Estructuras, primera parte

Estructuras, primera parte

Sé que algunos de ustedes podrían estar dispuestos a hacerse cargo del nuevo aprendizaje de Swift de hoy, pero esperen: acaban de terminar de aprender sobre los cierres, que son un tema difícil. Y volviste por más. En serio, eso merece mucho respeto.

Y tengo buenas noticias para ti. En primer lugar, no solo puedes evitar pensar en los cierres durante los próximos días, sino que una vez que hayas tenido un descanso, comenzaremos a ponerlos en práctica en proyectos reales de iOS. Por lo tanto, incluso si no estás 100% seguro de cómo funcionan o por qué se necesitan, todo quedará claro: ¡quédate con ello!

De todos modos, el tema de hoy son las estructuras. Las estructuras son una de las formas en que Swift nos permite crear nuestros propios tipos de datos a partir de varios tipos pequeños. Por ejemplo, podrías juntar tres cadenas y un booleano y decir que representa a un usuario en tu aplicación. De hecho, la mayoría de los tipos propios de Swift se implementan como estructuras, incluyendo String, Int, Bool, Array y más.

Estos tipos personalizados (usuarios, juegos, documentos y más) forman el núcleo real del software que construimos. Si los haces bien, a menudo tu código seguirá.

Como dijo una vez Fred Brooks, el autor del libro enormemente influyente The Mythical Man-Month, "el programador al final del ingenio... a menudo puede hacer lo mejor desvincularse de su código, levantarse y contemplar sus datos. La representación es la esencia de la programación".

Además, las estructuras son extremadamente comunes en SwiftUI, porque cada pieza de interfaz de usuario que diseñamos se basa en una estructura, con muchas estructuras en su interior. No son difíciles de aprender, pero para ser justos después de los cierres, ¡casi todo parece más fácil!

Hoy tienes cuatro tutoriales a seguir, donde te reunirás con estructuras personalizadas, propiedades calculadas, observadores de propiedades y más. Una vez que hayas visto cada vídeo y, opcionalmente, hayas pasado por la lectura adicional, hay pruebas cortas para asegurarte de que has entendido lo que se enseñó.

Cómo crear tus propias estructuras

Las estructuras de Swift nos permiten crear nuestros propios tipos de datos personalizados y complejos, completos con sus propias variables y sus propias funciones.

```
struct Album {
    let title: String
    let artist: String
    let year: Int

    func printSummary() {
        print("\(title) (\(year)) by \(artist)")
    }
}
```

Eso crea un nuevo tipo llamado Álbum, con dos constantes de cadena llamadas título y artista, más una constante entera llamada año. También agregué una función simple que resume los valores de nuestras tres constantes.

¿Notas cómo el álbum comienza con una A mayúscula? Ese es el estándar en Swift, y lo hemos estado usando todo el tiempo: piensa en String, Int, Bool, Set, etc. Cuando te refieres a un tipo de datos, usamos el caso camel donde la primera letra está en mayúsculas, pero cuando te refieres a algo dentro del tipo, como una variable o función, usamos el caso camel donde la primera letra está en minúsculas. Recuerde, en su mayor parte, esto es solo una convención en lugar de una regla, pero es útil seguirla.

En este punto, el álbum es como String o Int: podemos hacerlos, asignar valores, copiarlos, etc. Por ejemplo, podríamos hacer un par de álbumes, luego imprimir algunos de sus valores y llamar a sus funciones:

```
let red = Album(title: "Red", artist: "Taylor Swift", year: 2012)
let wings = Album(title: "Wings", artist: "BTS", year: 2016)

print(red.title)
print(wings.artist)

red.printSummary()
wings.printSummary()
```

Observe cómo podemos crear un nuevo álbum como si estuviéramos llamando a una función; solo necesitamos proporcionar valores para cada una de las constantes en el orden en que se definieron.

Como puedes ver, tanto el rojo como las alas provienen de la misma estructura del álbum, pero una vez que los creamos, están separados al igual que crear dos cuerdas.

Puedes ver esto en acción cuando llamamos a printSummary() en cada estructura, porque esa función se refiere al título, el artista y el año. En ambos casos, se imprimen los valores correctos para cada estructura: las impresiones rojas "Red (2012) de Taylor Swift" y las alas imprimen "Wings (2016) de BTS" - Swift entiende que cuando printSummary() se llama en rojo, debe usar las constantes de título, artista y año que también pertenecen al rojo.

Donde las cosas se ponen más interesantes es cuando quieres tener valores que puedan cambiar. Por ejemplo, podríamos crear una estructura para empleados que pueda tomar vacaciones según sea necesario:

```
struct Employee {
    let name: String
    var vacationRemaining: Int

    func takeVacation(days: Int) {
        if vacationRemaining > days {
            vacationRemaining -= days
            print("I'm going on vacation!")
            print("Days remaining: \(vacationRemaining)")
        } else {
            print("Oops! There aren't enough days remaining.")
        }
    }
}
```

Sin embargo, eso en realidad no funcionará: Swift se negará a construir el código.

Verás, si creamos un empleado como una constante usando let, Swift hace que el empleado y todos sus datos sean constantes: podemos llamar a las funciones bien, pero no se debería permitir que esas funciones cambien los datos de la estructura porque lo hicimos constante.

Como resultado, Swift nos hace dar un paso más: cualquier función que solo lea datos está bien como está, pero cualquier que cambie los datos pertenecientes a la estructura debe estar marcada con una palabra clave mutante especial, como esta:

```
mutating func takeVacation(days: Int) {
```

Ahora nuestro código se construirá bien, pero Swift nos evitará llamar a takeVacation() desde estructuras constantes.

En el código, esto está permitido:

```
var archer = Employee(name: "Sterling Archer", vacationRemaining: 14)
archer.takeVacation(days: 5)
print(archer.vacationRemaining)
```

Pero si cambias var archer para dejar que el arquero, encontrarás que Swift se niega a construir tu código de nuevo - estamos tratando de llamar a una función mutante en una estructura constante, lo cual no está permitido.

Vamos a explorar las estructuras en detalle en los próximos capítulos, pero primero quiero dar algunos nombres a las cosas.

Las variables y constantes que pertenecen a las estructuras se llaman propiedades.

Las funciones que pertenecen a las estructuras se denominan métodos.

Cuando creamos una constante o variable a partir de una estructura, lo llamamos una instancia; por ejemplo, podría crear una docena de instancias únicas de la estructura del álbum.

Cuando creamos instancias de estructuras, lo hacemos usando un inicializador como este: Álbum (título: "Wings", artista: "BTS", año: 2016).

Ese último puede parecer un poco extraño al principio, porque estamos tratando nuestra estructura como una función y pasando parámetros. Esto es un poco de lo que se llama azúcar sintáctico: Swift crea silenciosamente una función especial dentro de la estructura llamada init(), utilizando todas las propiedades de la estructura como sus parámetros. Luego trata automáticamente estas dos piezas de código como iguales:

```
var archer1 = Employee(name: "Sterling Archer", vacationRemaining: 14)
var archer2 = Employee.init(name: "Sterling Archer", vacationRemaining: 14
```

De hecho, nos basamos en este comportamiento anteriormente. Cuando introduje Double por primera vez, expliqué que no se puede agregar un Int y un Double y en su lugar necesitas escribir un código como este:

```
let a = 1
let b = 2.0
let c = Double(a) + b
```

Ahora puedes ver lo que realmente está sucediendo aquí: el propio tipo Double de Swift se implementa como una estructura y tiene una función inicializadora que acepta un entero.

Swift es inteligente en la forma en que genera su inicializador, incluso insertando valores predeterminados si los asignamos a nuestras propiedades.

Por ejemplo, si nuestra estructura tuviera estas dos propiedades

```
let name: String
var vacationRemaining = 14
```

Luego, Swift generará silenciosamente un inicializador con un valor predeterminado de 14 para el mantenimiento de vacaciones, lo que hará que ambos sean válidos:

```
let kane = Employee(name: "Lana Kane")
let poovey = Employee(name: "Pam Poovey", vacationRemaining: 35)
```

Consejo: Si asigna un valor predeterminado a una propiedad constante, se eliminará por completo del inicializador. Para asignar un valor predeterminado, pero dejar abierta la posibilidad de anularlo cuando sea necesario, utilice una propiedad variable

¿Cuál es la diferencia entre una estructura y una tupla?

Las tuplas de wift nos permiten almacenar varios valores con nombre diferentes dentro de una sola variable, y una estructura hace lo mismo, así que ¿cuál es la diferencia y cuándo deberías elegir uno sobre el otro?

Cuando solo estás aprendiendo, la diferencia es simple: una tupla es efectivamente solo una estructura sin nombre, como una estructura anónima. Esto significa que puedes definirlo como (nombre: Cadena, edad: Int, ciudad: Cadena) y hará lo mismo que la siguiente estructura:

```
struct User {
    var name: String
    var age: Int
    var city: String
}
```

Sin embargo, las tuplas tienen un problema: si bien son geniales para un uso único, especialmente cuando quieres devolver varios datos de una sola función, pueden ser molestos de usar una y otra vez.

Piénsalo: si tienes varias funciones que funcionan con la información del usuario, ¿prefieres escribir esto?

```
func authenticate(_ user: User) { ... }
func showProfile(for user: User) { ... }
func signOut(_ user: User) { ... }
```

O esto:

```
func authenticate(_ user: (name: String, age: Int, city: String)) { ... }
func showProfile(for user: (name: String, age: Int, city: String)) { ... }
func signOut(_ user: (name: String, age: Int, city: String)) { ... }
```

Piensa en lo difícil que sería agregar una nueva propiedad a tu estructura de usuario (muy fácil de hecho), en comparación con lo difícil que sería agregar otro valor a tu tupla en cualquier lugar donde se usara. (¡Muy difícil y propenso a errores!)

Por lo tanto, use tuplas cuando desee devolver dos o más valores arbitrarios de una función, pero prefiera estructuras cuando tenga algunos datos fijos que desea enviar o recibir varias veces.

¿Cuál es la diferencia entre una función y un método?

Las funciones de Swift nos permiten nombrar una pieza de funcionalidad y ejecutarla repetidamente, y los métodos de Swift hacen lo mismo, así que ¿cuál es la diferencia?

Honestamente, la única diferencia real es que los métodos pertenecen a un tipo, como estructuras, enumeraciones y clases, mientras que las funciones no. Eso es todo, esa es la única diferencia. Ambos pueden aceptar cualquier número de parámetros, incluidos los parámetros variados, y ambos pueden devolver valores. Diablos, son tan similares que Swift todavía usa la palabra clave func para definir un método.

Por supuesto, estar asociado con un tipo específico, como una estructura, significa que los métodos obtienen un superpoder importante: pueden referirse a las otras propiedades y métodos dentro de ese tipo, lo que significa que puede escribir un método describe() para un tipo de usuario que imprime el nombre, la edad y la ciudad del usuario.

Hay una ventaja más en los métodos, pero es bastante sutil: los métodos evitan la contaminación del espacio de nombres. Cada vez que creamos una función, el nombre de esa función comienza a tener significado en nuestro código: podemos escribir wakeUp() y hacer que haga algo. Por lo tanto, si escribes 100 funciones, terminas con 100 nombres reservados, y si escribes 1000 funciones, tienes 1000 nombres reservados.

Eso puede ensuciarse rápidamente, pero al poner funcionalidad en los métodos, restringimos dónde están disponibles esos nombres: wakeUp() ya no es un nombre reservado a menos que escribamos específicamente someUser.wakeUp(). Esto reduce la llamada contaminación, porque si la mayor parte de nuestro código está en métodos, entonces no obtendremos conflictos de nombres por accidente.

¿Por qué tenemos que marcar algunos métodos como mutantes?

Es posible modificar las propiedades de una estructura, pero solo si esa estructura se crea como una variable. Por supuesto, dentro de su estructura no hay forma de saber si trabajará con una estructura variable o una estructura constante, por lo que Swift tiene una solución simple: cada vez que el método de una estructura intente cambiar alguna propiedad, debe marcarla como mutante.

No es necesario hacer nada más que marcar el método como mutante, pero hacer eso le da a Swift suficiente información para evitar que ese método se utilice con instancias de estructura constantes.

Hay dos detalles importantes que encontrarás útiles:

Marcar los métodos como mutantes evitará que el método se llame en estructuras constantes, incluso si el método en sí no cambia realmente ninguna propiedad. Si dices que cambia las cosas, ¡Swift te cree!

Un método que no está marcado como mutante no puede llamar a una función mutante; debe marcar ambos como mutantes.

Cómo calcular los valores de las propiedades de forma dinámica

Las estructuras pueden tener dos tipos de propiedades: una propiedad almacenada es una variable o constante que contiene una pieza de datos dentro de una instancia de la estructura, y una propiedad calculada calcula el valor de la propiedad dinámicamente cada vez que se accede a ella. Esto significa que las propiedades calculadas son una mezcla de propiedades y funciones almacenadas: se accede a ellas como propiedades almacenadas, pero funcionan como funciones.

Como ejemplo, anteriormente teníamos una estructura de empleado que podía hacer un seguimiento de cuántos días de vacaciones quedaban para ese empleado. Aquí hay una versión simplificada:

```
struct Employee {
    let name: String
    var vacationRemaining: Int
}

var archer = Employee(name: "Sterling Archer", vacationRemaining: 14)
archer.vacationRemaining -= 5
print(archer.vacationRemaining)
archer.vacationRemaining -= 3
print(archer.vacationRemaining)
```

Eso funciona como una estructura trivial, pero estamos perdiendo información valiosa: estamos asignando a este empleado 14 días de vacaciones y luego restándolos a medida que se toman los días, pero al hacerlo hemos perdido cuántos días se les concedió originalmente.

Podríamos ajustar esto para usar la propiedad calculada, así:

```
struct Employee {
    let name: String
    var vacationAllocated = 14
    var vacationTaken = 0

    var vacationRemaining: Int {
        vacationAllocated - vacationTaken
    }
}
```

Ahora, en lugar de hacer vacaciones, que sigue siendo algo a lo que podemos asignar directamente, se calcula restando la cantidad de vacaciones que han tomado de la cantidad de vacaciones que se les asignó.

Cuando estamos leyendo de vacationRemaining, parece una propiedad almacenada normal:

```
var archer = Employee(name: "Sterling Archer", vacationAllocated: 14)
archer.vacationTaken += 4
print(archer.vacationRemaining)
archer.vacationTaken += 4
print(archer.vacationRemaining)
```

Esto es algo realmente poderoso: estamos leyendo lo que parece una propiedad, pero detrás de escena Swift está ejecutando algo de código para calcular su valor cada vez.

Sin embargo, no podemos escribirle, porque no le hemos dicho a Swift cómo se debe manejar eso. Para solucionar eso, necesitamos proporcionar tanto un getter como un setter: nombres elegantes para "código que lee" y "código que escribe", respectivamente.

En este caso, el getter es lo suficientemente simple, porque es solo nuestro código existente. Pero el setter es más interesante: si estableces la permanencia de vacaciones para un empleado, ¿quieres decir que quieres que se aumente o disminuya el valor asignado de sus vacaciones, o si las vacaciones asignadas deberían seguir siendo las mismas y en su lugar cambiamos las vacaciones?

Voy a suponer que el primero de esos dos es correcto, en cuyo caso así es como se vería la propiedad:

```
var vacationRemaining: Int {
    get {
        vacationAllocated - vacationTaken
    }

    set {
        vacationAllocated = vacationTaken + newValue
    }
}
```

Observe cómo obtener y establecer la marca de piezas individuales de código para que se ejecuten al leer o escribir un valor. Lo que es más importante, observe newValue, que Swift nos proporciona automáticamente y almacena el valor que el usuario estaba tratando de asignar a la propiedad.

Con un getter y un setter en su lugar, ahora podemos modificar el mantenimiento de vacaciones:

```
var archer = Employee(name: "Sterling Archer", vacationAllocated: 14)
archer.vacationTaken += 4
archer.vacationRemaining = 5
print(archer.vacationAllocated)
```

SwiftUI utiliza extensamente las propiedades calculadas, ¡las verás en el primer proyecto que crees!

¿Cuándo debe usar una propiedad computada o una propiedad almacenada?

Las propiedades nos permiten adjuntar información a las estructuras, y Swift nos da dos variaciones: propiedades almacenadas, donde un valor se guarda en alguna memoria para ser utilizado más tarde, y propiedades calculadas, donde un valor se vuelve a calcular cada vez que se llama. Detrás de escena, una propiedad calculada es en realidad solo una llamada de función que pertenece a su estructura.

Decidir cuál usar depende en parte de si el valor de su propiedad depende de otros datos, y en parte también del rendimiento. La parte de rendimiento es fácil: si lee regularmente la propiedad cuando su valor no ha cambiado, entonces usar una propiedad almacenada será mucho más rápido que usar una propiedad calculada. Por otro lado, si su propiedad se lee muy raramente y tal vez no en absoluto, entonces el uso de una propiedad calculada le ahorra tener que calcular su valor y almacenarla en algún lugar.

Cuando se trata de dependencias, ya sea que el valor de su propiedad dependa de los valores de sus otras propiedades, entonces se giran las tablas: este es un lugar donde las propiedades calculadas son útiles, porque puede estar seguro de que el valor que devuelven siempre tiene en cuenta el último estado del programa.

Las propiedades perezosas ayudan a mitigar los problemas de rendimiento de las propiedades almacenadas que rara vez se leen, y los observadores de propiedades mitigan los problemas de dependencia de las propiedades almacenadas; los veremos pronto.

Cómo tomar medidas cuando una propiedad cambia

Swift nos permite crear observadores de propiedades, que son piezas especiales de código que se ejecutan cuando las propiedades cambian. Estos toman dos formas: un observador didSet para ejecutarse cuando la propiedad acaba de cambiar, y un observador willSet para ejecutarse antes de que la propiedad cambie.

Para ver por qué podrían ser necesarios observadores de propiedades, piense en un código como este:

```
struct Game {
    var score = 0
}

var game = Game()
game.score += 10
print("Score is now \(game.score)")
game.score -= 3
print("Score is now \(game.score)")
game.score += 1
```

Eso crea una estructura de juego y modifica su puntuación unas cuantas veces. Cada vez que cambia la puntuación, una línea print() la sigue para que podamos hacer un seguimiento de los cambios. Excepto que hay un error: al final, la puntuación cambió sin ser impresa, lo cual es un error.

Con los observadores de propiedades podemos resolver este problema adjuntando la llamada print() directamente a la propiedad usando didSet, de modo que cada vez que cambie, dondequiera que cambie, siempre ejecutemos algo de código.

Aquí está el mismo ejemplo, ahora con un observador de propiedades en su lugar:

```
struct Game {
    var score = 0 {
        didSet {
            print("Score is now \(score)")
        }
    }
}

var game = Game()
game.score += 10
game.score -= 3
game.score += 1
```

Si lo desea, Swift proporciona automáticamente el oldValue constante dentro de didSet, en caso de que necesite tener una funcionalidad personalizada basada en lo que estaba cambiando. También hay una variante de willSet que ejecuta algo de código antes de que cambie la propiedad, que a su vez proporciona el nuevo valor que se asignará en caso de que desee realizar una acción diferente basada en eso.

Podemos demostrar toda esta funcionalidad en acción utilizando un ejemplo de código, que imprimirá los mensajes a medida que cambien los valores para que pueda ver el flujo cuando se ejecute el código:

Sí, añadir a una matriz activará tanto willSet como didSet, por lo que el código imprimirá mucho texto cuando se ejecute.

En la práctica, willSet se usa mucho menos que didSet, pero es posible que todavía lo veas de vez en cuando, por lo que es importante que sepas que existe. Independientemente de cuál elija, trate de evitar poner demasiado trabajo en los observadores de propiedades: si algo que parece trivial, como game.score += 1, desencadena un trabajo intensivo, lo atrapará de forma regular y causará todo tipo de problemas de rendimiento.

```
struct App {
    var contacts = [String]() {
        willSet {
            print("Current value is: \(contacts)")
            print("New value will be: \(newValue)")
        }

        didSet {
            print("There are now \(contacts.count) contacts.")
            print("Old value was \(oldValue)")
        }
    }
}

var app = App()
app.contacts.append("Adrian E")
app.contacts.append("Allen W")
app.contacts.append("Ish S")
```

¿Cuándo deberías usar observadores de propiedades?

VvvvvvLos observadores de propiedades de Swift nos permiten adjuntar la funcionalidad para que se ejecute antes o después de cambiar una propiedad, utilizando willSet y didSet, respectivamente. La mayoría de las veces no se requieren observadores de propiedades, solo es bueno tenerlo: podríamos actualizar una propiedad normalmente y luego llamar a una función nosotros mismos en código. Entonces, ¿por qué molestarse? ¿Cuándo usarías realmente observadores de propiedades?

La razón más importante es la conveniencia: el uso de un observador de propiedades significa que su funcionalidad se ejecutará cada vez que la propiedad cambie. Claro, podrías usar una función para hacer eso, pero ¿te acuerdas? ¿Siempre? ¿En cada lugar cambias la propiedad?

Aquí es donde el enfoque de la función se vuelve amargo: depende de ti recordar llamar a esa función cada vez que cambie la propiedad, y si lo olvidas, entonces tendrás misteriosos errores en tu código. Por otro lado, si adjuntas tu funcionalidad directamente a la propiedad usando didSet, siempre sucederá, y estás transfiriendo el trabajo de asegurarlo a Swift para que tu cerebro pueda centrarse en problemas más interesantes.

Hay un lugar donde usar un observador de propiedades es una mala idea, y eso es si pones un trabajo lento allí. Si tuvieras una estructura de usuario con un número entero de edad, esperarías que el cambio de edad tuviera lugar prácticamente al instante; después de todo, es solo un número. Si adjunta un observador de propiedades didSet que hace todo tipo de trabajo lento, entonces de repente cambiar un solo entero podría llevar mucho más tiempo de lo que esperaba, y podría causarle todo tipo de problemas.

¿Cuándo deberías usar willSet en lugar de didSet?

Tanto willSet como didSet nos permiten adjuntar observadores a las propiedades, lo que significa que Swift ejecutará algo de código cuando cambien para que tengamos la oportunidad de responder al cambio. La pregunta es: ¿quieres saber antes de que cambie la propiedad o después?

La respuesta simple es esta: la mayoría de las veces usarás didSet, porque queremos tomar medidas después de que se haya producido el cambio para que podamos actualizar nuestra interfaz de usuario, guardar los cambios o lo que sea. Eso no significa que willSet no sea útil, es solo que en la práctica es significativamente menos popular que su contraparte.

El tiempo más común en el que se utiliza willSet es cuando necesita conocer el estado de su programa antes de hacer un cambio. Por ejemplo, SwiftUI utiliza willSet en algunos lugares para manejar animaciones para que pueda tomar una instantánea de la interfaz de usuario antes de un cambio. Cuando tiene la instantánea "antes" y "después", puede comparar las dos para ver todas las partes de la interfaz de usuario que necesitan ser actualizadas.

Cómo crear inicializadores personalizados

Los inicializadores son métodos especializados que están diseñados para preparar una nueva instancia de estructura que se utilizará. Ya has visto cómo Swift genera silenciosamente uno para nosotros en función de las propiedades que colocamos dentro de una estructura, pero también puedes crear el tuyo propio siempre y cuando sigas una regla de oro: todas las propiedades deben tener un valor para el momento en que termine el inicializador.

Comencemos mirando de nuevo el inicializador predeterminado de Swift para las estructuras:

```
struct Player {
    let name: String
    let number: Int
}

let player = Player(name: "Megan R", number: 15)
```

Eso crea una nueva instancia de Player al proporcionar valores para sus dos propiedades. Swift llama a esto el inicializador por miembros, que es una forma elegante de decir un inicializador que acepta cada propiedad en el orden en que se definió.

Como dije, este tipo de código es posible porque Swift genera silenciosamente un inicializador que acepta esos dos valores, pero podríamos escribir los nuestros para hacer lo mismo. El único tracto aquí es que debe tener cuidado de distinguir entre los nombres de los parámetros que entran y los nombres de las propiedades que se están asignando.

Así es como se vería eso:

```
struct Player {
    let name: String
    let number: Int

    init(name: String, number: Int) {
        self.name = name
        self.number = number
    }
}
```

por lo que podemos agregar funcionalidad adicional allí si es necesario.

Sin embargo, hay un par de cosas que quiero que notes:

- No hay una palabra clave func. Sí, esto parece una función en términos de sintaxis, pero Swift trata a los inicializadores de manera especial.
- A pesar de que esto crea una nueva instancia de Player, los inicializadores nunca tienen explícitamente un tipo de retorno: siempre devuelven el tipo de datos al que pertenecen.
- He utilizado self para asignar parámetros a las propiedades para aclarar que queremos decir "asignar el parámetro de nombre a mi propiedad de nombre".

Ese último punto es particularmente importante, porque sin uno mismo tendríamos nombre = nombre y eso no tiene sentido: ¿estamos asignando la propiedad al parámetro, asignando el parámetro a sí mismo o algo más? Al escribir self.name estamos aclarando que nos referimos a "la propiedad del nombre que pertenece a mi instancia actual", en lugar de cualquier otra cosa.

Por supuesto, nuestros inicializadores personalizados no necesitan funcionar como el inicializador predeterminado por miembros que Swift nos proporciona. Por ejemplo, podríamos decir que debes proporcionar un nombre de jugador, pero el número de camiseta es aleatorio:

```
struct Player {
    let name: String
    let number: Int

    init(name: String) {
        self.name = name
        number = Int.random(in: 1...99)
    }
}

let player = Player(name: "Megan R")
print(player.number)
```

Solo recuerda la regla de oro: todas las propiedades deben tener un valor en el momento en que termine el inicializador. Si no habiéramos proporcionado un valor para el número dentro del inicializador, Swift se negaría a construir nuestro código.

Importante: Aunque puede llamar a otros métodos de su estructura dentro de su inicializador, no puede hacerlo antes de asignar valores a todas sus propiedades. Swift necesita asegurarse de que todo esté seguro antes de hacer cualquier otra cosa.

Puede agregar varios inicializadores a sus estructuras si lo desea, así como aprovechar características como los nombres de los parámetros externos y los valores predeterminados. Sin embargo, tan pronto como implemente sus propios inicializadores personalizados, perderá el acceso al inicializador de miembros generado por Swift, a menos que tome medidas adicionales para conservarlo. Esto no es un accidente: si tienes un inicializador personalizado, Swift asume efectivamente que es porque tienes una forma especial de inicializar tus propiedades, lo que significa que la predeterminada ya no debería estar disponible.

¿Cómo funcionan los inicializadores por miembros de Swift?

De forma predeterminada, todas las estructuras de Swift obtienen un inicializador de miembros sintetizado de forma predeterminada, lo que significa que obtenemos automáticamente un inicializador que acepta valores para cada una de las propiedades de la estructura. Este inicializador hace que sea fácil trabajar con estructuras, pero Swift hace dos cosas más que lo hacen especialmente inteligente.

En primer lugar, si alguna de sus propiedades tiene valores predeterminados, se incorporarán al inicializador como valores de parámetros predeterminados. Entonces, si hago una estructura como esta:

```
struct Employee {
    var name: String
    var yearsActive = 0
}
```

Entonces puedo crearlo de cualquiera de estas dos maneras:

```
let roslin = Employee(name: "Laura Roslin")
let adama = Employee(name: "William Adama", yearsActive: 45)
```

Esto los hace aún más fáciles de crear, porque solo puedes rellenar las partes que necesitas.

La segunda cosa inteligente que hace Swift es eliminar el inicializador por miembros si creas un inicializador propio.

Por ejemplo, si tuviera un inicializador personalizado que creara empleados anónimos, se vería así:

```
struct Employee {
    var name: String
    var yearsActive = 0

    init() {
        self.name = "Anonymous"
        print("Creating an anonymous employee…")
    }
}
```

Con eso en su lugar, ya no podía confiar en el inicializador por miembros, por lo que esto ya no estaría permitido:

```
let roslin = Employee(name: "Laura Roslin")
```

Esto no es un accidente, pero es una característica deliberada: creamos nuestro propio inicializador, y si Swift dejó su inicializador por miembro en su lugar, entonces podría faltar un trabajo importante que pusimos en nuestro propio inicializador.

Por lo tanto, tan pronto como agregue un inicializador personalizado para su estructura, el inicializador predeterminado por miembros desaparece. Si quieres que se quede, mueve tu inicializador personalizado a una extensión, como esta:

```
struct Employee {
    var name: String
    var yearsActive = 0
}

extension Employee {
    init() {
        self.name = "Anonymous"
        print("Creating an anonymous employee…")
    }
}

// creating a named employee now works
let roslin = Employee(name: "Laura Roslin")

// as does creating an anonymous employee
let anon = Employee()
```

¿Cuándo te usarías a ti mismo en un método?

Dentro de un método, Swift nos permite referirnos a la instancia actual de una estructura usando el yo, pero en términos generales, no quieres a menos que necesites distinguir específicamente lo que quieres decir.

Con mucho, la razón más común para usar self es dentro de un inicializador, donde es probable que quieras nombres de parámetros que coincidan con los nombres de propiedades de tu tipo, como este:

```
struct Student {
    var name: String
    var bestFriend: String

    init(name: String, bestFriend: String) {
        print("Enrolling \(name) in class…")
        self.name = name
        self.bestFriend = bestFriend
    }
}
```

No tienes que usar eso, por supuesto, pero se vuelve un poco torpe añadir algún tipo de prefijo a los nombres de los parámetros:

```
struct Student {
    var name: String
    var bestFriend: String

    init(name studentName: String, bestFriend studentBestFriend: String) {
        print("Enrolling \(studentName) in class…")
        name = studentName
        bestFriend = studentBestFriend
    }
}
```

Fuera de los inicializadores, la razón principal para usar el auto es porque estamos en un cierre y Swift lo requiere para que tengamos claro que entendemos lo que está sucediendo. Esto solo es necesario cuando se accede a sí mismo desde dentro de un cierre que pertenece a una clase, y Swift se negará a construir su código a menos que lo agregue.