Tema 6: POO con Swift

POO

- Uso de **objetos** o instancias de una **clase** con atributos y conducta.
- **Herencia**
- **Polimorfismo** de las funciones
- **Dispach dinámico** en invocación de funciones. El obj. determina qué código se ejecuta en tiempo de ejecución.

Estructuras vs clases

- **Tipo valor** vs **tipo referencia**
- Las **clases** usan **herencia, casting de tipos** (comprueban tipo del obj. en TE) y **deinicializadores** (liberan recursos).

EJ:

```
struct CoordsPantalla {
    var posX = 0
    var posY = 0
}

class Ventana {
    var esquina = CoordsPantalla()
    var altura = 0
    var anchura = 0
    var visible = true
    var etiqueta: String?
}
```

Instancias de clases y estructuras

```
var unasCoordsPantalla = CoordsPantalla()
var unaVentana = Ventana()
```

❏ Se ha usado un **inicializador por defecto** al no definir un inicializador explícito.

Acceso a propiedades

```
// Accedemos a la propiedad con punto
unasCoordsPantalla.posX // Devuelve 0
// Actualizamos la propiedad
unasCoordsPantalla.posX = 100
```

Inicialización de **estructuras** por sus propiedades

- **Inicializador memberwise** en el que proporcionar valores de sus propiedades.
- Podemos **omitir** las propiedades definidas por defecto u opcionales.

EJ:
```
let coords1 = CoordsPantalla(posX: 200)
print(coords1.posX, coords1.posY)
// Imprime 200 0
```

Declaración de instancias de estructuras y clases con let

- Las propiedades y la instancia de la estructura serán constantes, no podrán modificarse.

Ej:
```
let coords3 = CoordsPantalla(posX: 400, posY: 400)
coords3.posX = 800// error: 'coords3' is a 'let' constant
```

- La instancia de la clase será constante, las propiedades pueden modificarse.

Ej:
```
let ventana3 = Ventana()
// Sí que podemos modificar una propiedad de la instancia:
ventana3.etiqueta = "Listado"
// Pero no podemos reasignar la variable:
ventana3 = ventana1
// error: cannot assign to value: 'ventana3' is a 'let' constant
```

Operadores de identidad en clases

- “Idéntico a” (===) o “no idéntico a” (!==) -> ¿Misma instancia de una clase?
- “Igual a” (==) o “diferente a” (!=) -> ¿Dos instancias son iguales en su valor?
  Esta última es responsabilidad del programador implementarla.

Paso como parámetro

- Los parámetros de las funciones son constantes, se definen usando el operador let. Esto hace que haya un comportamiento diferente al pasar estructuras o clases.

EJ: pasamos una clase y modificamos sus propiedades
```
func mueve(ventana: Ventana, incX: Int, incY: Int) {
    var nuevaPos = CoordsPantalla()
    nuevaPos.posX = ventana.esquina.posX + incX
    nuevaPos.posY = ventana.esquina.posY + incY
    ventana.esquina = nuevaPos
}
```

```
EJ: pasamos una estructura no podemos modificar prop., devolvemos nueva
estructura.
func mueve(coordsPantalla: CoordsPantalla, incX: Int, incY: Int) ->
CoordsPantalla {
    var nuevaCoord = CoordsPantalla()
    nuevaCoord.posX = coordsPantalla.posX + incX
    nuevaCoord.posY = coordsPantalla.posY + incY
    return nuevaCoord
}
```

¿Cuando usamos clases?

- Cuando un objeto necesite referenciar a otro.
- Cuando nuestro objeto necesite heredar características de otro objeto.
- Cuando nuestro objeto use un comportamiento diferente de una función respecto a otros tipos de objeto.

¿Cuando usamos estructuras?

- Necesitamos encapsular pocos datos sencillos.
- Esperamos que los datos encapsulados serán copiados al pasar una instancia.
- No necesitamos heredar propiedades o funciones de otros tipos de objetos.

Propiedades

- Asocian valores con una estructura, clase o enumeración.
- Las enumeraciones solo contienen propiedades calculadas.
- Las clases y estructuras pueden contener prop. calculadas y almacenadas.

Propiedades almacenadas

- **Constante** (let) o **variable** (var) **almacenada** como parte de una **instancia** de una clase o estructura particular.
- Podemos dar un **valor por defecto**

Ej:

```
struct RangoLongitudFija {
    var primerValor: Int
    let longitud: Int
}
var rangoDeTresItemss = RangoLongitudFija(primerValor: 0, longitud: 3)
// el rango representa ahora valores enteros the range represents integer
values 0, 1, and 2
rangoDeTresItemss.primerValor = 6
// el rango representa ahora valores enteros 6, 7 y 8
rangoDeTresItemss.longitud = 4
//Error, no puede reasignarse la constante “let longitud: Int”
```

Propiedades calculadas

- Usan un **getter** y un opcional **setter** que devuelven y modifican otras propiedades de forma indirecta.

Ej:

```
struct Rectangulo {
    var origen = Punto()      //Struct con variables x e y
    var tamaño = Tamaño()           //Struct con variables ancho y alto
    var centro: Punto {             //Obtiene y devuelve otro punto
        get {
            let centroX = origen.x + (tamaño.ancho / 2)
            let centroY = origen.y + (tamaño.alto / 2)
            return Punto(x: centroX, y: centroY)
        }
        set(centroNuevo) {          //Si usamos newValue se omite el (c)
            origen.x = centroNuevo.x - (tamaño.ancho / 2)
            origen.y = centroNuevo.y - (tamaño.alto / 2)
        }
    }
}
var cuadrado = Rectangulo(origen: Punto(x: 0.0, y: 0.0),
                  tamaño: Tamaño(ancho: 10.0, alto: 10.0))
```

```
let centroCuadradoInicial = cuadrado.centro
cuadrado.centro = Punto(x: 15.0, y: 15.0)
print("cuadrado.origen está ahora en (\(cuadrado.origen.x),
\(cuadrado.origen.y))")
// Prints "cuadrado.origen está ahora en (10.0, 10.0)"
```

Observadores de propiedades

- Observan y responden a cambios en el valor de una propiedad.
- Se llaman cada vez que el valor de una **propiedad almacenada** es actualizado.
- Podemos añadir alguno o ambos de los observadores:
  - **willSet** -> **antes** de que el valor se modifique
  - **didSet** -> **después** de que el valor se modifique

Ej:

```
class ContadorPasos {
    var totalPasos: Int = 0 {
        willSet(nuevoTotalPasos) {        //Si usamos newValue omitimos ()
            print("A punto de actualizar totoalPasos a \(nuevoTotalPasos)")
        }
        didSet {                          //Podemos usar (nuevosPasos)
            if totalPasos > oldValue  {
                print("Añadidos \(totalPasos - oldValue) pasos")
            }
        }
    }
}
let contadorPasos = ContadorPasos()
contadorPasos.totalPasos = 200
// Imprime: "A punto de actualizar totalPasos a 200"
// Imprime: "Añadidos 200 pasos"
```

Variables locales y globales

- Pueden definirse como propiedades calculadas
- También pueden tener observadores

Ej:

```
var x = 10  {
   didSet {
      print("El nuevo valor: \(x) y el valor antiguo: \(oldValue)")
   }
}
var y = 2
var z : Int {
   get {
      return x + y
   }
   set {
      x = newValue / 2
      y = newValue / 2
```

```
    }
}
z = 100
//"El nuevo valor: 50 y el valor antiguo: 10"
```

Propiedades del tipo

- Usamos **static** para definirlas.
- No pertenecen a las instancias, sino al tipo.
- Una única copia
- Deben iniciarse por defecto
- Pueden ser variables (var) o constantes (let).

Ej:

```
struct UnaEstructura {
    static var almacenada = "A"
    static var calculada : Int {
        return 1
    }
}
enum UnaEnumeracion {
    static var almacenada = "A"
    static var calculada: Int {
        return 1
    }
}
class UnaClase {
    static var almacenada = "A"
    static var calculada: Int {
        return 1
    }
}
UnaEstructura.almacenada // devuelve "A"
UnaEnumeracion.almacenada = "B"
UnaClase.calculada // devuelve 1
```

Métodos

- Clases, estructuras y enumeraciones pueden definir métodos de instancia o de tipo.

Métodos de instancia

```
class Contador {
    var veces = 0
    func incrementa() {
        veces += 1
    }
    func incrementa(en cantidad: Int) {
        veces += cantidad
    }
    func reset() {
        veces = 0
    }
}
let contador = Contador()
// el valor inicial del contador es 0
contador.incrementa()
// el valor del contador es ahora 1
contador.incrementa(en: 5)
// el valor del contador es ahora 6
contador.reset()
// el valor del contador es ahora 0
```

La propiedad self

- Para referirnos a la instancia actual dentro de sus propios métodos de instancia.

EJ:

```
struct Punto {
    var x = 0.0, y = 0.0
    func estaAlaDerecha(de x: Double) -> Bool {
        return self.x > x
    }
}
```

Operaciones con instancias de tipo valor

- Las propiedades de una estructura o enumeración no pueden ser modificadas desde dentro de los métodos de instancia.

Ej:

```
struct Punto {
    var x = 0.0, y = 0.0
    func incrementa(incX: Double, incY: Double) -> Punto {
```

```
        return Punto(x: x+incX, y: y+incY)
    }
}
let unPunto = Punto(x: 1.0, y: 1.0)
var puntoMovido = unPunto.incrementa(incX: 2.0, incY: 3.0)
```

Modificación de tipos valor desde dentro de la instancia

- Podemos conseguir un método mutador colocando la palabra clave **mutating** antes de la palabra func del método:

```
struct Punto {
    var x = 0.0, y = 0.0
    mutating func incrementado(incX: Double, incY: Double) {
        x += incX
        y += incY
    }
}
var unPunto = Punto(x: 1.0, y: 1.0)
unPunto.incrementado(incX: 2.0, incY: 3.0)
print("El punto está ahora en (\(unPunto.x), \(unPunto.y))")
// Imprime "El punto está ahora en (3.0, 4.0)"
```

Asignación a self en un método mutador

- Podemos asignar a self una nueva instancia del tipo valor.

```
struct Punto {
    var x = 0.0, y = 0.0
    mutating func incrementa(incX: Double, incY: Double) {
        self = Punto(x: x + incX, y: y + incY)
    }
}
*** Este método es menos eficiente respecto al anterior apartado. ***
```

Métodos del tipo

- Escribiendo la palabra clave `static` antes de la palabra clave `func` del método.

EJ:

```
class NuevaClase {
    static func unMetodoDelTipo() {
        print("Hola desde el tipo")
    }
}
var clase1 = NuevaClase()
clase1.unMetodoDelTipo()      //Error
NuevaClase.unMetodoDelTipo()
```

Inicializadores

- Métodos que pueden llamarse para crear una nueva instancia de un tipo particular.

Inicializadores por defecto y memberwise

Ej: ini. por defecto

```
struct Punto2D {
    var x = 0.0
    var y = 0.0
}
class Segmento {
    var p1 = Punto2D()
    var p2 = Punto2D()
}

var s = Segmento()
```

Ej: ini. por memberwise

```
var p = Punto2D(x: 10.0, y: 10.0)
```

Inicialización de propiedades almacenadas

- Se escribe con la palabra clave **init**
- Se deben definir todas sus propiedades almacenadas a un valor inicial, y los opcionales a nil.

Ej:

```
struct Fahrenheit {
    var temperatura: Double
    init() {
        temperatura = 32.0
    }
}
var f = Fahrenheit()
print("La T es \(f.temperatura)") //Imprime "La T es 32.0°"
```

Inicializador personalizado

- Se proporcionan **parámetros** de inicialización como parte de la definición.

Ej:

```
class PreguntaEncuesta {
    let texto: String
    var respuesta: String?
    init(texto: String) {
        self.texto = texto
```

```
    }
    func pregunta() {
        print(texto)
    }
}
let preguntaQueso = PreguntaEncuesta(texto: "¿Te gusta el queso?")
preguntaQueso.pregunta() // -> "¿Te gusta el queso?
preguntaQueso.respuesta // -> nil
```

Herencia

- Las subclases **heredan propiedades y métodos** de las superclases.
- Las subclases pueden **sobrescribir propiedades o métodos** heredados.
- Las subclases pueden **añadir observadores** a las **propiedades heredadas**.

Definición de una clase base

- Aquella que no hereda de ninguna clase

Ej:

```
class Vehiculo {
    var velocidadActual = 0.0
    var descripcion: String {
        return "viajando a \(velocidadActual) kilómetros por hora"
    }
    func hazRuido() {
    }
}
let unVehiculo = Vehiculo()
```

Construcción de subclases

Ej:

```
class Bicicleta: Vehiculo {
    var tieneCesta = false
}

let bicicleta = Bicicleta()
bicicleta.tieneCesta = true
```

Sobreescritura

- Proporcionar su **propia implementación** de un método de la instancia, método del tipo, propiedad de la instancia o propiedad del tipo que hereda.
- Debemos usar el prefijo **override**
- Para acceder a los **valores de superclase** podemos usar el prefijo **super**

- Podemos **evitar** que una **clase, función o método sea sobrescrito** con **final**

Ej:
```
class Tren: Vehiculo {
    override func hazRuido() {
        print("Chuu Chuu")
    }
}
let tren = Tren()
tren.hazRuido()
// Imprime "Chuu Chuu"

class Coche: Vehiculo {
    var marcha = 1
    override var descripcion: String {
        return super.descripcion + " con la marcha \(marcha)"
    }
}
let coche = Coche()
coche.velocidadActual = 50.0
coche.marcha = 3
print("Coche: \(coche.descripcion)")
// Coche: viajando a 50.0 kilómetros por hora con la marcha 3
```

Ej: Añadir observador a la propiedad de la superclase
```
class CocheAutomatico: Coche {
    override var velocidadActual: Double {
        didSet {
            marcha = min(Int(velocidadActual / 25.0) + 1, 5)
        }
    }
}
```

Ej: Clase que no puede sobrescribirse
```
final class SomeClass {
    static func someTypeMethod() {
        print("Método de tipo o método estático!!!")
    }
}
SomeClass.someTypeMethod()
```

Bibliografía

Los enunciados de los ejercicios resueltos, y los resúmenes, se han elaborado a partir del material publicado en https://domingogallardo.github.io/ , material del que es propietario el Departamento de Ciencia de la Computación e Inteligencia Artificial de la Universidad de Alicante, Domingo Gallardo, Cristina Pomares, Antonio Botía y Francisco Martínez.