SWIFT

MEMORY MANAGEMENT

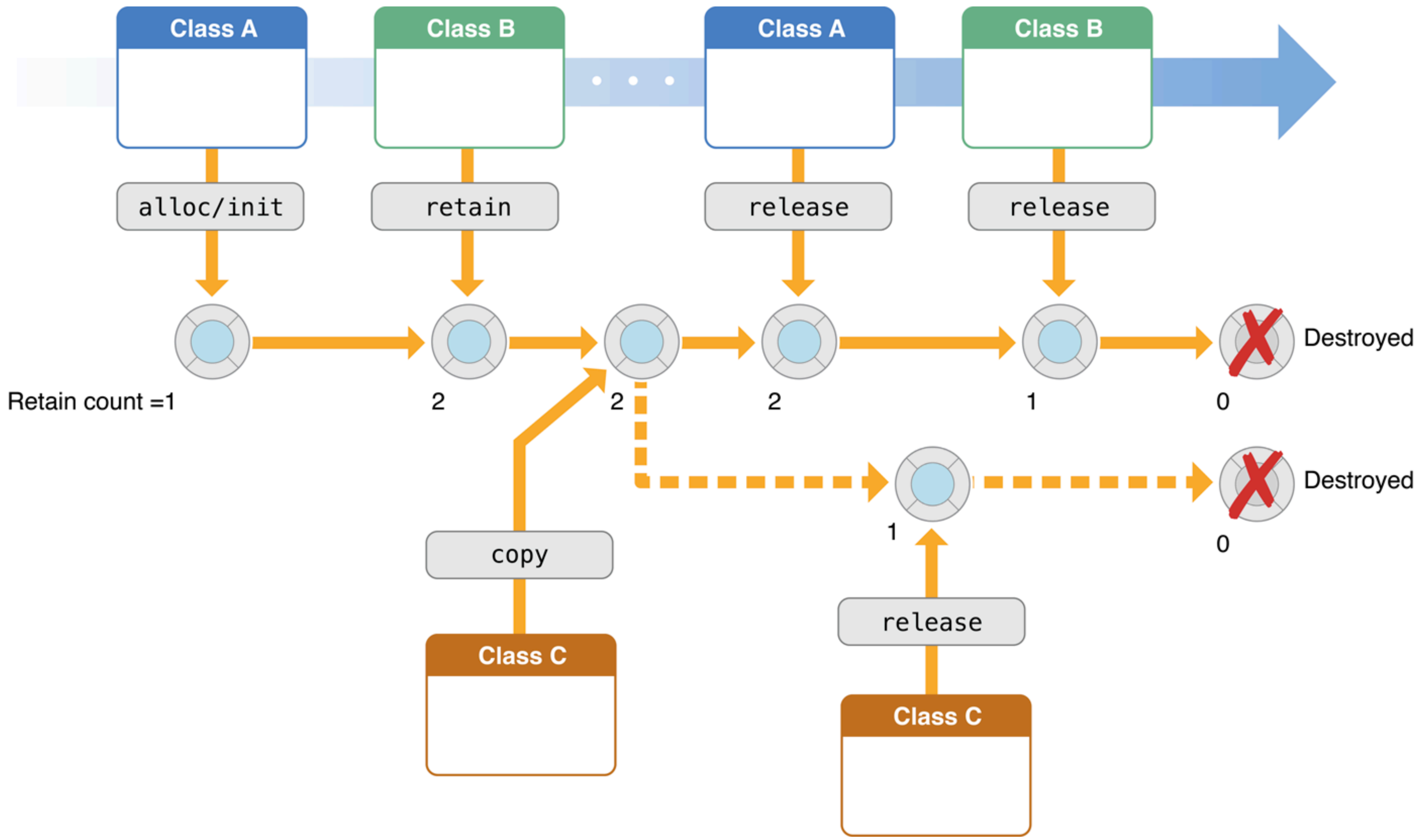
A WELL-WRITTEN PROGRAM USES AS LITTLE MEMORY AS POSSIBLE.

Apple Inc.

Your goal is actually to manage object graphs.

You want to make sure that you have no more objects in memory than you actually need.

WHAT WE WILL LEARN TO UNDERSTAND



METHODS

- MRR, “manual retain-release” – Objective-C
  - you explicitly manage memory by keeping track of objects you own
  - this is implemented using a model, known as reference counting, that class NSObject provides in conjunction with the runtime environment
- ARC, Automatic Reference Counting – Objective-C, Swift
  - the system uses the same reference counting system as MRR, but it inserts the appropriate memory management method calls for you at compile-time
  - see Transitioning to ARC Release Notes

ARC

- Xcode 4.2+
- new rules
- new lifetime qualifiers
- weak references

HOW ARC WORKS

- Every time you create a new instance of a class, ARC allocates a chunk of memory to store information about the type of the instance and any stored associated properties.
- When an instance is no longer needed, ARC frees up the memory used by that instance.
- 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.
- Whenever you assign a class instance to a property, constant, or variable, that property, constant, or variable makes a strong reference to the instance.
- “Strong” reference does not allow it to be deallocated for as long as that strong reference remains.

IMPORTANT

- Think about “strong and weak” pointers in your objects, about object ownership, and about possible retain cycles.
- So, are retain cycles still possible in ARC? Yes.
- Garbage collection is removed.

SOURCES

- https://developer.apple.com/library/archive/documentation/Cocoa/Conceptual/MemoryMgmt/Articles/MemoryMgmt.html
- https://developer.apple.com/library/archive/releasenotes/ObjectiveC/RN-TransitioningToARC/Introduction/Introduction.html#//apple_ref/doc/uid/TP40011226

SWIFT

REFERENCES AND RELATED CONTENT

WHAT WE NEED TO UNDERSTAND

- strong, weak, unowned
- retain cycles
- closures
- lazy initialization

STRONG REFERENCE

- 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**.
- Strong references are used almost everywhere in Swift (the declaration of a **property** by default)
- 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.

LINEAR HIERARCHY EXAMPLE

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

class Tentacle {
   let sucker = Sucker() //strong reference to child
}

class Sucker {
}

==> OK
```

WEAK REFERENCE

- 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.
- 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 vars.

… IN CLOSURE EXAMPLE

```
let closure = { [weak self] in
    self?.doSomething()
}
```

… IN PROTOCOLS EXAMPLE

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

REFERENCES

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

class Tentacle {
    weak var delegate: LossOfLimbDelegate? // Weak reference to delegate

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

CLASS INHERITANCE IN PROTOCOLS

- 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."*
  - If you have a reference hierarchy exactly like the one I showed above (in example), 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 REFERENCES

- 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 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**.
- 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."*

REFERENCES

- 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."*
  - *"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."*

UNOWNED REFERENCE EXAMPLE

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

class Kraken {
    let petName = "Krakey-poo"
    lazy var businessCardName: String = {
        return "Mr. Kraken AKA " + self.petName
    }()
}
==> lazy variables that AREN'T closures - also OK
```

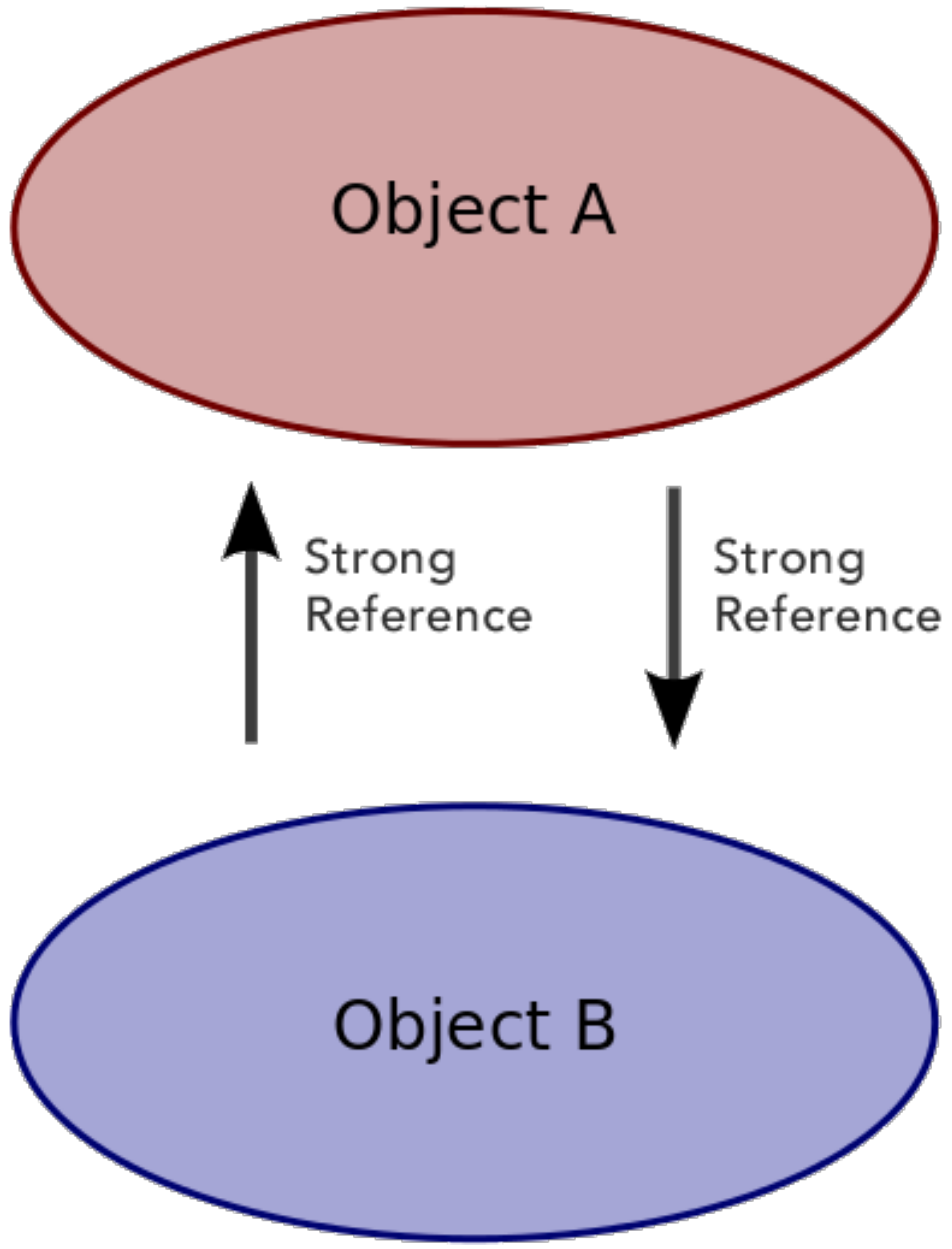
MULTIPLE CAPTURE VALUES IN CLOSURE

```
let closure = { [weak self, unowned krakenInstance] in
    self?.doSomething() //weak variables are Optionals!
    krakenInstance.eatMoreHumans() //unowned variables are not.
}

==> OK
```

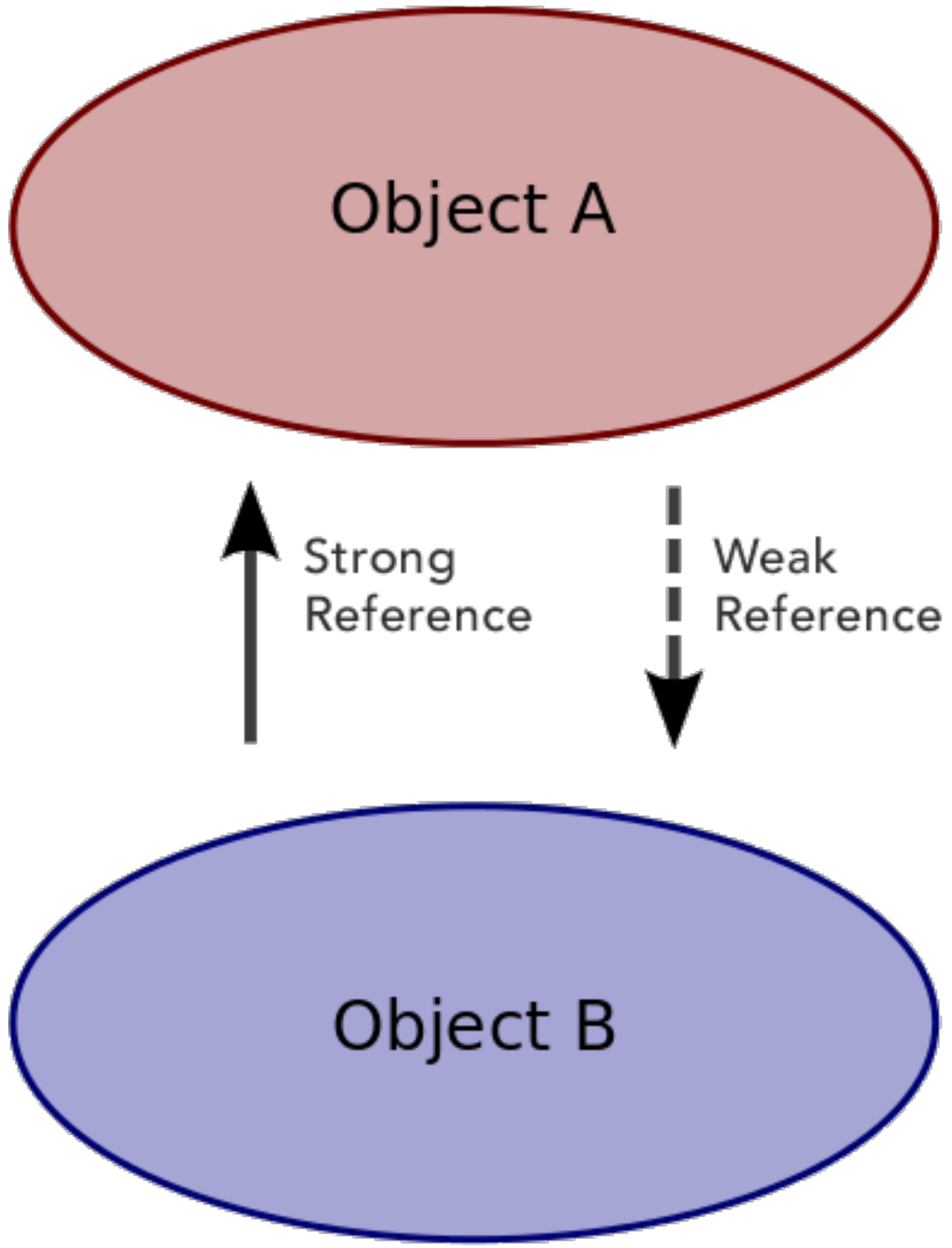
RETAIN CYCLES

- 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.

RETAIN CYCLE



with retain cycle



without retain cycle

EXPLANATION IN CODE

```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") }
}
```

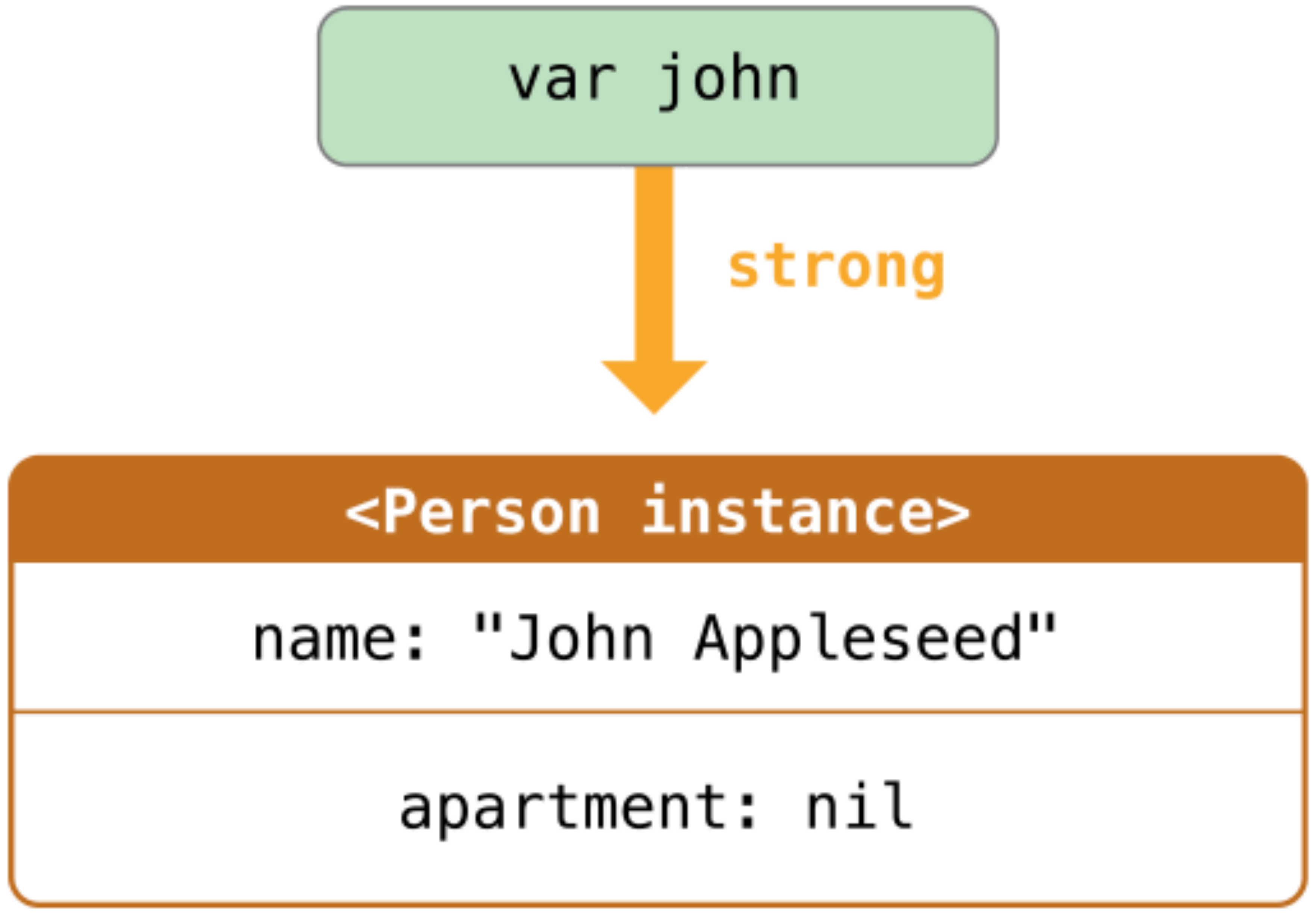
RETAIN CYCLE

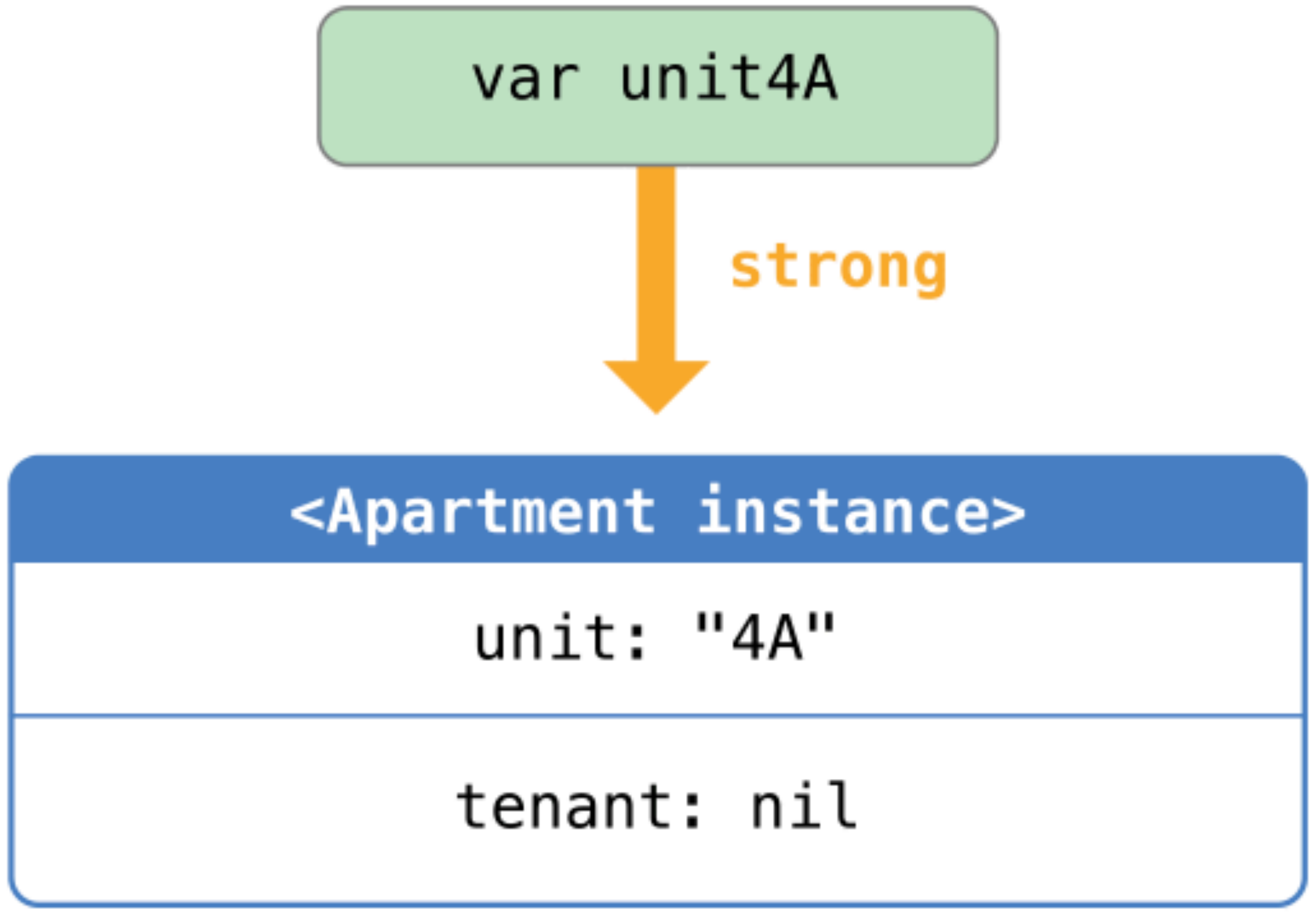
- Every `Person` instance has a name property of type String and an optional apartmentproperty that is initially nil. The apartment property is optional, because a person may not always have an apartment.
- Similarly, every `Apartment` instance has a unit property of type String and has an optional tenant property that is initially nil. The tenant property is optional because an apartment may not always have a tenant.
- Both of these classes also define a deinitializer, which prints the fact that an instance of that class is being deinitialized. This enables you to see whether instances of `Person` and `Apartment` are being deallocated as expected.

- This next code snippet defines two variables of optional type set to a specific `Apartment` and `Person` instance below.

```
var john: Person?
john = Person(name: "John Appleseed")
```

```
var unit4A: Apartment?
unit4A = Apartment(unit: "4A")
```

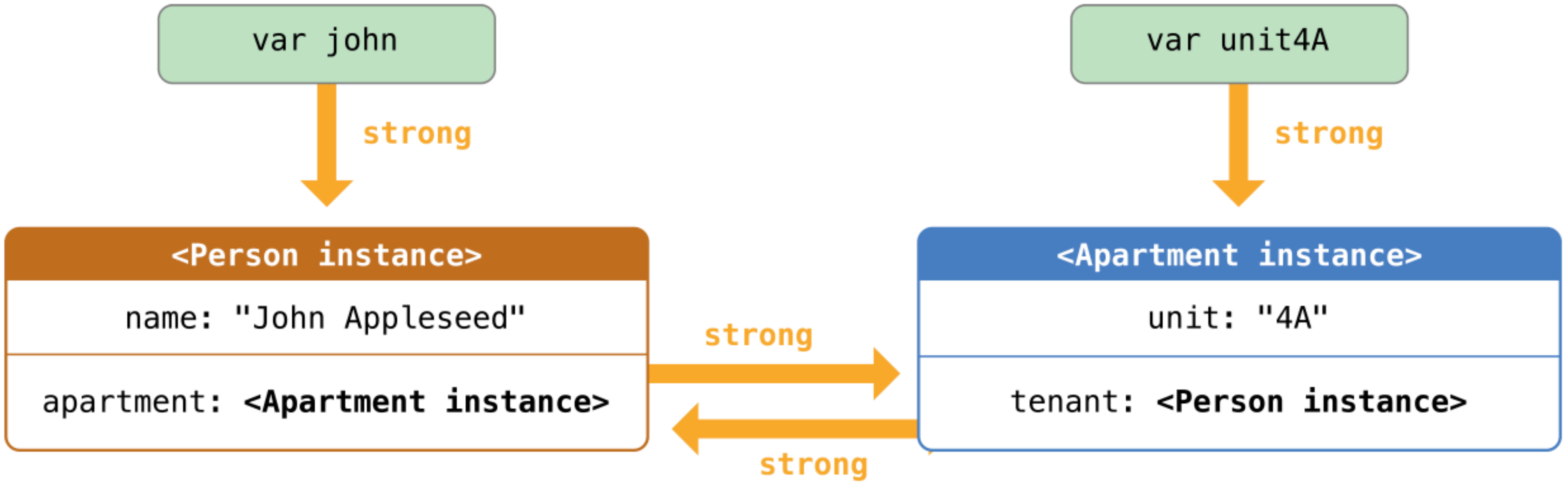




RETAIN CYCLE

- You can now link the two instances together so that the person has an apartment, and the apartment has a tenant.

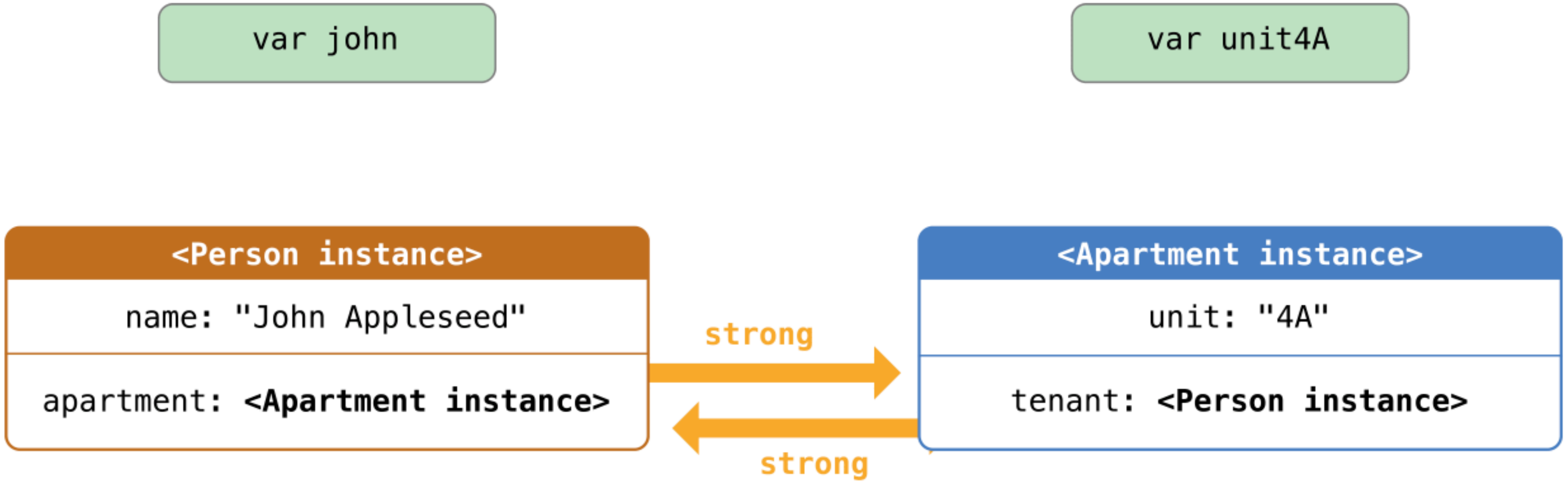
```
john!.apartment = unit4A
unit4A!.tenant = john
```



RETAIN CYCLE

```
john = nil
unit4A = nil
```

- How the strong references look after you set the john and unit4A variables to nil:



- The strong references between the `Person` instance and the `Apartment` instance remain and cannot be broken.

RESOLVING STRONG REFERENCE CYCLES BETWEEN CLASS INSTANCES

- Weak References
- Unowned References

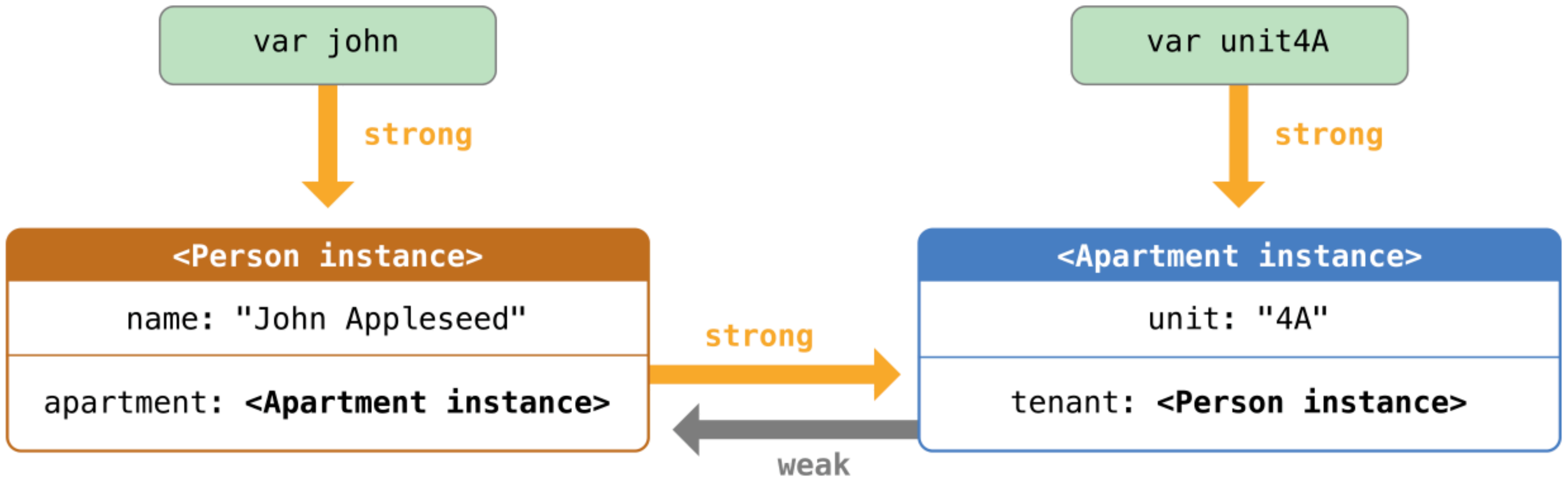
  - *both already mentioned*

WEAK REFERENCES - CODE EXAMPLE

```
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") }
}
```

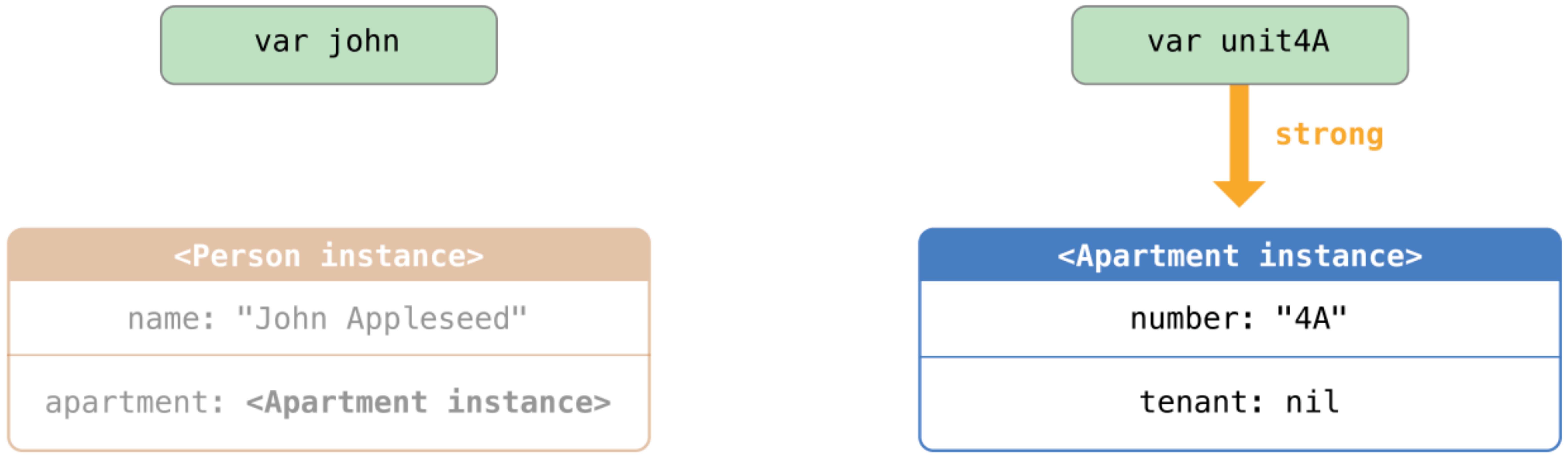
RETAIN CYCLE

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

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

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



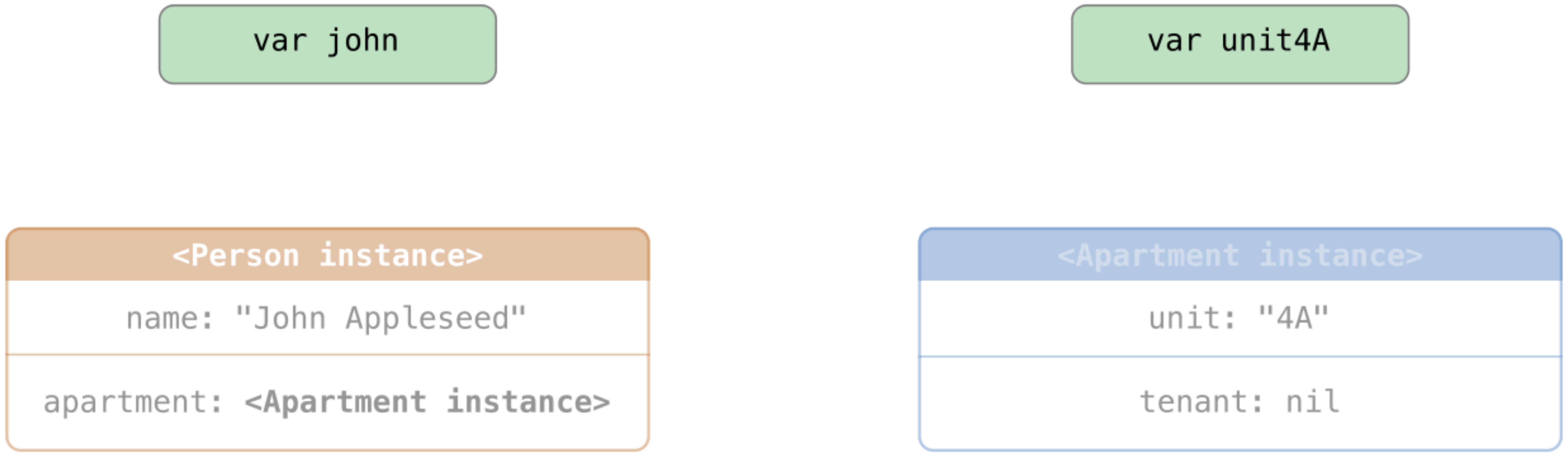
RETAIN CYCLE

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



RETAIN CYCLE

```
unit4A = nil
// Prints "Apartment 4A is being deinitialized"
```



NOTE

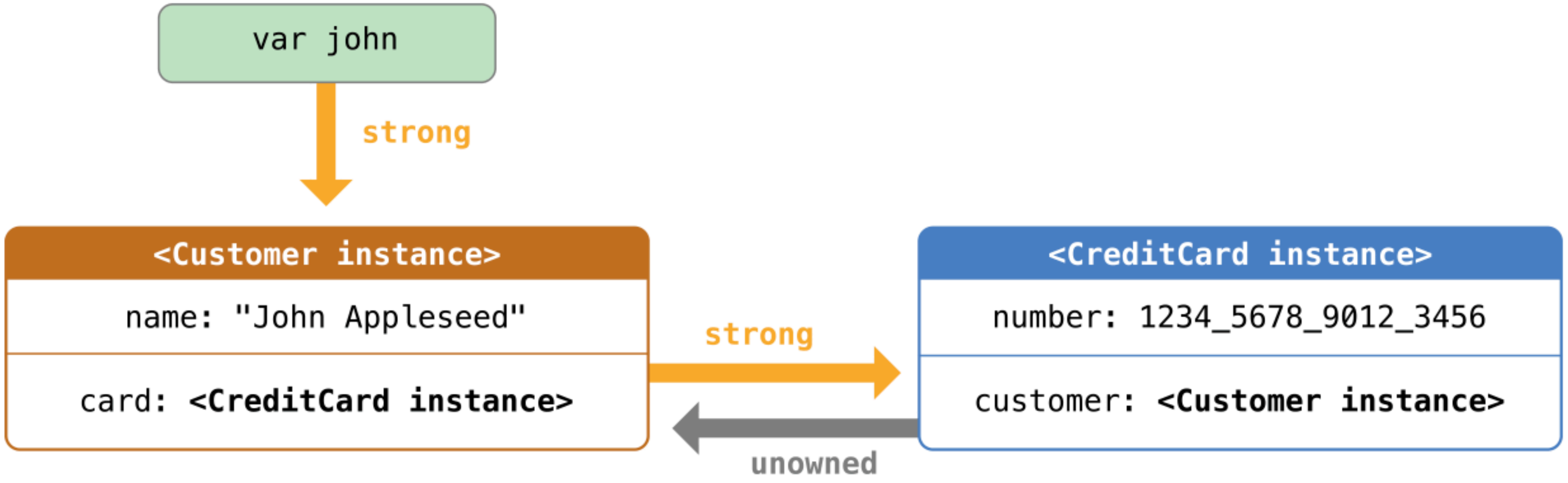
- In systems that use garbage collection, weak pointers are sometimes used to implement a simple caching mechanism because objects with no strong references are deallocated only when memory pressure triggers garbage collection. However, with ARC, values are deallocated as soon as their last strong reference is removed, making weak references unsuitable for such a purpose.

UNOWNED REFERENCES - CODE EXAMPLE

```
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") }
}
```

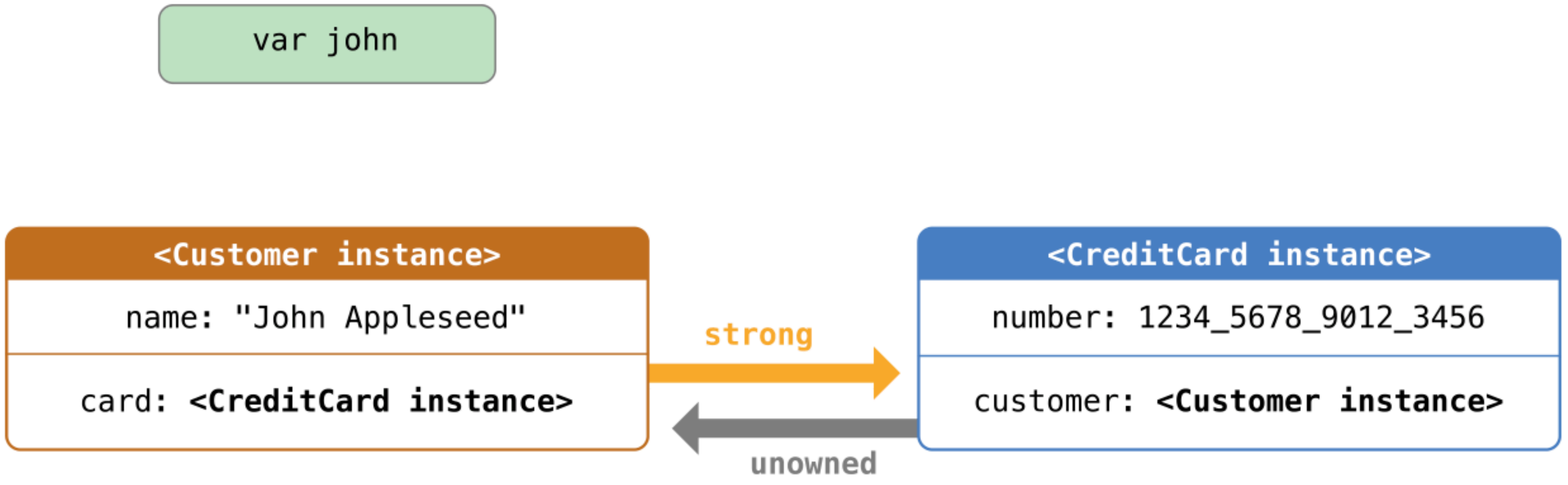
RETAIN CYCLE

```
var john: Customer?

john = Customer(name: "John Appleseed")
john!.card = CreditCard(number: 1234_5678_9012_3456, customer: john!)
```



RETAIN CYCLE

```swift
john = nil
// Prints "John Appleseed is being deinitialized"
// Prints "Card #1234567890123456 is being deinitialized"
```


var john
<Customer instance>
name: "John Appleseed"
card: <CreditCard instance>
strong
unowned
<CreditCard instance>
number: 1234_5678_9012_3456
customer: <Customer instance>

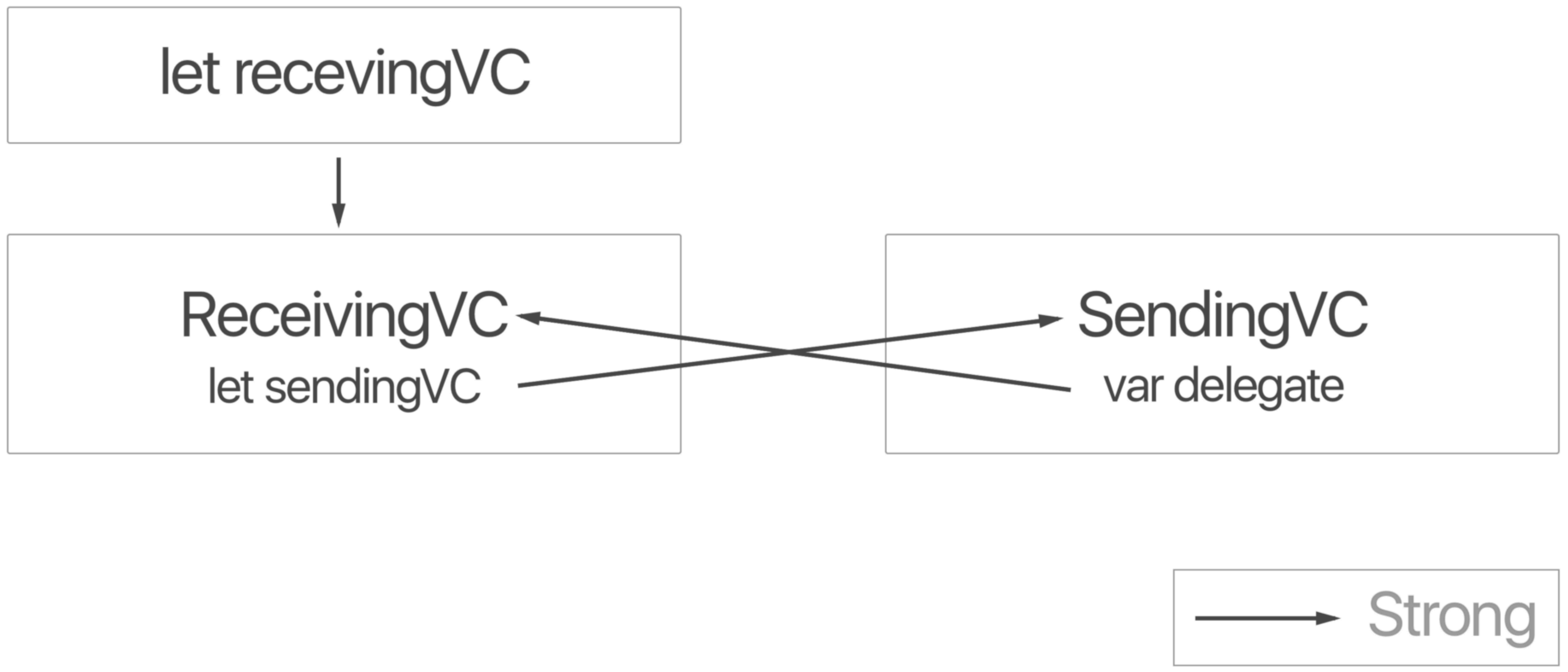
REMINDER

- 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.

RETAIN CYCLE AND PROTOCOLS

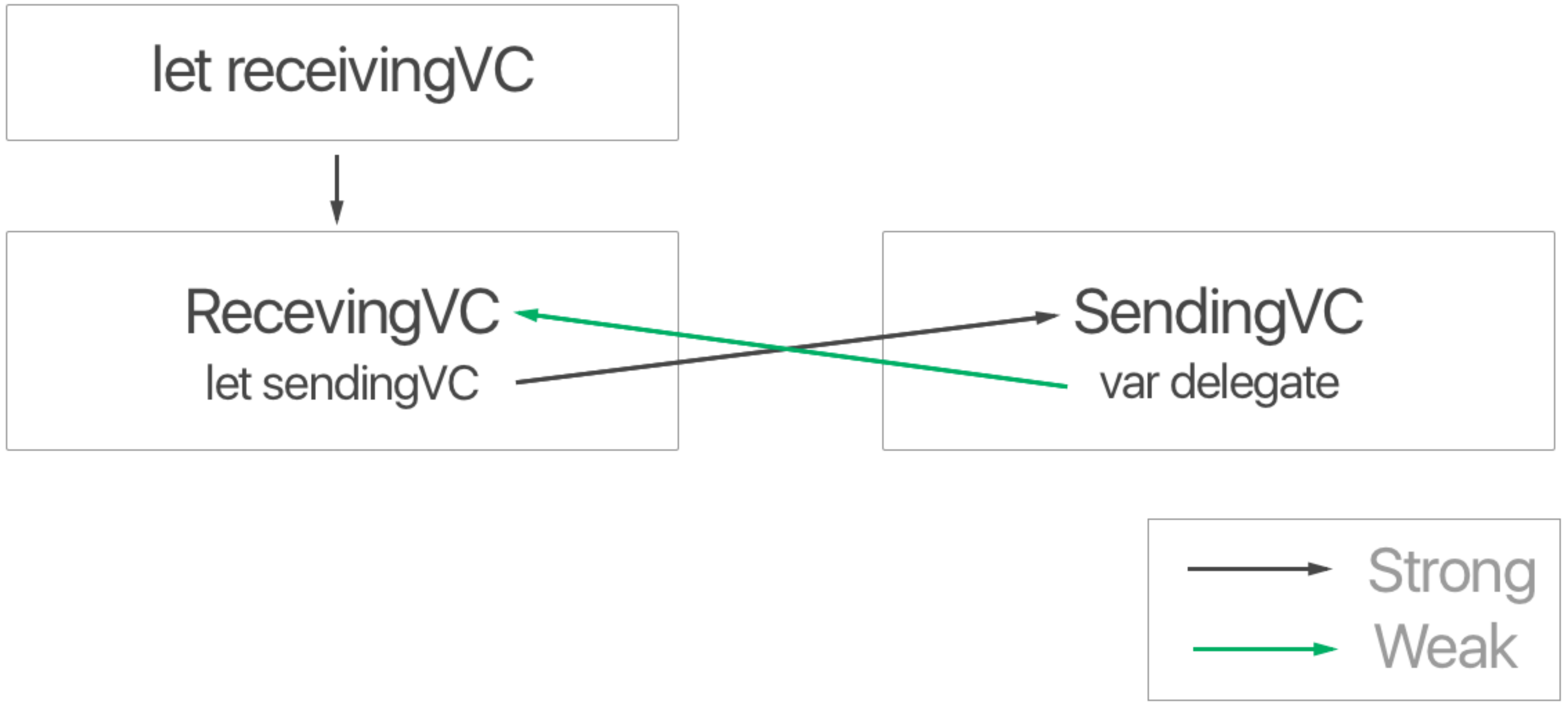
- *“When I encountered closures and delegate for the first time, I’ve noticed people put* `[weak self]` *in closures and* `weak var` *in front a delegate property. I’ve wondered why.”*

WITHOUT “WEAK VAR” & WITH RETAIN CYCLE



==> retain cycle

WITH “WEAK VAR” & WITHOUT RETAIN CYCLE



==> without cycle

EXAMPLE

```
// This Protocol can now only be used by Classes!
protocol ButtonDelegate: class {
    func onButtonTap(sender: UIButton)
}
class ViewWithTextAndButton: UIView {

    // Notice that we can now use the weak keyword!
    // this delegate now refers to a reference type only (UIViewController)
    // and that class will be weakly retained
    weak var delegate: ButtonDelegate?

    func onButtonTap(sender: UIButton) {
        delegate?.onButtonTap(sender: sender)
    }
}
```

GENERAL RULES

- As a general rule, delegates should be marked as weak because most delegates are referencing classes that they do not own. This is definitely true when a child is using a delegate to communicate with a parent. However, there are still some situations where a delegate can and should use a strong reference.

- Protocols can be used for both reference types (classes) and value types (structs, enums). So in the likely case that you need to make a delegate weak, you have to add the class keyword to the protocol so that it knows it is only to be used with reference types.

CLOSURES

- Closures are self-contained blocks of functionality that can be passed around and used in your code.
  - @nonescaping closures
  - @escaping closures
  - @autoclosures

@NONESCAPING CLOSURES

- When passing a closure as the function argument, the **closure gets execute with the function's body and returns the compiler back**. As the execution ends, the passed closure goes out of scope and have no more existence in memory.
- The default (when not specified) is @noescaping in Swift 3. They keyword does not actually exist anymore. There is only @escaping.

- Ways to use it
  - **Storage**: When you need to preserve the closure in storage that exist in the memory, past of the calling function get executed and return the compiler back. (Like waiting for the API response)
  - **Asynchronous Execution**: When you are executing the closure asynchronously on despatch queue, the queue will hold the closure in memory for you, to be used in future. In this case you have no idea when the closure will get executed.

EXAMPLE STORAGE

```
var complitionHandler: ((Int)->Void)?

func getSumOf(array:[Int], handler: @escaping ((Int)->Void)) {
    //step 2 (loop is just for example, in real it'll be something like API call)
    var sum: Int = 0
    for value in array {
        sum += value
    }
    //step 3
    self.complitionHandler = handler
}

func doSomething() {
    //setp 1
    self.getSumOf(array: [16,756,442,6,23]) { [weak self] sum in
        print(sum)
        //step 4, finishing the execution
    }
}
//Here we are storing the closure for future use.
```

EXAMPLE
ASYNCHRONOUS EXECUTION

```
func getSumOf(array:[Int], handler: @escaping ((Int)->Void)) {
    //step 2
    var sum: Int = 0
    for value in array {
        sum += value
    }
    //step 3
    Globals.delay(0.3, closure: {
        handler(sum)
    })
}

func doSomething() {
    //setp 1
    self.getSumOf(array: [16,756,442,6,23]) { [weak self] sum in
        print(sum)
        //step 4, finishing the execution
    }
}
//Here we are calling the closure with the delay of 0.3 seconds.
```

@ESCAPING CLOSURES

- When passing a closure as the function argument, the **closure is being preserve to be execute later and function's body gets executed, returns the compiler back**. As the execution ends, the scope of the passed closure exist and have existence in memory, till the closure gets executed.

EXAMPLE

```swift
func getSumOf(array:[Int], handler: ((Int)->Void)) {
    //step 2
    var sum: Int = 0
    for value in array {
        sum += value
    }
    //step 3
    handler(sum)
}

func doSomething() {
    //step 1
    self.getSumOf(array: [16,756,442,6,23]) { [weak self] sum in
        print(sum)
        //step 4, finishing the execution
    }
}
//As the step 4 get executed closure will have no existence in the memory.
```

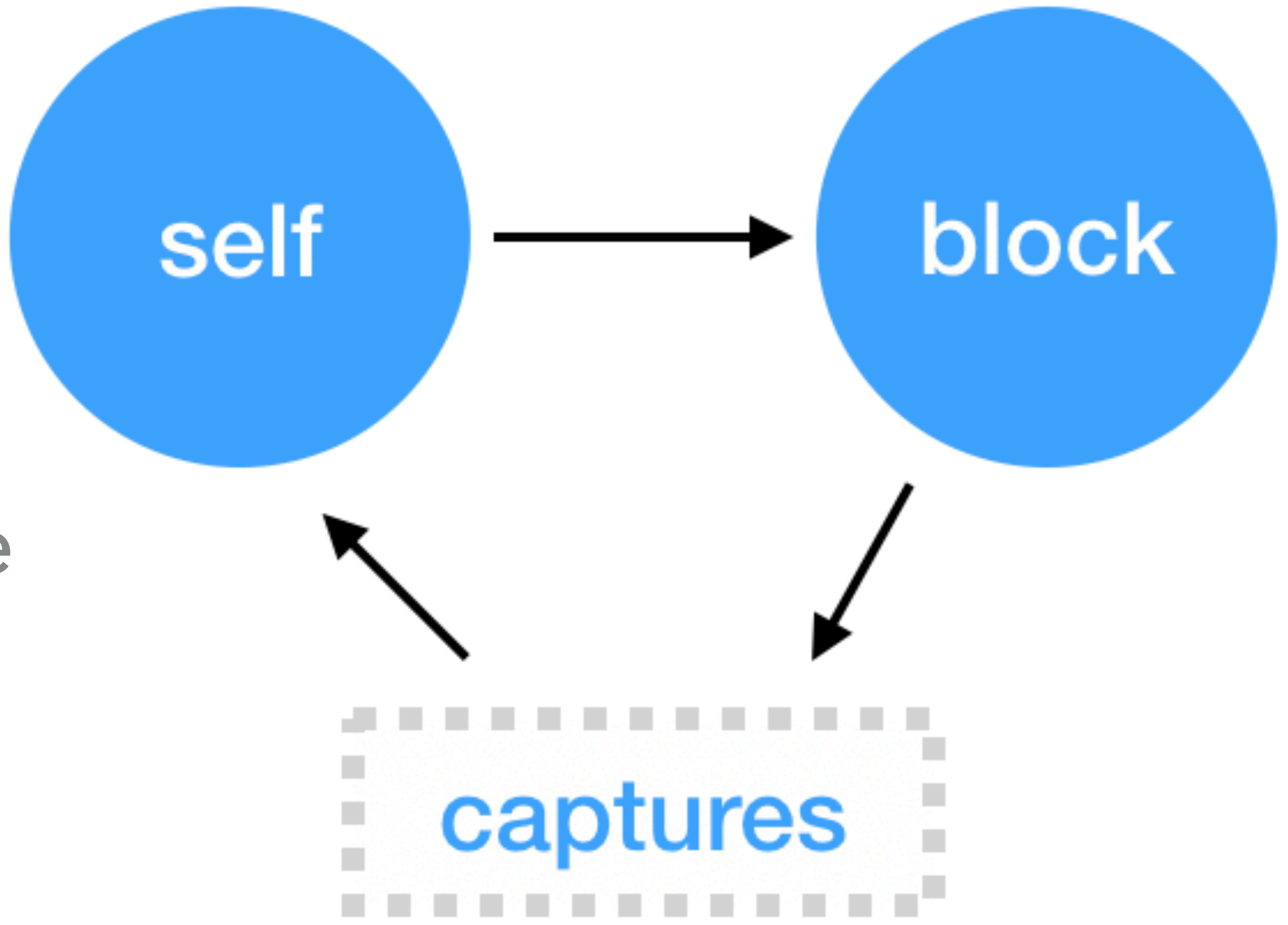
BENEFITS

- The most important benefits are performance and code optimisation by the compiler, because if the compiler knows that the closure is non-escaping, will take care about the memory allocation for the closure.
- In Swift 1.x and Swift 2.x, closure parameter was `@escaping` by default, means that closure can be escape during the function body execution. If don't want to escape closure parameters mark it as `@nonescaping`.
- In Swift 3.x, Apple made a **change**: closure parameters became `@nonescaping` **by default**, closure will also be execute with the function body, if wanna escape closure execution mark it as `@escaping`.

"THE STRONG-WEAK DANCE"

```
doSomething(then: {
    self.doSomethingElse()
}

==> with retain cycle
```

- The block will retain self and you've just created a bi-directional dependency graph, otherwise known as a **retain cycle**.



```
guard let self = self else { return }
```

- If you need to guarantee that the items in the block execute you can create a **new strong reference inside the block**. For instance the variable became nil while the block was executing, preventing some of the behavior from executing.

```
doSomething(then: { [weak self] in
    guard let self = self else { return }
    self.doSomethingElse()
}

==> without retain cycle
```

@AUTOCLOSURES

- In short, it allows you to **skip braces around the closure** in some situations.
- Apple docs: *"As with macros in C, auto-closures are a very powerful feature that must be used carefully because there is no indication on the caller side that argument evaluation is affected. Auto-closures are intentionally limited to only take an empty argument list, and you shouldn't use them in cases that feel like control flow. Use them when they provide useful semantics that people would expect (perhaps for a "futures" API) but don't use them just to optimize out the braces on closures."*

AUTOCLOSURE EXAMPLE

```
assert({ someExpensiveComputation() != 42 })
==> without autoclosure

assert(someExpensiveComputation() != 42)
==> with autoclosure


func &&(lhs: BooleanType, rhs: @autoclosure () -> BooleanType) -> Bool {
    return lhs.boolValue ? rhs().boolValue : false
}
==> usage of autoclosure
```

SWIFT 5 RELEASE NOTE

- In Swift 5 mode, @autoclosure parameters can no longer be forwarded to @autoclosure arguments in another function call. Instead, you must explicitly call the function value with parentheses: (); the call itself is wrapped inside an implicit closure, guaranteeing the same behavior as in Swift 4 mode.

```swift
func foo(_ fn: @autoclosure () -> Int) {}
func bar(_ fn: @autoclosure () -> Int) {
    foo(fn) // Incorrect, `fn` can't be forwarded and has to be called.
    foo(fn()) // OK
}
```

BACK TO LAZY INITIALIZATION

- Lazy initialization (also sometimes called lazy instantiation, or lazy loading) is a technique for **delaying the creation of an object or some other expensive process until it's needed**. When programming for iOS, this is helpful to make sure you utilize only the memory you need when you need it.
- This technique is so helpful, in fact, that Swift added direct support for it with the **lazy attribute**.

LAZY INITIALIZATION

- One example of when to use lazy initialization is **when the initial value for a property is not known until after the object is initialized**.

```
lazy var players: [String] = {
    var temporaryPlayers = [String]()
    temporaryPlayers.append("John Doe")
    return temporaryPlayers
}()
```

- This example **originally had unowned self** to prevent a strong reference cycle but this is not necessary. An immediately-applied closure (the above {}()) is **@noescape**, and does not retain the captured self.

- Another good time to use lazy initialization is **when the initial value for a property is computationally intensive**.

```
class MathHelper {
    lazy var pi: Double = {
        // Calculate pi to a crazy number of digits
        return resultOfCalculation
    }()
}
```

EXTRA RECOMMENDED

- retain cycle x closures x view controllers
  https://stackoverflow.com/questions/50261303/swift-closures-causing-strong-retain-cycle-with-self
- strong reference x delegates
  https://stackoverflow.com/questions/17348523/is-it-ever-ok-to-have-a-strong-reference-for-a-delegate
- weak reference x delegate –> supplemental answer
  https://stackoverflow.com/questions/24066304/how-can-i-make-a-weak-protocol-reference-in-pure-swift-without-objc
- Do you often forget [weak self]? Here's a solution
  https://medium.com/anysuggestion/preventing-memory-leaks-with-swift-compile-time-safety-49b845df4dc6
- Lazy initialisation and retain cycle –> dfri response
  https://stackoverflow.com/questions/38141298/lazy-initialisation-and-retain-cycle
- Memory Safety
  https://docs.swift.org/swift-book/LanguageGuide/MemorySafety.html

SOURCES

- https://krakendev.io/blog/weak-and-unowned-references-in-swift
- https://mikebuss.com/2014/06/22/lazy-initialization-swift/
- https://medium.com/@bestiosdevelope/what-do-mean-escaping-and-nonescaping-closures-in-swift-d404d721f39d
- https://stackoverflow.com/questions/39612938/what-is-difference-between-noescape-escaping-and-autoclosure
- https://developer.apple.com/swift/blog/?id=4
- https://developer.apple.com/documentation/xcode_release_notes/xcode_10_2_beta_4_release_notes/swift_5_release_notes_for_xcode_10_2_beta_4
- https://stackoverflow.com/questions/24066304/how-can-i-make-a-weak-protocol-reference-in-pure-swift-without-objc
- https://benscheirman.com/2018/09/capturing-self-with-swift-4-2/
- https://blog.bobthedeveloper.io/swift-retention-cycle-in-closures-and-delegate-836c469ef128
- https://www.natashatherobot.com/ios-weak-delegates-swift/#
- https://docs.swift.org/swift-book/LanguageGuide/AutomaticReferenceCounting.html#ID56

ANALYSIS TOOLS TO DEBUG MEMORY PROBLEMS

LINKS

- Instruments Tutorial with Swift: Getting Started
  https://www.raywenderlich.com/397-instruments-tutorial-with-swift-getting-started
- Profiling Memory Allocations In iOS With Instruments
  https://www.agnosticdev.com/blog-entry/ios/profiling-memory-allocations-ios-instruments
- Locating the source of a memory leak
  https://medium.com/@xcadaverx/locating-the-source-of-a-memory-leak-712667bf8cd5
- How to debug memory leaks when Leaks instrument does not show them?
  https://stackoverflow.com/questions/30992338/how-to-debug-memory-leaks-when-leaks-instrument-does-not-show-them

- Memory 2 - Finding and Fixing Memory Leaks (iOS, Xcode 9, Swift 4)
  https://www.youtube.com/watch?v=1LnipXiSrSM&t=599s
- iOS Memory Deep Dive
  https://developer.apple.com/videos/play/wwdc2018/416/

SOMETHING EXTRA

- Swift bugs
  https://bugs.swift.org/projects/SR/issues/SR-675?filter=allopenissues

FEBRUARY 2019

CREATED BY

Diana Brnovik