Developing applications for iOS

Lecture 3:
A Tour of Swift

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

Content

- Strings and Characters
- Collection Types
- Control Flow
- Functions and Closures
- Classes, Structures, Enumerations
- Properties and Methods

Strings and Characters

- A string is a series of characters, such as `"hello, world"`. Swift strings are represented by the `String` type. The contents of a `String` can be accessed in various ways, including as a collection of `Character` values.

- Swift's `String` type is a value type. If you create a new `String` value, that `String` value is copied when it is passed to a function or method, or when it is assigned to a constant or variable.

Strings and Characters

- To create an empty `String` value as the starting point for building a longer string, either assign an empty string literal to a variable, or initialize a new `String` instance with initializer syntax:

```
var emptyString1 = "" // empty string literal
var emptyString2 = String() // initializer
```

- Find out whether a `String` value is empty by checking its Boolean `isEmpty` property:

```
if emptyString.isEmpty
{
   print("Nothing to see here")
}
```

Strings and Characters

- You can access the individual `Character` values for a `String` by iterating over the string with a for-in loop:

```
for c in "Donald Duck!"
{
    print(c)
}
```

- `String` values can be constructed by passing an array of `Character` values as an argument to its initializer:

```
let chars: [Character] = ["D","u","c","k","!"]
let duck = String(chars)
print(duck)
```

Strings and Characters

- `String` values can be added together (or concatenated) with the addition operator to create a new `String` value:

```
let firstName = "Donald"
let lastName = " Duck"
var name = firstName + lastName
```

- You can append a `Character` value to a `String` variable with the `String` type's `append()` method:

```
let exclamationMark: Character = "!"
name.append(exclamationMark)
```

- String interpolation is a way to construct a new `String` value from a mix of constants, variables, literals, and expressions:

```
let x = 3
let message = "\(x) times 2 is \(x * 2)"
// message is "3 times 2 is 6"
```

Strings and Characters

- To retrieve a count of the `Character` values in a string, use the `count` property of the string:

```
var w = "cafe"
var n = w.count
print("\(w) has \(n) characters")
// Prints "cafe has 4 characters"
```

- `String` has an associated index type, `String.Index`, which corresponds to the position of each `Character` in the string. You can use subscript syntax to access the `Character` at a particular `String` index:

```
let w = "Butterfly!"
w[w.startIndex] // B
w[w.index(before: w.endIndex)] // !
w[w.index(after: w.startIndex)] // u
let index = w.index(w.startIndex, offsetBy: 7)
w[index] // l
```

Strings and Characters

- Attempting to access an index outside of a string's range or a `Character` at an index outside of a string's range will trigger a runtime error:

```
w[w.endIndex] // Error
w.index(after: w.endIndex) // Error
```

- Use the `indices` property to access all of the indices of individual characters in a string:

```
for index in w.indices
{
    print("\(w[index]) ", terminator: "")
}
// Prints "B u t t e r f l y ! "
```

Strings and Characters

- To insert a single character into a string at a specified index, use the `insert(_:at:)` method:

```
var w = "fly"
w.insert("!", at: w.endIndex)
// w now equals "fly!"
```

- To insert the contents of another string at a specified index, use the `insert(contentsOf:at:)` method:

```
w.insert(contentsOf: "Butter",
         at: w.startIndex)
// w now equals "Butterfly!"
```

Strings and Characters

- To remove a single character from a string at a specified index, use the `remove(at:)` method:

```
w.remove(at: w.index(before: w.endIndex))
// w now equals "Butterfly"
```

- To remove a substring at a specified range, use the `removeSubrange(_:)` method:

```
let last = w.index(w.startIndex, offsetBy: 6)
let range = w.startIndex..<last
w.removeSubrange(range)
// w now equals "fly"
```

Strings and Characters

- String and character equality is checked with the “equal to” operator (`==`) and the “not equal to” operator (`!=`):

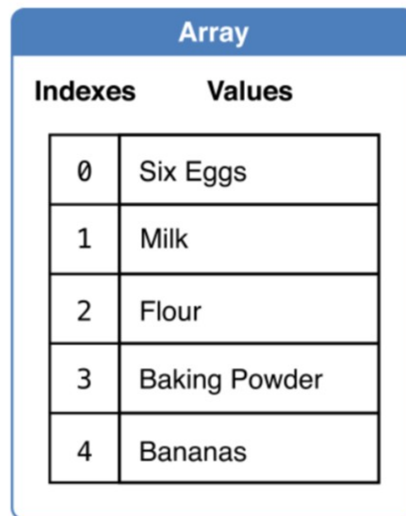
```
let quote = "I'll be back!"
let sameQuote = "I'll be back!"
if quote == sameQuote
{
   print("These two quotes are equal!")
}
```

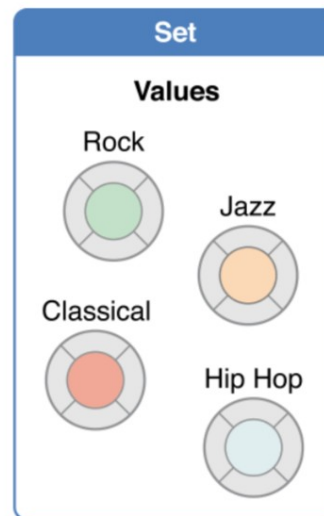
- To check whether a string has a particular string prefix or suffix, call the string’s `hasPrefix(_:)` and `hasSuffix(_:)` methods, both of which take a single argument of type `String` and return a `Boolean` value:

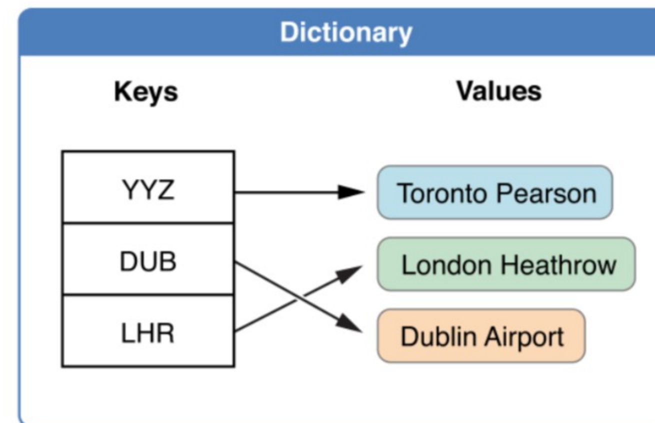
```
let duck = "Donald Duck!"
if duck.hasPrefix("Donald")
{
   print("Meet Donald, my favorite duck!")
}
```

Collection Types

- Swift provides three primary collection types, known as arrays, sets, and dictionaries, for storing collections of values.
- Arrays are ordered collections of values.
- Sets are unordered collections of unique values.
- Dictionaries are unordered collections of key-value pairs.







- Arrays, sets, and dictionaries in Swift are always clear about the types of values and keys that they can store. This means that you cannot insert a value of the wrong type into a collection by mistake.

Arrays

- The type of a Swift array is written as `Array<Element>`, where `Element` is the type of values the array is allowed to store. You can also write the type of an array in shorthand form as `[Element]` (preferred). You can create an empty array of a certain type using initializer syntax:

```
var v = [Int]()
print("v contains \(v.count) items.")
// Prints "v contains 0 items."
```

- Swift’s Array type also provides an initializer for creating an array of a certain size with all of its values set to the same default value:

```
var zeros = Array(repeating: 0.0, count: 3)
/* zeros is of type [Double], and equals
[0.0, 0.0, 0.0] */
```

Arrays

- You can create a new array by adding together two existing arrays with compatible types with the addition operator:

```
var ones = Array(repeating: 1.0, count: 3)
/* ones is of type [Double], and equals [1.0,
1.0, 1.0] */

var v = zeros + ones
/*  v  is  inferred  as  [Double],  and  equals
[0.0, 0.0, 0.0, 1.0, 1.0, 1.0] */
```

- You can also initialize an array with an array literal:

```
var v = ["Eggs", "Milk", "Apples"]
```

- To find out the number of items in an array, check its read-only `count` property:

```
print("My list contains \(v.count) items.")
// Prints "My list contains 3 items."
```

Arrays

- Use the Boolean `isEmpty` property as a shortcut for checking whether the `count` property is equal to 0:

```
if v.isEmpty {
    print("The shopping list is empty.")
} else {
    print("The shopping list is not empty.")
}
// Prints "The shopping list is not empty."
```

- You can add a new item to the end of an array by calling the array's `append(_:)` method:

```
v.append("Flour")
```

- Alternatively, append an array of one or more compatible items with the addition assignment operator (`+=`):

```
v += ["Chocolate", "Cheese", "Butter"]
// v now contains 7 items
```

Arrays

- Retrieve a value from the array by using subscript syntax:

```
var firstItem = v[0]
// firstItem is equal to "Eggs"
```

- You can use subscript syntax to change an existing value at a given index:

```
v[0] = "Six eggs"
// the first item is now equal to "Six eggs"
```

- You can also use subscript syntax to change a range of values at once, even if the replacement set of values has a different length than the range you are replacing:

```
v[4...6] = ["Bananas", "Grapes"]
// v now contains 6 items
```

Arrays

- To insert an item into the array at a specified index, call the array's `insert(_:at:)` method:

```
v.insert("Maple Syrup", at: 0)
```

- Similarly, you remove an item from the array with the `remove(at:)` method:

```
let mapleSyrup = v.remove(at: 0)
```

- If you want to remove the final item from an array, use the `removeLast()` method:

```
let apples = v.removeLast()
// v now contains 5 items
```

Arrays

- You can iterate over the entire set of values in an array with the for-in loop:

```
for item in v
{
    print(item)
}
```

- If you need the integer index of each item as well as its value, use the `enumerated()` method to iterate over the array instead. For each item in the array, the `enumerated()` method returns a tuple composed of an integer and the item:

```
for (index, value) in v.enumerated()
{
    print("Item \(index + 1): \(value)")
}
```

Sets

- A type must be hashable in order to be stored in a set, i.e. the type must provide a way to compute a hash value for itself.
- A hash value is an `Int` value that is the same for all objects that compare equally, such that if `a == b`, it follows that `a.hashValue == b.hashValue`.
- All of Swift's basic types (such as `String`, `Int`, `Double`, and `Bool`) are hashable by default.
- The type of a Swift set is written as `Set<Element>`, where `Element` is the type that the set is allowed to store. Unlike arrays, sets do not have an equivalent shorthand form:

```
var letters = Set<Character>()
```

- You can add a new item into a set by calling the set's `insert(_:)` method:

```
letters.insert("a")
```

Sets

- You can also initialize a set with an array literal, as a shorthand way to write one or more values as a set collection. A set type cannot be inferred from an array literal alone, so the type Set must be explicitly declared:

```
var s: Set = ["Shoe", "Shirt", "Hat"]
```

- You can use `count` property to find out the number of items in a set, and `isEmpty` property as a shortcut for checking whether the `count` property is equal to 0.
- You can remove an item from a set by calling the set's `remove(_:)` method. Alternatively, all items in a set can be removed with its `removeAll()` method:

```
if let removed = s.remove("Hat") {
   print("I took my \(removed) off.")
} else {
   print("I don't wear hats.")
}
```

Sets

- To check whether a set contains a particular item, use the `contains(_:)` method:

```
if s.contains("Hat") {
   print("I am wearing a hat.")
}
```

- You can iterate over the values in a set with a for-in loop:

```
for fashionItem in s {
   print("\(fashionItem)")
}
```

- Swift’s `Set` type does not have a defined ordering. To iterate over the values of a set in a specific order, use the `sorted()` method:

```
for fashionItem in s.sorted() {
   print("\(genre)")
}
```

Sets

- You can efficiently perform fundamental set operations, such as intersection, union, subtracting and symmetric difference:

```
let odd: Set = [1, 3, 5, 7, 9]
let even: Set = [0, 2, 4, 6, 8]
let prime: Set = [2, 3, 5, 7]
let morePrime: Set = [2, 3, 5, 7, 11, 13, 17]

odd.union(even).sorted()
// [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

odd.intersection(prime).sorted()
// [3, 5, 7]

odd.subtracting(prime).sorted()
// [1, 9]

odd.symmetricDifference(prime).sorted()
// [1, 2, 9]
```

Sets

- Use the "is equal" operator (`==`) to determine whether two sets contain all of the same values.
- Use the `isSubset(of:)` method to determine whether all of the values of a set are contained in the specified set:

```
prime.isSubset(of: morePrime) // true
```

- Use the `isSuperset(of:)` method to determine whether a set contains all of the values in a specified set:

```
morePrime.isSuperset(of: prime) // true
```

- Use `isStrictSubset(of:)` / `isStrictSuperset(of:)` methods to determine whether a set is a subset or superset, but not equal to, a specified set.
- Use the `isDisjoint(with:)` method to determine whether two sets have any values in common:

```
odd.isDisjoint(with: even) // true
```

Dictionaries

- The type of a Swift dictionary is written in full as `Dictionary<Key, Value>`, where `Key` is the type of value that can be used as a dictionary key, and `Value` is the type of value that the dictionary stores for those keys.
- You can also write the type of a dictionary in shorthand form as `[Key: Value]` (preferred).
- As with arrays, you can create an empty Dictionary of a certain type by using initializer syntax:

```
var errorCodes = [Int: String]()
```

- You can also initialize a dictionary with a dictionary literal:

```
var airports = ["DXB": "Dubai",
                "OTP": "Otopeni",
                "LAX": "Los Angeles"]
```

Dictionaries

- You can use `count` property to find out the number of items in a dictionary, and `isEmpty` property as a shortcut for checking whether the `count` property is equal to 0.

- You can add a new item to a dictionary or change the value of an item with subscript syntax:

```
airports["LHR"] = "London Heathrow"
```

- You can use subscript syntax to remove a key-value pair from a dictionary by assigning a value of `nil` for that key:

```
airports["LHR"] = nil
```

Dictionaries

- You can also use subscript syntax to retrieve a value from the dictionary for a particular key. If the dictionary contains a value for the requested key, the subscript returns an optional value containing the existing value for that key. Otherwise, the subscript returns `nil`:

```
if let name = airports["DXB"] {
    print("We are flying to \(name).")
} else {
    print("No tickets for that airport.")
}
// Prints "We are flying to Dubai."
```

- You can iterate over the key-value pairs in a dictionary with a for-in loop:

```
for (code, name) in airports {
    print("\(code): \(name)")
}
```

Dictionaries

- You can also retrieve an iterable collection of a dictionary's keys or values by accessing its `keys` and `values` properties:

```
for code in airports.keys {
    print("Airport code: \(code)")
}

for name in airports.values {
    print("Airport name: \(airportName)")
}
```

- If you need to use a dictionary's keys or values as `Array` instances, initialize a new array with the `keys` or `values` property:

```
let codes = [String](airports.keys)
let names = [String](airports.values)
```

Control Flow

Swift provides a variety of control flow statements:

- **while** loops to perform a task multiple times
- **if**, **guard**, and **switch** statements to execute different code branches based on certain conditions
- **break** and **continue** to transfer the flow of execution to another point in your code
- **for-in** loop that makes it easy to iterate over arrays, dictionaries, ranges, strings, and other sequences

For Loops

- This example prints the first few entries in the five-times table:

```
for index in 1...5 {
   print("\(index) times 5 is \(index * 5)")
}
```

- If you don’t need each value from a sequence, you can ignore the values by using an underscore in place of a variable name:

```
let base = 2
let power = 10
var answer = 1
for _ in 1...power {
   answer *= base
}
print("\(base)^\(power) = \(answer)")
// Prints "2^10 = 1024"
```

While Loops

- A while loop starts by evaluating a single condition. If the condition is true, a set of statements is repeated until the condition becomes false:

```
var i = 2
while i <= 10 {
    print("\(i) is an odd number")
    i += 2
}
```

- The repeat-while loop performs a single pass through the loop block first, before considering the loop's condition. It then continues to repeat the loop until the condition is false:

```
var i = 2
repeat {
    print("\(i) is an odd number")
    i += 2
} while i <= 10
```

Conditional Statements

- You use the if statement to evaluate simple conditions with only a few possible outcomes:

```
temperature = 32
if temperature <= 22 {
   print("It's not yet summer here.")
} else {
   print("Great news! Summer has arrived!")
}
```

- You use a guard statement to require that a condition must be true in order for the code after the guard statement to be executed. Unlike an if statement, a guard statement always has an else clause. The code inside the else clause is executed if the condition is not true:

```
guard let name = person["name"] else {
   return
}
```

Switch Statement

- A switch statement considers a value and compares it against several possible matching patterns. It then executes an appropriate block of code, based on the first pattern that matches successfully:

```
let studentCount = 36
var s: String?
switch studentCount {
   case 0:
      s = "no"
   case 1..<10:
      s = "a few"
   case 10..<20:
      s = "several"
   default:
      s = "many"
}
print("There are \(s!) students here.")
```

Switch Statement

- You can use tuples to test multiple values in the same switch statement. A switch case can use a `where` clause to check for additional conditions:

```
let p = (1, -1)
switch p {
    case let (x, y) where x == y:
        print("\(p) is on the line x == y")
    case let (x, y) where x == -y:
        print("\(p) is on the line x == -y")
    case (_, _):
        print("\(p) is an arbitrary point")
}
// Prints "(1, -1) is on the line x == -y"
```

Functions

- When you define a function, you can optionally define one or more parameters. You can also optionally define a return type. You can use a tuple as the return type for a function to return multiple values:

```
func minMax(x: [Int]) -> (min: Int, max: Int)
{
    var min = x[0]
    var max = x[0]
    for value in x[1..<x.count] {
        if value < min {
            min = value
        }
        if value > max {
            max = value
        }
    }
    return (min, max)
}
let x = minMax(x: [8, -6, 2, 3, 71])
print("min = \(x.min) and max = \(x.max)")
```

Functions

- Each function parameter has both an argument label and a parameter name. The argument label is used when calling the function. By default, parameters use their parameter name as their argument label:

```
func greet(person: String,
               from town: String) -> String
{
    return "Hello, \(person)  from \(town)!"
}

print(greet(person: "John", from: "Boston"))
// Prints "Hello, John from Boston!"
```

Functions

- A variadic parameter accepts zero or more values of a specified type. The values passed to a variadic parameter are made available within the function’s body as an array of the appropriate type:

```
func mean(_ numbers: Double...) -> Double
{
    var total: Double = 0
    for number in numbers
    {
        total += number
    }
    return total / Double(numbers.count)
}

mean(1.0, 2.0, 3.0, 4.0)
```

Functions

- Trying to change the value of a parameter from within the body of the function results in a compile-time error. If you want modify a parameter’s value and persist its value after the function call, define it as an in-out parameter instead:

```
func add(_ amount: Int = 1, to x: inout Int)
{
   x += amount
}

var amount = 3
var x = 7
add(amount, to: &x)
add(to: &x)
print("x is now \(x)")
// Prints "x is now 11"
```

- You can define a default value for any parameter by assigning a value in the declaration. If a default value is defined, you can omit that parameter when calling the function.

Functions Types

- Every function has a specific function type, made up of the parameter types and the return type of the function. The type of these two functions is `(Int, Int) -> Int`:

```
func add(_ a: Int, _ b: Int) -> Int {
   return a + b
}
func multiply(_ a: Int, _ b: Int) -> Int {
   return a * b
}
```

- You can now define a constant or variable to be of a function type and assign an appropriate function to that variable:

```
var mathFunction: (Int, Int) -> Int = add
```

- You can now call the assigned function with the name `mathFunction`:

```
print("Result: \(mathFunction(2, 3))")
```

Nested Functions

- You can define functions inside the bodies of other functions, known as nested functions. You can also use a function type as the return type of another function. An enclosing function can also return one of its nested functions to allow the nested function to be used in another scope:

```
func stepFcn(goBackward:  Bool) -> (Int) ->  Int
{
   func forward(x: Int) ->  Int { return x + 1 }
   func backward(x: Int) -> Int { return x - 1 }
   return goBackward ? backward : forward
}

var x = -4
let moveToZero = stepFcn(goBackward: x > 0)
// moveToZero now refers to the forward() function
while x != 0
{
   print("\(x)...")
   x = moveToZero(x)
}
```

Closures

- Closures are self-contained blocks of functionality that can be passed around and used in your code.
- Closures in Swift are similar to blocks in C and Objective-C.
- Closures can capture and store references to any constants and variables from the context in which they are defined.

Closures

Functions are actually special cases of closures. Closures take one of three forms:

- Global functions are closures that have a name and do not capture any values.
- Nested functions are closures that have a name and can capture values from their enclosing function.
- Closure expressions are unnamed closures written in a lightweight syntax that can capture values from their surrounding context.

Closures

- Swift's standard library provides a method called `sorted(by:)`, which sorts an array of values of a known type, based on the output of a sorting closure that you provide. In the following example, we are sorting an array of `String` values, and so the sorting closure needs to be a function of type `(String, String) -> Bool`:

```
let names = ["Radu", "Alex", "Eva", "Dan"]
```

```
func backward(_ s1: String,
              _ s2: String) -> Bool {

    return s1 > s2
}

var reversed = names.sorted(by: backward)
// reversed is ["Radu", "Eva", "Dan", "Alex"]
```

Closures

- The example below shows a closure expression version of the `backward(_:_:)` function from the previous slide:

```
reversed = names.sorted(by: {
    (s1: String, s2: String) -> Bool in
    return s1 > s2
})
```

- Because the sorting closure is passed as an argument to a method, Swift can infer the types of its parameters and the type of the value it returns. Because all of the types can be inferred, the return arrow (->) and the parentheses around the names of the parameters can also be omitted:

```
reversed = names.sorted(by: { s1, s2 in
    return s1 > s2
})
```

Closures

- Single-expression closures can implicitly return the result of their single expression by omitting the `return` keyword from their declaration, as in this version of the previous example:

```
reversed = names.sorted(by: { s1, s2 in s1 > s2 })
```

- Swift automatically provides shorthand argument names to inline closures, which can be used to refer to the values of the closure's arguments by the names `$0`, `$1`, `$2`, and so on.
- If you use these shorthand argument names within your closure expression, you can omit the closure's argument list from its definition. The `in` keyword can also be omitted:

```
reversed = names.sorted(by: { $0 > $1 })
```

Closures

- If you need to pass a closure expression to a function as the function's final argument and the closure expression is long, it can be useful to write it as a trailing closure instead:

```
func someFunction(closure: () -> Void) {
    // function body goes here
}

// Calling without using a trailing closure:

someFunction(closure: {
    // closure's body goes here
})

// Calling with a trailing closure instead:

someFunction() {
    // trailing closure's body goes here
}
```

Closures

- The string-sorting closure from the previous slides can be written outside of the `sorted(by:)` method's parentheses as a trailing closure:

```
reversed = names.sorted() { $0 > $1 }
```

- If a closure expression is provided as the function or method's only argument and you provide that expression as a trailing closure, you do not need to write a pair of parentheses after the function:

```
reversed = names.sorted { $0 > $1 }
```

Closures

- A closure can capture constants and variables from the surrounding context in which it is defined. The closure can then refer to and modify the values of those constants and variables from within its body, even if the original scope that defined the constants and variables no longer exists:

```
func makeAdder(amount: Int) -> () -> Int {

    var runningTotal = 0
    func adder() ->  Int {
        runningTotal += amount
        return runningTotal
    }
    return adder
}

let addTen = makeAdder(amount: 10)
addTen() // returns a value of 10
addTen() // returns a value of 20
```

Enumerations

- An enumeration defines a common type for a group of related values and enables you to work with those values in a type-safe way within your code

  Enumerations in Swift adopt many features traditionally supported only by classes, such as:
- Computed properties to provide additional information about the enumeration's current value
- Instance methods to provide functionality related to the values the enumeration represents
- Initializers to provide an initial case value
- Can be extended to expand their functionality beyond their original implementation
- Can conform to protocols to provide standard functionality

Enumerations

- You introduce enumerations with the `enum` keyword and place their entire definition within a pair of braces:

```
enum CompassPoint {
    case north
    case south
    case east
    case west
}

var directionToHead = CompassPoint.west
```

- The type of `directionToHead` is inferred when it is initialized with one of the possible values of `CompassPoint`. Once the variable is declared as a `CompassPoint`, you can set it to a different `CompassPoint` value using a shorter dot syntax:

```
directionToHead = .east
```

Enumerations

- When you’re working with enumerations that store integer or string raw values, you don’t have to explicitly assign a raw value for each case. When integers are used for raw values, the implicit value for each case is one more than the previous case. If the first case doesn’t have a value set, its value is 0:

```
enum Planet: Int {
   case mercury = 1, venus, earth, mars,
        jupiter, saturn, uranus, neptune
}

let x = Planet.earth.rawValue
print("Earth is the \(x)rd planet from Sun")
// Prints "Earth is the 3rd planet from Sun"
```

Classes and Structures

Classes and structures in Swift have many things in common. Both can:

- Define properties to store values
- Define methods to provide functionality
- Define subscripts to provide access to their values using subscript syntax
- Define initializers to set up their initial state
- Be extended to expand their functionality beyond a default implementation
- Conform to protocols to provide standard functionality of a certain kind

Classes and Structures

Classes have additional capabilities that structures do not:

- Inheritance enables one class to inherit the characteristics of another
- Type casting enables you to check and interpret the type of a class instance at runtime
- Deinitializers enable an instance of a class to free up any resources it has assigned
- Reference counting allows more than one reference to a class instance

Classes and Structures

- Classes and structures have a similar definition syntax. You introduce classes with the `class` keyword and structures with the `struct` keyword. Both place their entire definition within a pair of braces:

```
struct Resolution
{
    var width = 0
    var height = 0
}

class VideoMode
{
    var resolution = Resolution()
    var interlaced = false
    var frameRate = 0.0
    var name: String?
}
```

Classes and Structures

- The syntax for creating instances is very similar for both structures and classes:

```
let res = Resolution()
let mode = VideoMode()
```

- You can access the properties of an instance using dot syntax:

```
print("The width of res is \(res.width)")
// Prints "The width of res is 0"
```

- You can also use dot syntax to assign a new value to a variable property:

```
mode.resolution.width = 1280
print("Width is \(mode.resolution.width)")
// Prints "Width is 1280"
```

Classes and Structures

- All structures have an automatically-generated memberwise initializer, which you can use to initialize the member properties of new structure instances. Initial values for the properties of the new instance can be passed to the memberwise initializer by name:

```
let hd = Resolution(width: 1280, height: 720)
```

- All structures and enumerations are value types in Swift. This means that any structure and enumeration instances you create are always copied when they are passed around in your code:

```
var cinema = hd
cinema.width = 2048
print("hd is still \(hd.width) pixels wide")
// Prints "hd is still 1280 pixels wide"
```

Classes and Structures

- Classes are reference types. Unlike value types, reference types are not copied when they are assigned to a variable or constant, or when they are passed to a function:

```
let m1 = VideoMode()
m1.resolution = hd
m1.interlaced = true
m1.name = "720i"
m1.frameRate = 25.0

let m2 = m1
m2.frameRate = 30.0

print("Frame rate of m1 is \(m1.frameRate)")
 // Prints "Frame rate of m1 is 30.0"
```

Properties

- Stored properties store constant and variable values as part of an instance.
- Stored properties are provided only by classes and structures.
- Computed properties calculate (rather than store) a value.
- Computed properties are provided by classes, structures, and enumerations.

Stored Properties

- Stored properties can be either variable stored properties or constant stored properties:

```
struct FixedLengthRange {
    var first: Int
    let length: Int
}

var r = FixedLengthRange(first: 0, length: 3)
// r represents values 0, 1, and 2
r.first = 6
// r now represents values 6, 7, and 8
```

- If you declare `r` as a constant (with the `let` keyword), it is not possible to change its variable property:

```
let r = FixedLengthRange(first: 0, length: 3)
r.first = 6 // this will report an error
```

Stored Properties

- If you declare `r` as a constant (with the `let` keyword), it is not possible to change its variable property:

```
let r = FixedLengthRange(first: 0, length: 3)
r.first = 6 // this will report an error
```

- Tis behavior is due to structures being value types. When an instance of a value type is marked as a constant, so are all of its properties.

- The same is not true for classes, which are reference types. If you assign an instance of a reference type to a constant, you can still change that instance’s variable properties.

Lazy Stored Properties

- A lazy stored property is a property whose initial value is not calculated until the first time it is used. You indicate a lazy stored property by writing the `lazy` modifier before its declaration:

```
class DataImporter {

    var fileName = "data.txt"
    // Importing functionality here ...
}

class DataManager {
    lazy var importer = DataImporter()
    var data = [String]()
    // Data management functionality here ...
}

let manager = DataManager()
manager.data.append("Some data")
/* The DataImporter instance for the importer
property has not yet been created. */
```

Computed Properties

```
struct Point {
   var x = 0.0, y = 0.0
}
struct Size {
   var w = 0.0, h = 0.0
}
struct Rect {
   var origin = Point()
   var size = Size()
   var center: Point {
      get {
         let x = origin.x + (size.w / 2)
         let y = origin.y + (size.h / 2)
         return Point(x: x, y: y)
      }
   }
}
var sq = Rect(origin: Point(x: 0.0, y: 0.0),
              size: Size(w: 10.0, h: 10.0))
print("Center is at \(sq.center)")
// Prints "Center is at Point(x:5.0, y:5.0)"
```

Computed Properties

- A computed property with a getter but no setter is known as a **read-only** computed property.
- When declaring a setter, it is not mandatory to define a name for the new value to be set. In this case, `newValue` is used as default name:

```
struct Rect {
    var origin = Point()
    var size = Size()
    var center: Point {
        get {
            let x = origin.x + (size.w / 2)
            let y = origin.y + (size.h / 2)
            return Point(x: x, y: y)
        }
        set {
            origin.x = newValue.x - (size.w / 2)
            origin.y = newValue.y - (size.h / 2)
        }
    }
}
```

Property Observers

- Property observers observe and respond to changes in a property’s value. Property observers are called every time a property’s value is set:

```
class StepCounter {
   var steps: Int = 0 {
      willSet(newSteps) {
         print("Will set steps to \(newSteps)")
      }
      didSet {
         print("Added \(steps – oldValue) steps")
      }
   }
}

let stepCounter = StepCounter()
stepCounter.steps = 50
// Will set steps to 50
// Added 50 steps
```

Type Properties

- You define type properties with the `static` keyword. For computed type properties for class types, you can use the `class` keyword instead to allow subclasses to override the superclass's implementation:

```
class SomeClass {
    static var storedTypeProp = "Some value"
    static var computedTypeProp1: Int {
        return 27
    }
    class var computedTypeProp2: Int {
        return 107
    }
}

print(SomeClass.computedTypeProp1)
// Prints "27"
```

Methods

- Methods are functions that are associated with a particular type.
- Classes, structures, and enumerations can all define instance methods, which encapsulate specific tasks and functionality for working with an instance of a given type.
- Classes, structures, and enumerations can also define type methods, which are associated with the type itself.

Methods

- You use the self property to refer to the current instance within its own instance methods:

```
struct Point {
    var x = 0.0, y = 0.0
    func isToTheRightOf(x: Double) -> Bool {
        return self.x > x
    }
}

let P = Point(x: 4.0, y: 5.0)
if P.isToTheRightOf(x: 1.0) {
    print("P is to the right of x == 1")
}
// Prints "P is to the right of x == 1"
```

- Without the `self` prefix, Swift would assume that both uses of `x` referred to the method parameter called `x`.

Methods

- You can modify Value Types from within instance methods by placing the `mutating` keyword before the `func` keyword for that method:

```
struct Point {
    var x = 0.0, y = 0.0
    mutating func moveBy(x deltaX: Double,
                         y deltaY: Double) {
        x += deltaX
        y += deltaY
    }
}

var P = Point(x: 1.0, y: 1.0)
P.moveBy(x: 2.0, y: 3.0)
print("P is now at (\(P.x), \(P.y))")
// Prints "P is now at (3.0, 4.0)"
```

- Mutating methods can assign an entirely new instance to the implicit `self` property.

Class Methods

- You indicate type methods by writing the `static` keyword before the method's `func` keyword.
- Classes may also use the `class` keyword to allow subclasses to override the superclass's implementation of that method:

```
class SomeClass {
    class func someTypeMethod() {

        print(self)
    }
}

SomeClass.someTypeMethod()
// Prints "SomeClass"
```

Subscripts

- You write subscript definitions with the `subscript` keyword, and specify one or more input parameters and a return type, in the same way as instance methods:

```
struct TimesTable
{
    let multiplier: Int

    subscript(index: Int) -> Int
    {
        return multiplier * index
    }
}

let threeTimesTable = TimesTable(multiplier: 3)
print("6 x 3 = \(threeTimesTable[6])")
// Prints "6 x 3 = 18"
```

Next Time

More Swift:

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

Views:

- View Hierarchy
- View Coordinates