Swift 6

Tipos de datos personalizados

Enumeraciones

Enumeraciones

- Definen un tipo de dato para un grupo de valores relacionados
- En Swift, permiten asignar valores a los elementos (valores asociados) de cualquier tipo o predeterminarlos (valores raw)
- Pueden tener métodos de instancia, propiedades calculadas, inicializadores, se pueden extender y pueden adoptar protocolos

Definición de una enumeración

```
enum CompassPoint {
    case north
    case south
    case east
    case west
}

enum Planet {
    case mercury, venus, earth, mars, jupiter, saturn, uranus, neptune
}
```

Uso de una enumeración

```
var directionToHead = CompassPoint.west

directionToHead = .east
```

Comparando con switch

```
directionToHead = .south

switch directionToHead {
case .north:
    print("Lots of planets have a north")
case .south:
    print("Watch out for penguins")
case .east:
    print("Where the sun rises")
case .west:
    print("Where the skies are blue")
}

// Prints "Watch out for penguins
```

Comparando con switch

```
let somePlanet = Planet.earth

switch somePlanet {
case .earth:
    print("Mostly harmless")
default:
    print("Not a safe place for humans")
}

// Prints "Mostly harmless
```

Valores asociados

```
enum Barcode {
    case upc(Int, Int, Int, Int)
    case qrCode(String)
}
```


```
var productBarcode = Barcode.upc(8, 85909, 51226, 3)

productBarcode = .qrCode("ABCDEFGHIJKLMNOP")
```

Extraer valores asociados

```
switch productBarcode {
case .upc(let numberSystem, let manufacturer, let product, let check):
    print("UPC: \(numberSystem), \(manufacturer), \(product), \(check).")
case .qrCode(let productCode):
    print("QR code: \(productCode).")
}

// Prints "QR code: ABCDEFGHIJKLMNOP.
```

Extraer valores asociados

```
switch productBarcode {
case let .upc(numberSystem, manufacturer, product, check):
    print("UPC : \(numberSystem), \(manufacturer), \(product), \(check).")
case let .qrCode(productCode):
    print("QR code: \(productCode).")
}

// Prints "QR code: ABCDEFGHIJKLMNOP.
```

Valores raw

- Son valores asignados por defecto a los elementos de la enumeración
- El valor raw de un elemento no puede variar
- Pueden ser de tipo string, carácter o de tipos enteros o coma flotante
- Si se usan enteros y no se define algún valor, autoincrementan (valores raw implícitos)
- No son lo mismo que los valores asociados

Valores raw

```
enum ASCIIControlCharacter: Character {
    case tab = "\t"
    case lineFeed = "\n"
    case carriageReturn = "\r"
}


enum Planet: Int {
    case mercury = 1, venus, earth, mars, jupiter, saturn, uranus, neptune
}


enum CompassPoint: String {
    case north, south, east, west
}
```

Valores raw implícitos

```
let earthsOrder = Planet.earth.rawValue
// earthsOrder is 3

let sunsetDirection = CompassPoint.west.rawValue
// sunsetDirection is "west"
```

Inicialización desde raw

```
let possiblePlanet = Planet(rawValue: 7)
// possiblePlanet is of type Planet? and equals Planet.uranus
```

Clases y estructuras

Clases y estructuras

- Serán los bloques de construcción de nuestros programas
- Sus características son muy similares

Clases y estructuras

	Clases	Estructuras
Propiedades	✔	✔
Métodos	✔	✔
Subíndices	✔	✔
Inicializadores	✔	✔
Extensiones	✔	✔
Protocolos	✔	✔
Herencia	✔	✘
Conversión de tipo	✔	✘
Desinicializadores	✔	✘
ARC	✔	✘

¿Clase o estructura?

Criterio	Tipo
Encapsular sólo unos pocos valores simples	Estructura
Se espera que los valores que contiene se copien al asignarlos o pasarlos a funciones	Estructura
Las propiedades que contiene también son tipos por valor y se espera que se copien	Estructura
No necesita heredar propiedades o métodos de otros tipos	Estructura
Resto de situaciones (como norma general)	Clase

Definición

```
class SomeClass {
    // class definition goes here
}


struct SomeStructure {
    // structure definition goes here
}
```

Definición

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


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

Instanciación

```
let someResolution = Resolution()
let someVideoMode = VideoMode()
```

Acceso a propiedades

```
print("The width of someResolution is \(someResolution.width)")

print("The width of someVideoMode is \(someVideoMode.resolution.width)")

someVideoMode.resolution.width = 1280
print("The width of someVideoMode is now \(someVideoMode.resolution.width)")
```

Inicializador miembro a miembro para estructuras

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

Tipos por valor y referencia

- Los tipos por valor se copian al pasarlos a funciones o asignarlos a variables
- En los tipos por referencia no hay copia, sólo se pasa una referencia al valor original
- Las estructuras y enumeraciones son tipos por valor
- Las clases son tipos por referencia

Tipos por valor

```
let hd = Resolution(width: 1920, height: 1080)

var cinema = hd

cinema.width = 2048

print("cinema is now \(cinema.width) pixels wide")

print("hd is still \(hd.width) pixels wide")
```

Tipos por referencia

```
let tenEighty = VideoMode()
tenEighty.resolution = hd
tenEighty.interlaced = true
tenEighty.name = "1080i"
tenEighty.frameRate = 25.0

let alsoTenEighty = tenEighty

alsoTenEighty.frameRate = 30.0

print("The frameRate property of tenEighty is now \(tenEighty.frameRate)")
```

Operadores de identidad

```
if tenEighty === alsoTenEighty {
    print("tenEighty and alsoTenEighty refer to the same Resolution instance.")
}
```

Operadores de identidad

- Los operadores de identidad === y !== comprueban si dos constantes o variables se refieren a la misma instancia de una clase
- El operador de igualdad == compara los contenidos y dependerá de como haya sido implementado para un tipo concreto

Propiedades

Propiedades

- Permiten almacenar datos dentro de instancias de estructuras, clases o enumeraciones en forma de variables o constantes
- Pueden ser almacenadas o calculadas
- Se pueden añadir observers para ejecutar código cuando se modifican

Propiedades almacenadas

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


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

rangeOfThreeItems.firstValue = 6
```

Propiedades almacenadas

```
let rangeOfFourItems = FixedLengthRange(firstValue: 0, length: 4)

rangeOfFourItems.firstValue = 6  // Error
```

Propiedades almacenadas

- Cuando se marca como constante una instancia de un tipo por valor (estructura o enumeración) sus propiedades no pueden modificarse
- Si es un tipo por referencia (clase) aunque declaremos la instancia como constante sus propiedades se pueden modificar

Propiedades lazy stored

- No se les asigna valor hasta que no se acceden la primera vez
- Se utilizan cuando tenemos propiedades cuyo cálculo es costoso o que dependen de recursos externos
- Se crean poniendo el modificador **lazy** antes de la declaración

Propiedades lazy stored

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

class DataManager {
    lazy var importer = DataImporter()
    var data = [String]()
}

let manager = DataManager()
manager.data.append("Some data")
manager.data.append("Some more data")

// En este punto la instancia de DataImporter todavía no se ha creado

print(manager.importer.fileName) // Ahora sí
```

Propiedades calculadas

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

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

Propiedades calculadas

```
struct Rect {
    var origin = Point()
    var size = Size()
    var center: Point {
        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)
        }
    }
}
```

Propiedades calculadas

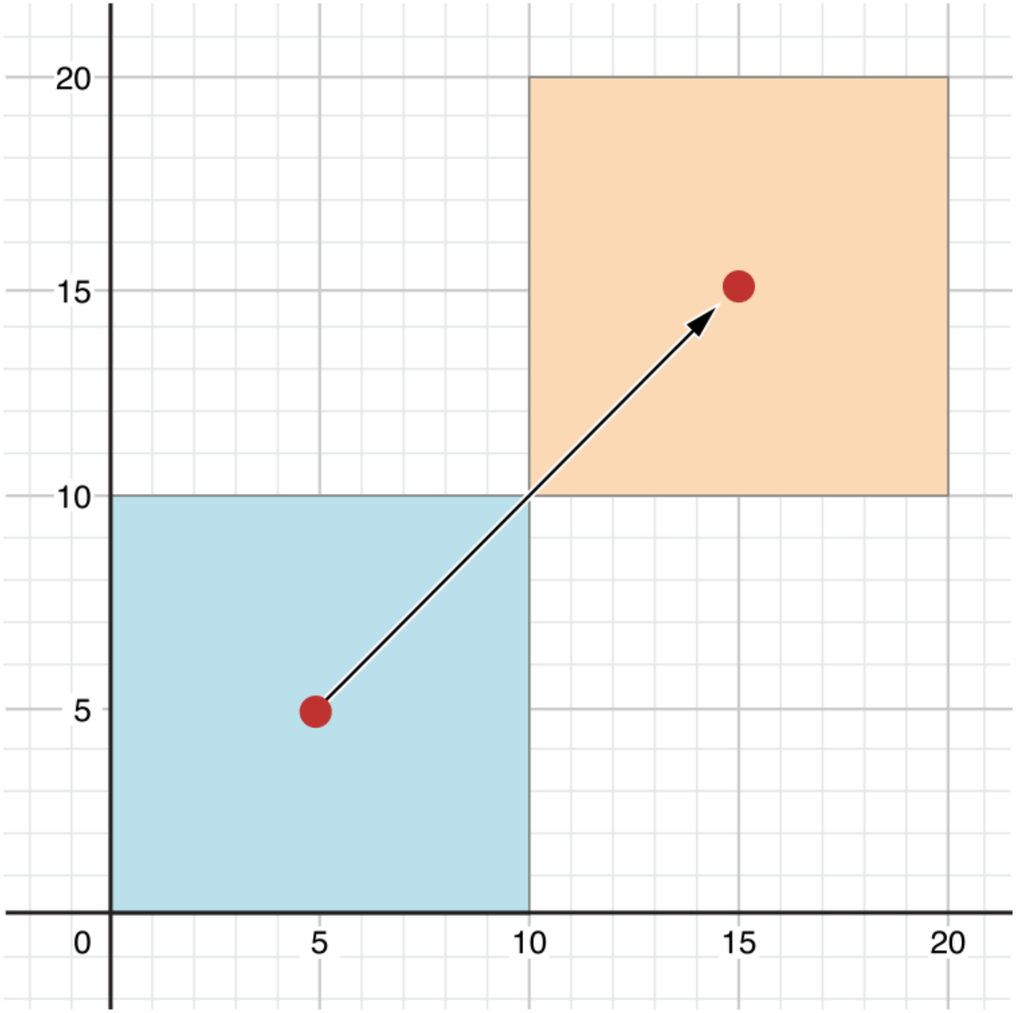
```
var square = Rect(origin: Point(x: 0.0, y: 0.0),
                  size: Size(width: 10.0, height: 10.0))

let initialSquareCenter = square.center

square.center = Point(x: 15.0, y: 15.0)

print("square.origin is now at (\(square.origin.x), \(square.origin.y))")
```

Propiedades calculadas



Sintaxis abreviada para los setter

```
struct AlternativeRect {
    var origin = Point()
    var size = Size()
    var center: Point {
        get {
            let centerX = origin.x + (size.width / 2)
            let centerY = origin.y + (size.height / 2)
            return Point(x: centerX, y: centerY)
        }
        set {
            origin.x = newValue.x - (size.width / 2)
            origin.y = newValue.y - (size.height / 2)
        }
    }
}
```

Propiedades calculadas de solo lectura

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

let fourByFiveByTwo = Cuboid(width: 4.0, height: 5.0, depth: 2.0)

print("the volume of fourByFiveByTwo is \(fourByFiveByTwo.volume)")
```

Observers

- Son métodos que se ejecutan siempre que se modifica el valor de la propiedad, aunque no se modifique el dato
- Pueden añadirse a propiedades almacenadas o calculadas heredadas
- No se ejecutan durante la inicialización

Observers

```
class StepCounter {
    var totalSteps: Int = 0 {
        willSet(newTotalSteps) {
            print("About to set totalSteps to \(newTotalSteps)")
        }
        didSet {
            if totalSteps > oldValue  {
                print("Added \(totalSteps - oldValue) steps")
            }
        }
    }
}
```

Observers

```
let stepCounter = StepCounter()

stepCounter.totalSteps = 200
stepCounter.totalSteps = 360
stepCounter.totalSteps = 896
```

Propiedades de tipo

- Son propiedades definidas a nivel de tipo de dato, no de instancia
- En otros lenguajes se conocen como estáticas (por ejemplo Java)
- Siempre deben tener valor inicial (no se ejecuta el inicializador)

Propiedades de tipo

```
struct SomeStructure {
    static var storedTypeProperty = "Some value."
    static var computedTypeProperty: Int {
        return 1
    }
}

enum SomeEnumeration {
    static var storedTypeProperty = "Some value."
    static var computedTypeProperty: Int {
        return 6
    }
}

class SomeClass {
    static var storedTypeProperty = "Some value."
    static var computedTypeProperty: Int {
        return 27
    }
    class var overrideableComputedTypeProperty: Int {
        return 107
    }
}
```

Propiedades de tipo

```
print(SomeStructure.storedTypeProperty)
// Prints "Some value."

SomeStructure.storedTypeProperty = "Another value."
print(SomeStructure.storedTypeProperty)
// Prints "Another value."

print(SomeEnumeration.computedTypeProperty)
// Prints "6"

print(SomeClass.computedTypeProperty)
// Prints "27"
```

Métodos

Métodos

- Son funciones asociadas con un tipo concreto
- Pueden ser de instancia o de tipo
- Las estructuras, enumeraciones y las clases pueden definir métodos

Métodos de instancia

```
class Counter {
    var count = 0
    func increment() {
        count += 1
    }
    func increment(by amount: Int) {
        count += amount
    }
    func reset() {
        count = 0
    }
}
```

Métodos de instancia

```
let counter = Counter()
// the initial counter value is 0

counter.increment()
// the counter's value is now 1

counter.increment(by: 5)
// the counter's value is now 6

counter.reset()
// the counter's value is now 0
```

La propiedad self

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

La propiedad self

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

let somePoint = Point(x: 4.0, y: 5.0)

if somePoint.isToTheRightOf(x: 1.0) {
    print("This point is to the right of the line where x == 1.0")
}
// Prints "This point is to the right of the line where x == 1.0"
```

Modificar tipos por valor

- Desde los métodos de una estructura o enumeración no podemos modificar sus propiedades, porque son tipos por valor
- Para poder hacerlo, hay que marcar el método que lo necesite con la palabra clave **mutating**
- Se puede incluso modificar el valor de **self** por otra instancia completamente distinta

Modificar tipos por valor

```
struct Point {
    var x = 0.0, y = 0.0
    mutating func moveBy(x deltaX: Double, y deltaY: Double) {
        x += deltaX
        y += deltaY
    }
}

var somePoint = Point(x: 1.0, y: 1.0)

somePoint.moveBy(x: 2.0, y: 3.0)
print("The point is now at (\(somePoint.x), \(somePoint.y))")
// Prints "The point is now at (3.0, 4.0)"
```

Modificar tipos por valor

```
let fixedPoint = Point(x: 3.0, y: 3.0)

fixedPoint.moveBy(x: 2.0, y: 3.0)
// this will report an error
```

Asignar a self desde un método mutante

```
struct Point {
    var x = 0.0, y = 0.0
    mutating func moveBy(x deltaX: Double, y deltaY: Double) {
        self = Point(x: x + deltaX, y: y + deltaY)
    }
}
```

Asignar a self desde un método mutante

```
enum TriStateSwitch {
    case off, low, high
    mutating func next() {
        switch self {
        case .off:
            self = .low
        case .low:
            self = .high
        case .high:
            self = .off
        }
    }
}

var ovenLight = TriStateSwitch.low

ovenLight.next()
// ovenLight is now equal to .high

ovenLight.next()
// ovenLight is now equal to .off
```

Métodos de tipo

- Son métodos definidos a nivel de tipo de dato, no de instancia
- En otros lenguajes se conocen como estáticos (por ejemplo Java)
- Se definen prefijando el método con **static**
- En el caso de las clases, también se pueden definir prefijando el método con **class** para indicar que las subclases pueden sobreescribir la implementación de dicho método

Métodos de tipo

```
class SomeClass {
    class func someTypeMethod() {
        // type method implementation goes here
    }
}

SomeClass.someTypeMethod()
```

Subíndices

Subíndices

- Permiten definir el acceso a los elementos de una colección, secuencia o lista, mediante **[]**
- Se pueden definir múltiples subíndices diferentes en el mismo tipo
- Se selecciona el correcto en función del tipo de dato que se le pase al acceder
- Pueden tener múltiples parámetros y devolver cualquier valor

Subíndices

```
subscript(index: Int) -> Int {
    get {
        // return an appropriate subscript value here
    }
    set(newValue) {
        // perform a suitable setting action here
    }
}
```

Subíndices de sólo lectura

```
subscript(index: Int) -> Int {
    // return an appropriate subscript value here
}
```

Subíndices

```
struct TimesTable {
    let multiplier: Int
    subscript(index: Int) -> Int {
        return multiplier * index
    }
}

let threeTimesTable = TimesTable(multiplier: 3)

print("six times three is \(threeTimesTable[6])")
// Prints "six times three is 18"
```

Herencia

Herencia

- En Swift, es una capacidad exclusiva de las clases
- Al heredar, se generan subclases a partir de superclases
- Las subclases pueden acceder a métodos y propiedades de las superclases o reemplazarlos con sus propias versiones
- En Swift, las clases no tienen un ancestro común

Clase base

```
class Vehicle {
    var currentSpeed = 0.0
    var description: String {
        return "traveling at \(currentSpeed) miles per hour"
    }
    func makeNoise() {
        // do nothing - an arbitrary vehicle doesn't necessarily make a noise
    }
}
```

Instancia

```
let someVehicle = Vehicle()

print("Vehicle: \(someVehicle.description)")
// Vehicle: traveling at 0.0 miles per hour
```

Herencia

```
class SomeSubclass: SomeSuperclass {
    // subclass definition goes here
}
```

Subclase

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

Subclase

```
let bicycle = Bicycle()
bicycle.hasBasket = true

bicycle.currentSpeed = 15.0

print("Bicycle: \(bicycle.description)")
// Bicycle: traveling at 15.0 miles per hour
```

Subclase

```
class Tandem: Bicycle {
    var currentNumberOfPassengers = 0
}

let tandem = Tandem()

tandem.hasBasket = true
tandem.currentNumberOfPassengers = 2
tandem.currentSpeed = 22.0

print("Tandem: \(tandem.description)")
// Tandem: traveling at 22.0 miles per hour
```

Override

- Permite reemplazar métodos de instancia, de clase, propiedades de instancia o subíndices en las subclases
- El elemento a reemplazar se marca con **override**

Acceso a la superclase

```
super.someMethod()

super.someProperty

super[someIndex]
```

Reemplazo de métodos

```
class Train: Vehicle {
    override func makeNoise() {
        print("Choo Choo")
    }
}

let train = Train()
train.makeNoise()
// Prints "Choo Choo"
```

Reemplazo de propiedades

```
class Car: Vehicle {
    var gear = 1
    override var description: String {
        return super.description + " in gear \(gear)"
    }
}

let car = Car()

car.currentSpeed = 25.0
car.gear = 3

print("Car: \(car.description)")
// Car: traveling at 25.0 miles per hour in gear 3
```

Reemplazo de observers

```
class AutomaticCar: Car {
    override var currentSpeed: Double {
        didSet {
            gear = Int(currentSpeed / 10.0) + 1
        }
    }
}

let automatic = AutomaticCar()

automatic.currentSpeed = 35.0

print("AutomaticCar: \(automatic.description)")
// AutomaticCar: traveling at 35.0 miles per hour in gear 4
```

Evitar reemplazos

- Si queremos evitar que una subclase pueda reemplazar lo que hemos definido, lo marcaremos con el modificador **final** (métodos, propiedades, subindices...)
- Cualquier intento de hacer un reemplazo dará un error de compilación
- Podemos marcar la clase con **final** y no se podrán generar subclases a partir de ella