Hoy vamos a ver algunos tipos de datos más complejos que agrupan los datos. Comprender las diferencias entre ellos, y especialmente saber qué usar y cuándo, a veces puede hacer tropezar a la gente cuando están aprendiendo, como dijo Joseph Campbell una vez: "las computadoras son como los dioses del Antiguo Testamento: muchas reglas y sin piedad".

Sin embargo, no te preocupes: encontrarás que los tres tipos que estamos aprendiendo hoy cubren la abrumadora mayoría de los requisitos que necesitarás para agrupar datos, y si eliges el equivocado cuando estás aprendiendo, entonces no va a pasar nada malo.

De hecho, un error común que la gente comete mientras aprende es que tienen miedo de ejecutar su código. Confía en mí en esto: mientras sigues todo el código aquí, ninguno de los códigos que escribes puede romper de alguna manera tu ordenador. Así que, haz un cambio y ejetrútalo. Haz otro cambio y ejecuta eso. Solo intenta cambiar tus valores hasta que te sientas cómodo con lo que está sucediendo, todo ayuda.

Hoy tienes cuatro tutoriales a seguir, y conocerás cosas como matrices, diccionarios, conjuntos y más. Después de ver cada vídeo, puedes pasar por una lectura adicional opcional si quieres una aclaración adicional sobre un tema, y hay algunas pruebas cortas para ayudarte a asegurarte de que has entendido lo que se enseñó.

1. How to store ordered data in arrays

Es extremadamente común querer tener muchos datos en un solo lugar, ya sean los días de la semana, una lista de estudiantes en una clase, la población de una ciudad durante los últimos 100 años o cualquiera de los innumerables otros ejemplos.

En Swift, hacemos esta agrupación usando una matriz. Las matrices son su propio tipo de datos al igual que String, Int y Double, pero en lugar de contener solo una cadena, pueden contener cero cadenas, una cadena, dos cadenas, tres, cincuenta, cincuenta millones o incluso más cadenas; pueden adaptarse automáticamente para contener tantas como necesite, y siempre contener los datos en el orden en que los agregue.

Comencemos con algunos ejemplos sencillos de creación de matrices:

```
var beatles = ["John", "Paul", "George", "Ringo"]
let numbers = [4, 8, 15, 16, 23, 42]
var temperatures = [25.3, 28.2, 26.4]
```

Eso crea tres matrices diferentes: una que contiene cadenas de nombres de personas, otra que contiene enteros de números importantes y otra que contiene decimales de temperaturas en Celsius. Observe cómo comenzamos y terminamos las matrices usando corchetes, con comas entre cada elemento.

Cuando se trata de leer valores de una matriz, pedimos valores por la posición en la que aparecen en la matriz. La posición de un elemento en una matriz se llama comúnmente su índice.

Esto confunde un poco a los principiantes, pero Swift en realidad cuenta el índice de un elemento desde cero en lugar de uno: beatles[0] es el primer elemento, y beatles[1] es el segundo, por ejemplo.

Por lo tanto, podríamos leer algunos valores de nuestras matrices como esta:

```
print(beatles[0])
print(numbers[1])
print(temperatures[2])
```

Consejo: Asegúrese de que exista un elemento en el índice que está solicitando, de lo contrario su código se bloqueará; su aplicación dejará de funcionar.

Si tu matriz es variable, puedes modificarla después de crearla. Por ejemplo, puedes usar append() para añadir nuevos elementos:

```
beatles.append("Adrian")
```

Y no hay nada que te impida añadir artículos más de una vez:

```
beatles.append("Allen")
beatles.append("Adrian")
beatles.append("Novall")
beatles.append("Vivian")
```

Sin embargo, Swift observa el tipo de datos que está tratando de agregar, y se asegurará de que su matriz solo contenga un tipo de datos a la vez. Por lo tanto, este tipo de código no está permitido:

```
temperatures.append("Chris")
```

Esto también se aplica a la lectura de datos de la matriz: Swift sabe que la matriz de beatles contiene cadenas, por lo que cuando leas un valor, siempre obtendrás una cadena. Si intentas hacer lo mismo con los números, siempre obtendrás un entero. Swift no te permitirá mezclar estos dos tipos diferentes, por lo que este tipo de código no está permitido:

```
let firstBeatle = beatles[0]
let firstNumber = numbers[0]
let notAllowed = firstBeatle + firstNumber
```

Este es el tipo de seguridad, al igual que Swift no nos permite mezclar enteros y decimales, excepto que se lleva a un nivel más profundo. Sí, todos los beatles y los números son matrices, pero son tipos especializados de matrices: una es una matriz de cadenas y la otra es una matriz de enteros.

Puedes ver esto más claramente cuando quieras comenzar con una matriz vacía y añadirle elementos uno por uno. Esto se hace con una sintaxis muy precisa:

```
var scores = Array<Int>()
scores.append(100)
scores.append(80)
scores.append(85)
print(scores[1])
```

Ya hemos cubierto las últimas cuatro líneas, pero esa primera línea muestra cómo tenemos un tipo de matriz especializado: esta no es una matriz cualquiera, es una matriz que contiene enteros. Esto es lo que le permite a Swift saber con seguridad que los beatles[0] siempre deben ser una cadena, y también lo que nos impide añadir enteros a una matriz de cadenas.

Los paréntesis abiertos y de cierre después de Array<Int> están ahí porque es posible personalizar la forma en que se crea la matriz si es necesario. Por ejemplo, es posible que desee llenar la matriz con muchos datos temporales antes de agregar las cosas reales más adelante.

Puedes hacer otros tipos de matriz especializándolo de diferentes maneras, como esta:

```
var albums = Array<String>()
albums.append("Folklore")
albums.append("Fearless")
albums.append("Red")
```

Una vez más, hemos dicho que siempre debe contener cadenas, por lo que no podemos intentar poner un entero allí.

Las matrices son tan comunes en Swift que hay una forma especial de crearlas: en lugar de escribir Array<String>, puedes escribir [String]. Por lo tanto, este tipo de código es exactamente el mismo que antes:

```
var albums = [String]()
albums.append("Folklore")
albums.append("Fearless")
albums.append("Red")
```

La seguridad del tipo de Swift significa que siempre debe saber qué tipo de datos almacena una matriz. Eso podría significar ser explícito al decir que los álbumes son un Array<String>, pero si proporcionas algunos valores iniciales, Swift puede averiguarlo por sí mismo:

```
var albums = ["Folklore"]
albums.append("Fearless")
albums.append("Red")
```

Antes de que terminemos, quiero mencionar algunas funcionalidades útiles que vienen con las matrices.

En primer lugar, puedes usar .count para leer cuántos elementos hay en una matriz, al igual que lo hiciste con las cadenas:

```
print(albums.count)
```

En segundo lugar, puede eliminar elementos de una matriz utilizando remove(at:) para eliminar un elemento en un índice específico, o removeAll() para eliminar todo:

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

characters.remove(at: 2)
print(characters.count)

characters.removeAll()
print(characters.count)
```

Eso imprimirá 4, luego 3 y luego 0 a medida que se eliminen los caracteres.

En tercer lugar, puede comprobar si una matriz contiene un elemento en particular usando contains(), de esta manera:

```
let bondMovies = ["Casino Royale", "Spectre", "No Time To Die"]
print(bondMovies.contains("Frozen"))
```

En cuarto lugar, puedes ordenar una matriz usando sorted(), de esta manera:

```
let cities = ["London", "Tokyo", "Rome", "Budapest"]
print(cities.sorted())
```

Eso devuelve una nueva matriz con sus elementos ordenados en orden ascendente, lo que significa alfabéticamente para las cadenas, pero numéricamente para los números: la matriz original permanece sin cambios.

Finalmente, puedes invertir una matriz llamando a reversed() en ella:

```
let presidents = ["Bush", "Obama", "Trump", "Biden"]
let reversedPresidents = presidents.reversed()
print(reversedPresidents)
```

Consejo: Cuando inviertes una matriz, Swift es muy inteligente: en realidad no hace el trabajo de reorganizar todos los elementos, sino que solo recuerda a sí mismo que quieres que los elementos se inviertan. Por lo tanto, cuando imprimas los presidentes invertidos, ¡no te sorprendas de ver que ya no es solo una simple matriz!

Las matrices son extremadamente comunes en Swift, y tendrás muchas oportunidades de aprender más sobre ellas a medida que progreses. Incluso mejor sorted(), reversed() y muchas otras funcionalidades de matriz también existen para cadenas: usando sorted() pone las letras de la cadena en orden alfabético, haciendo algo como "swift" en "fistw".

¿Por qué Swift tiene matrices?

Las cadenas, enteros, booleanos y dobles de Swift nos permiten almacenar temporalmente valores individuales, pero si desea almacenar muchos valores, a menudo utilizará matrices en su lugar.

Podemos crear constantes y variables de matrices al igual que otros tipos de datos, pero la diferencia es que las matrices tienen muchos valores dentro de ellas. Por lo tanto, si quieres almacenar los nombres de los días de la semana, el pronóstico de temperatura para la próxima semana o las puntuaciones más altas de un videojuego, querrás una matriz en lugar de un solo valor.

Las matrices en Swift pueden ser tan grandes o tan pequeñas como quieras. Si son variables, puede agregarlos libremente para acumular sus datos con el tiempo, o puede eliminar o incluso reorganizar elementos si lo desea.

Leemos los valores de las matrices usando su posición numérica, contando desde 0. Este "contar desde 0" tiene un término técnico: podemos decir que las matrices de Swift se basan en cero. Swift bloqueará automáticamente su programa si intenta leer una matriz utilizando un índice no válido. Por ejemplo, crear una matriz con tres elementos y luego intentar leer el índice 10.

Sé lo que estás pensando: un bloqueo de una aplicación es malo, ¿verdad? Correcto. Pero créeme: si Swift no se bloqueó, entonces es muy probable que recuperaras datos malos, porque intentaste leer un valor fuera de lo que contiene tu matriz**.**

2. Cómo almacenar y encontrar datos en diccionarios

Has visto cómo las matrices son una excelente manera de almacenar datos que tienen un orden particular, como los días de la semana o las temperaturas de una ciudad. Las matrices son una gran opción cuando los elementos deben almacenarse en el orden en que los agrega, o cuando podría tener elementos duplicados allí, pero muy a menudo acceder a los datos por su posición en la matriz puede ser molesto o incluso peligroso.

Por ejemplo, aquí hay una matriz que contiene los detalles de un empleado:

```
var employee = ["Taylor Swift", "Singer", "Nashville"]
```

Te he dicho que los datos son sobre un empleado, por lo que podrías adivinar lo que hacen las diferentes partes:

```
print("Name: \(employee[0])")
print("Job title: \(employee[1])")
print("Location: \(employee[2])")
```

Pero eso tiene un par de problemas. En primer lugar, no puedes estar seguro de que el empleado[2] es su ubicación, tal vez esa sea su contraseña. En segundo lugar, no hay garantía de que el artículo 2 esté allí, sobre todo porque hicimos de la matriz una variable. Este tipo de código causaría serios problemas:

```
print("Name: \(employee[0])")
employee.remove(at: 1)
print("Job title: \(employee[1])")
print("Location: \(employee[2])")
```

Eso ahora imprime Nashville como el título del trabajo, que está mal, y hará que nuestro código se bloquee cuando lea empleado[2], lo cual es simplemente malo.

Swift tiene una solución para ambos problemas, llamadas diccionarios. Los diccionarios no almacenan los elementos de acuerdo con su posición como lo hacen las matrices, sino que nos permiten decidir dónde se deben almacenar los artículos.

Por ejemplo, podríamos reescribir nuestro ejemplo anterior para ser más explícitos sobre lo que es cada elemento:

```
let employee2 = ["name": "Taylor Swift", "job": "Singer", "location": "Nasl
```

Si lo dividimos en líneas individuales, tendrás una mejor idea de lo que hace el código:

```
let employee2 = [
    "name": "Taylor Swift",
    "job": "Singer",
    "location": "Nashville"
]
```

Como puedes ver, ahora estamos siendo muy claros: el nombre es Taylor Swift, el trabajo es Singer y la ubicación es Nashville. Swift llama a las cadenas de la izquierda - nombre, trabajo y ubicación - las claves del diccionario, y las cadenas de la derecha son los valores.

Cuando se trata de leer datos del diccionario, se utilizan las mismas claves que se utilizan para crearlo:

```
print(employee2["name"])
print(employee2["job"])
print(employee2["location"])
```

Si lo intentas en un patio de recreo, verás que Xcode lanza varias advertencias a lo largo de la línea de "¿Expresión implícitamente coaccionada desde 'String?' A "Cualquier". Peor aún, si miras la salida de tu patio de recreo, verás que imprime Opcional ("Taylor Swift") en lugar de solo Taylor Swift, ¿qué da?

Bueno, piensa en esto:

```
print(employee2["password"])
print(employee2["status"])
print(employee2["manager"])
```

Todo eso es código Swift válido, pero estamos tratando de leer claves de diccionario que no tienen un valor adjunto. Claro, Swift podría bloquearse aquí al igual que se bloqueará si lees un índice de matriz que no existe, pero eso haría que fuera muy difícil trabajar con él, al menos si tienes una matriz con 10 elementos, sabes que es seguro leer los índices del 0 al 9. ("Índices" es solo la forma plural de "índice", en caso de que no estuvieras seguro.)

Por lo tanto, Swift proporciona una alternativa: cuando accedes a los datos dentro de un diccionario, nos dirá "podrías recuperar un valor, pero no podrías recuperar nada en absoluto". Swift llama a estos opcionales porque la existencia de datos es opcional, podría estar allí o no.

Swift incluso te advertirá cuando escribas el código, aunque de una manera bastante oscura: dirá "¿Expresión implícitamente coaccionada por 'String?' A "Cualquier", pero realmente significará "estos datos podrían no estar realmente allí, ¿estás seguro de que quieres imprimirlos?"

Los opcionales son un tema bastante complejo que cubriremos en detalle más adelante, pero por ahora te mostraré un enfoque más simple: al leer desde un diccionario, puedes proporcionar un valor predeterminado para usar si la clave no existe.

Así es como se ve eso:

```
print(employee2["name", default: "Unknown"])
print(employee2["job", default: "Unknown"])
print(employee2["location", default: "Unknown"])
```

Todos los ejemplos han utilizado cadenas tanto para las claves como para los valores, pero puedes usar otros tipos de datos para cualquiera de ellos. Por ejemplo, podríamos hacer un seguimiento de qué estudiantes se han graduado de la escuela usando cadenas para los nombres y booleanos para su estado de graduación:

```
let hasGraduated = [
    "Eric": false,
    "Maeve": true,
    "Otis": false,
]
```

O podríamos rastrear los años en que se llevaron a cabo los Juegos Olímpicos junto con sus ubicaciones:

```
let olympics = [
    2012: "London",
    2016: "Rio de Janeiro",
    2021: "Tokyo"
]

print(olympics[2012, default: "Unknown"])
```

También puede crear un diccionario vacío utilizando cualquier tipo explícito que desee almacenar, y luego establecer las claves una por una:

```
var heights = [String: Int]()
heights["Yao Ming"] = 229
heights["Shaquille O'Neal"] = 216
heights["LeBron James"] = 206
```

Observe cómo necesitamos escribir [String: Int] ahora, para decir, un diccionario con cadenas para sus claves y números enteros para sus valores.

Debido a que cada elemento del diccionario debe existir con una clave específica, los diccionarios no permiten que existan claves duplicadas. En su lugar, si estableces un valor para una clave que ya existe, Swift sobrescribirá lo que fuera el valor anterior.

Por ejemplo, si estabas charlando con un amigo sobre superhéroes y supervillanos, podrías guardarlos en un diccionario como este:

```
var archEnemies = [String: String]()
archEnemies["Batman"] = "The Joker"
archEnemies["Superman"] = "Lex Luthor"
```

Si tu amigo no está de acuerdo con que el Joker sea el archienemigo de Batman, puedes reescribir ese valor usando la misma clave:

```
archEnemies["Batman"] = "Penguin"
```

Finalmente, al igual que las matrices y los otros tipos de datos que hemos visto hasta ahora, los diccionarios vienen con algunas funcionalidades útiles que querrás usar en el futuro: count y removeAll() existen para los diccionarios y funcionan igual que para las matrices.

¿Por qué Swift tiene diccionarios y matrices?

Los diccionarios y las matrices son ambas formas de almacenar muchos datos en una variable, pero los almacenan de diferentes maneras: los diccionarios nos permiten elegir una "clave" que identifique el elemento que queremos agregar, mientras que las matrices solo agregan cada elemento de forma secuencial.

Por lo tanto, en lugar de tratar de recordar que el índice de matriz 7 significa el país de un usuario, podríamos simplemente escribir usuario["país"], es mucho más conveniente.

Los diccionarios no almacenan nuestros artículos utilizando un índice, sino que optimizan la forma en que almacenan los artículos para una recuperación rápida. Por lo tanto, cuando decimos usuario ["país"], devolverá el artículo en esa clave (o nil) al instante, incluso si tenemos un diccionario con 100.000 artículos dentro.

Recuerda, no se te puede garantizar que exista una clave en un diccionario. Esta es la razón por la que leer un valor de un diccionario podría no devolver nada: ¡es posible que hayas solicitado una clave que no existe!

¿Por qué Swift tiene valores predeterminados para los diccionarios?

Cada vez que lea un valor de un diccionario, puede recuperar un valor o puede recuperar el valor, es posible que no haya ningún valor para esa clave. No tener ningún valor puede causar problemas en su código, sobre todo porque necesita agregar funcionalidad adicional para manejar los valores que faltan de forma segura, y ahí es donde entran en adí los valores predeterminados del diccionario: le permiten proporcionar un valor de copia de seguridad para usar cuando la clave que solicita no existe.

Por ejemplo, aquí hay un diccionario que almacena los resultados del examen para un estudiante:

```
let results = [
    "english": 100,
    "french": 85,
    "geography": 75
]
```

Como puedes ver, hicieron tres exámenes y obtuvieron una puntuación del 100 %, el 85 % y el 75 % en inglés, francés y geografía. Si quisiéramos leer su puntuación de historia, la forma en que lo hagamos depende de lo que queramos:

Si un valor que falta significa que el estudiante no realtó la prueba, entonces podríamos usar un valor predeterminado de 0 para que siempre obtengamos un entero.

Si un valor que falta significa que el estudiante aún no ha hecho el examen, entonces deberíamos omitir el valor predeterminado y, en su lugar, buscar un valor nulo.

Por lo tanto, no es que siempre necesites un valor predeterminado cuando trabajas con diccionarios, pero cuando lo haces es fácil:

```
let historyResult = results["history", default: 0]
```

3. Cómo usar conjuntos para una búsqueda rápida de datos

Hasta ahora has aprendido sobre dos formas de recopilar datos en Swift: matrices y diccionarios. Hay una tercera forma muy común de agrupar datos, llamada conjunto: son similares a las matrices, excepto que no se pueden agregar elementos duplicados y no almacenan sus elementos en un orden en particular.

Crear un conjunto funciona como crear una matriz: dile a Swift qué tipo de datos almacenará, luego sigue adelante y añade cosas. Sin embargo, hay dos diferencias importantes, y es mejor demostrarlas usando algo de código.

En primer lugar, así es como harías un conjunto de nombres de actores:

```
let people = Set(["Denzel Washington", "Tom Cruise", "Nicolas Cage", "Samu
```

¿Te das cuenta de cómo eso realmente crea una matriz primero y luego pone esa matriz en el conjunto? Eso es intencional, y es la forma estándar de crear un conjunto a partir de datos fijos. Recuerde, el conjunto eliminará automáticamente cualquier valor duplicado y no recordará el orden exacto que se utilizó en la matriz.

Si tienes curiosidad por saber cómo el conjunto ha ordenado los datos, intenta imprimirlo:

```
print(people)
```

Es posible que veas los nombres en el orden original, pero también puedes obtener un pedido completamente diferente: al conjunto simplemente no le importa en qué orden vienen sus artículos.

La segunda diferencia importante al añadir elementos a un conjunto es visible cuando se añaden elementos individualmente. Aquí está el código:

```
var people = Set<String>()
people.insert("Denzel Washington")
people.insert("Tom Cruise")
people.insert("Nicolas Cage")
people.insert("Samuel L Jackson")
```

¿Te has dado cuenta de cómo estamos usando insert()? Cuando teníamos una matriz de cadenas, añadimos elementos llamando a append(), pero ese nombre no tiene sentido aquí: no estamos agregando un elemento al final del conjunto, porque el conjunto almacenará los elementos en el orden que quiera.

Ahora, podrías pensar que los conjuntos suenan como matrices simplificadas; después de todo, si no puedes tener duplicados y pierdes el orden de tus artículos, ¿por qué no usar matrices? Bueno, ambas restricciones en realidad se convierten en una ventaja.

En primer lugar, no almacenar duplicados a veces es exactamente lo que quieres. Hay una razón por la que elegí actores en el ejemplo anterior: el Sindicato de Actores requiere que todos sus miembros tengan un nombre artístico único para evitar confusiones, lo que significa que nunca se deben permitir duplicados. Por ejemplo, el actor Michael Keaton (Spider-Man Homecoming, Toy Story 3, Batman y más) en realidad se llama Michael Douglas, pero debido a que ya había un Michael Douglas en el gremio (Avengers, Falling Down, Romancing the Stone y más), tenía que tener un nombre único.

En segundo lugar, en lugar de almacenar sus artículos en el orden exacto que especifique, los conjuntos en su lugar los almacenan en un orden altamente optimizado que hace que sea muy rápido localizar los artículos. Y la diferencia no es pequeña: si tienes una matriz de 1000 nombres de películas y usas algo como contains() para comprobar si contiene "El Caballero Oscuro", Swift necesita revisar cada elemento hasta que encuentre uno que coincida, eso podría significar verificar los 1000 nombres de películas antes de devolver falso, porque El Caballero Oscuro no estaba en la matriz.

En comparación, llamar a contains() en un conjunto se ejecuta tan rápido que te costaría medirlo de manera significativa. Diablos, incluso si tuvieras un millón de artículos en el conjunto, o incluso 10 millones de artículos, todavía se ejecutaría al instante, mientras que una matriz podría tardar minutos o más en hacer el mismo trabajo.

La mayoría de las veces te encontrarás usando matrices en lugar de conjuntos, pero a veces, solo a veces, encontrarás que un conjunto es exactamente la opción correcta para resolver un problema en particular, y hará que el código se ejecute lentamente en muy poco tiempo.

Consejo: Junto con contains(), también encontrarás count para leer el número de elementos en un conjunto, y sorted() para devolver una matriz ordenada que contiene los elementos del conjunto.

¿Por qué los conjuntos son diferentes de las matrices en Swift?

Tanto los conjuntos como las matrices son importantes en Swift, y entender cuáles son sus diferencias te ayudará a entender cuál elegir para cualquier circunstancia dada.

Tanto los conjuntos como las matrices son colecciones de datos, lo que significa que tienen múltiples valores dentro de una sola variable. Sin embargo, lo que importa es cómo mantienen sus valores: los conjuntos no están ordenados y no pueden contener duplicados, mientras que las matrices conservan su orden y pueden contener duplicados.

Esas dos diferencias pueden parecer pequeñas, pero tienen un efecto secundario interesante: debido a que los conjuntos no necesitan almacenar sus objetos en el orden en que los agrega, en su lugar pueden almacenarlos en un orden aparentemente aleatorio que los optimiza para una recuperación rápida. Por lo tanto, cuando dices "este conjunto contiene el elemento X", obtendrás una respuesta en una fracción de segundo, sin importar lo grande que sea el conjunto.

En comparación, las matrices deben almacenar sus artículos en el orden que les das, por lo que para comprobar si el artículo X existe en una matriz que contiene 10.000 artículos, Swift debe comenzar en el primer artículo y comprobar cada artículo hasta que se encuentre, o tal vez no se encuentre en absoluto.

Esta diferencia significa que los conjuntos son más útiles para los momentos en los que quieres decir "¿existe este artículo?" Por ejemplo, si desea comprobar si una palabra aparece en un diccionario, debe usar un conjunto: no tiene duplicados y desea hacer una búsqueda rápida.

Para obtener más información sobre este tema, echa un vistazo a la publicación de Antoine van der Lee: https://www.avanderlee.com/swift/array-vs-set-differences-explained/

4. Cómo crear y usar enumeraciones

Una enumeración, abreviatura de enumeración, es un conjunto de valores con nombre que podemos crear y usar en nuestro código. No tienen ningún significado especial para Swift, pero son más eficientes y seguros, por lo que los usarás mucho en tu código.

Para demostrar el problema, digamos que querías escribir algo de código para que el usuario seleccionara un día de la semana. Podrías empezar así:

```
var selected = "Monday"
```

Más adelante en tu código lo cambias, así:

```
selected = "Tuesday"
```

Eso podría funcionar bien en programas muy simples, pero echa un vistazo a este código:

```
selected = "January"
```

¡Vaya! Escribiste accidentalmente un mes en lugar de un día, ¿qué hará tu código? Bueno, puede que tengas la suerte de que un colega detecte el error mientras revisa tu código, pero ¿qué tal esto?

```
selected = "Friday "
```

Eso tiene un espacio al final del viernes, y "Viernes" con un espacio es diferente de "Viernes" sin un espacio en los ojos de Swift. De nuevo, ¿qué haría tu código?

El uso de cuerdas para este tipo de cosas requiere una programación muy cuidadosa, pero también es bastante ineficiente: ¿realmente necesitamos almacenar todas las letras de "viernes" para rastrear un solo día?

Aquí es donde entran en ado las enumeraciones: nos permiten definir un nuevo tipo de datos con un puñado de valores específicos que puede tener. Piensa en un booleano, que solo puede tener verdadero o falso, no puedes establecerlo en "tal vez" o "probablemente", porque eso no está en el rango de valores que entiende. Las enumeraciones son las mismas: podemos enumerar por adelantado el rango de valores que puede tener, y Swift se asegurará de que nunca cometas un error al usarlos.

Por lo tanto, podríamos reescribir nuestros días de semana en una nueva enumeración como esta:

```
enum Weekday {
    case monday
    case tuesday
    case wednesday
    case thursday
    case friday
}
```

Eso llama a la nueva enumeración de día de la semana, y proporciona cinco casos para manejar los cinco días de la semana.

Ahora, en lugar de usar cadenas, usaríamos la enumerada. Prueba esto en tu patio de recreo:

```
var day = Weekday.monday
day = Weekday.tuesday
day = Weekday.friday
```

Con ese cambio no puedes usar accidentalmente "Viernes" con un espacio adicional allí, o poner un nombre de mes en su lugar; siempre debes elegir uno de los días posibles enumerados en la enumeración. Incluso verás que Swift ofrece todas las opciones posibles cuando hayas escrito Weekday, porque sabe que vas a seleccionar uno de los casos.

Swift hace dos cosas que hacen que las enumeraciones sean un poco más fáciles de usar. En primer lugar, cuando tienes muchos casos en una enumeración, solo puedes escribir el caso una vez, luego separar cada caso con una coma:

```
enum Weekday {
    case monday, tuesday, wednesday, thursday, friday
}
```

En segundo lugar, recuerde que una vez que asigna un valor a una variable o constante, su tipo de datos se vuelve fijo: no puede establecer una variable en una cadena al principio, y luego un entero más tarde. Bueno, para las enumeraciones, esto significa que puedes omitir el nombre de la enumeración después de la primera tarea, así:

```
var day = Weekday.monday
day = .tuesday
day = .friday
```

Swift sabe que .tuesday debe referirse a Weekday.tuesday porque el día siempre debe ser algún tipo de día de la semana.

Aunque no es visible aquí, un beneficio importante de las enumeraciones es que Swift las almacena en una forma optimizada, cuando decimos Weekday.monday es probable que Swift almacene eso usando un solo entero como 0, que es mucho más eficiente de almacenar y comprobar que las letras M, o, n, d, a, y.

¿Por qué Swift necesita enumeraciones?

Las enumeraciones son una característica extraordinariamente poderosa de Swift, y las verás usadas de muchas maneras y lugares. Muchos idiomas no tienen enumeraciones y lo hacen muy bien, ¡así que podrías preguntarte por qué Swift necesita enumeraciones!

Bueno, en su forma más simple, una enumerada es simplemente un buen nombre para un valor. Podemos hacer una enumerada llamada Dirección con casos para norte, sur, este y oeste, y referirnos a los de nuestro código. Claro, podríamos haber usado un entero en su lugar, en cuyo caso nos referiríamos a 1, 2, 3 y 4, pero ¿realmente podrías recordar lo que significaba 3? ¿Y si escribiste 5 por error?

Por lo tanto, las enumeraciones son una forma de decir que Direction.north significa algo específico y seguro. Si huéramos escrito Direction.thatWay y no existiera tal caso, Swift simplemente se negaría a construir nuestro código, no entiende el caso de la enumeración. Detrás de escena, Swift puede almacenar sus valores de enumeración de forma muy sencilla, por lo que son mucho más rápidos de crear y almacenar que algo así como una cadena.

A medida que progreses, aprenderás cómo Swift nos permite añadir más funcionalidad a las enumeraciones: son más potentes en Swift que en cualquier otro idioma que haya visto.

Ya has terminado el tercer día, y estás empezando a escribir código Swift útil. ¡Asegúrate de contarle al mundo sobre tu progreso!

¡Comparte tu progreso!

Si usas Twitter, el botón de abajo preparará un tuit que diga que has completado hoy, junto con un gráfico de celebración, la URL de esta página y el hashtag del desafío. No te preocupes, ¡no se enviará hasta que lo confirmes en Twitter!