Lecture 2:
Introduction to Swift

Prof. Phillip De Leon

Outline

- Tuples
- Functions and Closures (Chapter 21)
- Classes and Structures (Chapter 21)
- Properties (Chapter 21)
- Methods (Chapter 21)
- Subclassing (to be studied outside of class)
- Protocols (Chapter 21)

Tuples

- Unnamed tuple

```
let tipAndTotal = (4.00, 25.19)
let x = tipAndTotal.0 // access tuple elements
var y = tipAndTotal.1

// decomposing tuple by name
let (theTipAmt, theTotal) = tipAndTotal
let x = theTipAmt
var y = theTotal
```

Tuples (cont.)

- Named tuple

```
let tipAndTotalNamed = (tipAmt:4.00, total:25.19)
tipAndTotalNamed.tipAmt
tipAndTotalNamed.total
```

Outline

- Tuples
- Functions and Closures
- Classes and Structures
- Properties
- Methods
- Subclassing (to be studied outside of class)
- Protocols

Functions

- Functions bundle functionality into small re-usable units and live outside of objects (methods are functions that belong to an object)
- Example of a function declaration and invocation
- Calling order associates argument values

keyword function name argument list return type

```
func areaOfRectangle(width: Double, height: Double) -> Double {
    return width * height
}

let area = areaOfRectangle(width:20, height:10)
```

Argument labels are required!

Functions (cont.)

- Functions can return tuples (!)

return tuple

```
let total = 21.19
let taxPct = 0.06
let subtotal = total / (taxPct + 1)
func calcTipWithTipPct(tipPct:Double) -> (tipAmt:Double, total:Double) {
    let tipAmt = subtotal * tipPct
    let finalTotal = total + tipAmt
    return (tipAmt, finalTotal)
}
calcTipWithTipPct(tipPct:0.20)
```

Functions (cont.)

- Functions can use external and internal argument labels

internal label
external label
internal label
external label

```
func areaOfRectangle(width w: Double, height h: Double) -> Double {
    return w * h
}
```

internal label

external label

```
let area = areaOfRectangle(width: 20, height: 10)
```

Functions (cont.)

- Functions are types
- Create a variable of function type, assign a reference to it, use variable to call function
- You can pass a function as an argument to another function
- You can return a return a function from a function

Functions (cont.)

- Example of creating a variable of function type

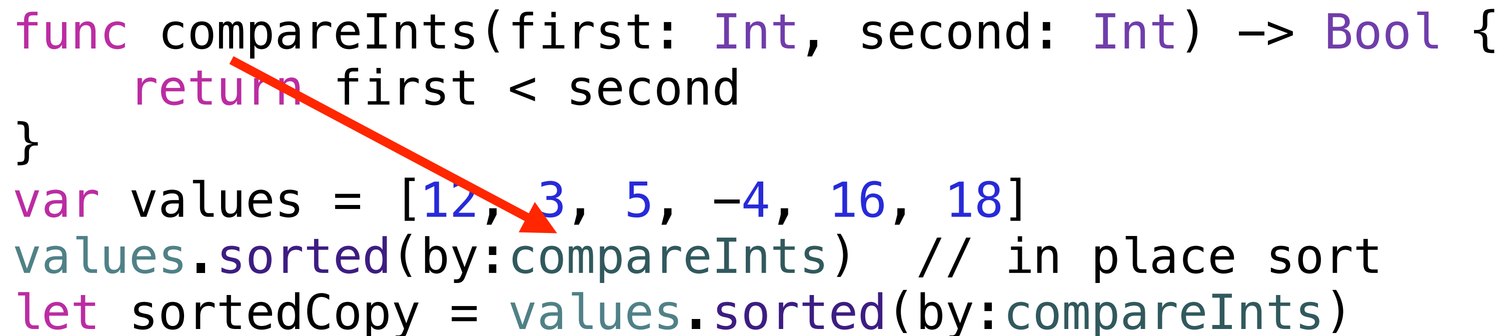
```
func double(number: Double) -> Double {
    return 2 * number
}
func quadruple(number: Double) -> Double {
    return 4 * number
}

var operation: (Double) -> Double

operation = double
operation(2)
operation = quadruple
operation(2)
```

Functions and Closures

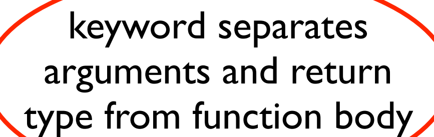
- Several functions in Swift standard library accept function arguments including sorted() (see “help sorted”)



Closures

- Closure: function written in-place as the argument to another function

```
let sorted = values.sorted(by:{(first: Int, second: Int) -> Bool in
    return first < second
})
```



- We can reduce the closure to one line of code!

```
let sorted = values.sorted(by:) { $0 < $1 }
```

  - If closure is last argument, we can bring it outside arg list
  - Swift can infer from sorted() what the arg types are

Outline

- Tuples
- Functions and Closures
- Classes and Structures
- Properties
- Methods
- Subclassing (to be studied outside of class)
- Protocols

Classes vs. Structures

- Classes are reference types, structures are value types
- Structure instances are passed by value, i.e. when you pass an instance of a structure to a function, the instance is copied
    - Same with returning an instance from a function or assigning the value of a variable that refers to a structure to another variable,
- Class instances (objects) are passed by reference the instance is not copied
- Swift strings, arrays, and dictionaries are all implemented as structures
- Both classes and structures may contain properties, initializers, and methods

Structures

- Both classes and structures may contain properties, initializers, and methods

```
struct CircleStruct {
    var radius: Double          ← Property

    func getArea() -> Double {          ← Method
        return M_PI * radius * radius
    }

    func getCircumference() -> Double {          ← Method
        return 2 * M_PI * radius
    }
}

var circleStruct = CircleStruct(radius: 10)          ← Uses "free" initializer
let r = circleStruct.radius          ← Access the property (accessor)
```

Structures

- Example of a structure with two initializers

```
struct CircleStruct {
    var radius: Double = 1

    init() {
    }

    init(radius: Double) {
        self.radius = radius
    }
.
.
.
```

Initializer for no input args, defaults radius = 1

Initializer for input arg

Example object instantiation:

```
let circleStructDefault = CircleStruct()
let circleStructRadius = CircleStruct(radius: 10.0)
```

- The self variable represents the instance of the structure that's being initialized

Classes

- Swift does not provide a "free" init() initializer
- Classes are not value objects, so assigning a class instance to a variable or passing one to a function does not create a copy

```
class CircleClass {
    var radius: Double = 1

    init() {
    }

    init(radius: Double) {
        self.radius = radius
    }

    func getArea() -> Double {
        return Double.pi * radius * radius
    }
    func getCircumference() -> Double {
        return 2 * Double.pi * radius
    }
}
```

Examples:

```
var circleClass = CircleClass(radius: 10)
var newCircleClass = circleClass // Not a copy!
newCircleClass.radius = 32
newCircleClass.radius // Result is 32
circleClass.radius // Result is 32
```

Outline

- Tuples
- Functions and Closures
- Classes and Structures
- Properties
- Methods
- Subclassing (to be studied outside of class)
- Protocols

Properties

- Example of a class with stored vs. computed properties

```
class CircleClass {
    var radius: Double = 1
    var area: Double {
        return Double.pi * radius * radius
    }
    var circumference: Double {
        return 2 * Double.pi * radius
    }

  init() {
  }
  init(radius: Double) {
        self.radius = radius
  }
}
```

Stored property (var radius)

Computed property (doesn't store value) (var area)

Computed property (var circumference)

Examples:

```
let circleClass = CircleClass(radius: 10)
circleClass.area
circleClass.circumference
```

Outline

- Tuples
- Functions and Closures
- Classes and Structures
- Properties
- Methods
- Subclassing (to be studied outside of class)
- Protocols

Methods

- A method is a function that belongs to a class

```
let myInstance = MyObject()   // instantiate the object
. . .
myInstance.doSomething()      // call the method
```

- Three kinds of methods: "instance", "class", and "init"
  - Instance methods are like functions applied to existing (instantiated) object
  - Class methods or "factory" methods used to create new object instances
  - Init methods are used to initialize an object's properties

Methods

- Methods' argument-naming conventions (see book for other options)

required argument

optional argument
with default value

```
func adjustRadiusBy(amount: Double, times: Int = 1) {
    radius += amount * Double(times)
}
```

- Method calls

```
var circleClass = CircleClass(radius: 10)
circleClass.radius
circleClass.adjustRadiusBy(amount: 5, times: 3)
circleClass.radius
circleClass.adjustRadiusBy(amount: 5)   // "times" is defaulted to 1
circleClass.radius
```

Outline

- Tuples
- Functions and Closures
- Classes and Structures
- Properties
- Methods
- Subclassing (to be studied outside of class)
- Protocols

Protocols

- A protocol is the declaration of a group of methods, initializers, and properties to which a class can conform to by providing implementations of them
- The example Resizable protocol requires any type that claims conformance to it to provide the two properties, one initializer, and one function that it declares.
- The protocol itself does not define anything; it simply dictates what a conforming type must do

```
protocol Resizable {
    var width: Float { get set }
    var height: Float { get set }
    init(width: Float, height: Float)
    func resizeBy(wFactor: Float, hFactor: Float) -> Void
}
```

Protocols (cont.)

- Example class conforming to the Resizable and Printable protocols
- Class provides the implementation of the protocol's required features

```
class Rectangle : Resizable, Printable {
    var width: Float
    var height: Float
    var description: String {
        return "Rectangle, width \(width), height \(height)"
    }
    required init(width: Float, height: Float) {
        self.width = width
        self.height = height
    }
    func resizeBy(wFactor: Float, hFactor: Float) -> Void {
        width *= wFactor
        height *= hFactor
    }
}
```

Rectangle objects conform to these protocols

In case Rectangle is subclassed, the subclass must (required!) have an init()

Protocols (cont.)

- Example of an instance of Rectangle class that conforms to Resizable protocol

```
let r: Resizable = Rectangle(width: 10, height: 20)
var s: Resizable = Rectangle(width: 10, height: 20)
r.resizeBy(wFactor: 4.0, hFactor: 2.0)

s = r // change reference
r = s // compiler error (cannot change reference)


// We cannot change reference (pointer) to the object
// but we can change the properties
```