Announcements

- **Lab 2 is due next Monday (Sept 25th) 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**

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**

```
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 stuct 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 you 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**

```
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#

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

```
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
}
```


Demo

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 insuffientFunds(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

```
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.

```
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

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


Demo

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()
```

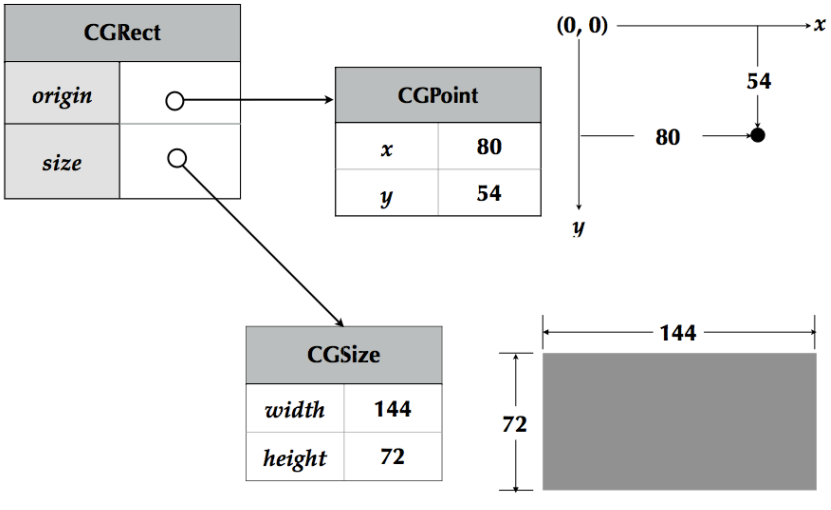
- **Manipulate the view hierarchy manually:**

```
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

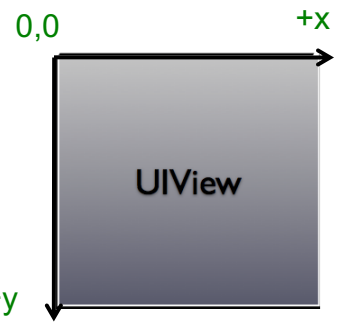


View-related Structure

Creation Function	Example
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

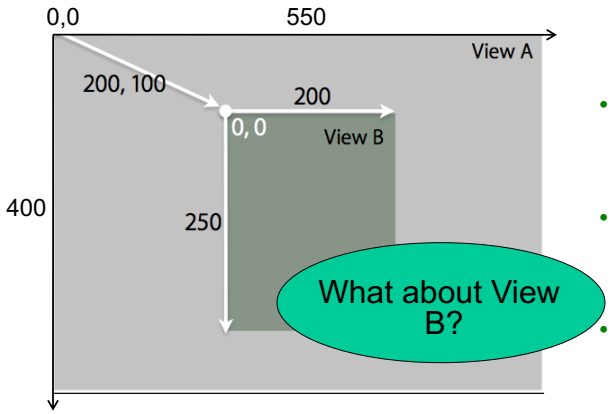
UIView Coordinate System

- **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**

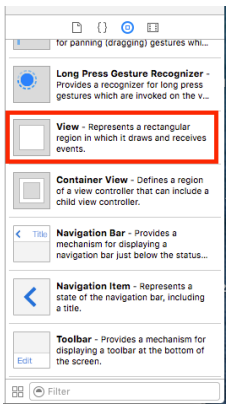
Frame and Bounds

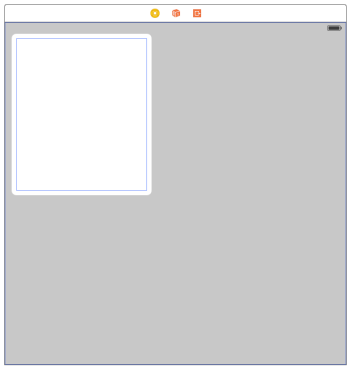
- **Which to use?**
  - Usually depends on the context
- **If you are using a view, typically you use bounds**
- **If you are implementing a view, typically you use frame**
- **Matter of perspective**
  - From outside it's usually the frame
  - From inside it's usually the bounds
- **Examples:**
  - Creating a view, positioning a view in superview - use frame
  - Handling events, drawing a view - use bounds

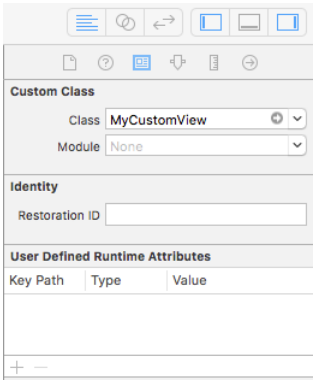
Creating Views

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:)**

```
let theFrame = CGRect(x:0, y:0, width:200, height:150)
let myView = UIView(frame: theFrame)
```

- **Example:**

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