Chapter 10.

Swift Programming for iOS Apps Development

2023-2024
COMP7506 Smart Phone Apps Development
Dr. T.W. Chim (E-mail: twchim@cs.hku.hk)
Department of Computer Science, The University of Hong Kong

Agenda

- Constant & Variable
- Flow Control
- Class
- Function
- String & Array
- Optionals
- Image
- Touching Events

Constant & Variable

Variable

Similar to the programming language Pascal

```
var width: Int = 5
```

Variable type — Variable name (identifier) — Initial value

```
var width = Int(5)
```

Variable type — Variable name (identifier) — Initial value

```
var width = 5
```

Variable name (identifier) — Initial value

Primitive Data Types

- Primitive data types in Swift is similar to Objective-C, but with slight syntax difference
- Below is some of the supported basic primitive data types

Swift Type	Description	Example
Int, Int8, Int16, Int32, Int64, UInt8, UInt16, UInt32, UInt64	Integer, Unsigned Integer	var A = 10 var A: Int = 10 var A = Int(10)
Float	Single precision floating point number	var A: Float = -0.333 var A=Float(A)
Double	Double precision floating point number	var A = 0.333 var A: Double = 0.333 var A=Double(A)
Character	Character	var S: Character = “c” var S = Character(“c”)
String	Collection of characters	var S = “hi” var S: String = “hi” var S = String(“hi”)
BOOL	Boolean variable	var B = true var B: Bool = true var B = Bool(true)

Constants vs. Variables in Swift

- Constants are a way of telling Swift that a particular value should not and will not change (i.e. ***immutable***)
  - Swift's compiler can optimize memory used for constants to make code perform better
  - Prevent programmer from changing the constant value accidentally
- How to define constants?
  - Use the same syntax for variable definition, but change the 'var' to 'let'
    - let A: Int = 10 + 12
    - let A = 0.333
    - let S = "Hello " + "World"

let width = 5

Image-Specific Variable Types

- We need image-specific variable type (class) for displaying the graphics on the screen when creating a GUI application
- Below are some examples:

Image-Specific Type (Class)	Description
UIImageView	Provides a view-based container for displaying a single image
UILabel	Implements a read-only text box on the view (e.g. The “Hello World” label we used last time)
UIButton	Implements a button on the touch screen that intercepts touch events (e.g. The “Click Me” button we used last time)

Flow Control

Flow Control – Conditional Statement

```
var a = 0
var b = 1
if (a < b) {
    // Do something here
} else {
    // Do another thing here
}
```

Flow Control – Loop Control

For-In loop:

```
for i in 0…b - 1 {
    // Do Something here
} // i = 0, 1, 2, 3, …, b - 1
```

While-loop:

```
var count = 0
while count < 10 {
    print("count is \(count)")
    count ++
}
```

Repeat-While-loop:

```
var count = 0
repeat {
    print("count is \(count)")
    count ++
} while count < 10
```

Class

OOP Example

- We use an example to show how to declare and define a class in Swift.
- Example: Fireball in a smart phone game
  - Fireballs on the screen are modeled as objects.
  - Each fireball has its own center position x, y, and radius
  - We can invoke methods to operate on a fireball. e.g., Tell it to move for a certain amount of pixels
  - When a fireball receives the method call, it will move itself according to the given parameters.

Class Definition

```
class Fireball {
    var centerX: Float
    var centerY: Float
    var radius: Float

    init(centerX: Float, centerY: Float, radius: Float) {   // constructor
        self.centerX = centerX
        self.centerY = centerY
        self.radius = radius
    }

    func move(_ moveX: Float, _ moveY: Float) {
        self.centerX += moveX
        self.centerY += moveY
    }

    deinit {   // for object deallocation
    }
}
```

“self” is analogous to “this” in Java.

Use of Class Instance

```
var fireball1 = Fireball?
var fireball2 = Fireball?
fireball1 = Fireball(centerX: 7.0, centerY: 5.0, radius: 6.0)
fireball2 = Fireball(centerX: 7.0, centerY: 8.0, radius: 4.0)

fireball1!.move(1.0, 1.0)
fireball2!.move(1.0, 1.0)

fireball1 = nil
fireball2 = nil
```

? and ! are about optional stuff and will be explained later.

Concept of Pointers

- Objects are usually created dynamically during runtime.
- A pointer is used to store the memory address that holds up the object ("points to" the object).
- Declaring a pointer variable and allocating an object with initial values, and pointing to that object:
  - var fireball1 = Fireball?
  - fireball1 = Fireball(centerX: 10.0, centerY: 10.0, radius: 5.0)

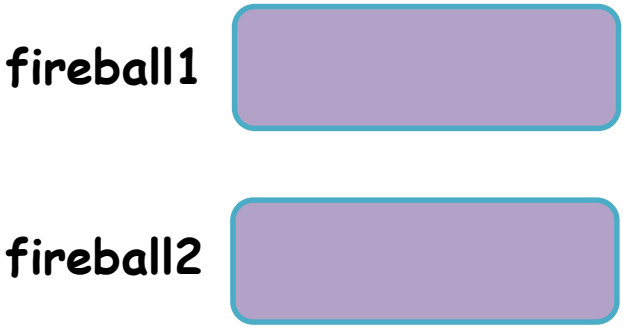
Concept of Pointers

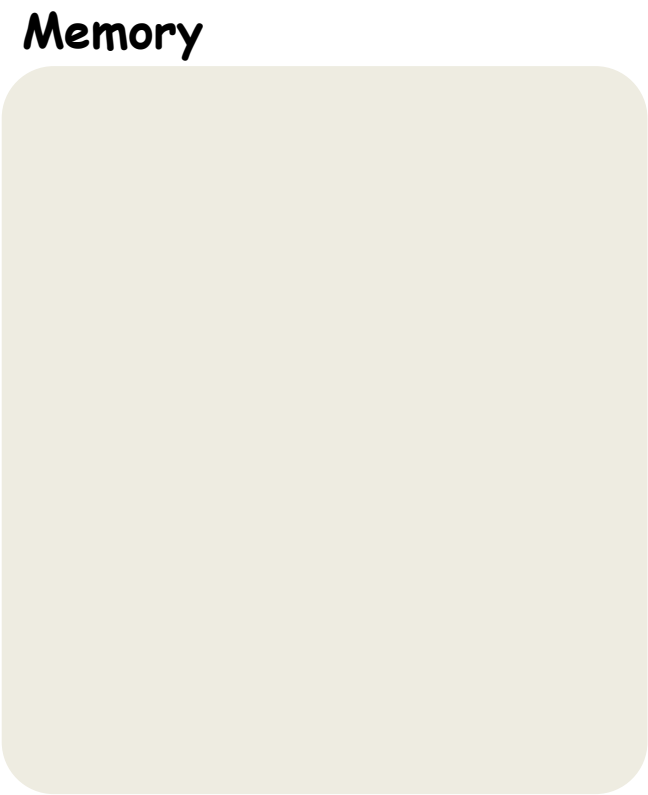
```
var fireball1 = Fireball?
var fireball2 = Fireball?
fireball1 = Fireball(centerX: 7.0, centerY: 5.0, radius: 6.0)
fireball2 = Fireball(centerX: 7.0, centerY: 8.0, radius: 4.0)

fireball1!.move(1.0, 1.0)
fireball2!.move(1.0, 1.0)

fireball1 = nil
fireball2 = nil
```

Memory

Concept of Pointers

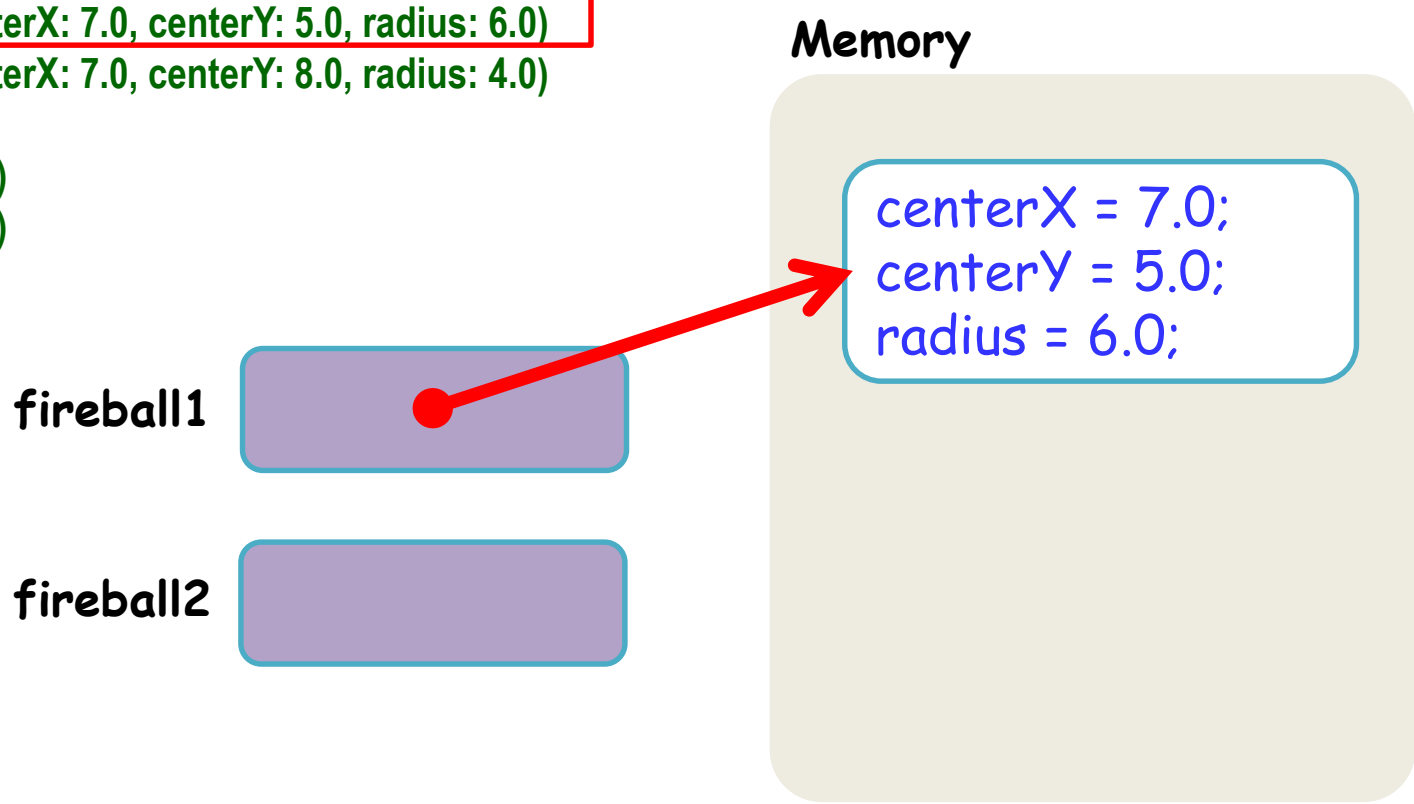
```
var fireball1 = Fireball?
var fireball2 = Fireball?
fireball1 = Fireball(centerX: 7.0, centerY: 5.0, radius: 6.0)
fireball2 = Fireball(centerX: 7.0, centerY: 8.0, radius: 4.0)

fireball1!.move(1.0, 1.0)
fireball2!.move(1.0, 1.0)

fireball1 = nil
fireball2 = nil
```





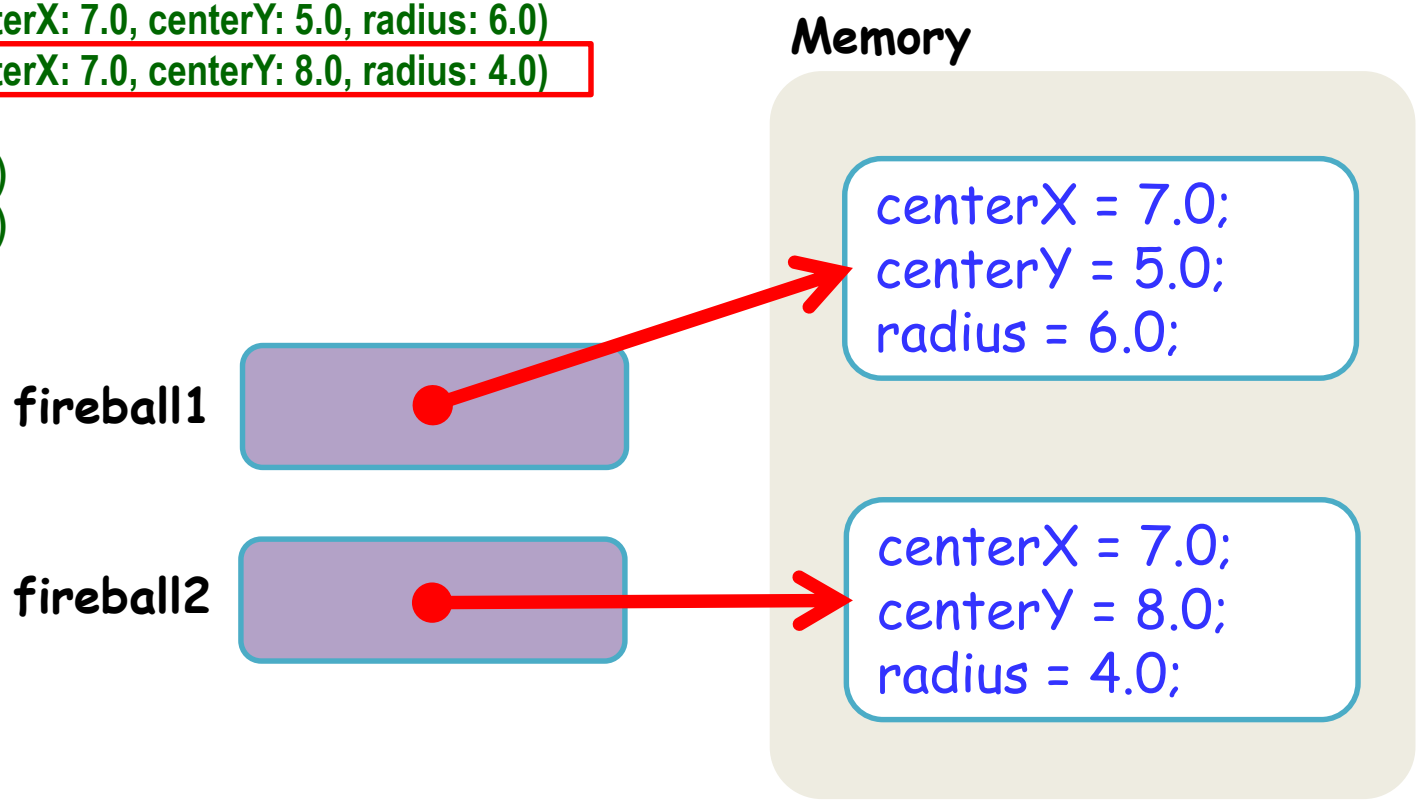
Concept of Pointers

```
var fireball1 = Fireball?
var fireball2 = Fireball?
fireball1 = Fireball(centerX: 7.0, centerY: 5.0, radius: 6.0)
fireball2 = Fireball(centerX: 7.0, centerY: 8.0, radius: 4.0)

fireball1!.move(1.0, 1.0)
fireball2!.move(1.0, 1.0)

fireball1 = nil
fireball2 = nil
```



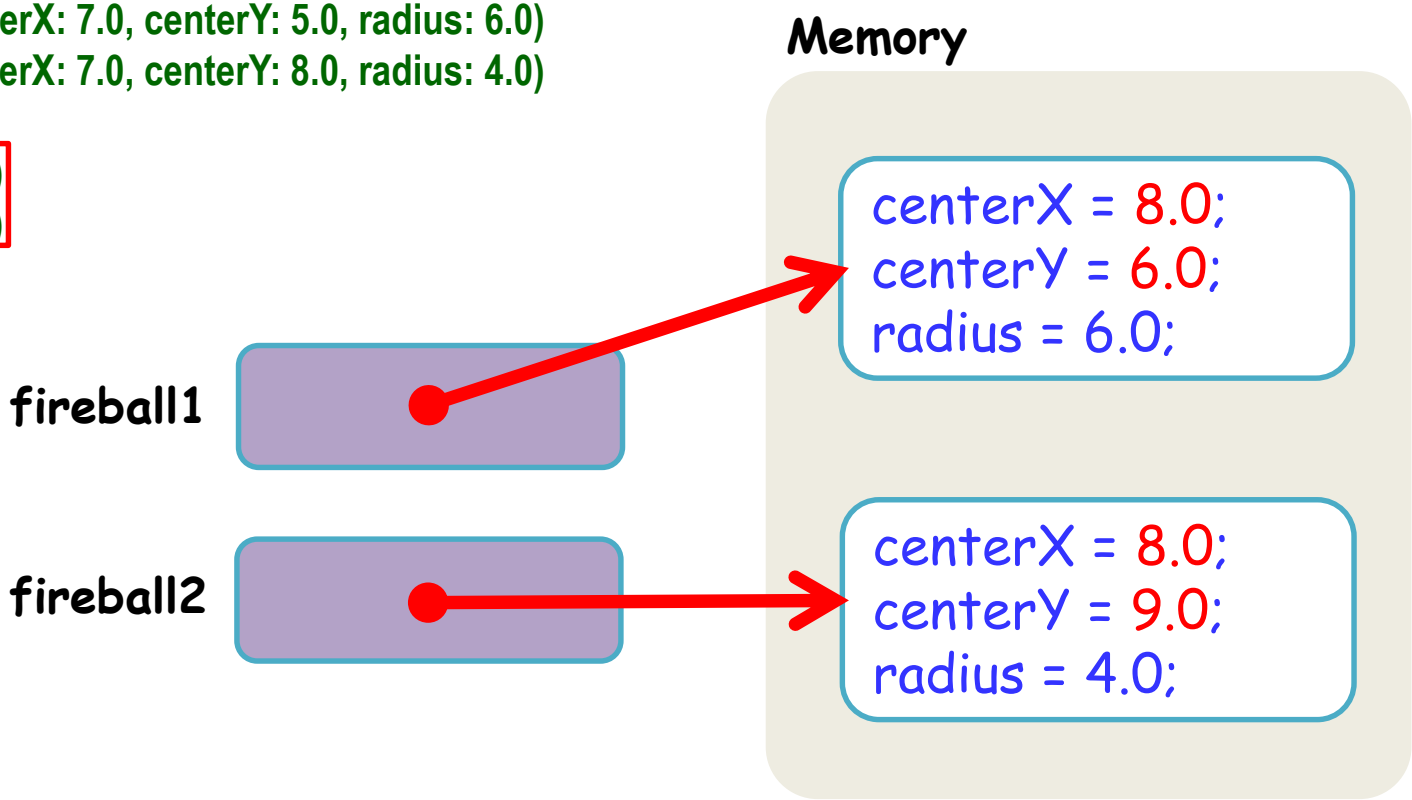
Concept of Pointers

```
var fireball1 = Fireball?
var fireball2 = Fireball?
fireball1 = Fireball(centerX: 7.0, centerY: 5.0, radius: 6.0)
fireball2 = Fireball(centerX: 7.0, centerY: 8.0, radius: 4.0)

fireball1!.move(1.0, 1.0)
fireball2!.move(1.0, 1.0)

fireball1 = nil
fireball2 = nil
```



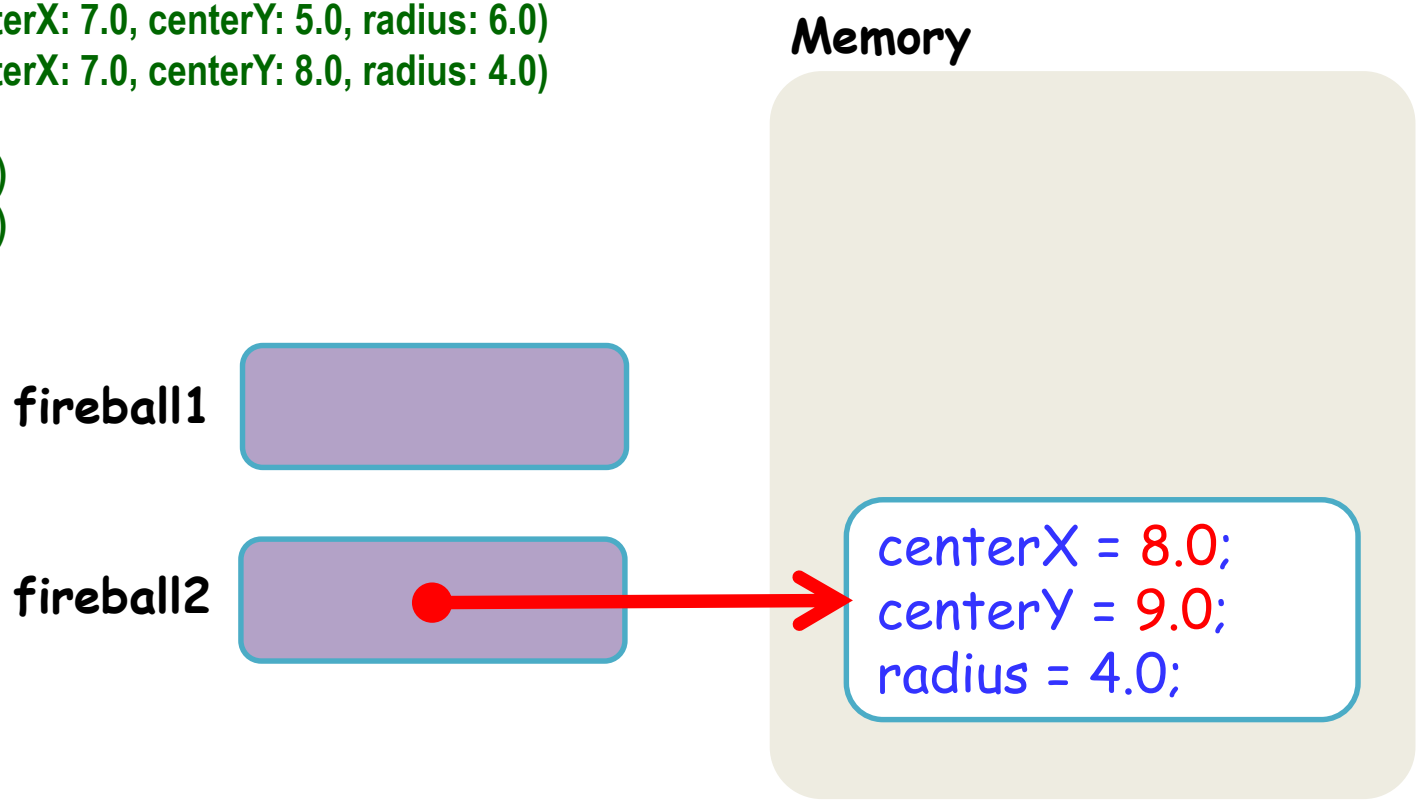
Concept of Pointers

```
var fireball1 = Fireball?
var fireball2 = Fireball?
fireball1 = Fireball(centerX: 7.0, centerY: 5.0, radius: 6.0)
fireball2 = Fireball(centerX: 7.0, centerY: 8.0, radius: 4.0)

fireball1!.move(1.0, 1.0)
fireball2!.move(1.0, 1.0)

fireball1 = nil
fireball2 = nil
```



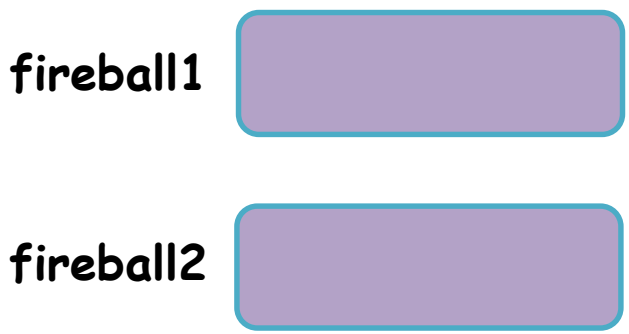
Concept of Pointers

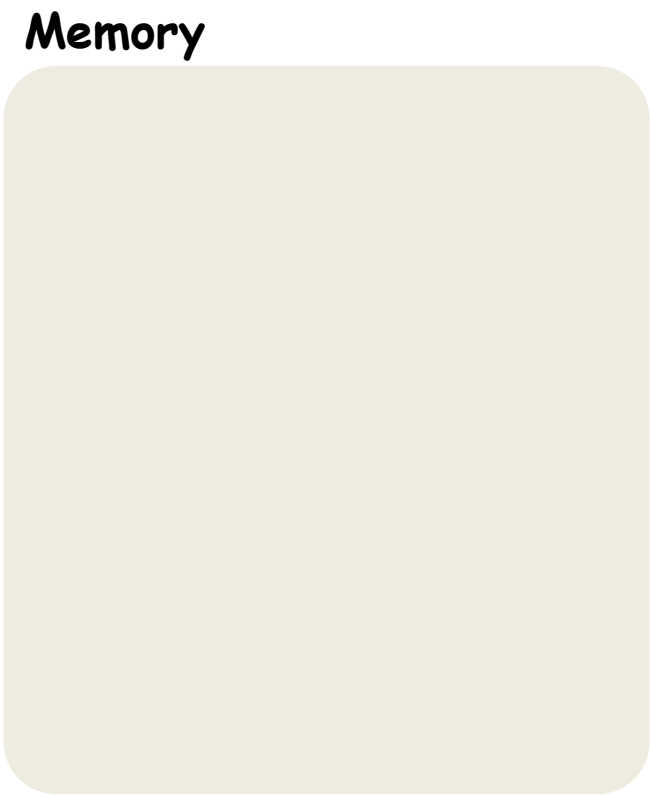
```
var fireball1 = Fireball?
var fireball2 = Fireball?
fireball1 = Fireball(centerX: 7.0, centerY: 5.0, radius: 6.0)
fireball2 = Fireball(centerX: 7.0, centerY: 8.0, radius: 4.0)

fireball1!.move(1.0, 1.0)
fireball2!.move(1.0, 1.0)

fireball1 = nil
fireball2 = nil
```



Concept of Pointers

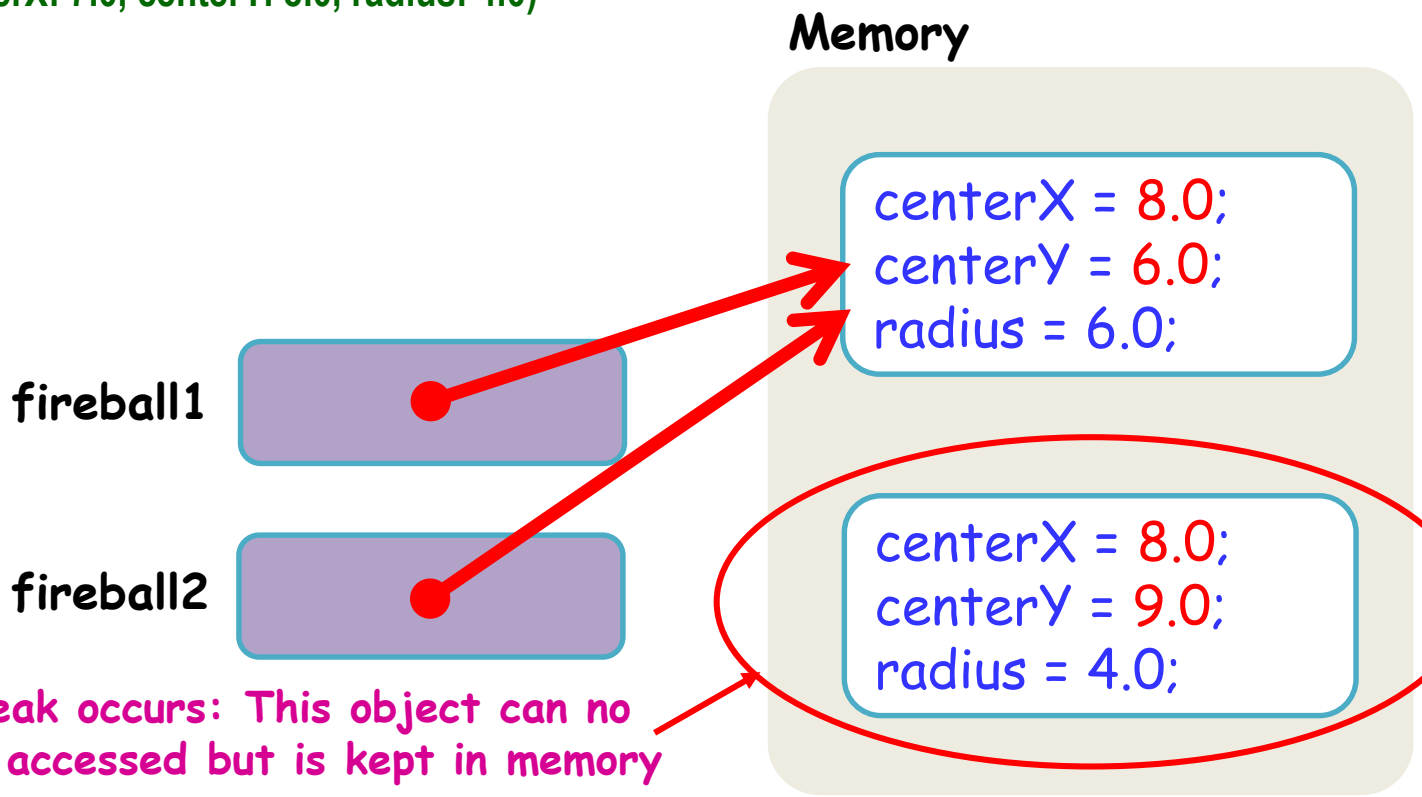
```
var fireball1 = Fireball?
var fireball2 = Fireball?
fireball1 = Fireball(centerX: 7.0, centerY: 5.0, radius: 6.0)
fireball2 = Fireball(centerX: 7.0, centerY: 8.0, radius: 4.0)

fireball1!.move(1.0, 1.0)
fireball2!.move(1.0, 1.0)

fireball1 = nil
fireball2 = nil
```





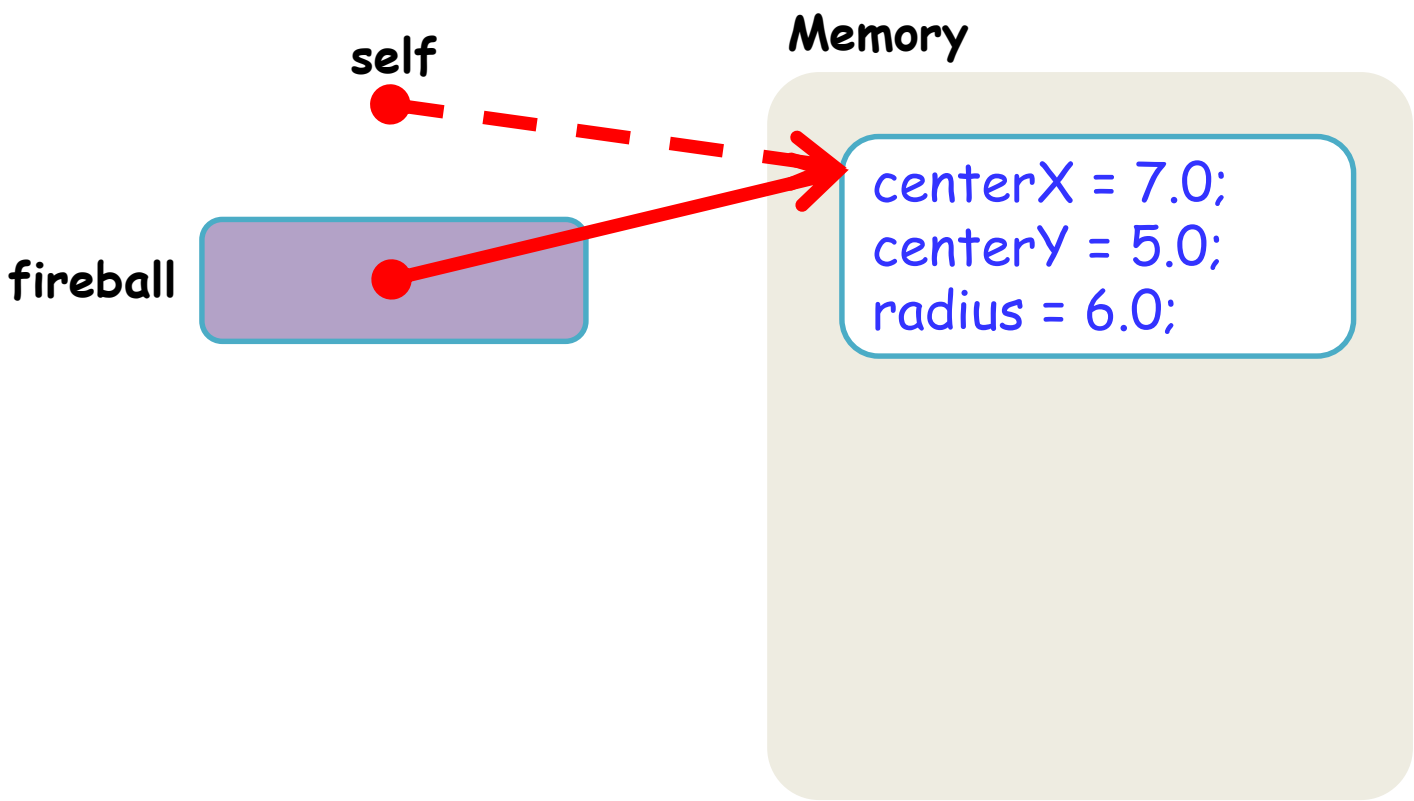
Concept of Pointers – Memory Leak

```
var fireball1 = Fireball?
var fireball2 = Fireball?
fireball1 = Fireball(centerX: 7.0, centerY: 5.0, radius: 6.0)
fireball2 = Fireball(centerX: 7.0, centerY: 8.0, radius: 4.0)

fireball1!.move(1.0, 1.0)
fireball2!.move(1.0, 1.0)

fireball2 = fireball1

fireball1 = nil
fireball2 = nil
```



Concept of Pointers – The "self" Reserved Word

- Imagine there is a pointer called **self** which points to the object itself when an object is created.



Concept of Pointers – The "self" Reserved Word

- Sometimes, we need to refer to the method or variable of the fireball object within a method implementation of the Fireball class
  - We should use the keyword "**self**" as the subject

```
class Fireball {
    var centerX: Float
    var centerY: Float
    var radius: Float

    init(centerX:Float,centerY:Float,radius:Float) {
        self.centerX = centerX
        self.centerY = centerY
        self.radius = radius
    }
```

Member variables

Function

Function Definition

- Function parameters have both an *argument label (external parameter name)* and a *parameter name (local parameter name)*. An argument label is used to label arguments passed to a function call. A parameter name is used in the implementation of the function.

```
func someFunction(firstParameterName: Int, secondParameterName: Int) {
        // function body goes here
        // firstParameterName and secondParameterName refer to
        // the argument values for the first and second parameters
}
someFunction(firstParameterName: 1, secondParameterName: 2)
// Did not define argument label and so parameter name is used in
function call
```

- All parameters must have unique parameter names. Although it's possible for multiple parameters to have the same argument label, unique argument labels help make your code more readable.

Function Definition

- You write an argument label before the parameter name it supports, separated by a space:

```
func someFunction(argumentLabel parameterName: Int) {
        // function body goes here, and can use parameterName
        // to refer to the argument value for that parameter
}
Call: someFunction(argumentLabel: 7506)
```

- If you provide an argument label for a parameter, that argument label must *always* be used when you call the function.

Function Definition

- Example:

```
func greet(person: String, from hometown: String) -> String {
    return "Hello \(person)!  Glad you could visit from \(hometown)."
}
print(greet(person: "Bill", from: "Cupertino"))
// Prints "Hello Bill!  Glad you could visit from Cupertino."
```

Function Definition

- Here's a version of the sayHello(_:) function that takes the names of two people and returns a greeting for both of them:

```
func sayHello(to person: String, and anotherPerson: String) -> String {
        return "Hello \(person) and \(anotherPerson)!"
}
print(sayHello(to: "Bill", and: "Ted"))
// prints "Hello Bill and Ted!"
```

- By specifying external parameter names for both parameters, both the first and second arguments to the sayHello(to:and:) function must be labeled when you call it.
- The use of external parameter names can allow a function to be called in an expressive, sentence-like manner, while still providing a function body that is readable and clear in intent.

Function Definition

- If you do not want to use an argument label for any parameter of a function in the function call, write an underscore (_) instead of an explicit argument label for that parameter.

```
func someFunction(_ firstParameterName: Int, secondParameterName: Int) {
        // function body goes here
        // firstParameterName and secondParameterName refer to
        // the argument values for the first and second parameters
}
someFunction(1, secondParameterName: 2)
```

Function Definition

ClassDemo 〉 ClassDemo 〉 Fireball 〉 Fireball

```swift
//
//  Fireball.swift
//  ClassDemo
//
//  Created by Tat Wing Chim on 7/3/2023.
//

import UIKit

class Fireball: NSObject {
    var centerX: Float
    var centerY: Float
    var radius: Float

    init(centerX: Float, centerY: Float, radius: Float) {
        self.centerX = centerX
        self.centerY = centerY
        self.radius = radius
    }

    func move(_ moveX: Float, _ moveY: Float) {
        self.centerX += moveX
        self.centerY += moveY
    }

    deinit {

    }
}
```

ClassDemo 〉 ClassDemo 〉 ViewController 〉 No Selection

```swift
//
//  ViewController.swift
//  ClassDemo
//
//  Created by Tat Wing Chim on 7/3/2023.
//

import UIKit

class ViewController: UIViewController {

    override func viewDidLoad() {
        super.viewDidLoad()
        var fireball1: Fireball?
        var fireball2: Fireball?
        fireball1 = Fireball(centerX: 7.0, centerY: 5.0, radius: 6.0)
        fireball2 = Fireball(centerX: 7.0, centerY: 8.0, radius: 4.0)
        fireball1!.move(1.0, 1.0)
        fireball2!.move(1.0, 1.0)
        fireball1 = nil
        fireball2 = nil
    }

}
```

Function Definition (Summary)

Function Definition:	Function Call:

With both argument labels and parameter names:
func ABC(e1 i1: Int, e2 i2: Int, e3 i3: Int) {
// i1 = e1, i2 = e2, i3 = e3 internally
// Use i1, i2 and i3 for internal operations
}
	ABC(e1:75, e2:6, e3:2020)

With only parameter names:
func ABC(i1: Int, i2: Int, i3: Int) {
// Use i1, i2 and i3 for internal operations
}
	ABC(i1:75, i2:6, i3:2020)

Without argument labels and parameter names:
func ABC(_ i1: Int, _ i2: Int) {
...
}
	ABC(75, 6)

Function Definition

- You can define a *default value* for any parameter in a function by assigning a value to the parameter after that parameter's type. If a default value is defined, you can omit that parameter when calling the function.

```
func someFunction(parameterWithoutDefault: Int, parameterWithDefault: Int = 12) {
        // If you omit the second argument when calling this function, then
        // the value of parameterWithDefault is 12 inside the function body.
}
someFunction(parameterWithoutDefault: 3, parameterWithDefault: 6)
// parameterWithDefault is 6
someFunction(parameterWithoutDefault: 4)
// parameterWithDefault is 12
```

Function Definition

- Variadic Parameters: A variadic parameter accepts zero or more values of a specified type. You use a variadic parameter to specify that the parameter can be passed a varying number of input values when the function is called. Write variadic parameters by inserting three period characters (...) after the parameter's type name.
- A function may have at most one variadic parameter.

```
func arithmeticMean(_ numbers: Double...) -> Double {
        var total: Double = 0
        for number in numbers {
                total += number
        }
        return total / Double(numbers.count)
}
arithmeticMean(1, 2, 3, 4, 5)
// returns 3.0, which is the arithmetic mean of these five numbers
arithmeticMean(3, 8.25, 18.75)
// returns 10.0, which is the arithmetic mean of these three numbers
```

Function Definition

- In-Out Parameters: Function parameters are constants by default. Trying to change the value of a function parameter from within the body of that function results in a compile-time error. This means that you can't change the value of a parameter by mistake. If you want a function to modify a parameter's value, and you want those changes to persist after the function call has ended, define that parameter as an in-out parameter instead.

```
func swapTwoInts(_ a: inout Int, _ b: inout Int) {
        let temporaryA = a
        a = b
        b = temporaryA
}
```

Recall "pass by value" in C++ functions. Your function can update the value of the parameters but the change will not take effect after function invocation.

Please refer to the site below for more details:
https://developer.apple.com/library/content/documentation/Swift/Conceptual/Swift_Programming_Language/Functions.html

Instance and Class Functions

```
class AClass {
    class func aClassMethod() {
        print("I am a class method")
    }
    class func bClassMethod() {
        aClassMethod()
    }
    func anInstanceMethod() {
        print("anInstanceMethod. Calling bClassMethod().")
        AClass.bClassMethod()
    }
}

AClass.aClassMethod()        // displays “I am a class method”
let aClass = AClass() / var aClass = Aclass()
aClass.anInstanceMethod()
// displays “anInstanceMethod. Calling bClassMethod(). I am a class method”
```

String & Array

String

- Mutable String
  - **var S = “hi”**
  - **var S: String = “hi”**
  - **var S = String(“hi”)**
- Non-Mutable String
  - **let S = “hi”**
  - **let S: String = “hi”**
  - **let S = String(“hi”)**
- Determine the length of a string
  - Use global function **countElements(someStringValue)**
  - Since Swift 5, can also use **someStringValue.count**
- Find substring
  - **var S = “hello Swift”**
  - **var range = S.rangeOfString(“Swift”)**
  - **range** is an **NSRange** object, which contains two parts:
    - **location = 6**
    - **length = 5**

Array of Primitive Data Types

- To make an array holding elements of primitive data type (e.g. **int**)
  - Mutable Array
    - **var myArray: Array<Int> = [555, 666]**
    - **var myArray: [Int] = [555, 666]**
    - **var myArray = [555, 666]**
  - Constant Array
    - **let myArray: Array<Int> = [555, 666]**
    - **let myArray: [Int] = [555, 666]**
    - **let myArray = [555, 666]**
  - Array elements can be read (and write if mutable)
    - **var somevalue = myArray[index]**
    - **myArray[index] = somevalue**

Array of Objects

- To make a mutable array holding objects of a class (e.g. Fireball)
  - Approach 1: **var myObjectArray:[Fireball]=[Fireball1, Fireball2]**
  - Approach 2: **var myObjectArray:[Fireball]=[]**

    myObjectArray.append(Fireball1)

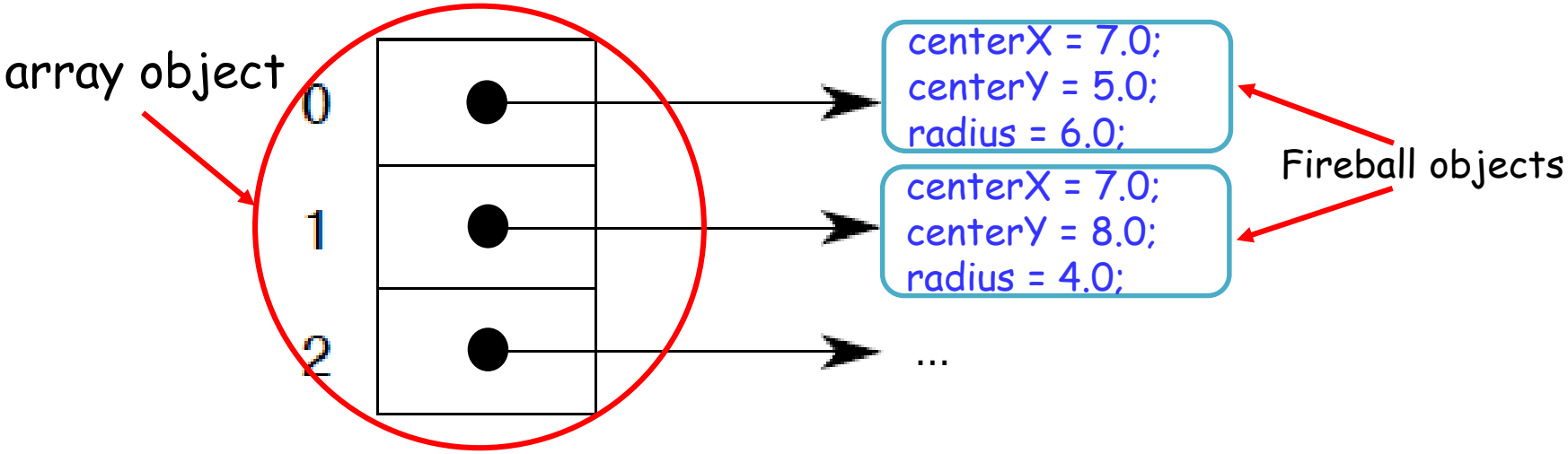
    myObjectArray.append(Fireball2)
- To make a constant array holding objects of a class (e.g. Fireball)
  - **let myObjectArray:[Fireball]=[Fireball1, Fireball2]**
- To loop through elements in an array
  - **for i in 0..<myObjectArray.count {**

    print("myObjectArray[\(i)].radius=\(myObjectArray[i].radius)")

    }
- It would output
  - **myObjectArray[0].radius=6**
  - **myObjectArray[1].radius=4**

Array of Objects

- To make an array holding objects of a class (e.g. Fireball)
  - Create a constant or mutable array object with base type Fireball
  - Add the objects of the same class into the array object



- Finding the size of it is important when we want to traverse the whole array
  - We can do so by invoking the count method
  - e.g. **array.count**

Optionals

Optionals in Swift

- Swift is a type safe language
  - How do we handle the absence of a value?
- Swift also introduces optional types, which handle the absence of a value
  - Optionals are similar to using nil with pointers in Objective-C, but they work for any type, not just classes
- Some places optionals are useful:
  - When a property can be there or not there
  - When a method can return a value or nothing, like searching for a match in an array
  - When a method can return either a result or get an error and return nothing
  - Delegate properties (which don't always have to be set)
  - For a large resource that might have to be released to reclaim memory

Optionals in Swift (Example 1)

An Optional is a variable that can either have a value or be 'nil'.

- Source

```
let possibleNumber = "123"
let convertedNumber = Int(possibleNumber)
// convertedNumber is inferred to be of type “Int?”, or “optional Int”
if convertedNumber != nil {
   print("convertedNumber contains some integer value.")
}
```

- Output
  - convertedNumber contains some integer value.

Optionals in Swift (Example 2)

- Source

```
let possibleNumber = "123a"
var convertedNumber: Int?
convertedNumber = Int(possibleNumber)
if convertedNumber != nil {
    print("convertedNumber contains some integer value
                                                \(convertedNumber).")
}
else
    print("\'\(possibleNumber)\' cannot be converted to integer.")
```

? to indicate optional Int

- Output
  - '123a' cannot be converted to integer.

Optionals in Swift (Example 3)

- Source

```
let possibleNumber = "1234"
var convertedNumber: Int?
convertedNumber = Int(possibleNumber)
if convertedNumber != nil {
    print("convertedNumber contains some integer value
                                        \(convertedNumber!).")
}
else
    print("\'\(possibleNumber)\' cannot be converted to integer.")
```

! to indicate the definite presence of non-nil value

- Output
  - convertedNumber contains some integer value 1234.

Example from Workshop

- Passing data to another View Controller:

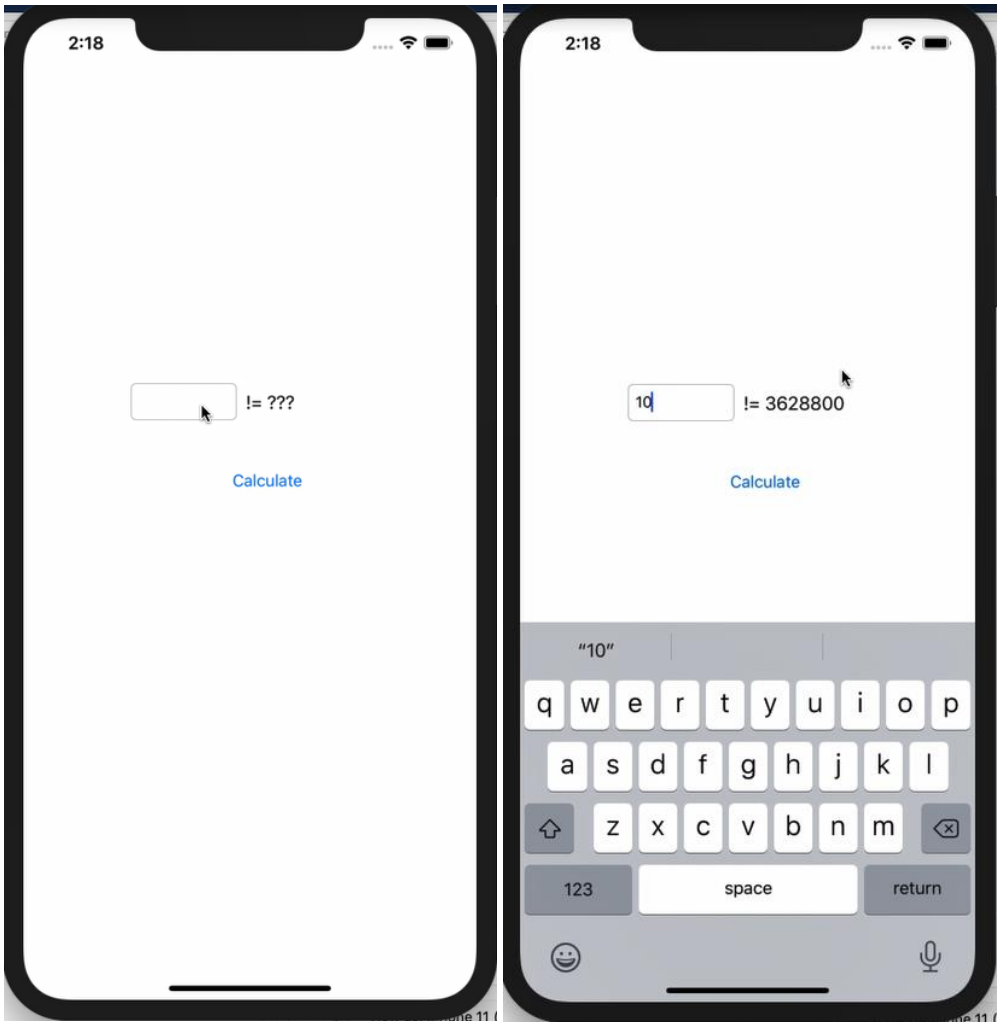
```
override func prepareForSegue(segue: UIStoryboardSegue, sender: AnyObject?)
{
   print("enter prepare for segue …")
   if (segue.identifier == "loginToGameBoardSeg") {
      let nav = segue.destinationViewController as! UINavigationController
      let vc = nav.topViewController as! GameBoardViewController
      vc.setNavTitle(userNameTF.text!)
   }
}
```

as: Forced conversion from one class to another
as!: The conversion may fail
userNameTF.text!: This field contains non-nil value

Exercise

UIImage & UIImageView

- Implement a simple application such that the user interface contains 3 components:
  - 1 UITextField (to accept an integer from the user)
  - 1 UILabel (to display the result)
  - 1 “Calculate” Button
- The user can enter an integer into the UITextField. Upon the “Calculate” Button is clicked, the factorial of the integer entered will be calculated. The result will then be displayed in the UILabel.



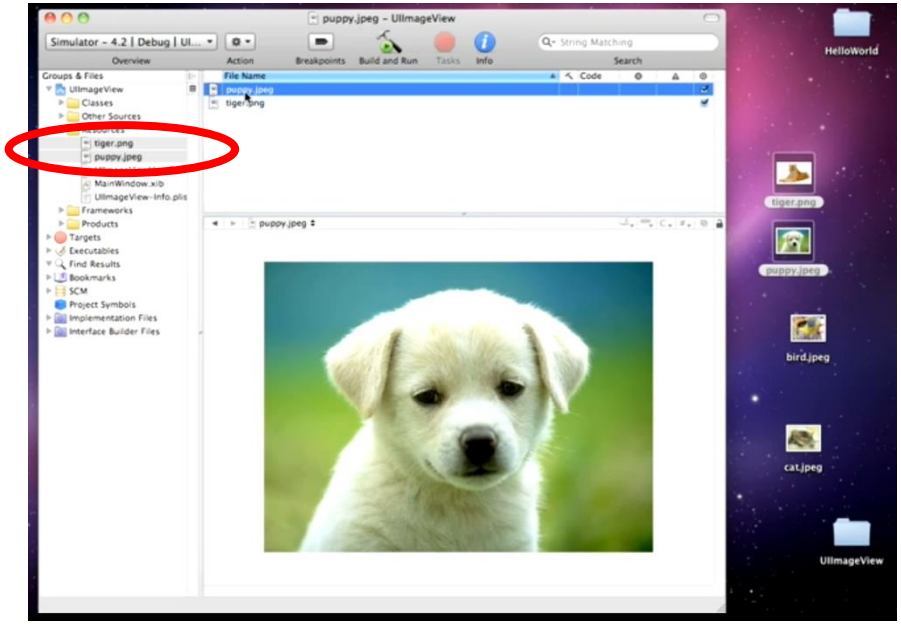
Image

UIImage & UIImageView

- **UIImage:**
  - A high-level way to display image data.
  - You can create images from files or from raw image data you receive.
  - The class offers several options for drawing images to the current graphics context using different blend modes and opacity values.
- **UIImageView:**
  - An image view object provides a view-based container for displaying either a single image or for animating a series of images.
  - For animating the images, the UIImageView class provides controls to set the duration and frequency of the animation. You can also start and stop the animation freely.

Adding Images

- Drag the images into the project folder
- After that, you can pre-view the images



Adding Images

- In the class ViewController.swift, you should find a function named viewDidLoad (something that will happen after the view is loaded) as follows:

```
override func viewDidLoad() {
        super.viewDidLoad()
}
```

Recall "onCreate()" method in Android

Adding Images

- Modify it as follows:

```
override func viewDidLoad() {
    let myImage = UIImageView(image: UIImage(named:
    "puppy.jpeg"))
    // Allocate a memory location for a UIImageView and initialize it by
    the image “puppy.jpeg”
    view.addSubview(myImage)
    // view is the whole screen. myImage is a subview in it.
    super.viewDidLoad()
}
```

Adding Images

- Next add in the following to include the second image:

```
override func viewDidLoad() {
    …
    let myImage2 = UIImageView(image: UIImage(named: "tiger.png"))
    view.addSubview(myImage2)
    super.viewDidLoad()
}
```

Positioning Images

- Next center the second image:

```
override func viewDidLoad() {
    …
    let myImage2 = UIImageView(image: UIImage(named: "tiger.png"))
    view.addSubview(myImage2)
    myImage2.center = CGPoint(x: 150, y: 200)
    // CGPoint(x: x1, y: y1) is an inline function that populates a CGPoint (2D
    vector) struct with the values you pass in (x1 as x-coordinate and y1 as
    y-coordinate).
    super.viewDidLoad()
}
```

Scaling Images

- Next make the second image smaller:

```
override func viewDidLoad() {
    …
    let myImage2 = UIImageView(image: UIImage(named: "tiger.png"))
    view.addSubview(myImage2)
    myImage2.center = CGPoint(x: 150, y: 200)
    myImage2.frame = CGRect(x: 0, y: 0, width: 50, height: 25)
    // CGRect(x: x1, y: y1, width: w, height: h) is a function to define a
    frame to wrap around a component (at location (x1, y1) and with
    dimension w x h pixels).
    super.viewDidLoad()
}
```

Scaling and Positioning Images

- How about changing

  myImage2.center = CGPointMake(x: 150, y: 200)

  myImage2.frame = CGRectMake(x: 0, y: 0, width: 50, height:25)

  into

  myImage2.frame = CGRectMake(x: 0, y: 0, width: 50, height: 25)

  myImage2.center = CGPointMake(x: 150, y: 200)

  ?
- The later positioning instruction overrides the first one!

Scaling Images

- Finally, it is a good habit to release memory reserved for images after they are loaded and displayed.

```
myImage.deallocate()
myImage2.deallocate()
```

- Yet, no error or no significant performance degradation even if you do not release them…

Touching Events

Touching Events

- Xcode provides several functions to detect touching events:
  - `override func touchesBegan(_ touches: Set<UITouch>, with event: UIEvent?)`
    - What you want your app to happen when you touch onto the screen?
  - `override func touchesMoved(_ touches: Set<UITouch>, with event: UIEvent?)`
    - What you want your app to happen after you touch onto and start moving across the screen?
  - `override func touchesEnded(_ touches: Set<UITouch>, with event: UIEvent?)`
    - What you want your app to happen when your finger leaves the screen?

Example

```
class YourView: UIView {
        override func touchesBegan(touches: Set<UITouch>, with event: UIEvent?) {
                if let touch = touches.first {
                        let currentPoint = touch.locationInView(self)
                        // do something with your currentPoint (e.g. image1.center = currentPoint)
                }
        }
        override func touchesMoved(touches: Set<UITouch>, with event: UIEvent?) {
                if let touch = touches.first {
                        let currentPoint = touch.locationInView(self)
                        // do something with your currentPoint (e.g. image1.center = currentPoint)
                }
        }
        override func touchesEnded(touches: Set<UITouch>, with event: UIEvent?) {
                if let touch = touches.first {
                        let currentPoint = touch.locationInView(self)
                        // do something with your currentPoint (e.g. image1.center = currentPoint)
                }
        }
}
```

Multiple Touches

- Only works in real device!
- Handling multiple touches:

```
override func touchesMoved(_ touches: Set<UITouch>, with event: UIEvent?) {
        for touch in touches {
                let newLocation = touch.location(in: self)
                …
        }
}
```

Example:
https://developer.apple.com/documentation/uikit/touches_presses_and_gestures/handling_touches_in_your_view/implementing_a_multi-touch_app

Chapter 10.

End

2023-2024
COMP7506 Smart Phone Apps Development
Dr. T.W. Chim (E-mail: twchim@cs.hku.hk)
Department of Computer Science, The University of Hong Kong