Swift Programming

iOS 9 • Xcode 7

STUDENT GUIDE

ABOUTOBJECTS

Contact

About Objects, Inc.
1818 Library Street
5th Floor
Reston, VA 20190

Main: 571–346–7544
Fax: 703–327–3396
email: info@aboutobjects.com
web: **www.aboutobjects.com**

Course Information

Author: Jonathan Lehr
Revision: 2.0
Last Update: 1/25/2016

Classroom materials for an course that provides a rapid introduction to programming in Swift. Geared to developers interested in learning to do Cocoa development on the iOS platform. Includes comprehensive lab exercise instructions and solution code.

Copyright Notice

Swift Programming

STUDENT GUIDE

About the Instructor

Jonathan Lehr

Founder and VP, Training

About Objects

jonathan@aboutobjects.com

ABOUTOBJECTS

CHAPTER ONE

Swift Basics

About Swift

- Why Swift?
  - Safety
  - Speed
  - Modern language features (functional programming, closures, pattern matching, tuples, optionals, etc.)
- Apple plans to use Swift for:
  1. Systems programming
  2. Scripting
  3. Cocoa development (replacing Objective-C)
- Existing Cocoa frameworks are all C/Objective-C
  - Swift designed to allow easy bridging between languages
  - Uses Objective-C runtime under the hood

Everything's an Object

There are three object types in Swift:

- Class
- Struct
- Enum

Even literals — for example the numeric literal `42` — are objects (in this case, an instance of a struct).

```
42.advancedBy(7)   // returns 49
```

You can even customize the behavior of fundamental types:

```
extension Int
{
    func printAsAge()
    {
        print("I'm \(self) years old")
    }
}

// Adds printAsAge method to the Int type


42.printAsAge()

// prints "I'm 42 years old"
```

Variable Declarations

Variable declarations must provide either explicit type information, or an explicit initial value.

```
var width: Int    // defines variable 'width' of type Int
var height = 12   // defines 'height' with inferred type Int
```

Note that the compiler will trap any attempts to use a variable before it has been initialized, as shown in the following example:

```
var count: Int
count++           // Compiler error.
```

You can declare several variables of the same type in a single statement:

```
var x, y, z: Int
var i = 0, j = 0
```

Constant Declarations

Constants declarations are similar to variable declarations, except they use the keyword `let` instead of `var`. However, constants must always be declared with an initial value.

```
let width: Double = 8.0  // declares constant of type Double
let height = 5.0  // defines constant with inferred type Double
```

The compiler will trap any attempt to modify the value of a constant.

```
let pi = 3.14159
pi += 2.0          // Compiler error.
```

Printing Text and Values

The Swift Standard Library provides `print` function for printing text and interpolated values:

```
print("Have a nice day")
// prints "Have a nice day"
```

`print` is a generic function that takes an argument of any type:

```
let message = "Have a nice day"
print(message)
// prints "Have a nice day"
let pi = 3.14159
print(pi)
// prints "3.14159"
```

Values of any type can be interpolated in a string literal with `\()`.

```
let temperature = 72
print("It's currently \(temperature) degrees")
// prints "It's currently 72 degrees"

let scale = "Fahrenheit"
print("It's currently \(temperature) degrees \(scale)")
// prints "It's currently 72 degrees Fahrenheit"
```


Functions and Methods

- Functions and methods share the same syntax

```
// Basic Function Syntax

func display(degrees: Double, scale: String)
{
    print("The temperature is \(degrees)° \(scale)")
}

// defines a function that takes a Double and a String as arguments

display(71.5, scale: Fahrenheit)

// prints "The temperature is 71.5° Fahrenheit"
```

- Parameter names, **temperature** and **scale**, are automatically available inside the function, but name of first parameter not visible when calling the function.

- Swift allows declarations to include *external* parameter name:

```
func display(temperatureInDegrees degrees: Double, scale: String)
{
    print("The temperature is \(degrees)° \(scale)")
}

// external parameter name 'temperatureInDegrees' adds clarity
// to calling code

display(temperatureInDegrees: 72, scale: "Fahrenheit")
```

Generics

- Swift provides rich support for generic types.
  - Used heavily in the standard library.

- Note: the example below uses the `Comparable` protocol as a type qualifier. We haven't introduced protocols yet, but they're conceptually similar to *interfaces* in other languages.
  - The `T` in the example is a *placeholder* type name — a stand-in for *actual* type names. Here it simply says that the arguments and return value must all be the same type.
  - The `<T: Comparable>` (called a *type constraint*) means that `T` only matches types that conform to the `Comparable` protocol.

```
func maxValue<T: Comparable>(x: T, y: T) -> T
{
    return x > y ? x : y
}

// defines a function that takes two Comparable values and returns the
// larger of the two


maxValue(22.5, 23)          // returns 23
maxValue(3, 2.9)            // returns 3
maxValue("Apple", "Banana")  // returns "Banana"
```


Tuples

- A tuple is a list of objects of any type.
  - You *create* a tuple with a pair of parens enclosing a comma-separated list of *values*.
  - You *declare* a tuple with a pair of parens enclosing a comma-separated list of *types*.

```
let vals = (12, "Hi")
// defines vals as a tuple containing two values, the first of type Int,
// and the second of type String
```

```
let explicitlyTypedVals: (Int, String) = (12, "Hi")
// explicit type information isn't mandatory above because the compiler
// can infer the type from the initializer value
```

- You can access elements based on their index by using the dot operator:

```
print(vals.0)   // prints "12"
print(vals.1)   // prints "Hi"
```

- Tuple elements can also be named, and then accessed by either name or index:

```
let vals2 = (x: 12, y: 24)
print(vals2.x)  // prints "12"
print(vals2.y)  // prints "Hi"
```

Tuples As Types

- You can use tuples in type declarations, for example as a function parameter or return value.

```
func area(size: (width: Int, height: Int)) -> Int
{
    return size.width * size.height
}
// declares a function that takes one argument, 'size', of type (Int, Int)


func calculateDiscount(originalPrice: Double, percentage: Double)
    -> (price: Double, discount: Double)
{
    let amount = originalPrice * percentage
    let price = originalPrice - amount

    return (price, amount)
}
// declares a function that returns an argument of type (Double, Double)

let (price, discount) = calculateDiscount(25.00, 0.15)
// defines two constants of type Double, 'price' and 'discount', initialized
// with the values of a tuple returned by the call to 'calculateDiscount'

print(price)     // prints "21.25"
print(discount)  // prints "3.75"

// use the special parameter name '_' for any values you wish to ignore

let (discountedPrice, _) = discount(19.95, 0.15)
// captures the price, ignoring the discount amount

print(discountedPrice)  // prints "21.25"
```

For Statements

- Swift supports two types of for loop constructs:
    - classic, three-expression syntax *(deprecated)*
    - **for–in** syntax

```
for var index = 0; index < 3; index++
{
    print("index is \(index)")
}

// index is 0
// index is 1
// index is 2
```

- The example above can be rewritten with for-in syntax

```
for index in 0...2
{
    print("index is \(index)")
}
// produces output identical to previous example
```

- The for–in syntax also works with collections:

```
let names = [ "Jane", "Bill", "Jan" ]

// you can use a for–in loop to enumerate the array defined above

for name in names
{
    print("name is \(name)")
}
// name is Jane
// name is Bill
// name is Jan
```

enumerate Method

- `enumerate` method sequences through a collection's elements.
    - Returns a tuple containing the index of the current element and the value at that index.

```
let names = [ "Jane", "Bill", "Jan", "Pat" ]
// defines an array of elements of type String


for (index, value) in names.enumerate()
{
    print("name \(index + 1) is \(value)")
}
// enumerates the 'names' array, printing the following:
// name 1 is Jane
// name 2 is Bill
// name 3 is Jan
// name 4 is Pat
```

Lab 1: Temperature Conversion

1. Write a function to convert Fahrenheit to Celsius
    - 1.1. Subtract 32 and multiply by 5/9.
    - 1.2. Write a unit test that passes in several different values and prints the results.
2. Write a function to convert Celsius to Fahrenheit
    - 2.1. The algorithm is the inverse of the one used in Step 1.
    - 2.2. Write a unit test that passes in several different values and prints the results.

Intentionally left blank

CHAPTER TWO

Swift Types

Structs

- Value types — can be allocated and passed by value; i.e., copied on stack.
- Declared with **struct** keyword, followed by curly braces
- Can contain properties, methods, and initializers (constructors)

```
struct Dog {
    var name = "Unknown"
    func bark() {
        print("Woof, woof!")
    }
}

// Declares Dog to be a struct with a 'name' property and 'bark' method.
```

```
var rover = Dog()      // allocates a Dog instance
rover.name = "Rover"   // modifies the name property (conceptually;
                       // actually, replaces rover with a new instance)
```

```
print("Name of \(Dog.self) is \(rover.name)")

// prints "Name of MyProj.Dog is Rover". ('MyProj' is name of Swift module.)

rover.bark()

// prints "Woof, woof!"
```


Stored Properties

- Value stored internally in struct, class or enum instance.
- **Properties only declare accessors**; actual storage is opaque.
- Compiler requires stored properties to be initialized.
    - Properties with no *default value* must be initialized in `init`.
    - Swift provides a *memberwise initializer* for read-write properties.
    - If there's no custom initializer, and all properties have default values, Swift provides a *default (no-arg) initializer*.

```
struct Point
{
    var x = 0.0
    var y = 0.0
}

// declares a struct with properties that all have default values


let p = Point()

// calls default initializer, which returns a point with origin 0.0, 0.0


let p2 = Point(x: 10.0, y: 20.0)

// calls memberwise initializer, returning a point with origin 10.0, 20.0
```

Custom Initializers

- Default values can also be provided in an `init` method's parameter list.
  - Parameters that have default values are not required in calls to initializers.

```
struct Point
{
    var x = 0.0
    var y = 0.0

    let pointsPerPixel: Double

    init(x: Double, y: Double, pointsPerPixel: Double = 2.0)
    {
        self.x = x
        self.y = y
        self.pointsPerPixel = pointsPerPixel
    }
}

// pointsPerPixel not required to be passed as argument to initializer


let p = Point(x: 10.0, y: 20.0)
let p2 = Point(x: 10.0, y: 20.0, pointsPerPixel: 3.0)

// both the above calls succeed
```

Computed Properties

- No stored value
    - Value is computed whenever the property is accessed.
    - Can be read-only or read-write.

```
struct Dog
{
    var name: String
    var toys: [String]

    // read-only computed property
    var favoriteToy: String {
        get { return toys.isEmpty ? "N/A" : toys[0] }
    }

    // ...
}

var rover = Dog(name: "Rover", toys: ["Ball", "Rope", "Frisbee"])
print("favorite toy: \(rover.favoriteToy)")

// prints "favorite toy: Ball"
```

```
struct Dog
{
    // ...

    // read-write computed property
    var toyNames: String {
        get { return toys.joinWithSeparator(", ") }
        set { toys = newValue.characters.split {
                         ", ".characters.contains($0).map { String($0) }} }
    }

    // ...
}

rover.toyNames = "Kong, Ball, KittyKat"
print("toys: \(rover.toys)")

// prints "toys: [Kong, Ball, KittyKat]"
```

Property Observers

- Can be used with anything declared as `var`, as long as it has an explicit type and initial value.
  - `willSet` automatically passed `newValue`
  - `didSet` automatically passed `oldValue`

```
var text: String = "Hello" {
    didSet {
        print("Set text to \(text), old value was \(oldValue)")
    }
}
// defines variable 'text' and implements 'didSet' observer

text = "Bye"
// prints "Set text to Bye, old value was Hello"

var number: Int = Int(42) {
    willSet {
        print("About to set number to new value \(newValue)")
    }
    didSet {
        print("Number is now \(number); old value was \(oldValue)")
    }
}
// defines variable 'number' and implements property observers

print(number)
// prints 42

number = 5
// prints "About to set number to new value 5"
//        "Number is now 5; old value was 123"
```

Methods

- Syntax the same as for functions, but calling semantics slightly different
    - For methods with multiple parameters, all but the first param name are *external* by default

```
// here, suffix and numberOfTimes have external names, prefix does not
func barkWithPrefix(prefix: String, suffix: String, numberOfTimes: Int)
{
    for _ in 1...numberOfTimes {
        print("\(prefix), \(barkText) \(suffix)")
    }
}

fido.barkWithPrefix("Grrr", suffix: "(pant, pant)", numberOfTimes: 2)
```

- To externalize internal names:

```
// here, all three params have external names
func bark(prefix prefix: String, suffix: String, numberOfTimes: Int)
{
    // ...
}

fido.bark(prefix: "Grrr", suffix: "(pant, pant)", numberOfTimes: 2)
```

- You can also specify explicit external names

```
// here, the external name for numberOfTimes is repetitions
func woof(prefix prefix: String, suffix: String, repetitions numberOfTimes:
Int)
{
    // ...
}

fido.woof(prefix: "Grrr", suffix: "(pant, pant)", repetitions: 2)
```

Lab 2: Rectangle

1. Declare three structs, as follows:

    1.1. A **Point** structure with properties **x** and **y** of type Double.

    1.2. A **Size** structure with properties **width** and **height** of type Double.

    1.3. A **Rectangle** structure with properties **origin** and **size** of type **Point** and **Size**, respectively.

    1.4. Add a computed property in **Rectangle** that returns the area of the rectangle as a Double.

    1.5. Add another computed property in **Rectangle** that returns a **Point** that represents the rectangle's center.

    1.6. Add a method to **Rectangle** that takes two parameters, **dx**, and **dy**, and returns a rectangle instance offset from the original by the amount of the arguments.

    1.7. Write unit tests to test the functionality developed in the previous steps.

Classes

- Reference types — typically allocated in heap and passed by reference.
- Similar to structs, but with additional capabilities, such as inheritance
- Declared with **class** keyword — optionally followed by colon and name of superclass — followed by curly braces
- Can contain properties, methods, and initializers/deinitializers (constructors/destructors)

```
class Animal {
    var isPet: Bool = false
}
// Declares Animal class

class Dog: Animal {
    var name: String = ""

    func bark() {
        print("Woof, woof!")
    }

    func description() -> String {
        return "name: \(name), is pet: " + (isPet ? "yes" : "no")
    }
}
// Declares Dog to be a subclass of Animal
```

```
var rover = Dog()       // allocates a Dog instance
rover.name = "Rover"    // modifies name
rover.isPet = true;     // modifies isPet

print(rover.description())

// prints "name: Rover, is pet: yes"
```

Type Properties and Methods

- In addition to *instance* properties and methods, structs, classes, and enums can have *type* properties and methods.
    - Declare with **static** keyword
    - Classes can have methods declared with **class** keyword (note: unlike static methods, class methods are inherited)
    - Swift 2 supports **class** type qualifier for class properties

```
class Person: Serializable, CustomDebugStringConvertible
{
    static var defaultAge: Int = 21

    // ...

    convenience init(firstName: String?, lastName: String?)
    {
        self.init(firstName: firstName,
                  lastName: lastName,
                  age: Person.defaultAge)
    }


    class func fetchPeople(path: String) -> [Person]
    {
        // Fetch people somehow...
    }

    // ...
```


Extensions

- Add methods and read-only properties to existing structs, classes, protocols (in Swift 2), and enums
- Declared with **extension** keyword, followed by the name of the type being extended, followed by curly braces
  - Can specify additional protocols

```
extension Dog {

    var numberOfLegs: Int { return 4 } // Computed property (read-only)

    func howl() {
        print("Awoooooooh!")
    }

    func growl() {
        print("Grrrrrr!")
    }
}
// Adds 'howl' and 'growl' methods to the Dog class
```

```
var fido = Dog()

fido.howl()                 // prints "Awoooooooh!"
fido.growl()                // prints "Grrrrrr!"
print(rover.numberOfLegs)   // prints "4"
```

Protocols

- Provide a means to share method and property *declarations* among structs, classes of various types
- Similar to **interfaces** in other languages, but allows methods to be declared as optional *(NOTE: Swift 2 protocol extensions can also provide* implementations *of protocol methods)*
- Declared with **protocol** keyword

```
protocol Likeable {

    var numberOfLikes: Int { get set } // property declaration

    // method declarations
    func like()
    func unlike()
}
```

```
class Person: Likeable {

    // ...

    var numberOfLikes = 0

    // ...

    func like() {
        numberOfLikes++;
    }

    func unlike() {
        if (numberOfLikes > 0) {
            numberOfLikes--
        }
    }
```


CustomDebugStringConvertible

- Protocol defining a single method, `debugDescription`.
- Used by the LLDB debugger to obtain a formatted description of an object for presentation in the console.

```
class Person: Likeable, CustomDebugStringConvertible
{
    // ...

    var debugDescription: String {
        return "\(firstName) \(lastName), +\(numberOfLikes)"
    }
}
```

```
let fred = Person(firstName: "Fred", lastName: "Smith")
print(fred)
fred.like()
print(fred)
fred.like()
print(fred)
fred.unlike()
print(fred)
fred.unlike()
print(fred)
fred.unlike()
print(fred)

// prints the following on the console:
// Fred Smith, +0
// Fred Smith, +1
// Fred Smith, +2
// Fred Smith, +1
// Fred Smith, +0
// Fred Smith, +0
```

The Equatable Protocol

- Declares a generic function that defines the behavior of the **==** operator

- Can be overloaded for custom types

```
// From declaration in Swift Standard Library:

protocol Equatable {
    func ==(lhs: Self, rhs: Self) -> Bool
}
```

```
// Overloading for Friendable type

func ==(lhs: Friendable, rhs: Friendable) -> Bool
{
    return lhs.friendID == rhs.friendID
}
```

```
func testEquatable() {
    let p1 = Person("Fred", "Smith", 100)
    let p2 = Person("Fred", "Smith", 100)
    let p3 = Person("Fred", "Smith", 99)

    print("p1 == p1 is \(p1 == p1)")
    print("p1 == p2 is \(p1 == p2)")
    print("p1 == p3 is \(p1 == p3)")

    print("p1 === p1 is \(p1 === p1)")
    print("p1 === p1 is \(p1 === p2)")
}

// prints the following on the console:
p1 == p1 is true
p1 == p2 is true
p1 == p3 is false
p1 === p1 is true
p1 === p1 is false
```

Numeric Types

- Swift Standard Library declares protocols such as **IntegerType**, **SignedIntegerType**, etc. that define requirements for integer types.
- Protocols used in declaring structs to represent various integer types, such as **Int**, **Int8**, **UInt8**, **Int16**, **UInt16**, etc.
- Similarly, protocol **FloatingPointType** is adopted by both **Float** and **Double**.
- You can use constructors to convert between dissimilar types.

```
let x = 42
let y: Float = x          // Illegal!
let z: Float = Float(x)   // This works fine
let easier = Float(x)     // This works fine too
let backToInt = Int(z)    // Yep
```


Lab 3: Classes and Protocols

1. Declare a class, **Person** that has two properties of type String named **firstName** and **lastName** with no default values.
    1.1. Write an initializer that takes a first and last name as arguments.
    1.2. Implement **debugDescription** to return the Person instance's first and last names and number of likes.
    1.3. Write a unit test to verify that you can instantiate and print a Person.
2. Declare a protocol named, **Likeable**, with a read-write property named **numberOfLikes** of type Int, and functions named **like** and **unlike** that take no arguments and have no return value. Then modify Person to adopt the **Likeable** protocol as follows:
    2.1. Add a **numberOfLikes** property with a default value of **0**.
    2.2. Add an extension that implements the **like** and **unlike** methods to increment and decrement **numberOfLikes**, but ensure that it never falls below zero.
    2.3. Modify **debugDescription** to include the number of likes in the string it returns.
    2.4. Write a unit test to verify that the new Likeable behavior works correctly.
3. Declare a protocol named **Friendable**, with read-only properties **friendID** of type Int, and **friends** of type array of **Friendable**, and functions **friend** and **unfriend**, both of which take a single argument of type **Friendable** and have no return value, Then modify Person to adopt the **Friendable** protocol as follows:
    3.1. Add a **friendID** property with a default value of **0**.
    3.2. Add a **friends** property that has an empty array as its default value.
    3.3. Add an extension that implements the **friend** function to add the provided friend to the **friends** array, and the **unfriend** function to remove the provided friend from the array.
    3.4. Write a unit test to verify that a Person can successfully friend and unfriend other Friendables.

Strings and Characters

- Ordered collection of **Character**
    - Each character represents a Unicode character.
    - Width of Unicode characters can vary, therefore indexes of characters are computed by enumerating the string.

```
let emojiText = "Hello World! 😊🌍"
count(emojiText)            // returns 15

let foundationEmojiText: NSString = emojiText
foundationEmojiText.length  // returns 17
```

- Working with String properties:

```
let text = "Hello World!"

text.isEmpty              // prints false

text.lowercaseString      // prints "hello world!"

text.uppercaseString      // prints "HELLO WORLD!"

text.hasPrefix("Hello")  // prints true

text.hasSuffix("World")  // prints false
```

String Methods

- Working with String methods:

```
let fruits = [ "Apple", "Pear", "Banana"]
let fruitString = fruits.joinWithSeparator(", ")
print(fruitString) // prints "Apple, Pear, Banana"

// Breaking split down into separate steps for clarity...
func isSeparator(character: Character) -> Bool {
    return ", ".characters.contains(character)
}
let elements = fruitString.characters.split(isSeparator: isSeparator)
let substrings = elements.map { String($0) }
print(substrings)     // prints "[Apple, Pear, Banana]"


let text = "Hello World!"

text.characters.count //prints 12
let hasH = text.characters.contains("H")    // arg is character, not string
print(hasH)          // prints true
let hasVowels = text.characters.contains({ "aeiou".characters.contains($0) })
print (hasVowels)    // prints true
let location = text.characters.indexOf("W") // arg is character, not string
print(location)      // prints "Optional(6)"
```

- Working with ranges:

```
for currChar in text {
    print(currChar)
}
// prints each character in the string, one per line

let firstChar = text[text.startIndex]
let secondChar = text[text.startIndex.successor()]
let thirdChar = text[text.startIndex.advancedBy(2)]

let startIndex = text.startIndex.advancedBy(6)
text[startIndex] // returns "W"

let endIndex = text.endIndex.advancedBy(-1)
text[endIndex]  // returns "!"

let range1 = Range(start: startIndex, end: endIndex)
text[range1] // returns "World"

let range2 = startIndex ..< endIndex
text[range2] // returns "World"
```

String Ranges

- A string's characters can be enumerated like an array:

```
for currChar in text.characters {
    print(currChar)
}
// prints each character in the string, one per line
```

- Use ranges to specify portions of a string:

```
let startIndex = text.startIndex.advancedBy(6)
text[startIndex] // returns "W"

let endIndex = text.endIndex.advancedBy(-1)
text[endIndex]   // returns "!"

let range1 = Range(start: startIndex, end: endIndex)
text[range1]     // returns "World"

// You can use a Range literal to specify a range:
//
let range2 = startIndex ..< endIndex  // Half-open range
text[range2]     // returns "World"

let range3 = startIndex ... endIndex  // Closed range
text[range3]     // returns "World!"
```

Bridging to Foundation

- A number of Swift library structs, including String, Array, Dictionary, Set, and Error, are bridged to equivalent Foundation classes.

- For example, String is bridged to the Foundation **NSString** and **NSMutableString** classes.

```
let foundationStr: NSString = text
foundationStr.length                     // 12
foundationStr.substringFromIndex(6)      // "World!"
foundationStr.substringToIndex(5)        // "Hello"

let range = foundationStr.rangeOfString("World") // (6, 5)
foundationStr.substringWithRange(range)          // "World"

let fruitText: NSString = fruitString
let fruits2 = fruitText.componentsSeparatedByString(", ")
// ["Apple", "Pear", "Banana"]
```

- Foundation classes have many powerful features; many have the ability to serialize and deserialize from a file system representation:

```
fruitText.writeToFile("fruit.txt", atomically: true,
    encoding: NSUTF8StringEncoding, error: nil)

// creates a file named 'fruit.txt' and populates it with the
// contents of the string referenced by fruitText.

let clonedFruitText = NSString(contentsOfFile: "fruit.txt",
    encoding: NSUTF8StringEncoding, error: nil)

// creates a new string populated with the contents of the file 'fruit.txt'.
```


String Formatting

- Strings can be initialized with a printf-style format string and variable length argument list (variadic):

- Arguments from variadic list are interpolated in place of *format specifiers*.

```
let foo: NSString = "Foo"
// Uses %d format specifier for Int length
let s1 = String(format: "foo's length is %d", foo.length)
// prints "foo's length is 3"


let fahrenheit = 78.5
// Uses %.1f format specifier for Double value, where the '.1'
// specifies one digit of decimal precision.
let s2 = String(format: "temperature is %.1f°F", fahrenheit)
// prints "temperature is 78.5°F"
```

- See Apple's documentation for a comprehensive list of format specifiers.

Intentionally left blank

CHAPTER THREE

Optionals and Error Handling

The Optional Enum

- **nil** is an illegal value in Swift
- Optionals are wrappers for values that may potentially be **nil**

Declaration from the Swift Standard Library

```
public enum Optional<Wrapped> : _Reflectable, NilLiteralConvertible {
    case None
    case Some(Wrapped)
    /// Construct a `nil` instance.
    public init()
    /// Construct a non-`nil` instance that stores `some`.
    public init(_ some: Wrapped)
    /// If `self == nil`, returns `nil`.  Otherwise, returns `f(self!)`.
    @warnunusedresult
    public func map<U>(@noescape f: (Wrapped) throws -> U) rethrows -> U?
    /// Returns `nil` if `self` is nil, `f(self!)` otherwise.
    @warnunusedresult
    public func flatMap<U>(@noescape f: (Wrapped) throws -> U?) rethrows -> U?
    /// Create an instance initialized with `nil`.
    public init(nilLiteral: ())
}

extension Optional : CustomDebugStringConvertible {

    /// A textual representation of `self`, suitable for debugging.
    var debugDescription: String { get }
}
```

Using Optionals

- You can use a question mark to declare a variable as optional:

```
var name: String?
// declares name to be an optional string

if name == nil {
    print("name is nil")
}
// prints "name is nil"

name = "Fred"
```

- You can use an exclamation point to unwrap an optional. This is called *forced unwrapping*.
    - As the name implies, it's less safe than other approaches.

```
if name != nil {
    print("name is \(name)")
    // prints "name is Optional("Fred")"

    print("name is " + name!)
    // prints "name is Fred"
}
```


Optional Binding

- You can safely unwrap an optional using an **if–let** statement.
  - If the assignment succeeds, the **let** constant contains the unwrapped value.

```
var name: String? = "Fred"
if let someName = name {
    print("name is " + someName)
}
// prints "name is Fred"

func formattedName1(name: String?) -> String {
    if let someName = name {
        return "name is \(someName)"
    } else {
        return "name unknown"
    }
}

formattedName("Fred")
// prints "name is Fred"

formattedName(nil)
// prints "name unknown"
```

- A **guard–let** statement is a convenient alternative to **if–let**:

```
func formattedName2(name: String?) -> String {
    guard let someName = name else {
        return "name unknown"
    }
    return "name is \(someName)"
}

formattedName("Fred")
// prints "name is Fred"

formattedName(nil)
// prints "name unknown"
```

Guard vs. the Pyramid of Doom

- Consider the 'pyramid of doom' style of nested **if–let** statements in the following example:

```
func format1(person: Person?) -> String
{
    if let p = person {
        if let name = p.fullName {
            if let ageStr = p.age, let age = Int(ageStr) {
                return "\(name), age: \(age)"
            } else { return name }
        } else { return "missing name" }
    }
    return "person cannot be nil"
}
```

- Here's the previous example rewritten using **guard–let** statements:

```
func format2(person: Person?) -> String
{
    guard let p = person else {
        return "person cannot be nil"
    }
    guard let name = p.fullName else {
        return "missing name"
    }
    guard let ageStr = p.age, let age = Int(ageStr) else {
        return name
    }
    return "\(name), age: \(age)"
}
```

Guard Statements

- Guard statements can be used with ordinary logic expressions:

```
func divide1(numerator: Int, denominator: Int) -> Int?
{
    guard denominator != 0 else {
        print("Zero divide")
        return nil
    }

    return numerator / denominator
}
```

- Control flow keywords such as **if**, **guard**, and **for** can also be used in combination with the **case** keyword:

```
enum Pet { case Dog, Cat, Bird, Skunk }

func showPet(person: Person)
{
    if case .Skunk = person.favoritePet {
        print("Skunks, seriously?")
    }

    guard case .Dog = person.favoritePet else {
        print("Okay, but you do like dogs, right?")
        return
    }

    print("Dogs, FTW!")
}
```


Where Clauses

- Used in combination with flow control statements.

```
let fred = Person(fullName: "Fred Smith", age: "42")

if let name: NSString = fred.fullName where name.hasPrefix("Fred") {
    print("Fred's full name is \(name)")
}
// prints "Fred's full name is Fred Smith"


guard let name = fred.fullName where name.characters.count > 1 else {
    return "must have name with length > 1"
}
// early return when fred.fullName is nil or contains less than 2 characters


let numbers = Array(1...5)

for value in numbers where value % 2 != 0 {
    print(value)
}
// prints:
// 1
// 3
// 5
```

Error Handling

- Swift throws errors rather than exceptions. Errors are defined by enums conforming to the ErrorType protocol.

```
enum MathError: ErrorType {
    case ZeroDivide
    case Overflow
}
```

- Methods and functions that throw must be annotated with the **throws** keyword

```
func divide2(numerator: Int, denominator: Int) throws -> Int? {
    if denominator == 0 {
        throw MathError.ZeroDivide
    }
    return numerator / denominator
}
```

- Each call to a method or functions that throws must be prefixed with **try?**, **try!**, or else **try** inside a **do–catch** block.

```
let x = try! divide2(12, denominator: 3)
// produces 4

if let x = try? divide2(12, denominator: 3) {
    print(x)
}
// produces Optional(4)


do {
    let result = try divide2(42, denominator: 0)
    print("result is \(result)")
}
catch MathError.ZeroDivide {
    print("Zero divide")
}
// prints "Zero divide"
```

- Note that thrown errors don't unwind the stack; instead they trigger an early return that incorporates the error value.

Implicitly Unwrapped Optionals

- Use exclamation point to qualify types you want *implicitly unwrapped*.
    - The resulting optional can then be used as if it were unwrapped.
    - The developer is responsible for ensuring that the value will not be directly accessed when **nil** (other than testing for nil).

```
let lastName: String! = "Smith"
// lastName contains an optional that wraps the string "Smith"

print("name is \(lastName)")
// prints "name is Smith"
```


??, The Nil Coalescing Operator

- `??` is the *nil coalescing operator*. (Confused yet?)
- Combines ternary expression and optional unwrapping.

```
let special: Double? = nil

let discount = special ?? 0.15

// assigns 0.15 to discount if special is nil,
// otherwise assigns unwrapped value of special


// the nil coalescing operator in the example above is
// essentially just shorthand for the following:

let discount = special == nil ? 0.15 : special!
```

- Can be used as part of a larger expression:

```
let Unknown = "Unknown"
let firstName: String? = "Fred"
let lastName: String? = "nil"

var description: String {
    return "first: \(firstName ?? Unknown), "
        +  "last: \(lastName ?? Unknown)"
}

print(description)

// prints "first: Fred, last: Unknown"
```

Optional Chaining

- Combines optional unwrapping and property access.

```
var person: Person? = Person(firstName: "Fred", lastName: nil, age: 29)
person!.dog = Dog(name: "Rover", toys: ["Ball", "Frisbee"])

// creates an optional Person with an optional Dog


print(person?.firstName)

// prints "Optional("Fred")


print(person?.dog?.toys)

// prints "Optional(["Ball, "Frisbee"])
```


Casts

- Check type with is, is?, or is!
- Downcast with as, as?, or as!

```
let words: [Any] = ["Hello", "World"]
let word: String = words[0] as! String   // force downcast from Any to String

// prints "Hello"


var things: [Any?] = [42, "Hello", 3.5]

if let answer: Int? = things[0] as? Int {  // downcast to optional Int
    print(answer)
}

// prints 42



var objects: [Any] = [42, "Fred", 3.5]
for object in objects {
    if object is Int    { print("The answer is \(object)") }
    if object is String { print("Hi \(object), how are you?") }
}

// prints "The answer is 42"
//        "Hi Fred, how are you?"



for object in objects {
    switch (object) {
    case let value as Int:
        print("The answer is \(value / 6)")
    case let value as String:
        print("Hi \(value), have a nice day!")
    case let value as Double:
        print("It's \(value)°F? Brr, that's cold.")
    default:
        print("The object is \(object)")
    }
}

// prints "The answer is 7"
//        "Hi Fred, have a nice day!"
//        "It's 3.5°F? Brr, that's cold."
```

CHAPTER FOUR

Collections and Closures

Arrays

- Ordered generic collection whose elements can be accessed by index using subscript notation.

- Swift provides syntax for defining literal arrays that can also be used in to provide type information in declarations.

```
var myArray: [Int]
// defines myArray as Array of Int
// shorthand for generic data type:
//
var myArray: Array<Int>

// Using an array literal:
let words = ["one", "two", "three"]
print(words)
// prints "[one, two, three]"
```

- The Array type has a compact public API that provides powerful features. Examples of a few of the simpler methods:

```
let reversedWords = words.reverse() as Array
print(reversedWords)
// prints "[three, two, one]"

var moreWords = words
moreWords.insert("two and a half", atIndex: 2)
print(moreWords)
// prints "[one, two, two and a half, three]"

var edibles = ["apple", "pear", "banana"]
edibles.appendContentsOf(["onion", "carrot", "celery"])
print(edibles)
// prints ["onion", "carrot", "celery", "apple", "pear", "banana"]

edibles.sortInPlace { $0 < $1 }
print(edibles)
// prints "[apple, banana, carrot, celery, onion, pear]"

let sortedEdibles = edibles.sort { $0 > $1 }
print(sortedEdibles)
// prints "[pear, onion, celery, carrot, banana, apple]"
```

Dictionaries

- Keyed generic collection whose elements can be accessed by key using subscript notation.
    - Keys must be instances of types that conform to the **Hashable** protocol

- Using a dictionary literal:

```
var info = [
    "name": "Fred",
    "email": "fred@foo.com",
    "age": 37
]

print(info)
// prints "[email: fred@foo.com, age: 37, name: Fred]"

print(info["name"])
// prints "Optional(Fred)"
```

- Dictionary access methods return optionals:

```
if let name = info["name"] as? String {
    print(name)
}
// prints "Fred"

let phoneKey = "phone"
if let phone = info[phoneKey] as? String {
    print(phone)
} else {
    print("No value for key \(phoneKey)")
}

// the above if-let-else prints "No value for key phone",
// but after executing the following line would print "703-321-1234"

info[phoneKey] = "703-321-1234"
```

Closures

- Swift functions can be passed as arguments and can be used as return values.
    - General form of closure syntax:

```
{ (parameters) -> return_type in
    // statements
}
```

- Example — passing a named function:

```
let fruit = ["Pear", "Apple", "Peach", "Banana"]

func ascending(s1: String, s2: String) -> Bool {
    return s1 < s2
}

let sortedFruit = fruit.sort(ascending)
// value is ["Apple", "Banana", "Peach", "Pear"]
```

- Examples of syntactic flexibility:

```
// passing anonymous closure
let sortedFruit2 = fruit.sort({ (s1: String, s2: String) -> Bool in
    return s1 < s2
})

// anonymous closure with streamlined syntax
let sortedFruit3 = fruit.sort({ s1, s2 -> Bool in
    return s1 < s2
})

// trailing closure syntax
let sortedFruit4 = fruit.sort { s1, s2 -> Bool in
    s1 < s2
}

// trailing closure with positional parameters
var sortedFruit5 = fruit.sort { $0 < $1 }


// passing '<' function
var sortedFruit6 = fruit.sort(<)
```

Working with Closures

- Executing a closure

```
let WavyDash = UnicodeScalar(0x03030)

func showDate(completion: () -> Void)
{
    print("The current date and time is now \(NSDate())")
    completion() // Executes the closure
}

showDate {
    let wavyLine = String(count: 7, repeatedValue: WavyDash)
    print("\(wavyLine)" + " MESSAGE " + "\(wavyLine)")
}

// prints:
//
// The current date and time is now 2015-07-19 15:56:23 +0000
// 〰〰〰〰〰〰〰 MESSAGE 〰〰〰〰〰〰〰
```

- Closures automatically capture and store references to constants and variables from the enclosing scope.

```
func growl(numberOfTimes: Int, performGrowl: () -> Void)
{
    for count in 1...numberOfTimes {
        print("\(count) of \(numberOfTimes): ")
        performGrowl()
    }
}

let n = 3
growl(n) {
    print("Grrr!")
}

// prints the following...
//
// 1 of 3: Grrr!
// 2 of 3: Grrr!
// 3 of 3: Grrr!
```


map

- The **map** method operates on collections by applying a closure to each element successively.
  - Returns an array of the closure's return values.

```
let fruits = ["apple", "pear", "banana"]
let capitalizedFruits = fruits.map { $0.capitalizedString }
print(capitalizedFruits)
// prints "[Apple, Pear, Banana]"
```

- Powerful way to work with arrays of complex objects:

```
struct Grocery {
    let name: String
    let price: Double
    let quantity: Int
}

let groceries = [
    Grocery(name: "Apples", price: 0.65, quantity: 12),
    Grocery(name: "Milk", price: 1.25, quantity: 2),
    Grocery(name: "Crackers", price: 2.35, quantity: 3),
]

let costs = groceries.map {
    // a tuple containing current item's name and cost
    ($0.name, $0.price * Double($0.quantity))
}

print(costs)
// prints "[(Apples, 7.8), (Milk, 2.5), (Crackers, 7.05)]"
```


reduce

- The **reduce** method operates similarly to **map**, but instead of returning an array, returns a single value.
    - First argument is initial value
    - Second argument is a closure that returns the value of the current item in the collection combined with the current sum.

```
let ints = [1, 2, 3, 4, 5]

let sum = ints.reduce(0) {
    currSum, currVal in        // parameter list
    return currSum + currVal
}
// produces 15

// same as the above example, but with streamlined syntax:
let sum2 = ints.reduce(0) { $0 + $1 }
// ditto
let sum3 = ints.reduce(0, combine: +)

// additional examples:

let factorial = ints.reduce(1, combine: *)
// produces 120

let fruitsText = fruits.reduce("favorites: ") { "\($0)\($1), " }
// produces "favorites: apple, pear, banana, "
```


reduce (Continued)

- As with **map**, provides a powerful way to work with arrays of complex objects:

```
let total = groceries.reduce(0.0) { sum, item in
    sum + (item.price * Double(item.quantity))
}
// produces 17.35
```

- Applying **reduce** to array of tuples from an earlier **map** example:

```
let string = costs.reduce("Costs") { text, item in
    text + "name: \(item.0), cost: \(item.1)\n"
}

print(string)
// name: Apples, cost: 7.8
// name: Milk, cost: 2.5
// name: Crackers, cost: 7.05


// fancier version of preceding example:

let text = costs.reduce("Costs\n==============\n") {
    $0 + String(format: "%8s%6.2f\n",
         NSString(string: $1.0).UTF8String, $1.1)
}

print(text)
Costs
==============
  Apples  7.80
    Milk  2.50
Crackers  7.05
```


filter

- The **filter** method returns an array of objects that match the condition specified by its closure argument.

```
let people = [
    Person(("Fred", "Smith", 27)),
    Person(("Janet", "Wade", 31)),
    Person(("Gale", "Dee", 42)),
    Person(("Jan", "Grey", 29)),
]

let under30 = people.filter { $0.age < 30 }
// produces an array containing Fred Smith and Jan Grey
```

<<<<>>>>

