Announcements

• Lab 2 is due next Monday (Sept 23rd) by 11:59 PM
  – Make sure your lab does not contain any warnings when you submit it

Today’s Topics

• Additional Swift Concepts
• Views
• Drawing
• Text & Images
Lazy Initialization of Properties (CS193p)

- Lazy properties do not get initialized until someone accesses them
- You can allocate objects, execute a closure, or call a method

```swift
lazy var theResult = LotsOfWorkObject()

lazy var someProperty: Type = {
    // construct the value of someProperty here
    return (the constructed value)
}()

lazy var myProperty = self.initializeMyProperty()
```

Initialization in Swift

- Classes and structures must set all of their stored properties when created
- Various way to set properties (without an init)
  - Define default values
  - Properties may be Optional (so they start out as nil)
  - Initialize a property by setting a closure
  - Use lazy instantiation
- Use an init when values can not be set using the previous examples
  - You can have as many init methods in your class or struct
  - Each init will have different arguments
Initialization (CS193p)

• Some init methods are for free
  – Free init() given to all base classes
    • A base class has no superclass
  – If a struct has no initializers, it will get a default one
    will all properties as arguments

Initialization (CS193p)

• What can I do with an init?
  – Set property values, even those that already had defaults
  – Constant properties (those declared with let) can be set
  – You can call other init methods in your own class or struct
    using self.init(args)
  – In a class, you can also call super.init(args)
    • There are some rules for calling inits from other inits in a class
Class Initialization Requirements (CS193p)

- After init completes all properties must have values (Optionals can be nil)
- A class has two types of inits
  - Convenience and designated
- Designated init
  - Must (and can only) call a designated init in its immediate superclass
  - You must initialize all properties introduced by your class before calling a superclass’s init
  - You must call a superclass’s init before you assign a value to an inherited property
- Convenience init
  - Must (and can only) call an init in its own class
  - Must call that init before it can set any property values
  - The call of other inits must be completed before you can access properties or invoke methods

Initialization (CS193p)

- Inheriting init
  - If you do not implement any designated inits, you will inherit all of your superclass’s designated inits
  - If you override all of your superclass’s designated inits, you’ll inherit all its convenience inits
  - If you implement no inits, you will inherit all of your superclass’s inits
  - Any init inherited by these rules qualifies to satisfy any of the rules on the previous slide
- Required init
  - A class can mark one or more of its init methods as required
  - Any subclass must implement those init methods
    - They can be inherited per rules above
Failable init (CS193p)

• If an init is declared with a ? after the word init, it returns an Optional
  
  ```swift
  init? (arg1: Type1,..) {
    // might return nil here (means init failed)
  }
  ```

• Example
  Let image = UIImage(named: "foo")  // image is Optional UIImage

• Typically use if-let for these cases
  If let image = UIImage(named: "foo") {
    // image was successfully created
  } else {
    // failed to create image
  }

Demo
Other Swift Concepts

- **Swift Syntax**
  - Guard
  - Defer

- **Swift Error Handling**
  - [https://docs.swift.org/swift-book/LanguageGuide/ErrorHandling.html#](https://docs.swift.org/swift-book/LanguageGuide/ErrorHandling.html#)

Guard

- Used to transfer program control out of a scope, if one or more conditions are not met
  - Early exit

- Improves readability of code
  - Avoid the if let, if let, if let

```swift
guard condition else {
  statements
}
```
Defer

- Used to execute a set of statements just before code execution leaves the current block of code
  - “defers” execution until the current scope is exited

defer {
  statements
}
Swift supports throwing, catching, propagating and manipulating recoverable errors at runtime.

Helpful when an operation does not complete execution or fails to provide useful output.

Represent and Throw Errors

Errors are represented by values of types conforming to the Error Protocol.

```swift
eenum VendingMachineError: Error {
    case invalidSelection
    case insufficientFunds(coinsNeeded: Int)
    case outOfStock
}

throw VendingMachineError.insufficientFunds(coinsNeeded: 5)
```
Four Ways to Handle Errors

- Propagate errors from a function to the code calling it
- Handle the error using a do-catch statement
- Handle the error as an optional value (try?)
- Assert the error will not occur (try!)

Propagating Errors

- To specify that a function, method, or initializer can throw an error, you write the throws keyword

```swift
func canThrowError() throws -> String
func cannnotThrowErrors() -> String

func buy(itemNumber: Int) throws -> String {
    if itemNumber > 20 {
        throw VendingMachineError.invalidNumber
    }
    return "Coke"
}
```
Error Handling – Do-Catch

- When handling errors in code use a do-catch statement of the following form:

  ```
  do {
  try expression
  statements
  } catch pattern 1 {
  statements
  } catch pattern 2 where condition {
  statements
  } catch {
  statements // all other error conditions
  }
  ```

- Not necessary to catch all conditions here, as error will propagate to surrounding scope
  - Must be caught by some surround scope

Handling Error as an optional value

- Use the `try?` to handle an error by converting it to an optional
  - If an error is thrown while evaluating the `try?` it will return nil.

  ```swift
  func someThrowingFunction() throws -> Int {
    // ...
  }
  let x = try? someThrowingFunction()
  ```

- Equivalent to writing the following code

  ```swift
  let y: Int?
  do {
    y = try someThrowingFunction()
  } catch {
    y = nil
  }
  ```
Disable Error propagation

- If you know that a function will not throw an error, you can disable error propagation.
  - If the error is thrown you will get a runtime error

```swift
let photo = try! loadImage(atPath: "/Resources/John Appleseed.jpg")
```
Views

View Fundamentals

- Rectangular area on screen
- Draws content
- Handles events
- Subclass of UIResponder (event handling class)
- Views arranged hierarchically
  - every view has one superview
  - every view has zero or more subviews
**View Hierarchy - UIWindow**

- Views live inside of a window

- UIWindow is actually just a view
  - adds some additional functionality specific to top level view

- One UIWindow for an iOS app
  - Contains the entire view hierarchy
  - Set up by default in Xcode template project

**View Hierarchy - Manipulation**

- Add/remove views in Storyboard or using UIView methods
  ```swift
  func addSubview(UIView)
  func removeFromSuperview()
  ```

- Manipulate the view hierarchy manually:
  ```swift
  func insertSubview(UIView, at: Int)
  func insertSubview(UIView, belowSubview: UIView)
  func insertSubview(UIView, aboveSubview: UIView)
  func exchangeSubview(at: Int, withSubviewAt: Int)
  ```
View-related Structures

- **CGPoint**
  - location in space: \( \{ x, y \} \)
  - sometimes used as an origin

- **CGSize**
  - dimensions: \( \{ \text{width}, \text{height} \} \)

- **CGRect**
  - location and dimension: \( \{ \text{origin}, \text{size} \} \)
**View-related Structure**

<table>
<thead>
<tr>
<th>Creation Function</th>
<th>Example</th>
</tr>
</thead>
</table>
| CGPoint(x: Double, y: Double) | var point = CGPoint(x: 100.0, y: 200.0)  
point.x = 300.0  
point.y = 30.0 |
| CGSize(width: Double, height: Double) | var size = CGSize (width: 42.0, height: 11.0);  
size.width = 100.0  
size.height = 72.0 |
| CGRect(x: Double, y: Double, width: Double, height: Double) | var rect = CGRect (x:100.0, y: 200.0, width: 42.0, height: 11.0)  
rect.origin.x = 0.0  
rect.size.width = 50.0 |

**UIView Coordinate System**

- Origin in upper left corner
- y axis grows downwards
- Units are points, not pixels
  - Points are units of coordinate system
  - Pixels are min size unit of drawing
  - Typically 2 pixels per point
    - var ContentScaleFactor
**Location and Size**

- View’s location and size expressed in two ways
  - Frame is in superview’s coordinate system
  - Bounds is in local coordinate system

- View A frame:
  - Origin: 0,0
  - Size: 550 x 400

- View A bounds:
  - Origin: 0,0
  - Size 550 x 400

- View B frame:
  - Origin: 200, 100
  - Size 200 x 250

- View B bounds:
  - Origin: 0,0
  - Size: 200 x 250

**Frame and Bounds**

- Which to use?
  - Usually depends on the context

- If you are using a view, typically you use bounds

- If you are implementing a view, typically you use frame

- Matter of perspective
  - From outside it’s usually the frame
  - From inside it’s usually the bounds

- Examples:
  - Creating a view, positioning a view in superview - use frame
  - Handling events, drawing a view - use bounds
Creating Views

Where do views come from?

- Commonly placed in Storyboard
- Drag out any of the existing view objects (buttons, labels, etc)
- Or drag generic UIView and set custom class
**Manual Creation**

- Views are initialized using UIView.init(frame:)
  
  ```swift
  let theFrame = CGRect(x:0, y:0, width:200, height:150)
  let myView = UIView(frame: theFrame)
  ```

---

**Defining Custom Views**

- Subclass UIView

- For custom drawing, you override:
  
  ```swift
  func draw(_ rect: CGRect)
  ```

- For event handling, you override:
  
  ```swift
  func touchesBegan(_ touches: Set<UITouch> withEvent:UIEvent?)
  func touchesMoved(_ touches: Set<UITouch> withEvent:UIEvent?)
  func touchesEnded(_ touches: Set<UITouch> withEvent:UIEvent?)
  ```
Drawing Views

**draw: Method**

- **draw: does nothing by default**
  - If not overridden, then backgroundColor is used to fill

- **Override – draw: to draw a custom view**
  - rect argument is area to draw

- **When is it OK to call draw:?**
Be Lazy

• **draw:** is invoked automatically
  – Don’t call it directly!

• **Being lazy is good for performance**

• **When a view needs to be redrawn, use:**
  setNeedsDisplay

Demo
CoreGraphics and Quartz 2D

- UIKit offers very basic drawing functionality
  - UIRectFill(CRect rect)
  - UIRectFrame(CRect rect)

- CoreGraphics: Drawing APIs

- CG is a C-based API, not Objective-C

- CG and Quartz 2D drawing engine define simple but powerful graphics primitives
  - Graphics context
  - Transformations
  - Paths
  - Colors
  - Fonts
  - Painting operations

CG Wrappers

- Some CG functionality wrapped by UIKit

- UIColor
  - Convenience for common colors
  - Easily set the fill and/or stroke colors when drawing

  ```swift
  UIColor.red.set()
  // drawing will be done in red
  ```

- UIFont
  - Access system font
  - Get font by name
  - Get preferred font for a given text style
    - Best way for font in code
      ```swift
      class func preferredFont(forTextStyle style: UIFontTextStyle) -> UIFont
      ```

  - A few examples of Text Styles
    - UIFontTextStyle.headline
    - UIFontTextStyle.body
    - UIFontTextStyle.footnote
Simple draw(_: ) example

- **Draw a solid color and shape**

```swift
override func draw(_ rect: CGRect) {
    let bounds = self.bounds
    UIColor.gray.set()
    UIRectFill(bounds)

    let myShape = CGRect(x: 10, y: 10, width: 50, height: 100)
    UIColor.red.set()
    UIRectFill(myShape)

    UIColor.black.set()
    UIRectFrame(myShape)
}
```

What shape is this?

Drawing More Complex Shapes

- **Common steps for draw:**
  - Get current graphics context
  - Define a path
  - Set a color
  - Stroke or fill path
  - Repeat, if necessary
Paths

- CoreGraphics paths define shapes
- Made up of lines, arcs, curves and rectangles
- Creation and drawing of paths are two distinct operations
  - Define path first, then draw it

Drawing Shapes using Bezier Paths

- First create a Bezier Path
  let path = UIBezierPath()

- Move around, add lines or arcs to path
  path.move(to: CGPoint(x:60,y:40))
  path.addLine(to: CGPoint(x:100,y:50))
Simple Example

```swift
override func draw(_ rect: CGRect){
    let path = UIBezierPath()
    path.move(to: CGPoint(x: 75, y: 10))
    path.addLine(to: CGPoint(x: 10, y: 150))
    path.addLine(to: CGPoint(x: 160, y: 150))
    path.close()
    UIColor.red.setFill()
    UIColor.black.setStroke()
    path.lineWidth = 3.0
    path.stroke()
    path.fill()
}
```

What shape is this?

More Drawing Information

- UIView Class Reference
- CGContext Reference
- “Quartz 2D Programming Guide”
- Lots of samples in the iPhone Dev Center