Gestión de la memoria en Swift

Una explicación sobre la gestión de la memoria y cómo aprovechar las estructuras para optimizar tus abstracciones

Introducción

En Swift, como en muchos otros lenguajes de programación, la gestión de la memoria es una parte esencial para garantizar un rendimiento óptimo y prevenir problemas como fugas de memoria o accesos inválidos a la misma. En Swift, la gestión de la memoria se lleva a cabo principalmente en dos áreas: la pila (**Stack**) y el montículo (**Heap**). Paciencia aquí, acabarás familiarizado con estos dos términos pronto...

Listemos algunas diferencias entre **Heap** y **Stack**. No os molestéis en intentar memorizar nada, esta tabla solo busca ser una referencia, un sitio al que podamos volver si tenemos dudas sobre una determinada característica de nuestras dos nuevas mejores amigas.

Stack y Heap, cara a cara

Característica	**Stack**	**Heap**
Asignación de memoria:	Estática, se realiza durante la compilación	Dinámica, tiene lugar en tiempo de ejecución
Acceso:	Rápido, debido a la asignación y liberación automática de memoria	Un poco más lento, implica más gestión de la memoria mediante contadores de referencias o ARC
Usada para almacenar:	Tipos de datos por valor: estructuras, enumerados... *	Tipos de datos por referencia: clases, actores... *
Seguridad:	Cada hilo tiene su propia **Stack** de memoria, por lo que no pueden producirse accesos simultáneos a datos "de estado compartido" que puedan producir las famosas "condiciones de carrera". **	Todos los hilos acceden al mismo **Heap** por lo que pueden producirse "condiciones de carrera", así que los datos deben de estar protegidos ante esta casuística. **
Rendimiento:	Rendimiento muy alto. ***	Menor rendimiento debido a diversos factores. ***

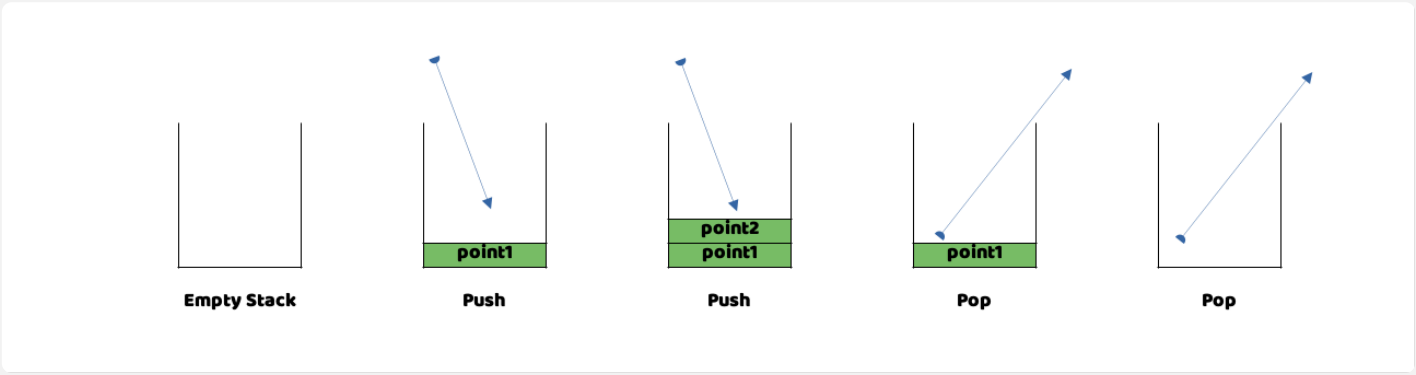
(*) ¿Quiere decir esto que siempre que cree un tipo de dato mediante **Struct** sus objetos van a ser

almacenados en el **Stack**?, no, desgraciadamente no es tan sencillo.

(**) Cada hilo de ejecución tiene su propio **Stack** de memoria mientras que el **Heap** es compartido por todos los hilos. Como varios hilos de ejecución pueden asignar memoria en el **Heap** al mismo tiempo éste debe protegerse mediante bloqueos, semáforos u otros mecanismos de sincronización. Esto supone un coste bastante grande de rendimiento.

(***) La gestión de la memoria que hace Swift mediante el **Stack**, de forma estática, es muy eficiente. La pila funciona mediante un sistema de LIFO, imagen 1. Swift puede agregar la información que necesite a la pila, "Allocation", de forma muy rápida y limpiarla de ésta igual de rápido, "Deallocation".

En la Imagen 2 podemos observar otra representación gráfica de cómo Swift agregaría datos al Stack y cómo los liberaría, cuando ya no fuesen necesarios, mediante dicho sistema LIFO (Last in, First Out).



Mientras tanto, la asignación dinámica en el **Heap** (montículo), implica que debamos encontrar un espacio en memoria con el tamaño apropiado y, cuando ya no la necesitemos, desasignarla adecuadamente, incurriendo en una gestión mucho más compleja, y por ende menos eficiente, que la usada para almacenar en la **Stack** (pila).

Si queremos que nuestras abstracciones sean rápidas, y óptimas, necesitamos tener en cuenta todas y cada una de las cuestiones que acabamos de leer, más algunas otras como el despachado dinámico y estático de métodos que veremos un poco más adelante.

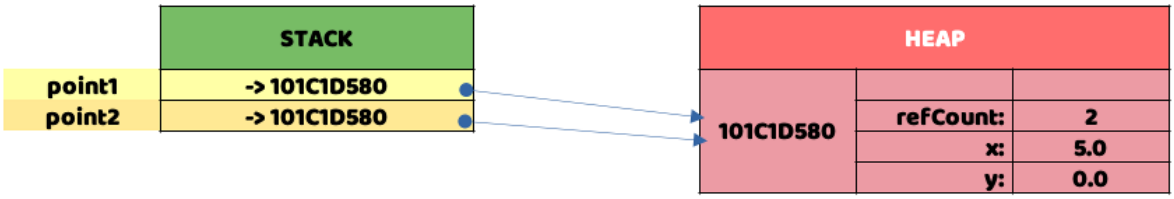
¿Hay código o no hay código?...

Vamos a verlo con código, usando los mismos ejemplos que puso Apple en la WWDC 2016, conferencia "Understanding Swift Performance".

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

let point1 = Point(x: 0, y: 1)
var point2 = point1
point2.x = 5
```

Este sencillo ejemplo no haría uso del Heap, ambas estructuras serían almacenadas en la Stack de memoria correspondiente. Hemos agregado una representación de cómo quedaría dicho Stack en memoria, solo se trata de una representación gráfica, sin más pretensión que la de aportar una ayuda visual a la explicación. ¿Cómo sería su comportamiento en memoria si usásemos una clase?, veámoslo con un ejemplo:

```swift
class Point {
    var x, y: Double
    init(x: Double, y: Double) {
        self.x = x
        self.y = y
    }
}

let point1 = Point(x: 0, y: 0)
let point2 = point1
point2.x = 5
```



Cómo podemos apreciar en la representación de memoria, para este ejemplo, Swift tiene que hacer uso del Heap para almacenar los datos del objeto creado, con todos los costes de rendimiento, seguridad, etc, que hemos visto en el bloque anterior. Además podemos observar que Swift sigue teniendo que hacer uso de la Stack correspondiente para almacenar las referencias a **101C1D580**, tanto para el objeto point1 como para el objeto point2.

101C1D580 es un referencia de memoria de ejemplo, para simular una real donde se almacenarían los datos. Muchos, al llegar hasta aquí, os preguntaréis por qué hay cuatro "espacios" de memoria cuando para "almacenar" la estructura solo hacían falta dos. Veamos, uno de los espacios, al que llamaremos **refCount**, es usado por ARC para almacenar el número de "referencias activas" que apuntan a dicha posición de memoria. Trataremos más adelante sobre el último espacio. Cuando hablemos del despachado dinámico y estático de métodos de clase este "espacio" cobrará su debida importancia, por ahora, obviémoslo, Swift lo necesita y lo reserva en consecuencia.

Por cierto, cuando este **refCount** llega a 0 implica que ya no está siendo "apuntado" por ningún objeto, con lo que Swift procede a reciclarlo de la memoria. ¿Alguna vez os han preguntado en una entrevista por los "retain cycles"?, ¿"circular references"?, no lo veremos hoy, pero como suele decirse: por aquí van los tiros...

Structs ineficientes y alternativas a éstos

Revisemos la siguiente implementación para aprender, con un ejemplo de Apple, de algunas trampas en las que podemos caer al implementar nuestras funcionalidades, abstracciones, etc:

```swift
enum Color { case blue, green, gray }
enum Orientation { case left, right }
enum Tail { case none, tail, buble }

var cache [String: UIImage]()

func makeBalloon(_ color: Color, orientation: Orientation, tail: Tail) -> UIImage {
    let key = "\(color):\(orientation):\(tail)"
    if let image = cache[key] { return image }
    // Aquí iría el resto de implementación para devolver el UIImage correspondiente,
    // no nos vale para el ejemplo así que la obvio.
}
```

La función se ocupa de crear los típicos "globitos" que se usan en viñetas, comics, etc, para mostrar una conversación con un lazo-flecha apuntando hacia la persona que está hablando. Como tanto el color, como la orientación y el tipo de lazo son finitos y pueden parametrizarse, contabilizarse, etc, nuestro desarrollador ha usado enums para establecer estos parámetros de forma muy acertada. También ha creado un array para almacenar diccionarios y en base a una key de tipo String poder devolver su UIImage correspondiente si esta ya ha sido procesada anteriormente.

Por desgracia no ha tenido en cuenta que dichos Strings harán uso del Heap e incurrirán en conteo de referencias, necesidad de protección de la información, acceso más lento, etc.
Y aquí es dónde podríamos estar pensando: "¿Pero el String en Swift no se implementa mediante una estructura?, ¿Las estructuras no tenían paso por valor?, ¿no nos has contado que las estructuras usan el Stack?"

En realidad no, ya que esto depende de la implementación interna de String, al igual que, como veremos más adelante en este mismo artículo, la implementación de nuestros tipos de datos serán los que determinen si acaban usando la Stack o el Heap.

¿Cómo podríamos optimizar nuestro código?, Apple nos da la siguiente alternativa:

```swift
enum Color { case blue, green, gray }
enum Orientation { case left, right }
enum Tail { case none, tail, buble }

struct Attributes: Hashable {
    var color: Color
    var orientation: Orientation
    var tail: Tail
}

var cache = [Attributes: UIImage]()

func makeBalloon(_ color: Color, orientation: Orientation, tail: Tail) -> UIImage {
    let key = Attributes(color: color, orientation: orientation, tail: tail)
    if let image = cache[key] { return image }
    // Aquí iría el resto de implementación para devolver el UIImage correspondiente,
    // no nos vale para el ejemplo así que la obvio.
    return UIImage()
}
```

Podemos crear una estructura en la que "almacenar" los valores que necesitamos para la creación de nuestro "balloon". Necesitamos que implemente el protocolo Hashable para que pueda hacer de clave en el diccionario. Con esto, además, evitaríamos usar claves que bien podrían no tener nada que ver con el contenido a almacenar. Pero, ¿realmente es notable el cambio?, usemos a nuestro buen amigo XCTest para realizar unas cuantas pruebas de rendimiento, en una de ellas usaremos una estructura para generar la clave que se usaría en el diccionario, en otra usaríamos una clase, también con el protocolo Hashable implementado, y por último usaríamos un String, generándolo igual a cómo se hacía en el ejemplo inicial:

```swift
enum Color { case blue, green, gray }
enum Orientation { case left, right }
enum Tail { case none, tail, buble }

struct Attributes: Hashable {
    var color: Color
    var orientation: Orientation
    var tail: Tail
}

class AttributesClass: Hashable {
    var color: Color
    var orientation: Orientation
    var tail: Tail

    init(color: Color, orientation: Orientation, tail: Tail) {
        self.color = color
        self.orientation = orientation
        self.tail = tail
    }

    func hash(into hasher: inout Hasher) {
        hasher.combine(color)
        hasher.combine(orientation)
        hasher.combine(tail)
    }

    static func == (lhs: AttributesClass, rhs: AttributesClass) -> Bool {
        lhs.hashValue == rhs.hashValue
    }
}

final class MemoryHeapAndStacksTests: XCTestCase {

    func testWithString() throws {
        self.measure {
            for _ in 1...1_000_000 {
                _ = "\(Color.blue):\(Orientation.left):\(Tail.buble)"
            }
        }
    }

    func testWithClass() throws {
        self.measure {
            for _ in 1...1_000_000 {
                _ = AttributesClass(color: .blue, orientation: .left, tail: .buble)
            }
        }
    }

    func testWithStruct() throws {
        self.measure {
            for _ in 1...1_000_000 {
                _ = Attributes(color: .blue, orientation: .left, tail: .buble)
            }
        }
    }
}
```

El primer test, usando Strings, tardó 1.7 segundos de media en ejecutarse (se ejecuta 10 veces y nos muestra la media de todas las ejecuciones). El segundo, con clases, 0.198 segundos y el tercero, usando estructuras, 0.160 segundos.

Con estos resultados podemos apreciar el alto coste de la asignación de strings en memoria. Por otra parte, la diferencia entre usar estructuras y clases puede no parecer muy grande, aproximadamente un 19%, pero hay que tener en cuenta que esto es solo el coste de su asignación, habría que sumarle también el posterior coste del uso de la clase en relación a la estructura...

Estructuras más ineficientes que las clases

¿Quiere esto decir que una estructura va a ser siempre más eficiente que una clase?, de nuevo, por desgracia, no. Y, de nuevo, dependerá de la implementación de la propia estructura. En la implementación de ejemplo para la estructura Point vimos que no se hacía uso del Heap, no había recuento de referencias, todo se almacenaba en la Stack, etc. ¿Pero qué pasaría con estructuras más complejas?, revisemos el siguiente ejemplo:

```
import UIKit

struct Label {
    var text: String
    var font: UIFont
}

let label1 = Label(text: "Hola!", font: .boldSystemFont(ofSize: 10))
let label2 = label1
```

Nos encontramos con una estructura que tiene dos propiedades, una de tipo String y otra de tipo UIFont. Como vimos en la sección anterior, String hace uso del Heap aún estando implementada como Struct y UIFont es una clase por lo que también haría uso de éste.

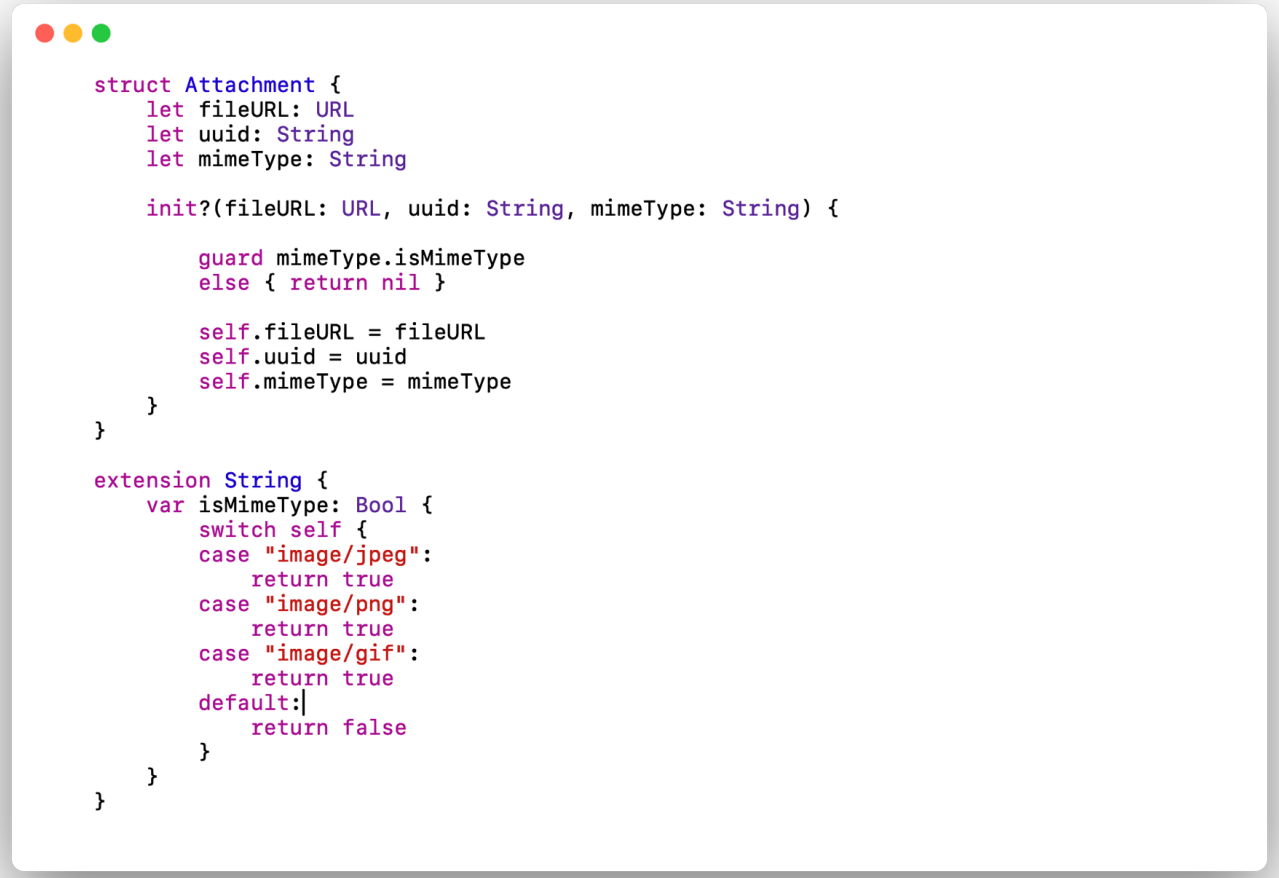
¿En qué se traduce esto?, la instancia **label1** tendría dos referencias, una para el String y otra para el UIFont. Y al hacer la copia con *"let label2 = label1"* estaríamos agregando dos referencias más, una para cada una de las propiedades.

En la conferencia de desarrolladores de 2016, Apple, comentaba que la gestión del recuento de dichas referencias no era algo trivial debido a que se realizaba con mucha frecuencia.

En este caso particular, el uso de estructuras incurriría en una sobrecarga del doble de referencias que las que habría tenido que gestionar de tratarse de una clase.

Optimicemos otro ejemplo

Revisemos otro ejemplo, en este caso se trata de una abstracción para un archivo adjunto que podría ser usado por un programa de gestión de correo electrónico.



```
struct Attachment {
    let fileURL: URL
    let uuid: String
    let mimeType: String

    init?(fileURL: URL, uuid: String, mimeType: String) {

        guard mimeType.isMimeType
        else { return nil }

        self.fileURL = fileURL
        self.uuid = uuid
        self.mimeType = mimeType
    }
}

extension String {
    var isMimeType: Bool {
        switch self {
        case "image/jpeg":
            return true
        case "image/png":
            return true
        case "image/gif":
            return true
        default:
            return false
        }
    }
}
```

Si tenemos en cuenta todo lo leído en la sección anterior, esta estructura, estaría incurriendo en un mayor recuento de referencias. ¿Vemos cómo mejorarlo?

```swift
enum MimeType: String {
    case jpeg = "image/jpeg"
    case png = "image/png"
    case gif = "image/gif"
}

struct Attachment {
    let fileURL: URL
    let uuid: UUID
    let mimeType: MimeType

    init?(fileURL: URL, uuid: UUID, mimeType: String) {
        guard let mimeType = MimeType(rawValue: mimeType)
        else { return nil }

        self.fileURL = fileURL
        self.uuid = uuid
        self.mimeType = mimeType
    }
}
```

En primer lugar podemos usar UUID, disponible desde iOS 6.0. Usando **UUID** conseguimos un identificador de 128 bits, generado aleatoriamente. Así, de paso, evitamos que pueda usarse como identificador único cualquier String que bien pudiera no tener nada que ver con su propósito. **UUID** es un tipo de dato por valor, almacena esos 128 bits directamente en la estructura, en su Stack correspondiente, sin hacer uso de ningún tipo de sobrecarga en el conteo de referencias.

Para el tipo de adjunto podemos usar los enumerados, que en Swift son muy potentes. Así pasamos, de nuevo, de tener una propiedad con recuento de referencias, y uso del Heap, a otra propiedad con almacenamiento en la propia estructura, dentro del Stack correspondiente.

Seguiríamos teniendo que gestionar una referencia, ya que la propiedad de tipo URL, aún siendo implementada como Struct, requiere recuento de referencias ya que se asignaría directamente en el Heap. Sería un caso similar al de String, ambos son estructuras, pero su implementación interna hace que requieran del uso de referencias.

Envío de métodos (estático y dinámico)

¿Recordáis que cuando hablábamos de espacios de memoria hacíamos referencia a dos espacios extra en el caso de usar clases?, si volvéis atrás en este mismo artículo encontraréis que uno de esos "espacios extra", Swift, lo dedicaba a guardar el recuento de referencias. Bien, veamos a qué dedica el restante.

Cuando usamos un método de clase, en tiempo de ejecución, Swift necesita saber que implementación del método es la correcta. Si es capaz de determinarlo en tiempo de compilación, Swift, podrá optimizar nuestro código de forma más eficiente. Esto es lo que llamamos "static dispatch", que puede ser traducido como despachado estático, envío estático, etc.

Este sistema de "envío", o "despachado", contrasta con el llamado "dynamic dispatch", envío o despachado dinámico. En este sistema, Swift, no podrá determinar en tiempo de compilación la implementación adecuada por lo que deberá "buscarla" en tiempo de ejecución y servirla. Esta búsqueda, en sí misma, no supone una pérdida de rendimiento excesiva en contraste al envío estático, pero sí que perdemos todas aquellas optimizaciones que, en tiempo de compilación, Swift podría haber aplicado a nuestro código.

Ok, volvamos al código para verlo con un ejemplo. Volvamos a la estructura Point y agreguémosle un método para dibujar el punto, no le agregaremos ninguna implementación porque no es importante para el caso:

```
struct Point {
    var x, y: Double
    func draw() {
        // ...
    }
}

func drawAPoint(_ point: Point) {
    point.draw()
}

let point = Point(x: 0, y: 0)
drawAPoint(point)
```

Nada complejo en el código mostrado. Pero en este ejemplo hay una parte del código que es candidata a ser optimizada, de forma automática, por el compilador mediante una técnica que, en Swift, se denomina **"inlining"**:

```
struct Point {
    var x, y: Double
    func draw() {
        // ...
    }
}

func drawAPoint(_ point: Point) {
    point.draw()
}

let point = Point(x: 0, y: 0)
point.draw() // <- (inlining)
```

Es un ejemplo muy simple en el que la llamada a la función drawAPoint ha sido sustituida, en tiempo de compilación, por la llamada directa al método draw de la instancia point. Veamos otro ejemplo sencillo:

```
func addOne(to num: Int) -> Int {
    return num + 1
}

let twoPlusOne = addOne(to: 2)
```

En el caso de que el compilador decidiese usar **"inlining"** con este código el código finalmente compilado podría ser similar a este otro ejemplo:

```
let twoPlusOne = 2 + 1
// o incluso:
let twoPlusOne = 3
```

¿Por qué 'let twoPlusOne = 3'?, porque en este ejemplo el compilador ya tendría toda la información necesaria para calcular el resultado, con lo que a la optimización **"inlining"** le sumaríamos una optimización adicional al tener el resultado de la ecuación en tiempo de compilación, sin necesidad de calcularla en tiempo de ejecución.

Como se trata de un solo ejemplo puede no dársele la importancia adecuada, cuando realmente la tiene ya que estas optimizaciones podrían afectar a estructuras de datos que estuviesen calculando algunas de estas operaciones en tiempo de ejecución un número muy alto de veces...

Resumiendo: **"inlining"** se refiere al proceso en el que el compilador reemplaza una función o método por su contenido directamente en el lugar donde se utiliza. Esto se hace para mejorar el rendimiento del programa al minimizar las llamadas a funciones y eliminar la necesidad de guardar y restaurar estados previos en instancias de estado compartido, etc.

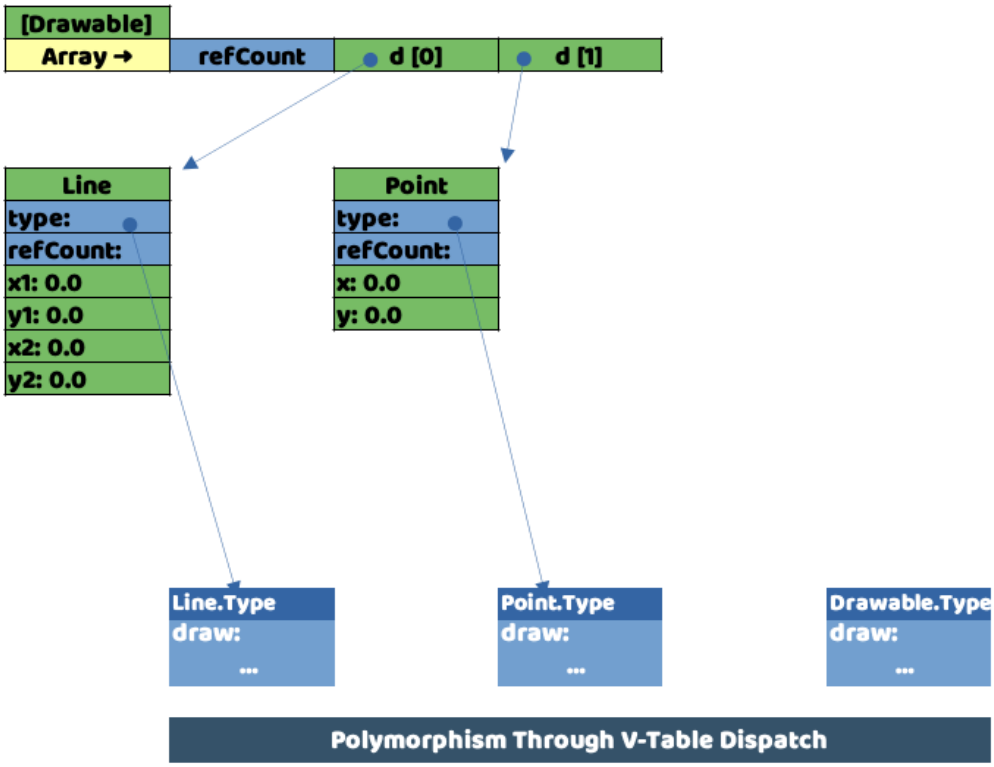
Bien, sigamos revisando código, que ya nos acercamos a completar el enigma de los espacios reservados para clases...

```swift
class Drawable { func draw() { } }

class Point: Drawable {

    var x, y: Double

    init(x: Double, y: Double) {
        self.x = x
        self.y = y
    }

    override func draw() { }
}

class Line: Drawable {

    var x1, y1, x2, y2: Double

    init(x1: Double, y1: Double, x2: Double, y2: Double) {
        self.x1 = x1
        self.y1 = y1
        self.x2 = x2
        self.y2 = y2
    }

    override func draw() { }
}

var drawables = [Drawable]()

// ... aquí un usuario podría estar creando diferentes tipos
// de objetos Drawable, bien puntos o bien líneas, e ir
// agregándolos a nuestra array de "drawables".

for drawable in drawables {
    drawable.draw()
}
```

Aquí encontramos una clase Drawable. Esta clase cuenta con una función draw(). Si te estás adelantando y en tu cabeza ya hay una vocecita diciendo "eso no debería ser una clase, debería estar implementado como protocolo...": vale, pero cómprame el ejemplo, o, mejor aún, cómpraselo a Apple, que es suyo...

También encontramos dos clases hijas de Drawable, Point y Line, ambas sobrecargan la función draw() de su clase padre e implementan su propia funcionalidad. En el caso de que trabajásemos con un Array de objetos de tipo Drawable y fuésemos agregándole diferentes instancias, tanto de puntos como de líneas, nos encontraríamos con un problema. Al recorrer el array y usar la función draw() de cada objeto de tipo Drawable, ¿qué implementación debe usar?

Aquí entra en juego el espacio restante: un puntero a la información de tipo de la clase correspondiente. Vamos a echar un vistazo a la siguiente representación:



¿Qué podemos sacar en claro de esta representación de la memoria?, pues por ejemplo, que los arrays también tienen su propio espacio reservado para contar referencias, lo que nos vendría a indicar que también se almacena en el Heap de la memoria y que cada uno de los Stack que lo usen estarán almacenando una referencia de memoria a éste.

Line y Point, aún siendo clases hijas de Drawable, tienen tamaños diferentes ya que Line debe almacenar más información que Point, ¿por qué no es un problema para el array?, porque en cada uno de los espacios reservados (d[0] y d[1]) lo que se almacena es la referencia a la dirección de memoria de cada uno de los objetos.

Y, por fin, llegamos a completar el dato que nos faltaba. En el espacio reservado por cada instancia de la clase lo que almacenamos es el, anteriormente mencionado, puntero a la dirección de memoria donde Swift ha almacenado la implementación del método a ejecutar para cada tipo de dato.

La parte inferior de la imagen la podemos llamar "tabla de métodos", o "tabla virtual de métodos". Como dato, este comportamiento es común a otros lenguajes de programación orientados a objetos como Java o C++.

De nuevo podemos volver a pensar que no debería de suponer una gran diferencia, pero estaríamos obviando muchos casos en los que no conocer el método a "servir", de modo estático, estaría evitando toda una serie de optimizaciones para nuestro código. Por ejemplo, en un encadenamiento de métodos el encontrar este nivel de indirección provocaría que el compilador no pudiese optimizar el resto de llamadas posteriores, aún en el caso de que el resto de "envíos" pudieran calcularse en tiempo de compilación.

¿Entiendes un poco mejor por qué debemos marcar como **"final"** todas aquellas clases que no vayan a

tener subclases que hereden de ésta?, efectivamente, no se trataba de una manía del Senior de turno. El compilador será consciente de esto y realizará el envío estático de métodos para las instancias de esta clase.

Conclusión:

En resumen, hemos explorado en profundidad el uso de la memoria Heap y Stack en Swift, comprendiendo las diferencias fundamentales entre ambas áreas y cómo afectan al rendimiento y la seguridad de nuestras aplicaciones.

Es importante destacar que el objetivo no es demonizar el uso de clases en Swift, sino entender cuándo y cómo utilizarlas de manera efectiva. Las clases son una herramienta poderosa que proporciona Swift, pero debemos ser conscientes de sus implicaciones en términos de asignación dinámica de memoria, gestión de referencias y rendimiento.

Por lo tanto, la clave está en utilizar las clases cuando realmente necesitemos sus capacidades de herencia, polimorfismo y referencia compartida entre instancias. En casos donde estas características no sean necesarias, como en tipos de datos simples o en estructuras que actúan como valores, es preferible optar por estructuras, que ofrecen un rendimiento más eficiente al utilizar la pila de memoria y evitar la sobrecarga de gestión de referencias.

En última instancia, la elección entre clases y estructuras en Swift debe basarse en una comprensión clara de los requisitos de diseño y las características de rendimiento de cada opción, buscando siempre maximizar la eficiencia del código y sin olvidar su claridad, pero esto da para otro artículo...