Swift 4

Varios

Optional chaining

Optional chaining

- Permite consultar y acceder a opcionales que puedan valer `nil`
- Si el opcional contiene un valor, el acceso tiene éxito
- Si no, devuelve `nil`
- Se pueden encadenar múltiples accesos y si algún acceso en la cadena es `nil`, toda la cadena devuelve `nil`
- Se diferencia del forced unwrapping en que no provoca un error en tiempo de ejecución

Optional chaining vs. forced unwrapping

```
class Person {
    var residence: Residence?
}

class Residence {
    var numberOfRooms = 1
}
```

Optional chaining vs. forced unwrapping

```swift
let john = Person()

let roomCount = john.residence!.numberOfRooms
// this triggers a runtime error
```

Optional chaining vs. forced unwrapping

```
if let roomCount = john.residence?.numberOfRooms {
    print("John's residence has \(roomCount) room(s).")
} else {
    print("Unable to retrieve the number of rooms.")
}
// Prints "Unable to retrieve the number of rooms."
```

Optional chaining vs. forced unwrapping

```
john.residence = Residence()

if let roomCount = john.residence?.numberOfRooms {
    print("John's residence has \(roomCount) room(s).")
} else {
    print("Unable to retrieve the number of rooms.")
}
// Prints "John's residence has 1 room(s)."
```

Modelo de clases de ejemplo

```swift
class Person {
    var residence: Residence?
}
```

Modelo de clases de ejemplo

```
class Residence {
    var rooms = [Room]()
    var numberOfRooms: Int {
        return rooms.count
    }
    subscript(i: Int) -> Room {
        get {
            return rooms[i]
        }
        set {
            rooms[i] = newValue
        }
    }
    func printNumberOfRooms() {
        print("The number of rooms is \(numberOfRooms)")
    }
    var address: Address?
}
```

Modelo de clases de ejemplo

```
class Room {
    let name: String
    init(name: String) { self.name = name }
}
```

Modelo de clases de ejemplo

```swift
class Address {
    var buildingName: String?
    var buildingNumber: String?
    var street: String?
    func buildingIdentifier() -> String? {
        if let buildingNumber = buildingNumber, let street = street {
            return "\(buildingNumber) \(street)"
        } else if buildingName != nil {
            return buildingName
        } else {
            return nil
        }
    }
}
```

Accediendo a propiedades

```
let john = Person()
if let roomCount = john.residence?.numberOfRooms {
    print("John's residence has \(roomCount) room(s).")
} else {
    print("Unable to retrieve the number of rooms.")
}
// Prints "Unable to retrieve the number of rooms."
```

Accediendo a propiedades

```swift
let someAddress = Address()
someAddress.buildingNumber = "29"
someAddress.street = "Acacia Road"
john.residence?.address = someAddress
```

Accediendo a propiedades

```
func createAddress() -> Address {
    print("Function was called.")

    let someAddress = Address()
    someAddress.buildingNumber = "29"
    someAddress.street = "Acacia Road"

    return someAddress
}
john.residence?.address = createAddress()
// residence vale nil, no se ejecuta la función
```

Llamando a métodos

```
if john.residence?.printNumberOfRooms() != nil {
    print("It was possible to print the number of rooms.")
} else {
    print("It was not possible to print the number of rooms.")
}
// Prints "It was not possible to print the number of rooms."
```

Comprobar el acceso a propiedades

```
if (john.residence?.address = someAddress) != nil {
    print("It was possible to set the address.")
} else {
    print("It was not possible to set the address.")
}
// Prints "It was not possible to set the address."
```

Utilizando subíndices

```swift
if let firstRoomName = john.residence?[0].name {
    print("The first room name is \(firstRoomName).")
} else {
    print("Unable to retrieve the first room name.")
}
// Prints "Unable to retrieve the first room name."
```

Utilizando subíndices

```
john.residence?[0] = Room(name: "Bathroom")
```

Utilizando subíndices

```
let johnsHouse = Residence()
johnsHouse.rooms.append(Room(name: "Living Room"))
johnsHouse.rooms.append(Room(name: "Kitchen"))
john.residence = johnsHouse

if let firstRoomName = john.residence?[0].name {
    print("The first room name is \(firstRoomName).")
} else {
    print("Unable to retrieve the first room name.")
}
// Prints "The first room name is Living Room."
```

Acceso a subíndices opcionales

```
var testScores = ["Dave": [86, 82, 84], "Bev": [79, 94, 81]]
testScores["Dave"]?[0] = 91
testScores["Bev"]?[0] += 1
testScores["Brian"]?[0] = 72
// the "Dave" array is now [91, 82, 84] and the "Bev" array is now [80, 94, 81]
```

Optional chaining con varios niveles

- Si el tipo al que estamos accediendo no es opcional, el resultado del optional chaining devolverá siempre un opcional

- Si ya es opcional, seguirá siéndolo, no se añaden "niveles de opcionalidad"

Optional chaining con varios niveles

```
if let johnsStreet = john.residence?.address?.street {
    print("John's street name is \(johnsStreet).")
} else {
    print("Unable to retrieve the address.")
}
// Prints "Unable to retrieve the address."
```

Optional chaining con varios niveles

```
let johnsAddress = Address()
johnsAddress.buildingName = "The Larches"
johnsAddress.street = "Laurel Street"
john.residence?.address = johnsAddress

if let johnsStreet = john.residence?.address?.street {
    print("John's street name is \(johnsStreet).")
} else {
    print("Unable to retrieve the address.")
}
// Prints "John's street name is Laurel Street."
```

Encadenando métodos que retornan opcionales

```swift
if let buildingIdentifier = john.residence?.address?.buildingIdentifier() {
    print("John's building identifier is \(buildingIdentifier).")
}
// Prints "John's building identifier is The Larches."
```

Encadenando métodos que retornan opcionales

```
if let beginsWithThe =
    john.residence?.address?.buildingIdentifier()?.hasPrefix("The") {
    if beginsWithThe {
        print("John's building identifier begins with \"The\".")
    } else {
        print("John's building identifier does not begin with \"The\".")
    }
}
// Prints "John's building identifier begins with "The"."
```

Gestión de errores

Gestión de errores

- Es el proceso de responder y recuperarse de errores en tiempo de ejecución

- Algunas operaciones (por ejemplo acceder a ficheros o conexiones de red) no garantizan que se complete la operación correctamente

- Nuestro programa debe ser capaz de reaccionar ante estas situaciones para resolverlas

- Si no puede resolverlas, debe comunicarle al usuario qué ha ocurrido de forma comprensible

Representar errores

- En Swift los errores se representan mediante tipos por valor que adoptan el protocolo `Error`
- Para modelar tipos de errores se usan enumeraciones

Representar errores

```swift
enum VendingMachineError: Error {
    case invalidSelection
    case insufficientFunds(coinsNeeded: Int)
    case outOfStock
}
```

Lanzar errores

```swift
throw VendingMachineError.insufficientFunds(coinsNeeded: 5)
```

Throwing functions

- Para indicar que una función, método o inicializador puede lanzar un error, la marcaremos con `throws` después de la lista de parámetros
- A este tipo de funciones se les llama *throwing functions*
- Si tienen valor de retorno, el throws va delante de la flecha (`->`)

Propagando errores

```swift
func canThrowErrors() throws -> String

func cannotThrowErrors() -> String
```

- Solo las *throwing functions* pueden propagar errores
- Los errores lanzados dentro de funciones que no usen `throws` deben gestionarse dentro de la función

Propagando errores

```swift
struct Item {
    var price: Int
    var count: Int
}

class VendingMachine {
    var inventory = [
        "Candy Bar": Item(price: 12, count: 7),
        "Chips": Item(price: 10, count: 4),
        "Pretzels": Item(price: 7, count: 11)
    ]
    var coinsDeposited = 0

    func vend(itemNamed name: String) throws {
        guard let item = inventory[name] else {
            throw VendingMachineError.invalidSelection
        }

        guard item.count > 0 else {
            throw VendingMachineError.outOfStock
        }

        guard item.price <= coinsDeposited else {
            throw VendingMachineError.insufficientFunds(coinsNeeded: item.price - coinsDeposited)
        }

        coinsDeposited -= item.price

        var newItem = item
        newItem.count -= 1
        inventory[name] = newItem

        print("Dispensing \(name)")
    }
}
```

Propagando errores

```swift
let favoriteSnacks = [
    "Alice": "Chips",
    "Bob": "Licorice",
    "Eve": "Pretzels",
]
func buyFavoriteSnack(person: String, vendingMachine: VendingMachine) throws {
    let snackName = favoriteSnacks[person] ?? "Candy Bar"
    try vendingMachine.vend(itemNamed: snackName)
}
```

Propagando errores

```swift
struct PurchasedSnack {
    let name: String
    init(name: String, vendingMachine: VendingMachine) throws {
        try vendingMachine.vend(itemNamed: name)
        self.name = name
    }
}
```

Gestión de errores con do-catch

```
do {
    try expresión
    sentencias
} catch patrón1 {
    sentencias
} catch patrón2 where condición {
    sentencias
}
```

Gestión de errores con do-catch

```
var vendingMachine = VendingMachine()
vendingMachine.coinsDeposited = 8
do {
    try buyFavoriteSnack(person: "Alice", vendingMachine: vendingMachine)
} catch VendingMachineError.invalidSelection {
    print("Invalid Selection.")
} catch VendingMachineError.outOfStock {
    print("Out of Stock.")
} catch VendingMachineError.insufficientFunds(let coinsNeeded) {
    print("Insufficient funds. Please insert an additional \(coinsNeeded) coins.")
}
// Prints "Insufficient funds. Please insert an additional 2 coins."
```

Convertir errores a opcionales

```
func someThrowingFunction() throws -> Int {
    // ...
}

let x = try? someThrowingFunction()

let y: Int?
do {
    y = try someThrowingFunction()
} catch {
    y = nil
}
```

Convertir errores a opcionales

```
func fetchData() -> Data? {
    if let data = try? fetchDataFromDisk() { return data }
    if let data = try? fetchDataFromServer() { return data }
    return nil
}
```

Desactivar la propagación de errores

```
let photo = try! loadImage(atPath: "./Resources/John Appleseed.jpg")
```

Especificar acciones de limpieza

- Podemos utilizar la sentencia `defer` para ejecutar sentencias justo antes de que termine el bloque actual
- Se ejecutará independientemente de cómo concluya el bloque, ya sea por un `throws`, `return` o `break`
- No puede contener sentencias de transferencia como `return`
- Se ejecutan en orden inverso a como las escribamos en el código fuente
- Las sentencias `defer` se pueden utilizar en cualquier ámbito, no son exclusivas de la gestión de errores

Especificar acciones de limpieza

```
func processFile(filename: String) throws {
    if exists(filename) {
        let file = open(filename)
        defer {
            close(file)
        }
        while let line = try file.readline() {
            // Work with the file.
        }
        // close(file) is called here, at the end of the scope.
    }
}
```

Conversión de tipos de datos

Conversión de tipos de datos

- Permite conocer el tipo de una instancia
- Permite manipular una instancia como si se tratara de una instancia de otro tipo diferente (normalmente de una subclase o de una superclase)
- Se utilizan los operadores `is` y `as`
- También permite comprobar si un tipo cumple un protocolo concreto

Jerarquía de clases de ejemplo

```
class MediaItem {
    var name: String
    init(name: String) {
        self.name = name
    }
}
```

Jerarquía de clases de ejemplo

```swift
class Movie: MediaItem {
    var director: String
    init(name: String, director: String) {
        self.director = director
        super.init(name: name)
    }
}

class Song: MediaItem {
    var artist: String
    init(name: String, artist: String) {
        self.artist = artist
        super.init(name: name)
    }
}
```

Jerarquía de clases de ejemplo

```
let library = [
    Movie(name: "Casablanca", director: "Michael Curtiz"),
    Song(name: "Blue Suede Shoes", artist: "Elvis Presley"),
    Movie(name: "Citizen Kane", director: "Orson Welles"),
    Song(name: "The One And Only", artist: "Chesney Hawkes"),
    Song(name: "Never Gonna Give You Up", artist: "Rick Astley")
]
// the type of "library" is inferred to be [MediaItem]
```

Operadores: conversión de tipos

Operador	Operación
`is`	Comprueba si una instancia es de una subclase concreta
`as?`	Devuelve un opcional del tipo de dato al que intentamos convertir Si no se puede hacer la conversión, devuelve `nil`
`as!`	Realiza la conversión y extrae el opcional resultante Si la conversión no es posible, genera una excepción

Comprobar el tipo (is)

```swift
var movieCount = 0
var songCount = 0

for item in library {
    if item is Movie {
        movieCount += 1
    } else if item is Song {
        songCount += 1
    }
}

print("Media library contains \(movieCount) movies and \(songCount) songs")
// Prints "Media library contains 2 movies and 3 songs"
```

Downcasting (as)

```swift
for item in library {
    if let movie = item as? Movie {
        print("Movie: \(movie.name), dir. \(movie.director)")
    } else if let song = item as? Song {
        print("Song: \(song.name), by \(song.artist)")
    }
}

// Movie: Casablanca, dir. Michael Curtiz
// Song: Blue Suede Shoes, by Elvis Presley
// Movie: Citizen Kane, dir. Orson Welles
// Song: The One And Only, by Chesney Hawkes
// Song: Never Gonna Give You Up, by Rick Astley
```

Any y AnyObject

- Son alias de tipos no específicos

- `Any` representa una instancia de cualquier tipo, incluso tipos de función

- `AnyObject` representa una instancia de cualquier clase

- Sólo se deben usar en casos puntuales, normalmente usaremos tipos de datos específicos

Any

```
var things = [Any]()

things.append(0)
things.append(0.0)
things.append(42)
things.append(3.14159)
things.append("hello")
things.append((3.0, 5.0))
things.append(Movie(name: "Ghostbusters", director: "Ivan Reitman"))
things.append({ (name: String) -> String in "Hello, \(name)" })
```

Any

```
for thing in things {
    switch thing {
    case 0 as Int:
        print("zero as an Int")
    case 0 as Double:
        print("zero as a Double")
    case let someInt as Int:
        print("an integer value of \(someInt)")
    case let someDouble as Double where someDouble > 0:
        print("a positive double value of \(someDouble)")
    case is Double:
        print("some other double value that I don't want to print")
    case let someString as String:
        print("a string value of \"\(someString)\"")
    case let (x, y) as (Double, Double):
        print("an (x, y) point at \(x), \(y)")
    case let movie as Movie:
        print("a movie called \(movie.name), dir. \(movie.director)")
    case let stringConverter as (String) -> String:
        print(stringConverter("Michael"))
    default:
        print("something else")
    }
}
```

Tipos de datos anidados

Tipos de datos anidados

- Swift permite anidar clases, estructuras y enumeraciones
- No hay límite en el número de niveles de anidamiento

Ejemplo: BlackjackCard

```swift
struct BlackjackCard {

    // nested Suit enumeration
    enum Suit: Character {
        case spades = "♠", hearts = "♡", diamonds = "♢", clubs = "♣"
    }

    // nested Rank enumeration
    enum Rank: Int {
        case two = 2, three, four, five, six, seven, eight, nine, ten
        case jack, queen, king, ace
        struct Values {
            let first: Int, second: Int?
        }
        var values: Values {
            switch self {
            case .ace:
                return Values(first: 1, second: 11)
            case .jack, .queen, .king:
                return Values(first: 10, second: nil)
            default:
                return Values(first: self.rawValue, second: nil)
            }
        }
    }

    // BlackjackCard properties and methods
    let rank: Rank, suit: Suit
    var description: String {
        var output = "suit is \(suit.rawValue),"
        output += " value is \(rank.values.first)"
        if let second = rank.values.second {
            output += " or \(second)"
        }
        return output
    }
}
```

Ejemplo: BlackjackCard

```swift
let theAceOfSpades = BlackjackCard(rank: .ace, suit: .spades)
print("theAceOfSpades: \(theAceOfSpades.description)")
// Prints "theAceOfSpades: suit is ♠, value is 1 or 11"

let heartsSymbol = BlackjackCard.Suit.hearts.rawValue
// heartsSymbol is "♡"
```

Seguridad en el acceso a memoria

Control de acceso