CSC 491 / 391
Mobile Application Development for iOS II

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

1

Memory Management

2

Outline

- Memory management
- Garbage collection
- Automatic reference counting
- Strong, weak, and unowned references

3

Memory Management

- Managing the *allocation* and *deallocation* of memory space
  - For variables, objects, closures, etc.
- Swift support *stack* and *heap* based allocations
- *Stack*-based allocation (LIFO)
  - For variables of value types:
    - Basic types, Structs, Enums, and tuples
- *Heap*-based allocation
  - For objects, i.e., variables of reference types:
    - Class types and Closures

4

Stack-Based Memory Allocation

- *Stack* (LIFO) allocated memory is owned by the enclosing block or function
- The lifetime of allocation *coincides* with the lifetime of the enclosing function call
  - Value semantics
  - Assignment by copying the values — Overhead of copying
- Deallocation is automatic
  - Memory is deallocated when exiting the enclosing scope or returning from a function

5

Definition: Lifetime

Lifetime, or life span, of an *object*, or *memory allocation*
The time span between the allocation and deallocation of the object/memory

Lifetime, or life span, of a *function call*
The time span between the invocation and the completion, i.e., return, of the function call

- A *run-time* concept, concerning the states of objects or functions during execution

6

Compare Lifetime vs. Scope

Scope, of a declaration of a variable or constant
The region of code where the declaration is in effect, i.e., visible.
Usually delimited by a pair of { and } that immediately encloses the declaration.

- A static concept defined *wrt.* to the static structure of the code, rather than the execution of the code.
- Easier to reason and manage.
- In stack-based allocation, the lifetime correspond to the scope

DePaul University 7

7

Stack-Based Memory Allocation

```
struct Point { var x, y: Int }
var p = Point(x: 0, y: 0)

func a() {
    var i: Int = 100
    b(i)
}
func b(_ n: Int) {
    var d: Double = 2.0
    c(d)
}
func c(_ v: Double) {
    var q = Point(x: 3, y: 3)
    p = q
}

a()
```

DePaul University 8

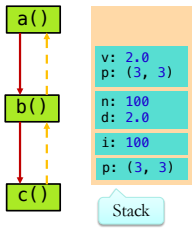
8

Stack-Based Memory Allocation

```
struct Point { var x, y: Int }
var p = Point(x: 0, y: 0)

func a() {
    var i: Int = 100
    b(i)
}
func b(_ n: Int) {
    var d: Double = 2.0
    c(d)
}
func c(_ v: Double) {
    var q = Point(x: 3, y: 3)
    p = q
}

a()
```



DePaul University 9

18

Stack-Based Memory Allocation – Lifetime of Allocation

```
struct Point { var x, y: Int }

func e(_ i: Int) -> Point {
    return f(i + 1)
}
func f(_ i: Int) -> Point {
    return g(i + 1)
}
func g(_ i: Int) -> Point {
    return Point(x: i, y: i)
}

e(1)
```

DePaul University 10

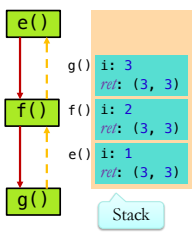
19

Stack-Based Memory Allocation – Lifetime of Allocation

```
struct Point { var x, y: Int }

func e(_ i: Int) -> Point {
    return f(i + 1)
}
func f(_ i: Int) -> Point {
    return g(i + 1)
}
func g(_ i: Int) -> Point {
    return Point(x: i, y: i)
}

e(1)
```



DePaul University 11

29

Heap-Based Memory Allocation

- *Heap* (unordered) allocated memory is shared application wide
- The lifetime of allocation is *independent* from the lifetime of the enclosing function call
  - May exceed the enclosing function
  - Assignment by copying the references
- Deallocation must be managed
  - When does an object become *garbage*?

Efficient, but allows sharing, i.e., aliasing

An object that is unreachable

DePaul University 12

30

Definition: Garbage

Garbage, in *memory management*
A object that cannot be reached by following references starting from the variables declared at the top, i.e., the outmost level of a program.

DePaul University 13

31

Heap-Based Memory Allocation

```
class Point { var x, y: Int …}
var p = Point(x: 0, y: 0)

func a() {
    var i: Int = 100
    b(i)
}
func b(_ n: Int) {
    var d: Double = 2.0
    c(d)
}
func c(_ v: Double) {
    var q = Point(x: 3, y: 3)
    p = q
}

a()
```

DePaul University 14

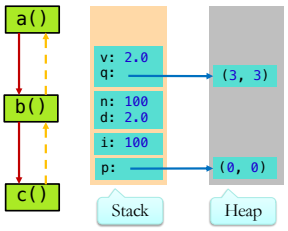
32

Heap-Based Memory Allocation

```
class Point { var x, y: Int …}
var p = Point(x: 0, y: 0)

func a() {
    var i: Int = 100
    b(i)
}
func b(_ n: Int) {
    var d: Double = 2.0
    c(d)
}
func c(_ v: Double) {
    var q = Point(x: 3, y: 3)
    p = q
}

a()
```



Assuming the heap is unmanaged

DePaul University 15

45

Heap-Based Memory Allocation – Lifetime of Allocation

```
class Point { var x, y: Int …}

func e(_ i: Int) -> Point {
    return f(i + 1)
}
func f(_ i: Int) -> Point {
    return g(i + 1)
}
func g(_ i: Int) -> Point {
    return Point(x: i, y: i)
}

e(1)
```

DePaul University 16

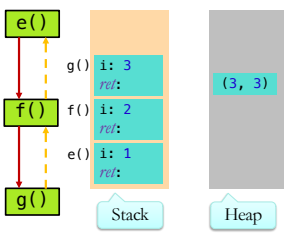
46

Heap-Based Memory Allocation – Lifetime of Allocation

```
class Point { var x, y: Int …}

func e(_ i: Int) -> Point {
    return f(i + 1)
}
func f(_ i: Int) -> Point {
    return g(i + 1)
}
func g(_ i: Int) -> Point {
    return Point(x: i, y: i)
}

e(1)
```



Assuming the heap is unmanaged

DePaul University 17

56

Memory Related Bugs

- Fail to deallocate memory no longer in use
  - ***Memory leaks***
  - Diminishing available memory space due to accumulating garbage
  - Serious consequences. Not immediate.
- Premature deallocation
  - Deallocating memory that is still in use
  - Accessing memory that has been deallocated
  - Memory corruption. Data corruption.
  - Erroneous Results! Crash!
- Double deallocation
  - Deallocating memory that is already deallocated.
  - Memory corruption. Crash!

DePaul University 18

57

Consequences of Memory Leaks

- Inefficient use of valuable memory resources
  - Filled with a increasing amount of garbage
- Preventing applications from running for a long duration
  - iOS apps remains in background, and retain resources
- Fragmentation of memory
  - Preventing large allocations
- Poor system performance
  - Increasing system burdens for house-keeping
- iOS may terminate some apps due to low memory
  - Low memory warnings

58

Memory Management Strategies

- Programmer managed
  - C/C++
- Garbage collection, or GC (system managed)
  - Java
- Reference counting
  - Objective-C, C/C++
- Automated Reference Counting, or ARC (mostly system managed)
  - Swift

59

Memory Management – Programmer Managed

- C/C++ style memory management
  - Memory is allocated, tracked and released by programmers.
  - Explicit alloc and dealloc calls
- Pros
  - Complete control of the life span of the memory
  - Fast, efficient, minimum overhead
- Cons
  - Memory leaks or other memory related bugs often occur
  - Memory related bugs are difficult to track down
    - Expensive for testing, debugging, and QA

60

Memory Management – Garbage Collection

- Java-style garbage collection
  - Introduced in LISP in the 1960's
  - Completely managed by the system
  - Programmers are completely relieved from the duties of memory management.

"There is no such thing as a free lunch"

- Classic algorithm: **mark and sweep**
  - Many refinements in modern implementations of garbage collection, such as JVM

61

Garbage Collection – Mark and Sweep [McCarthy 1960]

- Two-phase, stop and run algorithm
  - Mark: mark all objects that are reachable
  - Sweep: sweep the heap by a linear scan to free all unreachable objects
- The algorithm

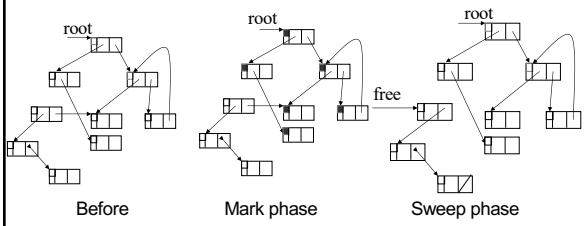
```
for each top-level object p
    mark(p)
sweep()
```

```
mark(p)
    if !p.marked
        p.marked = true
        for each object q referenced by p
            mark(q)
```

```
sweep()
    for each object p in the heap
        if !p.marked
            release p
```


62

Garbage Collection – Mark and Sweep [McCarthy 1960]



63

Memory Management – Garbage Collection

- Pros
  - Easy to use. Release is a no-op.
  - Memory related bugs are completely eliminated.
- Cons
  - Time consuming: **O**(*N*), *N* = size of the heap
    - Slow down normal operations, while running in background
    - May stop all threads, if memory is nearly exhausted
  - Unpredictable timing: may happened at any moment
    - Triggered by a timer or available resource level
  - Performance impact by GC is uneven, non-uniform
  - Noticeable degradation of user experience
- ***iOS does not support garbage collection!***

DePaul University 25

64

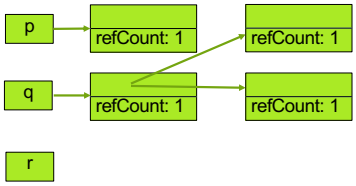
Memory Management – Reference Counting

- Each object maintains a *reference count*:
  - The total number of references to this object
- When an object is referenced or dereferenced, the reference count is incremented or decremented accordingly.
  - The reference count can be maintained by the program or by the system
- When the reference count reaches 0, the object is deallocated
  - The reference count of all object it refers to is decremented by 1
- Will not deallocate an object as long as at least one active reference to that object still remains.

DePaul University 26

65

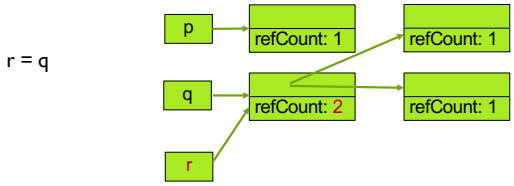
Memory Management – Reference Counting



DePaul University 27

66

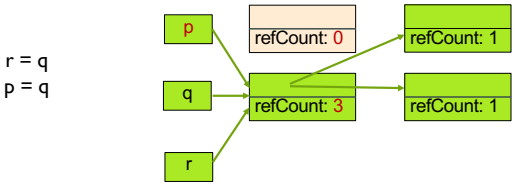
Memory Management – Reference Counting



DePaul University 28

67

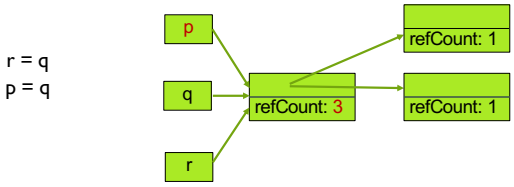
Memory Management – Reference Counting



DePaul University 29

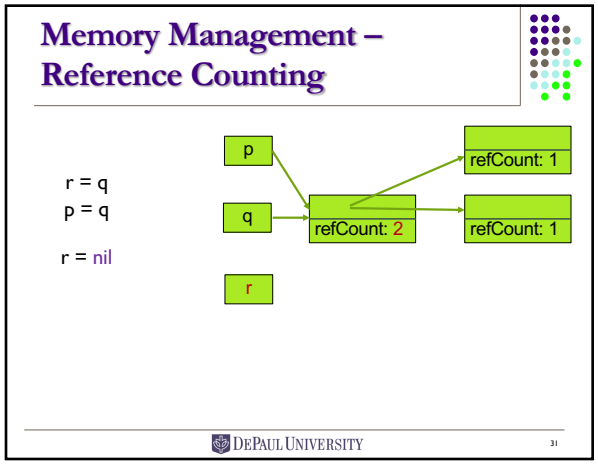
68

Memory Management – Reference Counting

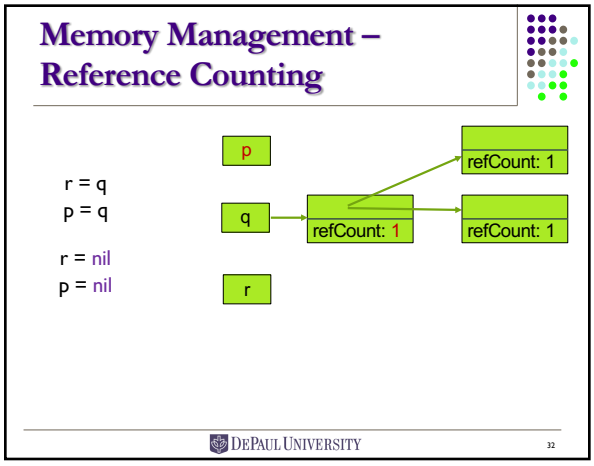


DePaul University 30

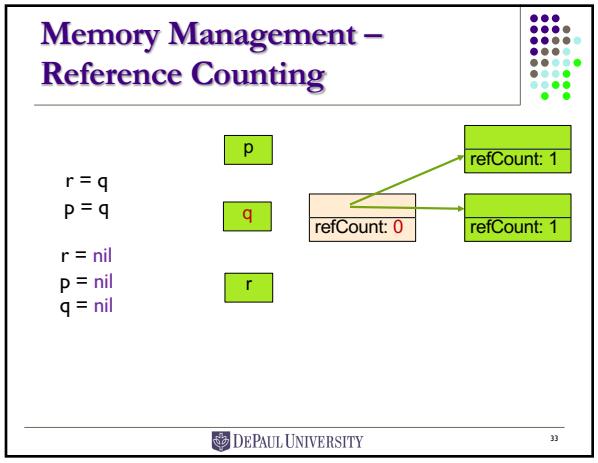
69



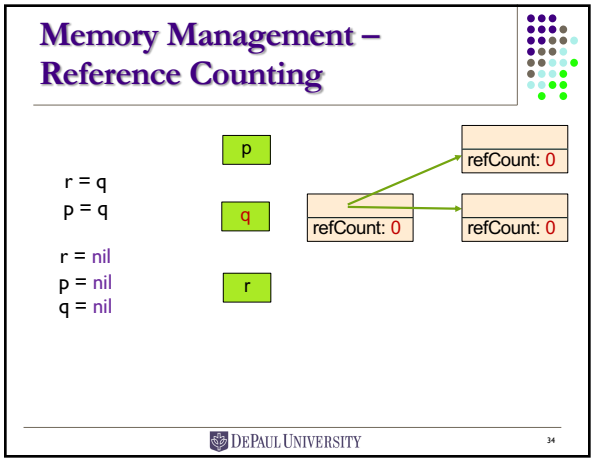
70



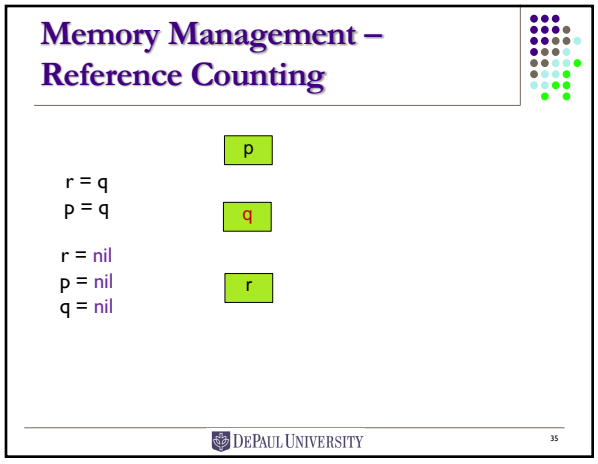
71



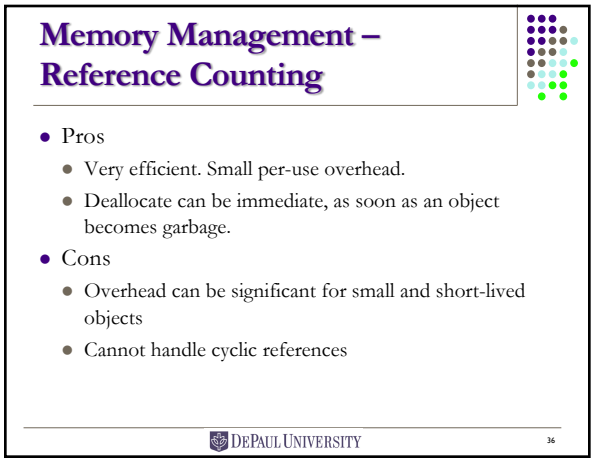
72



73

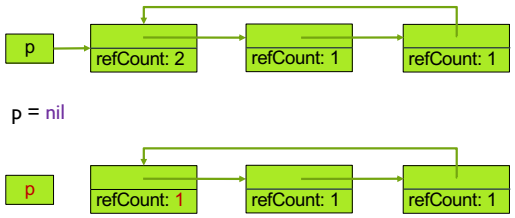


74



75

Reference Counting Cyclic References



- None of the objects is reachable; hence, garbage by definition.

76

Reference Counting in iOS

- *Reference counting* is the default method of memory management for iOS and Objective-C from the beginning.
- Every object has a *retain* count
  - If retain count > 0, the object is alive
  - If retain count == 0, the object is not reference by any other objects and is deallocated by the system
- Explicit methods for maintaining the reference count
  - *retain* and *release*, increment/decrement the reference count
- Programmers are responsible for
  - Calling the appropriate method at the right time
  - Avoiding cyclic references

77

Automated Reference Counting

- *Automated Reference Counting* (ARC) – an major enhancement of reference counting supported by Swift
  - Same underlying mechanism as before
- Reference counting is handled by the system automatically
  - Programmers are relieved from the duties of calling retain/release methods
- Programmers are responsible for identifying the different types of references
  - ***Strong***, ***weak***, or ***unowned***

78

Review of Optional Types

- The value `nil`
  - Representing a missing or invalid value
  - Optional types: allow `nil`
  - Non-optional types: do not allow `nil`
- Avoid bugs related to null-pointers by using non-optional types whenever possible
- Uniform handling of `nil` when necessary
  - *Nullable* reference types
    - Related objects with independent lifetimes
  - Failures and errors

79

Review of Optional Types

- Optional values must be *unwrapped* before it can be accessed.

`nil` *or* value

```
var str: String? = …
str
```

Optional value.
Type of `str`: `String?`

80

Review of Optional Types

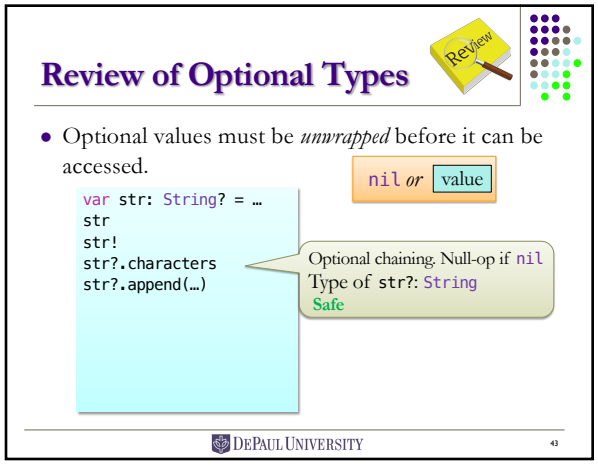
- Optional values must be *unwrapped* before it can be accessed.

`nil` *or* value

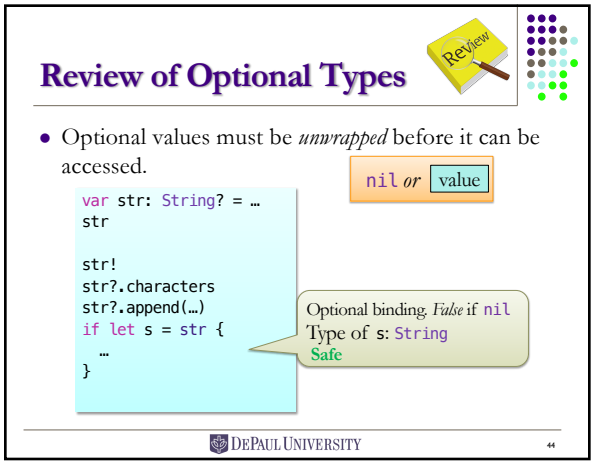
```
var str: String? = …
str
str!
```

Forced unwrapping
Type of `str!`: `String`
Unsafe!

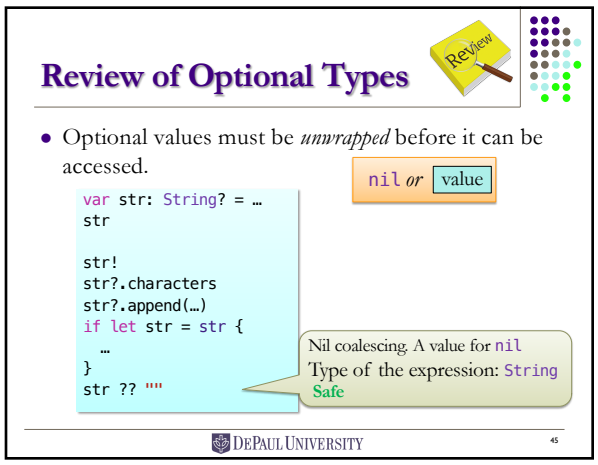
81



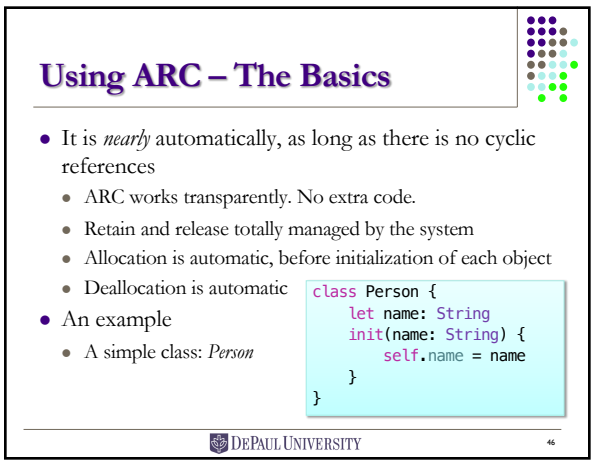
82



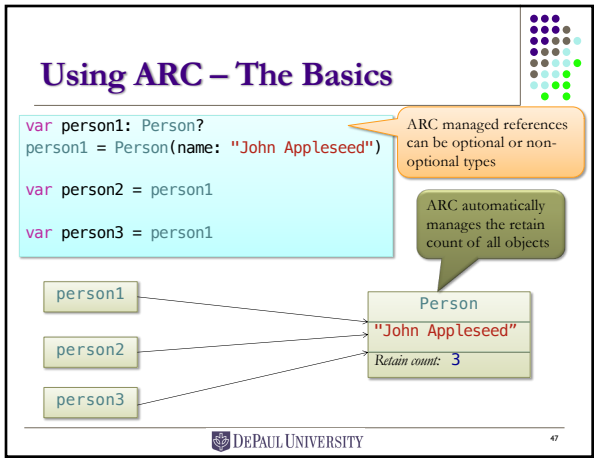
83



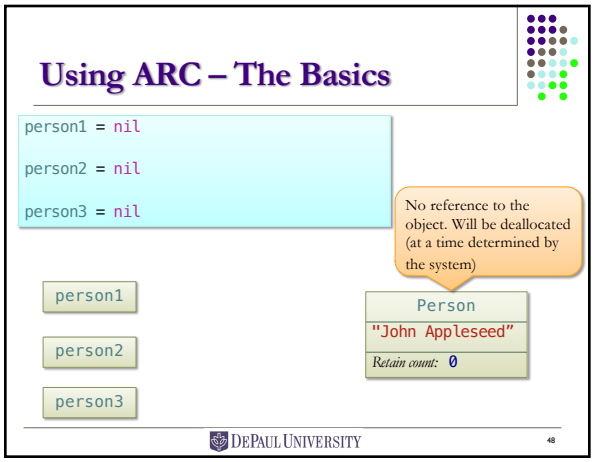
84



85



88



91

Using ARC – Cyclic References

- What if there are cyclic references?
- Consider the following classes

```
class Person {
    let name: String
    init(name: String) {
        self.name = name
    }
    var apartment: Apartment?
}
```

```
class Apartment {
    let unit: String
    init(unit: String) {
        self.unit = unit
    }
    var tenant: Person?
}
```

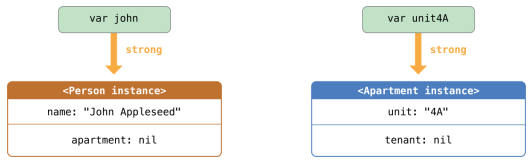
DePaul University 49

92

Using ARC – Cyclic References

```
var john: Person?
var unit4A: Apartment?

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

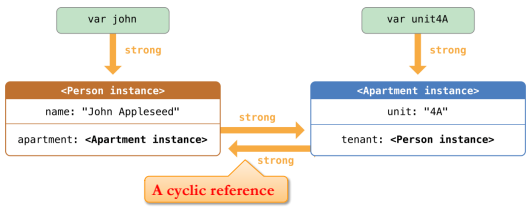


DePaul University 50

93

Using ARC – Cyclic References

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

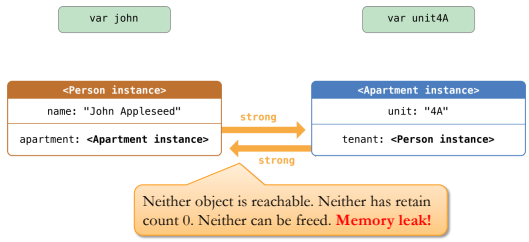


DePaul University 51

94

Using ARC – Cyclic References

```
john = nil
unit4A = nil
```



DePaul University 52

95

Resolving Cyclic References – Using *Weak* References

- By default, references are considered *strong*
  - Increment the retain count of the referenced object
  - An object referenced by a strong reference can never be freed
  - Strong cyclic references cause memory leaks
- *Weak* references do not prevent referenced object to be freed
  - *Do not* increment the retain count of the referenced object
  - Can be used to break the strong reference cycles
  - Referenced objects may be deallocated! Set to `nil` by the system
  - Must be declared as a variable of optional types — Can it be a constant?
- An object must be referenced by *at least one strong reference* to stay alive

DePaul University 53

96

Weak References

- Weak references are optional values.
  - Use optional binding to unwrap
- Accessing a nil weak reference will cause a crash!
- Weak references can be *implicitly unwrapped*
  - Must remain non-nil after initialization
  - Can be used without unwrapping

DePaul University 54

97

Resolving Cyclic References – Using *Weak* References

- Break the strong reference cycle

  Strong references should never form a cycle
- Declare one of the reference in the cycle to be weak

An optional reference type variable can be declared weak.

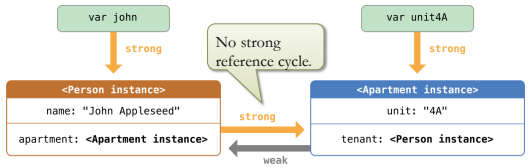
```
class Apartment {
    let unit: String
    init(unit: String) {
        self.unit = unit
    }
    weak var tenant: Person?
}
```

DePaul University 55

98

Resolving Cyclic References – Using *Weak* References

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

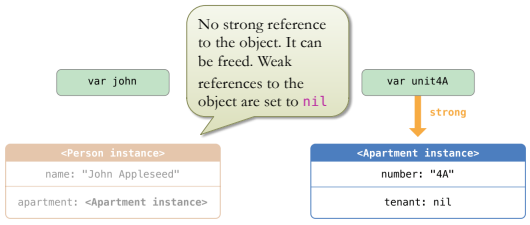


DePaul University 56

99

Resolving Cyclic References – Using *Weak* References

```
john = nil
```

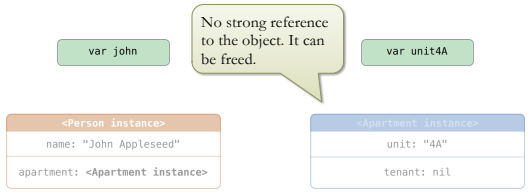


DePaul University 57

100

Resolving Cyclic References – Using *Weak* References

```
john = nil
unit4A = nil
```



DePaul University 58

101

Accessing Weak References

- Weak references must be declared as optional values

```
class Apartment {
    let unit: String
    init(unit: String) {
        self.unit = unit
    }
    weak var tenant: Person?
}
```

```
var john: Person?
var unit4A: Apartment?
john = Person(name: "John Appleseed")
unit4A = Apartment(unit: "4A")
…
unit4A?.tenant?.name
if let p = unit4A?.tenant {
    p.name
    p.apartment?.unit
}
```

DePaul University 59

102

Resolving Cyclic References – Using *Unowned* References

- *Unowned* references are ***similar*** to weak references in following aspects:
  - *Do not* prevent referenced object to be freed
  - *Do not* increment the retain count of the referenced object
  - Can be used to break the strong reference cycles
- *Unowned* references are ***different*** from weak references in following aspects:
  - Declared as variables or constants of ***non-optional types***
  - Cannot become `nil`
  - **The lifetime of the referenced object must be longer than the lifetime of this object**
  - The referenced object shall never be deallocated before this object

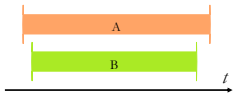
DePaul University 60

103

Why Unowned References?

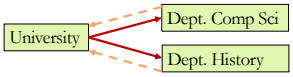
- Consider the *ownership* relation
  - A owns B, B is owned-by A
- The lifetime of owner must subsume or coincide with every object it owns.
  - A must exist before B.
  - A cannot vanish before B.
  - B cannot exist without A.
- The relationship is *acyclic* by definition
- Example of ownership relation
  - Organization – Unit, Book – Chapter – Section

Similar relationship: *containment, composition*



104

Represent Ownership Relation – Using Unowned Reference



Guarantees
- No memory leak
- No cyclic references

- The owner maintains a strong reference to the objects it owns
- Owned objects may have unowned references to the owner
- Assigning responsibilities for memory management
  - Owner is responsible for managing objects it owns.
  - Owned objects are not responsible for their owners

105

Unowned References

- Unowned references must be initialized in the initializer
  - Since it is non-optional type
- After initialization, the unowned reference must remain valid, i.e., not deallocated
- Accessing an unowned reference that has been deallocated will cause a crash!

106

Resolving Cyclic References – Using *Unowned* References

```
class Customer {
    let name: String
    var card: CreditCard?
    init(name: String) {
        self.name = name
    }
}
```

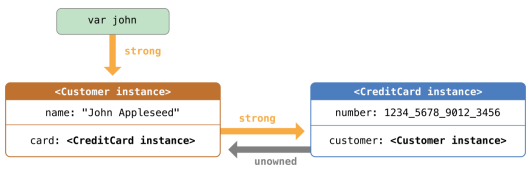
- An ownership relation
  A *Customer* owns a *Credit Card*

```
class CreditCard {
    let number: UInt64
    unowned let customer: Customer
    init(number: UInt64, customer: Customer) {
        self.number = number
        self.customer = customer
    }
}
```


107

Resolving Cyclic References – Using *Unowned* References

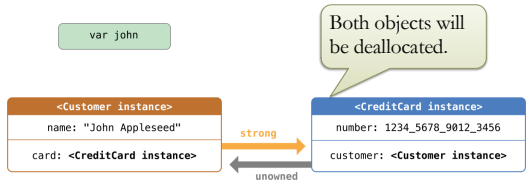
```
var john: Customer?
john = Customer(name: "John Appleseed")
john!.card = CreditCard(number: 1234_5678_9012_3456,
                        customer: john!)
```



108

Resolving Cyclic References – Using *Unowned* References

```
john = nil
```



109

De-Initialization

- The *de-initializer* is called just before an object is deallocated.
  - An optional `deinit` block in a class, no parameters
  - At most one per class
- ARC automatically cleans up memory resources during deallocation.
  - Typically, `deinit` is not needed
- Use a de-initializer to clean up non-memory related resources, e.g.,
  - Save and close opened files before deallocation to avoid lost content
  - Close an opened network connections before deallocation to avoid zombie connections

110

Example of a Deinitializer

Invoked before an object is deallocated

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


111

ARC Principles – Strong References

- **Only strong references keep objects alive**
  - Use strong reference if the referenced object should not be released while the referencing object is alive.
  - At lease one strong reference is needed to keep an object alive.
- **There should be no strong reference cycles**
  - Use weak or unowned references to break strong reference cycles

112

ARC Principles – Weak or Unowned References

- Use *weak references* if the referenced object may be deallocated during the lifetime of the referencing object.
  - Independ lifetime. May be deallocated in either order
- Use *unowned references* only when the referenced object may never be deallocated during the lifetime of the referencing object.
  - Ownership relation. Subsumed lifetime.
  - Programmers' responsibility to ensure this.

113

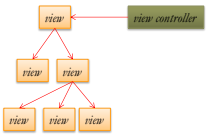
Automatic Reference Counting Recap

- The system automatically manages when to retain and when to release objects.
  - Minimum overhead, highly efficient
- Programmers don't have to explicitly deal with memory allocation and deallocation.
  - No need to call any method to manage memory
- Programmers need to determine whether a reference should be *strong*, *weak*, or *unowned*.
- Value types are not managed by ARC
  - **Use value types for objects of small sizes**

114

UI Outlets – Strong or Weak References

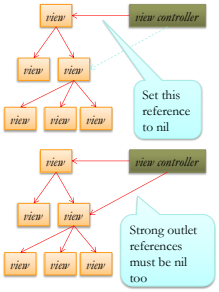
- UI view hierarchy created in IB maintains strong references from top down.
  - For the root view to all its subviews
- The reference from a view controller to its associated view is strong
  - Views will not disappear by themselves



116

UI Outlets – Strong or Weak References

- Outlets created in IB can be strong or weak
- Weak outlet reference is recommended
  - Allow the view hierarchy to be freed more easily if necessary
- Strong outlet reference creates no cycle
  - To free the view hierarchy, all strong outlet references must be set to nil too.



DePaul University 73

119

Next …

- More deep dives into Swift
- Error handling
- Multi-threading and background processing

✧ iOS is a trademark of Apple Inc.

DePaul University 74

120