Memory Management

Why is memory management such a big deal?

Because memory management plays a huge role in allocating memory so that the program can perform at the request of the user and could be free for reuse when no longer needed.

What could really happen if you exhaust the memory?

1. The task will stop performing meaning you won't be able to execute any of your task.
2. The task probably will not progress but continue running and running until it hits the limit and the program crashed.
3. You probably don't want the user to use a buggy program.

What is Automatic Reference Counting (ARC)?

- Swift uses Automatic Reference Counting (ARC) to keep tracking and managing all the objects created by the application. This signifies that it handles Memory Management itself and we do not need to take care of it.
- But there are few situations where we as a developer need to provide few other information, in particularly, the relationship between objects, to avoid Memory Leaks.
- Reference Counting applies only to the instances of Reference Type.

How does ARC work?

- Each time you create a class instance through init(), ARC automatically allocates some memory to store the information.

- To be more specific, that chunk of memory holds the instance, together with the values of the properties. When the instance is no longer needed, deinit()

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

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

Retain Cycles

- In Swift, when an object has a **strong association** with another object, it retains it. Then the object I'm talking about is Reference Types, Reference Types, Classes.
- When an object references a second object, it takes ownership. The second Object will remain alive until it is released. This is called a **Strong reference**. Only if you set the property to nil will the second object be destroyed.

```
class Server {
    var clients : [Client] //Because this reference is strong

    func add(client:Client){
        self.clients.append(client)
    }
}

class Client {
    var server : Server //And this one is also strong

    init (server : Server) {
        self.server = server

        self.server.add(client:self) //This line creates a Retain Cycle -> Leak!
    }
}
```

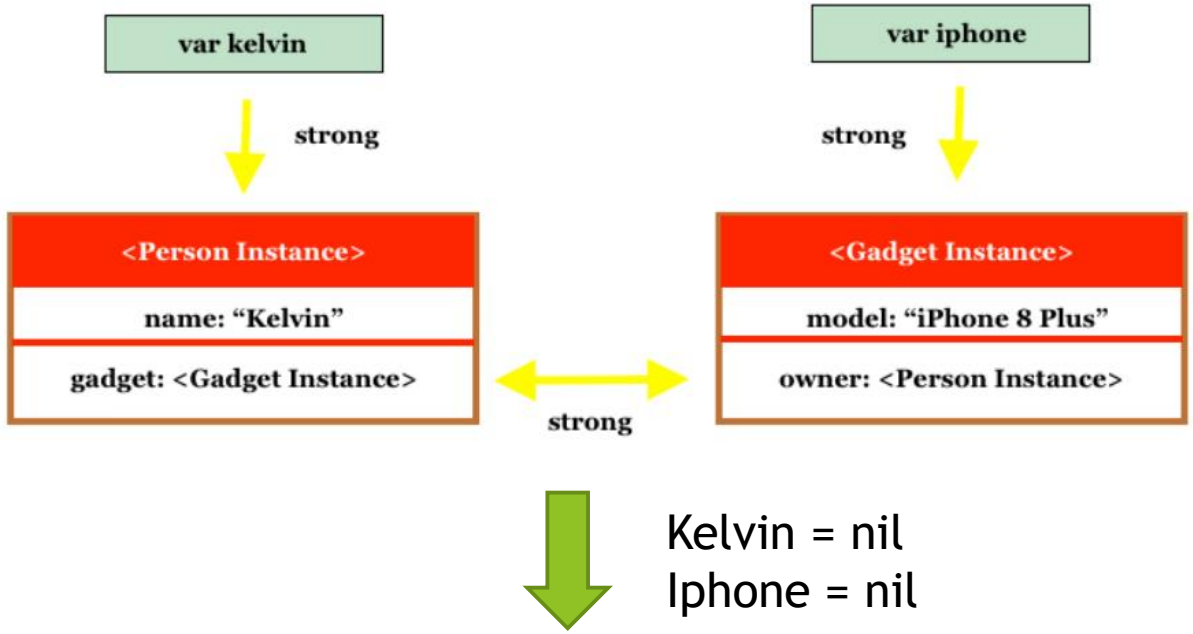
To be freed from memory, an object must first release all its dependencies.

The retain cycles are broken when one of the references in the cycle is **weak** or **unowned.**

Strong vs Weak vs Unowned

- Usually, when a property is being created, the reference is strong unless they are declared weak or unowned.
- With the property labelled as weak, it will not increment the reference count
- An unowned reference falls in between, they are neither strong nor or type optional. Compiler will assume that object is not deallocated as the reference itself remain allocated.

Strong reference



```
26
27  var kelvin: Person?                                  nil
28  var iphone: Gadget?                                  nil
29
30  kelvin = Person(name: "Kelvin")                      Person
31  iphone = Gadget(model: "iPhone 8 Plus")              Gadget
32
33  kelvin!.gadget = iphone                              Person
34  iphone!.owner = kelvin                               Gadget
35
36  kelvin = nil                                         nil
37  iphone = nil                                         nil
39
```

```
Kelvin is being initialized
iPhone 8 Plus is being initialized
```

This is what call strong reference cycle, leading to memory leaks in your apps. To break the strong reference cycle and prevent memory leaks, you will need to use weak and unowned references.

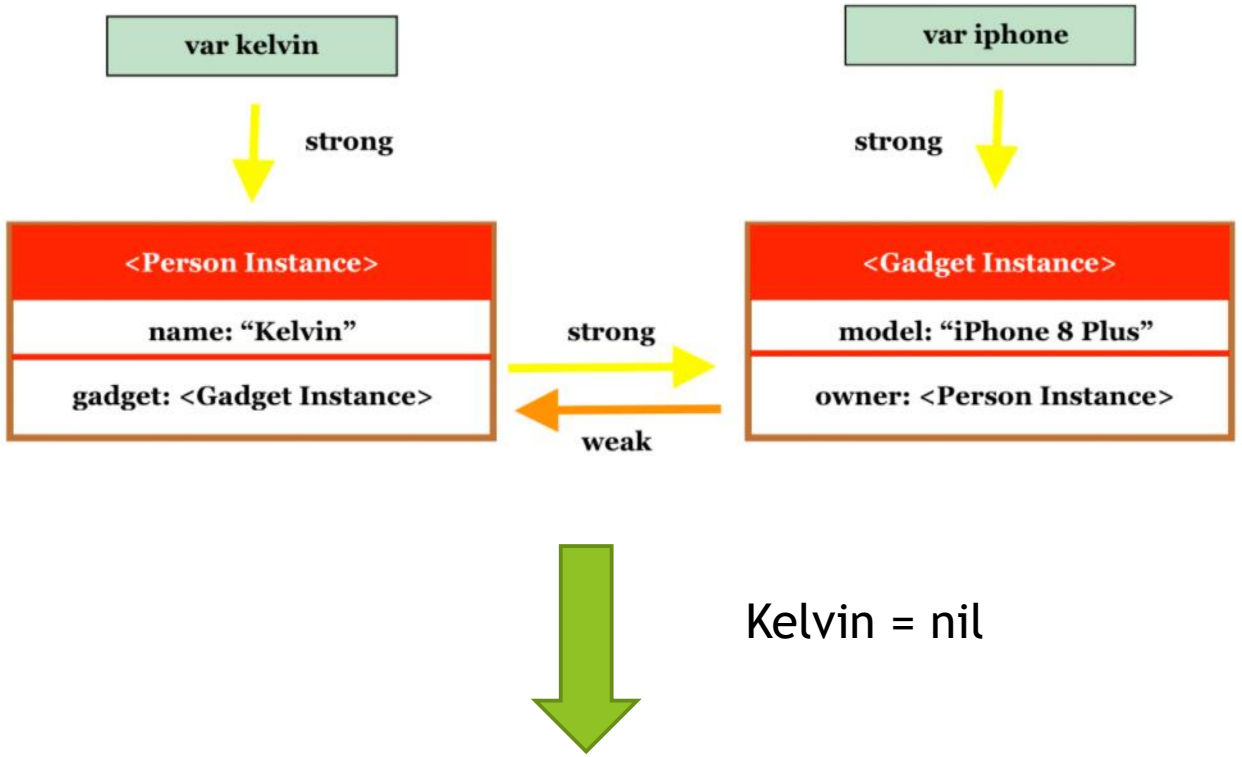
Weak reference

1. Weak reference are always declared as optional types because the value of the variable can be set to nil

2. ARC automatically sets weak reference to nil when the instance is deallocated. And because of the change of value, we know that variable will need to be used here as constants will not let you change the value.

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

```swift
class Gadget {
    let model: String
    init(model: String) {
        self.model = model
        print("\(model) is being initialized")
    }
    weak var owner: Person?
    deinit {
        print("\(model) is being deinitialized")
    }
}
```


var kelvin
strong
var iphone
strong
<Person Instance>
name: “Kelvin”
gadget: <Gadget Instance>
strong
weak
<Gadget Instance>
model: “iPhone 8 Plus”
owner: <Person Instance>
Kelvin = nil
var kelvin
var iphone
strong
<Person Instance>
name: “Kelvin”
gadget: <Gadget Instance>
<Gadget Instance>
model: “iPhone 8 Plus”
owner: nil

```
27  var kelvin: Person?
28  var iphone: Gadget?
29
30  kelvin = Person(name: "Kelvin")
31  iphone = Gadget(model: "iPhone 8 Plus")
32
33  kelvin!.gadget = iphone
34  iphone!.owner = kelvin
35
36  iphone = nil
37  kelvin = nil
38
```

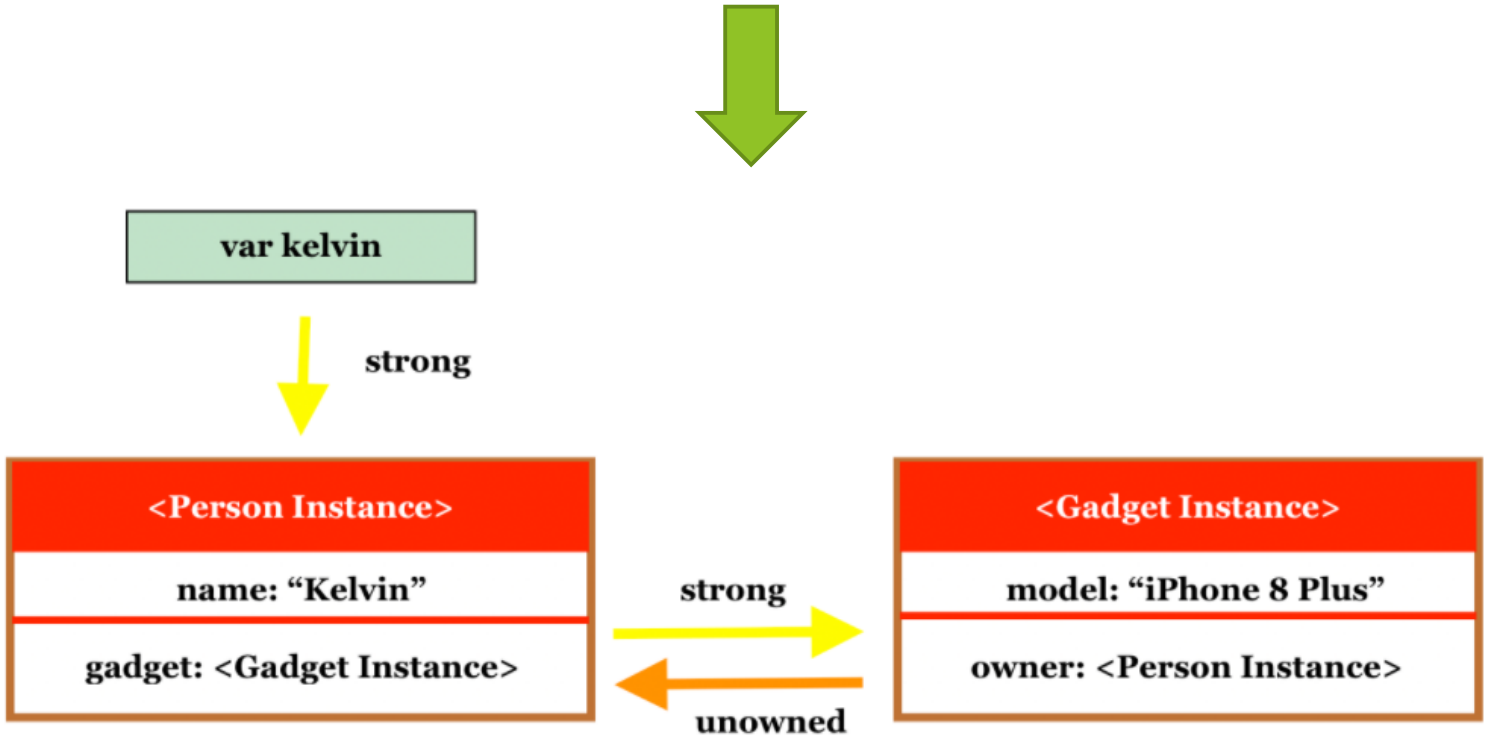
```
Kelvin is being initialized
iPhone 8 Plus is being initialized
Kelvin is being deinitialized
iPhone 8 Plus is being deinitialized
```

Unowned reference

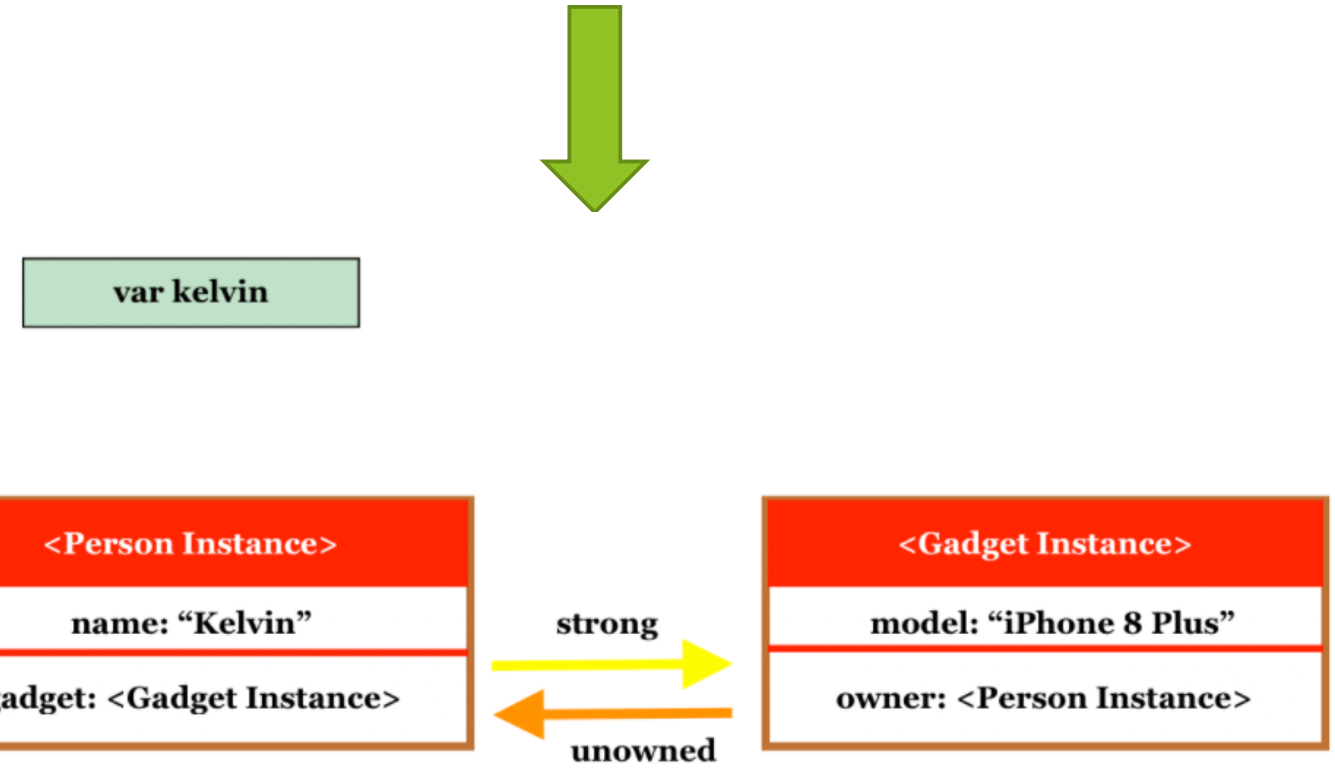
- An unowned reference is very similar to a weak reference that it can be used to resolve the strong reference cycle and the big difference is that an unowned reference **always have a value**.
- ARC will not set unowned reference's value to nil. In other words, the reference is declared as **non-optional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

```
class Gadget {
    let model: String
    unowned var owner: Person

    init(model: String, owner: Person) {
        self.model = model
        self.owner = owner
        print("\(model) is being initialized")
    }

    deinit {
        print("\(model) is being deinitialized")
    }
}
```

```swift
var kelvin: Person?


kelvin = Person(name: "Kelvin")
kelvin!.gadget = Gadget(model: "iPhone 8 Plus", owner: kelvin!)
```



```
kelvin = nil
```




```
Kelvin is being initialized
iPhone 8 Plus is being initialized
Kelvin is being deinitialized
iPhone 8 Plus is being deinitialized
```

Structures and Classes

Comparing Structures and Classes

		Struct	Class
Init		Optional	Required
Inheritance		NO	YES
Override		NO	YES
Destruction function		NO	Deinit
Change attribute value	Let	NO	YES
	Var	YES	YES
		Value Types	Reference Types

Change attribute value

```
class test{
    var name : String = ""
    func fullname(lastname : String) -> String {
        name = name + lastname
        return  name
    }
}

let test2 = test()                                                    test
test2.name = "ok1"                                                    test
print(test2.name)                                                     "ok1\n"

struct testStruct{
    var name : String = ""
    mutating func fullname(lastname : String) -> String {
        name = name + lastname
        return  name
    }
}

let test1 = testStruct()                                              testStruct
test1.name = "ok"    Cannot assign to property: 'test1' is a 'let' constant   testStruct
print(test1.name)                                                     "ok\n"
```

When to use Struct?

- Use Struct when creating objects with many properties of simple data types, usually of value type (like Int, Float, String, ...)

- When working multi-threaded. For example, the database connection is made on a thread parallel to the Main thread, using Struct is safer because it can copy values from one thread to another.

- Struct does not need to inherit properties or behavior from any other type.

- Using Struct will ensure that no part of the code gets a reference to our objects unless we directly access them. Therefore, it is easy to manage, and it becomes simpler to control the object value when it is changed.


Thanks for
Watching