CSC 491 / 391 Mobile Application Development for iOS II

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

1

Algorithms in Swift

2

Algorithms

- Useful features in Swift
  - Tuples
  - In-out parameters
  - Operators
- Algorithms
  - Shuffle
  - Sorting

DePaul University 3

3

Tuples

4

Tuples

- Aggregate multiple values into a single compound value
  - Value semantics
- A heterogeneous sequence of fixed size
  - Items are accessed by position, or optionally by names
  - Items can be of different types

```
var person = ("John", "Appleseed")
var firstName = person.0
var lastName = person.1
```

Output:
```
(.0 "John", .1 "Appleseed")
"John"
"Appleseed"
```

DePaul University 5

5

Tuples with Named Components

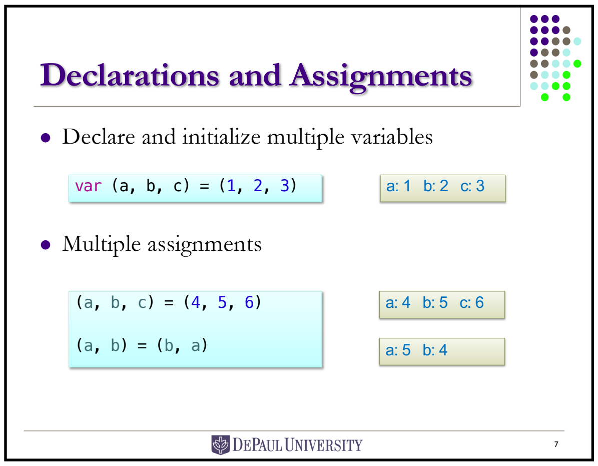
```
var john = (firstName: "John",
            lastName: "Appleseed")
john.firstName
john.lastName
john.0
john.1

var student = (name: person, ID: 1234,
               GPA: 3.7)
student.name
student.ID
student.GPA
student.name.0
student.name.1
```

```
"John"
"Appleseed"
"John"
"Appleseed"
```

```
(.0 "John", .1 "Appleseed")
1234
3.7
"John"
"Appleseed"
```

DePaul University 6

7



Declarations and Assignments

- Declare and initialize multiple variables

```
var (a, b, c) = (1, 2, 3)
```

```
a: 1  b: 2  c: 3
```

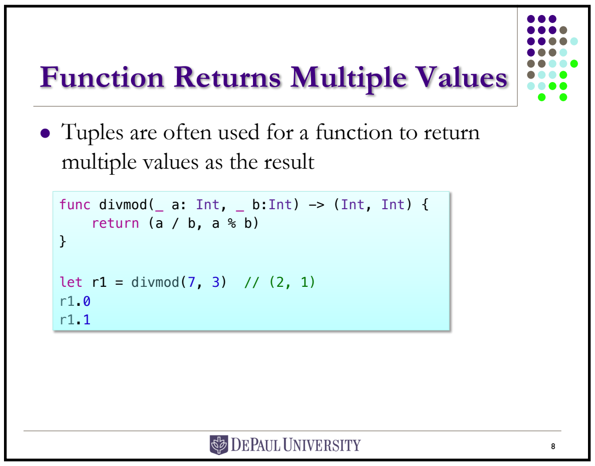
- Multiple assignments

```
(a, b, c) = (4, 5, 6)

(a, b) = (b, a)
```

```
a: 4  b: 5  c: 6
a: 5  b: 4
```

DEPAUL UNIVERSITY 7

10



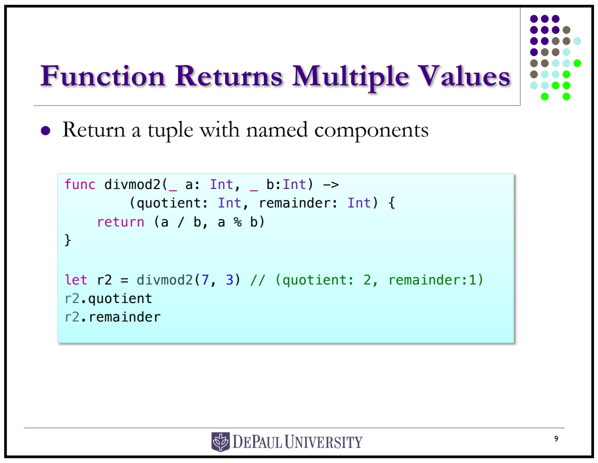
Function Returns Multiple Values

- Tuples are often used for a function to return multiple values as the result

```
func divmod(_ a: Int, _ b:Int) -> (Int, Int) {
    return (a / b, a % b)
}

let r1 = divmod(7, 3)  // (2, 1)
r1.0
r1.1
```

DEPAUL UNIVERSITY 8

11



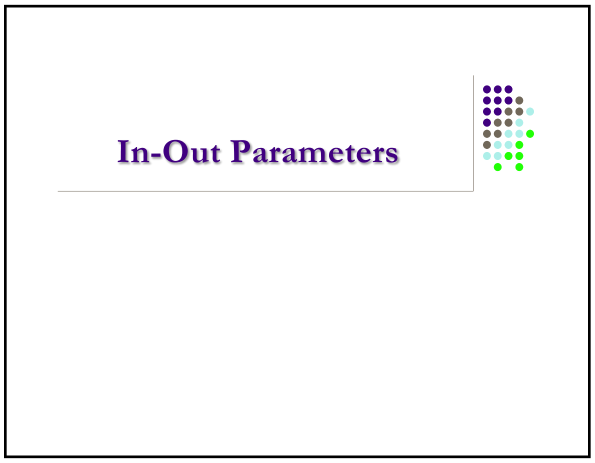
Function Returns Multiple Values

- Return a tuple with named components

```
func divmod2(_ a: Int, _ b:Int) ->
        (quotient: Int, remainder: Int) {
    return (a / b, a % b)
}

let r2 = divmod2(7, 3) // (quotient: 2, remainder:1)
r2.quotient
r2.remainder
```

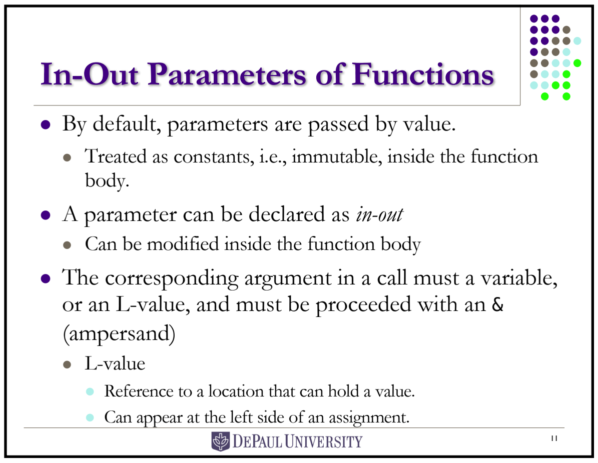
DEPAUL UNIVERSITY 9

12



In-Out Parameters

13

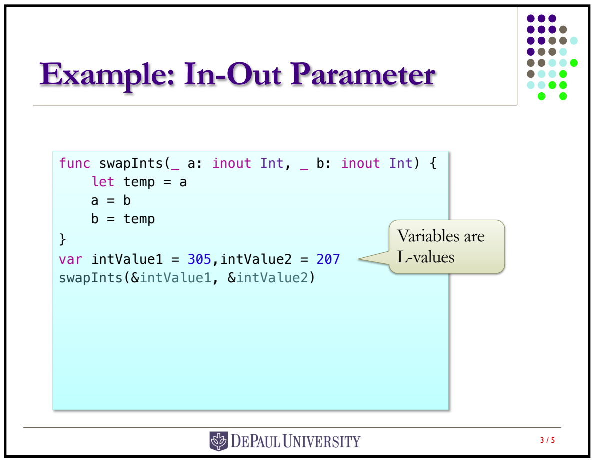


In-Out Parameters of Functions

- By default, parameters are passed by value.
  - Treated as constants, i.e., immutable, inside the function body.
- A parameter can be declared as *in-out*
  - Can be modified inside the function body
- The corresponding argument in a call must a variable, or an L-value, and must be proceeded with an & (ampersand)
  - L-value
    - Reference to a location that can hold a value.
    - Can appear at the left side of an assignment.

DEPAUL UNIVERSITY 11

14



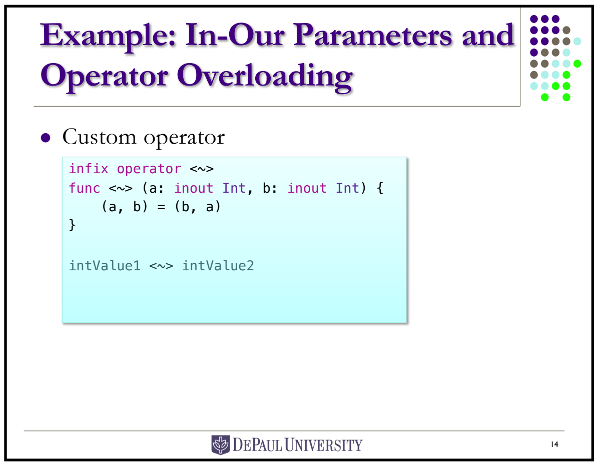
Example: In-Out Parameter

```
func swapInts(_ a: inout Int, _ b: inout Int) {
    let temp = a
    a = b
    b = temp
}
var intValue1 = 305,intValue2 = 207
swapInts(&intValue1, &intValue2)
```

Variables are L-values

DEPAUL UNIVERSITY 3 / 5

17



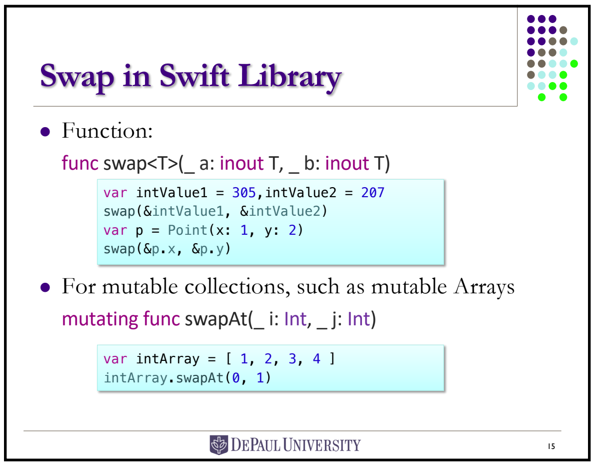
Example: In-Our Parameters and Operator Overloading

- Custom operator

```
infix operator <~>
func <~> (a: inout Int, b: inout Int) {
    (a, b) = (b, a)
}

intValue1 <~> intValue2
```

DEPAUL UNIVERSITY 14

18



Swap in Swift Library

- Function:

  func swap<T>(_ a: inout T, _ b: inout T)

```
var intValue1 = 305,intValue2 = 207
swap(&intValue1, &intValue2)
var p = Point(x: 1, y: 2)
swap(&p.x, &p.y)
```

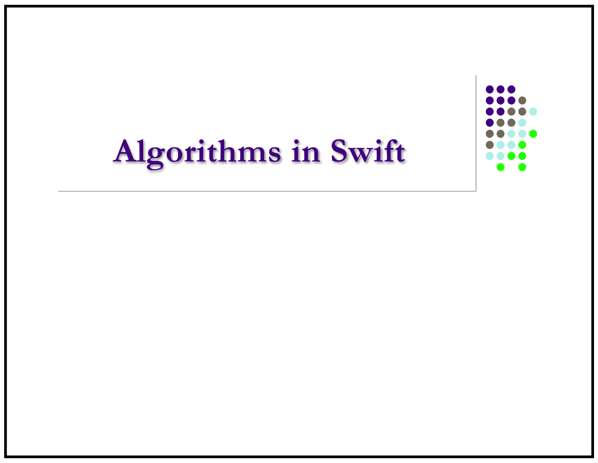
- For mutable collections, such as mutable Arrays

  mutating func swapAt(_ i: Int, _ j: Int)

```
var intArray = [ 1, 2, 3, 4 ]
intArray.swapAt(0, 1)
```

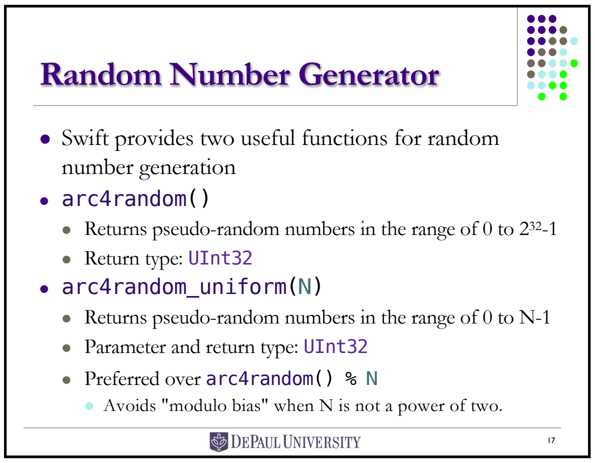
DEPAUL UNIVERSITY 15

19



Algorithms in Swift

20

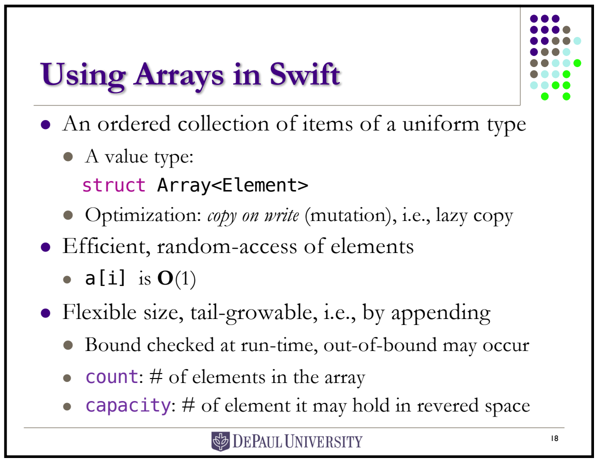


Random Number Generator

- Swift provides two useful functions for random number generation
- `arc4random()`
  - Returns pseudo-random numbers in the range of 0 to 2^{32}-1
  - Return type: `UInt32`
- `arc4random_uniform(N)`
  - Returns pseudo-random numbers in the range of 0 to N-1
  - Parameter and return type: `UInt32`
  - Preferred over `arc4random() % N`
    - Avoids "modulo bias" when N is not a power of two.

DEPAUL UNIVERSITY 17

21

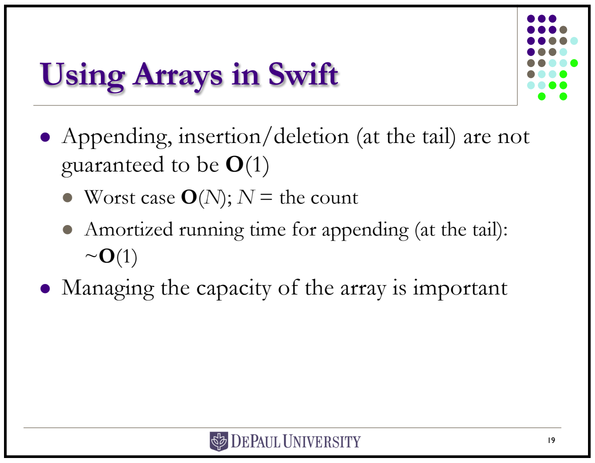


Using Arrays in Swift

- An ordered collection of items of a uniform type
  - A value type:
    `struct Array<Element>`
  - Optimization: *copy on write* (mutation), i.e., lazy copy
- Efficient, random-access of elements
  - `a[i]` is **O**(1)
- Flexible size, tail-growable, i.e., by appending
  - Bound checked at run-time, out-of-bound may occur
  - `count`: # of elements in the array
  - `capacity`: # of element it may hold in revered space

DEPAUL UNIVERSITY 18

22

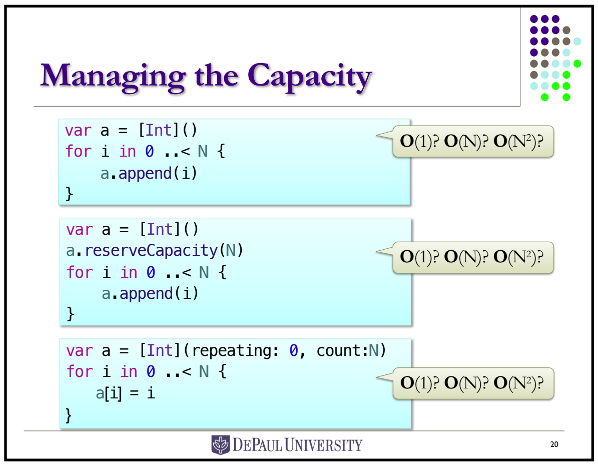


Using Arrays in Swift

- Appending, insertion/deletion (at the tail) are not guaranteed to be **O**(1)
  - Worst case **O**(*N*); *N* = the count
  - Amortized running time for appending (at the tail): ~**O**(1)
- Managing the capacity of the array is important

DEPAUL UNIVERSITY 19

23



Managing the Capacity

```
var a = [Int]()
for i in 0 ..< N {
    a.append(i)
}
```

O(1)? **O**(N)? **O**(N^2)?

```
var a = [Int]()
a.reserveCapacity(N)
for i in 0 ..< N {
    a.append(i)
}
```

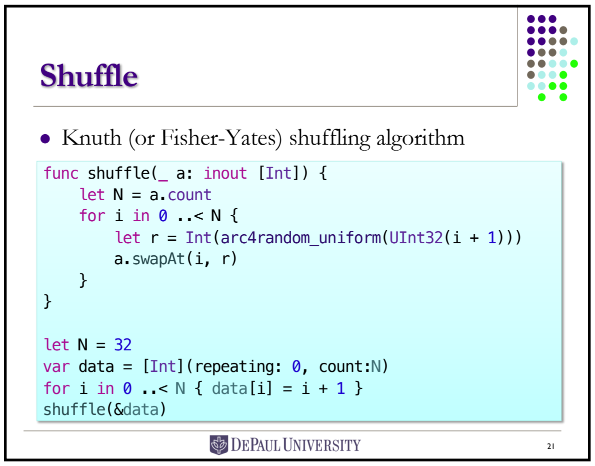
O(1)? **O**(N)? **O**(N^2)?

```
var a = [Int](repeating: 0, count:N)
for i in 0 ..< N {
   a[i] = i
}
```

O(1)? **O**(N)? **O**(N^2)?

DePaul University 20

29



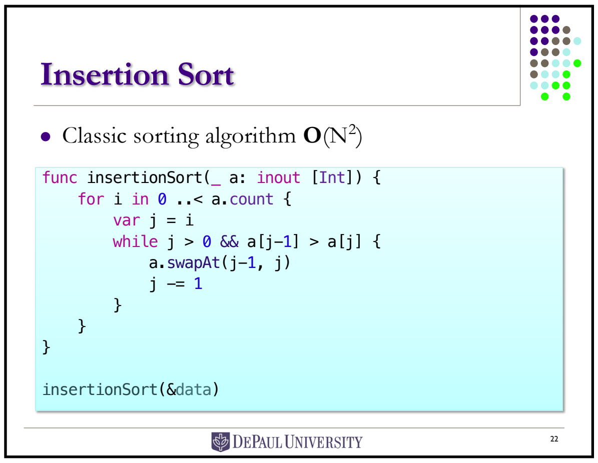
Shuffle

- Knuth (or Fisher-Yates) shuffling algorithm

```
func shuffle(_ a: inout [Int]) {
    let N = a.count
    for i in 0 ..< N {
        let r = Int(arc4random_uniform(UInt32(i + 1)))
        a.swapAt(i, r)
    }
}

let N = 32
var data = [Int](repeating: 0, count:N)
for i in 0 ..< N { data[i] = i + 1 }
shuffle(&data)
```

DePaul University 21

30



Insertion Sort

- Classic sorting algorithm **O**(N^2)

```
func insertionSort(_ a: inout [Int]) {
    for i in 0 ..< a.count {
        var j = i
        while j > 0 && a[j-1] > a[j] {
            a.swapAt(j-1, j)
            j -= 1
        }
    }
}

insertionSort(&data)
```

DePaul University 22

31