ATLS 4120/5120: Mobile Application Development
Week 7: Memory Management

Memory management is the process of controlling and coordinating computer memory, assigning portions called blocks to various running programs to optimize overall system performance. Memory management resides in hardware, in the operating system, and in programs and applications.

Memory management is especially important in mobile app development as devices have limited resources. When we create class instances, it's allocating memory for that instance, so we need to make sure we free up that memory when that instance is no longer being used. Before Xcode 4.2 we had to do all these things manually but starting in Xcode 4.2 Apple added Automatic Reference Counting (ARC) to iOS which automatically handles memory management by freeing up the memory used by class instances when those instances are no longer needed. As developers it's useful to understand how ARC works because we sometimes have to deal with memory leaks.

Automatic Reference Counting (ARC)

Every time you create a new instance of a class, ARC allocates a chunk of memory to store information about that instance. This memory holds information about the type of the instance, together with the values of any stored properties associated with that instance.

When an instance is no longer needed, ARC frees up the memory used by that instance so that the memory can be used for other purposes instead. This ensures that class instances do not take up space in memory when they are no longer needed. This is instrumental for the performance and efficiency of the operating system.

If ARC were to deallocate an instance that was still in use, it would no longer be possible to access that instance's properties, or methods and if you tried to access the instance, your app would crash.

To make sure that instances don't disappear while they are still needed, ARC tracks how many properties, constants, and variables are currently referring to each class instance. ARC will not deallocate an instance as long as at least one active reference to that instance still exists. When the reference count reaches 0 for an instance, it's no longer needed so ARC deallocates the memory for the instance and the memory is reclaimed by the OS.

Reference counting applies only to instances of classes. Structures and enumerations are value types, not reference types, and are not stored and passed by reference.

An Object's Lifetime

The lifetime of a Swift class instance consists of five stages:

1. *Allocation*: Takes memory from a stack or heap.
2. *Initialization*: `init` code runs.
3. *Usage*.
4. *Deinitialization*: `deinit` code runs.
5. *Deallocation*: Returns memory to a stack or heap.

Strong References

Whenever you assign a class instance to a property, constant, or variable, that property, constant, or variable makes a *strong reference* to the instance. The reference is called a "strong" reference because it keeps a firm hold on that instance and does not allow it to be deallocated for as long as that strong reference remains. When a property is being created, the reference is strong unless they are declared weak or unowned.

Example
arc.playground

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

We create a class called Person.
I've added in the init and deinit methods with print statements so we can track initialization and deinitialization.

Now we'll define three instances of the `Person` class. These are optionals so they're initialized with the value of nil and don't yet reference a `Person` instance.

```
var person1: Person?
var person2: Person?
var person3: Person?
```

Now we'll create an instance of `Person` by calling its initializer function.

```
person1 = Person(name: "John Appleseed")
```

Because the new `Person` instance has been assigned to the `person1` variable, there is now a strong reference from `person1` to the new `Person` instance. Because there is at least one strong reference, ARC makes sure that this `Person` is kept in memory and is not deallocated.

If you assign the same `Person` instance to two more variables, two more strong references to that instance are established:

```
person2 = person1
person3 = person1
// 2 more strong references, count=3
```

(run)

There are now *three* strong references to this single `Person` instance.
If you break two of these strong references by assigning `nil` to two of the variables, one strong reference remains, and the `Person` instance is not deallocated:

```
person1 = nil
```

```
person2 = nil
// 2 strong references removed, count=1
```
(run)

ARC does not deallocate the `Person` instance until the third and final strong reference is broken, at which point it's clear that you are no longer using the `Person` instance:
```
person3 = nil
// last strong reference removed, count=0, memeory deallocated
```
(run)

Strong Reference Cycle
However, it's possible to write code in which an instance of a class *never* gets to a point where it has zero strong references. This can happen if two class instances hold a strong reference to each other, such that each instance keeps the other alive. This is known as a *strong reference cycle*.

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

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

Every `Person` instance has a `name` property of type `String` and an optional `apartment` property that is initially `nil`. The `apartment` property is optional, because a person may not always have an apartment.

Every `Apartment` instance has a `unit` property of type `String` and has an optional `tenant` property that is initially `nil`. The tenant property is optional because an apartment may not always have a tenant.

We define a variable of type `Person` and a variable of type `Apartment`, both initially with the value of nil.

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

Now we'll create an instance of `Person` by calling its initializer function and the same for the instance of `Apartment`.

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

So `john` has a strong reference to the `Person` instance and `unit4A` has a strong reference to the `Apartment` instance.
(slide)

Now let's say the two instances get linked because John moves into unit4A and therefore unit4A has John as its tenant.

```
john?.apartment = unit4A
unit4A?.tenant = john
```
(slide)

Unfortunately, linking these two instances creates a strong reference cycle between them. The `Person` instance now has a strong reference to the `Apartment` instance, and the `Apartment` instance has a strong reference to the `Person` instance. Therefore, when you break the strong references held by the `john` and `unit4A` variables, the reference counts do not drop to zero, and the instances are not deallocated by ARC.

```
john = nil
unit4A = nil
```
(run)
Note that neither deinitializer was called when you set these two variables to `nil`. The strong reference cycle prevents the `Person` and `Apartment` instances from ever being deallocated, causing a memory leak in your app.
(slide)

The strong references between the `User` instance and the `Account` instance remain and cannot be broken. This is a *strong reference cycle*. It fools ARC and prevents it from cleaning up.

Resolving Strong Reference Cycles

Swift provides two ways to resolve strong reference cycles when you work with properties of class type: weak references and unowned references.

Weak and unowned references enable one instance in a reference cycle to refer to the other instance *without* keeping a strong hold on it. The instances can then refer to each other without creating a strong reference cycle.

Weak References

We can break the strong reference cycle by using weak references. A weak reference does not keep a strong hold on the instance.

Unless otherwise specified, all references are strong and impact reference counts. Weak references, however, don't increase the reference count of an object. Because a weak reference does not keep a strong hold on the instance it refers to, it's possible for that instance to be deallocated while the weak reference is still referring to it.

A weak reference is *always* optional and ARC automatically sets weak reference to `nil` when the instance is deallocated. Because the value changes it must be a variable as constants won't let you change its value.

By making the user property in the Account class weak we would avoid the strong reference cycle and the instances would be deallocated when they're assigned nil.

Let's look at that example again but this time the `tenant` property in the `Apartment` class is declared as a weak reference.

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

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

// define variables
var john: Person?
var unit4A: Apartment?

//initialize and create the instance
john = Person(name: "John Appleseed")
unit4A = Apartment(unit: "4A")

//instances get linked together
john?.apartment = unit4A
unit4A?.tenant = john
```

(slide)

The `Person` instance still has a strong reference to the `Apartment` instance, but the `Apartment` instance now has a *weak* reference to the `Person` instance. This means that when you break the strong reference held by the `john` variable by setting it to `nil`, there are no more strong references to the `Person` instance:

```
//break strong references
john = nil
```
(run)
(slide)

Because there are no more strong references to the `Person` instance, it's deallocated and the `tenant` property is set to `nil`. You will see the deinitializer method is called.

```
unit4A = nil
```
(run)
(slide)

The only remaining strong reference to the `Apartment` instance is from the `unit4A` variable. If you break *that* strong reference, there are no more strong references to the `Apartment` instance and you'll see its deinitializer method is called.

Unowned References

An unowned reference is very similar to a weak reference that it can be used to resolve the strong reference cycle. The big difference is that an unowned reference always have a value. ARC will not set unowned reference's value to `nil`. In other words, the reference is declared as non-optional types.

Use an unowned reference only when you are sure that the reference *always* refers to an instance that has not been deallocated.

If you try to access the value of an unowned reference after that instance has been deallocated, you'll get a runtime error.

Strong vs. Weak vs. Unowned

(slide)

Strong references should be used when a parent object is referencing a child object and never the other way around. That is, a child class should not have a strong reference to the parent class.

Use a `strong` reference whenever we want to guarantee that we are always able to access the variable. This is especially true for things like object properties which should always exist during their owner's lifetime.

Weak references should be used to avoid retain cycles and an object has the possibility to become nil at any point of its lifetime.

Use a weak reference whenever it is valid for that reference to become nil at some point during its lifetime.

As for **unowned**, it has a very specific use case, for when we know that the pointee's lifecycle is at least as long as the pointer's lifecycle. In real world though, we are always at risk of our assumptions being changed and no longer hold, leading our once safe code to a runtime crash. For this reason, `weak` reference is always safer to use than `unowned`.

Again, this only applies to Classes in Swift. Structs and Enums are value types so using any of the ARC keywords would not be applicable in these cases.