Módulo Swift [Swift intermedio]

Daniel Illescas Romero

KEEPCODING
Tech School

Índice

1. Funciones.

- Nos permiten reutilizar código para realizar una tarea específica múltiples veces.
- Permiten aceptar parámetros (o no) y devolver valores (o no).
- Usamos la palabra clave `func` para definir funciones libres o métodos.
- Las funciones tienen parámetros con nombre (*named parameters*).

```
func myFunction() {
    print("something")
}
myFunction()

func myFunction2(parameter1: String) { print(parameter1) }
myFunction2(parameter1: "something")

func myFunction3(_ parameter1: Int, parameter2 p2: Int) -> Int {
    return parameter1 + p2
}
myFunction3(10, parameter2: 30)
```

2. Enumeraciones.

- Una enumeración define un tipo común para un grupo de valores relacionados.
- Permite trabajar con valores de manera más segura (al trabajar con valores discretos).
- Los `case` pueden tener un valor de un mismo tipo (`raw value`) o un valor asociado a un caso particular (`associated values`).
- Pueden tener constructores personalizados, propiedades, métodos, implementar protocolos, etc.

```
enum Color: String { // aquí los `case` tienen un `RawValue` tipo `String`
    case blue, green
}
let color = Color.blue
color.rawValue // "blue"

enum Smartphone { // aquí hay varios `case` con valores asociados
    case android(model: String)
    case iOS(iOSSmartphone)
}
enum iOSSmartphone { case iPhoneX, iPhone14Pro } // aquí no hay raw value ni valores asociados
let smartPhone: Smartphone = .iOS(.iPhoneX) // Podemos omitir a veces el nombre del enum.
```

3. Estructuras para modelar datos.

`struct`	`class`
value type: sus instancias se copian.	***reference type***: sus instancias se referencian.
Constructor con parámetros automático.	Tenemos que crear constructores manualmente.
Implementación más sencilla de protocolos como `Equatable` y `Hashable`.	-
-	Herencia: puede heredar de otras clases.
Protocolos, métodos, constructores, `self`.	Protocolos, métodos, constructores, `self`.
-	Se puede definir un destructor.
Descripción del tipo en `String` de forma automática.	Podemos dar una descripción con `CustomStringConvertible`.

3.1 Estructuras (`struct`).

- Los usamos principalmente a la hora de representar información sin gran funcionalidad.
- **Son *value-type*: al cambiar valores en métodos lo marcamos como `mutating` y deberemos usar una variable en vez de constante**.
- Podemos declarar propiedades con `var` o `let` y usamos `self` para referirnos a la instancia actual.

```
struct Person: Equatable, Hashable {
    let name: String
    var age: Int
    func tellMyName() -> String { "My name is \(self.name)" }
    mutating func increaseAgeByOne() { self.age += 1 }
}
let daniel = Person(name: "Daniel", age: 123)
var goku = Person(name: "Goku", age: 41)
daniel.name
goku.increaseAgeByOne() //**IMPORTANTE**: necesita ser `var` para cambiar el valor
goku == daniel
daniel.hashValue
String(describing: daniel) // Person(name: "daniel", age: 123)
```

3.2. Clases (`class`).

- Usadas para representar procesos más complejos o transformar información de forma más sencilla.
- **Son *reference-type*, cuando una instancia se pase a una función o a otra variable, esta será referenciada en vez de copiada, esto significa que siempre mantendremos su mismo estado.**
- Tienen capacidad de heredar de otras clases, aunque es preferible el uso de protocolos.

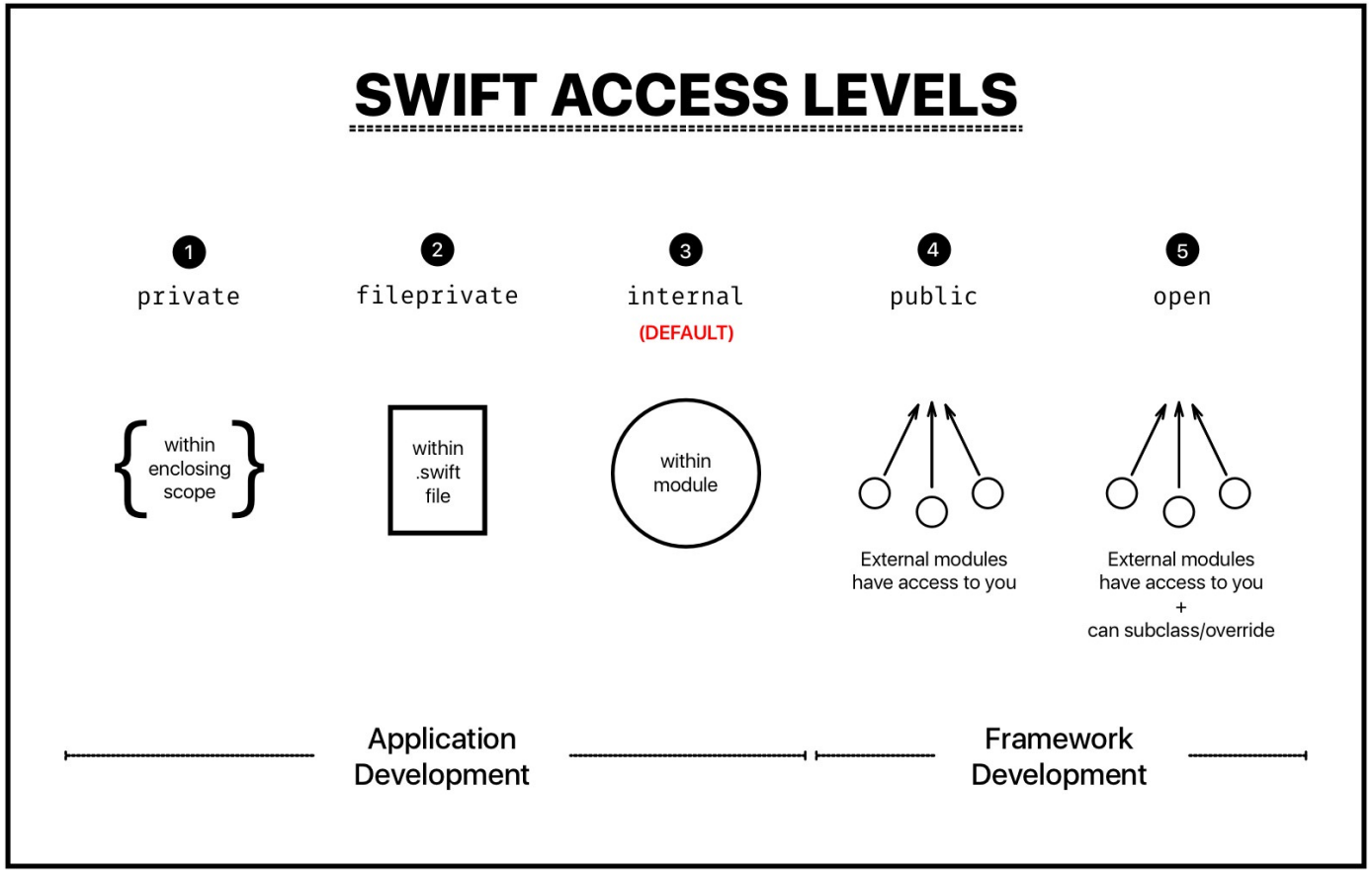
```
class RestaurantManager {
    private var waiters: [Waiter] = []
    private var customers: [Customer] = []
    // ...
    func serveTable(customer: Customer, food: Food) -> EstimatedTime {
        if self.customers.contains(customer) {
            self.customers.append(customer)
        }
        // ...
    }
}
// Fijaos que no hace falta `var` aunque al llamar a `serveTable` estemos modificando valores
let restaurantManager = RestaurantManager()
restaurantManager.serveTable(customer: customer, food: food)
```

3.3 Manejo de memoria (ARC).

Swift utiliza *Automatic Reference Counting* (ARC) para gestionar la memoria rastreando y desasignando instancias de clases que ya no son necesarias. Se asegura que las instancias permanezcan en memoria mientras se estén utilizando, y libera la memoria una vez que ya no se hacen referencias a ellas.

```
class MyClass {
    init() {
        print("inicializando")
    }
    deinit {
        print("deinicializando")
    }
}
var reference1: MyClass? = MyClass()
var reference2 = reference1
reference1 = nil
reference2 = nil // "deinicializando" se mostrará justo tras esta línea
```

3.4 Control de acceso.



https://dev-jiwon.github.io/swift-grammar-13/

3.5 Código genérico (*generics*).

El código genérico nos ayuda a escribir funciones y estructuras que puedan funcionar con los tipos que deseemos.

Si necesito una función que compare dos valores cualquiera:

```
func isEqual<T: Equatable>(value1: T, value2: T) -> Bool {
    return value1 == value2
}
```

Si necesito una estructura que almacene a su vez datos del tipo que sea:

```
struct OrderedSet<T: Hashable> {
    private var array: [T] = []
    private var set: Set<T> = []
    // ...
    mutating func add(_ value: T) {
        // ...
    }
}
```

4. Bibliografía y enlaces interesantes.

- *The Swift Programming Language*: https://docs.swift.org/swift-book/documentation/the-swift-programming-language
- ARC: https://docs.swift.org/swift-book/documentation/the-swift-programming-language/automaticreferencecounting/
- Imagen del control de acceso: https://dev-jiwon.github.io/swift-grammar-13/

KEEPCODING

Tech School

Madrid | Barcelona | Bogotá

Datos de contacto

Daniel Illescas Romero
email: daniel.illescas.r@gmail.com