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

Class

Instance

A

B

Class Instance

(2)

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

Class

Instance

A

B

Class Instance

(3)
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

Class Instance

A
B

CMSC 436 (iOS) • Spring 2019 • 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

```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
```
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

john?.apartment = unit4A
unit4A?.tenant = john
**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
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
  - unowned never set to `nil`
  - defined using non-optional types
  - must be set during an *init()*
  - used when referenced object has *longer* lifetime

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 showing weak references between var john and var unit4A, with a strong reference from john to apartment and a weak reference from apartment to tenant.]

WEAK REFERENCES

(code from previous page)

![Diagram showing weak references between var john and var unit4A, with a strong reference from john to apartment and a weak reference from apartment to tenant.]

CMSC 436 (iOS) • Spring 2019 • Pete Keleher
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!)
UNOWNED REFERENCES

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)
UNOWNED REFERENCES

```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)
```

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
DANGLING POINTERS

unowned can cause dangling pointers

```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!)
let card = john!.card
print(card?.customer.name) // fine
john = nil
print(card?.customer.name) // obj pointed to by card?.customer is gone
```

unowned is the most common way to get dangling references in swift!

- strong references “retain” their objects, and so won’t dangle
- weak references are set to nil by the runtime when object freed

- unowned pointers neither:
  - retain objects (like strong references), or
  - get set to nil (like weak references)
- so you end up with valid-looking references to memory that has been reclaimed
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 \texttt{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
  - \texttt{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.

```swift
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")
```
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")

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 let 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

```swift
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"
person?.fullName()
// Prints "Klay Thompson"
```

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

```swift
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")
    }
}
```

**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
}

// if no parameter or return type because they can be inferred from context
lazy var someClosure: () -> String = {
    [unowned self, weak delegate = self.delegate!] in
    // closure body goes here
}
```
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")
    }
}

TOOLS

• Xcode memory analysis
• SpecLeak
• Swift Lint