CSC 471 / 371 Mobile Application Development for iOS

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

1

Outline

- A primer on Swift programming language
  - Functions
  - Classes and objects

DEPAUL UNIVERSITY 2

Functions

3

Function Declaration

- A functions is a *block of code*
- Functions are declared in the *global* scope
  - The `func` keyword, a name, and optional parameters

```
func Identifier ( Parameter1 , Parameter2 , ... )
                                optional
    -> Type {
       optional
    Statements
}
```

Function name

Function body

Return type

DEPAUL UNIVERSITY 4

Function Calls – With Anonymous Arguments

- A *function call*
  - starts with the *name* of the function to be called
  - followed by zero or more *arguments* to the call
    - Swift allows arguments to be *anonymous* or *named*

Stay tuned

- Function calls with anonymous arguments
  - similar syntax to Java and C

```
Identifier ( Expression1 , Expression2 , ... )
             optional
```

Function name

Arguments

DEPAUL UNIVERSITY 5

A Simple Function

Function name

Function body

Function declaration

```
func helloWorld() {
    print("Hello, world!")
}

helloWorld()
```

Function call

DEPAUL UNIVERSITY 6

Functions with Parameters

- A functions may take one or more parameters:

  Identifier : *Type*

  - Parameter types *must* be declared

Parameter name

Parameter type

```
func greet(name : String) {
    print("Hello, \(name)!")
}

greet("Swift")
greet("iOS")
```

Function calls with arguments

Functions with Return Values

- A function may return values
  - The return type *must* be declared, if it returns a value
    - The default is no return value
  - A value must be returned in every path in the function body

Return type

Return statement

Return value

```
func square(n : Int) -> Int {
    return n * n
}

square(25)
square(128)
```


Functions with Multiple Parameters

- A function may take multiple parameters
- Let's declare a simple function with two parameters

```
func maximum(x: Int, y: Int) -> Int {
    return x >= y ? x : y
}
```

- Let's make a call

```
maximum(2, 5)
```

- You are in for a big surprise!
  - It is a compile **error**!

Different behavior from most languages! Different from Swift 1.0!

Swift Conversion for Parameters and Arguments

- Swift adopts different conversions for the *first* and the *rest* of the parameters
- The first parameter is *anonymous,* by default
- The second and subsequent parameters are *named,* by default
  - The arguments in function calls must be proceeded with the parameter name and a **:** (colon)

Function Calls – With Named Arguments

- Each argument in a function call may be preceded with an optional argument name

Function name

Argument name

Argument value

Identifier (*Identifier*$_1$: *Expression*$_1$,

optional

Identifier$_2$: *Expression*$_2$, ...)

optional

- Arguments must appear in the *same order* as the corresponding parameters in function declaration

Functions with Multiple Parameters

- Let's declare a simple function with two parameters

```
func maximum(x: Int, y: Int) -> Int {
    return x >= y ? x : y
}
```

- This call has an error

```
maximum(2, 5)
```

  - The second parameter `y` is named by default
- The right way to call

```
maximum(2, y: 5)
```

Okay, but why the *symmetric* parameters are treated *asymmetrically*?

Local and External Names of Parameters

- Each parameter has an *local* and *external* name
- Parameter syntax:

Identifier *Identifier* : *Type*

optional

External name

Local name

- Local names are used in the function body
- External names are used as the *argument names* in function calls
  - Default external name for the second and subsequent parameters: the local name

DEPAUL UNIVERSITY 13

External Names of Parameters

- Improve the readability of the function call
- Consider the function *maximum*

```
func maximum(x: Int, y: Int) -> Int {
    return x >= y ? x : y
}
```

- And a function call

```
maximum(2, y: 5)
```

y is the default external name

DEPAUL UNIVERSITY 14

External Names of Parameters

- Improve the readability of the function call
- Consider the function *maximum*

```
func maximumOf(x: Int, and y: Int) -> Int {
    return x >= y ? x : y
}
```

Add Of to the function name
Add an external name and

Local names x, y used in the function body.

- And a function call

```
maximumOf(2, and: 5)
```

External name and used in the function call
More readable function call

(the) maximum of 2 and 5 …

DEPAUL UNIVERSITY 15

Cryptic Function Calls

- Have you seen function calls like this?

```
printTicket("Paris", "Boston",
            "Orlando")
```

Which is the name?
Which are the locations?

```
printTicket("Tim Cook",
            "San Francisco, CA",
            "Chicago, IL")
```

Is Tim Cook coming to Chicago, or is he leaving from Chicago?

- Could be even more confusing with more arguments

DEPAUL UNIVERSITY 16

Named Arguments in Function Call

- Using external names makes the meaning clear

```
func printTicket(name: String,
                 origin: String,
                 destination: String) {
    print("Ticket\n  Passenger name: \(name)")
    print("  From: \(origin)\n  To: \(destination)")
}

printTicket("Tim Cook",
            origin: "San Francisco, CA",
            destination: "Chicago, IL")
```

Same external & local names: origin, destination

Clear meaning. But not quite a fluent sentence

DEPAUL UNIVERSITY 17

Make It More Fluent

```
func printTicketFor(name: String,
                    from origin: String,
                    to destination: String) {
    print("Ticket\n  Passenger name: \(name)")
    print("  From: \(origin)\n  To: \(destination)")
}

printTicketFor("Tim Cook",
               from: "San Francisco, CA",
               to: "Chicago, IL")
```

Add For to the name and external names: from, to

Function body uses the local names. Readable and clear in intent.

Readable and fluent function call

print ticket for Time Cook, from San Francisco, CA to Chicago, IL

DEPAUL UNIVERSITY 18

Thank You, Swift! But, I am a Traditionalist

- Consider the *median* function

```
func median(x: Int, y: Int, z: Int) -> Int {
    return x > y ? (y > z ? y : x > z ? z : x)
                 : (x > z ? x : y > z ? z : y)
}
```

- I want to call it like this

```
median(2, 5, 3)
```

Brevity, love it!

- Not like this

```
median(2, y: 5, z: 3)
```

Verbosity, not so much.

DePaul University 19

Anonymous External Names

- Use _ to indicate an *anonymous* external name
  - This is the default for the *first parameter* of functions.
- Now we can declare the *median* function as

```
func median(x: Int, _ y: Int, _ z: Int) -> Int {
    return x > y ? (y > z ? y : x > z ? z : x)
                 : (x > z ? x : y > z ? z : y)
}
```

- And call it the way you like

```
median(2, 5, 3)
```

DePaul University 20

It is a Matter of Style

- Again, here is the Swift way

Same external name but different local names

```
func medianOf(x: Int, and y: Int, and z: Int) -> Int {
    return x > y ? (y > z ? y : x > z ? z : x)
                 : (x > z ? x : y > z ? z : y)
}
```

- And here is the call, which reads like a sentence

```
medianOf(2, and: 5, and: 3)
```

(the) median of 2 and 5 and 3

DePaul University 21

Optional Parameters with Default Values

- You can provide a default value for a parameter

```
func greeting(name : String = "world") {
    print("Hello, \(name)!")
}

greeting("Swift")
greeting()
```

Default value

DePaul University 22

Classes and Objects

23

Some Basic Terminologies

- Class
  - A group of objects that share common characteristics or behaviors
  - Defines the *type* of the objects that belong to the class
- Object
  - An instance of a class
- Property
  - An attribute of an object
- Method
  - A function, or task, that can be performed by an object

More Object-Oriented terminology later

DePaul University 24

Class Declaration

- A class declaration include
  - A class *name*
  - *Properties* to store values
    - Declared as constants or variables inside a class
  - *Methods* to provide functionalities
    - Declared as functions inside a class
  - *Initializers* to set up the initial state of objects
- Swift *does not* separate class interfaces from implementations
  - A class declaration is contained in a single file.

DEPAUL UNIVERSITY 25

A Simple Class: Counter

- Class: *Counter*
  - A property *count*
  - A method *increment*

```
class Counter {
    var count = 0
    func increment() {
        count++
    }
}
```

The class name
A class declaration
A property
A method

DEPAUL UNIVERSITY 26

Creating and Using Objects

- Creating object instance
  ClassName ( )
  ClassName (*arguments*)
- Accessing properties
  object.property
  object.property = *expression*
- Calling methods
  object.method (*arguments*)

```
var c1 = Counter()
c1.increment()
c1.increment()
c1.count
c1.count = 0
c1.increment()
c1.count
```

DEPAUL UNIVERSITY 27

The Counter Class, Version 2 – Additional Methods

```
class Counter {
    var count = 0
    func increment() {
        count++
    }
    func decrement() {
        count--
    }
    func incrementBy(c: Int) {
        count += c
    }
    func decrementBy(c: Int) {
        count -= c
    }
}
```

```
var c2 = Counter()
c2.incrementBy(10)
c2.count
c2.decrement()
c2.decrementBy(5)
c2.count
```

DEPAUL UNIVERSITY 28

The Fraction Class – The Initial Version

- A class representing a fraction: a/b
  - Both a and b are integers, $b > 0$
  - a: numerator; b: denominator.

```
class Fraction {
    var numerator: Int = 0
    var denominator: Int = 1
    func printFraction() {
        print("\(numerator)/\(denominator)")
    }
    func toDouble() -> Double {
        return Double(numerator) / Double(denominator);
    }
}
```

DEPAUL UNIVERSITY 29

The Fraction Class – The Initial Version

```
var f1 = Fraction()
f1.printFraction()
f1.numerator = 1
f1.denominator = 3
f1.printFraction()
print(f1.numerator)
print(f1.denominator)
print(f1.toDouble())
```

DEPAUL UNIVERSITY 30

The Fraction Class – Initializers

- Initializers: set up the initial state of new instances
- Called when new instances are created

```
class Fraction {
    var numerator: Int = 0
    var denominator: Int = 1
    init(numerator: Int, denominator: Int) {
        self.numerator = numerator
        self.denominator = denominator
    }
    …
}
var f1 = Fraction(numerator: 1, denominator: 2)
```

An initializer

Note the external name of the first argument of the initializer

Calling the initializer

DePaul University 31

Rules for External Parameter Names For Functions, Methods, & Initializers

- If no external parameter name is provided, a default external parameter name is given
- For functions (global) and **methods** (inside classes)
  - For the first parameter: anonymous, i.e., _
  - For the second and subsequent parameters: same as the local parameter names
- For initializers, i.e., methods with the name `init`
  - All parameters: same as the local parameter names
  - No exception for the first parameter

The external name of the first parameter is useful, since the initializer name cannot be modified

DePaul University 32

The Keyword `self`

- Equivalent to the `this` keyword in Java and C++
- Refer to the object itself
- Used to distinguish a property of the class from a parameter of the initializer.

```
init(numerator: Int, denominator: Int) {
    self.numerator = numerator
    self.denominator = denominator
}
```

The property

The parameter

DePaul University 33

The Fraction Class – Initializers

- A second initializer

```
class Fraction {
    var numerator: Int = 0
    var denominator: Int = 1
    init(numerator: Int, denominator: Int) { … }
    init(_ numerator: Int, over denominator: Int) {
        self.numerator = numerator
        self.denominator = denominator
    }
    …
}
var f2 = Fraction(2, over: 3)
```

An initializer with external names

Calling the initializer with external names

DePaul University 34

The Fraction Class – Default Initializers

- The *default* initializer is available if no initializer is defined.

```
class Fraction {
    var numerator: Int = 0
    var denominator: Int = 1
    init(numerator: Int, denominator: Int) { … }
    init(_ numerator: Int, over denominator: Int) { … }
    init() {}
    …
}

var f3 = Fraction()
```

The default initializer

Calling the default initializer

DePaul University 35

The Fraction Class – Methods with Multiple Parameters

- Method `setTo`

```
class Fraction {
    var numerator: Int = 0
    var denominator: Int = 1
    func setTo(numerator: Int, denominator: Int) {
        self.numerator = numerator
        self.denominator = denominator
    }
    …
}
var f3 = Fraction()
f3.setTo(1, denominator: 3)
```

Note the external name for the second argument

DePaul University 36

The Fraction Class – Methods with Multiple Parameters

- Choose a better external name

```
class Fraction {
    var numerator: Int = 0
    var denominator: Int = 1
    func setTo(numerator: Int, denominator: Int) { … }
    func setTo(numerator: Int, over denominator: Int) {
        self.numerator = numerator
        self.denominator = denominator
    }
    …
}
var f4 = Fraction()
f4.setTo(1, over: 4)
```

Explicit external name

DePaul University 37

The Fraction Class – Methods with Multiple Parameters

```
class Fraction {
    var numerator: Int = 0
    var denominator: Int = 1
    func setTo(numerator: Int, denominator: Int) { … }
    func setTo(numerator: Int, over denominator: Int) { … }
    func setTo(numerator: Int, _ denominator: Int) {
        self.numerator = numerator
        self.denominator = denominator
    }
    …
}
var f5 = Fraction()
f5.setTo(3, 4)
```

Explicit anonymous external name

DePaul University 38

The Fraction Class – The Addition Method

```
class Fraction {
    var numerator: Int = 0
    var denominator: Int = 1
    func add(f: Fraction) {
        numerator = numerator * f.denominator
                    + denominator * f.numerator;
        denominator = denominator * f.denominator;
    }
    …
}
var f1 = Fraction(1, over: 2)
var f2 = Fraction(1, over: 4)
f1.add(f2)
```

Adding two fractions
$a/b + c/d = (a*d + c*b) / b*d$

DePaul University 39

The Fraction Class – The Addition Method

```
class Fraction {
    func reduce() {
        let sign = numerator >= 0 ? 1 : -1;
        var u = numerator * sign;
        var v = denominator;
        var temp: Int;
        while (v != 0) {
            temp = u % v;
            u = v;
            v = temp;
        }
        numerator /= u * sign;
        denominator /= u;
    }
    …
}
```

```
var f1 = Fraction(1, over: 2)
var f2 = Fraction(1, over: 4)
f1.add(f2)
f1.reduce()
```

40

The Fraction Class – The Addition Method

```
class Fraction {
    var numerator: Int = 0
    var denominator: Int = 1
    func add(f: Fraction) {
        numerator = numerator * f.denominator
                    + denominator * f.numerator;
        denominator = denominator * f.denominator;
        reduce()
    }
    func reduce() { … }
    …
}
var f1 = Fraction(1, over: 2)
var f2 = Fraction(1, over: 4)
f1.add(f2)
```

The Fraction Class – The Addition Method

```
class Fraction {
    var numerator: Int = 0
    var denominator: Int = 1
    func add(f: Fraction) -> Fraction {
        var result: Fraction = Fraction()
        result.numerator = numerator * f.denominator
          + denominator * f.numerator;
        result.denominator = denominator * f.denominator;
        result.reduce()
        return result
    }
    func reduce() { … }
    …
}
```

Return the result as a Fraction object

```
let f1 = Fraction(1, over: 2)
let f2 = Fraction(1, over: 4)
let f3 = f1.add(f2)
```

DePaul University 42

The Fraction Class – The Addition Function

- The fraction addition can also be defined as a global function
  - Outside the `Fraction` class

```
func add(a: Fraction, _ b: Fraction) -> Fraction {
    return a.add(b)
}

let f1 = Fraction(1, over: 2)
let f2 = Fraction(1, over: 4)
let f4 = add(f1, f2)
```


The Fraction Class – The Addition Operator

- The fraction addition can also be defined to use the operator +
  - Operator overloading
  - Similar syntax to function

```
func +(a: Fraction, b: Fraction) -> Fraction {
    return a.add(b)
}

let f1 = Fraction(1, over: 2)
let f2 = Fraction(1, over: 4)
let f5 = f1 + f2
```


The Fraction Class – The Compound Assignment

- You can also overload the += operator

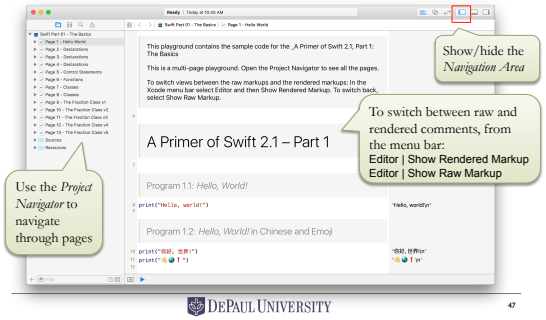
```
func +=(inout left: Fraction, right: Fraction) {
    left = left + right
}

let f2 = Fraction(1, over: 4)
var f6 = Fraction(1, over: 2)
f6 += f2
```


Sample Code & Materials

- All sample code in this lecture are in D2L
  - Swift 2.1 Examples
  - Run in Xcode 7 Playground
- *A Primer of Swift Programming Language, version 2.1*
  - Expanded materials & examples

Multipage Playground with Rich Markups



Next …

- Architecture of iOS
- Fundamentals of iOS apps
- Storyboard and Interface Builder
- *IBOutlet* and *IBAction*
- Buttons and Labels
- More Swift

✧ Xcode, iOS, WatchOS are trademarks of Apple Inc.