Memory Management, Extension, Protocol, Generic

448460-1
Fall 2015
10/05/2015
Kyoung Shin Park
Multimedia Engineering
Dankook University

Overview

- Automatic Reference Counting
- Extension
- Protocol
- Generic

Automatic Reference Counting

- Automatic Reference Counting (ARC) to manage app memory usage. ARC automatically frees up the memory used by instances when they are no longer needed.

```
class Person { // Person class
  let name: String
  init(name: String) {
    self.name = name
    print(“\(name) is being initialized”)
  }
  deinit {
    print(“\(name) is being deinitialized”)
  }
}
```

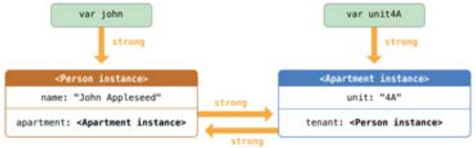

Automatic Reference Counting

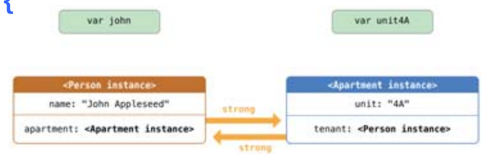
- Strong reference protects the referred object from getting deallocated by ARC by increasing it’s retain count by 1.

```
var per1: Person? // automatically initialized with nil
var per2: Person? // automatically initialized with nil
var per3: Person? // automatically initialized with nil
per1 = Person(name: “Steve”) // “Steve is being initialized” strong
reference increases retain count = 1
per2 = per1 // strong reference increases retain count = 2
per3 = per1 // strong reference increases retain count = 3
per1 = nil // retain count = 2
per2 = nil // retain count = 1
per3 = nil // retain count = 0 calls deinit “Steve is being deinitialized”
```

Reference Cycle

```
class Apartment {
   var tenant: User?
}
class User {
   var home: Apartment?
   func moveIn(apt: Apartment) {
      self.home = apt
      apt.tenant = self
   }
}
var renters = ["John": User()] // John: User init
var apts = [100: Apartment()] // 100: Apartment init
renters["John"]!.moveIn(apts[100]!) // ! is used to unwrap optional
renters["John"] = nil // After User & Apartment=nil, both retain count=1
apts[100] = nil // Reference Cycle (no deinit is called!!) Memory leak!!!
```





Weak Reference to resolve Strong Reference Cycle

```
class Apartment {
   weak var tenant: User? // weak reference (no retain count increase)
}
class User {
   weak var home: Apartment? // weak reference to Apartment
   func moveIn(apt: Apartment) {
      self.home = apt
      apt.tenant = self
   }
} // All weak variables MUST be mutable.
var renters = ["John": User()] // John: User is being initialized
var apts = [100: Apartment()] //100: Apartment is being initialized
renters["John"]!.moveIn(apts[100]!)
renters["John"] = nil // John: User is being deinitialized
apts[100] = nil // 100: Apartment is being deinitialized
```


Unowned Reference to resolve Strong Reference Cycle

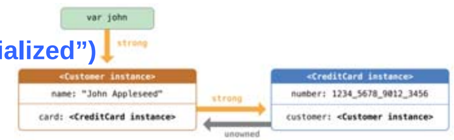
- **Unowned reference doesn't keep a strong reference**. However, it's assumed to always have a value.

```
class Customer {
   let name: String
   var card: CreditCard? // strong reference to CreditCard
   init(name: String) {
      self.name = name
      print("\(name) is being initialized")
   }
   deinit {
      print("\(name) is being deinitialized")
   }
}
```


Unowned Reference

- **Unowned reference is always defined as a nonoptional.**

```
class CreditCard { // CreditCard always belongs to Customer
   let number: UInt64
   unowned let customer: Customer // unowned reference to Customer
(no retain count increase)
   init(number: UInt64, customer: Customer) {
      self.number = number
      self.customer = customer
      print("\(number) is being initialized")
   }
   deinit {
      print("\(number) is being deinitialized")
   }
}
```



Unowned Reference

- **Customer allows CreditCard to be nil, but CreditCard cannot have Customer to be nil.** That is, CreditCard is always owned by Customer. Use **unowned reference** to resolve a strong reference cycle.
- In the previous example, **both User and Apartment can have a property (Apartment and User) allowed to be nil.** Use **weak reference** to resolve a strong reference cycle.

```
var john: Customer?
john = Customer(name: "John") // John is being initiailized
john!.card = CreditCard(number: 12345, customer: john!) // 12345 is being initiailized
john = nil // Both Person and CreditCard are being deinitialized!!
```


Unowned Reference and Implicitly Unwrapped Optional

```
class Country { // Country must always have a capital city
   let name: String
   var capitalCity: City! // implicitly unwrapped optional property
   init(name: String, capitalName: String) {
      self.name = name
      self.capitalCity = City(name: capitalName, country: self)
   }
}
class City { // City always belongs to Country
   let name: String
   unowned let country: Country // unowned reference
   init(name: String, country: Country) {
      self.name = name
      self.country = country
   }
}
```


Unowned Reference and Implicitly Unwrapped Optional

- **Both Country and City should always have a value,** (i.e., **neither property should ever be nil** once initialization is complete). Use **unowned property on one class with an implicitly unwrapped optional property on the other class**, to resolve a strong reference cycle.

```
var nation = Country(name: "Korea", capitalName: "Seoul") // both City and Country are being initialized, without creating a strong reference cycle
print("\(nation.name) capital city is \(nation.capitalCity.name)")

var nation2: Country? = Country(name: "Canada", capitalName: "Ottawa") // both City and Country are being initialized
print("\(nation2!.name) capital city is \(nation2!.capitalCity.name)")
nation2 = nil // both Country and City are being deinitialized
```


Strong Reference Cycles for Closures

- A strong reference cycle can also occur if you assign a **closure** to a property of a class instance, and the body of that closure captures the instance.
- A strong reference cycle occurs because closures (like classes) are reference types. Rather than two class instances, it's a class instance and a closure that are keeping each other alive.

Strong Reference Cycles for Closures

```
class Car {
   var totalMileage: Double = 0.0
   var totalGasUsed: Double = 0.0
   lazy var gasMilage: () -> Double = { // closure
      return self.totalMileage / self.totalGasUsed
   }
   func drive(mileage: Double, _ gas: Double) {
      self.totalMileage = mileage;  self.totalGasUsed = gas
   }
   deinit {
      print("Car is being deinitialized")
   }
}
var myCar: Car? = Car()
myCar!.drive(15000, 700)
print("gasMileage= " + myCar!.gasMilage().description) // 21.4285..
myCar = nil // deinit is NOT being called
```


Closure Capture List

- You resolve a strong reference cycle between a closure and a class instance by defining a capture list as part of the closure's definition.
- Defining a capture list

```
lazy var someClosure: (Int, String) -> String = {
   [unowned self, weak delegate = self.delegate!] (index: Int,
stringToProcess: String) -> String in
   // closure body
}
lazy var someClosure: Void -> String = {
   [unowned self, weak delegate = self.delegate!] in
   // closure body
}
```


Closure Capture List

```
class Car {
   // 중간생략...
   lazy var gasMilage: () -> Double = { // closure capture list
      [unowned self] in
      return self.totalMileage / self.totalGasUsed
   }
}
var myCar: Car? = Car()
myCar!.drive(15000, 700)
print("gasMileage= " + myCar!.gasMilage().description) // 21.4285..
myCar = nil // Car is being deinitialized
```


Define a Capture in a Closure as a Weak and Unowned Reference

- **Define a capture in a closure as an unowned reference** when the closure and the instance it captures will always refer to each other, and will always be deallocated at the same time.
- Conversely, **define a capture as a weak reference** when the captured reference may become nil at some point in the future. Weak references are always of an **optional** type, and automatically become nil when the instance they reference is deallocated. This enables you to check for their existence within the closure's body.

Protocols

- A protocol is a TYPE, except..
  - It has no storage or implementation associated with it
  - Any storage or implementation required to implement the protocol is in an implementing type
  - An implementing type can be any class, struct or enum
  - Otherwise, a protocol can be used as a type to declare variables, as a function parameter, etc
- There are three aspects to a protocol
  - The protocol declaration (what properties and methods are in the protocol)
  - The declaration where a class, struct or enum says that it implements a protocol
  - The actual implementation of the protocol in said class, struct, or enum

Protocols

- Declaration of the protocol itself

```
protocol SomeProtocol: class, InheritedProtocol1, InheritedProtocol2 {
  var someProperty: Int { get set }
  func aMethod(arg1: Double, arg2: String) -> Type
  mutating func changeIt()
  init(arg: Type)
}
```

  - Anyone that implements **SomeProtocol** must also implement **InheritedProtocol1** and **InheritedProtocol2**
  - You must specify whether a property is get only or both **get** and **set**
  - Any functions that are expected to mutate the receiver should be marked **mutating** (unless you are going to restrict your protocol to class implementers only with **class** keyword)
  - You can even specify that implementers must implement a given **intializer**

Protocols

- Implement that protocol

```
class SomeClass: SuperClass, SomeProtocol1, SomeProtocol2 {
  // implementation of SomeClass here, including..
  required init(…)
}
```

  - Claims of conformance to protocols are listed after the superclass for a class
  - Obviously, enums and structs would not have the superclass part
  - Any number of protocols can be implemented by a given class, struct, or enum
  - In a class, inits must be marked **required** (or otherwise a subclass might not conform)

Protocols

- Implement that protocol via extension

```
extension Something: SomeProtocol {
  // implementation of SomeProtocol here
  // no stored properties though
}
```

  - You are allowed to add protocol conformance via an **extension**

Protocols

```
protocol Bird { // some protocol
   var name: String { get set  }
   var canFly: Bool { get }
}
protocol Flyable { // another protocol
   var airSpeed: Double { get }
}
struct FlappyBird: Bird, Flyable { // struct inherits protocols
   var name: String
   let canFly = true
   let flappyAmplitude: Double
   var airSpeed: Double {
      return 3 * flappyAmplitude
   }
}
```


Protocols

```
protocol Moveable {
   mutating func moveBy(p: CGPoint)
}
class Car: Moveable {
   var point: CGPoint
   func moveBy(p: CGPoint) { … } // don't need mutating in class
   func drive() { … }
   init(point: CGPoint) { self.point = point }
}
struct Shape: Moveable {
   var point: CGPoint
   mutating func moveBy(p: CGPoint) { … } // only in struct or enum
   func draw() { … }
}
let sonata: Car = Car(point: CGPoint(x: 1, y: 1)) // sonata (1,1)
let square: Shape = Shape(point: CGPoint(x: 2, y: 2)) // square (2,2)
```


Protocols

```
var thingToMove: Moveable = sonata
thingToMove.moveBy(CGPoint(x: 1, y: 1)) // sonata (2,2) square (2,2)
(thingToMove as! Car).drive() // sonata drive
thingToMove = square
(thingToMove as! Square).draw() // square draw
let thingsToMove: [Moveable] = [sonata, square] // sonata & square (2,2
for var s in thingsToMove {
   s.moveBy(CGPoint(x: 1, y: 1)) // sonata (3,3) square (3,3)
}
// sonata (3,3) square (2,2)
func slide(var slider: Moveable) {
   slider.moveBy(CGPoint(x: 2, y:3)) // sonata (5,6) square (4,5)
}
slide(sonata) // sonata (5,6)
slide(square) // square (4,5)
// sonata (5,6) square (2,2)
```


Protocols

```
protocol Pullable: class {  // use only in the class (not struct or enum)
   func pull()
}
class Thing {
}
//class Boards: Thing, Pullable {
//} // compile error (due to protocol method is required; it needs pull()
method implementation)
class Boards: Thing, Pullable {
   func pull() {
      print("It is pullable object")
   }
}
let b = Boards()
b.pull() // It is pullable object
```

Protocols

```
@objc protocol Pullable { // Obj-C style protocol requirement optional
   func pull()
}
func performPull(object: Thing) {
   if let pullableObject = object as? Pullable { // as? returns Pullable or nil
      pullableObject.pull()
   }
   if object is Pullable { // is returns true or false
      (object as! Pullable).pull() // as! returns Pullable or run time error
   }
   //var pullable = object as! Pullable // as! returns Pullable or run time err
}
performPull(Boards()) // It is pullable object. It is pullable object
performPull(Thing()) // (cannot cast to Pullable)
```


Protocols

- Common Protocols

```
protocol Equatable
   ==(_: _:) -> Bool
protocol Hashable (inherits from Equatable)
   var hashValue: Int { get }
protocol Comparable (inherits from Equatable)
   <(_: _:) -> Bool
protocol CustomStringConvertible // (Printable in Swift 1)
   var description: String { get }
protocol CustomDebugStringConvert
   var debugDescription: String { get }
```


Protocol Extensions

- In Swift 1, protocols were like interfaces to specify a set of properties and methods that a class, struct, or enum would then conform to.
- In Swift 2, you can extend protocols and add default implementations for properties and methods.

```
extension CustomStringConvertible {
   var uppercaseDescription: String {
      return "\(self.description.uppercaseString)!!"
   }
}
let greetings = ["Hello", "Hi"]
print(greetings) // ["Hello", "Hi"]
print("\(greetings.description)") // ["Hello", "Hi"]
print("\(greetings.uppercaseDescription)") // ["HELLO", "HI"]!!
```


Extension

- String Extension

```
extension String { // String extension
   func beginsWith(str: String) -> Bool {
      if let range = self.rangeOfString(str) {
         return range.startIndex == self.startIndex
      }     return false
   }
 func endWith(str: String) -> Bool {
      if let range = self.rangeOfString(str, options:
NSStringCompareOptions.BackwardsSearch) {
         return range.endIndex == self.endIndex
      }     return false
   }
}
```

Extension

```
// String extension
print(str.beginsWith("H") // true
print(str.beginsWith("He") // true
print(str.beginsWith("Hello!") // false
print(str.endsWith("o") // true
print(str.endsWith("lo") // true
```

Generic

- Generic enables you to write flexible, reusable functions and types that can work with any type.
- Type Constraint Syntax

```
func someFunction<T: SomeClass, U: SomeProtocol>(someT: T,
someU: U) {
   // function body
}
```

Generic

```
func swapTwoInts(inout a: Int, inout b: Int)  {
   let tempA = a
   a = b
   b = tempA
}
func swapTwoDoubles(inout a: Double, inout b: Double)  {
   let tempA = a
   a = b
   b = tempA
}
func swapTwoValues<T>(inout a: T,  inout b: T) {
   let tempA = a
   a = b
   b = tempA
}
```

Generic

```
struct IntStack {
   var elements = [Int] ()
   mutating func push(element: Int) {
      elements.append(element)
   }
  mutating func pop() -> Int {
      return elements.removeLast()
   }
}
```

Generic

```
struct StringStack {
   var elements = [String] ()
   mutating func push(element: String) {
      elements.append(element)
   }
  mutating func pop() -> String {
      return elements.removeLast()
   }
}
```

Generic

```
struct Stack<T> {
   var elements = [T] ()
   mutating func push(element: T) {
      elements.append(element)
   }
  mutating func pop() -> T {
      return elements.removeLast()
   }
}
var intStack = Stack<Int>()
intStack.push(50)
print(intStack.pop()) // 50
var stringStack = Stack<String>()
stringStack.push("Hello")
print(stringStack.pop()) // Hello
```

Generic Extension

- When you extend a generic type, you do not provide a type parameter list as part of the extension's definition.

```
extension Stack {
   var topItem: Element? {
      return items.isEmpty ? Nil : items[items.count – 1]
   }
}

if let topItem = stackOfStrings.topItem {
   print("The top item on the stack is \(topItem)")
}
```

References

- Lecture 6 Slide from Developing iOS8 Apps with Swift (Winter 2015) @Stanford University