Condiciones

Si hay una línea del Hamlet de Shakespeare que la mayoría de la gente conoce, es "ser o no ser, esa es la pregunta". Shakespeare quiso decir eso como la última cuestión de la vida o la muerte, pero sucede que Hamlet golpea justo en el corazón de la lógica en la programación: evaluar si una condición es verdadera o no.

Hoy vamos a entrar en el detalle real de Swift: operadores y condiciones, que nos permiten evaluar el estado de nuestro programa a medida que se ejecuta y tomar diferentes medidas dependiendo del resultado. Hay varias formas de hacerlo y las necesitarás todas, pero las revisaremos paso a paso para que puedas ver cómo se comparan.

Hoy tienes cuatro tutoriales para trabajar, donde te encontrarás con cosas como si, cambiar y más. Una vez que hayas visto cada vídeo y, opcionalmente, hayas pasado por la lectura adicional, hay una breve prueba para asegurarte de que has entendido lo que se enseñó.

1. Cómo comprobar si una condición es verdadera o falsa

Los programas muy a menudo toman decisiones:

- Si la puntuación del examen del estudiante fue de más de 80, imprima un mensaje de éxito.
- Si el usuario ha introducido un nombre que viene después del nombre de su amigo en orden alfabético, ponga el nombre del amigo primero.
- Si añadir un número a una matriz hace que contenga más de 3 elementos, elimine el más antiguo.
- Si se le pidió al usuario que introdujera su nombre y no escribió nada en absoluto, déle un nombre predeterminado de "Anónimo".

Swift los maneja con declaraciones if, que nos permiten comprobar una condición y ejecutar algo de código si la condición es verdadera. Se ven así:

```
if someCondition {
    print("Do something")
}
```

Vamos a desglosar eso:

La condición comienza con **if**, que indica a Swift que queremos comprobar algún tipo de condición en nuestro código.

La parte de **someCondition** es donde escribes tu condición: ¿la puntuación superó los 80? ¿La matriz contiene más de 3 elementos?

Si la condición es cierta, si la puntuación realmente es de más de 80, entonces imprimimos el mensaje "Haz algo".

Por supuesto, eso no es todo en el código: no mencioné los pequeños símbolos { y }. Estos se llaman corsés, brackets de apertura y cierre, más específicamente, aunque a veces los escucharás referidos como "corséntes rizados" o "corchetes rizados".

Estas llaves se utilizan ampliamente en Swift para marcar bloques de código: la llave de apertura inicia el bloque y la llave de cierre lo termina. Dentro del bloque de código está todo el código que queremos ejecutar si nuestra condición es verdadera cuando se comprueba, que en nuestro caso es imprimir un mensaje.

Puedes incluir tanto código como quieras:

```
if someCondition {
    print("Do something")
    print("Do something else")
    print("Do a third thing")
}
```

Por supuesto, lo que realmente importa es la parte de alguna condición, porque ahí es donde entra tu código de verificación: ¿qué condición realmente quieres comprobar?

Bueno, vamos a probar el ejemplo de puntuación: si una constante de puntuación es más de 80, vamos a imprimir un mensaje. Así es como se vería en el código:

```
let score = 85

if score > 80 {
    print("Great job!")
}
```

En ese código, la puntuación > 80 es nuestra condición. Recordarás > de la escuela que significa "es mayor que", por lo que nuestra condición completa es "si la puntuación es mayor que 80". Y si es superior a 80, "¡Buen trabajo!" Se imprimirá, ¡genial!

Ese símbolo > es un operador de comparación, porque compara dos cosas y devuelve un resultado booleano: ¿es la cosa de la izquierda mayor que la de la derecha? También puedes usar < para menos de, >= para "mayor que o igual" y <= para "menor o igual".

Vamos a probarlo, ¿qué crees que imprimirá este código?

```
let speed = 88
let percentage = 85
let age = 18

if speed >= 88 {
    print("Where we're going we don't need roads.")
}

if percentage < 85 {
    print("Sorry, you failed the test.")
}

if age >= 18 {
    print("You're eligible to vote")
}
```

Intenta ejecutar el código mentalmente en tu cabeza: ¿qué líneas de **print()** se ejecutarán realmente?

Bueno, el primero se ejecutará si la velocidad es mayor o igual a 88, y como es exactamente 88, se ejecutará el primer código **print()**.

El segundo se ejecutará si el **percentage** es inferior a 85, y como es exactamente 85, el segundo **print()** no se ejecutará: usamos menos de, no menos o igual.

El tercero se ejecutará si la **age** es mayor o igual a 18, y como es exactamente 18, se ejecutará el tercer **print()**.

Ahora vamos a probar nuestra segunda condición de ejemplo: si el usuario ha introducido un nombre que viene después del nombre de su amigo alfabéticamente, ponga el nombre del amigo primero. Has visto cómo **<, >=** y otros funcionan muy bien con los números, pero también funcionan igual de bien con las cadenas desde el caja:

```
let ourName = "Dave Lister"
let friendName = "Arnold Rimmer"

if ourName < friendName {
    print("It's \(ourName) vs \(friendName)")
}

if ourName > friendName {
    print("It's \(friendName) vs \(ourName)")
}
```

Por lo tanto, si la cadena dentro de **ourName** viene antes de la cadena dentro de **friendName** cuando se ordena alfabéticamente, imprime **ourName** primero y luego **friendName**, exactamente como queríamos.
Echemos un vistazo a nuestra tercera condición de ejemplo: si agregar un número a una matriz hace que contenga más de 3 elementos, elimine el más antiguo. Ya has conocido **append()**, **count** y **remove(at:)**, así que ahora podemos juntar los tres con una condición como esta:

```
// Make an array of 3 numbers
var numbers = [1, 2, 3]

// Add a 4th
numbers.append(4)

// If we have over 3 items
if numbers.count > 3 {
    // Remove the oldest number
    numbers.remove(at: 0)
}

// Display the result
print(numbers)
```

Ahora echemos un vistazo a nuestra cuarta condición de ejemplo: si se le pidió al usuario que ingresara su nombre y no escribió nada en absoluto, déle un nombre predeterminado de "Anónimo".

Para resolver esto, primero tendrás que conocer a otros dos operadores de comparación que usarás mucho, los cuales manejan la igualdad. El primero es `==` y significa "es igual a", que se usa así:

```
let country = "Canada"

if country == "Australia" {
    print("G'day!")
}
```

¡El segundo es `!=`, que significa "no es igual a", y se usa así:

```
let name = "Taylor Swift"

if name != "Anonymous" {
    print("Welcome, \(name)")
}
```

En nuestro caso, queremos comprobar si el nombre de usuario introducido por el usuario está vacío, lo que podríamos hacer así:

```
// Create the username variable
var username = "taylorswift13"

// If `username` contains an empty string
if username == "" {
    // Make it equal to "Anonymous"
    username = "Anonymous"
}

// Now print a welcome message
print("Welcome, \(username)!")
```

Esa "" es una cadena vacía: comenzamos la cadena y terminamos la cadena, sin nada en el medio. Al comparar el nombre de usuario con eso, estamos comprobando si el usuario también ingresó una cadena vacía para su nombre de usuario, que es exactamente lo que queremos.

Ahora, hay otras formas de hacer esta comprobación, y es importante que entiendas lo que hacen.

En primer lugar, podríamos comparar el recuento de la cadena, cuántas letras tiene, con 0, así:

```
if username.count == 0 {
    username = "Anonymous"
}
```

Comparar una cadena con otra no es muy rápido en ningún idioma, por lo que hemos reemplazado la comparación de cadenas por una comparación de enteros: ¿el número de letras de la cadena es igual a 0?

En muchos idiomas eso es muy rápido, pero no en Swift. Verás, Swift es compatible con todo tipo de cadenas complejas: literalmente, todos los lenguajes humanos funcionan fuera de la caja, incluidos los emojis, y eso no es cierto en muchos otros lenguajes de programación. Sin embargo, este gran soporte tiene un costo, y una parte de ese costo es que pedir una cadena para su recuento hace que Swift revise y cuente todas las letras una por una, no solo almacena su longitud por separado de la cadena.

Por lo tanto, piensa en la situación en la que tienes una cuerda masiva que almacena las obras completas de Shakespeare. Nuestro pequeño cheque para el recuento == 0 tiene que pasar y contar todas las letras de la cadena, a pesar de que tan pronto como hemos contado al menos un carácter sabemos la respuesta a nuestra pregunta.

Como resultado, Swift añade una segunda pieza de funcionalidad a todas sus cadenas, matrices, diccionarios y conjuntos: isEmpty. Esto devolverá la verdad si lo que estás comprobando no tiene nada dentro, y podemos usarlo para arreglar nuestra condición de esta manera:

```
if username.isEmpty == true {
    username = "Anonymous"
}
```

Eso es mejor, pero podemos ir un paso más allá. Verás, en última instancia, lo que importa es que tu condición debe reducirse a verdadera o falsa; Swift no permitirá nada más. En nuestro caso, username.isEmpty ya es un booleano, lo que significa que será verdadero o falso, por lo que podemos hacer que nuestro código sea aún más simple:

```
if username.isEmpty {
    username = "Anonymous"
}
```

Si isEmpty es cierto, la condición pasa y el nombre de usuario se establece en Anónimo, de lo contrario, la condición falla.

¿Cómo nos permite Swift comparar muchos tipos de datos?

Swift nos permite comparar muchos tipos de valores fuera de la caja, lo que significa que podemos comprobar una variedad de valores para la igualdad y la comparación. Por ejemplo, si tuviéramos valores como estos:

```
let firstName = "Paul"
let secondName = "Sophie"

let firstAge = 40
let secondAge = 10
```

Entonces podríamos compararlos de varias maneras:

```
print(firstName == secondName)
print(firstName != secondName)
print(firstName < secondName)
print(firstName >= secondName)

print(firstAge == secondAge)
print(firstAge != secondAge)
print(firstAge < secondAge)
print(firstAge >= secondAge)
```

Detrás de escena, Swift implementa esto de una manera notablemente inteligente que en realidad le permite comparar una amplia variedad de cosas. Por ejemplo, Swift tiene un tipo especial para almacenar fechas llamado **Date**, y puede comparar fechas usando los mismos operadores: **someDate < someOtherDate**, por ejemplo.

Incluso podemos pedirle a Swift que haga que nuestras enumeraciones sean comparables, así:

```
enum Sizes: Comparable {
    case small
    case medium
    case large
}

let first = Sizes.small
let second = Sizes.large
print(first < second)
```

Eso imprimirá "true", porque **small** viene antes que **large** en la lista de casos de enumeración.

2. Cómo comprobar múltiples condiciones

Cuando usamos si debemos proporcionar a Swift algún tipo de condición que será verdadera o falsa una vez que haya sido evaluada. Si quieres comprobar si hay varios valores diferentes, puedes colocarlos uno tras otro de esta manera:

```
let age = 16

if age >= 18 {
    print("You can vote in the next election.")
}

if age < 18 {
    print("Sorry, you're too young to vote.")
}
```

Sin embargo, eso no es muy eficiente si lo piensas: nuestras dos condiciones son mutuamente excluyentes, porque si alguien es mayor o igual a 18 (la primera condición), entonces no puede ser menor que 18 (la segunda condición), y lo contrario también es cierto. Estamos haciendo que Swift haga un trabajo que simplemente no es necesario.

En esta situación, Swift nos proporciona una condición más avanzada que nos permite agregar un bloque else a nuestro código, algo de código para ejecutar si la condición no es verdadera.

Usando otra cosa, podríamos reescribir nuestro código anterior a esto:

```
let age = 16

if age >= 18 {
    print("You can vote in the next election.")
} else {
    print("Sorry, you're too young to vote.")
}
```

Ahora Swift solo necesita comprobar la edad una vez: si es mayor o igual a 18, se ejecuta el primer código print(), pero si es un valor menor que 18, se ejecuta el segundo código print().
Entonces, ahora nuestra condición se ve así:

```
if someCondition {
    print("This will run if the condition is true")
} else {
    print("This will run if the condition is false")
}
```

Hay una condición aún más avanzada llamada else if, que te permite ejecutar una nueva comprobación si la primera falla. Puedes tener solo uno de estos si quieres, o tener varios más si, e incluso combinar otro si con otro si es necesario. Sin embargo, solo puedes tener una más, porque eso significa "si todas las demás condiciones han sido falsas".

Así es como se ve eso:

```
let a = false
let b = true

if a {
    print("Code to run if a is true")
} else if b {
    print("Code to run if a is false but b is true")
} else {
    print("Code to run if both a and b are false")
}
```

Puedes seguir añadiendo más y más condiciones si quieres, ¡pero ten cuidado de que tu código no se complique demasiado!

Además de usar else y else si para hacer condiciones más avanzadas, también puedes comprobar más de una cosa. Por ejemplo, podríamos querer decir "si la temperatura de hoy es superior a 20 grados centígrados pero inferior a 30, imprima un mensaje".

```
let temp = 25

if temp > 20 {
    if temp < 30 {
        print("It's a nice day.")
    }
}
```

Aunque eso funciona lo suficientemente bien, Swift proporciona una alternativa más corta: podemos usar && para combinar dos condiciones juntas, y toda la condición solo será verdadera si las dos partes dentro de la condición son ciertas.

Por lo tanto, podríamos cambiar nuestro código a esto:

```
if temp > 20 && temp < 30 {
    print("It's a nice day.")
}
```

Deberías leer && como "y", por lo que todas nuestras condiciones dicen "si la temperatura es superior a 20 y la temperatura es inferior a 30, imprime un mensaje". Se llama operador lógico porque combina booleanos para hacer un nuevo booleano.

&& Tiene una contraparte que es de dos símbolos de tubería, ||, que significa "o". Mientras que && solo hará que una condición sea verdadera si ambas subcondiciones son verdaderas, || hará que una condición sea verdadera si cualquiera de las subcondiciones es verdadera.

Por ejemplo, podríamos decir que un usuario puede comprar un juego si tiene al menos 18 años, o si es menor de 18 años, debe tener permiso de un padre. Podríamos escribir eso usando || así:

```
let userAge = 14
let hasParentalConsent = true

if userAge >= 18 || hasParentalConsent == true {
    print("You can buy the game")
}
```

Eso imprimirá "Puedes comprar el juego", porque aunque la primera mitad de nuestra condición falla, el usuario no tiene al menos 18 años, la segunda mitad pasa, porque tienen el consentimiento de los padres.

Recuerde, el uso de == true en una condición se puede eliminar, porque obviamente ya estamos comprobando un booleano. Entonces, podríamos escribir esto en su lugar:

```
if userAge >= 18 || hasParentalConsent {
    print("You can buy the game")
}
```

Para terminar con la comprobación de múltiples condiciones, probemos un ejemplo más complejo que combina if, else if, else y || todo al mismo tiempo, e incluso muestra cómo las enumeraciones encajan en las condiciones.

En este ejemplo vamos a crear una enumeración llamada TransportOption, que contiene cinco casos: avión, helicóptero, bicicleta, coche y scooter. A continuación, asignaremos un valor de ejemplo a una constante y ejecutaremos algunas comprobaciones:

Si vamos a algún lugar en avión o en helicóptero, imprimiremos "¡Vamos a volar!"

Si vamos en bicicleta, imprimiremos "Espero que haya un carril bici..."

Si vamos en coche, imprimiremos "Es hora de quedarnos atascados en el tráfico".

De lo contrario, imprimiremos "¡Voy a alquilar un scooter ahora!"

Aquí está el código para eso:

```
enum TransportOption {
    case airplane, helicopter, bicycle, car, scooter
}

let transport = TransportOption.airplane

if transport == .airplane || transport == .helicopter {
    print("Let's fly!")
} else if transport == .bicycle {
    print("I hope there's a bike path…")
} else if transport == .car {
    print("Time to get stuck in traffic.")
} else {
    print("I'm going to hire a scooter now!")
}
```

Me gustaría elegir algunas partes de ese código:

Cuando establecemos el valor para el transporte, tenemos que ser explícitos de que nos referimos a TransportOption.airplane. No podemos simplemente escribir .airplane porque Swift no entiende que nos retendemos a la enumeración de TransportOption.

Una vez que eso ha sucedido, ya no necesitamos escribir TransportOption porque Swift sabe que el transporte debe ser algún tipo de TransportOption. Por lo tanto, podemos comprobar si es igual a .airplane en lugar de TransportOption.airplane.

El código que usa || para comprobar si el transporte es igual a .airplane o igual a .helicopter, y si alguno de ellos es verdadero, entonces la condición es verdadera, y "¡Vamos a volar!" Está impreso.

Si la primera condición falla, si el modo de transporte no es .airplane o .helicopter, entonces la segunda condición se ejecuta: ¿el modo de transporte es .bicicleta? Si es así, "Espero que haya un carril bici..." está impreso.

Si tampoco vamos en bicicleta, entonces comprobamos si vamos en coche. Si es así, se imprime "Es hora de quedarse atascado en el tráfico".

Finalmente, si todas las condiciones anteriores fallan, entonces se ejecuta el bloque else, lo que significa que vamos en scooter.

¿Cuál es la diferencia entre si y si?

Cuando solo estás aprendiendo Swift, puede ser un poco difícil saber cuándo usar otra cosa, cuándo usar otra cosa si y cuál es realmente la diferencia.

Bueno, comencemos con un valor de ejemplo con el que podemos trabajar:

```
let score = 9001
```

(En caso de que te lo preguntes, sí, esto significa que confiaremos en el meme de Dragonball Z).

Podríamos escribir una condición simple para comprobar si la puntuación es superior a 9000 de esta manera:

```
if score > 9000 {
    print("It's over 9000!")
}
```

Ahora, si queremos imprimir un mensaje diferente para puntuaciones iguales o inferiores a 9000, podríamos escribir esto:

```
if score > 9000 {
    print("It's over 9000!")
}

if score <= 9000 {
    print("It's not over 9000!")
}
```

Eso funciona perfectamente, y tu código haría exactamente lo que esperas. Pero ahora le hemos dado a Swift más trabajo que hacer: necesita comprobar el valor de la puntuación dos veces. Eso es muy rápido aquí con un número entero simple, pero si nuestros datos fueran más complejos, serían más lentos.

Aquí es donde entra otra cosa, porque significa "si la condición que comprobamos no era verdadera, ejecute este código en su lugar".

Por lo tanto, podríamos reescribir nuestro código anterior en esto:

```
if score > 9000 {
    print("It's over 9000!")
} else {
    print("It's not over 9000!")
}
```

Con ese cambio, Swift solo comprobará la puntuación una vez, además de que nuestro código también es más corto y más fácil de leer.

Ahora imagina que queríamos tres mensajes: uno cuando la puntuación es más de 9000, uno cuando exactamente 9000, y uno cuando está por debajo de 9000. Podríamos escribir esto:

```
if score > 9000 {
    print("It's over 9000!")
} else {
    if score == 9000 {
        print("It's exactly 9000!")
    } else {
        print("It's not over 9000!")
    }
}
```

Una vez más, eso está exactamente bien y funciona como esperarías. Sin embargo, podemos hacer que el código sea más fácil de leer usando else if, lo que nos permite combinar el else con el if directamente después de él, así:

```
if score > 9000 {
    print("It's over 9000!")
} else if score == 9000 {
    print("It's exactly 9000!")
} else {
    print("It's not over 9000!")
}
```

Para probar esto, quiero usar una función Swift llamada print(): la ejecutas con algo de texto y se imprimirá.

Eso hace que nuestro código sea un poco más fácil de leer y entender, porque en lugar de tener condiciones anidadas, tenemos un solo flujo que podemos leer.

Puedes tener tantos cheques como quieras, pero necesitas exactamente uno si y cero o uno más.

Cómo comprobar múltiples condiciones

Swift nos da && y || para comprobar múltiples condiciones al mismo tiempo, y cuando se usan con solo dos condiciones, son bastante sencillas.

Por ejemplo, imagina que estábamos ejecutando un foro donde los usuarios podían publicar mensajes y eliminar cualquier mensaje que poseyeran. Podríamos escribir un código como este:

```
if isOwner == true || isAdmin == true {
    print("You can delete this post")
}
```

Donde las cosas se vuelven más confusas es cuando queremos comprobar varias cosas. Por ejemplo, podríamos decir que los usuarios habituales solo pueden eliminar mensajes que les permitimos, pero los administradores siempre pueden eliminar publicaciones. Podríamos escribir un código como este:

```
if isOwner == true && isEditingEnabled || isAdmin == true {
    print("You can delete this post")
}
```

Pero, ¿qué es lo que está tratando de comprobar? ¿En qué orden se ejecutan los cheques && y ||? Podría significar esto:

```
if (isOwner == true && isEditingEnabled) || isAdmin == true {
    print("You can delete this post")
}
```

Eso dice "si somos el propietario y la edición está habilitada, puedes eliminar la publicación, o si eres un administrador, puedes eliminar la publicación incluso si no la posees". Eso tiene sentido: la gente puede eliminar sus propias publicaciones si se permite la edición, pero los administradores siempre pueden eliminar publicaciones.

Sin embargo, también podrías leerlo así:

```
if isOwner == true && (isEditingEnabled || isAdmin == true) {
    print("You can delete this post")
}
```

Y ahora significa algo bastante diferente: "si eres el propietario de la publicación, y la edición está habilitada o eres el administrador, entonces puedes eliminar la publicación". Esto significa que los administradores no pueden eliminar publicaciones que no poseen, lo que no tendría sentido.

Obviamente, a Swift no le gusta la ambigüedad como esta, por lo que siempre interpretará el código como si habiéramos escrito esto:

```
if (isOwner == true && isEditingEnabled) || isAdmin == true {
    print("You can delete this post")
}
```

Sin embargo, honestamente, no es una buena experiencia dejar esto a Swift para que lo averigüe, por lo que podemos insertar los paréntesis nosotros mismos para aclarar exactamente lo que queremos decir.

No hay un consejo específico sobre esto, pero de manera realista, cada vez que mezclas && y || en una sola condición, es casi seguro que deberías usar paréntesis para dejar claro el resultado.

3. Cómo usar las instrucciones de interruptor para comprobar múltiples condiciones

Puedes usar if y else if repetidamente para comprobar las condiciones tantas veces como quieras, pero se vuelve un poco difícil de leer. Por ejemplo, si tuviéramos un pronóstico del tiempo a partir de una enumeración, podríamos elegir qué mensaje imprimir en función de una serie de condiciones, como esta:

```
enum Weather {
    case sun, rain, wind, snow, unknown
}

let forecast = Weather.sun

if forecast == .sun {
    print("It should be a nice day.")
} else if forecast == .rain {
    print("Pack an umbrella.")
} else if forecast == .wind {
    print("Wear something warm")
} else if forecast == .rain {
    print("School is cancelled.")
} else {
    print("Our forecast generator is broken!")
}
```

Eso funciona, pero tiene problemas:

Seguimos teniendo que escribir un pronóstico, a pesar de que estamos comprobando lo mismo cada vez.

Accidentalmente revisé .rain dos veces, a pesar de que la segunda comprobación nunca puede ser cierta porque la segunda comprobación solo se realiza si la primera comprobación falló.

No revisé .snow en absoluto, así que nos falta funcionalidad.

Podemos resolver esos tres problemas usando una forma diferente de comprobar las condiciones llamada interruptor. Esto también nos permite comprobar los casos individuales uno por uno, pero ahora Swift puede ayudar. En el caso de una enumerada, conoce todos los casos posibles que la enumerada puede tener, por lo que si nos perdemos uno o revisamos uno dos veces, se quejará.

Por lo tanto, podemos reemplazar todos esos si y de lo contrario si comprueba con esto:

```
switch forecast {
case .sun:
    print("It should be a nice day.")
case .rain:
    print("Pack an umbrella.")
case .wind:
    print("Wear something warm")
case .snow:
    print("School is cancelled.")
case .unknown:
    print("Our forecast generator is broken!")
}
```

Vamos a desglosar eso:

Comenzamos con el pronóstico de cambio, que le dice a Swift que ese es el valor que queremos comprobar.

Luego tenemos una serie de declaraciones de casos, cada una de las cuales son valores que queremos comparar con el pronóstico.

Cada uno de nuestros casos enumera un tipo de clima, y debido a que estamos activando el pronóstico, no necesitamos escribir Weather.sun, Weather.rain, etc. Swift sabe que debe ser algún tipo de Weather.

Después de cada caso, escribimos dos puntos para marcar el inicio del código a ejecutar si ese caso coincide.

Utilizamos una llave de cierre para terminar la declaración de cambio.

Si intentas cambiar .snow por .rain, verás que Swift se queja en voz alta: una vez que hemos comprobado .rain dos veces, y de nuevo que nuestra declaración de cambio no es exhaustiva, que no maneja todos los casos posibles.

Si alguna vez has usado otros lenguajes de programación, es posible que te hayas dado cuenta de que la declaración switch de Swift es diferente en dos lugares:

Todas las declaraciones de cambio deben ser exhaustivas, lo que significa que todos los valores posibles deben manejarse allí para que no pueda dejar uno por accidente.

Swift ejecutará el primer caso que coincida con la condición que está comprobando, pero no más. Otros idiomas a menudo llevan a cabo la ejecución de otro código de todos los casos posteriores, lo que suele ser la cosa predeterminada completamente incorrecta.

Aunque ambas afirmaciones son ciertas, Swift nos da un poco más de control si lo necesitamos.

En primer lugar, sí, todas las declaraciones de cambio deben ser exhaustivas: debe asegurarse de que todos los valores posibles estén cubiertos. Si está activando una cadena, entonces claramente no es posible hacer una verificación exhaustiva de todas las cadenas posibles porque hay un número infinito, por lo que en su lugar necesitamos proporcionar un caso predeterminado: código para ejecutar si ninguno de los otros casos coincide.

Por ejemplo, podríamos cambiar una cadena que contenga un nombre de lugar:

```
let place = "Metropolis"

switch place {
case "Gotham":
    print("You're Batman!")
case "Mega-City One":
    print("You're Judge Dredd!")
case "Wakanda":
    print("You're Black Panther!")
default:
    print("Who are you?")
}
```

Ese valor predeterminado: al final está el caso predeterminado, que se ejecutará si todos los casos no coinciden.

Recuerda: Swift comprueba sus casos en orden y ejecuta el primero que coincida. Si coloca el valor predeterminado antes que cualquier otro caso, ese caso es inútil porque nunca se igualará y Swift se negará a construir su código.

En segundo lugar, si desea explícitamente que Swift continúe ejecutando casos posteriores, use la caída. Esto no se usa comúnmente, pero a veces, solo a veces, puede ayudarte a evitar repetir el trabajo.

Por ejemplo, hay una famosa canción navideña llamada Los doce días de Navidad, y a medida que la canción continúa, cada vez hay más regalos para una persona desafortunada que alrededor del sexto día tiene una casa bastante llena.

Podríamos hacer una aproximación simple de esta canción usando fallthrough. En primer lugar, así es como se vería el código sin fallthrough:

```
let day = 5
print("My true love gave to me…")

switch day {
case 5:
    print("5 golden rings")
case 4:
    print("4 calling birds")
case 3:
    print("3 French hens")
case 2:
    print("2 turtle doves")
default:
    print("A partridge in a pear tree")
}
```

Eso imprimirá "5 anillos de oro", lo cual no es del todo correcto. El día 1 solo se debe imprimir "Una perdiz en un peral", en el día 2 debe ser "2 palomas tortuga" y luego "Una perdiz en un peral", en el día 3 debería ser "3 gallinas francesas", "2 palomas tortuga" y... bueno, ya tienes la idea.
Podemos usar la caída para obtener exactamente ese comportamiento:

```
let day = 5
print("My true love gave to me…")

switch day {
case 5:
    print("5 golden rings")
    fallthrough
case 4:
    print("4 calling birds")
    fallthrough
case 3:
    print("3 French hens")
    fallthrough
case 2:
    print("2 turtle doves")
    fallthrough
default:
    print("A partridge in a pear tree")
}
```

Eso coincidirá con el primer caso y imprimirá "5 anillos de oro", pero la línea de caída significa que el caso 4 ejecutará e imprimirá "4 pájaros de llamada", que a su vez usa el caso de nuevo para que se imprima "3 gallinas francesas", y así sucede. No es una combinación perfecta para la canción, ¡pero al menos puedes ver la funcionalidad en acción!

¿Cuándo deberías usar las declaraciones de interruptor en lugar de si?

Los desarrolladores de Swift pueden usar tanto switch como if para comprobar múltiples valores en su código, y a menudo no hay una razón difícil por la que deberías elegir uno en lugar del otro. Dicho esto, hay tres razones por las que podrías considerar el uso de switch en lugar de si:

- Swift requiere que sus declaraciones de cambio sean exhaustivas, lo que significa que debe tener un bloque de casos para cada valor posible a verificar (por ejemplo, todos los casos de una enumeración) o debe tener un caso predeterminado. Esto no es cierto para si y si, por lo que podrías perderte accidentalmente un caso.

- Cuando usas el interruptor para comprobar un valor para obtener múltiples resultados posibles, ese valor solo se leerá una vez, mientras que si lo usas, se leerá varias veces. Esto se vuelve más importante cuando empiezas a usar llamadas de función, porque algunas de ellas pueden ser lentas.

- Los casos de cambio de Swift permiten una coincidencia de patrones avanzada que es difícil de manejar con if.

Hay una situación más, pero es un poco más difusa: en términos generales, si quieres comprobar el mismo valor para tres o más estados posibles, encontrarás que la gente prefiere usar el interruptor en lugar de si para fines de legibilidad, si nada más, se hace más claro que estamos comprobando el mismo valor repetidamente, en lugar de escribir condiciones diferentes.

PD: He cubierto la palabra clave fallthrough porque es importante para la gente que viene de otros lenguajes de programación, pero es bastante raro verla usada en Swift. No te preocupes si estás luchando por pensar en escenarios en los que podría ser útil, ¡porque honestamente la mayoría de las veces no lo es!

4. Cómo usar el operador condicional ternario para pruebas rápidas

Hay una última forma de comprobar las condiciones en Swift, y cuando lo veas, lo más probable es que te preguntes cuándo es útil. Para ser justos, durante mucho tiempo rara vez usé este enfoque, pero como verás más adelante, es muy importante con SwiftUI.

Esta opción se llama operador condicional ternario. Para entender por qué tiene ese nombre, primero debes saber que +, -, ==, y demás se llaman operadores binarios porque funcionan con dos piezas de entrada: 2 + 5, por ejemplo, funciona con 2 y 5.

Los operadores ternarias trabajan con tres piezas de entrada, y de hecho, debido a que el operador condicional ternario es el único operador ternaria en Swift, a menudo escucharás que se llama simplemente "el operador ternario".

De todos modos, ya basta de nombres: ¿qué hace esto en realidad? Bueno, el operador ternario nos permite comprobar una condición y devolver uno de dos valores: algo si la condición es verdadera, y algo si es falsa.

Por ejemplo, podríamos crear una constante llamada edad que almacena la edad de alguien, y luego crear una segunda constante llamada canVote que almacenará si esa persona puede votar o no:

```
let age = 18
let canVote = age >= 18 ? "Yes" : "No"
```

Cuando se ejecute ese código, canVote se establecerá en "Sí" porque la edad se establece en 18 años.

Como puede ver, el operador ternario se divide en tres partes: un cheque (edad >= 18), algo para cuando la condición es verdadera ("Sí"), y algo para cuando la condición es falsa ("No"). Eso lo hace exactamente como un bloque normal de if y else, en el mismo orden.

Si ayuda, Scott Michaud sugirió un mnemónico útil: WTF. Significa "qué, verdadero, falso", y coincide con el orden de nuestro código:

- ¿Cuál es nuestro estado? Bueno, es la edad >= 18 años.
- ¿Qué hacer cuando la condición es cierta? Envíe de vuelta "Sí", para que se pueda almacenar en canVote.
- ¿Y si la condición es falsa? Devuelve el envío de “No".

Echemos un vistazo a algunos otros ejemplos, comencemos con uno fácil que lea una hora en formato de 24 horas e imprima uno de dos mensajes:

```
let hour = 23
print(hour < 12 ? "It's before noon" : "It's after noon")
```

Observe cómo eso no asigna el resultado a ninguna parte: el caso verdadero o el falso solo se imprime dependiendo del valor de la hora.

O aquí hay uno que lee el recuento de una matriz como parte de su condición, y luego devuelve una de dos cadenas:

```
let names = ["Jayne", "Kaylee", "Mal"]
let crewCount = names.isEmpty ? "No one" : "\(names.count) people"
print(crewCount)
```

Se vuelve un poco difícil de leer cuando tu condición usa == para comprobar la igualdad, como puedes ver aquí:

```
enum Theme {
    case light, dark
}

let theme = Theme.dark

let background = theme == .dark ? "black" : "white"
print(background)
```

La parte = tema == suele ser la parte que a la gente le resulta difícil de leer, pero recuerda desglosarla:

¿Qué? Tema == .dark

Cierto: "negro"

Falso: "blanco"

Por lo tanto, si el tema es igual a .dark, devuelve "Negro", de lo contrario, devuelve "Blanco", entonces asígnalo al fondo.

Ahora, es posible que se pregunte por qué el operador ternario es útil, especialmente cuando tenemos condiciones regulares de if/else disponibles para nosotros. Me doy cuenta de que no es una gran respuesta, pero tendrás que confiar en mí en esto: hay algunos momentos, particularmente con SwiftUI, en los que no tenemos otra opción y debemos usar un ternario.

Puedes ver más o menos cuál es el problema con nuestro código para comprobar las horas:

```
let hour = 23
print(hour < 12 ? "It's before noon" : "It's after noon")
```

Si quisiéramos escribir eso usando si y de lo contrario, tendríamos que escribir este código no válido:

```
print(
    if hour < 12 {
        "It's before noon"
    } else {
        "It's after noon"
    }
)
```

O ejecuta print() dos veces, así:

```
if hour < 12 {
    print("It's before noon")
} else {
    print("It's after noon")
}
```

Ese segundo funciona bien aquí, pero se vuelve casi imposible en SwiftUI, como verás mucho más tarde. Así que, a pesar de que podrías mirar al operador ternario y preguntarte por qué lo usarías, por favor, confía en mí: ¡importa!

¿Cuándo deberías usar el operador ternario en Swift?

El operador ternario nos permite elegir entre uno de los dos resultados basados en una condición, y lo hace de una manera muy concisa:

```
let isAuthenticated = true
print(isAuthenticated ? "Welcome!" : "Who are you?")
```

Algunas personas dependen en gran medida del operador ternario porque hace que el código sea muy corto, mientras que algunos se mantienen alejados de él tanto como sea posible porque puede hacer que el código sea más difícil de leer.

Estoy muy en el campamento de "evitar siempre que sea posible" porque a pesar de que este código es más largo, me resulta más fácil de seguir:

```
if isAuthenticated {
    print("Welcome")
} else {
    print("Who are you?")
}
```

Ahora, hay un momento en el que el operador ternario se usa mucho y es con SwiftUI. No quiero dar ejemplos de código aquí porque puede ser un poco abrumador, pero realmente puedes ir a la ciudad con el operador de ternario allí si quieres. Incluso entonces, prefiero eliminarlos cuando sea posible, para que mi código sea más fácil de leer, pero deberías probarlo por ti mismo y llegar a tus propias conclusiones.

¿Recuerdas las dos reglas de esta serie? Ya estás siendo increíble en el primero porque sigues volviendo por más (¡eres genial!), pero no olvides el segundo: publica tu progreso en línea, para que puedas beneficiarte de todo el aliento.