Swift

Tema 6. Closures

Objetivos

Luego de navegar esta cápsula, deberás ser capaz de hacer lo siguiente:

- comprender qué son las closures en Swift;
- identificar las funciones como closures;
- escribir closures con la sintaxis *shorthand*;
- usar closures como parámetro de funciones;
- pasar una closure como parámetro de función como *trailing closures*;
- capturar y usar datos dentro de las closures; y
- conocer el término escaping en las closures y definirlas como tal.

Las librerías de Swift más importantes están pensadas para trabajar con closures. Un dominio de las closures es necesario para ser desarrollador Swift.

Qué es una closure

Una closure es un bloque de código que puede ser pasado entre funciones o asignado a variables. Muchos otros lenguajes llaman a esta característica *Lambda Expression* o simplemente *Lambda*.

Ejemplo simple de una closure:

```
{ (s1: String, s2: String) -> Bool in

return s1 > s2

}
```

En esta expresión se declara una función anónima (closure) que recibe dos Strings y retorna true si el segundo string es mayor que el primero. Es decir, es un criterio de orden al revés: xyz iría antes que abc.

Podemos asignar la closure anterior a una variable, como se muestra a continuación:

```
let mayorAMenor = { (s1: String, s2: String) -> Bool in return s2 < s1 }
```

mayorAMenor se comporta como una función:

```
mayorAMenor("abc", "xyz") // false

mayorAMenor("xyz", "abc") // true
```

Actividad de repaso:

¿Cuál es la relación que existe entre las librerías de Swift y Closures?

Las librerías Swift trabajan con closures**.**
Dominar closures **es necesario para ser desarrollador Swift.**
Ambos funcionan solamente el uno con el otro.
No existe relación alguna.

Justificación: a lo largo de la lectura, hemos mencionado que las librerías de Swift más importantes están pensadas para trabajar con closures. Un dominio de las closures es necesario para ser desarrollador Swift.

Closure como argumento a una función

En el siguiente ejemplo, usaremos la función sorted(by:) de las librerías estándar de Swift, que se utiliza para ordenar elementos. sorted(by:) recibe una función/closure para establecer el criterio de orden.

Este es un ejemplo de una función (sorted(by:)) que recibe como parámetro otra función/closure (mayorAMenor), la cual recibe 2 Stringy. Retorna true si el primer argumento debe ser ordenado antes que el segundo argumento; en caso contrario, retorna false.

```
// Definimos la Closure y la asignamos a una variable

let mayorAMenor = { (s1: String, s2: String) -> Bool in return s2 < s1 }



// Definimos un array para probar la función sorted by
```

```
let palabras = ["abc", "xyz"]


// Probemos que pasa si a `sorted(by:)` le pasamos mayorAMenor

let palabrasOrdenadasAlReves = palabras.sorted(by: mayorAMenor)

print(palabrasOrdenadasAlReves) // ["xyz", "abc"]
```

Completemos ahora el ejemplo anterior: a continuación, se tiene una función dentro de una función. La función de afuera ordernaAlReves recibe un *array* de palabras y lo retorna ordenado al revés. Para ordenar, usa la función sorted(by:) que acabamos de ver:

```
func ordernaAlReves(arreglo: [String]) -> [String] {


func mayorAMenor(s1: String, s2: String) -> Bool {

return s2 < s1

}


return arreglo.sorted(by: mayorAMenor(s1:s2:))

}


var libros = ["A Sangre Fría","Tarzan", "Cien Años de Soledad"]

ordernaAlReves(arreglo: libros) // ["Tarzan", "Cien Años de Soledad", "A Sangre Fría"]
```

En el primer ejemplo, a sorted(by:) le suministramos una Closure, mientras que en el ejemplo que acabamos de mostrar, le suministramos una función.

Si especificamos una función de orden para sorted(by:), esta debe ser de tipo (Element, Element) -> Bool, donde Element debe corresponder al tipo de valor del arreglo, en nuestro caso, String.
Convirtamos el ejemplo anterior a la sintaxis de closure:

```
func ordernaAlReves(arreglo: [String]) -> [String] {

return arreglo.sorted(by:

{ (s1: String, s2: String) -> Bool in return s2 < s1

return s2 < s1

} )

}
```

sorted antes recibía la función mayorAMenor, la cual fue reemplazada por la siguiente closure:
{ (s1: String, s2: String) -> Bool in return s2 < s1 }.
Como se puede observar, escribimos lo mismo de una forma mucho más compacta y sin perder expresividad.

Asignar una closure a una variable

Swift trata a las closures como un objeto más, por lo que podemos asignarlas a variables. Si seguimos con el ejemplo, obtenemos lo siguiente:

```
func ordernaAlReves(arreglo: [String]) -> [String] {

let mayorAMenor = { (s1: String, s2: String) -> Bool in

return s2 < s1

}

return arreglo.sorted(by: mayorAMenor )

}
```

Acá la closure está dentro de la variable mayorAMenor, la cual es suministrada a la función sorted(by:).

Inferencia de tipos por el contexto

Como la función sorted(by:) “sabe” qué tipo de argumentos recibe y qué tipo de valor retorna, Swift permite omitirlos.

Esto permite una expresión más limpia:

```
func ordernaAlReves(arreglo: [String]) -> [String] {

return arreglo.sorted(by: { s1, s2 in return s2 < s1 } )

}
```

Retorno implícito

Cuando la closure consta de una expresión simple, podemos simplificar aún más omitiendo el return:

```
func ordernaAlReves(arreglo: [String]) -> [String] {

return arreglo.sorted(by: { s1, s2 in s2 < s1 } )

}
```

Nombres de argumentos abreviados

Swift automáticamente introduce los nombres abreviados de los argumentos $0 al primero, $1 al segundo, y así sucesivamente.
Entonces, en el ejemplo anterior, podemos reemplazar s1 por $0 y s2 por $1:

```
func ordernaAlReves(arreglo: [String]) -> [String] {

return arreglo.sorted(by: { $1 < $0 } )

}
```

Como ya no se necesita el nombre específico para cada argumento, se omite el s1, s2 in y se realiza la validación usando únicamente el nombre abreviado con el $.

Trailing closures

Como se explicó en la cápsula de “Funciones”, las funciones pueden recibir como parámetro otras funciones, lo que es equivalente a recibir una closure como parámetro de una función.

Cuando se presenta esta situación, se puede recurrir a las *trailing closures*: una facilidad sintáctica para simplificar la escritura del código. Cuando se suministra una closure a una función, Swift permite sacar afuera de los paréntesis el código de la closure.
Esta función tiene como parámetro una closure de tipo () -> Void:

```
func funcionQueRecibeUnaClosure(closure: () -> Void) {

// ... código de la función

}
```

Normalmente, esta función se invocaría con la sintaxis clásica de esta forma:

```
funcionQueRecibeUnaClosure(closure: {

// ... código del closure

})
```

Utilizando *trailing closure*, se puede incluir la closure luego de los paréntesis de la función y omitir el nombre del parámetro:

```
funcionQueRecibeUnaClosure() {

// ... código de la trailing closure

}
```

Actividad de repaso

Sobre la base del contenido abordado, podemos afirmar que las funciones pueden recibir como parámetro otras funciones. Sin embargo, esto no equivale a recibir una closure como parámetro de una función.

Verdadero.
Falso.

Justificación: vimos durante el desarrollo de este tema que las funciones pueden recibir como parámetro otras funciones, y esto es igual a recibir una closure como parámetro de una función.

Captura de las variables del contexto

Las closures capturan y mantienen las variables del contexto donde fueron definidas.
El siguiente ejemplo muestra cómo la función interna incrementa "captura" las variables total y cant definidas en la función externa crearIncrementador.
Este ejemplo usa una función dentro de una función. Recuerda que al inicio de la cápsula se indicó que una función dentro de otra función es un caso especial de una closure con nombre.

```
func crearIncrementador(incrementaEn cant: Int) -> () -> Int {

var total = 0

func incrementar() -> Int {

total += cant

return total

}

return incrementar

}
```

crearIncrementador no retorna simplemente un valor, sino que retorna una función: una closure que no recibe nada y cada vez que es invocada suma cant a total y lo retorna.
Podemos asignar la closure retornada a una variable e invocarla:

```
let miIncrementador = crearIncrementador(incrementaEn: 10)

miIncrementador() // 10

miIncrementador() // 20

miIncrementador() // 30
```

Es interesante considerar que la función miIncrementador es invocada por fuera de crearIncrementador, y aún así tiene acceso a las variables cant y total. El lenguaje Swift garantiza la sobrevida de ambas variables aún cuando la función ya fue ejecutada. Esta es la clave del mecanismo de captura de las variables del contexto que tiene Swift.

Actividad de repaso

¿Podrías determinar qué tipo de función usa el ejemplo a continuación?

Una función dentro de una función.
Closures.
Trailing closure.
El ejemplo no usa funciones.

```
func crearIncrementador(incrementaEn cant: Int) -> () -> Int {

var total = 0

func incrementar() -> Int {

total += cant

return total

}

return incrementar

}
```

Justificación: este ejemplo usa una función dentro de una función, un caso especial de una closure con nombre.

Ejercitación 1 (en el Playground de Xcode)

max(by:) es una función de Swift para arreglos que retorna el elemento máximo de la secuencia usando la comparación indicada en la closure que recibe como parámetro.

Según esto y dada la siguiente función, lleva a cabo las tareas asignadas a continuación.

```
func obtenerMaximo(arreglo: [String]) -> String? {

func compararMaximo(s1: String, s2: String) -> Bool {
```

```
return s1.count < s2.count

}

return arreglo.max(by: compararMaximo(s1:s2))

}
```

1. Cambia la sintaxis para que `max(by:)` reciba una `closure` con nombre y no una función.

2. Modifica la `closure` del punto anterior para que utilice nombres de argumentos abreviados.

3. Modifica la `closure` del punto anterior para que tenga retorno implícito.

4. Cambia la sintaxis de la closure que recibe `max(by:)` del punto anterior a *trailing closure*.

Map, Reduce y Filter

Estas tres funciones se aplican sobre colecciones y son muy usadas en las diferentes aplicaciones realizadas con Swift. Cada una de ellas recibe como parámetro una `closure` que especifica el comportamiento que debe darse para transformar la colección.

Map

Se utiliza para transformar una colección, iterando sobre cada elemento, realizando una acción y obteniendo como resultado una nueva colección con alguna modificación.

Por ejemplo, supongamos que se tiene un *array* de `Strings` y que se lo quiere transformar para que cada `String` esté todo en mayúsculas. Sin utilizar `map`, obtendríamos lo siguiente:

```
var frutas = ["pera", "manzana","naranja"]

for i in 0..<frutas.endIndex {

frutas[i] = frutas[i].uppercased()

}
```

Con map, obtenemos una expresión más simple:

```
var frutas = ["pera", "manzana","naranja"]

frutas = frutas.map { $0.uppercased() }
```

{ $0.uppercased() } es la closure correspondiente que indica que para cada valor $0 se realice la acción uppercased().

Reduce

Se utiliza para reducir todos los elementos de una función a un único resultado, aplicando una función sobre cada elemento de la colección y acumulando el resultado en cada iteración. Puede usarse, por ejemplo, para hallar una sumatoria total, producto total, concatenación de *strings*, etc.

La función reduce recibe un valor inicial para la acumulación y una *closure* que especifica cómo aumenta dicha acumulación. Siguiendo el ejemplo de obtener la sumatoria total, supongamos que se tiene un arreglo con las notas de un estudiante:

```
var notas = [9, 6, 7, 4]
```

Se obtiene el promedio utilizando reduce, teniendo como valor inicial 0, y especificando que cada valor debe sumarse a la acumulación:

```
var notas = [9, 6, 7, 4]

let total = notas.reduce(0, { $0 + $1 } )

let promedio = total / notas.count

print(promedio) // 6
```

Otro ejemplo puede ser obtener el factorial de 5, recibiendo un rango del 1 al 5 y multiplicando los valores:

```
let cincoFactorial = (1...5).reduce(1, { $0 * $1 })
```

Siempre debe recibirse un valor inicial el cual, en la primera iteración, se encontrará en $0 y el valor de la iteración en $1, de forma que el acumulado se seguirá almacenando y usando en las siguientes iteraciones en $0.

Filter

Filter aplica una función booleana sobre un array y selecciona solo los elementos que cumplan dicha condición.

El siguiente ejemplo usa la expresión { $0 % 2 == 0 } para filtrar los elementos pares. Recordemos que si el resto de un número entre 2 es 0, entonces, el número es par:

```
let primeros100 = 1...100

let pares = primeros100.filter { $0 % 2 == 0 }
```

Escaping closures

Las *escaping closures* son usadas para suministrar el código que se deberá ejecutar al completarse (con éxito o fracaso) una operación asincrónica. Este tipo de código, a veces, se denomina *completion block* o *completion handler*.

Cuando pasamos como argumento una *closure* a una función, decimos que la *closure* escapa a la función cuando es invocada luego de que la función retorna: la *closure* "sobrevive" a la función.

Para declarar una *escaping closure* la sintaxis es anteponer @escaping al tipo de parámetro.
Supongamos que se tiene una función que descarga y procesa un PDF:

```
func descargaPDF(url: URL, alTerminar: @escaping (Bool) -> Void) {

let tarea = URLSession.shared.dataTask(with: url) { data, response, error in

if error == nil {

// ...

// procesar PDF...

// ...

alTerminar(true) // avisa que no hubo errores
```

```
} else {

alTerminar(false) // avisa que no pudo completar la tarea

}

}

tarea.resume() // se dispara asincrónicamente la tarea de descarga del PDF

}
```

descargaPDF recibe como primer parámetro la URL del archivo, como segundo parámetro el *completion handler* que es el código que se ejecutará cuando finalice el procesamiento de PDF. Ese *completion handler* debe ser pasado en forma de *escaping closure*, ya que, como se dijo anteriormente, es un bloque de código que debe ejecutarse cuando descargaPDF retorna.

URLsession es una librería de Swift que se usa para realizar peticiones. A continuación, se explica un poco su funcionamiento, pero no es necesario que se comprenda en su totalidad, lo importante es resaltar el uso del *escaping.*

La función descargaPDF, a su vez, está invocando en forma asincrónica a la descarga mediante la función resume() del objeto dataTask:

```
let tarea = URLSession.shared.dataTask(with: url) {

// ...

// esto, a su vez, también es una escaping closure

// que está siendo pasada a la dataTask(...)

// ... código de proceso

}

tarea.resume() // se dispara asincrónicamente la tarea de descarga del PDF

}
```

Cuando dataTask finalice se ejecutará el código interior entre { }:

```
{ data, response, error in
if error == nil {
// ...
// procesar PDF...
// ...
alTerminar(true) // avisa que no hubo errores
} else {
alTerminar(false) // avisa que no pudo completar la tarea
}
}
}
```

Dicho código invocará la *escaping closure* suministrada: alTerminar.
Veamos un ejemplo de cómo invocar a descargaPDF:

```
let pdfUrl = URL(string: "http://www.orimi.com/pdf-test.pdf")!
descargaPDF(url: pdfUrl) { isOK in
if isOK {
print("ok")
} else {
print("no ok")
}
}
```

Cuando descargaPDF finalice invocará a la *closure* suministrada:

```
{ isOK in

if isOK {

print("ok")

} else {

print("no ok")

}

}
```

Nótese que el valor de ok fue suministrado por descargaPDF.
Si bien es un concepto complejo, en un proyecto real las *escaping closures* son muy usadas.

Ejercitación 2 (en el Playground de Xcode)

1. Dado el siguiente diccionario de códigos Http con sus mensajes:

```
var respuestasHttp = [200: "OK",

403: "Acceso denegado",

404: "No encontrado",

500: "Error interno en el servidor"]
```

Mediante la función map obtén un array que contenga solo los mensajes.

2. Dada la siguiente lista de nombres, obtén solo aquellos que tienen más de 7 letras. ¿Qué función de Swift debes usar?

Enrique, Matías, Franco, Valentina, Federico, Alan, Francisco, Carolina, María, Lucas, Pedro, Juan, Guido.

3. Dada la función descargaXML que descarga en forma asincrónica un XML, el cual luego es enviado al completionHandler, escribe una función que invoque a descargaXML, y mediante la función print imprime dicho XML.

```
func descargaXML(alTerminar completionHandler: @escaping (String) -> Void) {

// ...

// descarga el XML

// al terminar invocará completionHandler enviándole el XML descargado

// ...

}
```

Respuestas (para los evaluadores) – <<No incluir en la cápsula>>

3. Para que la podamos testear en el *playground* completamos la función descargaXML:

```
func descargaXML(alTerminar completionHandler: @escaping (String) -> Void) {

// ...

// descarga el XML

// al terminar invocará completionHandler enviándole el XML descargado

completionHandler("XML")

// ...

}
```

```
descargaXML { xml in

print(xml)

}
```