Developing Applications for iOS

Lecture 4:
More Swift, Views

Prof. Dr. Radu Ionescu
raducu.ionescu@gmail.com
Faculty of Mathematics and Computer Science
University of Bucharest

Content

More Swift:

- Inheritance
- Initialization and Deinitialization
- Automatic Reference Counting
- Extensions
- Protocols

Views:

- View Hierarchy
- View Coordinates

Inheritance

- The example below defines a base class called `Vehicle`:

```
class Vehicle {
    var topSpeed = 0.0
    var description: String {
        return "can reach \(currentSpeed) km/h"
    }
    func makeNoise() {
        /* do nothing - an arbitrary vehicle
           doesn't necessarily make a noise */
    }
}
```

- The following example defines a subclass called `Bicycle`, that inherits from the superclass `Vehicle`:

```
class Bicycle: Vehicle {
    var hasBasket = false
}
```

Inheritance

- The new `Bicycle` class automatically gains all of the characteristics of `Vehicle`, such as its `currentSpeed` and `description` properties and its `makeNoise()` method.
- In addition to the characteristics it inherits, the `Bicycle` class defines a new stored property, `hasBasket`, with a default value of `false`:

```
let bicycle = Bicycle()
bicycle.hasBasket = true
bicycle.topSpeed = 25.0
print("Bicycle: \(bicycle.description)")
// Bicycle: can reach 25.0 km/h
```

Overriding

- To override a characteristic that would otherwise be inherited, you prefix your overriding definition with the `override` keyword.

  When necessary, you access the superclass version of a method, property, or subscript by using the `super` prefix:
- An overridden method named `someMethod()` can reach the superclass version by calling `super.someMethod()` within the overriding method implementation.
- An overridden property called `someProperty` can access the superclass version as `super.someProperty` within the overriding getter or setter implementation.
- An overridden subscript for `someIndex` can access the superclass version as `super[someIndex]` from within the overriding subscript implementation.

Method Overriding

- The following example defines a new subclass of `Vehicle` called `Train`, which overrides the `makeNoise()` method that `Train` inherits from `Vehicle`:

```
class Train: Vehicle
{
    override func makeNoise()
    {
        print("Choo Choo")
    }
}

let train = Train()
train.makeNoise()
// Prints "Choo Choo"
```

Property Overriding

- You can provide a custom getter (and setter, if appropriate) to override any inherited property.
- The stored or computed nature of an inherited property is not known by a subclass; it only knows that the inherited property has a certain name and type. You must always state both the name and the type of the property you are overriding:

```
class Car: Vehicle {
    var gear = 1
    override var description: String {
        return super.description +
                        " in gear \(gear)"
    }
}
let car = Car()
car.topSpeed = 212.0
car.gear = 5
print("Car: \(car.description)")
// Car: can reach 212.0 km/h in gear 5
```

Preventing Overrides

- You can prevent a method, property, or subscript from being overridden by marking it as `final`. The `final` modifier must be placed before the method, property, or subscript's introducer keyword, e.g. `final var`, `final func`, etc.
- You can mark an entire class as final by writing the `final` modifier before the `class` keyword in its class definition (`final class`). Any attempt to subclass a final class is reported as a compile-time error.

Initialization

- Initializers are special methods that can be called to create a new instance of a particular type.
- Initialization involves setting an initial value for each stored property on that instance and performing any other setup that is required before the new instance is ready for use.
- Classes and structures must set all of their stored properties to an appropriate initial value by the time an instance of that class or structure is created.
- In its simplest form, an initializer is like an instance method with no parameters, written using the `init` keyword:

```
init() {
    // perform some initialization here
}
```

Initialization

- You can set the initial value of a stored property from within an initializer, as shown below. Alternatively, specify a default property value as part of the property's declaration.
- You can provide initialization parameters as part of an initializer's definition:

```
struct Celsius {
    var temperature: Double
    init(_ celsius: Double) {
        temperature = celsius
    }
    init(fromKelvin kelvin: Double) {
        temperature = kelvin – 273.15
    }
}
let waterBoiling = Celsius(100.0)
// waterBoiling.temperature is 100.0
let waterFreezing = Celsius(fromKelvin: 273.15)
// waterFreezing.temperature is 0.0
```

Initialization

- Properties of an optional type are automatically initialized with a value of `nil`.
- You can assign a value to a constant property at any point during initialization, as long as it is set to a definite value by the time initialization finishes:

```
class Question {
    let text: String
    var response: String?
    init(text: String) {
        self.text = text
    }
    func ask() {
        print(text)
    }
}
let q = Question(text: "Do you like popcorn?")
q.ask() // Prints "Do you like popcorn?"
q.response = "Yes, but only with a good movie."
```

Initialization

- Swift provides a default initializer for any structure or class that provides default values for all of its properties and does not provide at least one initializer itself:

```
class ShoppingListItem {
    var name: String?
    var quantity = 1
}
var item = ShoppingListItem()
```

- Structure types automatically receive a memberwise initializer if they do not define any of their own custom initializers:

```
struct Size {
    var width = 0.0, height = 0.0
}
let twoByTwo = Size(width: 2.0, height: 2.0)
```

Initializer Delegation

- Initializers can call other initializers to perform part of an instance's initialization. This process, known as initializer delegation, avoids duplicating code across multiple initializers:

```
struct Size {
  var width = 0.0, height = 0.0
}
struct Point {
  var x = 0.0, y = 0.0
}
struct Rect {
  var origin = Point()
  var size = Size()
  init(origin: Point, size: Size) {
    self.origin = origin
    self.size = size
  }
  init(center: Point, size: Size) {
    let x = center.x - (size.width / 2)
    let y = center.y - (size.height / 2)
    self.init(origin: Point(x: x, y: y), size: size)
  }
}
```

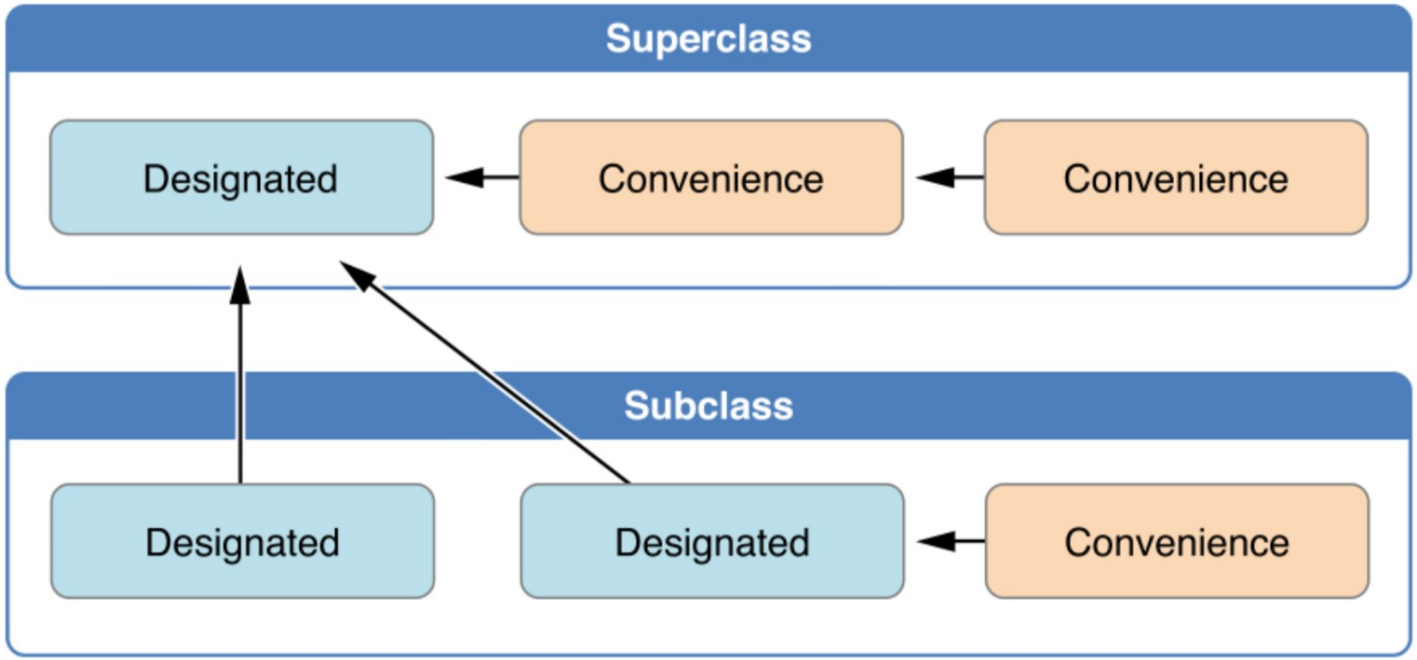
Class Inheritance and Initialization

- Designated initializers are the primary initializers for a class. A designated initializer fully initializes all properties introduced by that class and calls an appropriate superclass initializer to continue the initialization process up the superclass chain.
- Convenience initializers are secondary, supporting initializers for a class. You can define a convenience initializer to call a designated initializer from the same class as the convenience initializer with some of the designated initializer's parameters set to default values.

Class Inheritance and Initialization

Swift applies the following three rules for delegation calls between initializers:

- A designated initializer must call a designated initializer from its immediate superclass.
- A convenience initializer must call another initializer from the same class.
- A convenience initializer must ultimately call a designated initializer.



Class Inheritance and Initialization

- The following example shows designated initializers, convenience initializers, and automatic initializer inheritance in action:

```
class Food {
    var name: String
    init(name: String) {
        self.name = name
    }
    convenience init() {
        self.init(name: "[Unnamed]")
    }
}
class RecipeIngredient: Food {
    var quantity: Int
    init(name: String, quantity: Int) {
        self.quantity = quantity
        super.init(name: name)
    }
    override convenience init(name: String) {
        self.init(name: name, quantity: 1)
    }
}
```

Class Inheritance and Initialization

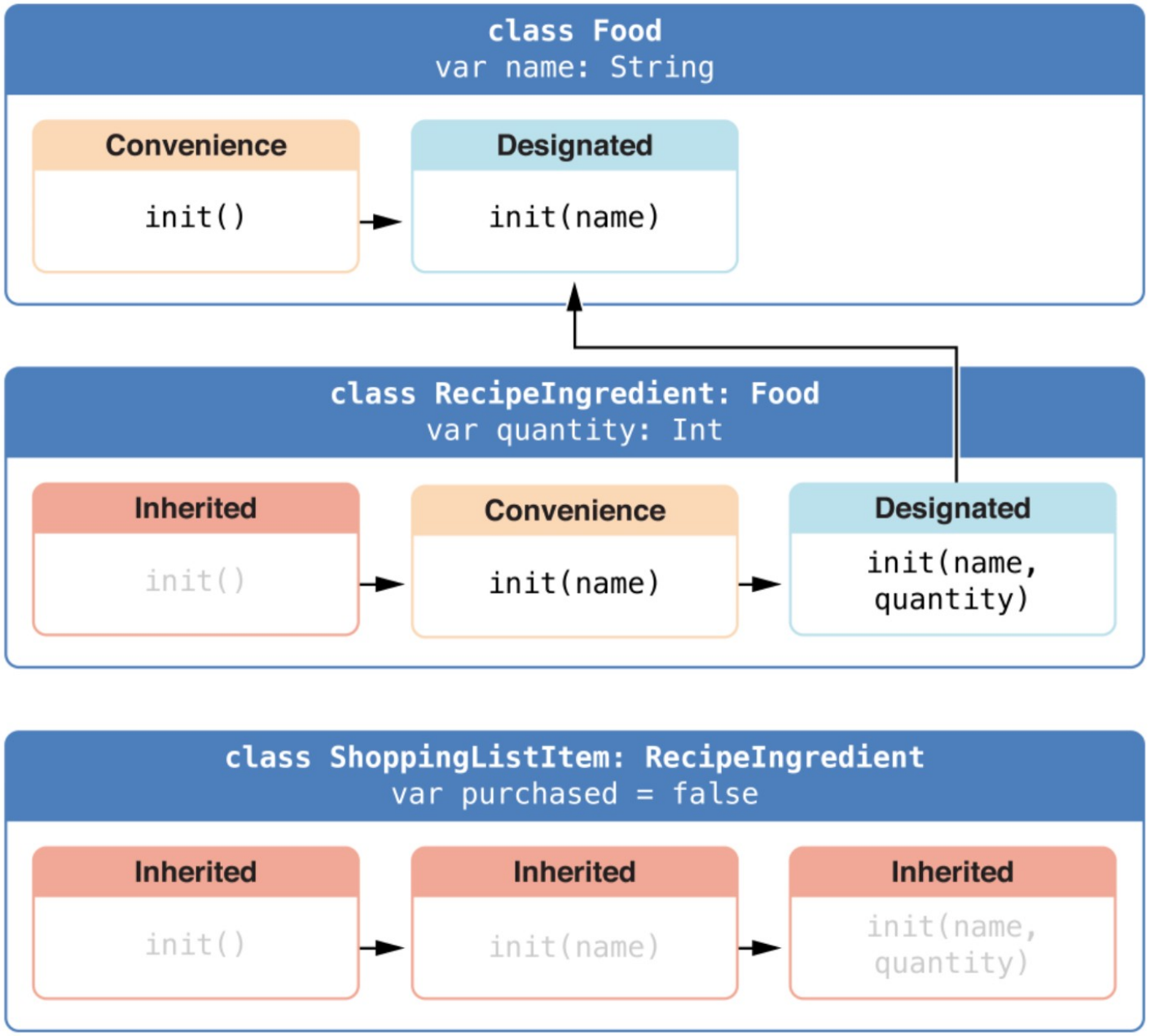
```
class ShoppingListItem: RecipeIngredient {
    var purchased = false
    var description: String {
        var output = "\(quantity) x \(name)"
        output += purchased ? " ✔" : " ✘"
        return output
    }
}

var breakfastList = [
    ShoppingListItem(),
    ShoppingListItem(name: "Bacon"),
    ShoppingListItem(name: "Eggs", quantity: 6)]

breakfastList[0].name = "Orange juice"
breakfastList[0].purchased = true

for item in breakfastList {
    print(item.description)
}
// 1 x Orange juice ✔
// 1 x Bacon ✘
// 6 x Eggs ✘
```

Class Inheritance and Initialization



Failable Initializers

- A failable initializer creates an optional value of the type it initializes. You write `return nil` within a failable initializer to indicate a point at which initialization failure can be triggered:

```
class CartItem {
    let name: String
    let quantity: Int
    init?(name: String, quantity: Int) {
        if quantity < 1 { return nil }
        self.name = name
        self.quantity = quantity
    }
}

if let zeroShirts = CartItem(name: "shirt",
                             quantity: 0) {
    print("Initialized \(zeroShirts.name) ")
} else {
    print("Unable to initialize zero shirts")
}
// Prints "Unable to initialize zero shirts"
```

Required Initializers

- Write the `required` modifier before the definition of a class initializer to indicate that every subclass of the class must implement that initializer:

```
class SomeClass {
    required init() {
        // initializer implementation goes here
    }
}
```

- You must also write the `required` modifier before every subclass implementation of a required initializer, to indicate that the requirement applies to further subclasses in the chain:

```
class SomeSubclass: SomeClass {
    required init() {
        // initializer implementation goes here}
    }
}
```

Set Default Property with a Closure

- If a stored property's default value requires some customization or setup, you can use a closure or global function to provide a customized default value for that property.
- If you use a closure to initialize a property, remember that the rest of the instance has not yet been initialized at the point that the closure is executed.
- You cannot access any other property values from within your closure, even if those properties have default values.
- You also cannot use the implicit `self` property, or call any of the instance's methods.
- The closure's end curly brace must be followed by an empty pair of parentheses. This tells Swift to execute the closure immediately.

Set Default Property with a Closure

```
struct Chessboard {
    let colors: [Bool] = {
        var tempBoard = [Bool]()
        var isBlack = false
        for i in 1...8 {
            for j in 1...8 {
                tempBoard.append(isBlack)
                isBlack = !isBlack
            }
            isBlack = !isBlack
        }
        return tempBoard
    }()
}
let board = Chessboard()
for (i, cell) in board.colors.enumerated() {
    if i % 8 == 0 { print("") }
    if cell {
        print("   ", terminator:"")
    } else {
        print("   ", terminator:"")
    }
}
```

Deinitialization

- A deinitializer is called immediately before a class instance is deallocated.
- You write deinitializers with the deinit keyword, similar to how initializers are written with the init keyword.
- Deinitializers are only available on class types.
- The deinitializer does not take any parameters and is written without parentheses:

```
deinit {
    // perform the deinitialization
}
```

Deinitialization

- A deinitializer is called immediately before a class instance is deallocated.
- You write deinitializers with the deinit keyword, similar to how initializers are written with the init keyword.
- Deinitializers are only available on class types.
- The deinitializer does not take any parameters and is written without parentheses:

```
deinit {
    // perform the deinitialization
}
```

Automatic Reference Counting

- Swift uses Automatic Reference Counting (ARC) to track and manage your app's memory usage.
- To make sure that instances don't disappear while they are still needed, ARC tracks how many properties, constants, and variables are currently referring to each class instance.
- ARC will not deallocate an instance as long as at least one active reference to that instance still exists.
- To make this possible, whenever you assign a class instance to a property, constant, or variable, that property, constant, or variable makes a strong reference to the instance.

strong vs weak

- `strong` means “keep this in the heap until I don’t point to it anymore”

  I won’t point to it anymore if I set my pointer to it to `nil`.

  Or if I myself am removed from the heap because no one has a `strong` pointer to me!

- `weak` or `unowned` means “keep this as long as someone else has a `strong` pointer to it”.

  If it gets thrown out of the heap, set my pointer to it to `nil` automatically.

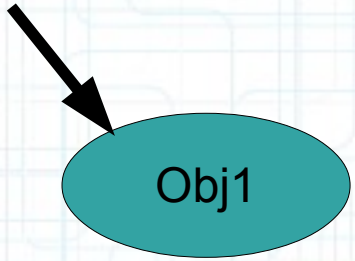
- This is not garbage collection!

  It’s way better.

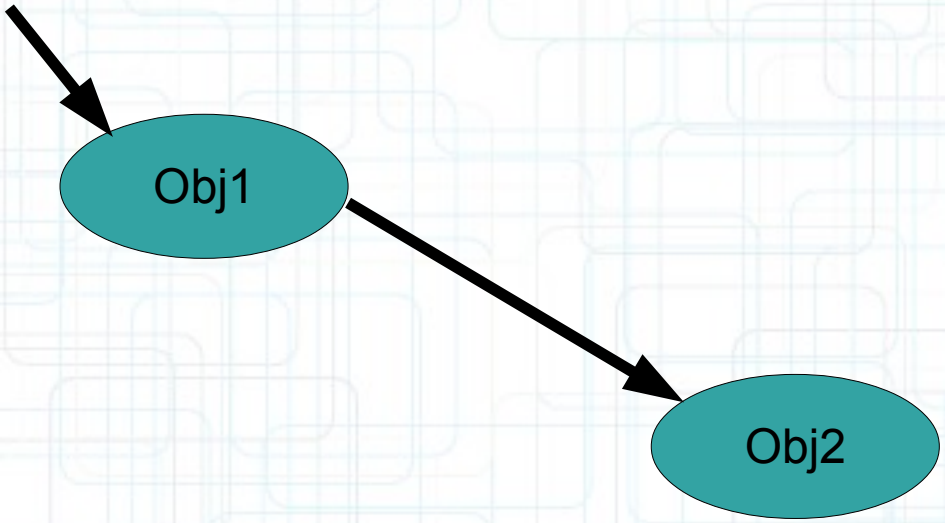
  It’s reference counting done automatically for you.

  It happens at compile time not at run time!

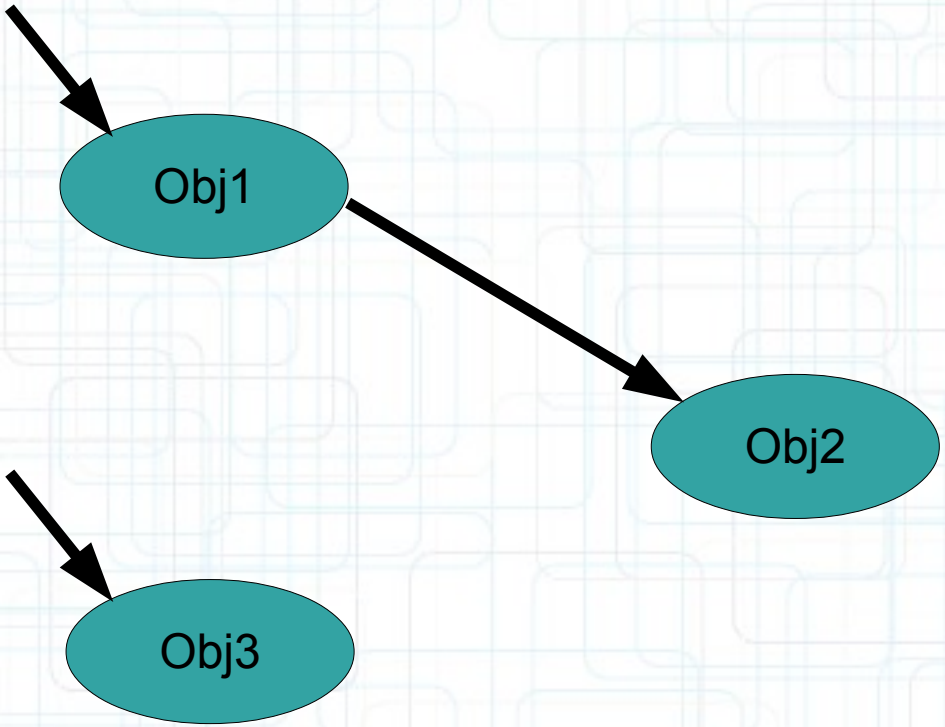
strong vs weak



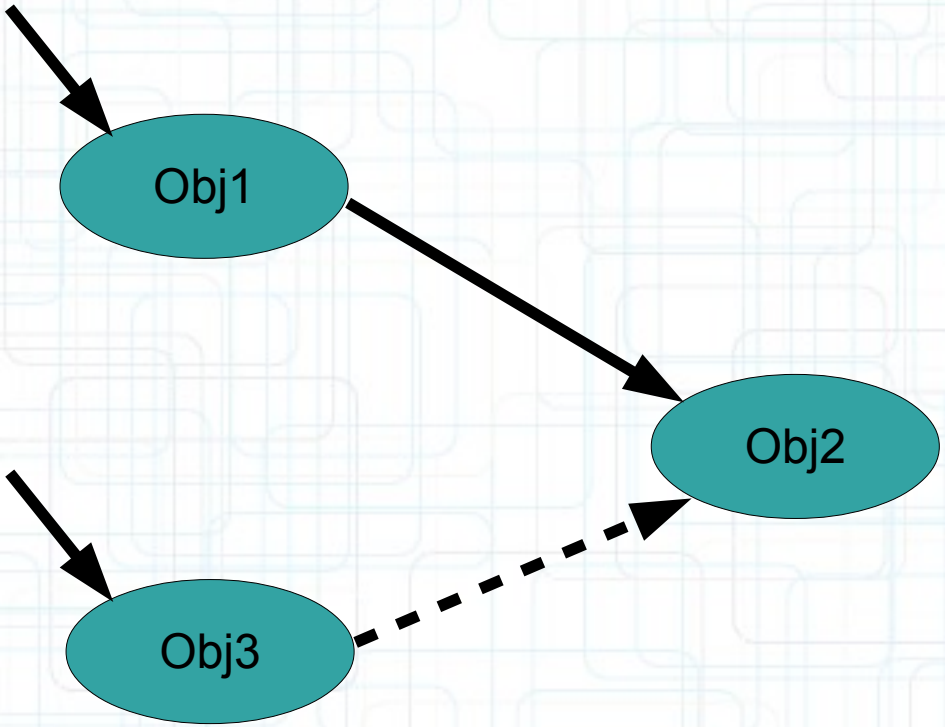
strong vs weak



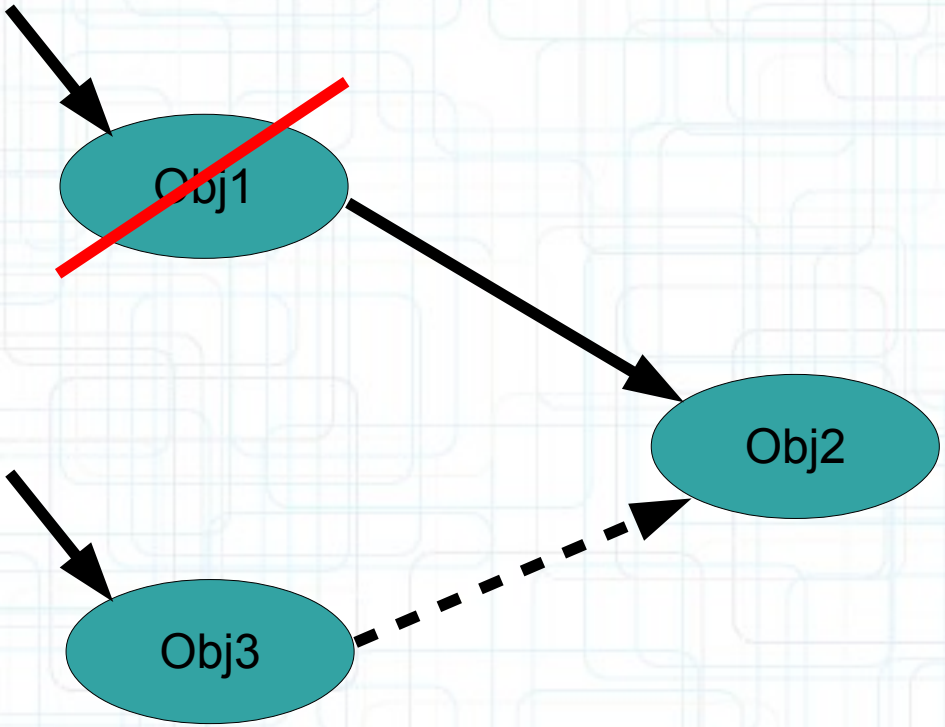
strong vs weak



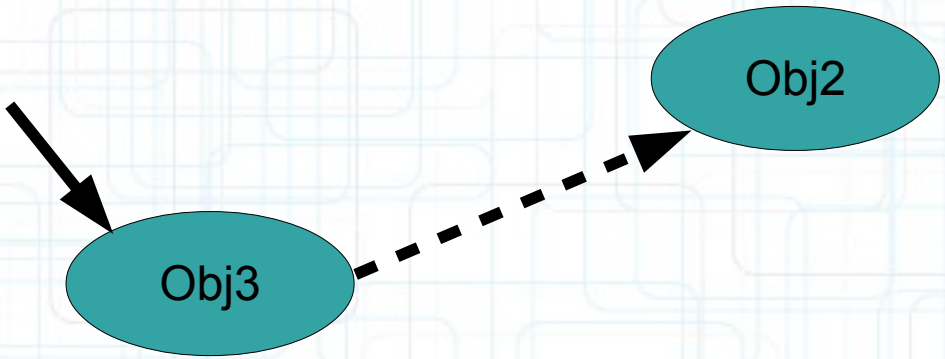
strong vs weak



strong vs weak

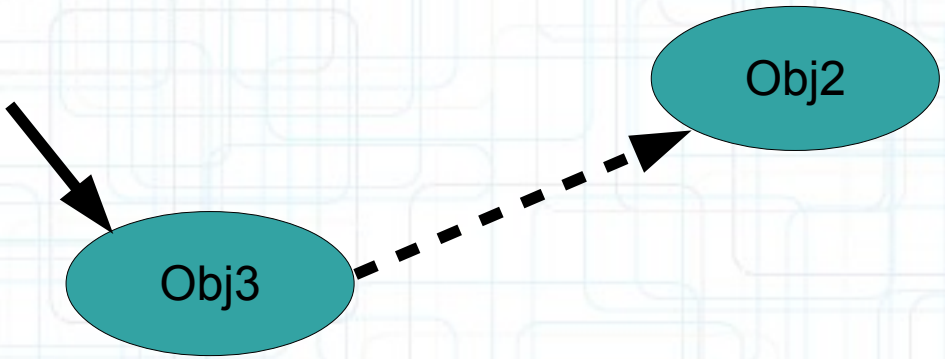


strong vs weak

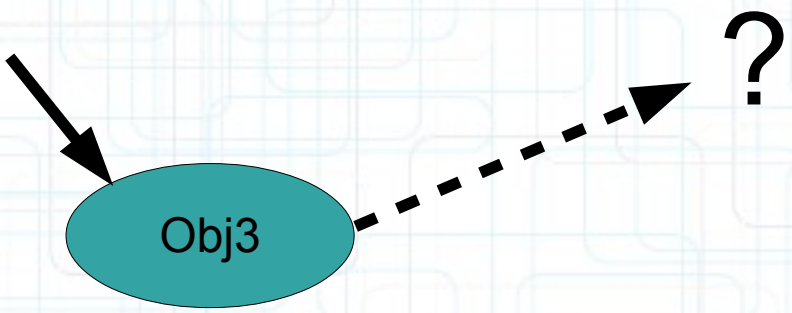


strong vs weak

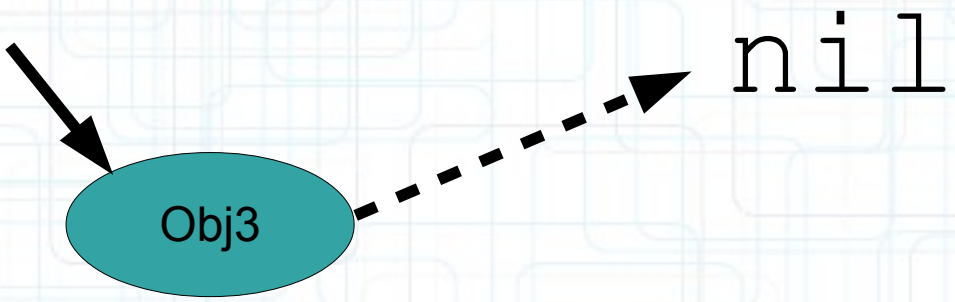
no `strong` reference to `Obj2`



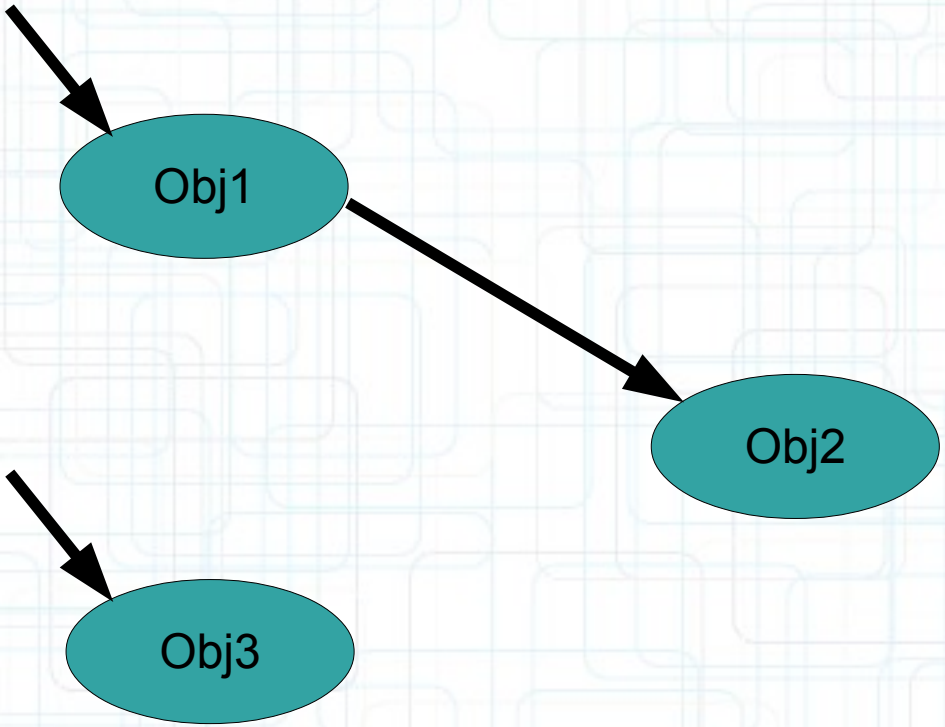
strong vs weak



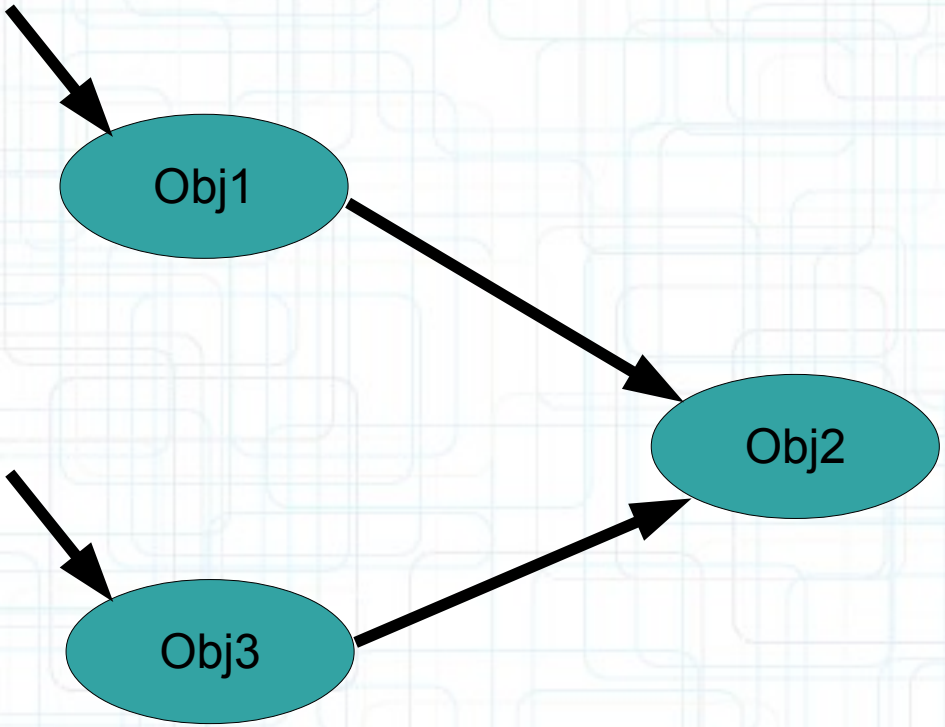
strong vs weak



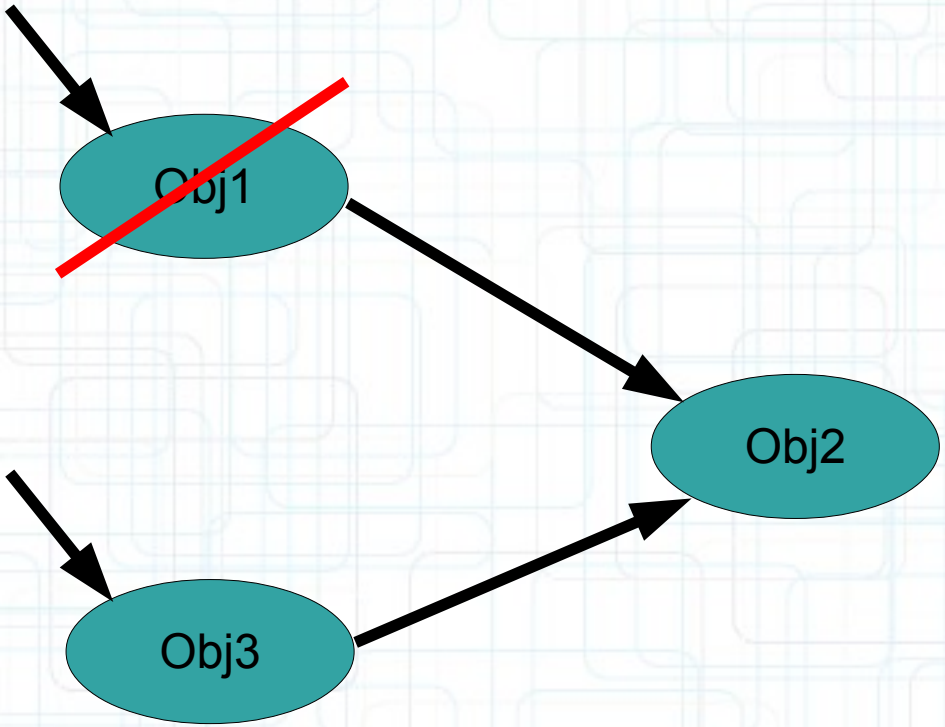
strong vs weak



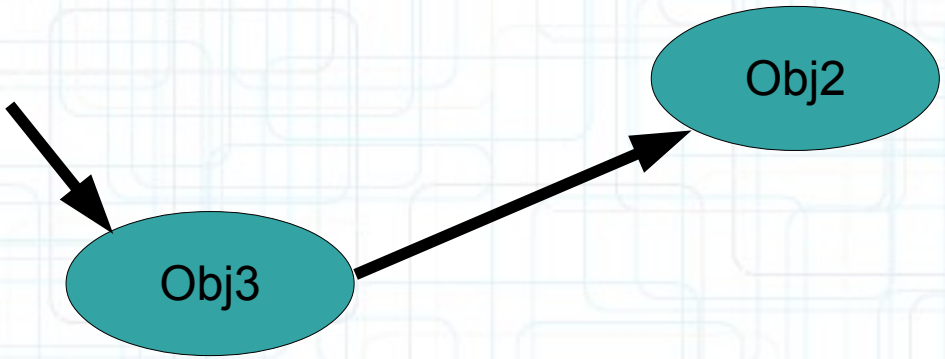
strong vs weak



strong vs weak

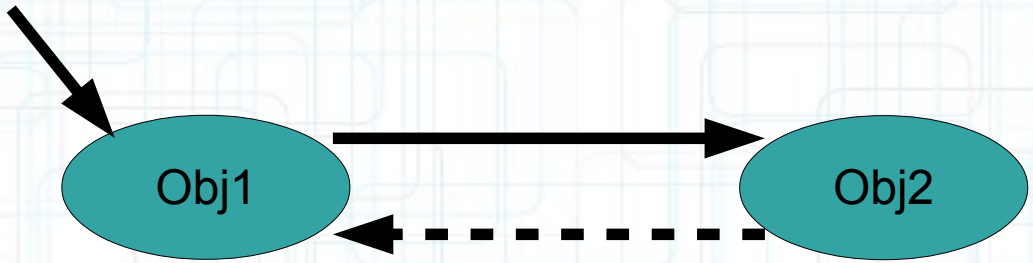


strong vs weak

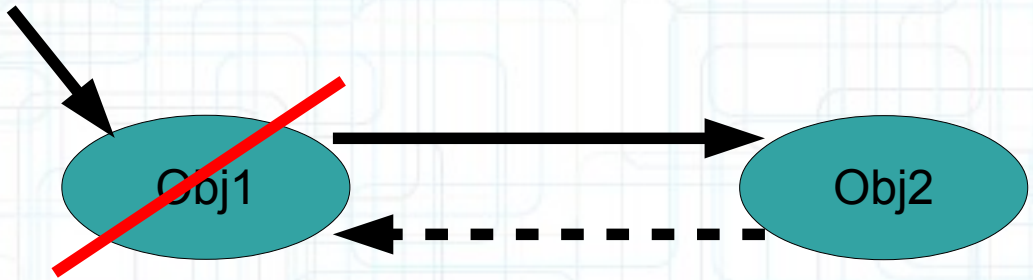


strong vs weak

strong vs weak

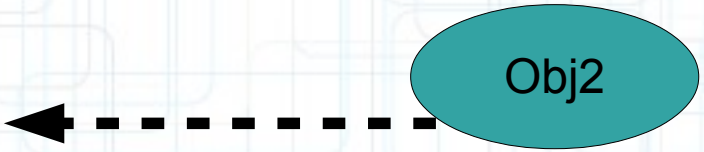


strong vs weak



strong vs weak

no `strong` reference to `Obj2`



strong vs weak

strong vs weak

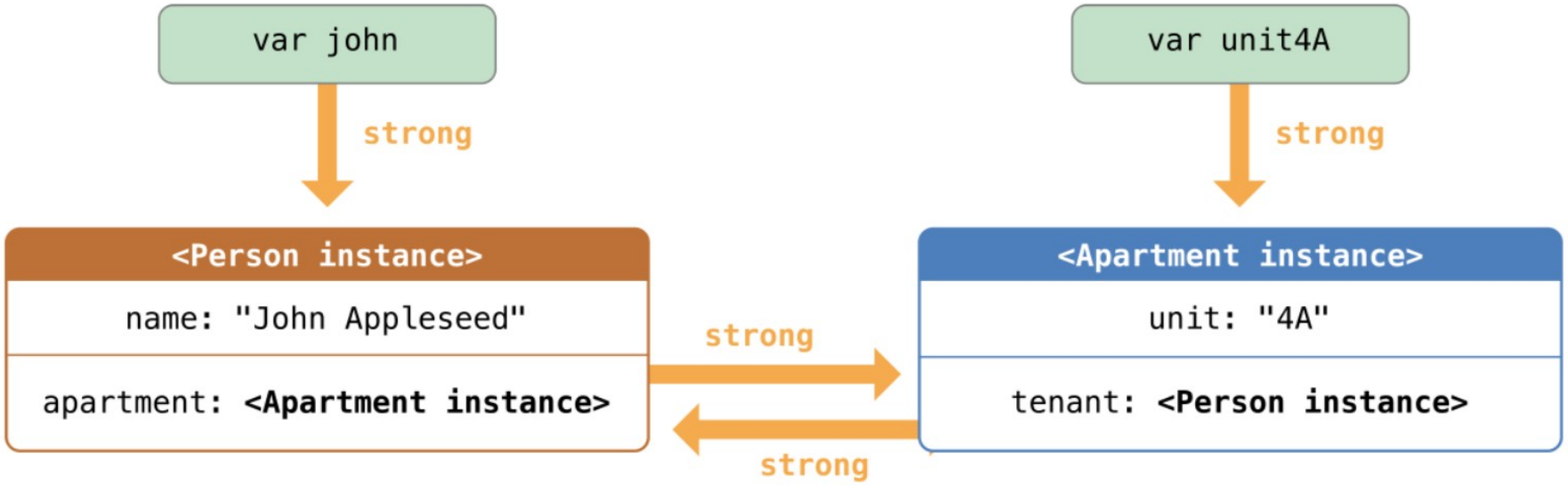
strong vs weak

strong vs weak

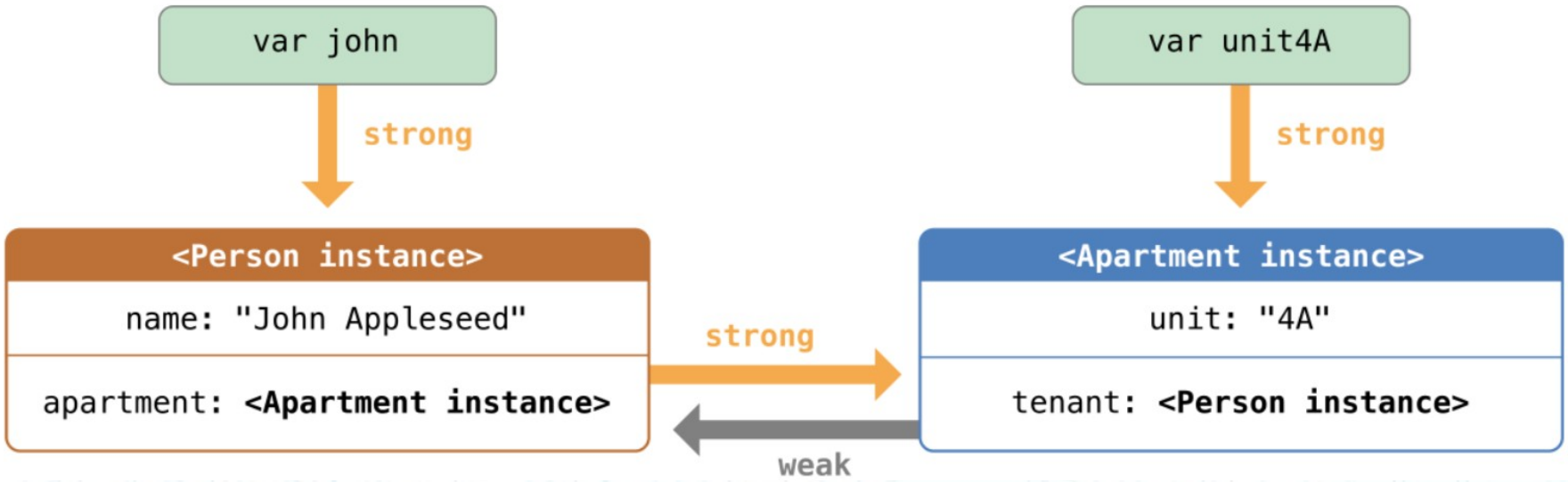
strong vs weak

Nothing gets deallocated. This problem is called a reference (or memory) cycle. It can be solved using `weak` or `unowned` references.

Strong Reference Cycles



Weak References



Weak References

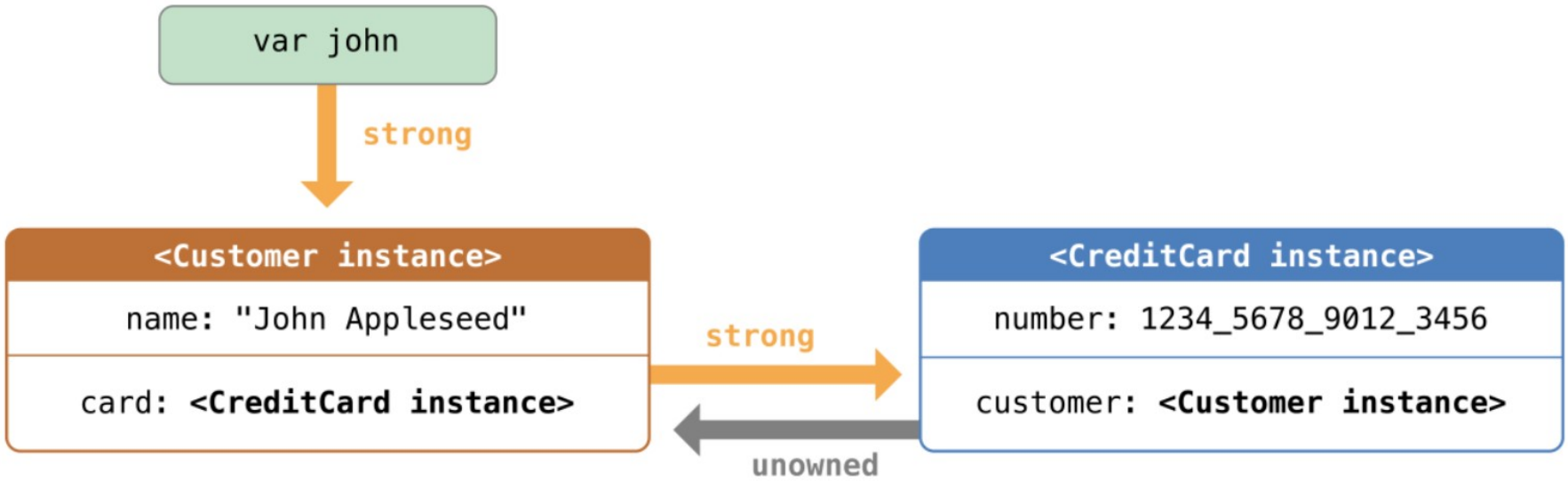
```
class Person {
    let name: String
    init(name: String) { self.name = name }
    var apartment: Apartment?
    deinit { print("\(name) is deinitialized") }
}
class Apartment {
    let unit: String
    init(unit: String) { self.unit = unit }
    weak var tenant: Person?
    deinit { print("Ap. \(unit) is deinitialized") }
}
var john: Person? = Person(name: "John Appleseed")
var unit4A: Apartment? = Apartment(unit: "4A")

john!.apartment = unit4A
unit4A!.tenant = john

john = nil
// Prints "John Appleseed is deinitialized"

unit4A = nil
// Prints "Ap. 4A is deinitialized"
```

Unowned References



Unowned References

```
class Customer {
    let name: String
    init(name: String) { self.name = name }
    var card: CreditCard?
    deinit { print("\(name) is deinitialized") }
}
class CreditCard {
    let number: UInt64
    unowned let customer: Person
    init(number: UInt64, customer: Customer) {
        self.number = number
        self.customer = customer
    }
    deinit { print("Credit card is deinitialized") }
}
var john: Customer?
john = Customer(name: "John Appleseed")
john!.card = CreditCard(number: 1234_5678_9012_3456,
                        customer: john!)

john = nil
// Prints "John Appleseed is deinitialized"
// Prints "Ap. 4A is deinitialized"
```

Strong Reference Cycles for Closures

- A strong reference cycle can also occur if you assign a closure to a property of a class instance, and the body of that closure captures the instance.
- The capture might occur because the closure's body accesses a property of the instance, such as `self.someProperty`, or because the closure calls a method on the instance, such as `self.someMethod()`.
- In either case, these accesses cause the closure to "capture" `self`, creating a strong reference cycle.
- You resolve a strong reference cycle between a closure and a class instance by defining a capture list as part of the closure's definition:

```
lazy var someClosure: () -> String = {
    [unowned self,
     weak delegate = self.delegate!] in
       // closure body goes here
}
```

Extensions

- Extensions add new functionality to an existing class, structure, enumeration, or protocol type, including types for which one does not have access to the original source code (known as retroactive modeling):

```
extension Double
{
    var km: Double { return self * 1_000.0 }
    var m: Double { return self }
    var cm: Double { return self / 100.0 }
    var mm: Double { return self / 1_000.0 }
}

let aMarathon = 42.km + 195.m
print("A marathon is \(aMarathon) meters long")
// Prints "A marathon is 42195.0 meters long"
```

Protocols

- A **protocol** defines a blueprint of methods, properties, and other requirements that suit a particular task or piece of functionality.
- The protocol can then be **adopted** by a class, structure, or enumeration to provide an actual implementation of those requirements.
- Any type that satisfies the requirements of a protocol is said to **conform** to that protocol:

```
protocol HasArea {
    var area: Double { get }
}

class Circle: HasArea {
    let pi = 3.1415927
    var R: Double
    var area: Double { return pi * R * R }
    init(radius: Double) { self.R = radius }
}
```

Protocols

- You can use the `is` and `as` operators to check for protocol conformance, and to cast to a specific protocol:

```
let objects: [Any] = [Circle(radius: 2.0),
                      "string",
                      127]
for object in objects {
    if let objectWithArea = object as? HasArea {
        print("Area is \(objectWithArea.area)")
    } else {
        print("Object doesn't have an area")
    }
}
// Area is 12.5663708
// Obj doesn't have an area
// Obj doesn't have an area
```

Protocols

The first use of protocols in iOS: delegates and data sources

- A `delegate` or `dataSource` is pretty much always defined as a `weak` property, by the way:

```
weak var delegate: UIScrollViewDelegate? { get set }
```

- This assumes that the object serving as delegate will outlive the object doing the delegating.
- Especially true in the case where the delegator is a View object (e.g. `UIScrollView`) and the delegate is that View's Controller.
- Controllers always create and clean up their View objects.
- Thus the Controller will always outlive its View objects.
- `dataSource` is just like a `delegate`, but, as the name implies, we are delegating provision of data.
- Views commonly have a `dataSource` because Views cannot own their data!

Views

- A view (i.e. `UIView` subclass) represents a rectangular area.
- It defines a coordinate space.
- Draws and handles events in that rectangle.

Hierarchical

- A view has only one superview:

```
var superview: UIView? { get }
```

- But can have many (or zero) subviews:

```
var subviews: [UIView] { get }
```

- Subview order (in `subviews` array) matters: those later in the array are on top of those earlier.

Views

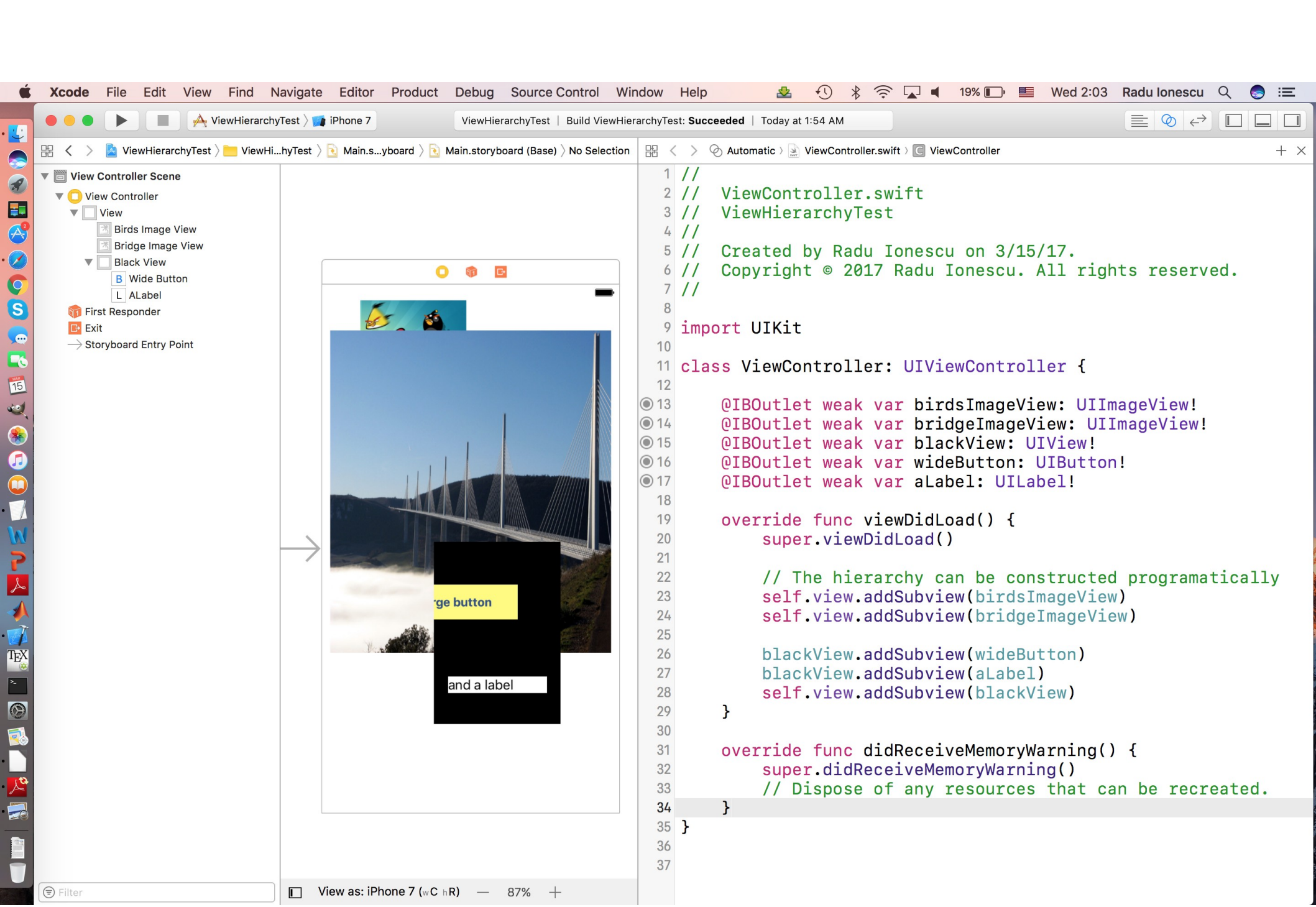
`UIWindow`

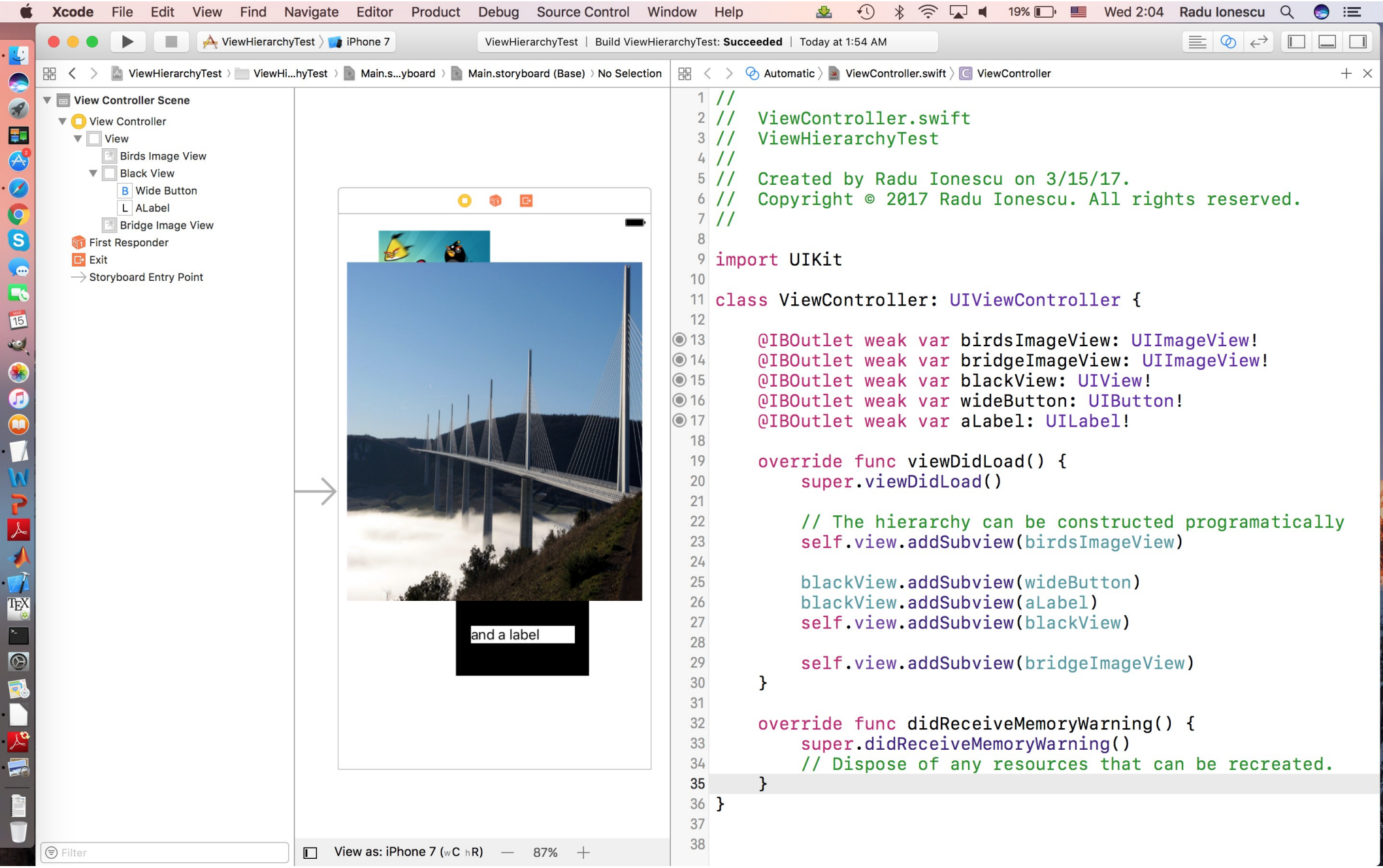
- The `UIView` at the top of the view hierarchy.
- Only have one `UIWindow` (generally) in an iOS application.
- It's all about views, not windows.

- The hierarchy is most often constructed in Interface Builder graphically.
- Even custom views are added to the view hierarchy using Interface Builder (more on this later).
- But it can be done in code as well:

```
func addSubview(_ view: UIView)

func removeFromSuperview()
```


Xcode File Edit View Find Navigate Editor Product Debug Source Control Window Help
Wed 2:03 Radu Ionescu
ViewHierarchyTest 〉 iPhone 7
ViewHierarchyTest | Build ViewHierarchyTest: Succeeded | Today at 1:54 AM
ViewHierarchyTest 〉 ViewHi...hyTest 〉 Main.s...yboard 〉 Main.storyboard (Base) 〉 No Selection
View Controller Scene
View Controller
View
Birds Image View
Bridge Image View
Black View
Wide Button
ALabel
First Responder
Exit
Storyboard Entry Point
rge button
and a label
View as: iPhone 7 (wC hR)
87%
Automatic 〉 ViewController.swift 〉 ViewController
//
// ViewController.swift
// ViewHierarchyTest
//
// Created by Radu Ionescu on 3/15/17.
// Copyright © 2017 Radu Ionescu. All rights reserved.
//
import UIKit
class ViewController: UIViewController {
@IBOutlet weak var birdsImageView: UIImageView!
@IBOutlet weak var bridgeImageView: UIImageView!
@IBOutlet weak var blackView: UIView!
@IBOutlet weak var wideButton: UIButton!
@IBOutlet weak var aLabel: UILabel!
override func viewDidLoad() {
super.viewDidLoad()
// The hierarchy can be constructed programatically
self.view.addSubview(birdsImageView)
self.view.addSubview(bridgeImageView)
blackView.addSubview(wideButton)
blackView.addSubview(aLabel)
self.view.addSubview(blackView)
}
override func didReceiveMemoryWarning() {
super.didReceiveMemoryWarning()
// Dispose of any resources that can be recreated.
}
}


Xcode File Edit View Find Navigate Editor Product Debug Source Control Window Help
Wed 2:04 Radu Ionescu
ViewHierarchyTest 〉 iPhone 7
ViewHierarchyTest | Build ViewHierarchyTest: Succeeded | Today at 1:54 AM
ViewHierarchyTest 〉 ViewHi...hyTest 〉 Main.s...yboard 〉 Main.storyboard (Base) 〉 No Selection
View Controller Scene
View Controller
View
Birds Image View
Black View
Wide Button
ALabel
Bridge Image View
First Responder
Exit
Storyboard Entry Point
and a label
View as: iPhone 7 (wC hR)
87%
Automatic 〉 ViewController.swift 〉 ViewController
//
// ViewController.swift
// ViewHierarchyTest
//
// Created by Radu Ionescu on 3/15/17.
// Copyright © 2017 Radu Ionescu. All rights reserved.
//
import UIKit
class ViewController: UIViewController {
@IBOutlet weak var birdsImageView: UIImageView!
@IBOutlet weak var bridgeImageView: UIImageView!
@IBOutlet weak var blackView: UIView!
@IBOutlet weak var wideButton: UIButton!
@IBOutlet weak var aLabel: UILabel!
override func viewDidLoad() {
super.viewDidLoad()
// The hierarchy can be constructed programatically
self.view.addSubview(birdsImageView)
blackView.addSubview(wideButton)
blackView.addSubview(aLabel)
self.view.addSubview(blackView)
self.view.addSubview(bridgeImageView)
}
override func didReceiveMemoryWarning() {
super.didReceiveMemoryWarning()
// Dispose of any resources that can be recreated.
}
}

View Coordinates

`CGFloat`

- Just a floating point number, but we always use it for graphics.

`CGPoint`

- A struct with two `CGFloat`s in it: `x` and `y`.

```
var p = CGPoint(x: 34.5, y: 22.0);
p.x += 20; // move right by 20 points
```

`CGSize`

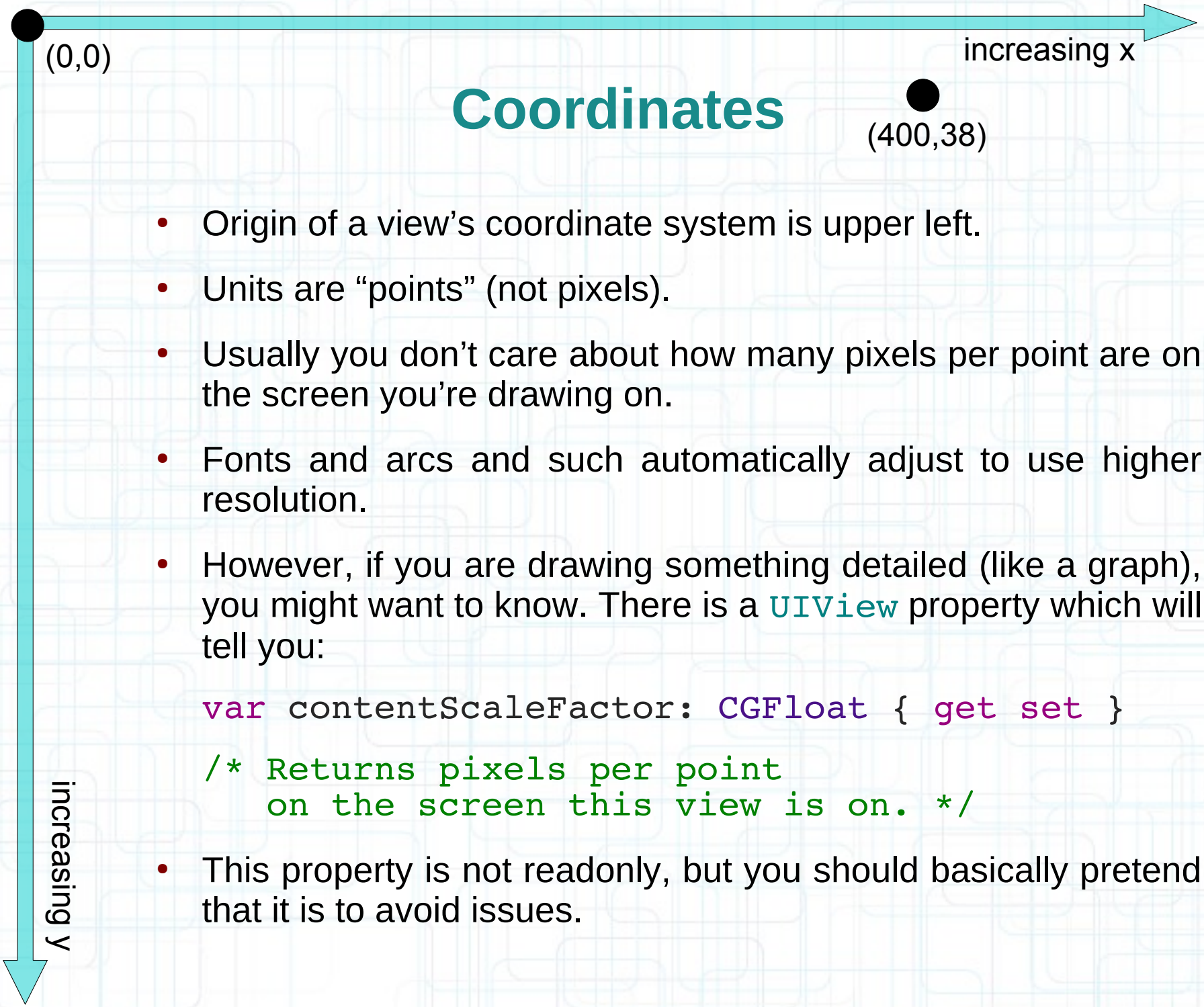
- A struct with two `CGFloat`s in it: `width` and `height`.

```
var s = CGSize(width: 100.0, height: 80.0);
s.height += 50; // make the size 50 points taller
```

`CGRect`

- A struct with a `CGPoint` origin and a `CGSize` size.

```
var r = CGRect(x: 10, y: 5, width: 300, height: 160)
r.size.height += 45; //make r 45 points taller
r.origin.x += 30; //move r to right 30 points
```



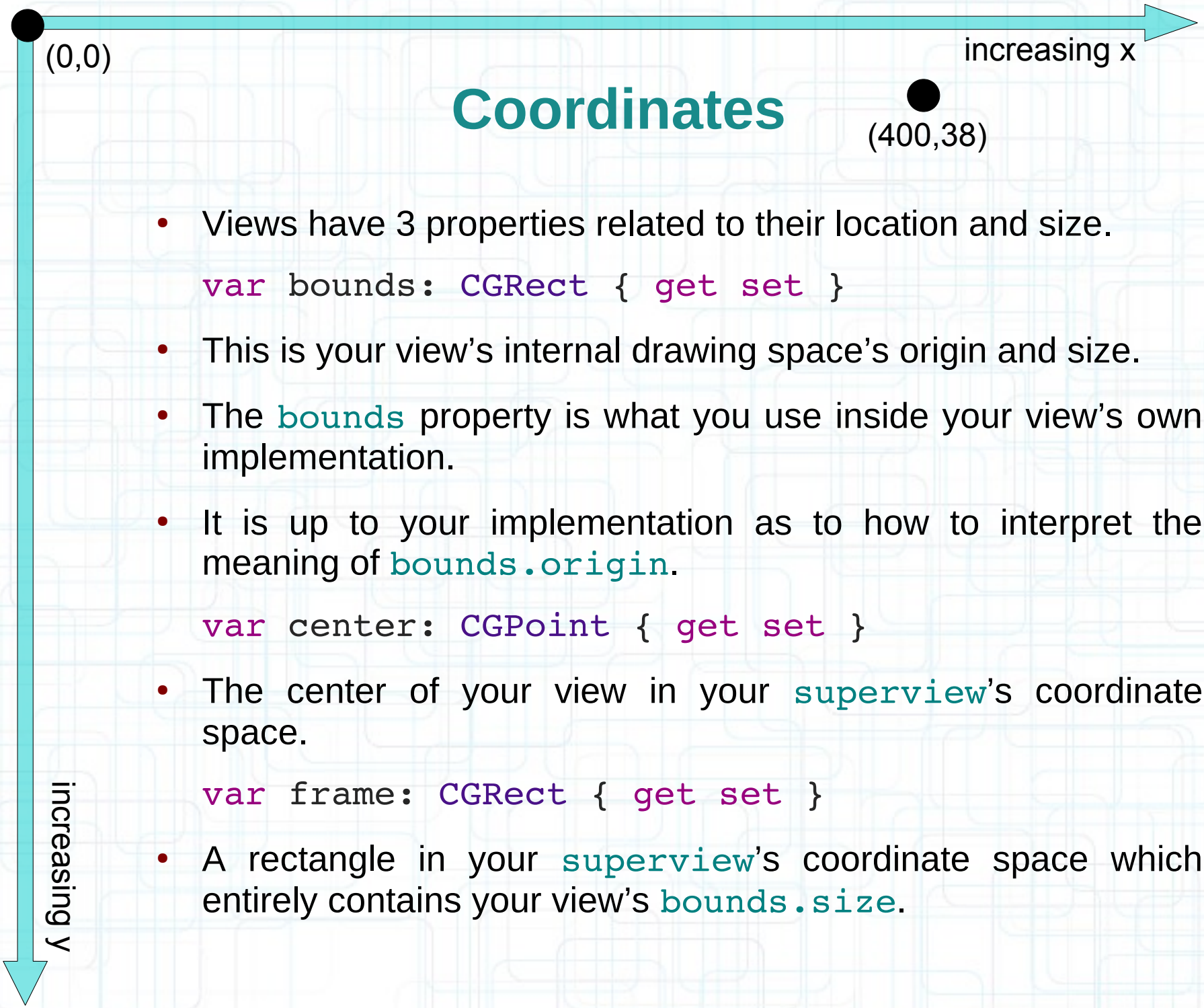
Coordinates

- Origin of a view's coordinate system is upper left.
- Units are "points" (not pixels).
- Usually you don't care about how many pixels per point are on the screen you're drawing on.
- Fonts and arcs and such automatically adjust to use higher resolution.
- However, if you are drawing something detailed (like a graph), you might want to know. There is a `UIView` property which will tell you:

```
var contentScaleFactor: CGFloat { get set }

/* Returns pixels per point
   on the screen this view is on. */
```

- This property is not readonly, but you should basically pretend that it is to avoid issues.



Coordinates

- Views have 3 properties related to their location and size.

```
var bounds: CGRect { get set }
```

- This is your view's internal drawing space's origin and size.
- The `bounds` property is what you use inside your view's own implementation.
- It is up to your implementation as to how to interpret the meaning of `bounds.origin`.

```
var center: CGPoint { get set }
```

- The center of your view in your `superview`'s coordinate space.

```
var frame: CGRect { get set }
```

- A rectangle in your `superview`'s coordinate space which entirely contains your view's `bounds.size`.

Coordinates

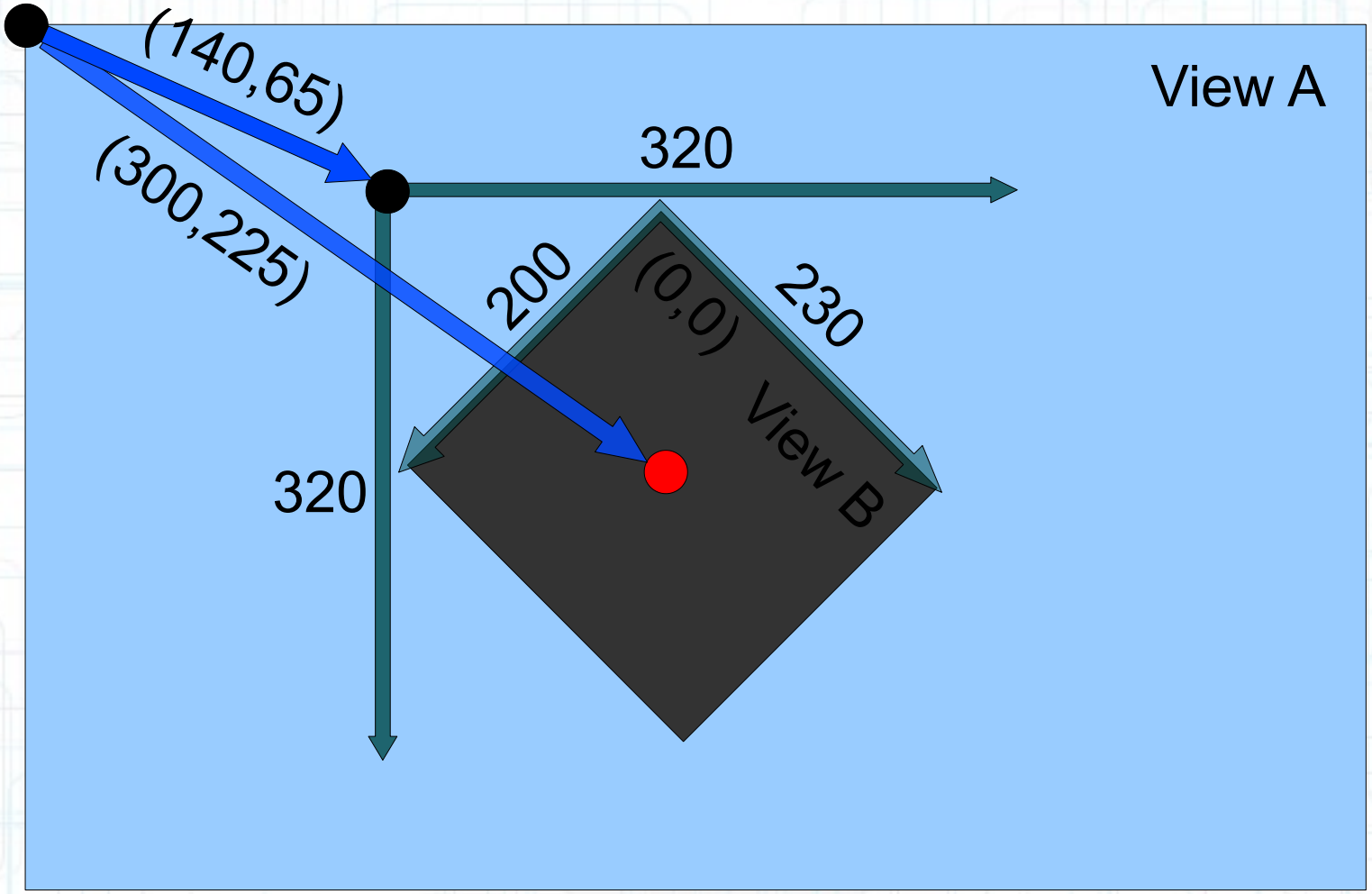
Use `frame` and `center` to position the view in the hierarchy

- These are used by `superview`, never inside your `UIView` subclass's implementation.
- You might think `frame.size` is always equal to `bounds.size`, but you would be wrong because views can be rotated (and scaled and translated too).
- Views are rarely rotated, but don't misuse `frame` or `center` by assuming that.

Coordinates

Use `frame` and `center` to position the view in the hierarchy

- Let's take a look at the following example:



Coordinates

Use `frame` and `center` to position the view in the hierarchy

- Let's take a look at the following example:

  View B's `bounds` = ((0,0),(230,200))

  View B's `frame` = ((140,65),(320,320))

  View B's `center` = (300,225)

  View B's middle in its own coordinate space is:

```
(bounds.size.width / 2 + bounds.origin.x,
bounds.size.height / 2 + bounds.origin.y)
```

  which equals (115,100) in this case.

Creating Views

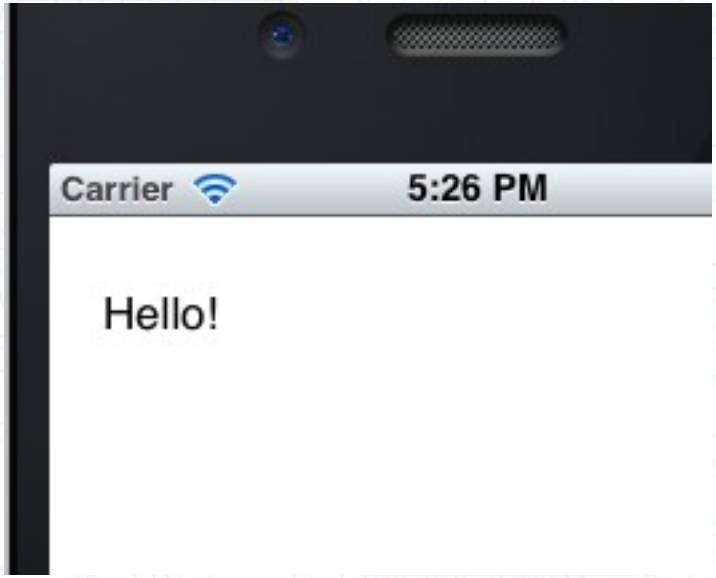
- Most often you create views in Interface Builder.
- Of course, Interface Builder's palette knows nothing about a custom view class you might create.
- In that case, you drag out a generic `UIView` from the palette and use the Inspector to change the class of the `UIView` to your custom class.
- How do you create a `UIView` in code (i.e. not in Interface Builder)?

  Just call `init(frame:)`, the `UIView`'s designated initializer.

Creating Views

- Example:

```
let rect = CGRect(x: 20, y: 20,
                  width: 50, height: 30)
let label = UILabel(frame:rect)
label.text = "Hello"
self.view.addSubview(label)
// (self.view is a Controller's top-level view)
```



Next Time

Views, Drawing and Gestures:

- Drawing Paths
- Drawing Text
- Drawing Images
- Error Handling
- Gesture Recognizers