Los Closures

Prepárate, porque hoy estamos cubriendo un tema inicial en Swift que a muchas personas les cuesta entender. Por favor, ten en cuenta la ley de Flip Wilson: "no puedes esperar ganar el premio mayor si no pones unas cuantas monedas de cinco centavos en la máquina".

Hoy en día, el tema son los cierres, que son un poco como funciones anónimas, funciones que podemos crear y asignar directamente a una variable, o pasar a otras funciones para personalizar cómo funcionan. Sí, lo has leído bien: pasar una función a otra como parámetro.

Los cierres son bastante difíciles. No lo digo para desanimarte, solo para que sepas de antemano que si encuentras los cierres difíciles de entender o difíciles de recordar, está bien, ¡todos hemos estado ahí!

A veces, la sintaxis de los cierres puede ser un poco difícil para tus ojos, y esto será realmente evidente a medida que trabajes en las lecciones de hoy. Si lo encuentras un poco abrumador, si estás mirando algún código y no estás 100% seguro de lo que significa, simplemente retrocede un vídeo y míralo de nuevo para refrescar tu memoria. Encontrarás que hay más pruebas y enlaces de lectura opcionales de lo habitual a continuación, lo que espero que ayude a solidificar tus conocimientos.

SwiftUI utiliza los cierres ampliamente, por lo que vale la pena tomarse el tiempo para entender lo que está pasando aquí. Sí, los cierres son probablemente la característica más compleja de Swift, pero es un poco como subir en bicicleta una colina: una vez que has llegado a la cima, una vez que has dominado los cierres, todo se vuelve mucho más fácil.

Hoy tienes tres tutoriales para seguir, además de un resumen y otro punto de control. Como siempre, una vez que hayas completado cada vídeo, hay algunas lecturas adicionales opcionales y pruebas cortas para asegurarte de que has entendido lo que se enseñó. Esta vez te darás cuenta de que hay bastante de cada uno de ellos porque los cierres realmente pueden llevar algún tiempo entenderse, ¡así que no tengas miedo de explorar!

Cómo crear y usar cierres

Las funciones son cosas poderosas en Swift. Sí, has visto cómo puedes llamarlos, pasar valores y devolver datos, pero también puedes asignarlos a variables, pasar funciones a funciones e incluso devolver funciones de funciones.

Por ejemplo:

```
func greetUser() {
    print("Hi there!")
}

greetUser()

var greetCopy = greetUser
greetCopy()
```

Eso crea una función trivial y la llama, pero luego crea una copia de esa función y llama a la copia. Como resultado, imprimirá el mismo mensaje dos veces.

Importante: Cuando estás copiando una función, no escribes los paréntesis después de ella, es var greetCopy = greetUser y no var greetCopy = greetUser(). Si pones los paréntesis allí, estás llamando a la función y asignando su valor de retorno a otra cosa.

Pero, ¿y si quisieras omitir la creación de una función separada y simplemente asignar la funcionalidad directamente a una constante o variable? Bueno, resulta que tú también puedes hacer eso:

```
let sayHello = {
    print("Hi there!")
}

sayHello()
```

Swift le da a esto la grandiosa expresión de cierre del nombre, que es una forma elegante de decir que acabamos de crear un cierre, un trozo de código que podemos pasar y llamar cuando queramos. Este no tiene un nombre, pero por lo demás es efectivamente una función que no toma parámetros y no devuelve un valor.

Si quieres que el cierre acepte parámetros, deben escribirse de una manera especial. Verás, el cierre comienza y termina con las llaves, lo que significa que no podemos poner código fuera de esas llaves para controlar los parámetros o devolver el valor. Por lo tanto, Swift tiene una solución adecuada: podemos poner esa misma información dentro de los frenos, así:

```
let sayHello = { (name: String) -> String in
    "Hi \(name)!"
}
```

Agregué una palabra clave adicional allí, ¿la has visto? Es la palabra clave in, y viene directamente después de los parámetros y el tipo de retorno del cierre. Una vez más, con una función regular, los parámetros y el tipo de retorno saldrían de las llaves, pero no podemos hacer eso con los cierres. Por lo tanto, in se utiliza para marcar el final de los parámetros y el tipo de retorno, todo después de eso es el cuerpo del cierre en sí. Hay una razón para esto, y lo verás por ti mismo muy pronto.

Mientras tanto, es posible que tengas una pregunta más fundamental: "¿por qué necesitaría estas cosas?" Lo sé, los cierres parecen terriblemente oscuros. Peor aún, parecen oscuros y complicados: muchas, muchas personas realmente luchan con los cierres cuando se encuentran por primera vez, porque son bestias complejas y parece que nunca van a ser útiles.

Sin embargo, como verás, esto se usa ampliamente en Swift, y en casi todas partes en SwiftUI. En serio, los usarás en todas las aplicaciones SwiftUI que escribas, a veces cientos de veces, tal vez no necesariamente en el formulario que ves arriba, pero lo vas a usar mucho.

Para tener una idea de por qué los cierres son tan útiles, primero quiero presentarte los tipos de funciones. Has visto cómo los enteros tienen el tipo Int, y los decimales tienen el tipo Doble, etc., y ahora quiero que pienses en cómo las funciones también tienen tipos.

Tomemos la función greetUser() que escribimos antes: no acepta parámetros, no devuelve ningún valor y no lanza errores. Si tuviéramos que escribir eso como una anotación de tipo para greetCopy, escribiríamos esto:

```
var greetCopy: () -> Void = greetUser
```

Vamos a desglosar eso...

- Los paréntesis vacíos marcan una función que no toma parámetros.
- La flecha significa justo lo que significa al crear una función: estamos a punto de declarar el tipo de retorno de la función.
- **Void** significa "nada" - esta función no devuelve nada. A veces puedes ver esto escrito como (), pero generalmente lo evitamos porque se puede confundir con la lista de parámetros vacíos.

El tipo de cada función depende de los datos que reciba y envíe de vuelta. Eso puede sonar simple, pero oculta una trampa importante: los nombres de los datos que recibe no forman parte del tipo de función.

Podemos demostrar esto con un poco más de código:

```
func getUserData(for id: Int) -> String {
    if id == 1989 {
        return "Taylor Swift"
    } else {
        return "Anonymous"
    }
}

let data: (Int) -> String = getUserData
let user = data(1989)
print(user)
```

Eso comienza con bastante facilidad: es una función que acepta un entero y devuelve una cadena. Pero cuando tomamos una copia de la función, el tipo de función no incluye el nombre del parámetro externo, por lo que cuando se llama a la copia, usamos datos (1989) en lugar de datos (para: 1989).

Astutamente, esta misma regla se aplica a todos los cierres: es posible que te hayas dado cuenta de que en realidad no usé el cierre sayHello que escribimos antes, y eso se debe a que no quería dejarte cuestionando la falta de un nombre de parámetro en el sitio de llamada. Llamémoslo ahora:

```
sayHello("Taylor")
```

Eso no usa ningún nombre de parámetro, al igual que cuando copiamos funciones. Así que, de nuevo: los nombres de parámetros externos solo importan cuando estamos llamando a una función directamente, no cuando creamos un cierre o cuando tomamos una copia de la función primero.

Probablemente todavía te estés preguntando por qué todo esto importa, y todo está a punto de quedar claro. ¿Recuerdas que dije que podemos usar sorted() con una matriz para que ordene sus elementos?

Significa que podemos escribir código como este:

```
let team = ["Gloria", "Suzanne", "Piper", "Tiffany", "Tasha"]
let sortedTeam = team.sorted()
print(sortedTeam)
```

Eso es realmente genial, pero ¿y si quisiéramos controlar ese tipo? ¿Y si siempre quisiéramos que una persona fuera lo primero porque era el capitán del equipo, y el resto está ordenado alfabéticamente?

Bueno, sorted() en realidad nos permite pasar una función de clasificación personalizada para controlar exactamente eso. Esta función debe aceptar dos cadenas, y devolver true si la primera cadena debe ordenarse antes de la segunda, o false si la primera cadena debe ordenarse después de la segunda.

Si Suzanne fuera la capitana, la función se vería así:

```
func captainFirstSorted(name1: String, name2: String) -> Bool {
    if name1 == "Suzanne" {
        return true
    } else if name2 == "Suzanne" {
        return false
    }

    return name1 < name2
}
```

Por lo tanto, si el primer nombre es Suzanne, devuelva true para que el nombre1 esté ordenado antes del nombre2. Por otro lado, si el nombre 2 es Suzanne, devuelve false para que el nombre 1 se ordene después del nombre2. Si ninguno de los nombres es Suzanne, solo usa < para hacer una clasificación alfabética normal.

Como dije, a sorted() se le puede pasar una función para crear un orden de clasificación personalizado, y siempre y cuando esa función a) acepte dos cadenas y b) devuelva un booleano, sorted() puede usarlo.

Eso es exactamente lo que hace nuestra nueva función captainFirstSorted(), para que podamos usarla de inmediato:

```
let captainFirstTeam = team.sorted(by: captainFirstSorted)
print(captainFirstTeam)
```

Cuando eso se ejecute, se imprimirá ["Suzanne", "Gloria", "Piper", "Tasha", "Tiffany"], exactamente como queríamos.

Ahora hemos cubierto dos cosas aparentemente muy diferentes. En primer lugar, podemos crear cierres como funciones anónimas, almacenándolos dentro de constantes y variables:

```
let sayHello = {
    print("Hi there!")
}

sayHello()
```

Y también podemos pasar funciones a otras funciones, al igual que pasamos captainFirstSorted() a sorted():

```
let captainFirstTeam = team.sorted(by: captainFirstSorted)
```

El poder de los cierres es que podemos juntar estos dos: sorted() quiere una función que acepte dos cadenas y devuelva un booleano, y no le importa si esa función se creó formalmente usando func o si se proporciona usando un cierre.

Por lo tanto, podríamos llamar a sorted() de nuevo, pero en lugar de pasar la función captainFirstTeam(), en su lugar, comenzar un nuevo cierre: escribir una llave abierta, enumerar sus parámetros y el tipo de retorno, escribir y luego poner nuestro código de función estándar.

Esto va a lastimar tu cerebro al principio. No es porque no seas lo suficientemente inteligente como para entender los cierres o que no estés hecho para la programación Swift, solo que los cierres son realmente difíciles. No te preocupes, ¡vamos a buscar formas de hacer esto más fácil!

Vale, vamos a escribir un código nuevo que llame a sorted() usando un cierre:

```
let captainFirstTeam = team.sorted(by: { (name1: String, name2: String) ->
    if name1 == "Suzanne" {
        return true
    } else if name2 == "Suzanne" {
        return false
    }

    return name1 < name2
})
```

Esa es una gran parte de la sintaxis a la vez, y de nuevo quiero decir que va a ser más fácil - en el próximo capítulo vamos a buscar formas de reducir la cantidad de código para que sea más fácil ver lo que está pasando.

Pero primero quiero desglosar lo que está pasando allí:

- Estamos llamando a la función sorted() como antes.
- En lugar de pasar una función, estamos pasando un cierre: todo, desde el corsé de apertura después de: hasta el corsé de cierre en la última línea es parte del cierre.
- Directamente dentro del cierre enumeramos los dos parámetros sorted() nos pasará, que son dos cadenas. También decimos que nuestro cierre devolverá un booleano, luego marcará el inicio del código de cierre usando in.
- Todo lo demás es solo un código de función normal.

Una vez más, hay mucha sintaxis allí y no te culparía si sientes que viene un dolor de cabeza, pero espero que puedas ver el beneficio de los cierres al menos un poco: funciones como sorted() nos permiten pasar código personalizado para ajustar cómo funcionan, y hacerlo directamente - no necesitamos escribir una nueva función solo para ese uso.

Ahora que entiendes lo que son los cierres, veamos si podemos hacerlos más fáciles de leer…

¿A Swift le gustan tanto?

Adelante, admítelo: te hiciste exactamente esta pregunta. Si no lo hicieras, me sorprendería, porque los cierres son una de las características más poderosas de Swift, pero también fácilmente la característica que confunde a la gente.

Entonces, si estás sentado aquí pensando "guau, los cierres son muy difíciles", no te desanimes, son difíciles, y si los encuentras difíciles solo significa que tu cerebro está funcionando correctamente.

Una de las razones más comunes para los cierres en Swift es almacenar la funcionalidad, poder decir "aquí hay algo de trabajo que quiero que hagas en algún momento, pero no necesariamente ahora". Algunos ejemplos:

- Ejecutando algo de código después de un retraso.
- Ejecutando algo de código después de que una animación haya terminado.
- Ejecutando algo de código cuando una descarga ha terminado.
- Ejecutando algo de código cuando un usuario ha seleccionado una opción de su menú.

Los cierres nos permiten envolver alguna funcionalidad en una sola variable, y luego almacenarla en algún lugar. También podemos devolverlo desde una función y almacenar el cierre en otro lugar.

Cuando estás aprendiendo, los cierres son un poco difíciles de leer, especialmente cuando aceptan y/o devuelven sus propios parámetros. Pero eso está bien: da pequeños pasos y da marcha atrás si te quedas atascado, y estarás bien.

¿Cómo se devuelve un valor de un cierre que no toma parámetros?

Los cierres en Swift tienen una sintaxis distinta que realmente los separa de las funciones simples, y un lugar que puede causar confusión es la forma en que aceptamos y devolvemos los parámetros.

En primer lugar, aquí hay un cierre que acepta un parámetro y no devuelve nada:

```
let payment = { (user: String) in
    print("Paying \(user)…")
}
```

Ahora aquí hay un cierre que acepta un parámetro y devuelve un booleano:

```
let payment = { (user: String) -> Bool in
    print("Paying \(user)…")
    return true
}
```

Si quieres devolver un valor sin aceptar ningún parámetro, no puedes simplemente escribir -> Bool in - Swift no entenderá lo que quieres decir. En su lugar, debe usar paréntesis vacíos para su lista de parámetros, como esta:

```
let payment = { () -> Bool in
    print("Paying an anonymous person…")
    return true
}
```

Si lo piensas, eso funciona igual que una función estándar donde se escribiría func payment() -> Bool.

Cómo usar los cierres finales y la sintaxis abreviada

Swift tiene algunos trucos bajo la manga para reducir la cantidad de sintaxis que viene con los cierres, pero primero recordémonos el problema. Aquí está el código con el que terminamos al final del capítulo anterior:

```
let team = ["Gloria", "Suzanne", "Piper", "Tiffany", "Tasha"]

let captainFirstTeam = team.sorted(by: { (name1: String, name2: String) ->
    if name1 == "Suzanne" {
        return true
    } else if name2 == "Suzanne" {
        return false
    }

    return name1 < name2
})

print(captainFirstTeam)
```

Si recuerdas, sorted() puede aceptar cualquier tipo de función para hacer una clasificación personalizada, con una regla: esa función debe aceptar dos elementos de la matriz en cuestión (es decir, dos cadenas aquí), y devolver un conjunto booleano como verdadero si la primera cadena debe ordenarse antes de la segunda.

Para que quede claro, la función debe comportarse así: si no devolviera nada, o si solo aceptara una cadena, entonces Swift se negaría a construir nuestro código.

Piénsalo bien: en este código, la función que proporcionamos para ordenar() debe proporcionar dos cadenas y devolver un booleano, así que ¿por qué tenemos que repetirnos en nuestro cierre?

La respuesta es: no lo hacemos. No necesitamos especificar los tipos de nuestros dos parámetros porque deben ser cadenas, y no necesitamos especificar un tipo de retorno porque debe ser un booleano. Por lo tanto, podemos reescribir el código en esto:

```
let captainFirstTeam = team.sorted(by: { name1, name2 in
```

Eso ya ha reducido la cantidad de desorden en el código, pero podemos ir un paso más allá: cuando una función acepta otra como su parámetro, como lo hace sorted(), Swift permite una sintaxis especial llamada sintaxis de cierre final. Se ve así:

```
let captainFirstTeam = team.sorted { name1, name2 in
    if name1 == "Suzanne" {
        return true
    } else if name2 == "Suzanne" {
        return false
    }

    return name1 < name2
}
```

En lugar de pasar el cierre como un parámetro, simplemente seguimos adelante y comenzamos el cierre directamente, y al hacerlo eliminamos (por: desde el principio, y un paréntesis de cierre al final). Esperemos que ahora puedas ver por qué la lista de parámetros y entrar dentro del cierre, ¡porque si estuvieran fuera se vería aún más raro!

Hay una última forma en que Swift puede hacer que los cierres sean menos desordenados: Swift puede proporcionarnos automáticamente nombres de parámetros, utilizando la sintaxis abreviada. Con esta sintaxis, ya ni siquiera escribimos name1, name2, sino que dependemos de los valores con nombres especiales que Swift nos proporciona: $0 y $1, para la primera y la segunda cadena, respectivamente.

Usando esta sintaxis, nuestro código se vuelve aún más corto:

```
let captainFirstTeam = team.sorted {
    if $0 == "Suzanne" {
        return true
    } else if $1 == "Suzanne" {
        return false
    }

    return $0 < $1
}
```

Dejé este para que durara porque no es tan claro como los otros - algunas personas ven esa sintaxis y la odian porque es menos clara, y eso está bien.

Personalmente, no lo usaría aquí porque estamos usando cada valor más de una vez, pero si nuestra llamada sorted() fuera más simple, por ejemplo, si solo quisiéramos hacer una clasificación inversa, entonces lo haría:

```
let reverseTeam = team.sorted {
    return $0 > $1
}
```

Por lo tanto, in se utiliza para marcar el final de los parámetros y el tipo de retorno, todo después de eso es el cuerpo del cierre en sí. Hay una razón para esto, y lo verás por ti mismo muy pronto.

Allí he cambiado la comparación de < a >, así que obtenemos una clasificación inversa, pero ahora que estamos en una sola línea de código, podemos eliminar la devolución y reducirla a casi nada:

```
let reverseTeam = team.sorted { $0 > $1 }
```

No hay reglas fijas sobre cuándo usar la sintaxis abreviada y cuándo no, pero en caso de que sea útil, uso la sintaxis abreviada a menos que sea cierto alguna de las siguientes condiciones:

- El código del cierre es largo.
- $0 y los amigos se usan más de una vez cada uno.
- Tienes tres o más parámetros (por ejemplo, 2 $, 3 $, etc.).

Si todavía no estás convencido sobre el poder de los cierres, echemos un vistazo a dos ejemplos más.

En primer lugar, la función filter() nos permite ejecutar algo de código en cada elemento de la matriz, y enviará de vuelta una nueva matriz que contiene cada elemento que devuelva true para la función. Por lo tanto, podríamos encontrar a todos los jugadores del equipo cuyo nombre comience con T de esta manera:

```
let tOnly = team.filter { $0.hasPrefix("T") }
print(tOnly)
```

Eso imprimirá ["Tiffany", "Tasha"], porque esos son los únicos dos miembros del equipo cuyo nombre comienza con T.

Y en segundo lugar, la función map() nos permite transformar cada elemento de la matriz utilizando algún código de nuestra elección, y envía de vuelta una nueva matriz de todos los elementos transformados:

```
let uppercaseTeam = team.map { $0.uppercased() }
print(uppercaseTeam)
```

Eso imprimirá ["GLORIA", "SUZANNE", "PIPER", "TIFFANY", "TASHA"], porque ha en mayúsculas todos los nombres y ha producido una nueva matriz a partir del resultado.

Consejo: Cuando se trabaja con map(), el tipo que devuelve no tiene que ser el mismo que el tipo con el que comenzó; podría convertir una matriz de enteros en una matriz de cadenas, por ejemplo.

Como dije, vas a usar mucho los cierres con SwiftUI:

- Cuando cree una lista de datos en la pantalla, SwiftUI le pedirá que proporcione una función que acepte un elemento de la lista y lo convierta como algo que pueda mostrar en la pantalla.
- Cuando cree un botón, SwiftUI le pedirá que proporcione una función para ejecutar cuando se presiona el botón, y otra para generar el contenido del botón: una imagen, o algún texto, y así sucede.
- Incluso simplemente apilar trozos de texto verticalmente se hace con un cierre.

Sí, puedes crear funciones individuales cada vez que SwiftUI haga esto, pero confía en mí: no lo harás. Los cierres hacen que este tipo de código sea completamente natural, y creo que te sorprenderá cómo SwiftUI los usa para producir un código notablemente simple y limpio.

¿Por qué Swift tiene sintaxis de cierre final?

La sintaxis de cierre final está diseñada para hacer que el código Swift sea más fácil de leer, aunque algunos prefieren evitarlo.

Empecemos primero con un ejemplo simple. Aquí hay una función que acepta un doble y luego un cierre lleno de cambios para hacer:

```
func animate(duration: Double, animations: () -> Void) {
    print("Starting a \(duration) second animation…")
    animations()
}
```

(En caso de que te lo estuvieras preguntando, ¡esa es una versión simplificada de una función UIKit real y muy común!)

Podemos llamar a esa función sin un cierre final como este:

```
animate(duration: 3, animations: {
    print("Fade out the image")
})
```

Eso es muy común. Mucha gente no usa cierres finales, y eso está bien. Pero muchos más desarrolladores de Swift miran el }) al final y hacen un poco de vida, no es agradable.

Los cierres de seguimiento nos permiten limpiar eso, a la vez que eliminamos la etiqueta de parámetros de animaciones. Esa misma llamada de función se convierte en esto:

```
animate(duration: 3) {
    print("Fade out the image")
}
```

Los cierres de trailing funcionan mejor cuando su significado está directamente vinculado al nombre de la función: puedes ver lo que está haciendo el cierre porque la función se llama animate().

Si no estás seguro de si usar cierres de trailing o no, mi consejo es que empieces a usarlos en todas partes. Una vez que les hayas dado uno o dos meses, tendrás suficiente uso para mirar hacia atrás y decidir más claramente, ¡pero espero que te acostumbres a ellos porque son muy comunes en Swift!

¿Cuándo deberías usar los nombres de los parámetros abreviados?

Cuando se trabaja con cierres, Swift nos da una sintaxis especial de parámetros abreviados que hace que sea extremadamente conciso escribir cierres. Esta sintaxis numera automáticamente los nombres de los parámetros como $0, $1, $2, y así suces. - no podemos usar nombres como estos en nuestro propio código, por lo que cuando los ves, está inmediatamente claro que se trata de sintaxis abreviada para los cierres.

En cuanto a cuándo deberías usarlos, es realmente un gran "depende":

- ¿Hay muchos parámetros? Si es así, la sintaxis abreviada deja de ser útil y, de hecho, comienza a ser contraproducente: ¿fue $3 o $4 que necesitas comparar con $0? Dales nombres reales y su significado se vuelve más claro.

- ¿Se usa comúnmente la función? A medida que progresen tus habilidades de Swift, comenzarás a darte cuenta de que hay un puñado, tal vez 10 más o menos, de funciones extremadamente comunes que usan cierres, por lo que otros que lean tu código entenderán fácilmente lo que significa $0.

- ¿Los nombres abreviados se usan varias veces en su método? Si necesitas referirte a 0 $ más de tal vez dos o tres veces, probablemente deberías darle un nombre real.

Lo que importa es que tu código sea fácil de leer y de entender. A veces eso significa hacerlo corto y simple, pero no siempre: elija la sintaxis abreviada caso por caso.

Cómo aceptar funciones como parámetros

Hay un último tema relacionado con el cierre que quiero ver, que es cómo escribir funciones que aceptan otras funciones como parámetros. Esto es particularmente importante para los cierres debido a la sintaxis de cierre final, pero es una habilidad útil para tener a pesar de todo.

Anteriormente vimos este código:

```
func greetUser() {
    print("Hi there!")
}

greetUser()

var greetCopy: () -> Void = greetUser
greetCopy()
```

He añadido la anotación de tipo allí intencionalmente, porque eso es exactamente lo que usamos al especificar funciones como parámetros: le dicemos a Swift qué parámetros acepta la función, así como su tipo de retorno.

Una vez más, prepárate: ¡la sintaxis de esto es un poco difícil para los ojos al principio! Aquí hay una función que genera una matriz de enteros repitiendo una función un cierto número de veces:

```
func makeArray(size: Int, using generator: () -> Int) -> [Int] {
    var numbers = [Int]()

    for _ in 0..<size {
        let newNumber = generator()
        numbers.append(newNumber)
    }

    return numbers
}
```

Vamos a desglosar eso...

- La función se llama makeArray(). Toma dos parámetros, uno de los cuales es el número de enteros que queremos, y también devuelve una matriz de enteros.
- El segundo parámetro es una función. Esto no acepta ningún parámetro en sí, pero devolverá un entero cada vez que se llame.
- Dentro de makeArray() creamos una nueva matriz vacía de enteros, luego hacemos un bucle tantas veces como se solicite.
- Cada vez que el bucle da la vuelta, llamamos a la función del generador que se pasó como parámetro. Esto devolverá un nuevo entero, por lo que lo ponemos en la matriz de números.
- Finalmente, se devuelve la matriz terminada.

El cuerpo del makeArray() es en su mayoría sencillo: llame repetidamente a una función para generar un entero, agregando cada valor a una matriz y luego envíelo todo de vuelta.

La parte compleja es la primera línea:

```
func makeArray(size: Int, using generator: () -> Int) -> [Int] {
```

Allí tenemos dos conjuntos de paréntesis y dos conjuntos de tipos de retorno, por lo que puede ser un poco confuso al principio. Si lo divides, deberías poder leerlo de forma lineal:

- Estamos creando una nueva función.
- La función se llama makeArray().
- El primer parámetro es un número entero llamado tamaño.
- El segundo parámetro es una función llamada generador, que a su vez no acepta parámetros y devuelve un entero.
- Todo el asunto - makeArray() - devuelve una matriz de enteros.

El resultado es que ahora podemos hacer matrices enteras de tamaño arbitrario, pasando una función que debería usarse para generar cada número:

```
let rolls = makeArray(size: 50) {
    Int.random(in: 1...20)
}

print(rolls)
```

Y recuerda, esta misma funcionalidad también funciona con funciones dedicadas, por lo que podríamos escribir algo como esto:

```
func generateNumber() -> Int {
    Int.random(in: 1...20)
}

let newRolls = makeArray(size: 50, using: generateNumber)
print(newRolls)
```

Eso llamará a generateNumber() 50 veces para llenar la matriz.

Mientras aprendes Swift y SwiftUI, solo habrá un puñado de veces en las que necesites saber cómo aceptar funciones como parámetros, pero al menos ahora tienes una idea de cómo funciona y por qué es importante.

Hay una última cosa antes de que sigamos adelante: puede hacer que su función acepte múltiples parámetros de función si lo desea, en cuyo caso puede especificar varios cierres finales. La sintaxis aquí es muy común en SwiftUI, por lo que es importante al menos mostrarte una muestra de ella aquí.

Para demostrar esto, aquí hay una función que acepta tres parámetros de función, cada uno de los cuales no acepta parámetros y no devuelve nada:

```
func doImportantWork(first: () -> Void, second: () -> Void, third: () -> V
    print("About to start first work")
    first()
    print("About to start second work")
    second()
    print("About to start third work")
    third()
    print("Done!")
}
```

He añadido llamadas adicionales de print() para simular el trabajo específico que se está haciendo entre la primera, la segunda y la tercera llamada.

Cuando se trata de llamar a eso, el primer cierre final es idéntico a lo que ya hemos usado, pero el segundo y el tercero tienen un formato diferente: terminas la llave del cierre anterior, luego escribes el nombre del parámetro externo y los dos puntos, y luego comienzas otra llave.

Así es como se ve eso:

```
doImportantWork {
    print("This is the first work")
} second: {
    print("This is the second work")
} third: {
    print("This is the third work")
}
```

Tener tres cierres finales no es tan raro como cabría esperar. Por ejemplo, hacer una sección de contenido en SwiftUI se hace con tres cierres finales: uno para el contenido en sí, uno para un encabezado que se pondrá arriba y otro para un pie de página que se pondrá abajo.

¿Por qué querrías usar los cierres como parámetros?

Los cierres de Swift se pueden usar como cualquier otro tipo de datos, lo que significa que puedes pasarlos a funciones, tomar copias de ellos, etc. Pero cuando solo estás aprendiendo, esto puede parecerse mucho a "solo porque puedas, no significa que debas" - es difícil ver el beneficio.

Uno de los mejores ejemplos que puedo dar es la forma en que Siri se integra con las aplicaciones. Siri es un servicio de sistema que se ejecuta desde cualquier lugar de tu dispositivo iOS, pero es capaz de comunicarse con aplicaciones: puedes reservar un viaje con Lyft, puedes comprobar el clima con Carrot Weather, y así suces. Detrás de escena, Siri lanza una pequeña parte de la aplicación en segundo plano para transmitir nuestra solicitud de voz, y luego muestra la respuesta de la aplicación como parte de la interfaz de usuario de Siri.

Ahora piensa en esto: ¿qué pasa si mi aplicación se comporta mal y tarda 10 segundos en responder a Siri? Recuerda, el usuario en realidad no ve mi aplicación, solo Siri, así que desde su perspectiva parece que Siri se ha congelado por completo.

Esta sería una experiencia de usuario terrible, por lo que Apple utiliza cierres en su lugar: inicia nuestra aplicación en segundo plano y pasa un cierre al que podemos llamar cuando hayamos terminado. Nuestra aplicación puede tardar todo el tiempo que quiera en averiguar qué trabajo hay que hacer, y cuando hayamos terminado, llamamos al cierre para devolver los datos a Siri. El uso de un cierre para devolver datos en lugar

de devolver un valor de la función significa que Siri no necesita esperar a que se complete la función, por lo que puede mantener su interfaz de usuario interactiva, no se congelará.

Otro ejemplo común es hacer solicitudes de red. Tu iPhone promedio puede hacer varios miles de millones de cosas por segundo, pero conectarse a un servidor en Japón lleva medio segundo o más, es casi glacial en comparación con la velocidad a la que suceden en tu dispositivo. Por lo tanto, cuando solicitamos datos de Internet, lo hacemos con cierres: "por favor, btenga estos datos y, cuando haya terminado, ejecute este cierre". Una vez más, significa que no forzamos que nuestra interfaz de usuario se congele mientras se está produciendo un trabajo lento.

Resumen: Cierres

Hemos cubierto mucho sobre los cierres en los capítulos anteriores, así que recapitulemos:

Puedes copiar funciones en Swift, y funcionan igual que el original, excepto que pierden sus nombres de parámetros externos.

Todas las funciones tienen tipos, al igual que otros tipos de datos. Esto incluye los parámetros que reciben junto con su tipo de retorno, que podría ser Vacío, también conocido como "nada".

Puedes crear cierres directamente asignando a una constante o variable.

Los cierres que aceptan parámetros o devuelven un valor deben declarar esto dentro de sus llaves, seguido de la palabra clave in.

Las funciones pueden aceptar otras funciones como parámetros. Deben declarar por adelantado exactamente qué datos deben usar esas funciones, y Swift se asegurará de que se sigan las reglas.

En esta situación, en lugar de pasar una función dedicada, también puedes pasar un cierre: puedes hacer uno directamente. Swift permite que ambos enfoques funcionen.

Al pasar un cierre como parámetro de función, no es necesario escribir explícitamente los tipos dentro de su cierre si Swift puede averiguarlo automáticamente. Lo mismo ocurre con el valor de retorno: si Swift puede averiguarlo, no es necesario que lo especifiques.

Si uno o más de los parámetros finales de una función son funciones, puede usar la sintaxis de cierre final.

También puede usar nombres de parámetros abreviados como 0 $ y 1 $, pero recomendaría hacerlo solo bajo algunas condiciones.

Puedes hacer tus propias funciones que acepten funciones como parámetros, aunque en la práctica es mucho más importante saber cómo usarlas que cómo crearlas.

De todas las partes del lenguaje Swift, diría que los cierres son lo más difícil de aprender. La sintaxis no solo es un poco difícil para tus ojos al principio, sino que el concepto mismo de pasar una función a una función toma un poco de tiempo para hundirse.

Entonces, si has leído estos capítulos y sientes que tu cabeza está a punto de explotar, eso es genial, ¡significa que estás a mitad de camino de entender los cierres!

Punto de control 5

Con los cierres en tu haber, es hora de probar un pequeño desafío de codificación usándolos.

Ya has conocido a sorted(), filter(), map(), así que me gustaría que los juntaras en una cadena: llama a uno, luego al otro, luego al otro espalda con espalda sin usar variables temporales.

Su opinión es la siguiente:

```
let luckyNumbers = [7, 4, 38, 21, 16, 15, 12, 33, 31, 49]
```

Tu trabajo es:

1. Filtra cualquier número que sea par
2. Ordenar la matriz en orden ascendente
3. Asignarlos a cadenas en el formato "7 es un número de la suerte"
4. Imprima la matriz resultante, un elemento por línea

Por lo tanto, su resultado debería ser el siguiente:

```
7 is a lucky number
15 is a lucky number
21 is a lucky number
31 is a lucky number
33 is a lucky number
49 is a lucky number
```

Si necesitas pistas, están a continuación, pero honestamente deberías poder abordar esto, ya sea de memoria o haciendo referencia a capítulos recientes de este libro.

¿Sigues aquí? Vale, aquí hay algunas pistas:

- Necesitas usar las funciones filter(), sorted() y map().
- El orden en el que ejecutas las funciones es importante: si primero conviertes la matriz en una cadena, sorted() hará una clasificación de cadena en lugar de una clasificación de enteros. Eso significa que 15 vendrá antes de 7, porque Swift comparará el "1" en "15" con "7".
- Para encadenar estas funciones, use luckyNumbers.first { }.second { }, obviamente poniendo las llamadas a la función real allí.
- Deberías usar isMultiple(of:) para eliminar los números pares.