Funciones, parte 2

Hoy vas a aprender a manejar errores en las funciones. Eso puede sonar terriblemente pesimista, pero como dijo John Lennon, "la vida es lo que sucede cuando estás ocupado haciendo otros planes", ¡nadie quiere problemas, pero la vida tiene el hábito de encontrarlos de todos modos!

Afortunadamente, Swift hace que el manejo de errores sea sencillo y algo infalible: requiere que manejemos los errores, o al menos reconozcamos que podrían ocurrir. Si al menos no intentas manejar bien los errores, tu código simplemente no se compilará.

Hoy tienes dos tutoriales para trabajar, en los que cumplirás con los valores predeterminados para los parámetros y las funciones de lanzamiento, y luego resumiremos las funciones y miraremos el punto de control 4. Una vez que haya completado cada vídeo, hay una breve lectura adicional opcional si está buscando obtener más detalles, y también hay una breve prueba para ayudar a asegurarse de que ha entendido lo que se enseñó.

Cómo proporcionar valores predeterminados para los parámetros

Agregar parámetros a las funciones nos permite agregar puntos de personalización, para que las funciones puedan operar con diferentes datos dependiendo de nuestras necesidades. A veces queremos que estos puntos de personalización estén disponibles para mantener nuestro código flexible, pero otras veces no quieres pensar en ello, quieres lo mismo nueve veces de cada diez.

Por ejemplo, anteriormente vimos esta función:

```
func printTimesTables(for number: Int, end: Int) {
    for i in 1...end {
        print("\(i) x \(number) is \(i * number)")
    }
}

printTimesTables(for: 5, end: 20)
```

Que imprime cualquier tabla de tiempos, comenzando en 1 vez el número hasta cualquier punto final. Ese número siempre va a cambiar en función de la tabla de multiplicar que queramos, pero el punto final parece un gran lugar para proporcionar un valor predeterminado sensato: es posible que queramos contar hasta 10 o 12 la mayor parte del tiempo, sin dejar abierta la posibilidad de ir a un valor diferente algunas veces.

Para resolver este problema, Swift nos permite especificar los valores predeterminados para cualquiera o todos nuestros parámetros. En este caso, podríamos establecer que el final tenga el valor predeterminado de 12, lo que significa que si no lo especificamos, 12 se utilizará automáticamente.

Así es como se ve en el código:

```
func printTimesTables(for number: Int, end: Int = 12) {
    for i in 1...end {
        print("\(i) x \(number) is \(i * number)")
    }
}

printTimesTables(for: 5, end: 20)
printTimesTables(for: 8)
```

Observe cómo ahora podemos llamar a printTimesTables() de dos maneras diferentes: podemos proporcionar ambos parámetros para las veces que lo queremos, pero si no lo hacemos, si solo escribimos printTimesTables (para: 8), entonces el valor predeterminado de 12 se utilizará para el final.

De hecho, hemos visto un parámetro predeterminado en acción, en algún código que usamos antes:

```
var characters = ["Lana", "Pam", "Ray", "Sterling"]
print(characters.count)
characters.removeAll()
print(characters.count)
```

Eso añade algunas cadenas a una matriz, imprime su recuento, luego las elimina todas e imprime el recuento de nuevo.

Como optimización del rendimiento, Swift da a las matrices suficiente memoria para guardar sus artículos, además de un poco más para que puedan crecer un poco con el tiempo. Si se añaden más elementos a la matriz, Swift asigna más y más memoria automáticamente, de modo que se desperdicia lo menos posible.

Cuando llamamos a removeAll(), Swift eliminará automáticamente todos los elementos de la matriz y luego liberará toda la memoria que se asignó a la matriz. Por lo general, eso es lo que querrás, porque después de todo, estás quitando los objetos por una razón. Pero a veces, solo a veces, es posible que estés a punto de agregar muchos elementos nuevos de nuevo a la matriz, por lo que hay una segunda forma de esta función que elimina los elementos mientras mantiene la capacidad anterior:

```
characters.removeAll(keepingCapacity: true)
```

Esto se logra utilizando un valor de parámetro predeterminado: keepingCapacity es un booleano con el valor predeterminado de false para que haga lo sensato de forma predeterminada, al tiempo que deja abierta la opción de que pasemos en true durante las veces que queremos mantener la capacidad existente de la matriz.

Como puede ver, los valores de los parámetros predeterminados nos permiten mantener la flexibilidad en nuestras funciones sin hacer que sean molestos para llamar la mayor parte del tiempo; solo necesita enviar algunos parámetros cuando necesita algo inusual.

Cuándo usar los parámetros predeterminados para las funciones

Los parámetros predeterminados nos permiten hacer que las funciones sean más fáciles de llamar al permitirnos proporcionar valores predeterminados comunes para los parámetros. Por lo tanto, cuando queremos llamar a esa función usando esos valores predeterminados, podemos ignorar los parámetros por completo, como si no existieran, y nuestra función hará lo correcto. Por supuesto, cuando queremos algo personalizado, está ahí para que lo cambiemos.

Los desarrolladores de Swift utilizan los parámetros predeterminados con mucha frecuencia, porque nos permiten centrarnos en las partes importantes que necesitan cambiar regularmente. Esto realmente puede ayudar a simplificar funciones complejas y hacer que su código sea más fácil de escribir.

Por ejemplo, imagina algún código de búsqueda de rutas como este:

```
func findDirections(from: String, to: String, route: String = "fastest", a
    // code here
}
```

Eso supone que la mayoría de las veces la gente quiere conducir entre dos lugares por la ruta más rápida, sin evitar las autopistas, valores predeterminados sensatos que probablemente funcionen la mayor parte del tiempo, al tiempo que nos da el margen para proporcionar valores personalizados cuando sea necesario.

Como resultado, podemos llamar a esa misma función de cualquiera de tres maneras:

```
findDirections(from: "London", to: "Glasgow")
findDirections(from: "London", to: "Glasgow", route: "scenic")
findDirections(from: "London", to: "Glasgow", route: "scenic", avoidHighwa
```

Código más corto y simple la mayor parte del tiempo, pero flexible cuando lo necesitamos, perfecto.

Cómo manejar errores en las funciones

Las cosas van mal todo el tiempo, como cuando el archivo que querías leer no existe, o cuando los datos que intentaste descargar fallaron porque la red estaba inactivo. Si no manejáramos los errores con gracia, nuestro código se bloquearía, por lo que Swift nos hace manejar los errores, o al menos reconocer cuándo podrían ocurrir.

Esto tiene tres pasos:

- Contarle a Swift sobre los posibles errores que pueden ocurrir.
- Escribir una función que pueda marcar los errores si ocurren.
- Llamar a esa función y manejar cualquier error que pueda ocurrir.

Hagamos un ejemplo completo: si el usuario nos pide que comprobemos qué tan fuerte es su contraseña, marcaremos un error grave si la contraseña es demasiado corta o es obvia.

Por lo tanto, tenemos que empezar por definir los posibles errores que podrían ocurrir. Esto significa hacer una nueva enumeración que se base en el tipo de error existente de Swift, como esta:

```
enum PasswordError: Error {
    case short, obvious
}
```

Eso dice que hay dos posibles errores con la contraseña: corto y obvio. No define lo que significan, solo que existen.

El segundo paso es escribir una función que desencadene uno de esos errores. Cuando se activa un error, o se lanza en Swift, estamos diciendo que algo fatal salió mal con la función, y en lugar de continuar como de costumbre, termina inmediatamente sin devolver ningún valor.

En nuestro caso, vamos a escribir una función que compruebe la fuerza de una contraseña: si es realmente mala, menos de 5 caracteres o es extremadamente conocida, entonces lanzaremos un error de inmediato, pero para todas las demás cadenas devolveremos las calificaciones de "OK", "Bueno" o "Excelente".

Así es como se ve en Swift:

```
func checkPassword(_ password: String) throws -> String {
    if password.count < 5 {
        throw PasswordError.short
    }

    if password == "12345" {
        throw PasswordError.obvious
    }

    if password.count < 8 {
        return "OK"
    } else if password.count < 10 {
        return "Good"
    } else {
        return "Excellent"
    }
}
```

Vamos a desglosar eso...

Si una función es capaz de lanzar errores sin manejarlos por sí misma, debe marcar la función como lanzamientos antes del tipo de retorno.

No especificamos exactamente qué tipo de error lanza la función, solo que puede generar errores.

Estar marcado con lanzamientos no significa que la función deba lanzar errores, solo que podría hacerlo.

Cuando llega el momento de lanzar un error, escribimos lanzamiento seguido de uno de nuestros casos de PasswordError. Esto sale inmediatamente de la función, lo que significa que no devolverá una cadena.

Si no se lanzan errores, la función debe comportarse como de costumbre: debe devolver una cadena.

Eso completa el segundo paso de lanzar errores: definimos los errores que podrían ocurrir, luego escribimos una función usando esos errores.

El paso final es ejecutar la función y manejar cualquier error que pueda ocurrir. Los Swift Playgrounds son bastante laxos con el manejo de errores porque están destinados principalmente al aprendizaje, pero cuando se trata de trabajar con proyectos reales de Swift, encontrarás que hay tres pasos:

Iniciando un bloque de trabajo que podría lanzar errores, usando do.

Llamar a una o más funciones de lanzamiento, usando try.

Manejar cualquier error lanzado usando catch.

En pseudocódigo, se ve así:

```
do {
    try someRiskyWork()
} catch {
    print("Handle errors here")
}
```

Si quisiéramos escribir, prueba eso usando nuestra función actual checkPassword(), podríamos escribir esto:

```
let string = "12345"

do {
    let result = try checkPassword(string)
    print("Password rating: \(result)")
} catch {
    print("There was an error.")
}
```

Si la función checkPassword() funciona correctamente, devolverá un valor en el resultado, que luego se imprimirá. Pero si se produce algún error (que en este caso habrá), el mensaje de clasificación de la contraseña nunca se imprimirá; la ejecución saltará inmediatamente al bloque de captura.

Hay algunas partes diferentes de ese código que merecen discusión, pero quiero centrarme en la más importante: intentarlo. Esto debe escribirse antes de llamar a todas las funciones que puedan generar errores, y es una señal visual para los desarrolladores de que la ejecución regular del código se interrumpirá si se produce un error.

Cuando usas try, necesitas estar dentro de un bloque de do, y asegurarte de tener uno o más bloques de captura capaces de manejar cualquier error. ¡En algunas circunstancias puedes usar una alternativa escrita como prueba! Que no requiere hacer y atrapar, pero bloqueará su código si se lanza un error - debe usar esto rara vez, y solo si está absolutamente seguro de que no se puede lanzar un error.

Cuando se trata de detectar errores, siempre debe tener un bloque de captura que sea capaz de manejar todo tipo de errores. Sin embargo, también puedes detectar errores específicos, si quieres:

```
let string = "12345"

do {
    let result = try checkPassword(string)
    print("Password rating: \(result)")
} catch {
    print("There was an error.")
}
```

A medida que progreses, verás cómo las funciones de lanzamiento se incorporan a muchos de los propios marcos de Apple, por lo que, aunque es posible que no las crees mucho tú mismo, al menos necesitarás saber cómo usarlas de forma segura.

Consejo: La mayoría de los errores lanzados por Apple proporcionan un mensaje significativo que puede presentar a su usuario si es necesario. Swift hace que esto esté disponible utilizando un valor de error que se proporciona automáticamente dentro de su bloque de captura, y es común leer error.localizedDescription para ver exactamente lo que sucedió.

¿Cuándo deberías escribir funciones de lanzamiento?

Las funciones de lanzamiento en Swift son aquellas que son capaces de encontrar errores que no pueden o no quieren manejar. Eso no significa que lancen errores, solo que es posible que puedan. Como resultado, Swift se asegurará de que tengamos cuidado cuando los usemos, para que cualquier error que ocurra sea atendido.

Pero cuando escribas tu código, lo más probable es que pienses para ti mismo "¿debería esta función lanzar algún error que encuentre, o tal vez debería manejarlos por sí misma?" Esto es muy común, y para ser honesto, no hay una sola respuesta: puede manejar los errores dentro de la función (por lo que no es una función de lanzamiento), puede enviarlos todos de vuelta a lo que sea que se llame la función (llamada "propación de errores" o a veces "errores de burbuja"), e incluso puede manejar algunos errores en la función y enviar algunos de vuelta. Todas esas son soluciones válidas, y las usarás todas en algún momento.

Cuando acabas de empezar, te recomiendo que evites lanzar funciones la mayor parte del tiempo. Al principio pueden sentirse un poco torpes porque necesitas asegurarte de que todos los errores se manejen dondequiera que uses la función, por lo que se siente casi un poco "infeccioso" - de repente tienes errores que necesitan ser manejados en varios lugares de tu código, y si esos errores burbujean más, entonces la "infección" simplemente se propaga.

Por lo tanto, cuando estés aprendiendo, empieza poco a poco: mantén el número de funciones de lanzamiento bajo y trabaja hacia a partir de ahí. Con el tiempo, obtendrá un mejor control sobre la gestión de errores para mantener el flujo de su programa sin problemas, y se sentirá más seguro a la hora de agregar funciones de lanzamiento.

Para obtener una perspectiva diferente sobre las funciones de lanzamiento, consulte esta entrada de blog de Donny Wals: https://www.donnywals.com/working-with-throwing-functions-in-swift/

¿Por qué Swift nos hace probar antes de cada función de lanzamiento?

El uso de las funciones de lanzamiento de Swift se basa en tres palabras clave únicas: hacer, probar y atrapar. Necesitamos los tres para poder llamar a una función de lanzamiento, lo cual es inusual: la mayoría de los otros lenguajes usan solo dos, porque no necesitan escribir intentos antes de cada función de lanzamiento.

La razón por la que Swift es diferente es bastante simple: al obligarnos a probar antes de cada función de lanzamiento, estamos reconociendo explícitamente qué partes de nuestro código pueden causar errores. Esto es particularmente útil si tienes varias funciones de lanzamiento en un solo bloque de do, como esta:

```
let string = "12345"

do {
    let result = try checkPassword(string)
    print("Password rating: \(result)")
} catch PasswordError.short {
    print("Please use a longer password.")
} catch PasswordError.obvious {
    print("I have the same combination on my luggage!")
} catch {
    print("There was an error.")
}
```

errores, pero la segunda y la cuarta no pueden.

Resumen: Funciones

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

- Las funciones nos permiten reutilizar el código fácilmente tallando trozos de código y dándole un nombre.
- Todas las funciones comienzan con la palabra func, seguida del nombre de la función. El cuerpo de la función está contenido dentro de las llaves de apertura y cierre.
- Podemos agregar parámetros para que nuestras funciones sean más flexibles: enumerarlos uno por uno separados por comas: el nombre del parámetro, luego los dos puntos y luego el tipo del parámetro.
- Puede controlar cómo se utilizan esos nombres de parámetros externamente, ya sea utilizando un nombre de parámetro externo personalizado o usando un guión bajo para desactivar el nombre externo de ese parámetro.
- Si crees que hay ciertos valores de parámetros que usarás repetidamente, puedes hacer que tengan un valor predeterminado para que tu función tome menos código para escribir y haga lo inteligente de forma predeterminada.
- Las funciones pueden devolver un valor si lo desea, pero si desea devolver varios datos de una función, debe usar una tupla. Estos contienen varios elementos con nombre, pero está limitado de una manera en que un diccionario no lo es: enumeras cada elemento específicamente, junto con su tipo.
- Las funciones pueden lanzar errores: crea una enumeración que define los errores que desea que ocurran, arroja esos errores dentro de la función según sea necesario, luego usa do, try y catch para manejarlos en el sitio de la llamada.

Punto de control 4

Con las funciones en tu haber, es hora de probar un pequeño desafío de codificación. No te preocupes, no es tan difícil, pero puede que te lleve un tiempo pensar y llegar a algo. Como siempre, te daré algunas pistas si las necesitas.

- El desafío es el siguiente: escribir una función que acepte un entero del 1 al 10.000, y devuelva la raíz cuadrada entera de ese número. Eso suena fácil, pero hay algunas trampas:
- No puedes usar la función sqrt() integrada de Swift o similar; necesitas encontrar la raíz cuadrada tú mismo.
- Si el número es menor de 1 o superior a 10.000, deberías lanzar un error de "fuera de los límites".
- Solo debes considerar las raíces cuadradas enteras; no te preocupes por que la raíz cuadrada de 3 sea 1,732, por ejemplo.
- Si no puedes encontrar la raíz cuadrada, lanza un error de "sin raíz".

Como recordatorio, si tienes el número X, la raíz cuadrada de X será otro número que, cuando se multiplica por sí mismo, da X. Por lo tanto, la raíz cuadrada de 9 es 3, porque 3x3 es 9, y la raíz cuadrada de 25 es 5, porque 5x5 es 25.

Te daré algunas pistas en un momento, pero como siempre, te animo a que lo pruebes tú mismo primero: luchar por recordar cómo funcionan las cosas, y a menudo tener que buscarlas de nuevo, es una forma poderosa de progresar.

¿Sigues aquí? Vale, aquí hay algunas pistas:

- Este es un problema que deberías "fuerza bruta": crear un bucle con multiplicaciones en el interior, buscando el entero en el que te pasaron.
- La raíz cuadrada de 10.000, el número más grande que quiero que manejes, es 100, por lo que tu bucle debería detenerse allí.
- Si llegas al final de tu bucle sin encontrar una coincidencia, lanza el error "sin raíz".
- Puedes definir diferentes errores fuera de los límites para "menos de 1" y "mayor de 10.000" si quieres, pero no es realmente necesario, solo tener uno está bien.