CSC 471 / 371 Mobile Application Development for iOS

Prof. Xiaoping Jia
School of Computing, CDM
DePaul University
xjia@cdm.depaul.edu
@DePaulSWEng

Swift Primer, Part 4 Collections, Closures, and Enums

Outline

- Collections
  - Arrays
  - Dictionaries
  - Sets
- Closures
- Enumeration types
- The Foundation framework

DePaul University 3

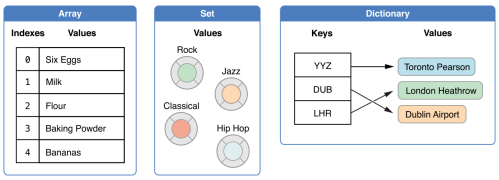
Collections

Swift Collection

- Swift provides three *collection types* for storing collections of values
  - Array – *ordered* list of values of the same type
  - Dictionaries – *unordered* collection of *key-value* pairs (of the same types)
  - Set – *unordered* collection of *unique* values of the same type
- Collections are typed
  - Elements are of the same type
- Collections are *value* types

DePaul University 5

Swift Collections



DePaul University 6

Mutability of Collections

- Collections have *immutable* and *mutable* versions
  - Declared with `let` – constant, immutable
  - Declared with `var` – variable, mutable
- Immutable collections
  - Prevents unexpected changes
  - Can be optimized by compiler
  - **Good practice**: use immutable collections when possible
- Mutable collections
  - Typically carry a performance overhead

Arrays

- Array type with element type *T*
  - `Array[T]`, or shorthand `[T]`
  - e.g., `Array[Int]`, or `[Int]`
- Arrays are typed. All values are of the same type.
  - Different from `NSArray` and `NSMutableArray`, which are untyped
- Array literals
  - [$value_1$, $value_2$, …]
  - Empty array: `[]`, or `[Int]()`

Using Arrays

```
let months = [
    "January", "February", "March",
    "April", "May", "June",
    "July", "August", "September",
    "October", "November", "December"
]

print("Month Name");
print("===== ====");
for var i = 0; i < months.count; ++i {
    let str = String(format: "%2i", i+1)
    print(" \(str)   \(months[i])")
}
```

The output:

```
Month Name
===== ====
  1   January
  2   February
  3   March
  4   April
  5   May
  6   June
  7   July
  8   August
  9   September
 10   October
 11   November
 12   December
```


Using Arrays, with Tuple

```
let months = [
    "January", "February", "March",
    "April", "May", "June",
    "July", "August", "September",
    "October", "November", "December"
]

print("Month Name");
print("===== ====");
for (i, name) in months.enumerate() {
    let str = String(format: "%2i", i+1)
    print(" \(str)   \(name)")
}
```

A tuple

The output:

```
Month Name
===== ====
  1   January
  2   February
  3   March
  4   April
  5   May
  6   June
  7   July
  8   August
  9   September
 10   October
 11   November
 12   December
```


Modifying Arrays – A Shopping List

```
var shoppingList = [ "Eggs", "Milk" ]
```

The shopping list:

```
[ "Eggs", "Milk" ]
```


Modifying Arrays – A Shopping List

```
var shoppingList = [ "Eggs", "Milk" ]
shoppingList += [ "Flour" ]
```

Concatenate another array

The shopping list:

```
[ "Eggs", "Milk", "Flour" ]
```

Modifying Arrays – A Shopping List

```
var shoppingList = [ "Eggs", "Milk" ]
shoppingList += [ "Flour" ]
shoppingList += [ "Gruyère Cheese", "Butter" ]
```

Concatenate another array

The shopping list:
["Eggs", "Milk", "Flour", "Gruyère Cheese", "Butter"]

Modifying Arrays – A Shopping List

```
var shoppingList = [ "Eggs", "Milk" ]
shoppingList += [ "Flour" ]
shoppingList += [ "Gruyère Cheese", "Butter" ]
shoppingList[0] = "Beef broth"
```

Replace an element

The shopping list:
["Beef broth", "Milk", "Flour", "Gruyère Cheese", "Butter"]

Modifying Arrays – A Shopping List

```
var shoppingList = [ "Eggs", "Milk" ]
shoppingList += [ "Flour" ]
shoppingList += [ "Gruyère Cheese", "Butter" ]
shoppingList[0] = "Beef broth"
shoppingList[1...2] = [ "Onion", "Bay leaves", "Baguette" ]
```

Replace a section

The shopping list:
["Beef broth", "Onion", "Bay leaves", "Baguette", "Gruyère Cheese", "Butter"]

Modifying Arrays – Append

```
var array: [Int] = []
for i in 0 ... 10 {
    print(i)
    array.append(i)
}
array += [ 11, 12 , 13]
```

Append a single element

Concatenate another array

The output:
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13]

Modifying Arrays – Insert

```
var array: [Int] = []
for i in 0 ... 10 {
    print(i)
    array.append(i)
}
array += [ 11, 12 , 13]
array[0] = 100
array.insert(200, atIndex: 11)
array.insert(201, atIndex: 12)
```

Insert an element at the specified position

The output:
[100, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 200, 201, 11, 12, 13]

Modifying Arrays – Section

```
var array: [Int] = []
for i in 0 ... 10 {
    print(i)
    array.append(i)
}
array += [ 11, 12 , 13]
array[0] = 100
array.insert(200, atIndex: 11)
array.insert(201, atIndex: 12)
array[1...9]
```

A section of an array

The output:
[1, 2, 3, 4, 5, 6, 7, 8, 9]

Modifying Arrays – Replace

```
var array: [Int] = []
for i in 0 ... 10 {
    print(i)
    array.append(i)
}
array += [ 11, 12 , 13]
array[0] = 100
array.insert(200, atIndex: 11)
array.insert(201, atIndex: 12)
array[1...9]
array[3...5] = [33, 44, 55]
```

Replace a section

The output:

```
[100, 1, 2, 33, 44, 55, 6, 7, 8, 9, 10, 200, 201, 11, 12, 13]
```


Modifying Arrays – Remove

```
var array: [Int] = []
for i in 0 ... 10 {
    print(i)
    array.append(i)
}
array += [ 11, 12 , 13]
array[0] = 100
array.insert(200, atIndex: 11)
array.insert(201, atIndex: 12)
array[1...9]
array[3...5] = [33, 44, 55]
array[11...14] = []
```

Remove a section

The output:

```
[100, 1, 2, 33, 44, 55, 6, 7, 8, 9, 10, 13]
```


Modifying Arrays – Remove

```
var array: [Int] = []
for i in 0 ... 10 {
    print(i)
    array.append(i)
}
array += [ 11, 12 , 13]
array[0] = 100
array.insert(200, atIndex: 11)
array.insert(201, atIndex: 12)
array[1...9]
array[3...5] = [33, 44, 55]
array[11...14] = []
array.removeAtIndex(5)
```

Remove the element at the specified position

The output:

```
[100, 1, 2, 33, 44, 6, 7, 8, 9, 10, 13]
```


Dictionaries

- Dictionary type with key type *K*, and value type *V*
  - `Dictionary[K, V]`, or shorthand `[K, V]`
  - e.g., `Dictionary[String,Int]`, or `[String,Int]`
- Dictionaries are typed. All keys are of the same type and all values are of the same type.
  - Different from `NSDictionary` and `NSMutableDictionary`, which are untyped
- Dictionary literals
  - [key_1 : $value_1$, key_2 : $value_2$, …]
  - Empty dictionary: `[:]`, or `[String,Int]()`

Using Dictionaries

```
let numbers = [
    "zero": "zéro",   "one": "un",
    "two": "deux",    "three": "trois",
    "four": "quatre", "five": "cinq",
    "six": "six",     "seven": "sept",
    "eight": "huit",  "nine": "neuf",
    "ten": "dix" ]

for (key, value) in numbers {
    print("English: \(key) \tFrench: \(value)")
}
```


Using Dictionaries

```
let numbers = [
    "zero": "zéro",   "one
    "two": "deux",    "thr
    "four": "quatre", "fiv
    "six": "six",     "sev
    "eight": "huit",  "nir
    "ten": "dix" ]

for (key, value) in number
    print("English: \(key)
}
```

The output:

```
English: eight  French: huit
English: one    French: un
English: three  French: trois
English: seven  French: sept
English: nine   French: neuf
English: six    French: six
English: ten    French: dix
English: zero   French: zéro
English: five   French: cinq
English: four   French: quatre
English: two    French: deux
```

Using Dictionaries

```
var dictionary = [
    "zéro" : 0,
    "un" : 1,
    "deux" : 2,
    "trois" : 3,
]

dictionary.count
var value = dictionary["deux"]
```

The output:

```
4
2
```

An optional value

Using Dictionaries

```
var dictionary = [
    "zéro" : 0,
    "un" : 1,
    "deux" : 2,
    "trois" : 3,
]

dictionary.count
var value = dictionary["deux"]

if let value = dictionary["deux"]  {
    print("The value of duex is \(value)")
}
```

The output:

```
4
2
"The value of duex is 2"
```


Modifying Dictionaries – Animal Legs

```
var numberOfLegs = [ "ant": 6, "snake": 0, "cheetah": 4 ]

for (animalName, legCount) in numberOfLegs {
    print("\(animalName)s have \(legCount) legs")
}

numberOfLegs["spider"] = 273

numberOfLegs["spider"] = 8

numberOfLegs["cheetah"] = nil
```

A new entry

Update an existing entry

Remove an existing entry

```
[ "ant": 6,
  "snake": 0,
  "spider": 8 ]
```


Sets

- Unordered collections of unique values
- Set type with element type *T*
  - `Set[T]`, e.g., `Set[Int]`
- Sets are typed. All values are of the same type.
- Set literals
  - [$value_1$, $value_2$, …]
  - Empty set: `[] as Set`, or `Set[Int]()`

Using Sets

```
var genres: Set<String> = ["Rock", "Classical", "Hip hop"]
```

A new set

Using Sets

```
var genres: Set = ["Rock", "Classical", "Hip hop"]
```

A new set, just the same

Using Sets

```
var genres: Set = ["Rock", "Classical", "Hip hop"]

var cities = [ "Beijing", "London", "Rio de Janeiro" ] as Set
cities.sort()
```

A sorted array

Another new set

Using Sets

```
var genres: Set = ["Rock", "Classical", "Hip hop"]

var cities = [ "Beijing", "London", "Rio de Janeiro" ] as Set
cities.sort()

var newSports = Set<String>()
newSports.insert("Golf")
newSports.insert("Rugby seven")
print(newSports)
```

A new empty set

Insert elements to a set

Using Sets

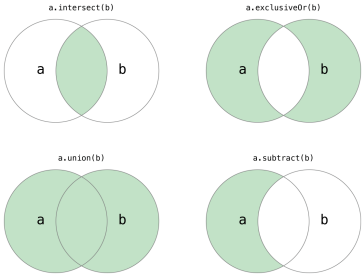
```
var genres: Set<String> = ["Rock", "Classical", "Hip hop"]
var genres: Set = ["Rock", "Classical", "Hip hop"]

var cities = [ "Beijing", "London", "Rio de Janeiro" ] as Set
cities.sort()

var newSports = Set<String>()
newSports.insert("Golf")
newSports.insert("Rugby seven")
print(newSports)
for s in newSports {
    print("\(s) has been added to Olympic sports")
}
```

Iterate through a set

Set Operations



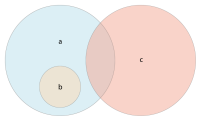
Set Operations

```
let oddNums: Set = [1, 3, 5, 7, 9]
let evenNums: Set = [0, 2, 4, 6, 8]
let smallPrimes: Set = [2, 3, 5, 7]

oddNums.union(evenNums).sort()
// [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
oddNums.intersect(evenNums).sort()
// []
oddNums.subtract(smallPrimes).sort()
// [1, 9]
oddNums.exclusiveOr(smallPrimes).sort()
// [1, 2, 9]
```


Set Relationships

- Set a is a *superset* of set b, if a contains all elements in b.
- Set b is a *subset* of set a, if all elements in b are also contained by a.
- Set b and set c are *disjoint* with one another, if they share no elements in common.



Set Relationships

- The "is equal" operator (==)
  - determine whether two sets contain all of the same values.
- `isSubsetOf(_:)`
  - determine whether all of the values of a set are contained in the specified set.
- `isSupersetOf(_:)`
  - determine whether a set contains all of the values in a specified set.
- `isStrictSubsetOf(_:)` or `isStrictSupersetOf(_:)`
  - determine whether a set is a subset or superset, but not equal to, a specified set.
- `isDisjointWith(_:)`
  - determine whether two sets have any values in common.

Set Relationships

```
let houseAnimals: Set = ["🐶", "🐱"]
let farmAnimals: Set = ["🐮", "🐔", "🐑", "🐶", "🐱"]
let cityAnimals: Set = ["🐦", "🐭"]

houseAnimals.isSubsetOf(farmAnimals)

farmAnimals.isSupersetOf(houseAnimals)

farmAnimals.isDisjointWith(cityAnimals)
```


Functions & Closures

Closure

- *Closures* are self-contained blocks of code that can be called later.
  - Closures are name-less functions
  - Functions are simply named closures
- Closure use simple familiar syntax
- Closures and functions are first class
  - Like objects

Closure Syntax

```
{ ( Parameters ) -> Return Type in
    Statements
}
```

Closure body.

- Parameters and return types use the same syntax as in functions
- Types can be omitted if they can be inferred from the context
- If the body contains a single statement

  `return` *Expression*

  The `return` keyword is optional

Functions and Closures

```
func hello(name: String) {
    print("Hello, \(name)")
}
hello("Function")
```

A simple function and an invocation.

```
Hello, Function
```

```
let c1 = { (name: String) in
    print("Hello, \(name)")
}
c1("Closure")
```

The same function as a closure, and an invocation of the closure.

```
Hello, Closure
```

```
var c2 = c1
c2("Closure")
c2 = hello
c2("Closure")
```

Closures are objects.

```
Hello, Closure
```

Functions are closures too.

Concise Closure Syntax

```
func threeTimes(n: Int) -> Int {
    return n * 3
}
threeTimes(4)

let c3 = { (n: Int) -> Int in
    return n * 3
}
c3(5)

let c4 = { n in n * 3 }
c4(6)
```

A function

12

A closure.

15

The same closure.

18

More Closure Examples

- The `sort` method

collection`.sort(`*order*`)`

A closure compares two elements. Return true if the first is before the second

```
let cities = [ "Barcelona", "Atlanta", "Athens",
    "Sydney", "Beijing", "London", "Rio de Janeiro" ]

cities.sort({(a: String, b: String) -> Bool in
    return a < b
})
```

["Athens", "Atlanta", "Barcelona", "Beijing", "London", "Rio de Janeiro", "Sydney"]

```
cities.sort({(a: String, b: String) -> Bool in
    return a > b
})
```

["Sydney", "Rio de Janeiro", "London", "Beijing", "Barcelona", "Atlanta", "Athens"]

More Closure Examples

```
cities.sort({(a: String, b: String) -> Bool in
    return a.characters.count < b.characters.count
})
```

["Athens", "Sydney", "London", "Atlanta", "Beijing", "Barcelona", "Rio de Janeiro"]

Trailing closure.

```
cities.sort {(a: String, b: String) -> Bool in
    return a.characters.count < b.characters.count
}
```

["Athens", "Sydney", "London", "Atlanta", "Beijing", "Barcelona", "Rio de Janeiro"]

More Closure Examples

```
cities.sort({ a, b in a < b })
```

Shortened syntax.

```
cities.sort({ $0 < $1 })
```

Shortened argument names.

```
cities.sort() { $0 < $1 }
```

Trailing closure.

```
cities.sort { $0 < $1 }
```

```
cities.sort(<)
```

Function as a closure.

["Athens", "Sydney", "London", "Atlanta", "Beijing", "Barcelona", "Rio de Janeiro"]

Enumeration Types

Enum Types

- A type consists of a group of related values
  - Type safe. Not the same as Int.

```
enum Planet {
    case Mercury, Venus, Earth, Mars, Jupiter,
         Saturn, Uranus, Neptune, Planet_9
}

var planet = Planet.Earth
```

- If the enum type can be inferred from the context, the type name can be omitted

```
planet = .Venus
```

Because we know the type of `planet` is `Planet`

The Raw Value of Enum

- A enum type may declare a *raw value type*
- It assigns a *raw value* to each member.
- The raw value types can be any integer or float-point types, String, or Character

Raw value type

```
enum Direction : Int {
    case North
    case South
    case East
    case West
}

var dir : Direction
dir = .East
dir.rawValue
```

Raw values: 0, 1, 2, 3

The output:
```
East
2
```

DePaul University 49

Enum Types with Functions

- Enum types can define functions and properties
- Like structs

```
dir = .North
dir.rawValue
dir.name()
```

The output:
```
2
"North"
```

```
enum Direction : Int {
    case North
    case South
    case East
    case West
    func name() -> String {
        switch self {
        case .North: return "North"
        case .South: return "South"
        case .East: return "East"
        case .West: return "West"
        }
    }
}
```

DePaul University 50

Enum Types with Properties

- An enum with a computed property

```
enum City: String {
    case Barcelona = "Barcelona"
    case Atlanta = "Atlanta"
    case Sydney = "Sydney"
    case Athens = "Athens"
    case Beijing = "Beijing"
    case London = "London"
    case Rio_de_Janeiro = "Rio de Janeiro"
    case Tokyo = "Tokyo"
    var name: String { return self.rawValue }
}
```

DePaul University 51

Enum Types with Properties

```
let hostCityYear: [City: Int] = [
    .Barcelona: 1992,
    .Atlanta: 1996,
    .Sydney: 2000,
    .Athens: 2004,
    .Beijing: 2008,
    .London : 2012,
    .Rio_de_Janeiro: 2016,
    .Tokyo: 2020
]

for (c, _) in hostCityYear {
    print("City:", c.rawValue)
}
```

The output:
```
City: Sydney
City: Beijing
City: London
City: Rio de Janeiro
City: Tokyo
City: Barcelona
City: Atlanta
City: Athens
```

DePaul University 52

Enum Types with Properties

```
for (city, year) in hostCityYear {
    print(city.name,
          year < 2016 ? "hosted" : "will host",
          "the Olympic Games in \(year)")
}
```

The output:
```
Sydney hosted the Olympic Games in 2000
Beijing hosted the Olympic Games in 2008
London hosted the Olympic Games in 2012
Rio de Janeiro will host the Olympic Games in 2016
Tokyo will host the Olympic Games in 2020
Barcelona hosted the Olympic Games in 1992
Atlanta hosted the Olympic Games in 1996
Athens hosted the Olympic Games in 2004
```

DePaul University 53

Enum Types with Properties

```
for (city, year) in hostCityYear.sort({
    (cy1: (City, Int), cy2: (City, Int)) -> Bool in
          cy1.1 < cy2.1 }) {
    print(city.name,
          year < 2016 ? "hosted" : "will host",
          "the Olympic Games in \(year)")
}
```

The output:
```
Barcelona hosted the Olympic Games in 1992
Atlanta hosted the Olympic Games in 1996
Sydney hosted the Olympic Games in 2000
Athens hosted the Olympic Games in 2004
Beijing hosted the Olympic Games in 2008
London hosted the Olympic Games in 2012
Rio de Janeiro will host the Olympic Games in 2016
Tokyo will host the Olympic Games in 2020
```

DePaul University 54

Value Types vs. Reference Types

- Enums are *values* types
- Value types
  - Basic types: `Int, Flaot, Double, Character,` etc.,
  - `String, Array,` and `Dictionary`
  - Structs and enums
  - Use == to compare contents, i.e., values
- Reference types
  - Classes, closures
  - Use == to compare contents
  - Use === to compare identities

Foundation Framework

Foundation Framework

- A base layer of Objective-C classes
  - A root class: `NSObject`
  - Utility classes:
    - `NSString NSDate NSTimeZone NSCalendar`
- Available (bridged) in Swift
  - Direct import
    - `import Foundation`
  - Indirect import
    - `import UIKit`
    - `import Cocoa`

Math Constants and Functions

- Constants

```
let π = M_PI
let e = M_E
let log2e = M_LOG2E
let sqrt2 = M_SQRT2
```

- Functions

```
var square = 9.4
var f = floor(square)
var c = ceil(square)
var root = sqrt(f)
var p = pow(2.5, 3)
```

- Trigonometry

```
sin(π / 3)
cos(π / 3)
tan(0.6)
atan2(2.5, 4.8)
```

- Random numbers

```
let N : UInt32 = 100

arc4random() % N
arc4random() % N
arc4random() % N
```


Swift String and NSString

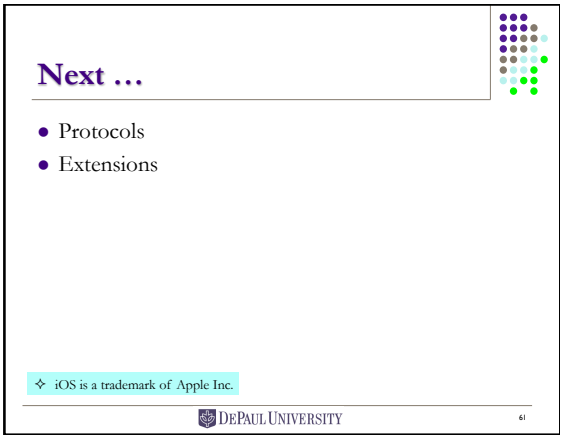
- Not the same class.
  - Both available in Swift
  - But *not compatible*
    - `NSString` assumes fixed width characters, i.e., supports a subset of Unicode
- **If the Foundation framework is imported**
  - Some methods of `NSString` class are available in `String`
  - The `String` class is *extended*

```
let s0 = String(format: "  %.2f", π)
```


NSObject, AnyObject, Any

- All are *types* in Swift
- `NSObject`
  - A root class.
  - A super type of all classes defined in Objective-C, including classes in Foundation, and UIKit
- `AnyObject`
  - A super type of all classes in Swift, including `NSObject`
  - Not a class
- `Any`
  - A super type of all types in Swift, including structs, and enums

Then what is it?
Stay tuned


Next …
Protocols
Extensions
iOS is a trademark of Apple Inc.
DePaul University
61