Programación en Swift: Orientación a objetos

✅ ***Diploma:*** https://platzi.com/p/rubseg/curso/1791-swift-5-poo/diploma/detalle/

Estructuras y clases

Estructura vs Clases

Son construcciones flexibles de propósito general que se convierten en los componentes básicos del código de su programa.

Se define propiedades y métodos para agregar funcionalidad a sus estructuras y clases usando la misma sintaxis que usa para definir constantes, variables y funciones.

Las estructuras y clases en Swift tienen muchas cosas en común. Ambos pueden:

- Definir propiedades para almacenar valores.
- Definir métodos para proporcionar funcionalidad.
- Definir subíndices para proporcionar acceso a sus valores utilizando la sintaxis de subíndices
- Definir inicializadores para configurar su estado inicial.
- Ampliarse para ampliar su funcionalidad más allá de una implementación predeterminada.
- Cumplir con los protocolos para proporcionar una funcionalidad estándar de cierto tipo.

Las clases tienen capacidades adicionales que las estructuras no tienen:

- La herencia permite que una clase herede las características de otra.
- La conversión de tipos le permite verificar e interpretar el tipo de una instancia de clase en tiempo de ejecución.
- Los desinicializadores permiten que una instancia de una clase libere cualquier recurso que haya asignado.
- El recuento de referencias permite más de una referencia a una instancia de clase.

```
struct Resolution {
    var width = 0
    var height = 0
}

class VideoMode {
```

```
    var resolution = Resolution()
    var frameRate = 0.0
    var interlaced = false
    var name: String?
}

var iPhoneResolution = Resolution()
iPhoneResolution.height = 100
iPhoneResolution.width = 480

var someVideoMode = VideoMode()
someVideoMode.frameRate = 720
someVideoMode.interlaced = true
someVideoMode.name = "Fixed"
```

Estructuras: datos copiados por valor

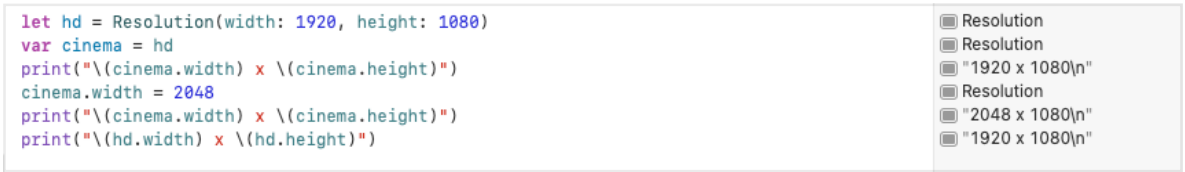
struct = copias, tienen su propio espacio de memoria
class = referencias, apuntan a un espacio de memoria

Podemos inicializarlas de la siguiente forma:

```
let vga = Resolution(width: 640, height: 480)
```

Y podemos ver la copia por valor entre estructuras, porque no tienen una referencia de donde proceden y donde van.

```
let hd = Resolution(width: 1920, height: 1080)
var cinema = hd
print("\(cinema.width) x \(cinema.height)")
cinema.width = 2048
print("\(cinema.width) x \(cinema.height)")
print("\(hd.width) x \(hd.height)")
```



Clases: datos referenciados

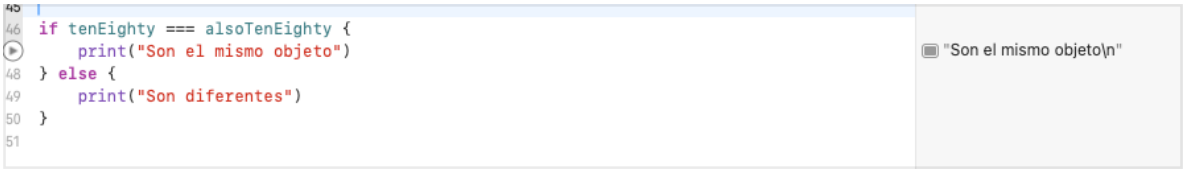
struct = copias, tienen su propio espacio de memoria
class = referencias, apuntan a un espacio de memoria

Al cambiar la copia cambia la estructura original, al ser valores por referencia cambia los 2 objetos.

Podemos comparar 2 objetos para ver si son exactamente igual con los operadores de identidad.

Operadores de identidad:

- *Idéntico a (===)*
- *No es idéntico a (! ==)*



Son el mismo objeto porque proceden el uno del otro

Tipos de propiedades

Las propiedades asocian valores con una clase, estructura o enumeración particular.

Las stored properties almacenan valores constantes y variables como parte de una instancia, mientras que las propiedades calculadas calculan (en lugar de almacenar) un valor.

Las computed properties las proporcionan clases, estructuras y enumeraciones. Las propiedades almacenadas las proporcionan únicamente clases y estructuras.

Las propiedades almacenadas y calculadas suelen estar asociadas con instancias de un tipo particular. Sin embargo, las propiedades también se pueden asociar con el tipo mismo. Estas propiedades se conocen como propiedades de tipo.

Además, puede definir observadores de propiedades para monitorear los

cambios en el valor de una propiedad, a los que puede responder con acciones personalizadas.

Los observadores de propiedades se pueden agregar a las propiedades almacenadas que usted mismo define y también a las propiedades que una subclase hereda de su superclase.

Stored properties

En su forma más simple, una propiedad almacenada es una constante o variable que se almacena como parte de una instancia de una clase o estructura particular.

Las propiedades almacenadas pueden ser propiedades almacenadas variables (introducidas por la palabra clave var) o propiedades almacenadas constantes (introducidas por la palabra clave let).

También puede establecer y modificar el valor inicial de una propiedad almacenada durante la inicialización. Esto es cierto incluso para propiedades almacenadas constantes LET, como se describe en Asignación de propiedades constantes durante la inicialización.

```
struct FixedLengthRange {
    var firstValue: Int
    let length: Int
}

var rangeOfThreeItems = FixedLengthRange(firstValue: 0,
length: 3)
rangeOfThreeItems.firstValue = 6
```

Lazy Stored Properties

No se crean estas propiedades (y su correspondiente espacio en memoria) hasta que no son utilizadas.

```
class DataImporter {
    var filename = "data.txt"
}

class DataManager {
    lazy var importer = DataImporter() // No se crea al
crear DataManager sino cuando es
necesario
}
```

Computed Properties

Se diferencia en que son variables calculadas por un algoritmo.

Con getter y setter.

```
struct Point {
    var x = 0.0, y = 0.0
}

struct Size {
    var width = 0.0, height = 0.0
}

struct Rect {
    var origin = Point()
    var size = Size()
    var center: Point { // esta es computed (calculada a
partir de otras props)
        get {
            let centerX = origin.x + size.width / 2
            let centerY = origin.y + size.height / 2
            return Point(x: centerX, y: centerY)
        }
        set(newCenter) {
            origin.x = newCenter.x - size.width / 2
            origin.y = newCenter.y - size.height / 2
        }
    }
}

var square = Rect(origin: Point(x: 0, y: 0), size:
Size(width: 10, height: 10))
square.center
let initialSquareCenter = square.center
square.center = Point(x: 20, y: 20)
square.center
```

Computed Properties de sólo lectura

Sólo tiene getter, no puedes modificar su valor.

```
struct Cuboid {
  var width = 0.0, height = 0.0, depth = 0.0
  var volume: Double {
    return width * height * depth
  }
}
```

```
let cuboidFive: Cuboid = Cuboid(width: 5, height: 5, depth: 5)
print("Cuboid 5 volume = \(cuboidFive.volume)")
```

Property Observers

Observan y reaccionan a los cambios en el valor de una propiedad.

Se llama a los property observer cada vez que se establece el valor de una propiedad, incluso si el nuevo valor es el mismo que el valor actual de la propiedad.

2 tipos:

- Willset -> futuro: se llamará justo ANTES de cambiar el valor de una property.
- Didset -> pasado: se llamará justo DESPUÉS de cambiar el valor de una property.

```
class StepCounter {
    var totalSteps: Int = 0 {
        willSet(newTotalSteps) { // se ejecutará
automáticamente antes de que vaya a cambiar
            print("El número de pasos va a subir hasta \
(newTotalSteps)")
        }
        didSet {
            if totalSteps > oldValue {
                print("El número de pasos ha incrementado en
\(totalSteps - oldValue)")
            }
        }
    }
}
```

```
let stepCounter = StepCounter()
stepCounter.totalSteps = 253
stepCounter.totalSteps = 37
stepCounter.totalSteps *= 100
```

```
Cuboid 5 volume = 125.0
El número de pasos va a subir hasta 253
El número de pasos ha incrementado en 253
El número de pasos va a subir hasta 37
El número de pasos va a subir hasta 3700
El número de pasos ha incrementado en 3663
```

Métodos, subíndices y herencia

Los métodos son funciones que están asociadas con un tipo particular.

Las clases, estructuras y enumerados pueden definir métodos de instancia, que encapsulan tareas y funcionalidades específicas para trabajar con una instancia de un tipo determinado.

En Swift, puede elegir si desea definir una clase, estructura o enumeración y aún tener la flexibilidad de definir métodos en el tipo que cree.

La propiedad self

Cada instancia de un tipo tiene una propiedad implícita llamada self, que es exactamente equivalente a la instancia misma.

Utilice la propiedad self para hacer referencia a la instancia actual dentro de sus propios métodos de instancia.

```
func increment() {
    self.count += 1
}
```

Métodos de instancia

Los métodos de instancia son funciones que pertenecen a instancias de una clase, estructura o enumeración particular.

Admiten la funcionalidad de esas instancias, ya sea proporcionando formas de acceder y modificar las propiedades de la instancia, o proporcionando funcionalidad relacionada con el propósito de la instancia.

Un método de instancia sólo se puede llamar en una instancia específica del tipo al que pertenece. No se puede llamar de forma aislada sin una instancia existente.

```
class Counter {
    var count = 0

    func increment() {
        self.count += 1
    }

    func increment(by amount: Int) {
        self.count += amount
    }

    func reset() {
        self.count = 0
    }
}

let counter = Counter()
counter.increment()
counter.increment(by: 5)
counter.reset()
```

Mutating Methods

Para poder modificar una propiedad desde un método de una estructura o enumerado el método debe ser marcado como mutating.

```
struct Point {
    var x = 0.0, y = 0.0

    func isTotheRight(of x: Double) -> Bool {
        return self.x > x
    }

    mutating func moveBy(x deltaX: Double, y deltaY: Double)
{
        x += deltaX
        y += deltaY
    }
}

var somePoint = Point(x: 2, y: 32)
somePoint.moveBy(x: 2, y: -20)
```

Métodos de clase

Parecidos a los Properties.

En lugar de ser invocado en instancia, será la propia clase la que puede invocarlos.

Son métodos estáticos.

```
class SomeClass {
    static func someMethod() {
        print("Ey someMethod!")
    }
}

SomeClass.someMethod()
```

Herencia

Una clase puede heredar métodos, propiedades de otra clase.

Cuando una clase hereda de otra, la que hereda se la llama **subclase** y de la clase de la que hereda se la llama **superclase**.

Cuando heredamos de una clase podemos sobrescribir sus métodos y propiedades. También podemos añadir **property observers** en las propiedades heredadas para "escuchar" cuando el valor de la propiedad cambia.

```
class Vehicle {
    var currentSpeed = 0.0
    var description: String {
        return "Viajando a \(currentSpeed) km/h"
    }

    func makeNoise() {
        // do nothing, cada vehículo cado vehículo tiene su
propio ruido
    }
}

let someVehicle = Vehicle()
someVehicle.description

class Bicycle : Vehicle {
    var hasBasket = false
}
```

```
let bicycle = Bicycle()
bicycle.hasBasket = true
bicycle.currentSpeed = 8
bicycle.description

class Tandem : Bicycle {
    var currentNumberOfPasengers = 0
}

var tandem = Tandem()
tandem.hasBasket = true
tandem.currentNumberOfPasengers = 2
tandem.currentSpeed = 14
tandem.description
```

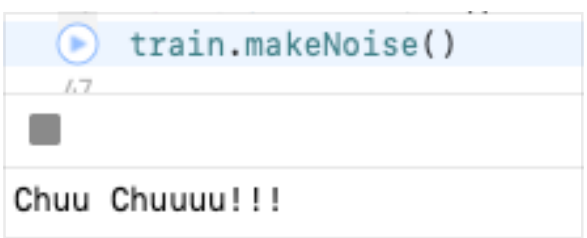
Sobrescritura de variables y métodos

Una subclase puede proporcionar su propia implementación personalizada de un método de instancia, método de tipo, propiedad de instancia, propiedad de tipo o subíndice que de otro modo heredaría de una superclase.

Esto se conoce como sobrescritura.

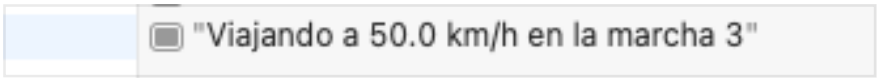
Se utiliza la palabra **override** anteponiendo a la variables o método que queremos sobrescribir.

```
class Train: Vehicle {
    var numberOfWagons = 0

    override func makeNoise() {
        print("Chuu Chuuuu!!!")
    }
}

var train = Train()
train.makeNoise()
```




```
class Car : Vehicle {
    var gear = 1
```

```
    override var description: String {
        return super.description + " en la marcha \(gear)"
    }
}

let car = Car()
car.currentSpeed = 50
car.gear = 3
car.description
```



Inicializadores o Constructores

La inicialización es el proceso de preparar una instancia de una clase, estructura o enumeración para su uso.

Este proceso implica establecer un valor inicial para cada propiedad almacenada en esa instancia y realizar cualquier otra configuración o inicialización necesaria antes de que la nueva instancia esté lista para su uso.

ste proceso de inicialización se implementa definiendo inicializadores, que son como métodos especiales que se pueden llamar para crear una nueva instancia de un tipo particular.

Las instancias de tipos de clase también pueden implementar un desinicializador, que realiza cualquier limpieza personalizada justo antes de que se desasigne una instancia de esa clase.

Inicializadores

Se llama a los inicializadores para crear una nueva instancia de un tipo particular. En su forma más simple, un inicializador es como un método de instancia sin parámetros, escrito usando la palabra clave init.

```
struct Fahrenheit {
    var temperature: Double

    init() {
        self.temperature = 32
    }

    init(fromFarenheit fahrenheit: Double) {
        self.temperature = (fahrenheit - 32) / 1.8
    }
```

```
    init(fromKelvin kelvin: Double) {
        self.temperature = kelvin - 273.15
    }

}

var f1 = Fahrenheit()
var f2 = Fahrenheit(fromFarenheit: 15)
var f3 = Fahrenheit(fromKelvin: 287)
```

Nombres, etiquetas y optionals

Si no desea utilizar una etiqueta de argumento para un parámetro inicializador, escriba un guión bajo (_) en lugar de una etiqueta de argumento explícita para ese parámetro para anular el comportamiento predeterminado.

```
init(_ celsius: Double) {
        self.temperature = celsius
 }

let bodyTemperature = Fahrenheit(37)
```

Al igual que con los parámetros de función y método, los parámetros de inicialización pueden tener tanto un nombre de parámetro para usar dentro del cuerpo del inicializador como una etiqueta de argumento para usar al llamar al inicializador.

```
struct Color {
    let red, green, blue: Double
    init(red: Double, green: Double, blue: Double) {
        self.red = red
        self.green = green
        self.blue = blue
    }
    init(white: Double) {
        self.red = white
        self.green = white
        self.blue = white
    }
}

let magenta = Color(red: 1, green: 0, blue: 1)
let halfGrey = Color(white: 0.5)
```

Las propiedades de tipo opcional se inicializan automáticamente con un valor

nulo, lo que indica que la propiedad está destinada deliberadamente a "no tener valor todavía" durante la inicialización.

Inicialización en subclases

A todas las propiedades almacenadas de una clase, incluidas las propiedades que la clase hereda de su superclase, se les debe asignar un valor inicial durante la inicialización.

Swift define dos tipos de inicializadores para tipos de clases para ayudar a garantizar que todas las propiedades almacenadas reciban un valor inicial. Estos se conocen como inicializadores designados e inicializadores de conveniencia.

Los inicializadores designados son los inicializadores principales de una clase. Un inicializador designado inicializa completamente todas las propiedades introducidas por esa clase y llama a un inicializador de superclase apropiado para continuar el proceso de inicialización en la cadena de superclases.

Los inicializadores de conveniencia son secundarios y admiten inicializadores para una clase. Puede definir un inicializador de conveniencia para llamar a un inicializador designado de la misma clase que el inicializador de conveniencia con algunos de los parámetros del inicializador designado establecidos en valores predeterminados. También puede definir un inicializador conveniente para crear una instancia de esa clase para un caso de uso específico o un tipo de valor de entrada.

Designado -> desigando super clase
Conveniencia -> Otro init de la misma clase
El último init que se llame siempre debe ser designado

```
class Vehicle {
    var numberOfWheels = 0
    var description: String {
```

```
        return "\(numberOfWheels) ruedas"
    }
}

let vehicle = Vehicle()
vehicle.description

class Bicycle: Vehicle {
    override init() {
        super.init() // corre el init del padre
        numberOfWheels = 2 // definimos el nuestro para
sobrescribir
    }
}

let bicycle = Bicycle()
bicycle.description

class Hoverboard: Vehicle {
    var color: String
    init(color: String) {
        self.color = color // esto es inicializador por
conveniencia
        // aqui se llama implícitamente a super.init()
    }
    override var description: String {
        return "\(super.description) en el color \(color)"
    }
}

let hoverboard =  Hoverboard(color: "silver")
hoverboard.description
```

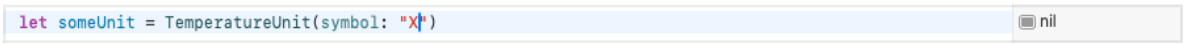
Failable Initializer

A veces resulta útil definir una clase, estructura o enumeración cuya inicialización puede fallar. Este error puede deberse a valores de parámetros de inicialización no válidos, la ausencia de un recurso externo requerido o alguna otra condición que impida que la inicialización se realice correctamente.

Para hacer frente a las condiciones de inicialización que pueden fallar, defina uno o más inicializadores fallidos como parte de una definición de clase, estructura o enumeración. Para escribir un inicializador fallido, coloque un signo de interrogación después de la palabra clave init (¿init?).

```
enum TemperatureUnit {
    case kelvin, celsius, fahrenheit
```

```
    init? (symbol: Character) {
        switch symbol {
            case "K":
                self = .kelvin
            case "C":
                self = .celsius
            case "F":
                self = .fahrenheit
            default:
                return nil
        }
    }
}
```



Destrucción de objetos con deinit

Se llama a un desinicializador inmediatamente antes de que se desasigne una instancia de clase. Los desinicializadores se escriben con la palabra clave deinit, de forma similar a cómo se escriben los inicializadores con la palabra clave init.

Los desinicializadores solo están disponibles en tipos de clases.

```
class Bank {
    static var coinsInBank = 1_000
        static func distribute(coins numberOfCoinsRequested:
Int) -> Int {
            let numberOfCoinsToVend =
min(numberOfCoinsRequested, coinsInBank)
            coinsInBank -= numberOfCoinsToVend
            return numberOfCoinsToVend
        }
        static func receive(coins: Int) {
            coinsInBank += coins
        }
}

class Player {
    var coinsInPurse: Int
    init(coins: Int) {
        coinsInPurse = Bank.distribute(coins: coins)
    }
    func win(coins: Int) {
        coinsInPurse += Bank.distribute(coins: coins)
```

```
    }
    deinit {
        Bank.receive(coins: coinsInPurse)
    }
}

var playerOne: Player? = Player(coins: 100)

Bank.coinsInBank
playerOne!.win(coins: 2000)
Bank.coinsInBank
playerOne = nil
Bank.coinsInBank
```

Encadenamiento opcional

Optional Chaining

Opcional Chaining es un proceso para consultar y llamar propiedades, métodos y subíndices en un opcional que actualmente podría ser nulo.

Si el opcional contiene un valor, la llamada a la propiedad, método o subíndice se realiza correctamente; si el opcional es nulo, la llamada a la propiedad, método o subíndice devuelve nulo.

Se pueden encadenar varias consultas y toda la cadena falla correctamente si algún eslabón de la cadena es nulo.