Pázmány Péter Catholic University
Faculty of Information Technology and Bionics

Basics of Mobile Application Development

Swift II.

Multiple branches – revisited

Intervals

```
let approximateCount = 62
let countedThings = "moons orbiting Saturn"
let naturalCount: String
switch approximateCount {
case 0:
    naturalCount = "no"
case 1..<5:
    naturalCount = "a few"
case 5..<12:
    naturalCount = "several"
case 12..<100:
    naturalCount = "dozens of"
case 100..<1000:
    naturalCount = "hundreds of"
default:
    naturalCount = "many"
}
print("There are \(naturalCount) \(countedThings).")
```

Multiple branches – revisited

Tuples

```
let somePoint = (1, 1)
switch somePoint {
case (0, 0):
    print("\(somePoint) is at the origin")
case (_, 0):
    print("\(somePoint) is on the x-axis")
case (0, _):
    print("\(somePoint) is on the y-axis")
case (-2...2, -2...2):
    print("\(somePoint) is inside the box")
default:
    print("\(somePoint) is outside of the box")
}
```

Control Transfer Statements

- Control transfer statements change the order in which your code is executed, by transferring control from one piece of code to another.
- Swift has five control transfer statements:
  - `continue`
  - `break`
  - `fallthrough`
  - `return`
  - `throw`

Continue

- The continue statement tells a loop to stop what it is doing and start again at the beginning of the next iteration through the loop

```
let puzzleInput = "great minds think alike"
var puzzleOutput = ""
let charactersToRemove: [Character] = ["a", "e", "i", "o",
"u", " "]
for character in puzzleInput {
    if charactersToRemove.contains(character) {
        continue
    } else {
        puzzleOutput.append(character)
    }
}
print(puzzleOutput)
```

Break

- Break in a Loop Statement
  - When used inside a loop statement, break ends the loop's execution immediately and transfers control to the code after the loop's closing brace
- Break in a Switch Statement
  - When used inside a switch statement, break causes the switch statement to end its execution immediately and to transfer control to the code after the switch statement's closing brace

Fallthrough

- To enable the C-style fallthrough behavior in a switch

```
let integerToDescribe = 5
var description = "The number \(integerToDescribe) is"
switch integerToDescribe {
case 2, 3, 5, 7, 11, 13, 17, 19:
    description += " a prime number, and also"
    fallthrough
default:
    description += " an integer."
}
print(description)
```

Function parameter argument label

- Each parameter has a name and argument label
  - By default, the argument label is the same as the parameter name

```
func greet(name: String, day: String) -> String {
  return "Hello \(name), today is \(day)."
}
```

  - When a function is invoked the argument label has to be specified as it was seen last week

```
greet(name: "Bob", day: "Tuesday")
```

  - You can also specify you own name

```
func greet(name s1: String, day s2: String) -> String
```

  - It can be invoked the same way

```
greet(name: "Bob", day: "Tuesday")
```

  - You also can omit the label:

```
func greet(_ s1: String, _ s2: String) -> String
```

  - Then

```
greet("Bob", "Tuesday")
```

Enum

- Enum type

```
enum Rank: Int {
    case Ace = 1
    case Two, Three, Four, Five, Six, Seven,
Eight, Nine, Ten
    case Jack, Queen, King
}
let ace = Rank.Ace
let aceRawValue = ace.rawValue
```

Enum in switch

- A function for the previous enum

```
func simpleDescription() -> String {
        switch self {
        case .Ace:
            return "ace”
        case .Jack:
            return "jack”
        case .Queen:
            return "queen”
        case .King:
            return "king”
        default:
            return String(self.rawValue)
        }
}
```

Another example

```
enum Suit {
    case Spades, Hearts, Diamonds, Clubs
    func simpleDescription() -> String {
        switch self {
        case .Spades:
            return "spades"
        case .Hearts:
            return "hearts"
        case .Diamonds:
            return "diamonds"
        case .Clubs:
            return "clubs"
        }
    }
}
```

Further options

- You can add additional values to the instance of an enum

```
enum ServerResponse {
    case Result(String, String)
    case Error(String)
}
let success = ServerResponse.Result("6:00 am", "8:09 pm")
let failure = ServerResponse.Error("Out of cheese.")
switch success {
case let .Result(sunrise, sunset):
    let serverResponse = "Time1 \(sunrise), time2 \(sunset)."
case let .Error(error):
    let serverResponse = "Failure...  \(error)”
}
```

Struct

```
struct Card {
    var rank: Rank
    var suit: Suit
    func simpleDescription() -> String {
        return "The \(rank.simpleDescription()) of
                    \(suit.simpleDescription())"
    }
}
let threeOfSpades = Card(rank: .Three, suit: .Spades)
let threeOfSpadesDescription =
threeOfSpades.simpleDescription()
```

Class and struct

- Both classes and structs can have
  - Properties to store data
    - Get and set functions
  - Methods to implements functions
  - Initializer, to set the initial state
  - Indexer
  - Extensions to add new capabilities to the default functionalities
  - The also can implement (match to) protocols, to provide standard functions
- Classes
  - There is inheritance between classes
  - Type conversion in runtime is possible
  - You can deinitialize classes to free up resources
  - By using reference counting the life cycle of the classes is handled automatically

Differences

- The struct and enum are by value types while the classes are reference types
  - It means that in case of parameter passing the first ones are copied
  - In case of the latter one, the reference is copied thus the instance is not duplicated

- The built-in String, Array and Dictionary are implemented as structs

Properties – fields

Example

```
struct FixedLengthRange {
    var firstValue: Int
    let length: Int
}
var rangeOfThreeItems =
FixedLengthRange(firstValue: 0, length: 3)
rangeOfThreeItems.firstValue = 6
!!! Error
let rangeOfThreeItems =
FixedLengthRange(firstValue: 0, length: 3)
rangeOfThreeItems.firstValue = 6
```

Properties – fields

Calculated, manipulated value

```
struct Rect {
    var origin = Point()
    var size = Size()
    var center: Point {
        get {
            let centerX = origin.x + (size.width / 2)
            let centerY = origin.y + (size.height / 2)
            return Point(x: centerX, y: centerY)
        }
        set(newCenter) {
            origin.x = newCenter.x - (size.width / 2)
            origin.y = newCenter.y - (size.height / 2)
        }
    }
}

struct Point {
    var x = 0.0, y = 0.0
}
struct Size {
    var width = 0.0, height = 0.0
}
```

Observing a property

```
class StepCounter {
    var totalSteps: Int = 0 {
        willSet(newTotalSteps) {
            print("About to set totalSteps to \(newTotalSteps)")
        }
        didSet {
            if totalSteps > oldValue  {
                print("Added \(totalSteps - oldValue) steps")
            }
        }
    }
}
```

Properties – fields

- Class variables are defined with static keyword
- If there is no set function defined then the property is read only

```
class Counter {
      static var maxCount = 10
      var count = 0
      func incrementCount() {
            count += 1
      }
      static func incrementMaxCount(value : Int) {
            maxCount += value
      }
}
Counter.incrementMaxCount(value: 5) // 15
```

Changing the state of a class

```
class Counter {
    var count = 0
    func increment() {
        count += 1
    }
    func incrementBy(amount: Int) {
        count += amount
    }
    func reset() {
        count = 0
    }
}
```

Changing the state of a struct

```
struct Counter {
    var count = 0
    mutating func increment() {
        count += 1
    }
}
```

- In case of struct the self variable is immutable

Initialization – Example

```
struct Color {
    let red, green, blue: Double
    init(red: Double, green: Double, blue: Double) {
        self.red   = red
        self.green = green
        self.blue  = blue
    }
    init(white: Double) {
        red   = white
        green = white
        blue  = white
    }
}
```

Inheritance, overriding

```
class Counter {
    var count = 0
    func incrementCount() {
        count += 1
    }
}
class ChildCounter : Counter{
    override func incrementCount() {
        count += 2
    }
}
var c = ChildCounter()
c.incrementCount()
print(c.count)
```

Inheritance, overriding

- Overriding a class function (observe the class keyword)

```
class Counter {
      static var maxCount = 10
      var count = 0
      func incrementCount() {
            ++count
      }
      class func incrementMaxCount(value : Int) {
                  maxCount += value
      }
}
Counter.incrementMaxCount(5) // 15
```

Inheritance, overriding

- Overriding a class function (observe the class keyword)

```
Counter.incrementMaxCount(5) // 15

class SuperCounter : Counter {
    override class
      func incrementMaxCount(value : Int) {
        maxCount += 2*value
    }
}

SuperCounter.incrementMaxCount(5) // 25
Counter.maxCount
```

Preventing the overriding

- Obviously
  - `final func`
  - `final class func`
  - `final var`
  - `final subscript`
- And
  - `final class`
- Without these modifiers, you can override
  - Functions
  - Properties
  - Get and Set functions
  - Indexers

Extension – protocol

- Adding a new function to any struct, enum
  - Possibilities
    - New calculated (derived) property
    - New class/instance function
    - New initializer
    - New indexer
    - New nested type
    - Existing type can be prepared to implement a new protocol
  - Keyword: `extension`
- Protocol
  - It it the well-known conception of interface

Examples

```
protocol Animal {
    func name() -> String
}
class Dog : Animal {
    func name() -> String {
        return "Jack"
    }
}
let favorite : Animal = Dog()
favourite.name() // "Jack"
```

Examples

```
class Greyhound : Dog {
    override func name() -> String {
        return "Bruno"
    }
}
let favourite2 : Animal = Greyhound()
favourite2.name() // "Bruno"
```

Extension of a protocol

```
extension Animal {
    func doubleName() -> String {
        return name() + " " + name()
    }
}
let favourite2 : Greyhound = Greyhound()
favourite2.doubleName() // "Bruno Bruno"
favourite.doubleName() // "Jack Jack"
```

Controlling the access

- The visibility of the members can ben controlled with the common keywords
  - `public` – the public interface of the module, can be accessed from anywhere)
  - `internal` – can be accessed anywhere in the module (public in the module for interoperability)
  - `private` – can be accessed in the environment where it is defined

Serizalization of optionals

```
class One {
    var number: Int?
}
class Simple {
    var oneOrNot: One?
}
let optAndSimple: Simple? = Simple()
optAndSimple!.oneOrNot = One()
optAndSimple!.oneOrNot!.number = 5

if let c = optAndSimple?.oneOrNot?.number {
    print("The number is \(c)")
} else {
    print("No number")
}
```

Homework – Deadline 10/22/2019 10.15 am

- You have to create a demonstration of SWIFT object oriented capabilities
- Details
  - Create a class to represent any cards
    - Content, initialization
    - Comparison
  - Create a class to represent playing cards
    - Suit, rank
    - Use inheritance
    - Use enum
  - Create a deck for cards and playing cards as well
- Test your code

Objective-C

Next week