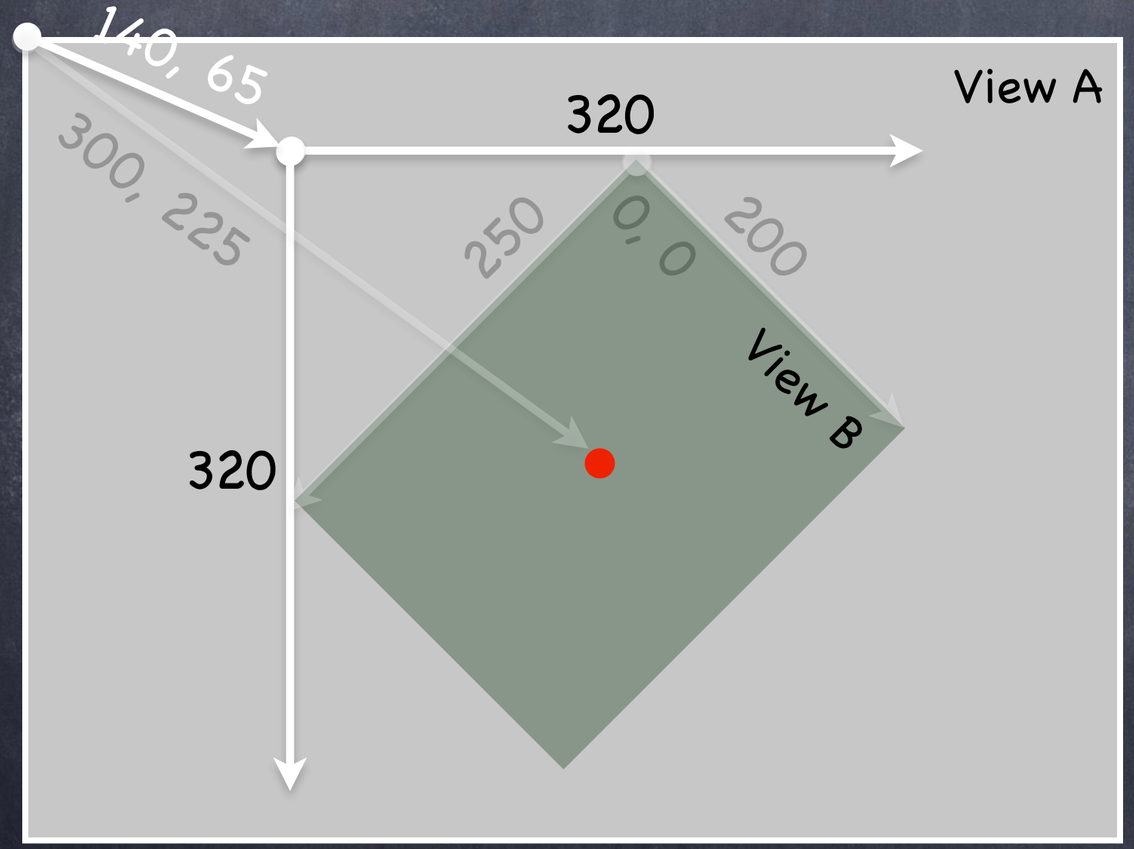


View Coordinate System

- 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 even 3
  - 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

bounds vs frame

- Use `frame` and/or `center` to position a UIView
  - These are never used to draw inside a view's coordinate system
  - 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)`

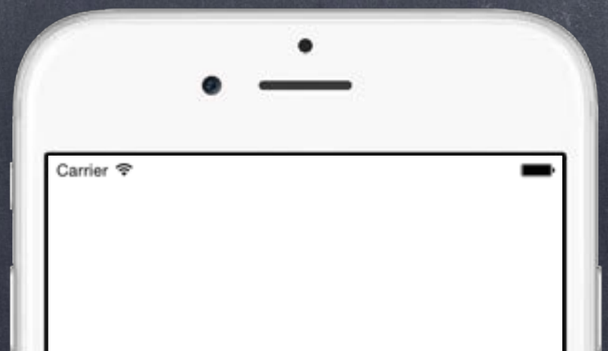
View B's middle in its own coordinates is …
`(bounds.midX, bounds.midY) = (100, 125)`

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 `CGRect.zero`)
- Example

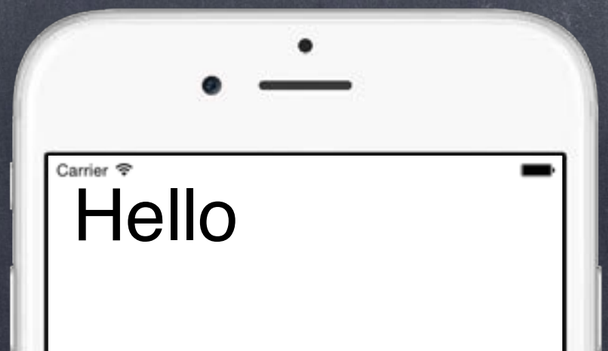
```
let labelRect = CGRect(x: 20, y: 20, width: 100, height: 50)
let label = UILabel(frame: labelRect) // UILabel is a subclass of UIView
label.text = “Hello”
```



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 `CGRect.zero`)
- Example

```
// assuming this code is in a UIViewController (and thus the var view is the root view)
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 `draw(CGRect)`
  - `override func draw(_ rect: CGRect)`
  - You can draw outside the `rect`, but it's never required to do so.
  - The `rect` is purely an optimization.
  - It is our `UIView`'s `bounds` that describe the entire drawing area (the `rect` is a subarea).
- NEVER call `draw(CGRect)`!! EVER! Or else!
  - Instead, if you view needs to be redrawn, let the system know that by calling ...
  - `setNeedsDisplay()`
  - `setNeedsDisplay(_ rect: CGRect)` // `rect` is the area that needs to be redrawn
  - iOS will then call your `draw(CGRect)` at an appropriate time

Custom Views

- So how do I implement my `draw(CGRect)`?
  - You can either get a drawing context and tell it what to draw, or …
  - You can create a path of drawing using `UIBezierPath` class
- Core Graphics Concepts
  1. You get a context to draw into (other contexts include printing, off-screen buffer, etc.)
     The function `UIGraphicsGetCurrentContext()` gives a context you can use in `draw(CGRect)`
  2. Create paths (out of lines, arcs, etc.)
  3. Set drawing attributes like colors, fonts, textures, linewidths, linecaps, etc.
  4. 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 (preset up for you in `draw(CGRect)`)
  - `UIBezierPath` has methods to draw (lineto, arcs, etc.) and to set attributes (linewidth, etc.)
  - Use `UIColor` to set stroke and fill colors
  - `UIBezierPath` has methods to `stroke` and/or `fill`

Defining a Path

- Create a UIBezierPath

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

Defining a Path

- Create a UIBezierPath

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

- Move around, add lines or arcs to the path

```
path.move(to: CGPoint(80, 50))
```

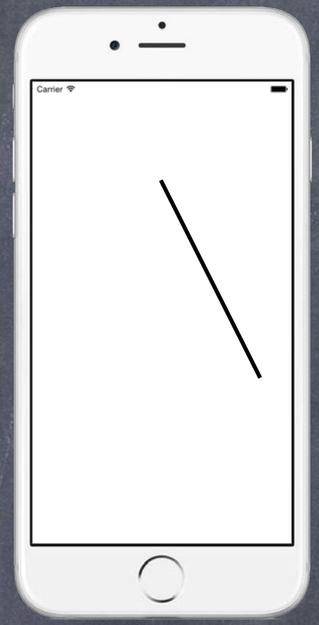
Defining a Path

- Create a UIBezierPath

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

- Move around, add lines or arcs to the path

```
path.move(to: CGPoint(80, 50))
path.addLine(to: CGPoint(140, 150))
```



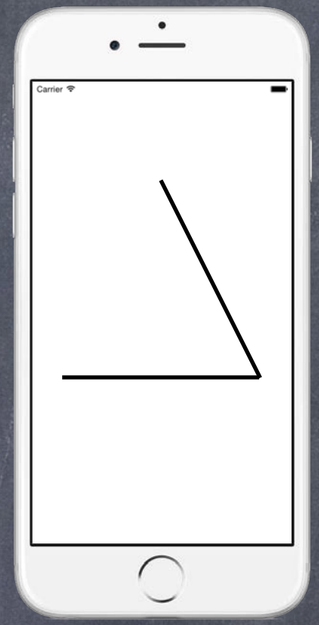
Defining a Path

- Create a UIBezierPath

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

- Move around, add lines or arcs to the path

```
path.move(to: CGPoint(80, 50))
path.addLine(to: CGPoint(140, 150))
path.addLine(to: CGPoint(10, 150))
```



Defining a Path

- Create a UIBezierPath

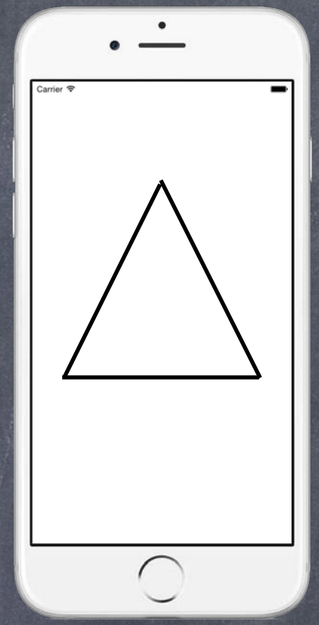
```
let path = UIBezierPath()
```

- Move around, add lines or arcs to the path

```
path.move(to: CGPoint(80, 50))
path.addLine(to: CGPoint(140, 150))
path.addLine(to: CGPoint(10, 150))
```

- Close the path (if you want)

```
path.close()
```



Defining a Path

- Create a UIBezierPath

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

- Move around, add lines or arcs to the path

```
path.move(to: CGPoint(80, 50))
path.addLine(to: CGPoint(140, 150))
path.addLine(to: CGPoint(10, 150))
```

- Close the path (if you want)

```
path.close()
```

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

Defining a Path

- Create a UIBezierPath

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

- Move around, add lines or arcs to the path

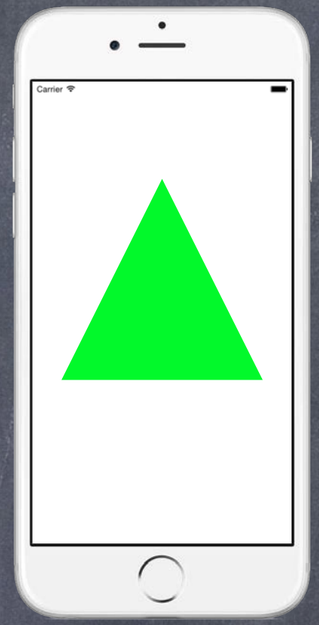
```
path.move(to: CGPoint(80, 50))
path.addLine(to: CGPoint(140, 150))
path.addLine(to: CGPoint(10, 150))
```

- Close the path (if you want)

```
path.close()
```

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

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



Defining a Path

- Create a UIBezierPath

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

- Move around, add lines or arcs to the path

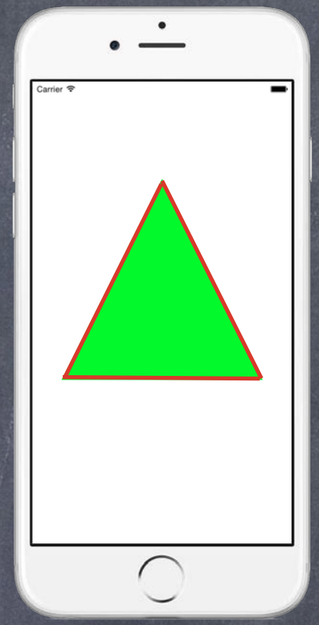
```
path.move(to: CGPoint(80, 50))
path.addLine(to: CGPoint(140, 150))
path.addLine(to: CGPoint(10, 150))
```

- Close the path (if you want)

```
path.close()
```

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

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



Drawing

- You can also draw common shapes with UIBezierPath

  ```
  let roundedRect = 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(_ point: 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 (aka `static`) vars 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` // we used this for our Concentration buttons
- Colors can have alpha (transparency)
  - `let semitransparentYellow = 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`

Layers

- Underneath `UIView` is a drawing mechanism called `CALayer`

  You usually do not care about this.

  But there is some useful API there.

  You access a `UIView`'s "layer" using this var ...

  ```
  var layer: CALayer
  ```

  The `CA` in `CALayer` stands for "Core Animation".

  Mostly we can do animation in a `UIView` without accessing this layer directly.

  But it is where the actual animation functionality of `UIView` is coming from.

  We'll talk about animation next week.

  But `CALayer` can do some cool non-animation oriented things as well, for example ...

  ```
  var cornerRadius: CGFloat    // make the background a rounded rect
  var borderWidth: CGFloat     // draw a border around the view
  var borderColor: CGColor?    // the color of the border (if any)
  ```

  You can get a `CGColor` from a `UIColor` using `UIColor`'s `cgColor` var.

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 isHidden: Bool`
  - An `isHidden` 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 `draw(CGRect)`
- To draw in `draw(CGRect)`, use `NSAttributedString`

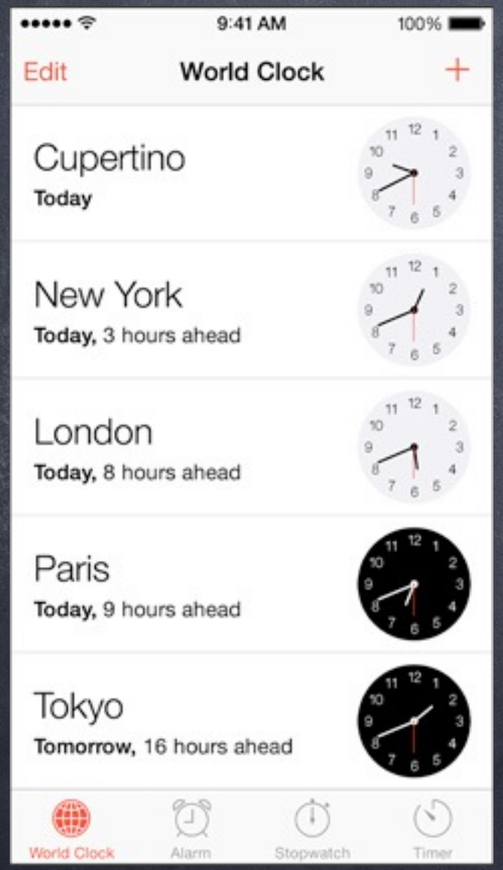
```
let text = NSAttributedString(string: “hello”) // probably would set some attributes too
text.draw(at: aCGPoint)                        // or draw(in: CGRect)
let textSize: CGSize = text.size               // how much space the string will take up
```

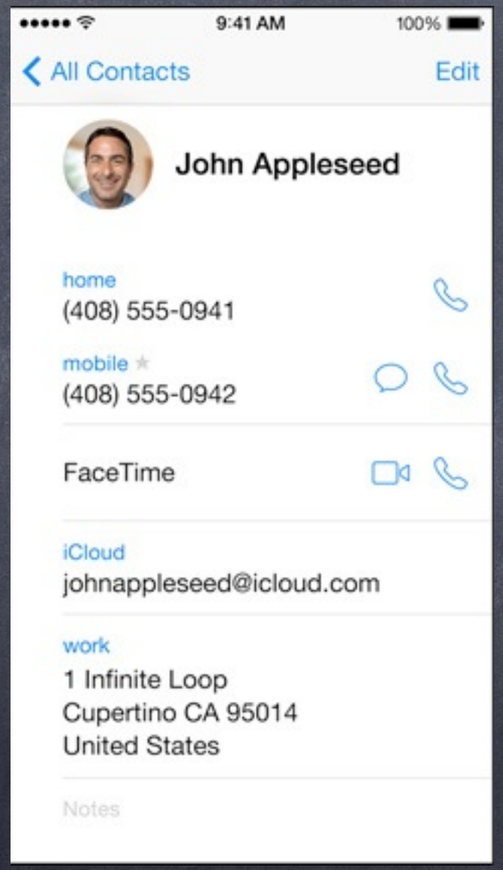
- Accessing a range of characters in an `NSAttributedString`
  - `NSRange` has an `init` which can handle the `String` vs. `NSString` weirdness …

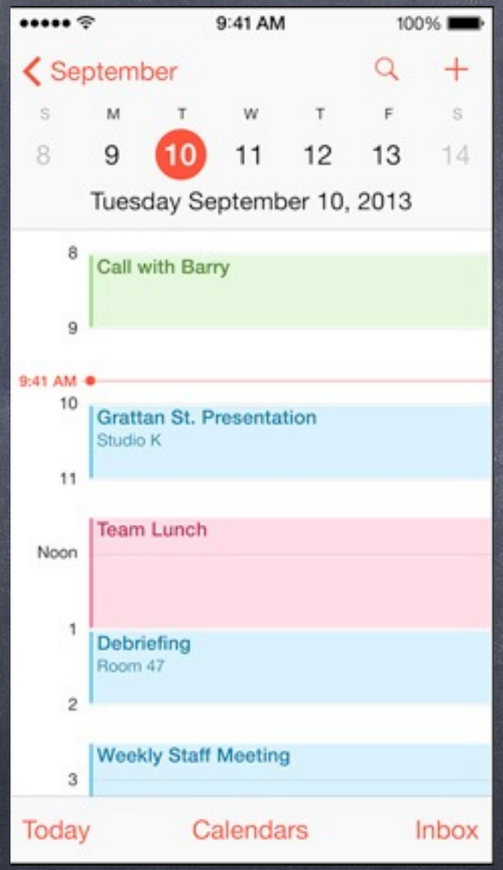
```
let pizzaJoint = “café pesto”
var attrString = NSMutableAttributedString(string: pizzaJoint)
let firstWordRange = pizzaJoint.startIndex..<pizzaJoint.indexOf(“ “)!
let nsrange = NSRange(firstWordRange, in: pizzaJoint) // convert Range<String.Index>
attrString.addAttribute(.strokeColor, value: UIColor.orange, range: nsrange)
```

Fonts

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







Fonts

- Usually you set fonts in UI elements like `UIButton`, `UILabel`, etc.
  - Be sure to choose a "preferred font" for user content in the Inspector in InterfaceBuilder.
  - User content is information generated by or requested by the user (not button titles, etc.).
  - But you might want to set a font in an `NSAttributedString` too …
- Simple way to get a font in code
  - Get preferred font for a given text style (e.g. body, etc.) using this `UIFont` type method …

```
static func preferredFont(forTextStyle: UIFontTextStyle) -> UIFont
```

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

```
UIFontTextStyle.headline
               .body
               .footnote
```

  - Importantly, the size of the font you get is determined by user settings (esp. for Accessibility).
  - You'll want to make sure your UI looks good with all size fonts!

Fonts

- More advanced way …

  Choose a specific font by name …

  ```
  let font = UIFont(name: “Helvetica”, size: 36.0)
  ```

  You can also use the `UIFontDescriptor` class to get the font you want.

  Now get metrics for the text style you want and scale `font` to the user's desired size …

  ```
  let metrics = UIFontMetrics(forTextStyle: .body) // or UIFontMetrics.default
  let fontToUse = metrics.scaledFont(for: font)
  ```

Fonts

- There are also "system fonts"

  These appear usually on things like buttons.

```
static func systemFont(ofSize: CGFloat) -> UIFont
static func boldSystemFont(ofSize: CGFloat) -> UIFont
```

  But again, don't use these for your user's content. Use preferred fonts for that.

Drawing Images

- There is a UILabel-equivalent for images
  - UIImageView
  - But, again, you might want to draw the image inside your draw(CGRect) …
- Creating a UIImage object
  - `let image: UIImage? = UIImage(named: "foo")` // note that its an Optional
  - You add foo.jpg to your project in the Assets.xcassets file (we've ignored this so far)
  - Images will have different resolutions for different devices (all managed in Assets.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: pathString)`
  - `let image: UIImage? = UIImage(data: aData)` // 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.draw(at point: aCGPoint)          // the upper left corner put at aCGPoint
image.draw(in rect: aCGRect)            // scales the image to fit aCGRect
image.drawAsPattern(in rect: 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 the bits (intact) 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 `draw(CGRect)` again (costly, but for certain content, better results) …
  - `.redraw`

Layout on bounds change?

- What about your subviews on a bounds change?

  If your bounds change, you may want to reposition some of your subviews

  Usually you would set this up using Autolayout constraints

  Or you can manually reposition your views when your bounds change by overriding …

```
override func layoutSubviews() {
    super.layoutSubviews()
    // reposition my subviews's frames based on my new bounds
}
```

Demo Code

Download the demo code from today's lecture.