CSC 581: Mobile App Development

Spring 2018

Unit 2: Swift features

- Strings
- functions
- structures
- classes & inheritance
- collections
- loops

Strings

- Swift Strings have many similarities to Python/Java strings

```
var str1 = "foobar"
var str2 = "State \"your name.\""
var cute = "👍😃🌈"                // can contain Unicode symbols

str2 = str1 + str1
str2 += str1

var single = """
    line 1 \
    line2
    """

var multi= """
    line 1
    line2
    """
```

- a single character is inferred to be a String, but can be declared to be a char

```
var ch1 = "q"                  // type(of: ch1) → String.Type

var ch2: Character = "q"       // type(of: ch2) → Character.Type
```

String interpolations & comparisons

- String interpolations make it easy to embed values in Strings

```
var a = 7
var b = 2
print("The sum of \(a) and \(b) is \(a+b)")
```

- comparison operators can be applied to Strings

```
if str1 == str2 {
    print("same")
}

if str1 != "foobar" {
    print("not foobar")
    if str1 < "bar" {
      print("before bar")
    }
    else if str1 >= "foo" {
        print("foo or after")
    }
    else {
        print("between bar and foo")
    }
}
```

3

String properties & methods

```
var str = "Creighton"

str.count             → 9

str.lowercased()      → "creighton"

str.uppercased()      → "CREIGHTON"

str.contains("igh")   → true

str.hasPrefix("Cr")   → true

str.hasSuffix("ton")  → true

for ch in str {                        // iterates over the characters
    print(ch)
}

for index in str.indices {             // iterates over indices, then
    print(str[index])                  // prints each character
}
```

4

Functions

- general form of a Swift function

```
func functionName(param1: Type1, param2: Type2, …) -> ReturnType {
    // BODY OF THE FUNCTION
}
```

e.g.

```
func increment(num: Int) -> Int {
    return num+1
}

func greet(name: String) -> String {
    return "Hello, " + name + "!"
}

func compare(num1: Int, num2: Int) {
    if (num1 == num2) {
        print("same")
    }
    else {
        print("different")
    }
}
```

5

Internal vs. external names

- `low` and `high` are reasonable names within the function, but not so much in the call
- better:

```
var value = sum(from: 1, to: 10)
```

- but then the function looks strange

```
func sum(from: Int, to: Int) -> Int {
    var total = 0;
    for i in from...to {
        total += i
    }
    return total
}
```

- solution: specify two names – an external one for the call, an internal one for the function

```
func sum(from low: Int, to high : Int) -> Int {
    var total = 0;
    for i in low...high {
        total += i
    }
    return total
}
```

6

Swift parameter style

- if different external & internal names read better, specify both

```
display(message: String, in: UILabel)

func display(message msg: String, in label: UILabel) {
    ...
}
```

- if the same name works externally and internally, can list it twice
- better yet, only specify a single name (assumed to be the external name)

```
func display(message message: String, in label: UILabel) { ... }

func display(message: String, in label: UILabel) { ... }
```

- if you really don't want to specify an external name, can use an underscore (but not considered good style)

```
func increment(_ num: Int) -> Int {
    return num+1
}

x = increment(x)
```

7

Default parameters

- you can specify default values for parameters

```
func rollDie(sides: Int = 6) -> Int {
    return Int(arc4random_uniform(UInt32(sides)))+1
}

print(rollDie())              // rolls a 6-sided die,
                              // same as print(rollDie(sides: 6))

print(rollDie(sides: 8))      // rolls an 8-sided die
```

- if a function has multiple parameters, default parameters must be at the end

```
func drawSquare(size side: Int, at x: Int = 0, and y: Int = 0) {
    // draws a square side pixels big, centered at (x, y)
}

drawSquare(side: 50)    // draws a square with 50-pixel sides at (0,0)
                        // same as drawSquare(side: 50, at: 0, and: 0)

drawSquare(side: 50, at: 30)  // draws 50-pixel square at (30, 0)
                        // same as drawSquare(side: 50, at: 30, and: 0)

drawSquare(side: 50, at: 30, and: 40)
                        // draws a 50-pixel square at (30, 40)
```

8

In-class exercise

1. go to your playground, enter the `rollDie` function, and verify that it works

```
func rollDie(sides: Int = 6) -> Int {
    return Int(arc4random_uniform(UInt32(sides)))+1
}
```

2. define a function named `rollDice` that rolls a specified number of dice

```
rollDice(thisMany: 2, withSides: 6)  // returns the sum of 2 6-sided dice

rollDice(thisMany: 4, withSides: 8)  // returns the sum of 4 8-sided dice


rollDice()                 // specify defaults for dice (2) and sides (6)
```

3. define a function named `rollPercentage` that repeatedly rolls a pair of 6-sided dice and returns the percentage of a specified total

```
rollPercentage(of: 7, outOf: 10_000)    // returns the percentage of 7's
                                        // out of 10,000 rolls

rollPercentage(of: 7)                   // specify a default for the number
                                        // of rolls (10,000)
```

9

Defining types

so far, we have written stand-alone functions using primitive data values

- `struct` and `class` are two ways to define a new type
- both can:
  - ✓ encapsulate fields/properties and methods/functions
  - ✓ hide fields/methods by making them private
  - ✓ fields/methods can be class-wide by declaring them `static`
  - ✓ utilize `init` to initialize fields (i.e., a constructor)
- significant differences:
  - ✓ structs are copied whenever you assign or pass as parameters; class objects are shared references (like in Java)
  - ✓ classes can utilize inheritance; structs cannot

common practice:

- structs are used for lightweight data types
  - ✓ fields are usually public, and so directly accessible
  - ✓ methods are for convenience, not for protecting the fields
- classes are used for full-strength, reusable data types
  - ✓ the only option if inheritance/polymorphism is desired

10

Name struct

a struct groups related data values into a single object

- specify the field values when creating a struct object

note: no information hiding or protection

- e.g., can access and change the fields directly

```
struct Name {
    var first: String
    var middle: Character
    var last: String
}


var me = Name(first: "David", middle: "W", last: "Reed")


print("\(me.last), \(me.first) \(me.last)")
                                    → "Reed, David Reed"

me.first = "Dave"

print("\(me.last), \(me.first) \(me.last)")
                                    → "Reed, Dave Reed"
```

11

Struct methods

structs can have methods

- e.g., add a comparison method for names (similar to Java compareTo)

  returns -1 if <
  returns 0 if ==
  returns 1 if >

```
struct Name {
    var first: String
    var middle: Character
    var last: String

    func compare(with other: Name) -> Int {
        if self.last == other.last &&
            self.first == other.first &&
            self.middle == other.middle {
            return 0
        } else if self.last < other.last ||
            (self.last == other.last &&
               self.first < other.first) ||
            (self.last == other.last &&
               self.first == other.first &&
               self.middle < other.middle) {
            return -1
        }
        else {
            return 1
        }
    }
}

var me = Name(first: "David", middle: "W", last: "Reed")
var him= Name(first: "Mark", middle: "V", last: "Reedy")

me.compare(with: him)      → -1
me.compare(with: me)       → 0
him.compare(with: me)      → 1
```

12

Temperature struct

consider a struct for storing a temperature

- might want to specify in C or F
- but only need to store one format

`init` is a constructor for initializing fields

- like Java, can have more than one init
- unlike Java, can even have same parameter types as long as names are different

```
struct Temperature {
    var celsius: Double

    init(inCelsius temp: Double) {
        self.celsius = temp
    }

    init(inFahrenheit temp: Double) {
        self.celsius = (temp - 32)/1.8
    }
}


var current = Temperature(inCelsius: 21.4)

current.celsius                          → 21.4

current = Temperature(inFahrenheit: 32.0)

current.celsius                          → 0.0
```

13

Expanded Temperature

if we also wanted to access the temp in Fahrenheit:

- need another field
- both are initialized in each init

OK, but what if we wanted to add Kelvin?

- would need 3rd field
- each init would have to initialize all three
- TEDIOUS

```
struct Temperature {
    var celsius: Double
    var fahrenheit: Double

    init(inCelsius temp: Double) {
        self.celsius = temp
        self.fahrenheit = 1.8*temp + 32
    }

    init(inFahrenheit temp: Double) {
        self.celsius = (temp - 32)/1.8
        self.fahrenheit = temp
    }
}


var current = Temperature(inCelsius: 0.0)

current.celsius          → 0.0
current.fahrenheit       → 32.0

current = Temperature(inFahrenheit: 212.0)

current.celsius          → 100.0
current.fahrenheit       → 212.0
```

14

Computed properties/fields

alternatively, a *computed field*

- is never assigned a value directly
- its value is automatically computed from other fields

- odd side effect:

```
current.celsius = 0
```
is OK

```
current.fahrenheit = 0
```
is NOT OK

```
struct Temperature {
    var celsius: Double
    var fahrenheit: Double {
        return 1.8*celsius + 32
    }

    init(inCelsius temp: Double) {
        self.celsius = temp
    }

    init(inFahrenheit temp: Double) {
        self.celsius = (temp - 32)/1.8
    }
}


var current = Temperature(inCelsius: 0.0)

current.celsius          → 0.0
current.fahrenheit       → 32.0

current = Temperature(inFahrenheit: 212.0)

current.celsius          → 100.0
current.fahrenheit       → 212.0
```

15

Die struct

BACK TO DICE!

we could define a Die struct to encapsulate #sides and #rolls

- also, provide a mutating roll method

note: no information hiding or protection

- e.g., can change the number of sides

```
struct Die {
    var numSides: Int
    var numRolls: Int

    mutating func roll() -> Int {
        self.numRolls += 1
        return Int(arc4random_uniform(UInt32(self.numSides)))+1
    }
}


var ds = Die(numSides: 6, numRolls: 0)

for _ in 1...10 {
    print(ds.roll())      → displays 10 6-sided rolls
}
ds.numSides               → 6


ds.numSides = 12


for _ in 1...10 {
    print(ds.roll())      → displays 10 12-sided rolls
}
ds.numRolls               → 20
```

16

Private

if we wanted information hiding/protection

- declare the fields to be `private`

but then can't refer to those names when constructing

- must utlize init and provide accessor methods

AT THIS POINT, WE ARE DEALING WITH FULL-STRENGTH OBJECTS

```
struct Die {
    private var numSides: Int
    private var numRolls: Int

    init(sides: Int = 6) {
        self.numSides = sides
        self.numRolls = 0
    }

    mutating func roll() -> Int {
        self.numRolls += 1
        return Int(arc4random_uniform(UInt32(self.numSides)))+1
    }

    func numberOfSides() -> Int {
        return self.numSides
    }

    func numberOfRolls() -> Int {
        return self.numRolls
    }
}


var ds = Die()

for _ in 1...10 {
    print(ds.roll())              // displays 10 6-sided rolls
}

print(ds.numberOfSides())         // displays 6
```

17

Die class

a class definition has similarities to Java class

- fields (a.k.a. properties) should be private
- init serves as a constructor
- can define methods that access and/or change the fields (no mutating label)

```
class Die {
    private var numSides: Int
    private var numRolls: Int

    init(withSides sides: Int = 6) {
        self.numSides = sides
        self.numRolls = 0
    }

    func roll() -> Int {
        self.numRolls += 1
        return Int(arc4random_uniform(UInt32(self.numSides)))+1
    }

    func numberOfSides() -> Int {
        return self.numSides
    }

    func numberOfRolls() -> Int {
        return self.numRolls
    }
}


var d8 = Die(withSides: 8)

for _ in 1...10 {
    print(d8.roll())
}
```

18

Temperature class

we could reimplement Temperature as a class

- could have both fields, but only need one
- could have an accessor for each temperature scale
- or, could combine using a parameter (with a default)

```
class Temp {
    private var celsius: Double

    init(inCelsius temp: Double) {
        self.celsius = temp
    }

    init(inFahrenheit temp: Double) {
        self.celsius = (temp - 32)/1.8
    }

    func get(in scale: String = "Celsius") -> Double {
        let capScale = scale.uppercased()
        if capScale[capScale.startIndex] == "C" {
            return self.celsius
        }
        else if capScale[capScale.startIndex] == "F" {
            return 1.8 * self.celsius + 32
        }
        return -9999
    }
}


var current = Temperature(inCelsius: 0.0)

current.get()                       → 0.0
current.get(in: "C")                → 0.0
current.get(in: "Fahrenheit")       → 32.0
```

19

In-class exercise

1. go to your playground, enter the `Die` class definition, and verify that it works

2. define a function named `rollUntilDoubles` that rolls a pair of dice (default is 6-sided) until doubles are obtained
   - it should display the rolls, return the number of rolls needed

`rollUntilDoubles()` → DISPLAYS: `1-3` `4-2` `6-5` `3-4` `2-2` RETURNS: `5`

`rollUntilDoubles(dieSides: 8)` → DISPLAYS: `7-3` `1-5` `8-8` RETURNS: `3`

20

Inheritance

only classes can utilize inheritance & polymorphism

- specify the parent class in the header of the derived class
- can then add new fields and methods (as in Java)
- if you want to override a field or method, must specify with `override`

```
class Derived: Parent {
    // NEW FIELDS AND METHODS

    override func sameAsParent() {

    }
}
```

let's consider our old friend (from CSC222), `BankAccount`

21

BankAccount

a basic bank account has a balance and an account number

- similar to Java, has a static field to automatically generate unique account numbers
- has accessor methods for seeing balance and account #
- has mutator methods for updating the balance

```
class BankAccount {
    private var balance: Double
    private var accountNumber: Int
    private static var nextNumber = 1

    init() {
        self.balance = 0.0
        self.accountNumber = BankAccount.nextNumber
        BankAccount.nextNumber += 1
    }

    func getBalance() -> Double {
        return self.balance
    }

    func getAccountNumber() -> Int {
        return self.accountNumber
    }

    func deposit(howMuch amount: Double) {
        self.balance += amount
    }

    func withdraw(howMuch amount: Double) {
        self.balance -= amount
    }
}

var acc = BankAccount()
acc.deposit(howMuch: 50.0)
acc.getBalance()                    → 50.0
```

22

SavingsAccount

a savings account inherits fields & methods from the generic bank account

- adds a field for the interest rate
- adds an init with the rate as parameter (and calls super.init)
- adds a method for adding the interest

```
class SavingsAccount: BankAccount {
    private var interestRate: Double

    init(rate: Double) {
        self.interestRate = rate
        super.init()
    }

    func addInterest() {
        var amount = self.getBalance()*self.interestRate/100.0
        self.deposit(howMuch: amount)
    }
}


var sav = SavingsAccount(rate: 5.0)

sav.deposit(howMuch: 50.0)
sav.withdraw(howMuch: 20.0)
sav.getBalance()                    → 30.0

sav.addInterest()
sav.getBalance()                    → 31.5
```

23

CheckingAccount

a checking account inherits fields & methods from the generic bank account

- adds a field for # of transactions field
- overrides init to also initialize the #
- overrides deposit and withdraw to also increment the #
- adds new method for deducting fees

```
class CheckingAccount: BankAccount {
    private static let NUM_FREE = 3
    private static let TRANS_FEE = 2.0
    private var numTransactions: Int

    override init() {
        self.numTransactions = 0
        super.init()
    }

    override func deposit(howMuch amount: Double) {
        self.numTransactions += 1
        super.deposit(howMuch: amount)
    }

    override func withdraw(howMuch amount: Double) {
        self.numTransactions += 1
        super.withdraw(howMuch: amount)
    }

    func deductFees() {
        if self.numTransactions > CheckingAccount.NUM_FREE {
            let fee = CheckingAccount.TRANS_FEE *
           Double(self.numTransactions - CheckingAccount.NUM_FREE)
            super.withdraw(howMuch: fee)
        }
        self.numTransactions = 0
    }
}
```

24

Values vs. reference

structs are value types (as in Python)

- when you assign a *struct* object to a variable or pass it as a parameter, a *copy* of that object is assigned

```
var me = Name(first: "David", middle: "W", last: "Reed")
var copy = me

me.first = "Dave"

me.first        →       "Dave"
copy.first      →       "David"
```

classes are reference types (as in Java)

- when you assign a *class* object to a variable or pass it as a parameter, a *reference* to that object is assigned

```
var die1 = Die(sides: 6)
var die2 = die1

die1.roll()

die1.numberOfRolls()   → 1
die2.numberOfRolls()   → 1
```

25

How do you choose?

the book suggests that struct should be the default, only use class if you need inheritance

- the approach is makes sense if you are new to programming or if you are writing code that is unlikely to be reused

experienced OO programmers understand the advantages of encapsulation + info hiding/protection + polymorphism

- think Model-View-Controller (MVC) pattern
- if you are implementing the model, which has potential for reuse, use class with private fields
- if you are implementing the controller, which is unlikely to be reused, use struct and don't worry about hiding/protection

26

Collections

Swift provides three different collection structs for storing values

- array: an indexed list
- set: a non-index list of unique items (*we will ignore for now*)
- dictionary: a list of key-value pairs

note: all three are homogeneous (items must all be of the same type)

arrays

```
var example: Array<String>        // declares an array of Strings
var example: [String]             // same thing, but cleaner

var empty: Array<String>()        // declares & inits to be empty
var empty: [String]()             // same thing
var empty: [String] = []          // same thing

var words = ["foo", "bar"]        // declares & initializes
```

27

Array properties & methods

```
var words = ["foo", "bar"]

words.count                   → 2

words.append("biz")
words                         → ["foo", "bar", "biz"]

words += ["baz", "boo"]
words                         → ["foo", "bar", "biz", "baz", "boo"]

words[0]                      → "foo"
words[words.count-1]          → "boo"

for str in words {            // displays all five words
    print(str)
}

for i in 0...words.count-1 {  // also displays all five words
    print(words[i])
}

words.remove(at: 2)
words                         → ["foo", "bar", "baz", "boo"]

words.insert("zoo", at: 1)
words                         → ["foo", "zoo", "bar", "baz", "boo"]
```

28

Dictionaries (a.k.a. Maps)

suppose you wanted to store people and their ages

- e.g., Pat is 18, Chris is 20, …
- once stored, could look up a person to get their age

a dictionary is a collection of key:value pairs, can access via the key

```
var ages = ["Pat": 18, "Chris": 20, "Kelly": 19]

ages["Chris"]      → Optional(20)
```

- what is Optional? recall Swift is extremely picky about types
  it is possible that you might try a key that does not exist, e.g., `ages["Sean"]`
  Optional is a wrapper to guard against this – you must check then unwrap it

```
if ages["Chris"] != nil {          // if Optional has a value,
    print(ages["Chris"]!)          //   unwrap (using !) and print
}

if let age = ages["Chris"] {       // nicer notation, does the
    print(age)                     //   same thing
}
```

29

Dictionary properties & methods

```
var example = Dictionary<String, Int>  // declares Dictionary of
var example = [String: Int]            //   String:Int pairs

var example: Dictionary<String: Int>() // declares and initializes as
var example: [String: Int]()           //   empty
var example: [String: Int] = [:]       //


var ages = ["Pat": 18, "Chris": 20, "Kelly": 19]

for (name, age) in ages {
    print("\(name) is \(age) years old.")
}

ages["Pat"] = 19                       // updates "Pat"'s value

ages["Sean"] = 22                      // adds "Sean":22 to end

for name in Array(ages.keys) {         // adds 1 to each age
    if let age = ages[name] {
        ages[name] = age+1
    }
}
```

30

Copy care

arrays and dictionaries, like other built-in Swift types, are structs

- that means that assigning an array/dictionary or passing it as a parameter to a function makes a copy

```
var nums = [3, 6, 1, 7]

var copy = nums

nums += [10, 20, 30]

nums.count                → 7

copy.count                → 4


func zeroOut(numList: [Int]) {
    for index in 0..<numList.count {
        numList[index] = 0
    }
}

zeroOut(numList: nums)

nums                      → [3, 6, 1, 7, 10, 20, 30]
```

31

Loops

traditional while loop, based on a Boolean expression

```
var sum = 1024
while sum > 1 {
    sum /= 2
}
```

for loop can iterate over any countable range

```
for index in 1...10 {             // displays 1 through 10
    print(index)                  // alternatively, ..<
}                                 //   would display 1 through 9

for ch in "abcd" {
    print(ch)
}

var words = ["foo", "bar", "biz"]
for w in words {
    print(w)
}

var ages = ["Pat": 18, "Chris": 20, "Kelly": 19]
for (name, age) in ages {
    print("\(name) is \(age) years old.")
}
```

32

In-class exercise

1. write a function named `average` that takes a list of numbers and returns the average of those numbers

```
var numbers = [3.5, 2, 1.2, 5.3]

average(of: numbers)            → 3
```

2. write a function named `averageLength` that takes a list of Strings and returns the average length of those Strings

```
var strings = ["apple", "banana", "casaba", "date"]

averageLength(of: strings)      → 5.25
```

33