Announcements

• Lab 2 is due on Wednesday by 11:59 PM
  – Late policy is 10% of lab total per day late
    • So -7.5 points per day late for lab 2

Today’s Topics

• Additional Swift Concepts

• Views

• Drawing

• Text & Images
Additional 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

```go
defer {
  statements
}
```
Error Handling

• 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

```
enum 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 cannotThrowErrors() -> 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.

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

- Equivalent to writing the following code
  ```
  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

  func addSubview(UIView)
  func removeFromSuperview()

• Manipulate the view hierarchy manually:

  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: \{ width, height \}

- **CGRect**
  - location and dimension: \{ origin, size \}

**Rects, Points and Sizes**

![Diagram showing the relationship between CGRect, CGPoint, and CGSize]
View-related Structure

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

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

What about View B?

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)
  ```

- Example:
  
  ```swift
  let frame = CGRect(x:20, y:45, width: 140, height: 20)
  let myLabel = UILabel(frame:frame)
  myLabel.text = "Hello Class"
  view.addSubview(myLabel)
  ```

### 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(CGRect rect)`
  - `UIRectFrame(CGRect 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

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)
}

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()
}
```

More Drawing Information

- UIView Class Reference
- CGContext Reference
- “Quartz 2D Programming Guide”
- Lots of samples in the iPhone Dev Center
Lab 3 Preview