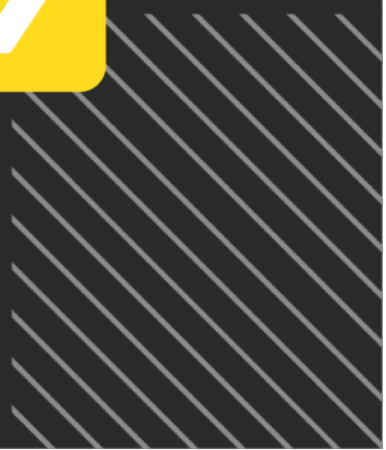




Swift





Tema 4. Control de flujos

Objetivos

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

- realizar validaciones usando `if / else`, `guard` y operador ternario;
- comprender el concepto de *optional binding* y su uso en las validaciones;
- ejecutar ciclos usando `While`, `For` y `Repeat`;
- iterar en colecciones;
- realizar validaciones utilizando `Switch`; y
- comprender el uso de las sentencias de flujo `continue`, `break`, `fallthrough`.

Sentencias condicionales

Son utilizadas para decidir qué bloques de código ejecutar sobre la base de si una condición se cumple o no.

If

En su versión más simple, una sentencia `If` evalúa si una condición es verdadera; en caso de que sea así, se ejecutará el bloque de código siguiente dentro de llaves (`{ }`).

```
if condición {  
    // bloque de código  
}  
var temperaturaActual = 5 // Temperatura en Celsius  
if temperaturaActual < 18 {  
    print("Hace mucho frío")  
}
```

Ya que 5 es menor que 18, se ingresa al bloque del `If` y se imprime "Hace mucho frío".

Cláusula else

La sentencia `if` también provee un bloque de código para cuando la sentencia evaluada es falsa.

```
var temperaturaActual = 30 // Temperatura en Celsius
if temperaturaActual < 18 { // 30 no es menor a 18
    print("Hace mucho frío")
} else {
    print("Hace calor")
}
```

Ya que 30 no es menor que 18, se ingresa al bloque del `else` y se imprime "Hace calor".

Multiples sentencias If

Pueden agregarse varias sentencias para validar diferentes casos:

```
var temperaturaActual = 23 // Temperatura en Celsius
if temperaturaActual < 15 {
    print("Hace mucho frío, ya que hace menos de 15 grados.")
} else if temperaturaActual > 30 {
    print("Hace calor, ya que hace más de 30 grados.")
} else {
    print("No hace ni frío ni calor, la temperatura estará entre 15 y 30")
}
```

El valor por pantalla será "No hace ni frío ni calor, la temperatura estará entre 15 y 30".

If anidados

Es posible anidar sentencias `if` para realizar diferentes validaciones posteriores a una validación inicial.

```
var temperaturaActual = 23 // Temperatura en Celsius
if temperaturaActual < 15 {
    if temperaturaActual >= 0 {
        print("Hace mucho frío, ya que estamos entre 15 y 0 grados.")
    } else {
        print("Está helado afuera, ya que estamos a menos de 0 grados.")
    }
}
```

```
}  
} else if temperaturaActual > 30 {  
    print("Hace calor, ya que hace más de 30 grados.")  
} else {  
    print("No hace ni frío ni calor, la temperatura estará entre 15 y 30")  
}
```

Normalmente, no es recomendable usar muchos `if` anidados, pues se dificulta la comprensión y escalabilidad del código.

Guard

Una sentencia `guard` ejecuta el bloque de código según el valor booleano de una expresión. Es usada, generalmente, cuando se requiere que una validación sea verdadera para que se continúe con la ejecución del código o, en caso contrario, salir del flujo de forma rápida.

- Una sentencia `guard` requiere que una condición sea verdadera para que se ejecute el código después de la sentencia `guard`.
- La sentencia `guard` siempre tiene una cláusula `else`: el código dentro de la cláusula `else` se ejecuta si la condición no es verdadera.
- Dentro de la cláusula `else`, debe incluirse un `return` que indique la finalización del flujo.

```
let usuarioRegistrado = true  
guard usuarioRegistrado else {  
    print("El usuario debe estar registrado")  
    return  
}  
mostrarPantallaDelInicio()  
configurarPreferenciasDelUsuario()  
print("Bienvenido!")
```

En este ejemplo, se requiere que el usuario esté registrado para continuar con el flujo normal de la aplicación. En caso que no lo esté, se mostrará el mensaje "El usuario debe estar registrado" y se terminará el flujo. En caso de que sí esté registrado, se continuará el flujo llamando las funciones correspondientes y finalmente mostrando el mensaje "Bienvenido!".

En resumen, el `guard` es muy útil para chequear qué condiciones se tienen que cumplir para poder ejecutar el código posterior al `guard`.

Es muy bueno para detectar errores y así evitar que se siga corriendo el programa.

Más adelante veremos qué otro tipo de palabras reservadas además de `return` se puede utilizar para retornar el control del programa en la cláusula `else` de un `guard` (`throw`, `break`, `continue`, `fatalError`).

Optional binding

En ocasiones, se requiere validar que el valor de un opcional no sea `nil` para continuar con el flujo; por ejemplo, si se definen el usuario y contraseña como opcionales (dado que inicialmente no se han ingresado los valores), solo debe permitirse avanzar cuando ambos valores no sean nulos.

Para este tipo de casos, se utiliza lo que se conoce como *optional binding*, que implica validar si el valor de un opcional existe, almacenarlo en una nueva variable y seguir trabajando con esta nueva variable.

```
let valorOpcional: String? = "Existe Valor"
if let valor = valorOpcional {
    print("\(valor)")
}
```

Dado que `valorOpcional` tiene como valor `"Existe Valor"`, al llevar a cabo el *optional binding*, la validación es verdadera y se entra al bloque de código del `if`.

La versión para `guard` de esto sería la siguiente:

```
let valorOpcional: String? = "Existe Valor"
guard let valor = valorOpcional else { return }
print("\(valor)")
```

Otro ejemplo obtenemos también de los diccionarios. Como se aprendió en la cápsula “Tipos de datos II”, al intentar acceder a un valor de un diccionario, se puede obtener un `nil` en caso que la clave no exista; para esto, se puede usar el *optional binding*:

```
let persona: [String: String] = [:]

guard let nombre = persona["nombre"] else { return }
print("Hola \(nombre)!")
```

```
guard let locacion = persona["locacion"] else {
    print("No tenemos información sobre su ubicación.")
    return
}
print("\n(nombre), espero que el clima este lindo en \(locacion).")
Inicialmente persona es un diccionario vacío, por tanto al validar el
primer guardse entraría al else y no se mostraría ningún mensaje.
```

¡Hazlo tú!

Prueba en el Playground los siguientes casos y valida el resultado:

```
// Ejecutar el código modificando la primera sentencia.
let persona: [String: String] = [:]
// No hay mensaje por consola.
let persona: [String: String] = ["nombre": "Roberto"]
// "Hola Roberto!"
// Consola: "No tenemos información sobre su ubicación."
let persona: [String: String] = ["nombre": "Martin",
                                "locacion": "Caba"]

// Consola: "Hola Martin!"
// Consola: "Martin, espero que el clima este lindo en Caba."
```

Operador ternario

Existe una forma simplificada de realizar validaciones sin tener que incluir un if-else. Esto es útil cuando el código dentro del bloque if-else es corto. La sintaxis es condición? siEsVerdadero : siEsFalso

```
let lamparaPrendida = true
let estadoLampara = lamparaPrendida ? "Prendida" : "Apagada"
print("La lámpara está: \(estadoLampara)" // "La lámpara esta:
Prendida"
```

Ejercitación 1 en el Playground de Xcode

1. Declara una constante con un valor entero correspondiente a la edad de una persona, luego define un If que valide lo siguiente respecto de la edad:

- Es menor a 13 ➡ imprima “Niño”
- Está entre 14 y 17 ➡ imprima “Adolescente”
- Está entre 18 y 39 ➡ imprima “Adulto joven”

- Está entre 40 y 49 ➡ imprima “Adulto medio”
- Mayor a 50 ➡ imprima “Adulto maduro”
-

2. Teniendo tres número enteros en constantes `a`, `b` y `c`, imprime dichos números ordenados de mayor a menor. Asume que los números son diferentes entre sí.

3. Teniendo un número entero en una constante, imprime “válido” si el número leído es par y es menor que -100; de lo contrario, no imprimas nada. Resuelve este problema con `if` anidado; luego, como una expresión booleana compuesta; y finalmente, usando `guard`.

4. Teniendo una constante con un `String?`, utiliza la sentencia `guard` para imprimir por pantalla “El valor ingresado no existe” en caso de que el valor sea nulo. Valida la longitud del texto usando la función `length`; y si el valor tiene una longitud menor que 5 caracteres, imprime en consola “El valor ingresado es menor a 5 caracteres”. En el caso de que el valor exista y tenga más de 5 caracteres, imprime por consola el valor ingresado.

Ciclos For-In

Se utiliza este ciclo para iterar sobre una secuencia. La sintaxis es la siguiente: `for element in sequence { ... }`, donde `sequence` corresponde al grupo de elementos sobre los que se quieren iterar, y `element` corresponde al elemento actual que se está iterando:

```
let nombres = ["Diego", "Camila", "Belen"]
for nombre in nombres {
  print("Hola \(nombre)")
}
```

```
// "Hola Diego"
// "Hola Camila"
// "Hola Belen"
```

También puede utilizarse especificando un rango:

```
for numero in (0...10) {
  print("Número: \(numero)")
}
```

```
// "Hola Diego"
// "Hola Camila"
// "Hola Belen"
```

Ciclo While

Un ciclo `while` ejecuta un conjunto de sentencias hasta que la condición validada sea falsa. Se utiliza preferentemente cuando no se conoce el número de iteraciones antes de que comience la primera iteración. Hay dos opciones:

While

Un ciclo `while` empieza evaluando una condición. Si es verdadera, un conjunto de sentencias serán ejecutadas hasta que la condición se haga falsa.

```
var i = 0
```

```
while i < 10 {  
    print("Índice: \(i)")  
    i = i + 1  
}
```

```
// Se ejecuta 10 veces el código y se imprimirá "Índice: i"  
// Siendo i un número del 0 al 9.
```

Repeat-While

Este ciclo realiza una pasada inicial por el conjunto de sentencias antes de evaluar la condición. Si es verdadera, volverá a ejecutar el bloque de código hasta que la condición se haga falsa.

```
i = 0  
repeat {  
    print("Índice: \(i)")  
    i = i + 1  
} while i < 10
```

```
// Se imprimirá 10 veces el código y se imprimirá "Índice i"  
// Siendo i un número del 0 al 9.
```

En resumen, el ciclo `repeat-while` se ejecuta al menos una vez, mientras que el `while` puede no ejecutarse ninguna vez.

Ejercitación 2 en el Playground de Xcode



1. Dada la siguiente lista de números [12, 24, 36, 48] y utilizando el ciclo adecuado, calcula en una variable auxiliar la sumatoria de todos ellos.
2. Dada la siguiente lista de números [10, 7, 8, 4, 2, 24, 30], sin utilizar el ciclo `for-In`, calcula la sumatoria de números hasta que la sumatoria parcial sea mayor que 30.
3. Dada la lista de números [20, 5, 10, 9, 15, 8, 3, 30, 35], realiza la sumatoria, con un ciclo `for-In`, solo de los números que sean mayores que 10.
4. Utilizando ciclos anidados, escribe las tablas de multiplicar del 1 al 10.
5. Dado un conjunto de letras ["A", "a", "B", "c", "D", "s", "z", "i", "l"], concatena todas aquellas que sean minúsculas hasta que se encuentre la letra "z".

Switch

Una sentencia `switch` proporciona una alternativa a la sentencia `if` para responder a múltiples potenciales estados, donde se considera un valor y se compara con varios patrones posibles; luego, se ejecuta el bloque de código apropiado basado en el primer patrón que coincida con éxito. Al cumplirse una validación y ejecutar el código correspondiente, el flujo del `switch` se termina, es decir que no siguen realizando las validaciones siguientes.

Los `switch` deben ser exhaustivos, es decir, que para cualquier valor que se ingrese a la validación, se debe tener una sentencia válida. Para esto, se usa el `default`.

La sintaxis genérica es la siguiente:

```
switch (valorAconsiderar) {  
  case valor 1:  
    // Bloque de Código para Valor 1  
  case valor 2, valor 3:  
    // Bloque de Código para valor 2 o valor 3  
  default:  
    // Si no se cumple ninguna condición anterior, ejecutará este  
    bloque de código  
}
```

Caso Base

```
let unCharacter: Character = "z"
switch unCharacter {
case "a":
    print("Es la primera letra del abecedario")
case "z":
    print("Es la última letra del abecedario")
default:
    print("Es otro carácter")
}
```

En este caso, se imprimirá "Es la última letra del abecedario".

Caso compuesto

Pueden realizarse varias validaciones dentro de un mismo caso del switch.

```
let unCharacter: Character = "Z"
switch unCharacter {
case "a", "A":
    print("Es la primera letra del abecedario")
case "z", "Z":
    print("Es la última letra del abecedario")
default:
    print("Es otro carácter")
}
```

En este caso, se imprimirá "Es la última letra del abecedario".

Caso por intervalos

Puedes utilizar, también, rangos para realizar la validaciones:

```
let numero = 60
switch numero {
case 0: print("Cero")
case 1..<10: print("Mayor a Cero, menor a 10")
case 10..<100: print("Menor a 100, pero al menos 10")
default: print("Mayor a 100")
}
```

En este caso, se imprimirá "Menor a 100, pero al menos 10".

Caso tupla

Se pueden usar tuplas para validar múltiples valores en una sentencia switch. Cada elemento puede ser probado contra un valor diferente o intervalo de valores. Es posible utilizar el comodín `_` para validar contra cualquier valor.

Considerando los ejes cartesianos x, y , donde x va de -2 a 2 e y de -2 a 2:

```
let algunPunto = (1, 0)
switch algunPunto {
case (0, 0): print("El punto está en el origen")
case (_, 0): print("El punto está sobre el eje X")
case (0, _): print("El punto está sobre el eje Y")
case (-2...2, -2...2): print("El punto está adentro del plano")
default: print("El punto está por fuera del plano")
}
```

En este caso, se imprimirá "El punto está sobre el eje X".

Vinculación de valores (*value binding*)

Los `switch case` también pueden dar un nombre a los valores en constantes o variables, para ser usados durante el alcance del `case` correspondiente.

Si retomamos el ejemplo anterior, obtenemos lo siguiente:

```
let algunPunto = (1, 0)
switch algunPunto {
case (0, 0): print("El punto está en el origen")
case (let x, 0): print("El punto está sobre el eje X con un valor de \(x)")
case (0, let y): print("El punto está sobre el eje Y con un valor de \(y)")
case (-2...2, -2...2): print("El punto está adentro del plano")
default: print("El punto está por fuera del plano")
}
```

En este caso, se imprimirá "Está sobre el eje x con un valor de 2".

Where

Es posible utilizar la sentencia `where` para verificar por otras condiciones específicas.

Con el plano cartesiano, obtenemos lo siguiente:

```
let otroPunto = (1, -1)
switch otroPunto {
case let (x, y) where x == y:
    print("\(x), \(y) está sobre la línea x == y")
case let (x, y) where x == -y:
    print("\(x), \(y) está sobre la línea x == -y")
case let (x, y):
    print("\(x), \(y) está en algún punto arbitrario")
}
```

En este caso, se imprimirá `"(1, -1) está sobre la línea x == -y"`.

Sentencias de control de flujo

Las sentencias de transferencia de control cambian el orden de ejecución del código, transfiriendo el control de una parte del código a otra (`continue`, `break`, `fallthrough`).

Continue

Le dice a un ciclo que deje de hacer lo que está haciendo y comience de nuevo al principio de la siguiente iteración del ciclo.

```
let input = "algunas cosas"
var output = ""
let caracteresARemover: [Character] = ["a", "e", "i", "o", "u", " "]
for caracter in input {
    if caracteresARemover.contains(caracter) {
        continue
    }
    output.append(caracter)
}
print(output)
```

En este ejemplo, hay un ciclo que itera sobre los caracteres de `input`. Si el carácter pertenece a las vocales o es un espacio vacío, entonces se avanza a la siguiente iteración; en caso contrario, se adiciona el carácter a `output`, lo que da como resultado `"lgnsCSS"`.

Break

Termina la ejecución de una sentencia de flujo de control completa de forma inmediata.

Se puede utilizar dentro de una sentencia `switch` o un ciclo cuando se desea terminar la ejecución de estos antes de lo que se haría de otro modo.

```
let valor: String = "2"
switch valor {
case "a", "e", "i", "o", "u": print("Es una vocal")
case "1", "2", "3": break
default: print("Es un caracter no contemplado")
}
```


En este caso, no se mostrará nada en consola, ya que pasará por el caso `"1","2","3"` y al ejecutarse el `break`, se saldrá del `switch`.

Fallthrough

Como se indicó anteriormente, en Swift, toda la sentencia `switch` completa su ejecución tan pronto como el primer caso coincidente es completado.

```
let numero = 5
var descripcion = "El número \(numero) es"
switch numero {
case 2, 3, 5, 7, 11, 13, 17, 19:
    descripción += " un número primo, y también"
    fallthrough
default:
    descripción += " es un entero."
}
print(descripción)
```

Se mostrará en consola `"El número 5 es un número primo, y también es un entero"`.



⚠ La palabra clave `fallthrough` no comprueba la siguiente condición; simplemente hace que la ejecución del código se mueva directamente a las sentencias dentro del siguiente bloque `case` (o `default case`).

Ejercitación 3 en el Playground de Xcode

1. Retoma el ejercicio 1 de la ejercitación 1 de esta cápsula e imprime el mensaje correspondiente según la edad utilizando `switch`.

2. Dados los siguientes números `[10, 60, -5, 0, -30, 29, 21, 40]`, realiza la sumatoria de puntos dadas por las reglas a continuación:

- Se debe utilizar un `guard` para eliminar aquellos valores que no sean positivos.
- Se otorgará 1 punto si el número es menor que 10.
- Se otorgará 0 puntos si el valor está entre 10 y 30.
- Se otorgarán 3 puntos si el valor es mayor que 50.
- Para cualquier otro valor, se otorgarán 10 puntos.
-

3. Indique si las siguientes expresiones con relación a los `switch` son verdaderas o falsas:

- Se deben especificar todos los posibles casos de la validación o incluir un `default`.
- Se pueden tener un máximo de 10 `case` en un `switch`.
- `fallthrough` valida y ejecuta el siguiente `case` de un `switch`.
- Es necesario siempre agregar un `default`.

Crea un `enum` con las estaciones del año. Agrega una variable `rangoTemperatura`, que corresponde a una tupla de 2 enteros, y una constante `estacionActual`, y asígnale cualquiera de las estaciones del `enum`. Agrega un `switch` que asigne a `rangoTemperatura` la temperatura máxima y mínima aproximada que se tiene en cada estación. Por último, imprime en consola dicho rango; por ejemplo, “En invierno la temperatura máxima es de 16 grados y la mínima de 3”.

- Verano: Máxima 28 y mínima 21.
- Otoño: Máxima 24 y mínima 7.
- Invierno: Máxima 15 y mínima 5.



- Primavera: Máxima 23 y mínima 14.