Memory Management

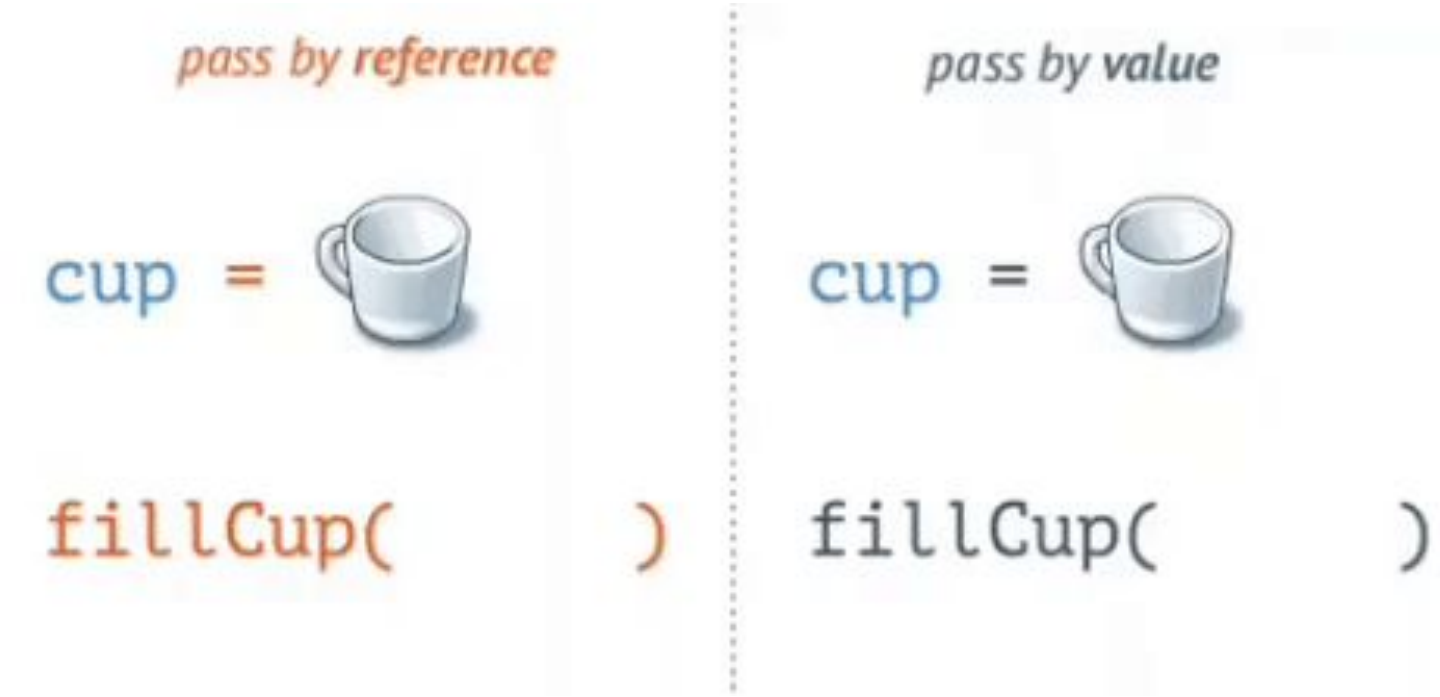
Outline

1. Reference Types vs Value Types
2. Stack and Heap
3. Automatic Reference Counting (ARC)

1. Reference Types vs Value Types

1. Reference Types
2. Value Types
3. Mutability
4. Usage
5. Memory Allocation

1. Reference Types vs Value Types



1.1 Reference Types

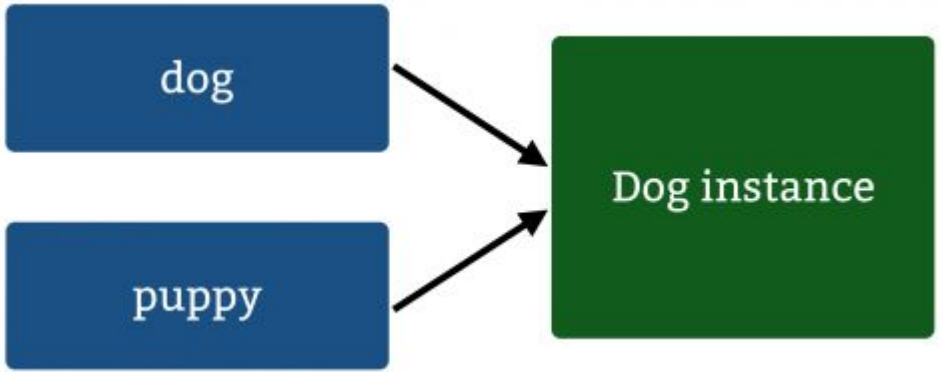
- Reference type: each instances share a single copy of the data. A type that once initialized, when assigned to a variable or constant, or when passed to a function, returns a reference to the same existing instance.

- Function, Closure, Class are reference types in Swift

1.1 Reference Types (cont)

- Example of reference type:

```
class Dog {
    var wasFed = false
}

let dog = Dog()
let puppy = dog
puppy.wasFed = true
print(dog.wasFed) // true
print(puppy.wasFed) // true
```



1.2 Value Types

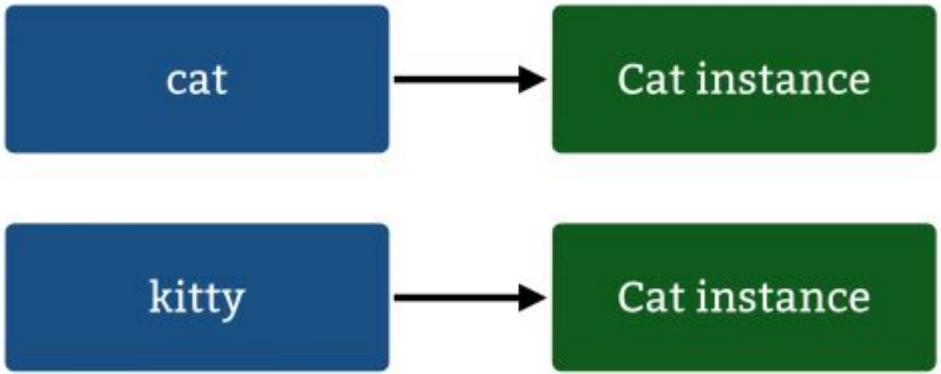
- Value type: each instance keeps a unique copy of its data. A type that creates a new instance (copy) when assigned to a variable or constant, or when passed to a function.

- Some value types in Swift: Int, Double, String, Array, Dictionary, Set, Struct, Enum, Tuple

1.2 Value Types (cont)

- Example of value type:

```
struct Cat {
  var wasFed = false
}

var cat = Cat()
var kitty = cat
kitty.wasFed = true

cat.wasFed          // false
kitty.wasFed        // true
```



1.3 Mutability

- `var` and `let` function differently for reference types and value types:

  - For reference types, `let` means the reference must remain constant. In other words, you can't change the instance the constant references, but you can mutate the instance itself.

  - For value types, `let` means the instance must remain constant. No properties of the instance will ever change, regardless of whether the property is declared with `let` or `var`.

1.4 Usage

- Use a value type when:

  - ➔ Comparing instance data with == makes sense. A double equal operator (aka ==) compare **values**.

  - ➔ You want copies to have independent state.

  - ➔ The data will be used in code across multiple threads. So that you don't have to worry about the data being changed from another thread.

1.4 Usage (cont)

- Use a reference type when:

  - ➔ Comparing instance identity with === makes sense. === checks if two objects are exactly identical, right down to the memory address that stores the data.

  - ➔ You want to create shared, mutable state.

1.5 Memory Allocation

- In Common Type System:
  - ➔ Value type get stored on Stack memory
  - ➔ Reference type get stored on Managed Heap memory

2. Stack vs Heap

1. Stack
2. Heap
3. Compare Stack vs Heap

2.1 Stack

- The stack is a region of memory which contains storage for local variables, as well as internal temporary values and housekeeping.

- On a modern system, there is one stack per thread of execution.

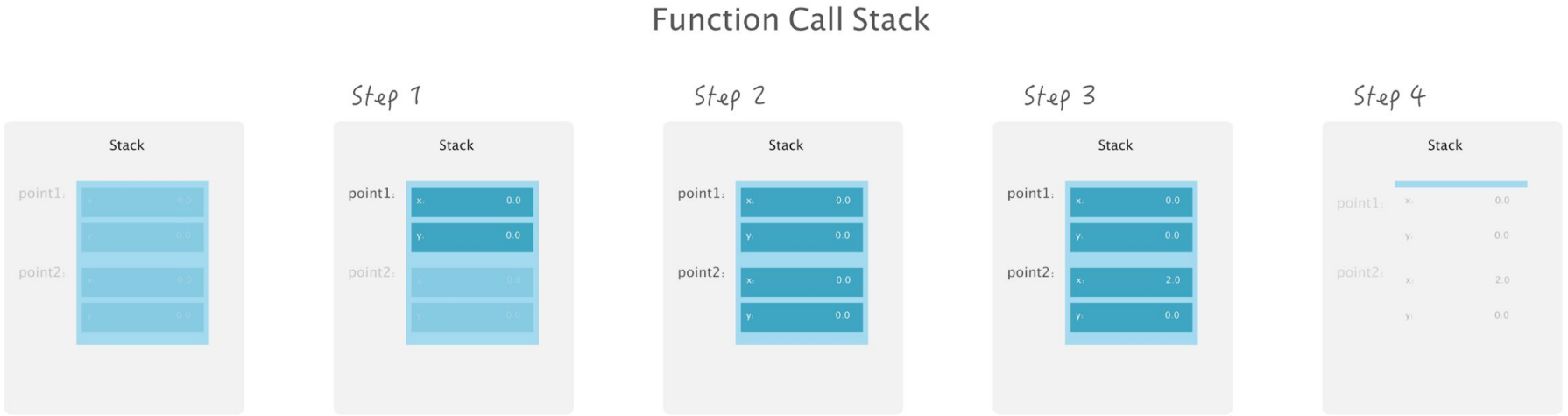
- When a function is called, a stack frame is pushed onto the stack, and function-local data is stored there. When the function returns, its stack frame is destroyed. All of this happens automatically, without the programmer taking any explicit action other than calling a function.

2.1 Stack (cont)

- All of allocation happens in what's called a ***function call stack***. When a function exits, all the data will pop off the stack. Data will only exist while the function is still running.

```
struct Point {
  var x: Double
  var y: Double
}

// Step 1
let point1 = Point(x: 0, y: 0)

// Step 2
var point2 = point1

// Step 3
point2.x = 2

// Step 4
// completed all tasks above
```

2.1 Stack (cont)



2.1 Stack (cont)

- Pros:
  - ➔ Very fast access
  - ➔ More efficient when allocating and deallocating data
  - ➔ Data stored in the stack is only there temporarily until the function exits and causes all memory on the stack to be automatically deallocated
  - ➔ Space is managed efficiently by CPU, memory will not become fragmented

2.1 Stack (cont)

- Cons:
  - ➔ Stack memory is very limited.
  - ➔ Creating too many objects on the stack can increase the risk of stack overflow.
  - ➔ Random access is not possible.
  - ➔ Variable storage will be overwritten, which sometimes leads to undefined behavior of the function or program.
  - ➔ The stack will fall outside of the memory area, which might lead to an abnormal termination.

2.2 Heap

- The heap is a region of your computer's memory that is not managed automatically for you, and is not as tightly managed by the CPU. It is a more free-floating region of memory (and is larger).

- The heap is dynamic in memory and allocation happens during runtime. Values can be referenced at anytime through a memory address.

- When you create a new object, some memory gets allocated to the heap. Just like stacks, the memory goes away once it’s not being used. However, in heaps, the memory isn’t tied to functions or programs; instead, the data in the heap is global.

2.2 Heap (cont)

- Pros:
  - Heap helps you to find the greatest and minimum number
  - Garbage collection runs on the heap memory to free the memory used by the object.
  - Heap method also used in the Priority Queue.
  - It allows you to access variables globally.
  - Heap doesn't have any limit on memory size.

2.2 Heap (cont)

- Cons:
  - ➔ It can provide the maximum memory an OS can provide
  - ➔ It takes more time to compute.
  - ➔ Memory management is more complicated in heap memory as it is used globally.
  - ➔ It takes too much time in execution compared to the stack.

2.3 Compare Stack vs Heap

Parameter	Stack	Heap
Type of data structures	A stack is a linear data structure.	Heap is a hierarchical data structure.
Access speed	High-speed access	Slower compared to stack
Space management	Space managed efficiently by OS so memory will never become fragmented.	Heap Space not used as efficiently. Memory can become fragmented as blocks of memory first allocated and then freed.
Access	Local variables only	It allows you to access variables globally.
Limit of space size	Limit on stack size dependent on OS.	Does not have a specific limit on memory size.
Resize	Variables cannot be resized	Variables can be resized.

2.3 Compare Stack vs Heap (cont)

Parameter	Stack	Heap
Memory Allocation	Memory is allocated in a contiguous block.	Memory is allocated in any random order.
Allocation and Deallocation	Automatically done by compiler instructions.	It is manually done by the programmer.
Deallocation	Does not require to de-allocate variables.	Explicit de-allocation is needed.
Cost	Less	More
Implementation	A stack can be implemented in 3 ways simple array based, using dynamic memory, and Linked list based.	Heap can be implemented using array and trees.
Main Issue	Shortage of memory	Memory fragmentation

2.3 Compare Stack vs Heap (cont)

Parameter	Stack	Heap
Locality of reference	Automatic compile time instructions.	Adequate
Flexibility	Fixed size	Resizing is possible
Access time	Faster	Slower

3. Automatic Reference Counting (ARC)

1. Overview
2. How ARC works
3. ARC in Action
4. Reference Cycles
5. Weak, Unowned References
6. Capture List

3.1 Overview

- 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.

3.2 How ARC works

- Every time we create an instance of a class, ARC allocates memory that holds information about the Type and Value it holds to.

- 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.

3.2 How ARC works

- 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.

3.3 ARC in Action

- Here’s an example of how Automatic Reference Counting works:

```
class Person {
  let name: String

  init(name: String) {
    self.name = name
    print("\(name) is being initialized")
  }

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

3.3 ARC in Action (cont)

- Because these variables are of an optional type (`Person?`, not `Person`), they are automatically initialized with a value of `nil`, and do not currently reference a `Person` instance.

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

3.3 ARC in Action (cont)

- When you create a new **`Person`** instance and assign it 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.

```
person1 = Person(name: "John")
// Prints "John is being initialized"
```

3.3 ARC in Action (cont)

- There are now three strong references to this single `Person` instance:

```
person2 = person1
person3 = person1
```

3.3 ARC in Action (cont)

- If you break two of these strong references (including the original reference) by assigning `nil` to two of the variables, a single strong reference remains, and the `Person` instance is not deallocated:

```
person1 = nil
person2 = nil
```

3.3 ARC in Action (cont)

- 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
// Prints "Jogn is being deinitialized"
```

3.4 Reference Cycles

- A strong reference prevents the class instance it points from being deallocated.

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

- Some cases can cause that: The Delegation Pattern, Dependencies, Leaky Closures

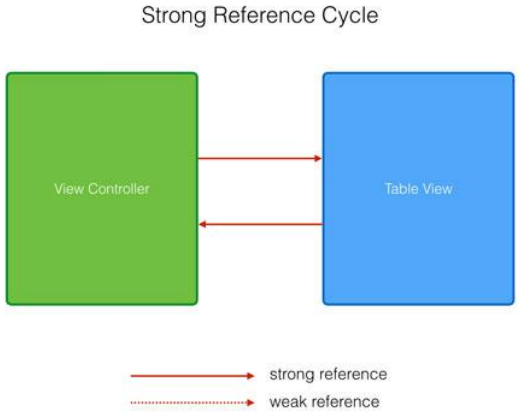
3.4 Reference Cycles (cont)

- The Delegation Pattern is simple but powerful. Apple's frameworks make heavy use of delegation:

```
import UIKit

class ViewController: UIViewController {
  @IBOutlet private var tableView: UITableView!

  override func viewDidLoad() {
    tableView.delegate = self
  }
}

extension ViewController: UITableViewDelegate {
  func tableView(_ tableView: UITableView, heightForRowAt indexPath: IndexPath) -> CGFloat {
        return 150
    }
}
```

3.4 Reference Cycles (cont)

- The view controller and the tableview keep each other alive. Even if the view controller and its table view are no longer needed by the application, they are not deallocated. They continue to take up memory and the application is leaking memory.



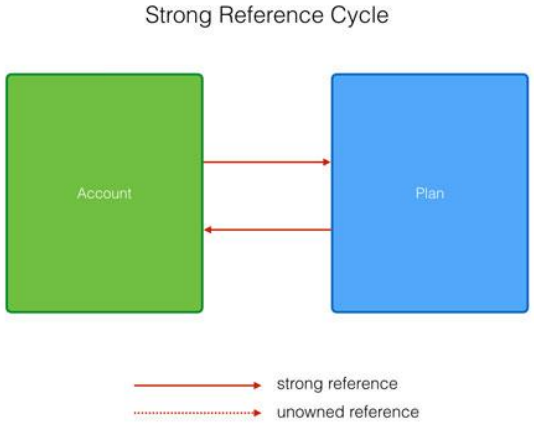
3.4 Reference Cycles (cont)

- Strong reference cycles can also be caused by objects that depend on one another:

```
class Account {
  var plan: Plan

  init(plan: Plan) {
    self.plan = plan
  }
}

class Plan {
  var account: Account

  init(account: Account) {
    self.account = account
  }
}
```

3.4 Reference Cycles (cont)

- Because references are strong by default, an **`Account`** instance holds a strong reference to a **`Plan`** instance. The same is true for a **`Plan`** instance. A **`Plan`** instance holds a strong reference to an **`Account`** instance. This means we end up with another strong reference cycle.



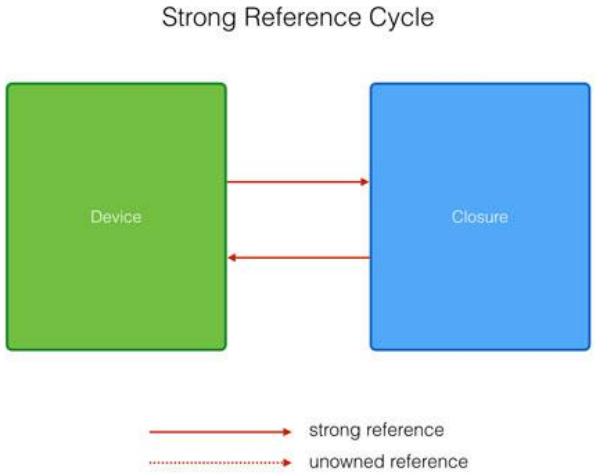
3.4 Reference Cycles (cont)

- Another common source of reference cycles or retain cycles is through the use of closures. Closures are a powerful tool in your toolbox, but you need to be careful in some scenarios. Like classes, closures are reference types:

```
class Device {

    let model: String
    let manufacturer: String

    lazy var summary: () -> String = {
        return "\(self.model) (\(self.manufacturer))"
    }

    init(model: String, manufacturer: String) {
        self.model = model
        self.manufacturer = manufacturer
    }
}
```

3.4 Reference Cycles (cont)

- The `Device` instance holds a reference to the closure stored in the `summary` property. But the closure stored in the `summary` property also holds a reference to the `Device` instance, `self`, causing a strong reference cycle.



3.5 Weak and Unowned References

- Automatic Reference Counting usually works without you having to do anything. But there are scenarios in which ARC needs a little bit of help.

- To avoid strong reference cycles, you sometimes need to give the compiler a hand. Weak and unowned references in particular often cause confusion.

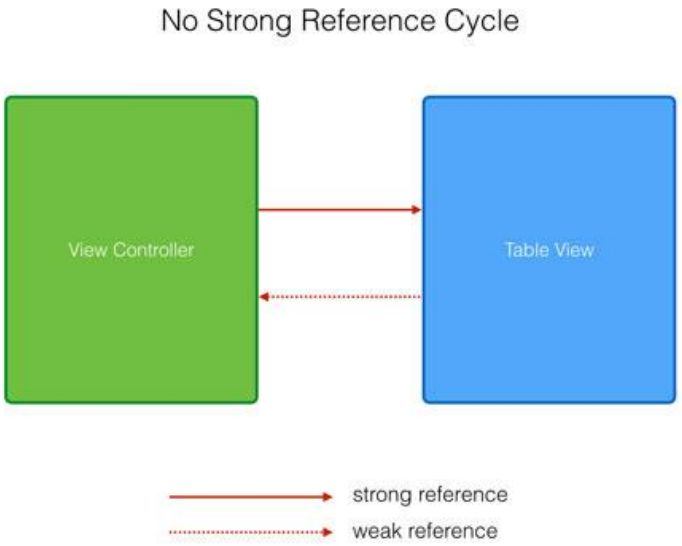
3.5 Weak and Unowned References (cont)

- A ***Weak Reference*** keeps a **weak** reference to the instance it references. This means that the reference to the instance is not taken into account by ARC. That instance is deallocated if no other objects have a strong reference to the instance.

```
import UIKit

class ViewController: UIViewController {
  @IBOutlet private weak var tableView: UITableView!

  override func viewDidLoad() {
    tableView.delegate = self
  }
}

extension SearchMovieController: UITableViewDelegate {
    func tableView(_ tableView: UITableView, heightForRowAt indexPath: IndexPath) -> CGFloat {
        return 150
    }
}
```

3.5 Weak and Unowned References (cont)

- The strong reference cycle is broken by declaring the `tableView` property as **`weak`**.



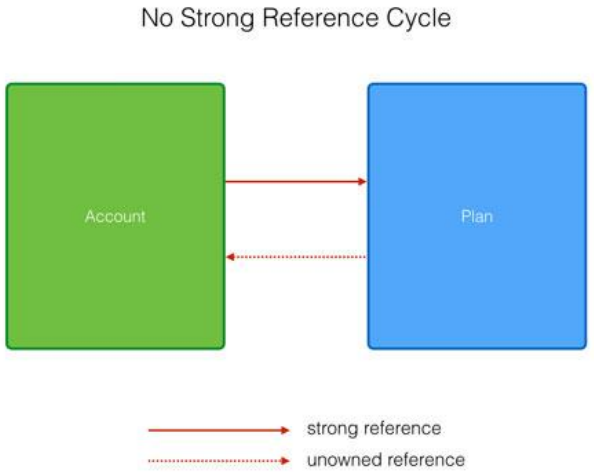
3.5 Weak and Unowned References (cont)

- ***Unowned references*** are similar to ***weak references*** in that they don't keep a strong reference to the instance they are referencing. They serve the same purpose as ***weak references***, that is, they avoid strong reference cycles.

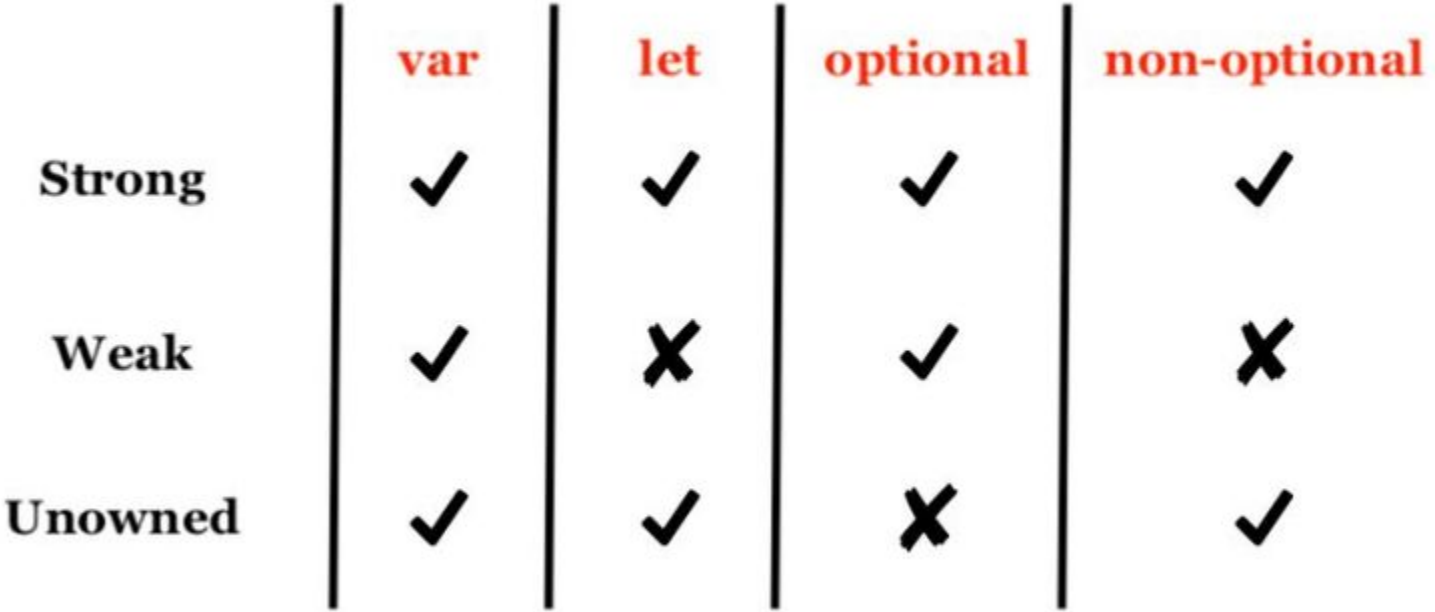
```
class Account {
  var plan: Plan

  init(plan: Plan) {
    self.plan = plan
  }
}

class Plan {
  unowned var account: Account

  init(account: Account) {
    self.account = account
  }
}
```

3.5 Weak and Unowned References (cont)

- It is important to understand why we marked the `account` property as **`unowned`** instead of **`weak`**. An account should always have a plan associated with it and a plan shouldn't exist without an account.



3.5 Weak and Unowned References (cont)



	var	let	optional	non-optional
Strong	✓	✓	✓	✓
Weak	✓	✗	✓	✗
Unowned	✓	✓	✗	✓

3.6 Capture List

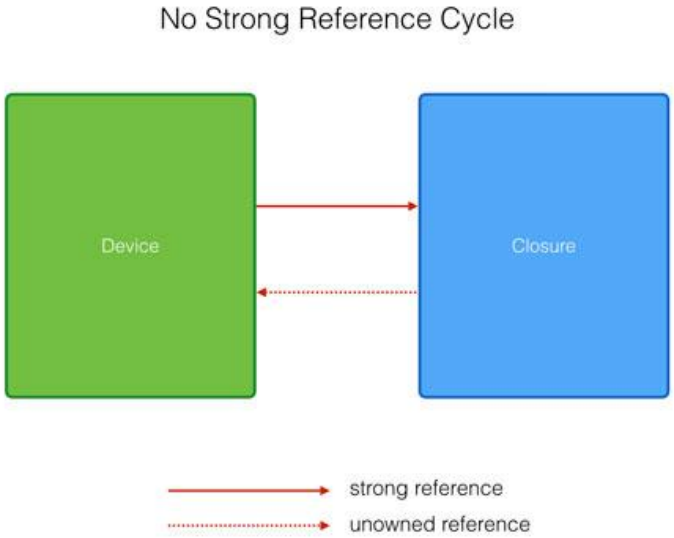
- The references a closure holds to reference types are strong by default. We can change this behavior by defining a capture list.

- The capture list of a closure determines how the reference types captured by the closure are managed in memory.

```
class Device {

    let model: String
    let manufacturer: String

    lazy var summary: () → String = { [unowned self] in
        return "\(self.model) (\(self.manufacturer))"
    }
}
```

3.6 Capture List

- The strong reference cycle is broken by declaring the **`self`** as **`weak`** in the capture list.



Question & Answer?