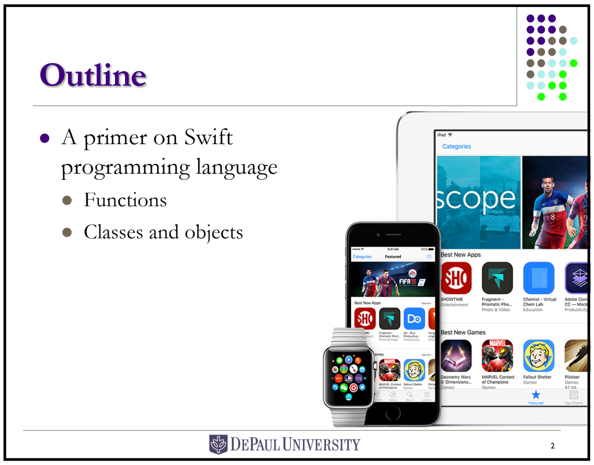
CSC 471 / 371 Mobile Application Development for iOS

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

1

0

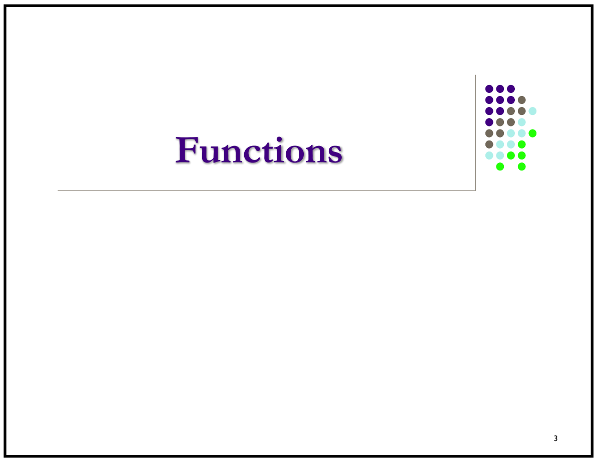


Outline

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

DePaul University 2

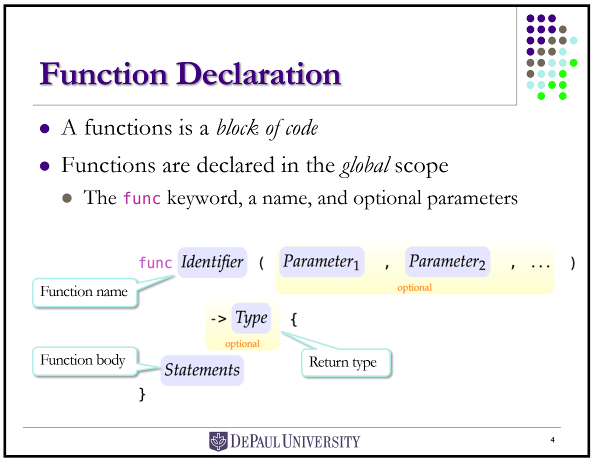
1



Functions

3

2



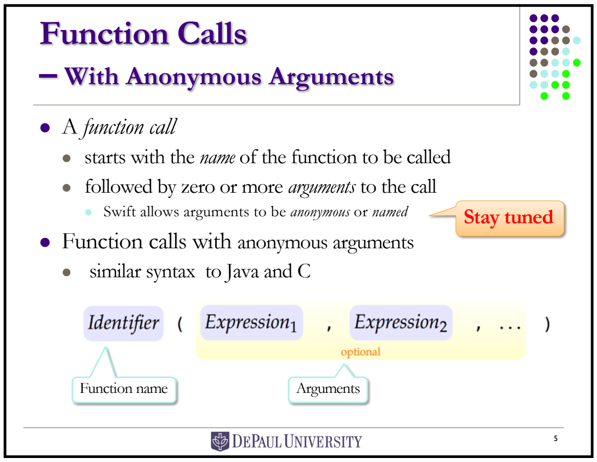
Function Declaration

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

func *Identifier* (*Parameter*$_1$, *Parameter*$_2$, …)
optional
-> *Type* {
optional
Statements
}

DePaul University 4

6



Function Calls – With Anonymous Arguments

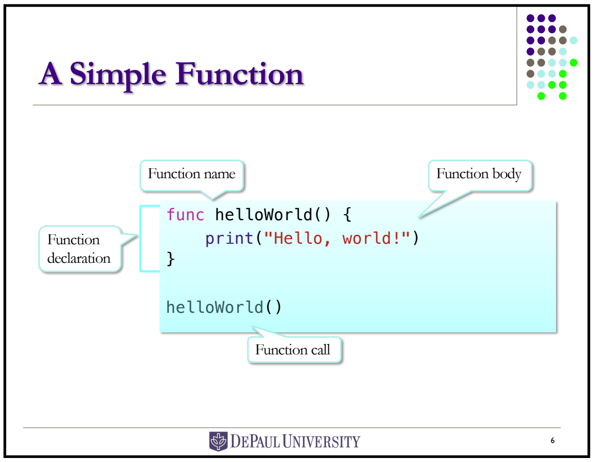
- A *function call*
  - starts with the *name* of the function to be called
  - followed by zero or more *arguments* to the call
    - Swift allows arguments to be *anonymous* or *named*
- Function calls with anonymous arguments
  - similar syntax to Java and C

Stay tuned

Identifier (*Expression*$_1$, *Expression*$_2$, …)
optional

DePaul University 5

11

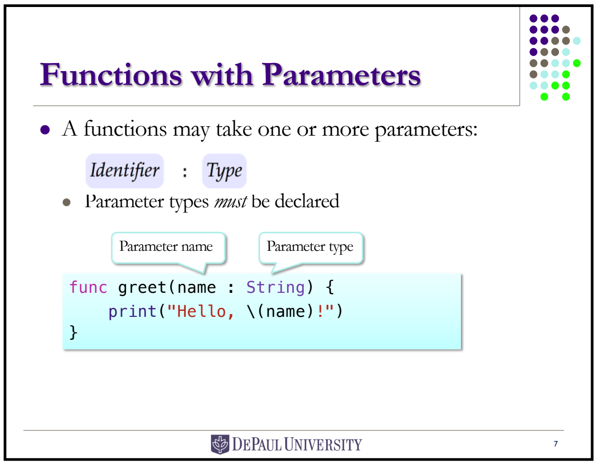


A Simple Function

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

helloWorld()
```

DePaul University 6

17



Functions with Parameters

- A functions may take one or more parameters:

  Identifier : *Type*

  - Parameter types *must* be declared

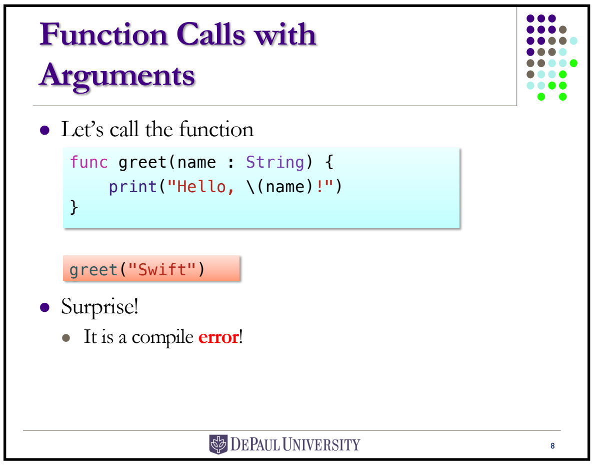
Parameter name

Parameter type

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

DePaul University 7

21



Function Calls with Arguments

- Let's call the function

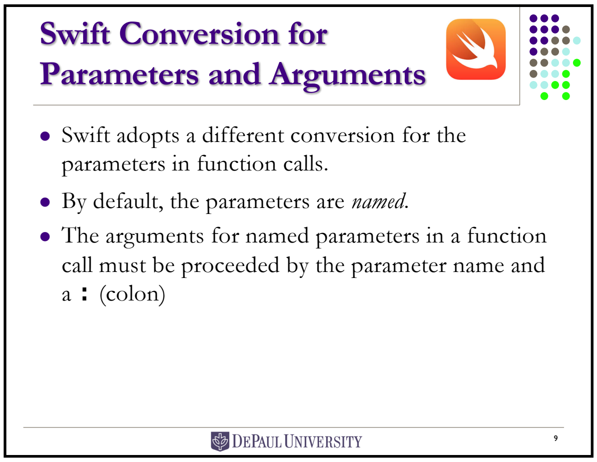
```
func greet(name : String) {
    print("Hello, \(name)!")
}
```

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

- Surprise!
  - It is a compile **error**!

DePaul University 8

24

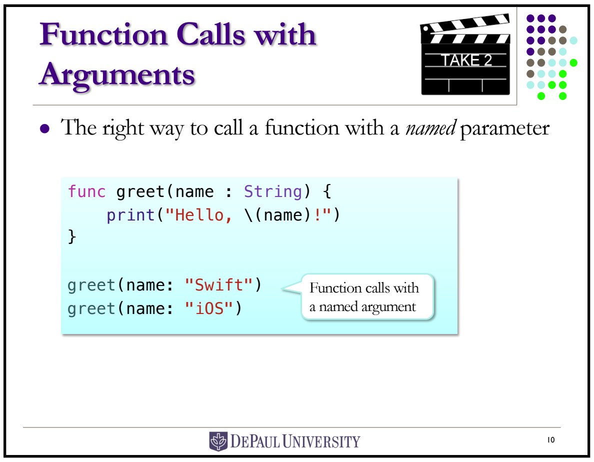


Swift Conversion for Parameters and Arguments

- Swift adopts a different conversion for the parameters in function calls.
- By default, the parameters are *named.*
- The arguments for named parameters in a function call must be proceeded by the parameter name and a **:** (colon)

DePaul University 9

25



Function Calls with Arguments

TAKE 2

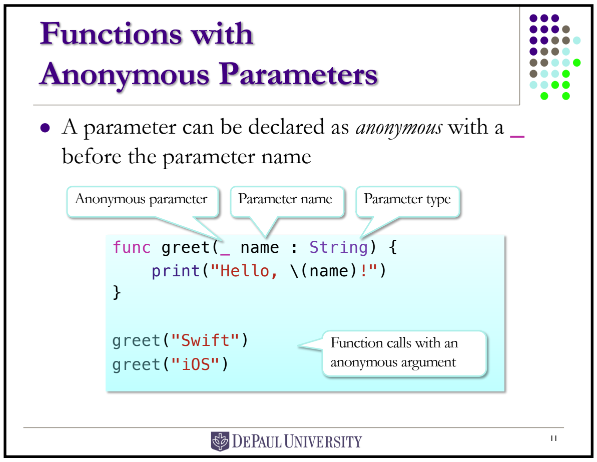
- The right way to call a function with a *named* parameter

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

greet(name: "Swift")
greet(name: "iOS")
```

Function calls with a named argument

DePaul University 10

26



Functions with Anonymous Parameters

- A parameter can be declared as *anonymous* with a _ before the parameter name

Anonymous parameter

Parameter name

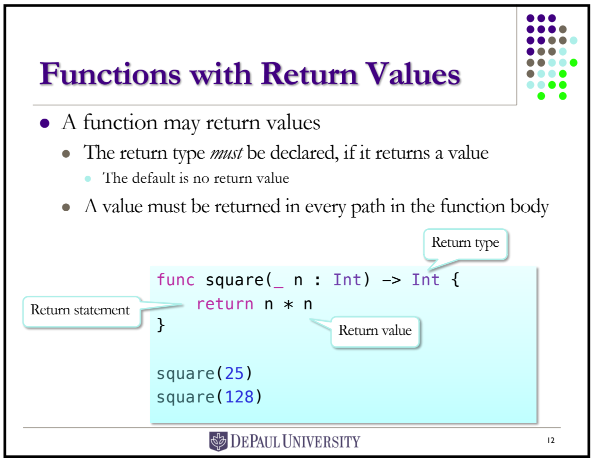
Parameter type

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

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

Function calls with an anonymous argument

DePaul University 11

30



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

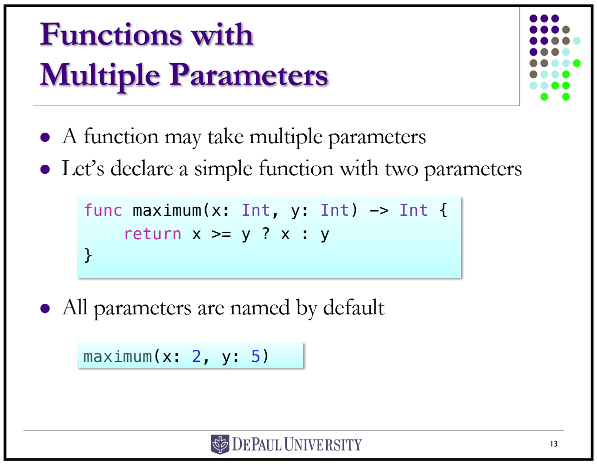
Return statement

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

square(25)
square(128)
```

Return value

DePaul University 12

36



Functions with Multiple Parameters

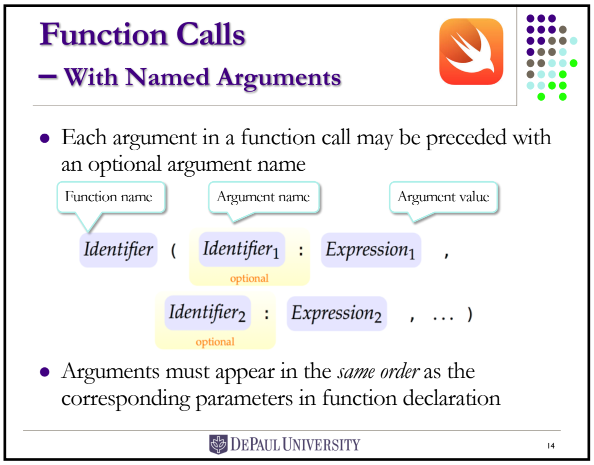
- A function may take multiple parameters
- Let's declare a simple function with two parameters

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

- All parameters are named by default

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

38



Function Calls – With Named Arguments

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

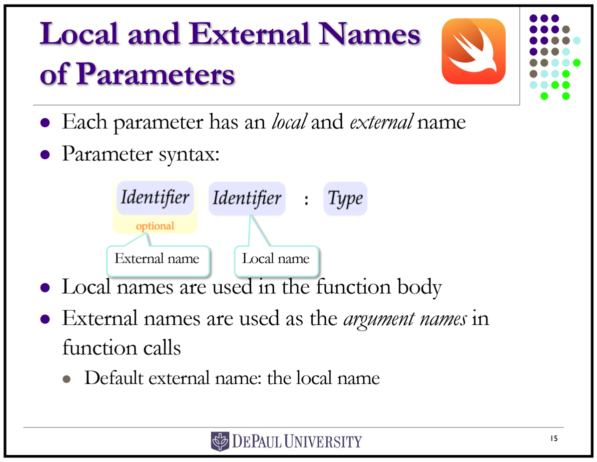
Function name
Argument name
Argument value

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

43



Local and External Names of Parameters

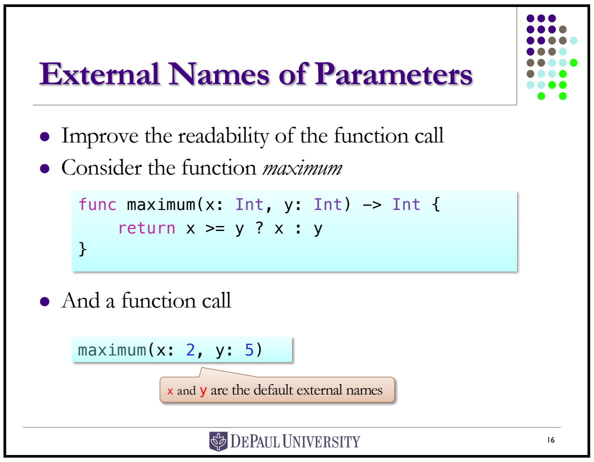
- Each parameter has an *local* and *external* name
- Parameter syntax:

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: the local name

47



External Names of Parameters

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

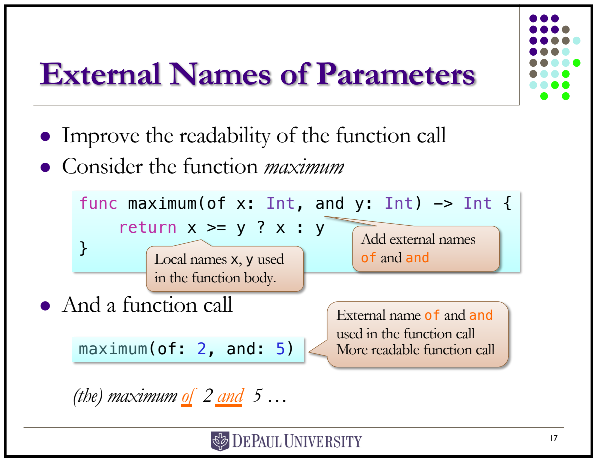
```
func maximum(x: Int, y: Int) -> Int {
    return x >= y ? x : y
}
```

- And a function call

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

x and y are the default external names

50



External Names of Parameters

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

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

Local names x, y used in the function body.

Add external names of and and

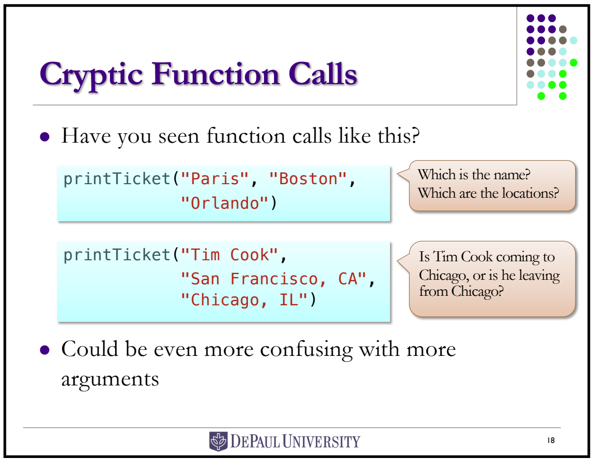
- And a function call

```
maximum(of: 2, and: 5)
```

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

(the) maximum of 2 and 5 …

55



Cryptic Function Calls

- Have you seen function calls like this?

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

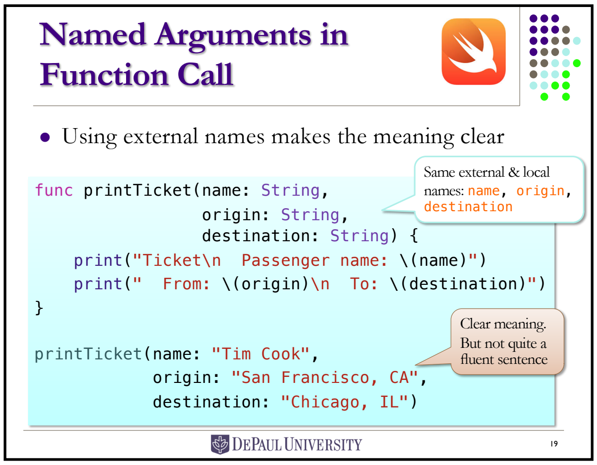
Which is the name? Which are the locations?

```
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

60



Named Arguments in Function Call

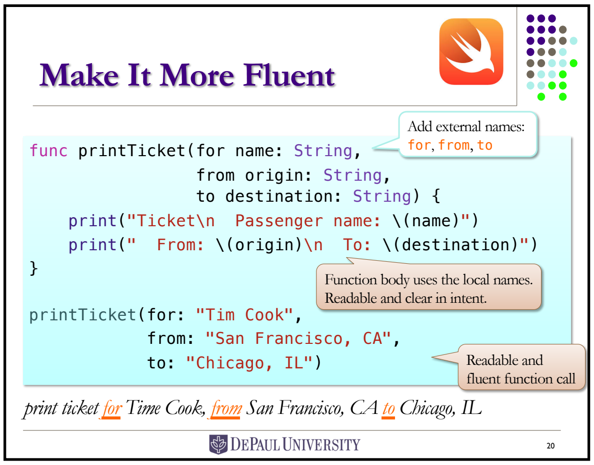
- Using external names makes the meaning clear

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

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

Same external & local names: name, origin, destination

Clear meaning. But not quite a fluent sentence

64



Make It More Fluent

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

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

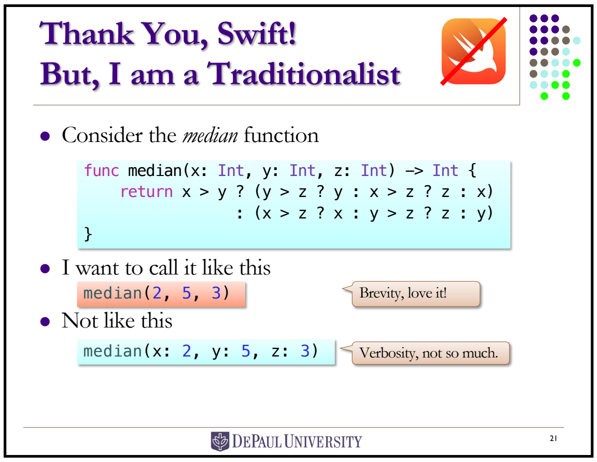
Add external names: for, from, to

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

68



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)
```

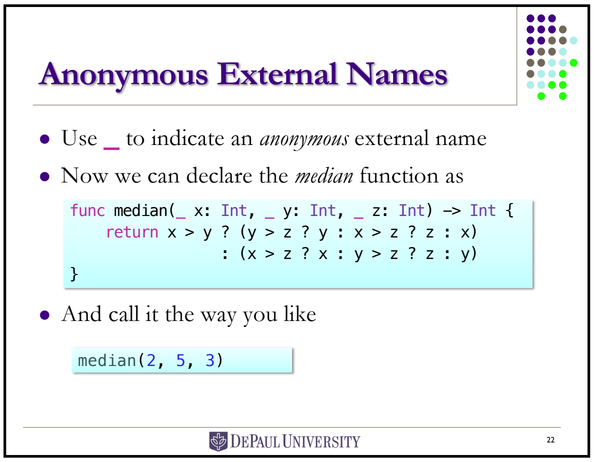
Brevity, love it!

- Not like this

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

Verbosity, not so much.

72



Anonymous External Names

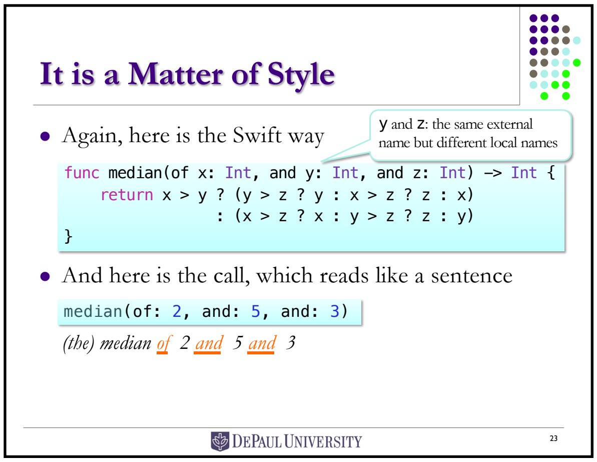
- Use _ to indicate an *anonymous* external name
- Now we can declare the *median* function as

```
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)
```

75



It is a Matter of Style

- Again, here is the Swift way

y and z: the same external name but different local names

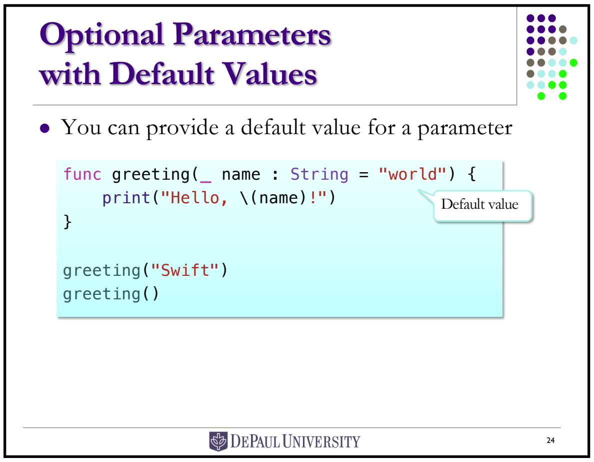
```
func median(of x: Int, and y: Int, and z: Int) -> Int {
    return x > y ? (y > z ? y : x > z ? z : x)
                 : (x > z ? x : y > z ? z : y)
}
```

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

```
median(of: 2, and: 5, and: 3)
```

(the) median of 2 and 5 and 3

78



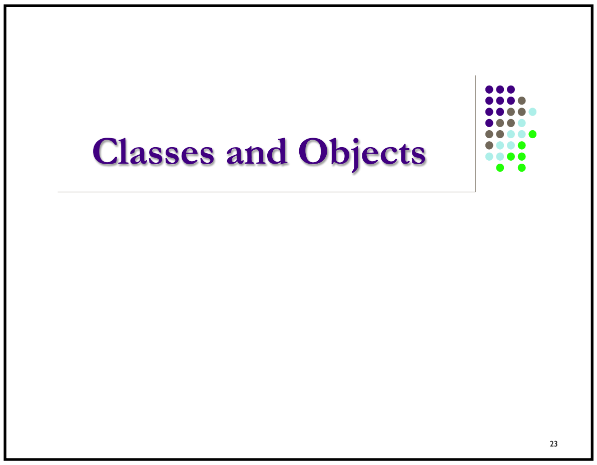
Optional Parameters with Default Values

- You can provide a default value for a parameter

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

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

Default value

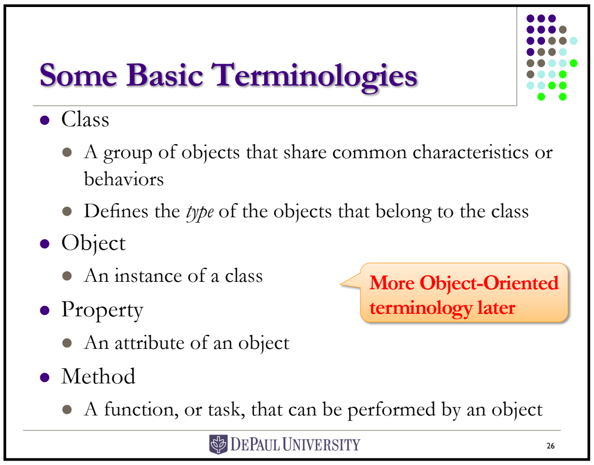
80



Classes and Objects

23

81



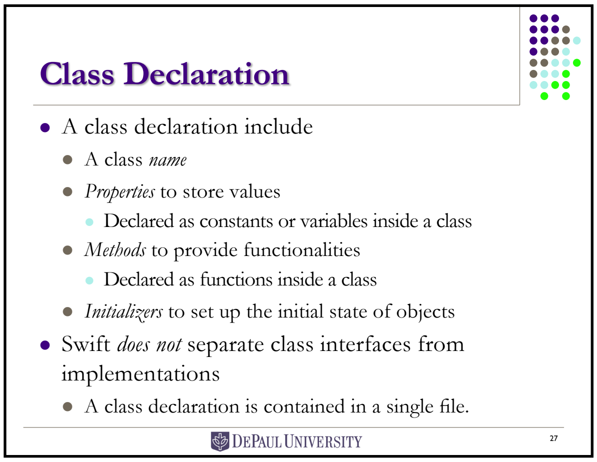
Some Basic Terminologies

- 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 26

83

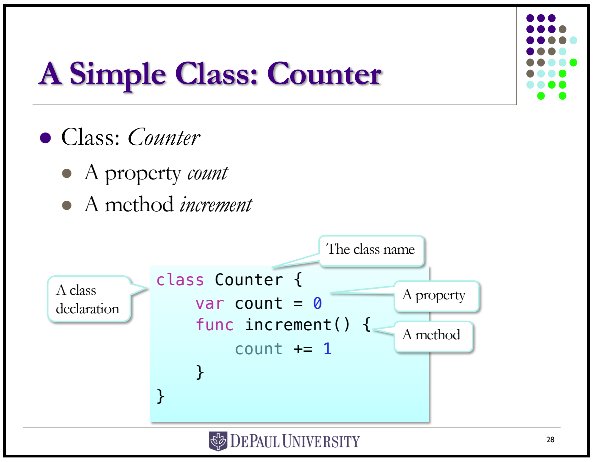


Class Declaration

- 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 27

84



A Simple Class: Counter

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

The class name

A class declaration

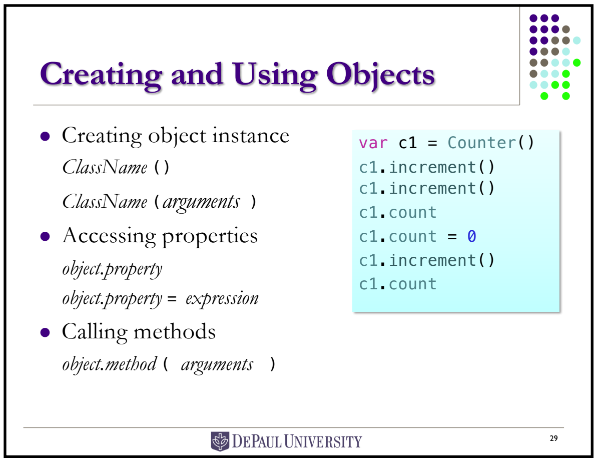
A property

A method

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

DePaul University 28

89



Creating and Using Objects

- Creating object instance

  ClassName ( )

  ClassName (*arguments*)
- Accessing properties

  object.property

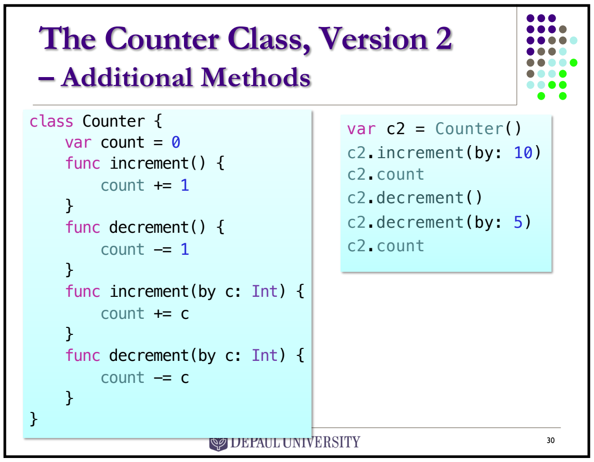
  object.property = *expression*
- Calling methods

  object.method (*arguments*)

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

DePaul University 29

90



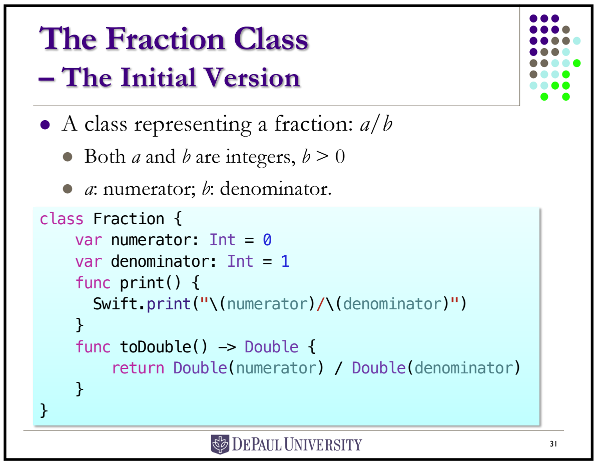
The Counter Class, Version 2 – Additional Methods

```
class Counter {
    var count = 0
    func increment() {
        count += 1
    }
    func decrement() {
        count -= 1
    }
    func increment(by c: Int) {
        count += c
    }
    func decrement(by c: Int) {
        count -= c
    }
}
```

```
var c2 = Counter()
c2.increment(by: 10)
c2.count
c2.decrement()
c2.decrement(by: 5)
c2.count
```

DePaul University 30

91

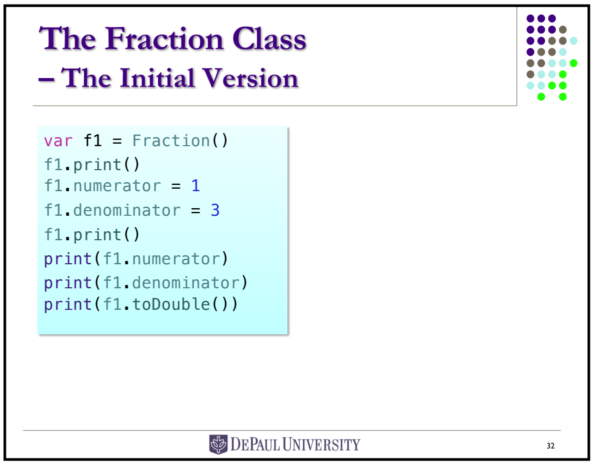


The Fraction Class – The Initial Version

- 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 print() {
      Swift.print("\(numerator)/\(denominator)")
    }
    func toDouble() -> Double {
        return Double(numerator) / Double(denominator)
    }
}
```

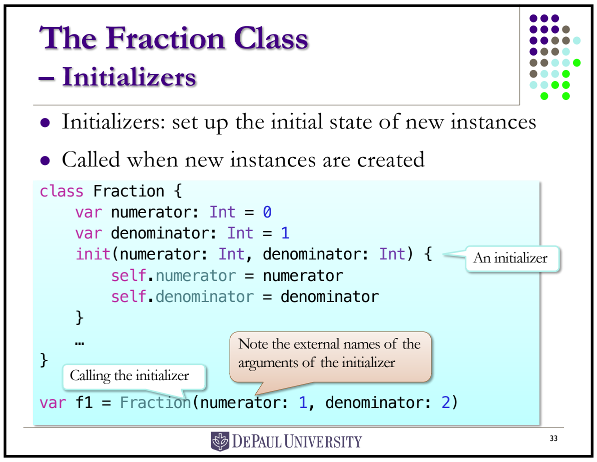
92



The Fraction Class – The Initial Version

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

93



The Fraction Class – Initializers

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

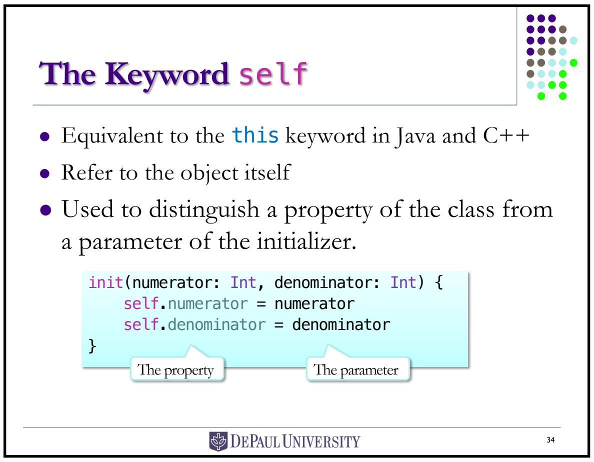
```
class Fraction {
    var numerator: Int = 0
    var denominator: Int = 1
    init(numerator: Int, denominator: Int) {
        self.numerator = numerator
        self.denominator = denominator
    }
    …
}
var f1 = Fraction(numerator: 1, denominator: 2)
```

An initializer

Note the external names of the arguments of the initializer

Calling the initializer

99



The Keyword `self`

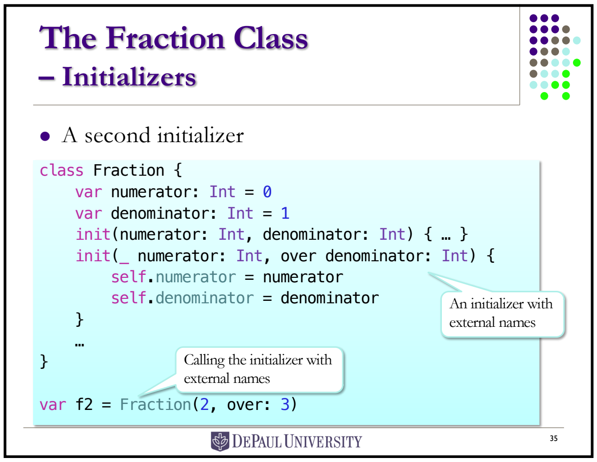
- Equivalent to the `this` keyword in Java and C++
- Refer to the object itself
- Used to distinguish a property of the class from a parameter of the initializer.

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

The property

The parameter

102



The Fraction Class – Initializers

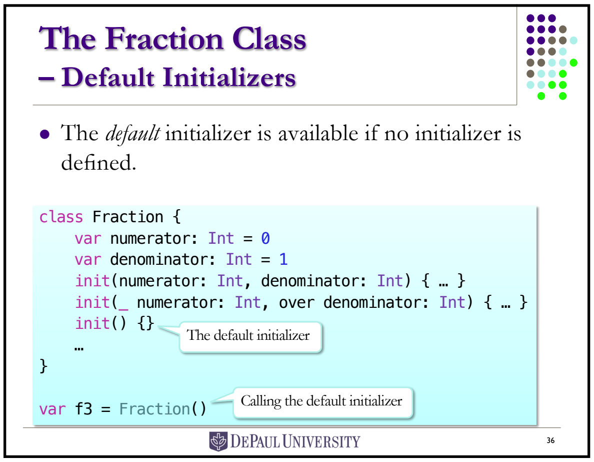
- A second initializer

```
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

105



The Fraction Class – Default Initializers

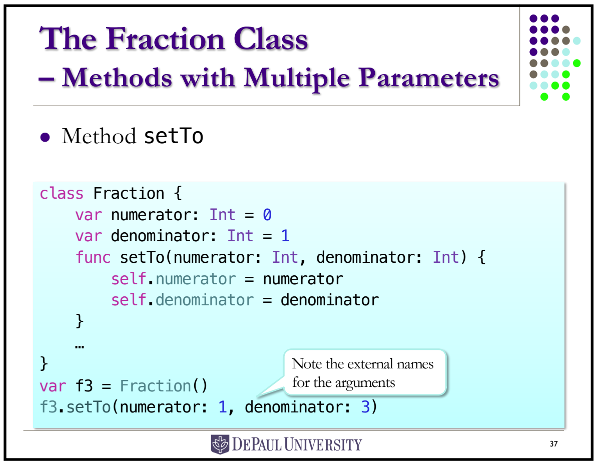
- The *default* initializer is available if no initializer is defined.

```
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

108



The Fraction Class – Methods with Multiple Parameters

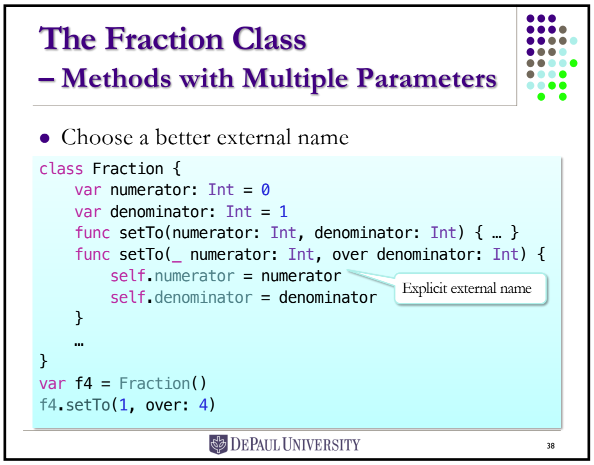
- Method `setTo`

```
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(numerator: 1, denominator: 3)
```

Note the external names for the arguments

DEPAUL UNIVERSITY 37

110



The Fraction Class – Methods with Multiple Parameters

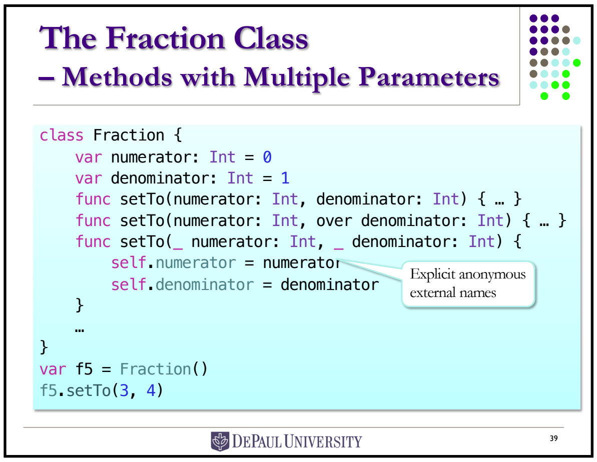
- Choose a better external name

```
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 38

112



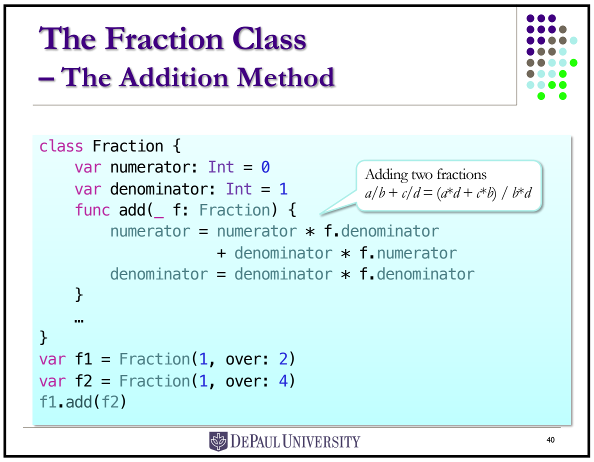
The Fraction Class – Methods with Multiple Parameters

```
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 names

DEPAUL UNIVERSITY 39

114



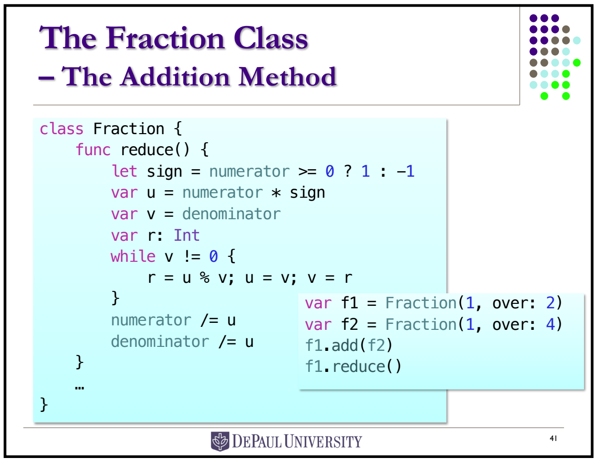
The Fraction Class – The Addition Method

```
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 40

116



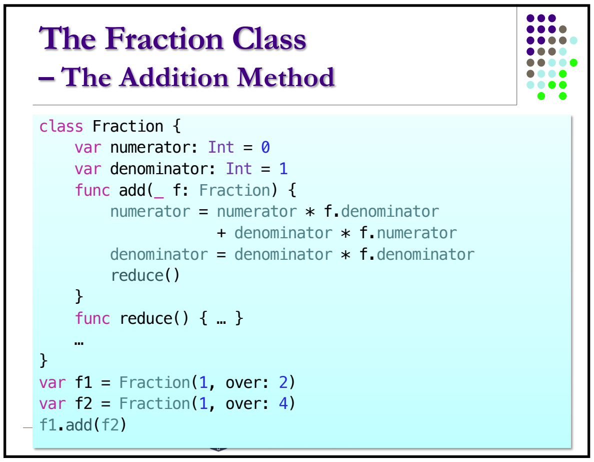
The Fraction Class – The Addition Method

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

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

DEPAUL UNIVERSITY 41

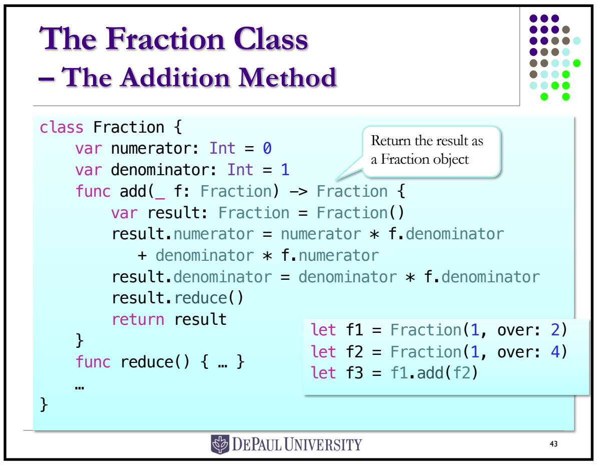
119



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)
```

121



The Fraction Class – The Addition Method

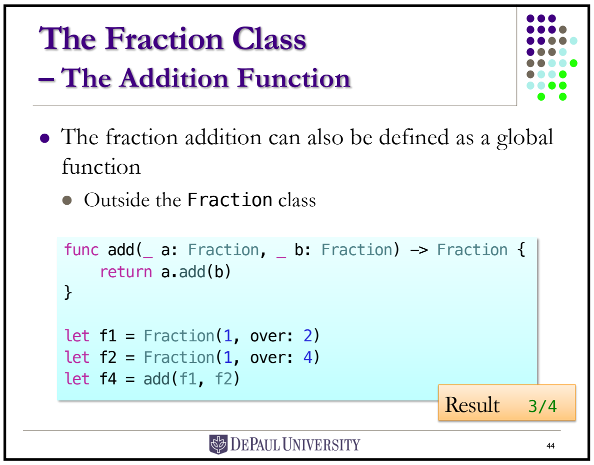
```
class Fraction {
    var numerator: Int = 0
    var denominator: Int = 1
    func add(_ f: Fraction) -> Fraction {
        var result: Fraction = Fraction()
        result.numerator = numerator * f.denominator
           + denominator * f.numerator
        result.denominator = denominator * f.denominator
        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 43

124



The Fraction Class – The Addition Function

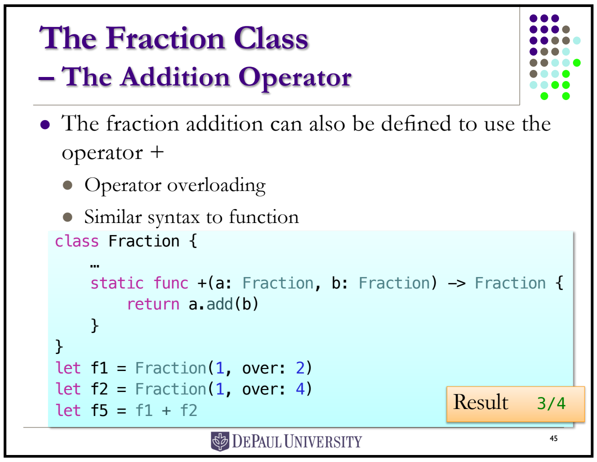
- The fraction addition can also be defined as a global function
  - Outside the `Fraction` class

```
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)
```

Result 3/4

DePaul University 44

125



The Fraction Class – The Addition Operator

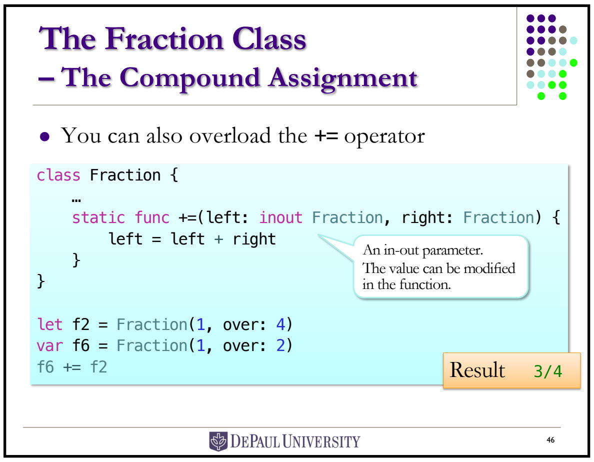
- The fraction addition can also be defined to use the operator +
  - Operator overloading
  - Similar syntax to function

```
class Fraction {
    …
    static 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
```

Result 3/4

DePaul University 45

126



The Fraction Class – The Compound Assignment

- You can also overload the += operator

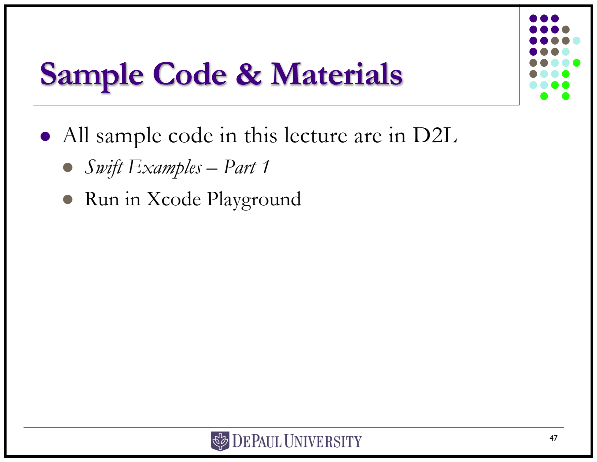
```
class Fraction {
    …
    static func +=(left: inout Fraction, right: Fraction) {
        left = left + right
    }
}

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

An in-out parameter. The value can be modified in the function.

Result 3/4

DePaul University 46

127

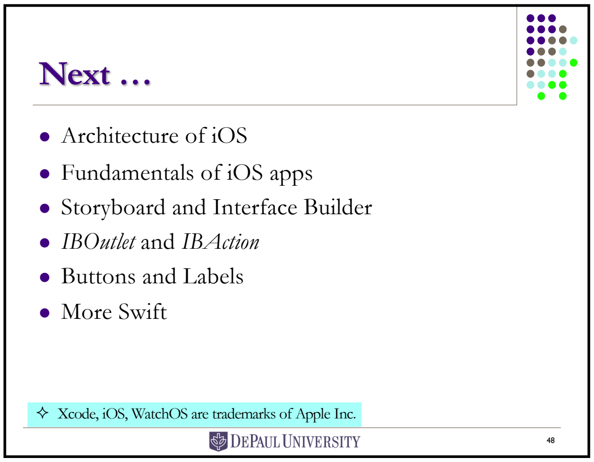


Sample Code & Materials

- All sample code in this lecture are in D2L
  - *Swift Examples – Part 1*
  - Run in Xcode Playground

DePaul University 47

128



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.

DePaul University 48

129