MEMORY MODEL
Programming Handheld Systems—iOS

CMSC 436
Spring 2019

TODAY

• Automatic Reference Counting

• Memory Safety
AUTOMATIC REFERENCE COUNTING

- tracks and manages app’s memory usage
  - “just works”
  - allocate (correctly) and forget
  - no explicit deallocation
- only for reference data
  - classes, closures
  - not structs or enumerations
- Works by reclaiming objects with no strong references, through:
  - properties
  - constants
  - variables

HOW IT WORKS

1. class instance creation allocates memory hunk
   1. stored properties,
   2. type info
2. ARC automatically frees when instance no longer needed
3. Detects “need” through reference counting
HOW IT WORKS

1. class instance creation allocates memory hunk
   1. stored properties,
   2. type info
2. ARC automatically frees when instance no longer needed
3. Detects “need” through reference counting
HOW IT WORKS

1. class instance creation allocates memory hunk
   1. stored properties,
   2. type info
2. ARC automatically frees when instance no longer needed
3. Detects “need” through reference counting

![Diagram showing A and B connected to a class instance, with numbers 1 and 2 indicated.]

CMSC 436 (iOS) • Fall 2021 • Pete Keleher
HOW IT WORKS

1. class instance creation allocates memory hunk
   1. stored properties,
   2. type info
2. ARC automatically frees when instance no longer needed
3. Detects “need” through reference counting

Freed!
LIFETIME OF A SWIFT OBJECT

1. Allocation (memory taken from stack or heap)
2. Initialization (init() code runs)
3. Usage (the object is used)
4. Deinitialization (deinit() code runs)
5.Deallocation (memory returned to stack or heap)

CREATING, DEINIT()-ING

class Person {
    let name: String
    init(name: String) { self.name = name }
    var apartment: Apartment?
    deinit { print("\(name) is being deinitialized") }
}

class Apartment {
    let unit: String
    init(unit: String) { self.unit = unit }
    var tenant: Person?
    deinit { print("Apartment \(unit) is being deinitialized") }
}

var john: Person?
var unit4A: Apartment?

john = Person(name: "John Appleseed")
unit4A = Apartment(unit: "4A")

// john = nil
// unit4a = nil
CREATING, DEINIT()-ING

```swift
class Person {
    let name: String
    init(name: String) { self.name = name }
    var apartment: Apartment?
    deinit { print("\(name) is being deinitialized") }
}
class Apartment {
    let unit: String
    init(unit: String) { self.unit = unit }
    var tenant: Person?
    deinit { print("Apartment \(unit) is being deinitialized") }
}

var john: Person?
var unit4A: Apartment?

john = Person(name: "John Appleseed")
unit4A = Apartment(unit: "4A")

// john = nil
// unit4a = nil
```

CREATING, DEINIT()-ING

```swift

class Person {
    let name: String
    init(name: String) { self.name = name }
    var apartment: Apartment?
    deinit { print("\(name) is being deinitialized") }
}
class Apartment {
    let unit: String
    init(unit: String) { self.unit = unit }
    var tenant: Person?
    deinit { print("Apartment \(unit) is being deinitialized") }
}

var john: Person?
var unit4A: Apartment?

john = Person(name: "John Appleseed")
unit4A = Apartment(unit: "4A")

john!.apartment = unit4A
unit4A!.tenant = john
```
CREATING, DEINIT()-ING

class Person {
    let name: String
    init(name: String) { self.name = name }
    var apartment: Apartment?
    deinit { print("(name) is being deinitialized") }
}
class Apartment {
    let unit: String
    init(unit: String) { self.unit = unit }
    var tenant: Person?
    deinit { print("Apartment \(unit) is being deinitialized") }
}

var john: Person?
var unit4A: Apartment?
john = Person(name: "John Appleseed")
unit4A = Apartment(unit: "4A")
john?.apartment = unit4A
unit4A?.tenant = john
john = nil
unit4A = nil

Strong reference cycle

CREATING, DEINIT()-ING

class Person {
    let name: String
    init(name: String) { self.name = name }
    var apartment: Apartment?
    deinit { print("(name) is being deinitialized") }
}
class Apartment {
    let unit: String
    init(unit: String) { self.unit = unit }
    var tenant: Person?
    deinit { print("Apartment \(unit) is being deinitialized") }
}

var john: Person?
var unit4A: Apartment?
john = Person(name: "John Appleseed")
unit4A = Apartment(unit: "4A")
john?.apartment = unit4A
unit4A?.tenant = john
john = nil
unit4A = nil

Strong reference cycle
RESOLVING STRONG CYCLES

- **weak** references
  - does not keep strong hold on instance
  - references might be set to `nil` by runtime system
  - must be optional variables (not constants)
  - **use when other side shorter lifetime**

- **unowned** references
  - does not keep strong hold on instance
  - never set to `nil`
  - must not be optional
  - **used when referenced object has longer lifetime**
  - must be set during an `init()`

WEAK REFERENCES

```swift
class Person {
    let name: String
    init(name: String) { self.name = name }
    var apartment: Apartment?
    deinit { print("\(name) is being deinitialized") }
}
class Apartment {
    let unit: String
    init(unit: String) { self.unit = unit }
    weak var tenant: Person?
    deinit { print("Apartment \(unit) is being deinitialized") }
}

var john: Person?
var unit4A: Apartment?

john = Person(name: "John Appleseed")
unit4A = Apartment(unit: "4A")

john!.apartment = unit4A
unit4A!.tenant = john

john = nil
unit4A = nil
```
WEAK REFERENCES

(code from previous page)

![Diagram of weak references]

WEAK REFERENCES

(code from previous page)

![Diagram of weak references]
VS Garbage Collection - CACHING

- Systems with automatic garbage collection
  - delay deallocation until next “collection”
  - weak pointers aren’t automatically deallocated
  - effectively “cache” values that might be re-instantiated
- ARC deallocates immediately
  - no caching
class Customer {
    let name: String
    var card: CreditCard?
    init(name: String) {
        self.name = name
    }
    deinit { print("\(name) is being deinitialized") }
}

class CreditCard {
    let number: UInt64
    unowned let customer: Customer
    init(number: UInt64, customer: Customer) {
        self.number = number
        self.customer = customer
    }
    deinit { print("Card \(number) is being deinitialized") }
}

var john: Customer?
john = Customer(name: "John Appleseed")
john!.card = CreditCard(number: 1234_5678_9012_3456, customer: john!)
```swift
class Customer {
    let name: String
    var card: CreditCard?

    init(name: String) {
        self.name = name
    }

    deinit {
        print("\(name) is being deinitialized")
    }
}

class CreditCard {
    let number: UInt64
    unowned let customer: Customer

    init(number: UInt64, customer: Customer) {
        self.number = number
        self.customer = customer
    }

    deinit {
        print("Card \(number) is being deinitialized")
    }
}

var john: Customer?
john = Customer(name: "John Appleseed")
john!.card = CreditCard(number: 1234_5678_9012_3456, customer: john)
john = nil
```

---

```swift
class Customer {
    let name: String
    var card: CreditCard?

    init(name: String) {
        self.name = name
    }

    deinit {
        print("\(name) is being deinitialized")
    }
}

class CreditCard {
    let number: UInt64
    unowned let customer: Customer

    init(number: UInt64, customer: Customer) {
        self.number = number
        self.customer = customer
    }

    deinit {
        print("Card \(number) is being deinitialized")
    }
}

var john: Customer?
john = Customer(name: "John Appleseed")
john!.card = CreditCard(number: 1234_5678_9012_3456, customer: john)
john = nil
```
**UNOWNED REFERENCES**

```swift
class Customer {
    let name: String
    weak var card: CreditCard?
    init(name: String) {
        self.name = name
    }
    deinit { print("\(name) is being deinitialized") }
}

class CreditCard {
    let number: UInt64
    unowned let customer: Customer
    init(number: UInt64, customer: Customer) {
        self.number = number
        self.customer = customer
    }
    deinit { print("Card #\(number) is being deinitialized") }
}

var john: Customer?
john = Customer(name: "John Appleseed")
john!.card = CreditCard(number: 1234_5678_9012_3456, customer: john)
john = nil
```

**WHY NOT WEAK?**

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

class CreditCard {
    let number: UInt64
    let customer: Customer
    init(number: UInt64, customer: Customer) {
        self.number = number
        self.customer = customer
    }
    deinit { print("Card #\(number) is being deinitialized") }
}

var john: Customer?
john = Customer(name: "John Appleseed")
john!.card = CreditCard(number: 1234_5678_9012_3456, customer: john)
```

No strong ref to card, immediately disappears.
UNOWNED REFERENCES, REDUX

- *unowned* references
  - never set to `nil`
  - must be set during an `init()`
  - defined using non-optional types
  - used when referenced object has *longer* lifetime

CYCLE SCENARIOS

- person / apartment
  - both could be `nil`
  - resolved w/ person having weak ref to apartment
- customer creditcard
  - card reference to customer will never be `nil`

- What if neither can be `nil`?  (both must be initialized in `init()`)  
  - unowned property on one class
  - implicitly unwrapped optional on other
BACKGROUND: INITIALIZATION

• Phase 1:
  • initializer runs, ensures all properties have values
  • super of same init called, does same
  • init considered complete

• Phase 2:
  • super can continue customizing
  • init customizes
  • self can be accessed
  • properties can be modified
  • instance methods can be called

Compiler / Runtime guarantees this through following checks:

1) Designated initializer must ensure all properties introduced by class are initialized ➔ calling superclass initializer

2) Designated initializer must call superclass initializer ➔ assigning values to inherited properties

3) Convenience initializer must call designated initializer ➔ assigning value to any property.

4) Phase 1 complete ➔ calling any instance methods, reading any property values, or referring to self.
class Country {
    let name: String
    var capitalCity: City
    init(nameIn: String, capitalName: String) {
        name = nameIn
        capitalCity = City(nameIn: capitalName, countryIn: self)
    }
}

class City {
    let name: String
    let country: Country
    init(nameIn: String, countryIn: Country) {
        name = nameIn
        country = countryIn
    }
}

var country = Country(nameIn: "Canada", capitalName: "Ottawa")
UNOWNED / UNWRAPPED

class Country {
    let name: String
    var capitalCity: City!
    init(nameIn: String, capitalName: String) {
        name = nameIn
        capitalCity = City(nameIn: capitalName, countryIn: self)
    }
}

class City {
    let name: String
    unowned var country: Country
    init(nameIn: String, countryIn: Country) {
        name = nameIn
        country = countryIn
    }
}

var country = Country(nameIn: "Canada", capitalName: "Ottawa")

CYCLE SCENARIOS: CLOSURES

- Can occur (for example) if:
  - closure assigned to a property of class instance
  - closure body references a property of the instance
class Person {
    var firstName: String?
    var lastName: String?
    lazy var fullName: () -> String = {
        return "\(self.firstName!) \(self.lastName!)"
    }
    init(firstName: String, lastName: String) {
        self.firstName = firstName
        self.lastName = lastName
        print("Person Class is being initialised")
    }
    deinit {
        print("Person Class is being de-initialised")
    }
}

var person: Person? = Person(firstName: "Klay", lastName: "Thompson")
// Prints "Person Class is being initialised"
print(person?.fullName())
// Prints “Klay Thompson”

Lazy means “self” won’t be accessed until after initialization, which does not violate safety checks.
SOLVING CLOSURES

• Capture lists
  • add type annotations to closure parameters
  • can make weak or unowned, or define new property

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

// parameter and return types can often be inferred from context
lazy var someClosure: () -> String = {
    [unowned self, weak delegate = self.delegate!] in
    // closure body goes here
}
```

CAPTURE LIST

```swift
class Person {
    var firstName: String?
    var lastName: String?
    lazy var fullName: () -> String = {
        [unowned self] in
        return "\(self.firstName!) \(self.lastName!)"
    }
    init(firstName: String, lastName: String) {
        self.firstName = firstName
        self.lastName = lastName
        print("Person Class is being initialised")
    }
    deinit {
        print("Person Class is being de-initialised")
    }
}
```