# "WEAK, STRONG, UNOWNED, OH MY!" - A GUIDE TO REFERENCES IN SWIFT

June 25, 2015
*by* Hector Matos

---

I often find myself worrying about retain cycles in my code. I feel like this is a common concern amongst others as well. I don't know about you, but it seems like I am constantly hearing "When am I supposed to use weak? And what the hell is this 'unowned' crap?!" The issue we find is that we know to use strong, weak, and unowned specifiers in our swift code to avoid retain cycles, but we don't quite know which specifier to use. Fortunately, I happen to know what they are AND when to use them! I hope this guide helps you to learn when and where to use them on your own.

## LET'S GET STARTED

### ARC

ARC is a compile time feature that is Apple's version of automated memory management. It stands for *Automatic Reference Counting.* This means that it ***only*** frees up memory for objects when there are ***zero strong*** references to them.

### STRONG

Let's start off with what a strong reference is. It's essentially a normal reference (pointer and all), but it's special in it's own right in that it ***protects*** the referred object from getting deallocated by ARC by increasing it's retain count by 1. In essence, ***as long as anything*** has a strong reference to an object, it will not be deallocated. This is important to remember for later when I explain retain cycles and stuff.

Strong references are used almost everywhere in Swift. In fact, the declaration of a property is strong by default! Generally, we are safe to use strong references when the hierarchy relationships of objects are **linear.** When a hierarchy of strong references flow from parent to child, then it's always ok to use strong references.

Here is an example of strong references at play.

```swift
class Kraken {
    let tentacle = Tentacle() //strong reference to child.
}
class Tentacle {
    let sucker = Sucker() //strong reference to child
}

class Sucker {}
```

Here we have a linear hierarchy at play. `Kraken` has a strong reference to a `Tentacle` instance which has a strong reference to a `Sucker` instance. The flow goes from Parent (`Kraken`) all the way down to child (`Sucker`).

Similarly, in animation blocks, the reference hierarchy is similar as well:

```swift
UIView.animate(withDuration: 0.3) {
    self.view.alpha = 0.0
}
```

Since `animateWithDuration` is a static method on `UIView`, the closure here is the parent and self is the child.

What about when a child wants to reference a parent? Here is where we want to use weak and unowned references.

# WEAK AND UNOWNED REFERENCES

## WEAK

A weak reference is just a pointer to an object that ***doesn't protect*** the object from being deallocated by ARC. While strong references increase the retain count of an object by 1, weak references ***do not***. In addition, weak references zero out the pointer to your object when it successfully deallocates. This ensures that when you access a weak reference, it will either be a valid object, or nil.

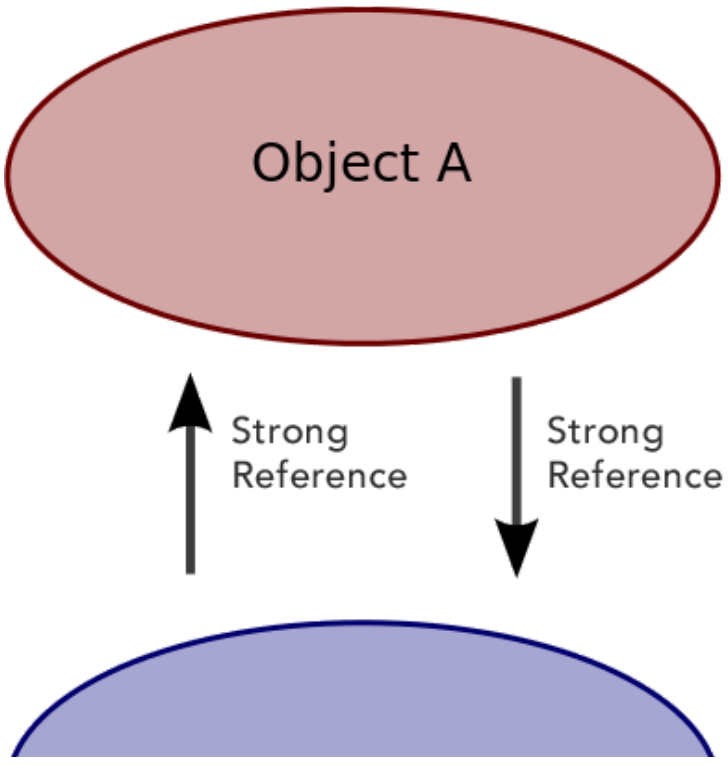
In Swift, all weak references are non-constant `Optionals` (think `var` vs. `let`) because the reference **can** and **will** be mutated to nil when there is no longer anything holding a strong reference to it.

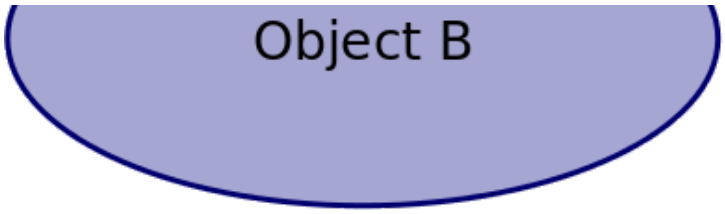
For example, this won't compile:

```swift
class Kraken {
    //let is a constant! All weak variables MUST be mutable.
    weak let tentacle = Tentacle()
}
```

because `tentacle` is a `let` constant. `Let` constants by definition cannot be mutated at runtime. Since weak variables can be nil if nobody holds a strong reference to them, the Swift compiler requires you to have weak variables as `var`s.

Important places to use weak variables are in cases where you have potential ***retain cycles.*** A retain cycle is what happens when two objects both have ***strong*** references to each other. If 2 objects have strong references to each other, ARC will not generate the appropriate `release` message code on each instance since they are keeping each other alive. Here's a neat little image from Apple that nicely illustrates this:






A perfect example of this is with the (fairly new) `NSNotification` APIs. Take a look at the codes:

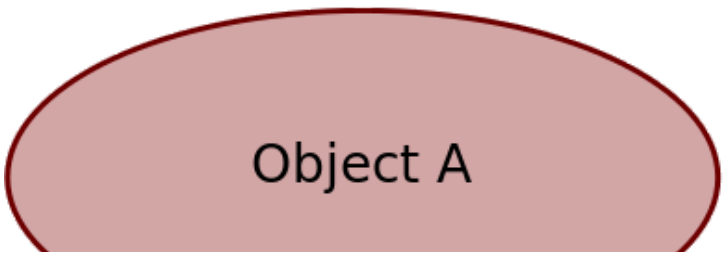
```swift
class Kraken {
    var notificationObserver: ((Notification) -> Void)?
    init() {
        notificationObserver = NotificationCenter.default.addObserve
            self.eatHuman()
        }
    }

    deinit {
        NotificationCenter.default.removeObserver(notificationObserv
    }
}
```

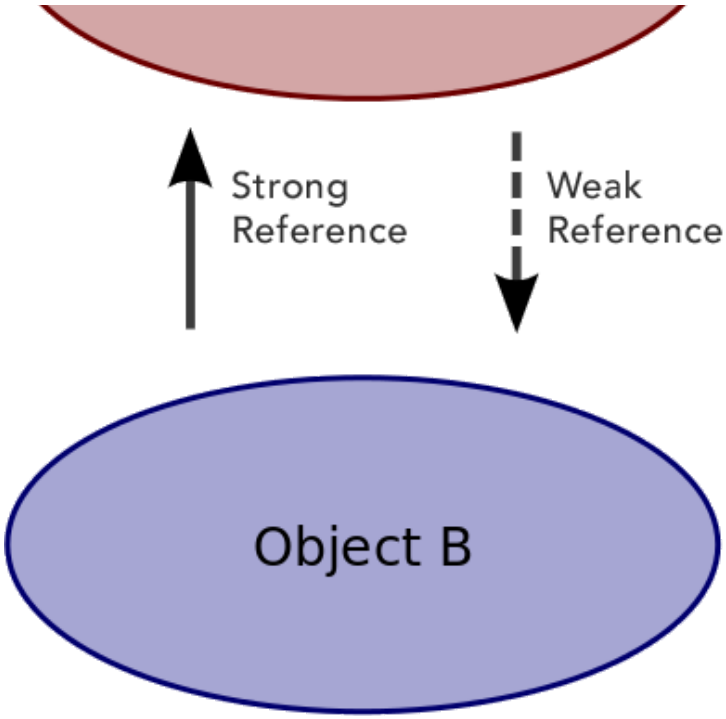
At this point we have a retain cycle. You see, Closures in swift behave exactly like blocks in Objective-C. If any variable is declared ***outside*** of the closure's scope, referencing that variable ***inside*** the closure's scope creates ***another strong reference*** to that object. The only exceptions to this are variables that use value semantics such as `Ints`, `Strings`, `Arrays`, and `Dictionaries` in Swift.

Here, `NotificationCenter` retains a closure that captures self strongly when you call `eatHuman()`. Best practice says that you clear out notification observers in the deinit function. The problem here is that we don't clear out the block until `deinit`, but `deinit` won't ever be called by ARC because the closure has a strong reference to the Kraken instance!

Other gotchas where this could happen is in places like `NSTimers` and `NSThread.`

The fix is to use a weak reference to `self` in the closure's *capture list.* This breaks the strong reference cycle. At this point, our object reference graph will look like this:






Changing self to weak won't increase `self` 's retain count by 1, therefore allowing to ARC to deallocate it properly at the correct time.

To use `weak` and `unowned` variables in a closure, you use the [] in syntax inside of the closure's body. Example:

```swift
let closure = { [weak self] in
    self?.doSomething() //Remember, all weak variables are Optionals
}
```

Why is the weak self inside of square brackets? That looks weird! In Swift, we see square brackets and we think `Arrays` . Well guess what? You can specify multiple capture values in a closure! Example:

```swift
//Look at that sweet, sweet Array of capture values.
let closure = { [weak self, unowned krakenInstance] in
    self?.doSomething() //weak variables are Optionals!
    krakenInstance.eatMoreHumans() //unowned variables are not.
}
```

That looks more like an `Array` right? So, now you know why capture values are in square brackets. So, now, using what we've learned so far, we can fix the retain cycle in the notification code we posted above by adding [weak self] to the closure's capture list:

```swift
NotificationCenter.default.addObserver(forName: "humanEnteredKrakens
    self?.eatHuman() //self is now an optional!
}
```

One other place we need to use weak and unowned variables is when using protocols to employ delegation amongst ***classes*** in Swift, since classes use reference semantics. In Swift, structs and enums can conform to protocols as well, but they use value semantics. If a parent class uses delegation with a child class like so:

```swift
class Kraken: LossOfLimbDelegate {
    let tentacle = Tentacle()
    init() {
        tentacle.delegate = self
    }

    func limbHasBeenLost() {
        startCrying()
    }
}

protocol LossOfLimbDelegate {
    func limbHasBeenLost()
}

class Tentacle {
    var delegate: LossOfLimbDelegate?

    func cutOffTentacle() {
        delegate?.limbHasBeenLost()
    }
}
```

Then we need to use a weak variable.

Here's why:

`Tentacle` in this case holds a strong reference to `Kraken` in the form of it's `delegate` property.

AT THE SAME TIME

`Kraken` holds a strong reference to `Tentacle` in it's `tentacle` property.

To use a weak variable in this scenario, we add a weak specifier to the beginning of the delegate declaration:

```swift
weak var delegate: LossOfLimbDelegate?
```

What's that you say? Doing this won't compile?! Well, the problem is because non class type protocols cannot be marked as weak.

At this point, we have to use a class protocol to mark the delegate property as weak by having our protocol inherit `:class`.

```swift
protocol LossOfLimbDelegate: class { //The protocol now inherits cla
    func limbHasBeenLost()
}
```

When do we not use `:class` ? Well according to Apple:

> *"Use a class-only protocol when the behavior defined by that protocol's requirements assumes or requires that a conforming type has reference semantics rather than value semantics."*

Essentially, if you have a reference hierarchy exactly like the one I showed above, you use `:class`. In struct and enum situations, there is no need for `:class` because structs and enums use value semantics while classes use reference semantics.

## UNOWNED

Weak and unowned references behave similarly but are NOT the same. Unowned references, like weak references, ***do not*** increase the retain count of the object being referred. However, in Swift, an unowned reference has the added benefit of ***not being an Optional.*** This makes them easier to manage rather than resorting to using optional binding. This is not unlike Implicitly Unwrapped Optionals . In addition, unowned references are non-zeroing. This means that when the object is deallocated, it does not zero out the pointer. This means that use of unowned references can, in some

cases, lead to dangling pointers. For you nerds out there that remember the Objective-C days like I do, unowned references map to `unsafe_unretained` references.

This is where it gets a little confusing. Weak and unowned references both do not increase retain counts. They can both be used to break retain cycles. So when do we use them?! According to Apple's docs:

> *"Use a weak reference whenever it is valid for that reference to become nil at some point during its lifetime. Conversely, use an unowned reference when you know that the reference will never be nil once it has been set during initialization."*

Well there you have it: Just like an implicitly unwrapped optional, If you can ***guarantee*** that the reference will not be nil at its point of use, use unowned. If not, then you should be using weak.

Here's a good example of a class that creates a retain cycle using a closure where the captured self will not be nil:

swift

```swift
class RetainCycle {
    var closure: (() -> Void)!
    var string = "Hello"

    init() {
        closure = {
            self.string = "Hello, World!"
        }
    }
}

//Initialize the class and activate the retain cycle.
let retainCycleInstance = RetainCycle()
retainCycleInstance.closure() //At this point we can guarantee the ca
```

In this case, the retain cycle comes from the closure capturing self strongly while self has a strong reference to the closure via the closure property. To break this we simply add `[unowned self]` to the

closure assignment:

```swift
closure = { [unowned self] in
    self.string = "Hello, World!"
}
```

In this case, we can assume self will never be nil since we call closure immediately after the initialization of the RetainCycle class.

Apple also says this about unowned references:

> *"Define a capture in a closure as an unowned reference when the closure and the instance it captures will always refer to each other, and will always be deallocated at the same time."*

If you know your reference is going to be zeroed out properly and your 2 references are ***MUTUALLY DEPENDENT*** on each other (one can't live without the other), then you should prefer unowned over weak, since you aren't going to want to have to deal with the overhead of your program trying to unnecessarily zero your reference pointers.

A really good place to use unowned references is when using self in closure properties that are lazily defined like so:

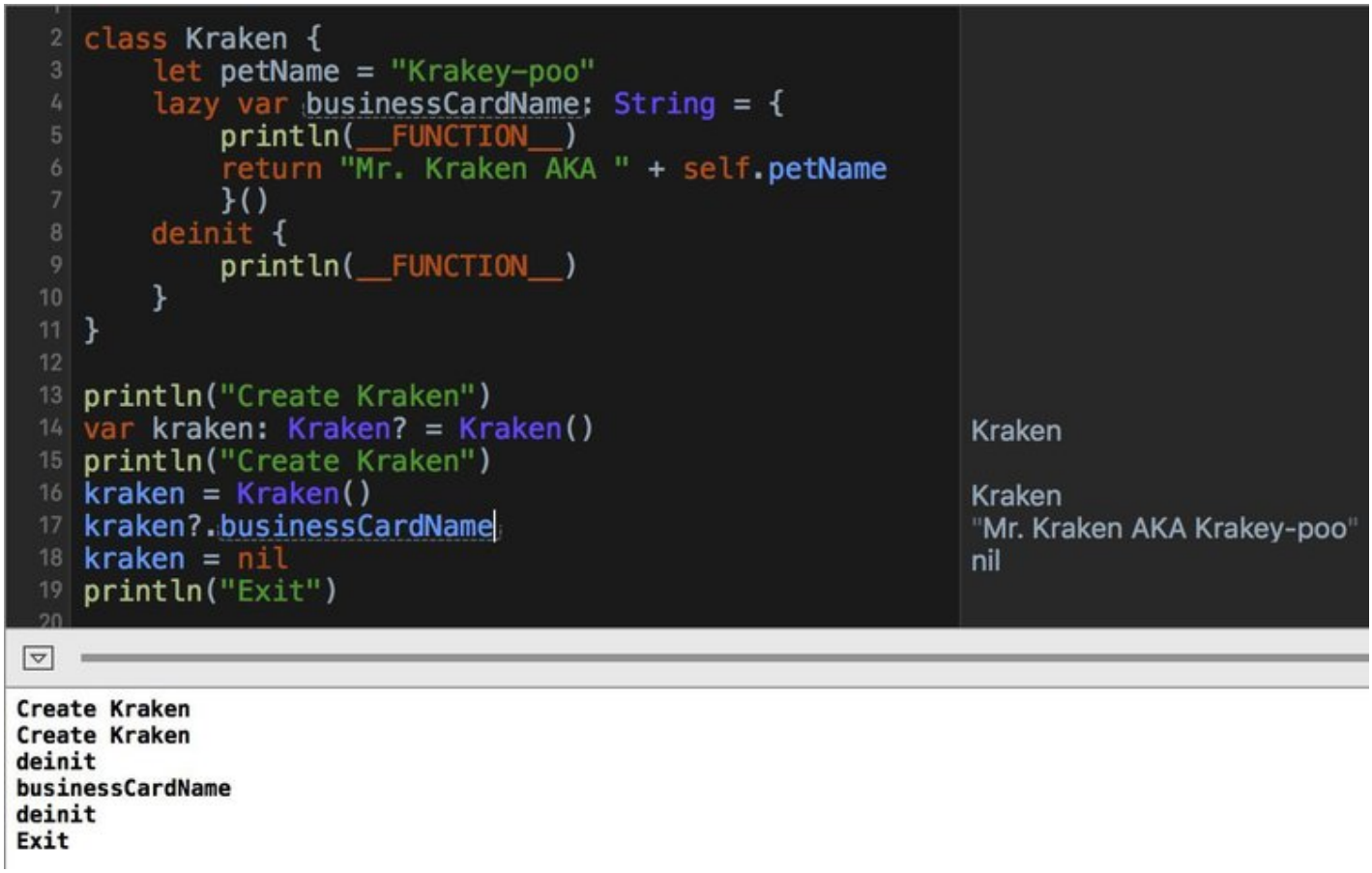
```swift
class Kraken {
    let petName = "Krakey-poo"
    lazy var businessCardName: (Void) -> String = { [unowned self] in
        return "Mr. Kraken AKA " + self.petName
    }
}
```

We need unowned self here to prevent a retain cycle. `Kraken` holds on to the `businessCardName` closure for it's lifetime and the `businessCardName` closure holds on to the `Kraken` for it's lifetime. They are mutually dependent, so they will always be deallocated at the same time. Therefore, it satisfies the rules for using unowned!

HOWEVER, this is not to be confused with lazy variables that ***AREN'T*** closures such as this:

```swift
class Kraken {
    let petName = "Krakey-poo"
    lazy var businessCardName: String = {
        return "Mr. Kraken AKA " + self.petName
    }()
}
```

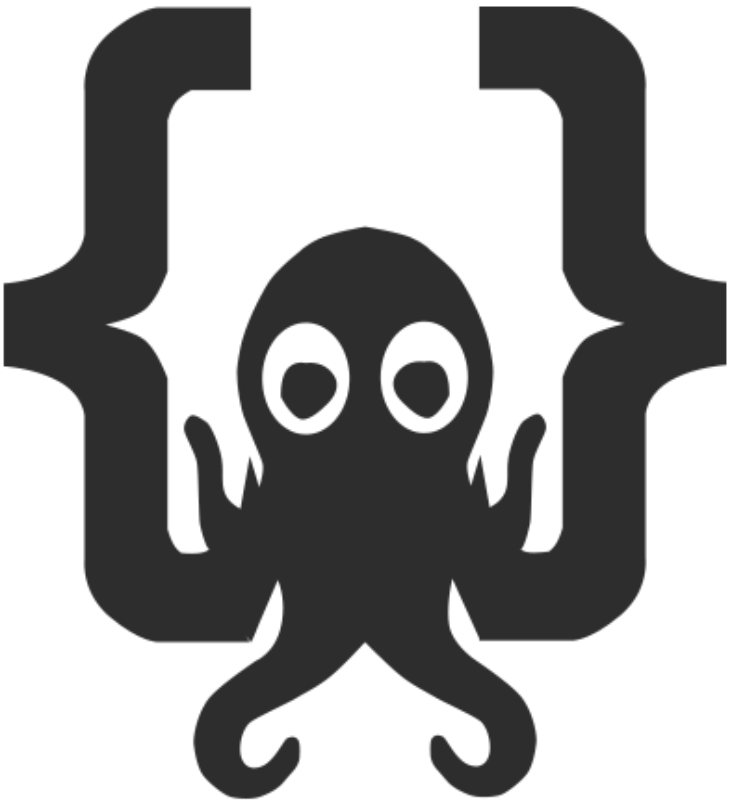
`Unowned self` is not needed since nothing actually retains the closure that's called by the lazy variable. The variable simply assigns itself the result of the closure and deallocates the closure (and decrements the captured self's reference count) immediately after it's first use. Here's a screenshot that proves this! (Screenshot taken shamelessly from Алексей in the comments section!)




# CONCLUSION

Retain cycles suck. But with careful coding and consideration of your reference hierarchy, memory leaks and abandoned memory can be avoided through careful use of weak and unowned. I hope this guide helps you in your endeavors.

Happy coding fellow nerds!



26 Comments ♥ 56 Likes Share

24 Comments KrakenDev 1 Login

Recommend 37 Tweet Share Sort by Newest

Join the discussion…

LOG IN WITH

OR SIGN UP WITH DISQUS ?

Name

yojkim • a month ago

Thank you!

Reply • Share ›

Mihael Isaev • 5 months ago

This is more than awesome article! Thank you very much for your work!!!!!!!

Reply • Share ›

**Sudhanshu** • 7 months ago

This is an awesome article. Perfectly elaborated. Thanks

Reply • Share ›

**Jura Shikin** • 10 months ago

I think using unowned self in closures, like in example ‘businessCardName‘, very dangerous!
I can save link to this closure, deallocate object, after, try execute closure - we receive crash :(

Reply • Share ›

**Padmaja An** • a year ago

Great article. All of sudden everything makes sense. I am trying to understand a feasible explanation for not using an [unowned self] in UIView.animate blocks. I referred a couple of books for learning about animations, but none of them mentioned using [unowned self] in the animations closure. For instance: when you call the following in a view controller

```
self.view.layoutIfNeeded()
UIView.animate(withDuration: 0.5,
animations: {
self.testView.layer.backgroundColor = UIColor.black.withAlphaComponent(0.50).cgColor
self.view.layoutIfNeeded()
},
completion: { _ in
self.testView.removeFromSuperview()
}
)
```

What I understand is, the animations closure is being called inline, but no where it is mentioned if I have to capture [unowned self]. I am trying to figure out why, do you have a feasible explanation?

1 Reply • Share ›

**Hector Matos** Mod ➚ Padmaja An • a year ago

Thank you so much for reading! The reason you don't need to specially capture self within animations is because there isn't a cyclic reference. Regardless of how the closure is being executed, the only way for a cyclic reference to occur is if self stores a closure property that capture self for animations, and then feeds it to the animation API.

No matter what the API is, you should always check the API's function declarations. If the closure parameter use marked as @escaping, then your closure can be stored somewhere, therefore introducing the capability of storing a capture self longer than need be.

TL;DR - don't worry about animation APIs. Write self as much as you want, just don't feed a stored closure property within self that capture self to the animation API.

1 Reply • Share ›

**jrock_wh** • a year ago

best article i have read in regard to arc and references. thank you.

2 Reply • Share ›

**Bomi Chen** • a year ago

Super great articles. This is the most clear & complete article about ARC that I've ever read.
Thanks a lot !

1 • Reply • Share ›

**eford** • a year ago

Really great article. Thanks 👍

1 • Reply • Share ›

**Mike Critchley** • 2 years ago

Awesome! Thanks!

1 • Reply • Share ›

**Geo** • 2 years ago

Excellent Article! Kudos. But I have a small question.. What is the difference b/w ownership qualifiers and modifiers?

• Reply • Share ›

**Hector Matos** Mod ➦ Geo • 2 years ago

In that image it explains it pretty well. The _qualifiers_ are how the reference is retained. That can be as auto releasing, weak, or strong. The _modifiers_ are the semantics of the retainment. For example, structs in swift are under the copy modifier. They are still strong references, but on assignment they get copied. This means that structure references inside closures don't have to worry about leaking memory since a completely different reference by the time it's assigned inside the captured value.

1 • Reply • Share ›

**Geo** ➦ Hector Matos • 2 years ago

Can you please explain this with regards to "Assign" modifier. I believe I used to use that

for delegates (property declaration) in Objective C (back in 'of days).

Reply • Share ›

Show more replies

**Felipe Ricieri** • 2 years ago

Neat! Thank you very much Hector 👊

Reply • Share ›

**ungato** • 2 years ago

I don't get it though. If unowned is for when two objects are mutually dependent on each other existing, then wouldn't unowned not work because if you only had a strong reference to one of the objects, then the other would be deallocated. Am I missing something?

Reply • Share ›

**Hector Matos** Mod ungato • 2 years ago

I would say it's more accurate to say that it's really only safe if both are strong references that are mutually dependant on eachother. For example a singleton that owns a block that captures another object that can be deallocated sometime in the future.

The thing you're missing is that for weak AND unowned they are both used to break strong references cycles. The example that you gave wouldn't be applicable because there are not two strong references referencing eachither. If you can understand mechanics behind weak, and why it's needed, then you can understand unowned. The only difference is that weak gives you an optional type in the closure. In contrast, unowned gives you an implicitly unwrapped optional in the closure.

Reply • Share ›

**ungato** ungato • 2 years ago

EDIT: cated. -> cated?

Reply • Share ›

**NSAmi** • 2 years ago

This is wonderful..... Conceptually cleared everything. Much better than docs for sure.

Reply • Share ›

**Stefan DeClerck** • 2 years ago

This is awesome! thanks!

50 Reply • Share ›

**Sudara Madushan** • 2 years ago

Nice article, great explanation

Reply • Share ›

**JaponicaGr** • 2 years ago

Thanks a lot Hector! Great explanation in an easily misunderstood subject.

3 Reply • Share ›

**juan pablo Velasquez** • 2 years ago

Is it necessary to use unowned for computed properties in classes that are not lazy? For example

Is it necessary to use unowned for computed properties in classes that are not lazy? For example

class MyObject: NSObject {

var otherObject = OtherObjectWithName("Kraken")
var someGreetingComputedProperty: String { //Is unowned needed here?
return "Hello \(otherObject.name)"
}
}

As far as I understand it may not be needed for structs or if we use value types inside the computed property

Reply • Share ›

**Hector Matos** Mod juan pablo Velasquez • 2 years ago

Computed properties do not need unowned references. There's no chance of a reference cycle there so you should be good. When in doubt, implement a deinit to see if it gets called. If it doesn't then chances are you are doing something that needs an unowned or weak reference somewhere. From what I understand, you can think of computed variables as a fancy function. This would be the equivalent of your variable:

func someGreetingComputedProperty() -> String {
return "Hello \(otherObject.name)"
}

Also, try putting an unowned self in that computed variable. I highly doubt the compiler will even let you.

Reply • Share ›

**Samuel Weir** • 2 years ago

Thanks for the excellent and clearly written article. It helped my understanding a lot.

Reply • Share ›

*Posted in* Swift *and*
*tagged with* value semantics, protocol, closure, reference semantics, weak reference, break retain cycle, iOS programming, references, capture list, strong, weak, capture lists, unowned reference, retain cycles, ARC, unowned, Mac OS X, retain cycle, iOS, strong reference, programming, swift language, delegation, memory management, swift

Newer / Older