

Automatic reference counting é a forma que o Swift lida com a alocação e desalocação de recursos de memória para Classes que são passadas por referência e não valor, como structs e enums.

Como funciona o ARC, há a criação de uma classe:

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

Note como a classe ainda não é usada em código, o que implica que nenhuma memória foi alocada pra ela. Criamos uma var optional com seu tipo que ainda não recebe valor:

```
var reference1: Person?
```

O print ainda não vai disparar, a classe ainda não foi inicializada.

```
reference1 = Person(name: "John Appleseed")
```

Agora teríamos um print nesse momento de "John Appleseed is being initialized".

Agora reference1 tem uma referência forte a Person, como há pelo menos uma referência forte a Person, o ARC se certifica que ele está sendo alocado em memória. Agora se fizermos:

```
reference1 = nil
```

O ARC retira o Person da alocação da memória e o print do deinit seria disparado.

Há um caso quando uma classe mantém a outra viva na memória e é chamado de **Strong Reference Cycle**

E isso gera problemas de memory leak, acontece quando:

```
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") }  
  
}
```

Cria duas classes em que ambas possuem um optional que começa como nil de uma outra classe.

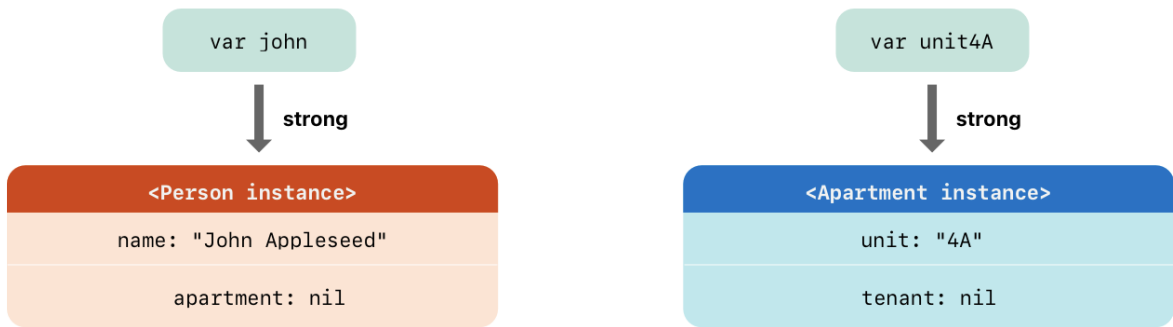
Você cria a variável das classes:

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

Da valor para eles:

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

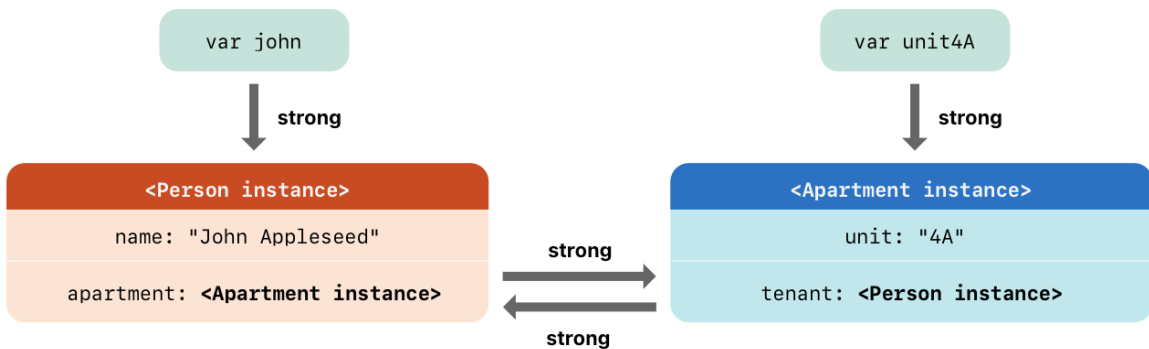
Gerando assim:



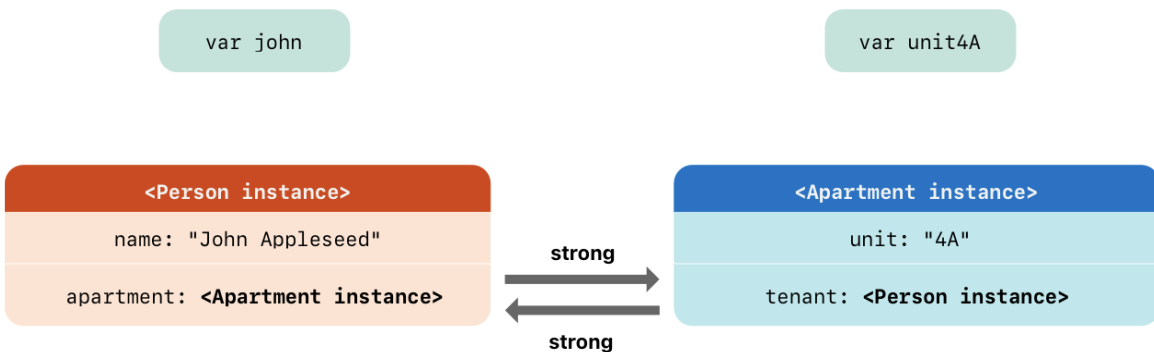
Agora supostamente se déssemos nil pro john e pro unit4A o ARC liberaria a memória.

Só que ai:

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



Um vai ter referência forte pro outro e mesmo dando nil pra var john e unit4A a memória nao desalocaria e teríamos vazamento de memória.



Como resolver

Usando weak e unowned references, weak quando tiver um lifetime menor e unowned quando a outra instância tiver um lifetime parecido ou maior.

Uso do weak reference:

São declarados como var e optional.

Weak permite que o ARC libere a memória daquela variável. E como em runtime o valor precisa ser mudado para nil ele tem que ser var e optional.

Uso da weak var

```
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") }  
  
}
```

Veja que o Person no Apartment ta declarado como weak var optional Person, já que ele precisa poder receber nil em algum momento. Usamos o tenant que é uma Person como weak porque ele tem um lifetime menor e pode ser desalocado antes do Apartment.

Damos as mesmas propriedades:

```

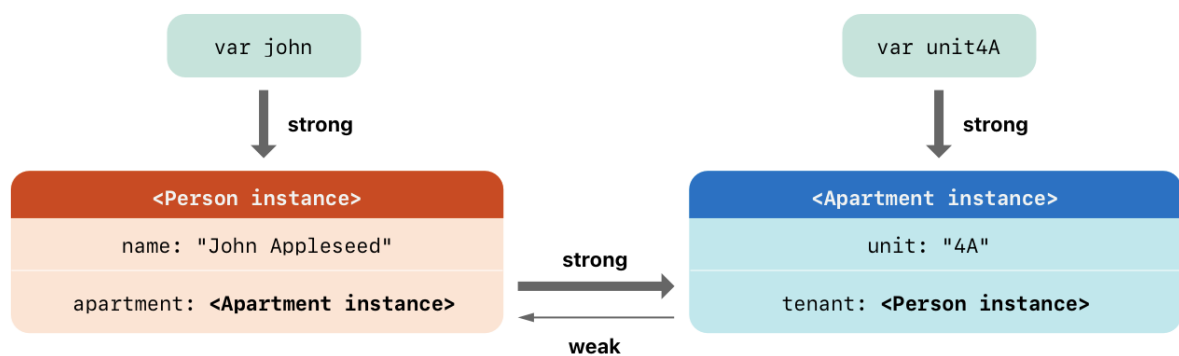
var john: Person?
var unit4A: Apartment?

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

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

```

E agora ficou desse jeito:



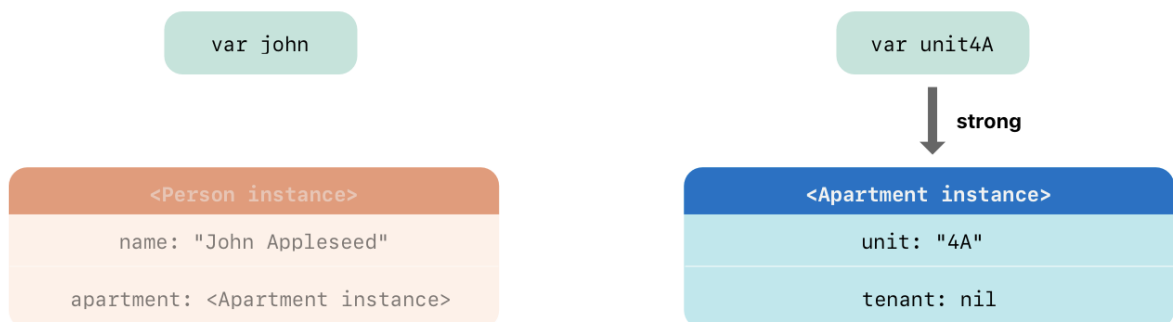
Como julgamos que john seria desalocado primeiro fazemos:

```

john = nil
// Prints "John Appleseed is being deinitialized"

```

E como ele não está apoiado pelo Apartment mais, o ARC desaloca ele:



Se dermos um nil pro unit4A ele também será desalocado.

Uso de unowned reference

Funciona bem quando um pode ser nil e o outro não

O que vamos fazer aqui é, vamos referenciar algo como unowned quando essa var tem o lifetime igual ou maior, temos:

```
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") }
}
```

Isso implica que o customer tem que ter lifetime maior ou igual que o CreditCard já que o credit card só existe se um cliente existir.

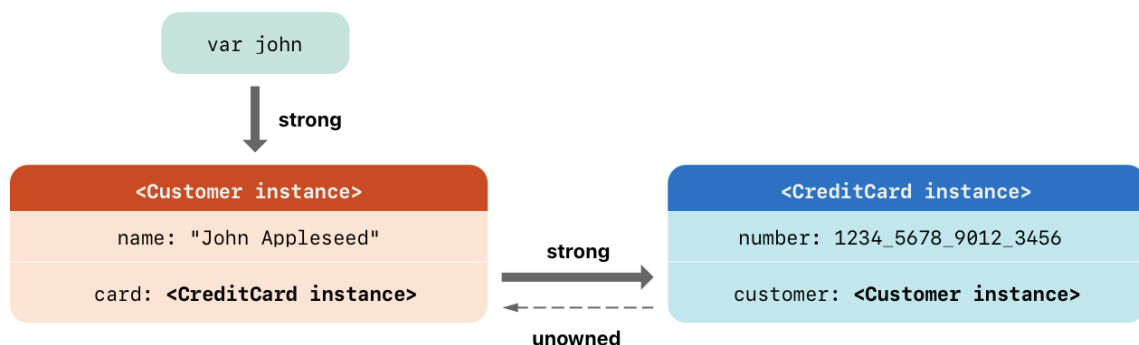
Criamos inicialmente um cliente:

```
var john: Customer?
```

Inicializamos o cliente e já criamos o creditCard dele.

```
john = Customer(name: "John Appleseed")
```

```
john!.card = CreditCard(number: 1234_5678_9012_3456, customer: john!)
```



Um está referenciando o outro.

Se dermos um john = nil, desaloca as duas classes.

Strong Reference Cycles for Closures

Uma closure pode gerar um strong reference cycle quando usa um `self.algumaCoisa` pra poder acessar algo dentro do corpo da closure.

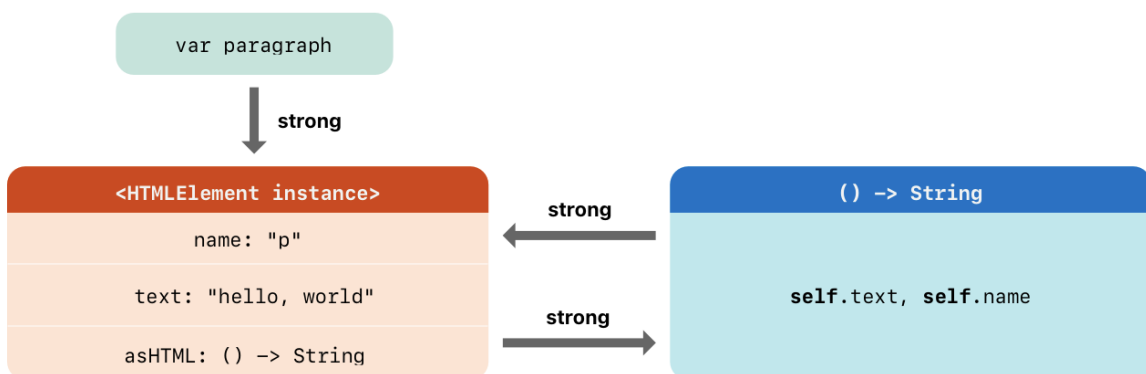
E ocorre porque closures são reference types também.

```
class HTMLElement {
    let name: String
    let text: String?
    lazy var asHTML: () -> String = {
        if let text = self.text {
            return "<\(self.name)>\(text)</\ \(self.name)>"
        } else {
            return "<\(self.name) />"
        }
    }
    init(name: String, text: String? = nil) {
        self.name = name
        self.text = text
    }
    deinit {
        print("\(name) is being deinitialized")
    }
}
```

Veja que a closure usa o `self`.

```
1 var paragraph: HTMLElement? = HTMLElement(name: "p", text: "hello, world")
2 print(paragraph!.asHTML())
3 // Prints "<p>hello, world</p>"
```

Se usarmos a closure fica assim:



E mesmo se dermos `nil` no `paragraph` ambas vão dar memory leak ainda.

Resolvemos usando uma **capture list**

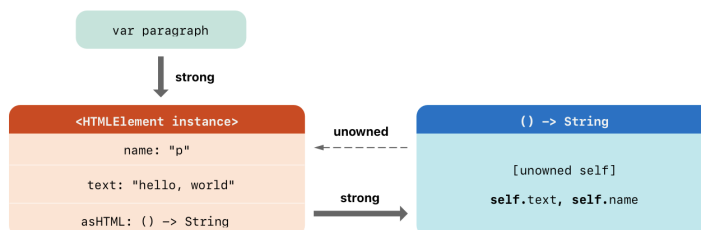
```
lazy var someClosure = {
    [unowned self, weak delegate = self.delegate]
    (index: Int, stringToProcess: String) -> String in
    // closure body goes here
}
```

Usamos unowned quando a closure e a instância se referem a si mesmos e vão sempre ser desalocadas ao mesmo tempo. E use weak quando a variável a que se refere pode ser nil em algum momento do código.

```
1 class HTMLElement {
2
3     let name: String
4     let text: String?
5
6     lazy var asHTML: () -> String = {
7         [unowned self] in
8         if let text = self.text {
9             return "<\(self.name)>\(text)</\(\(self.name))>"
10        } else {
11            return "<\(self.name) />"
12        }
13    }
14
15    init(name: String, text: String? = nil) {
16        self.name = name
17        self.text = text
18    }
19
20    deinit {
21        print("\(name) is being deinitialized")
22    }
23
24 }
```

```
1 var paragraph: HTMLElement? = HTMLElement(name: "p", text: "hello, world")
2 print(paragraph!.asHTML())
3 // Prints "<p>hello, world</p>"
```

Here's how the references look with the capture list in place:



Dai se dermos um nil no paragraph o ARC libera a memória normalmente.