CSC 471 / 371 Mobile Application Development for iOS

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

1

Swift Primer, Part 4 Collections, Closures, and Enums

2

Outline

- Collections
  - Arrays
  - Dictionaries
- Closures
- Enumeration types

3

Collections

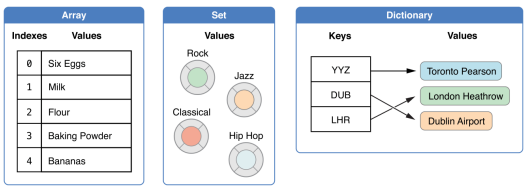
4

Swift Collection

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

5

Swift Collections



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

DePaul University 7

7

Arrays

- Array 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]()`

DePaul University 8

8

Using Arrays – Count and Index

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

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

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
```

DePaul University 9

10

Using Arrays – Start and End Index

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

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

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
```

DePaul University 10

12

Using Arrays – Iteration

```
let months = [
    "January", "February", "March",
    "April", "May", "June",
    "July", "August", "September",
    "October", "November", "December"
]
print("Month Name")
print("===== ====")
var i = 1
for name in months {
    let str = String(format: "%2i", i)
     print(" \(str)   \(name)")
     i += 1
}
```

Iterate through each element in the array

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
```

DePaul University 11

14

Using Arrays – Iteration with Tuples

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

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

A collection pairs of (index, value)

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
```

DePaul University 12

17

Modifying Arrays – A Shopping List

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

The shopping list:
["Eggs", "Milk"]

DePaul University 13

18

Modifying Arrays – A Shopping List

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

Concatenate another array

The shopping list:
["Eggs", "Milk", "Flour"]

DePaul University 14

19

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"]

DePaul University 15

20

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"]

DePaul University 16

21

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"]

DePaul University 17

22

Modifying Arrays – Append

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

Append a single element

Concatenate another array

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

DePaul University 18

23

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, at: 11)
array.insert(201, at: 12)
print(array)
```

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]
```

DePaul University 19

24

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, at: 11)
array.insert(201, at: 12)
array[1...9]
```

A section of an array

The output:

```
[1, 2, 3, 4, 5, 6, 7, 8, 9]
```

DePaul University 20

25

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, at: 11)
array.insert(201, at: 12)
array[1...9]
array[3...5] = [33, 44, 55]
print(array)
```

Replace a section

The output:

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

DePaul University 21

26

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, at: 11)
array.insert(201, at: 12)
array[1...9]
array[3...5] = [33, 44, 55]
array[11...14] = []
print(array)
```

Remove a section

The output:

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

DePaul University 22

27

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, at: 11)
array.insert(201, at: 12)
array[1...9]
array[3...5] = [33, 44, 55]
array[11...14] = []
array.remove(at: 5)
print(array)
```

Remove the element at the specified position

The output:

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

DePaul University 23

28

Array Slices with Open-Ended Ranges

```
array
```

The output:

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

```
array[0...4]
array[...4]
array[..<4]
array[9...]
```

The output:

```
[100, 1, 2, 33, 44]
[100, 1, 2, 33, 44]
[100, 1, 2, 33]
[10, 13]
```

DePaul University 24

29

Dictionaries

- Dictionary 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]()`

30

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)")
}
```


31

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
```


32

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

34

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"
```


35

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 ]
```


39

Functions & Closures

40

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 citizens
  - Like objects

DePaul University 32

41

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

DePaul University 33

42

Functions and Closures

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

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

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

A simple function and an invocation.

```
Hello, Function
```

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

```
Hello, Closure
```

Closures are objects.

```
Hello, Closure
```

Functions are closures too.

DePaul University 34

49

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
```

DePaul University 35

55

More Closure Examples

- The `sorted` method

  collection`.sorted(by:` *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.sorted(by: {(a: String, b: String) -> Bool in
    return a < b
})
```

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

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

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

DePaul University 36

60

More Closure Examples

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

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

Trailing closure.

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

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


66

More Closure Examples

```
cities.sorted(by: { a, b in a < b })
```

Shortened syntax.

```
cities.sorted(by: { $0 < $1 })
```

Shortened argument names.

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

Trailing closure.

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

cities.sorted(by: <)
```

Function as a closure.

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


71

Sort vs. Sorted

- **Sorted**: *adjective*. a state
- **Sort**: *verb*. an action

```
let cities = [ "Barcelona", "Atlanta", "Athens",
    "Sydney", "Beijing", "London", "Rio de Janeiro" ]
cities.sorted()
cities.sorted(by: <)

var cities2 = cities
cities2.sort()
```

Constant

Copies of the array in sorted order

Variable

In-place sort. No copy.

```
["Athens", "Atlanta", "Barcelona", "Beijing",
"London", "Rio de Janeiro", "Sydney"]
cities2.sort(by: >=)
```

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


73

Enumeration Types

74

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`

75

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
```


76

Enum Types with Functions

- Enum types can define functions and properties
- Like structs

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

The output:
```
0
"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 43

77

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 44

78

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 45

79

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 hosted 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 46

80

Enum Types with Properties

```
for (city, year) in hostCityYear.sorted(by: {
    (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 hosted the Olympic Games in 2016
Tokyo will host the Olympic Games in 2020
```

DePaul University 47

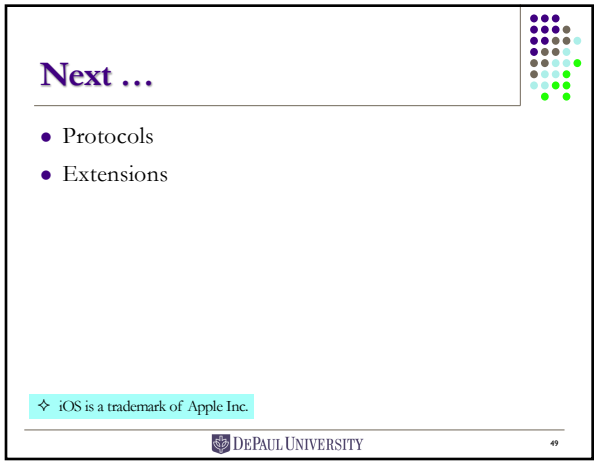
81

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

DePaul University 48

82


Next …
Protocols
Extensions
iOS is a trademark of Apple Inc.
DePaul University
49

83