Swift Beyond the Basics

Topics

- Optionals
- Tuples
- Objects
  - Enum
  - Struct
  - Class
- Closures/Lambdas
- Protocols
- Extensions

Optionals
(Kotlin Nullables)

Optionals (Kotlin Nullables)

- A **safe** mechanism to express *presence* or *absence* of data
  - Unsafe: Java null => NullPointerException!
- Declare optional variables using `Optional<typename> or typename?`
- Three ways to put values into optionals
  - Uninitialized: implicit absence
  - Initialize to `.none`: explicit absence
  - Initialize to data of valid type: presence of data
- Internally, Swift uses `nil` to indicate absence of data

```
var result: Optional<Float>
// var result: Float?
var userName: String? = .none

// Assign value like ordinary var
result = 3.14
userName = “swiftybird”
result = nil
result = .none
```

Practical Use Cases of Optional

Use Case Scenario	Function/Variable Declaration
Last login date (but the user may never previously logged in)	`var lastLogin: LocalDate?`
Three-state boolean (**YES/NO NOT-YET**)	`var agreeTermOfServices: Bool?`
Find the largest value in an array (but the array may be empty)	`func largestValue(_ arr: [Float]) -> Float?`
Find the most repeated characters in a String (but the string may not have any)	`func mostRepeatedChar(s: String) -> Character?`

Test for Presence of Value

```
var userName: String? = .none

// userName = “swiftybird”
```

```
if let u = userName {
  print(“User is \(u)”)
} else {
  print(“User is absent”)
}
```

Swift 5.7

```
if let userName {
  print(“User is \(userName)”)
} else {
  print(“User is absent”)
}
```

```
// Kotlin .let

var userName:String? = null
userName?.let { u ->
  println (“User is $u”)
}

userName?.let {
  println (“User is $it”)
}
```

Unwrap (!) to access optional value

```
let userSecret: Optional<Int> = 123456
// OR: let userSecret: Int? = 123456
let loginCode: Int = 391176

// Attempt to do arithmetic between Optional and Int !!!
if userSecret + loginCode > 700_000 { // ERROR: userSecret must be unwrapped
   print (“Code accepted”)
}

if userSecret! + loginCode > 700_000 { // userSecret! yields an Int value
   print (“Code accepted”)
}
// This program will crash IF userSecret is .none (or nil)
```

Invoking Methods on Optionals

- Safe unwrap method calls (?.)

```
var userName: String? = .none

// userName = “swiftybird”
```

```
// u is also String?
let u = userName?.uppercased()
```

- **Forced** unwrap using ! *Use it sparingly*

```
// Program may crash when data is absent
let uRiskIt = userName!.uppercased()
```

Use of "!" with Nullable/Optional Types

Use of !	Example	Purpose

After a nullable variable name	`val prime: Optional<Int> = 17`
	`// Compute square of prime`
	`val sqp = prime! * prime!`
		Unwrap the value inside an Optional

After a type name	`val prime: Int! = 17`
	`// Much later in code`
	`// Compute square of prime`
	`val sqp = prime * prime`
		Automatic unwrap *by the compiler*

*In **both cases**, the program will crash if* `prime` *was nil at the time of multiplication*

Type vs. Type? vs. Type!

	Nullable	Require Manual Unwrapping	What happen if value is nil
Type	No	N/A	N/A
Type?	Yes	Yes	Program crashed
Type!	Yes	No	Program crashed

Type vs. Type? vs. Type!

- Type? requires *explicit* unwrapping *by you (the code developer)*
- Type! enables *implicit* unwrapping by the *compiler*
- For either type, beware of **runtime error when data is absent!!!**

```
func shout1 (s: String) {
  s.append("!!!")
}
```

```
func shout2 (s: String?) {
  s?.append("!!!")
}
```

```
func shout3 (s: String!) {
  s.append("!!!")
}
```

```
shout1("Hello")
shout2("Hello")
shout3("Hello")
shout2(nil)       // OK
shout3(nil)       // crashed!
```

When to use Type!

Use Type! when

- A variable is initially absent of valid data but we know for sure that at the time of its (first) use, a valid data will be present
  - either initialized by our own code later (before first use)
  - or *initialized by something a third-party library or by iOS at runtime*

Using Optional in Comparisons

- When used in comparison, it is unnecessary to unwrap optionals

```
var userName: String? = .none
var cityName: String? = “Dakota”

// No unwrapping required
if userName == “Dakota” { // Evaluates to false

}

// No unwrapping required
if cityName == “Dakota” { // Evaluates to true

}
```

Unwrapping Optionals

```
// Swift
var cityName: String? = “Grand Rapids”
var zipCode: String? = nil

if let c = cityName {
   print(“We are in \(c)”)    // Prints: We are in Grand Rapids
}

if let z = zipCode {
   print(“ZIP \(z)”)          // DOES NOT print
}
```

```
// Kotlin
val cityName: String? = “Grand Rapids”
val zipCode: String? = null

cityName?.let {
   print(“We are in $it”)
}

zipCode?.let { z ->
   print(“ZIP \(z)”
}
```

Unwrapping Multiple Optionals

```
var userName: String?
var cityName: String?
var zipCode: String?

if let u = userName {
  if let c = cityName {
    if let z = zipCode {
      print(u, c, z)    // print only when data present in all the three vars
    }
  }
}
```

Use guard-else

```
var userName: String?
var cityName: String?
var zipCode: String?

guard let u = userName, let c = cityName, let z = zipCode else {
  return // skip print() when data is absent in one of the vars
}
print(u, c, z)
```

Why Use Optionals?

- Interoperability between Objective-C and Swift
  - Allow Swift code to send/receive nil to/from Objective-C
  - All Objective-C objects are handled as Optionals in Swift

Tuples

Tuples

- A lightweight **ordered collection** of multiple values
- Elements can be of different types
- Accessing elements of a tuple
  - Assign multiple values simultaneously
  - Use the dot-number
- Elements of a tuple may have label
  - Access using dot-number or label name

```
let coord = (2.4, 8.3)
let dailyTemp = (“Monday”, 51.5)



let (day, temp) = dailyTemp

print (“On \(dailyTemp.0) the temperature is ”,
  dailyTemp.1)

let dTemp = (day: “Monday”, temperature: 51.5)
print (dTemp.day)
print (dTemp.0)
```

Object in Swift

Object in Swift

- Elements of an object
  - Initializer (constructors)
  - Properties (data fields)
  - Methods
- **Three variants**
  - Enum (value type): for defining a finite set of related values
  - Struct (value type)
  - Class (reference type)
- When assigned to variable or passed to a function
  - The recipient of a value type gets a copy/clone of the instance (enum/struct)
  - The recipient of a reference type gets the reference to the same instance (object) itself

Swift: Value Type vs. Reference Type

	Value Type	Reference type
Characteristics	Each instance keeps a unique copy of its data	Instances share a single copy of the data
Examples	struct, enum, tuple	class
Effect	Copying creates an independent instance with its own copy of its data	Copying implies sharing one common instance

Copying: *the effect of assignment, initialization, argument passing*

Simple Forms of Enums

- A finite set of alternative values

```
enum CompassDirection {
  case North
  case East
  case South
  case West
}

// Or on a single line separated by commas
enum CompassDirection {
  case North, East, South, West
}
```

- Each alternative can be associated with a raw value (String, Character, Int, floating-point)

```
enum CompassWithBearing: Float {
  case North = 0, East = 90,
    South = 180, West = 270
}
let d1 : CompassDirection = .North
let d2 : CompassWithBearing = .East
print(d1)               // North
print(d1.rawValue)      // Compile error
print(d2.rawValue)      // output 90
```

Swift struct
is important in SwiftUI

struct vs. class

```
// A value type
struct Person {
  let name: String = “nobody”
  var age: Int = 0
}

var me = Person(age:22)
let twinOfMe = me // twinOfMe is an indep copy of me

me.age += 5


print(me)         // Person(name: “nobody”, age: 27)
print(twinOfMe)   // Person(name: “nobody”, age: 22)
```

```
// A reference type
class CPerson {
  let name: String = “nobody”
  var age: Int = 0
}

var a = CPerson()
a.age = 22

let b = a   // b and a are references to the same obj
a.age += 5` // updates from a affects what b sees

print(“Age”, a.age)     // Age 27
print(“Age”, b.age)     // Age 27
```

Swift Fiddle (online playground)

Swift Struct vs. Class

	Struct	Class
Define initializer	Yes	Yes
Define properties	Yes	Yes
Define methods	Yes	Yes
Apply extensions	Yes	Yes
Inheritance	No	Yes
Define de-initializer	No	Yes
Reference counting	No	Yes
Kind	Value Type	Reference Type

Struct: Properties, Initializers, and Methods

```
struct Student {
  let name: String
  var grades: [Int]

  init(name:String) {
    self.name = name
    self.grades = []
  }

  mutating func addGrade(val:Int) {
    self.grades.append(val)
  }

  func getGPA() -> Float {
    // calculate GPA from all the grades
  }
}
```

Two properties without default value
Use let for immutable properties

(Designated) Initializer is responsible for initializing all properties without default value

By default, methods can only read properties
Those that alter any properties must be declared as `mutating`

```
// Usage
let steve = Student(name: "Steve Jobs")
steve.addGrade(83)
```

Class: Properties, Initializers, and Methods

```
class Student {
  let name: String
  var grades: [Int]

  init(name:String) {
    self.name = name
    self.grades = []
  }

  func addGrade(val:Int) {
    self.grades.append(val)
  }

  func getGPA() -> Float {
    // calculate GPA from all the grades
  }
}
```

Two properties without default value
Use let for immutable properties

(Designated) Initializer is responsible for initializing all properties without default value

All methods have R/W access to any properties

```
// Usage
let steve = Student(name: “Steve Jobs”)
steve.addGrade(83)
```

Class Property Getter & Setter

```
class Student {
  let name: String
  var grades: [Int]

  init(name:String) {
    self.name = name
    self.grades = []
  }

  func getGPA() -> Float {
    // calculate GPA from all the grades
    ___
  }
}


// Use the function
let me = Student()
print(me.getGPA())
```

```
class Student {
  let name: String
  var grades: [Int]

  init(name:String) {
    self.name = name
    self.grades = []
  }

  val GPA: Float {
     get {
        // calculate grade
        return _____
     }
  }
}

// Use the property
let me = Student()
print(me.GPA)     // Not a function call
```

Class/Struct Initializers

- Must provide `init()` for properties which do not have default value

```
class Timer {
  var hour: Int = 0
  var minute: Int = 0
  var second: Int = 0

  // No init() is required
}
```

```
class Timer {
  var hour: Int
  var minute: Int = 0
  var second: Int = 0

  init() { // Required to initialize self.hour
    self.hour = 0
  }
}
```

Class/Struct: Convenience initializers

- A **designated**/primary `init()` initializes all (non-default) properties
- A **convenience**/secondary `init()` initializes only some (or none) of the properties
- To complete initialization a secondary `init()` must
  - Invoke a primary `init()` or
  - Invoke another secondary `init()`
  - A call to a convenience/secondary `init()` must eventually end up in a designated/primary `init()`

Class/Struct: Convenience initializers

```
class Timer {
  var hour, minute, second: Int
  init(_ h:Int, _ m: Int, _ s: Int) {
    self.hour = h
    self.minute = m
    self.second = s
  }

  convenience init() {
    self.init(0, 0, 0)
  }
  convenience init(numSeconds: Int) {
    var h, m, s: Int
    // compute h, m, s from numSeconds
    self.init(h, m, s)
  }
}
```

Designated/primary initializer

Convenience/secondary initializers

Closures / Lambdas

Closure: a *transportable* and *anonymous* block of code

Closures & Capturing

- Recall that in Swift functions can be passed around like data (into arguments or function return value)
- Usually defined *locally* within the scope of another function
  - A closure may refer to other constants/variables defined in the enclosing function
  - For the closure to execute properly these constants/variables must be **captured** by ("bundled together") the block of code
- Closures = anonymous function + surrounding context/variables
- Closures are a **reference type** (not a value type)
  - When passed around into/from functions, a **reference to the block of code** is passed around and NOT a COPY of the block of code

Four Kinds of Swift Function

Has Name	Captures Constants/Variables	
No	No	Closure expressions
No	Yes	Closure expressions
Yes	No	Global functions
Yes	Yes	Nested functions

Closure Expressions (Anonymous Functions)

```
// Ordinary function
func greet(toName: String) {
  // code for greeting the person
}
```

```
// Closure expression equivalent
let grt = { (toName: String) in
  // code for greeting the person
}
```

```
// Ordinary function
func genPassword(len: Int) -> String {
  // code for generating password here
}
```

```
// Closure expression equivalent
let genPas = {(len: Int) -> String in
  // code for generating password here
}
```

```
// Invoking the “closurized” functions
grt(“Jane”)
let myPass = genPas(12)
```

Capture: Closure & Surrounding Context

- Closures can be stored and “moved” around
  - From one function to another
  - From one scope to another
- Code in a closure may reference variables outside the closure
- Execution of a closure may take place (much) later
- But, when the closure is executed those variables may already be out of scope

Closure Capturing Example

```
typealias MyStrFunc = (String) -> Void

func buildFunction() -> MyStrFunc {
  let secret:String;
  // calculate secrote code here

  let greet = { (name: String) in
    print (“Hi \(name) your secret code is \(secret)”)
  }

  return greet
}
```

```
// Elsewhere in program

let toRun = buildFunction()
toRun(“Bob”)

// Output: Hi Bob your secret code is ________
```

Closure: Usage Example

```
// Without closure
func myStringCheck(_ a: String, _ b: String) {
  return a < b
}

let names = [“brad”, “fred”, “candy”]
let out = names.sorted(by: myStringCheck)
print(out); // [“brad”, “candy”, “fred”]
```

```
let names = [“brad”, “fred”, “candy”]
let out = names.sorted(by: { (a: String, b: String) in
 return a < b
})
print(out);     // [“brad”, “candy”, “fred”]
```

Closure Arguments Shortcuts & Implied Return

```
let names = [“brad”, “fred”, “candy”]
let out = names.sorted (by: { (a: String, b: String) in return a < b })
```

When parameter type can be inferred from context

```
let names = [“brad”, “fred”, “candy”]
let out = names.sorted(by: { /* no (params) in */       return $0 < $1 })
```

When closure body is just a single return

```
let names = [“brad”,  “fred”, “candy”]
let out = names.sorted(by: {  /* no return keyword */          $0 < $1 })
```

Trailing Closures: last argument of a func call

```
let names = [“brad”, “candy”, “fred”]
let out = names.sorted(by: { $0 < $1 })
```

When passed as the LAST arg of a function, a closure can appear outside of the ()

```
let names = [“brad”, “candy”, “fred”]
let out = names.sorted() { $0 < $1 }
```

When the function call has no other params, the () may disappear

```
let names = [“brad”, “candy”, “fred”]
let out = names.sorted { $0 < $1 }
```

[Optional] Escaping Closures: “run away” code

- A closure **C** can be passed to a function **F**, but **C** will run after **F** terminates
- **Problem**: when C runs, the scope of constants/variables captured by C is no longer valid
- **Solution**: Use @escaping annotation in the closure parameter of F

```
// Defined by a library
var logoutCallbacks: Array<() -> Void> = []

func onUserLogout(@escaping exitWork: () -> Void)
{
      logoutCallbacks.append(exitWork);
}
```

Function F

```
// Called by a user app
func remindUserOfUnsavedWork() {
  // code that runs when user logged out
}
func cleanupCache() {
  // code that runs when user logged out
}

// Registered at the beginning of session
onUserLogout(remindUserOfUnsavedWork);
onUserLogout(cleanupCache);
```

Closure C

Protocols

Protocols in Swift (Java Interfaces Twin)

- A protocol provides a mechanism to define expected behaviors between unrelated objects
  - Similar to Java interfaces or C++ abstract base class
- Practical Use Cases:
  - Defining delegate functions in iOS
- A class can inherit a parent class and implement one or more protocol. In the class declaration
  - The name of the parent class must be listed first
  - The name of the protocols follow

Subclassing & Protocol Implementation

Java

```
public class RolloverTimer extends Timer implements Formatable, Persistable {
  // Java code goes here
}
```

Parent class name must be specified first on the list

Swift

```
class RolloverTimer: Timer, Formatable, Persistable {
  // Swift code goes here
}
```

Extensions

Extensions

- A mechanism to **enhance/add** (but not replace/override) new functionality to an existing type, class, struct, enum, or protocol
  - New computed properties
  - New initializers
  - New methods
  - Define subscript operation
  - Define new nested type
  - Make the exiting type conform to a protocol
- Use the `self` keyword in the extension to refer to the type/class/struct/enum/protocol being enhanced

Extension: new computed property

- Use `self` in the extension to refer to properties of the existing type

```
class Timer {
  var hour, minute, second: Int

  // More code goes here
}
```

```
extension Timer {
  var sec: Int {
    // self refers to the Timer obj
    return self.hour * 3600 +
           self.minute * 60 +
           self.second
  }
}
```

```
// Usage
val t = Timer(3, 14, 15)

print ("Total seconds", t.sec)
```

Extension: new property for builtin types

```
extension Int {
  var hhmm: String {
    // self refers to the Int value
    if self >= 3600 { return "too big" }
    let h = self / 60
    let m = self % 60
    return String(format: "%02d:%02d", arguments: [h, m])
  }
}
```

```
// Usage
print (“Duration ”, 100.hhmm)       // Output Duration 01:40
```

Extension: new function for builtin types

```
extension String {
  func middleChar(): Character? {
    let len = self.count
    if len % 2 == 0 {  // length is even, no middle char
      return nil
    }
    else {
       let midIndex = self.index(self.startIndex, offsetBy: len / 2)
       return self[midIndex]
    }
  }
}
```

```
// Usage
print (“Left”.middleChar())    // nil
print (“Leave”.middleChar())   // Optional(‘a’)
```

Reading Assignment

Appendix A7 - A11
Engelsma/Dulimarta textbook