Swift

Express

Introducción a Swift

Swift

- Lenguaje de programación multiparadigma desarrollado por Apple
- Reemplazo de Objective-C
- Presentado en Junio de 2014 durante la WWDC
- Manual completo disponible online
- Open Source

Swift

https://developer.apple.com/swift/
https://docs.swift.org/swift-book/documentation/the-swift-programming-language/

Elementos básicos

Sentencias, bloques y comentarios

Sentencias

- En Swift, las sentencias se escriben una en cada línea
- No es necesario incluir un ; al final

Sentencias y ;

```
let cat = "🐱"; print(cat)
```

Bloques

- Agrupan instrucciones
- Definen el ámbito de las variables
- En Swift se utilizan las llaves { y } para delimitarlos

Comentarios

- De una línea, //
- De múltiples líneas, /* */
- Se pueden anidar, si están balanceados /* /* */ */

Variables y constantes

Declaración

```
let maximumNumberOfLoginAttempts = 10  // Constante
var currentLoginAttempt = 0  // Variable


var x = 0.0, y = 0.0, z = 0.0   // Múltiple
```

Anotaciones de tipo

```
var welcomeMessage: String

welcomeMessage = "Hello"

var red, green, blue: Double
```

Tipos de datos básicos

Tipo	Descripción
Int	Valor numérico entero
Float	Valor numérico de precisión simple
Double	Valor numérico de precisión doble
Bool	Valor lógico, verdadero o falso
Character	Caracter individual
String	Cadena de texto

Nombres

```
let π = 3.14159
let 你好 = "你好世界"
let 🐶🐮 = "dogcow"
```

- No pueden contener espacios ni símbolos de flecha o bloques
- No pueden comenzar por número
- Admiten Unicode

Salida por consola

```
print("This is a string")
```

```
var friendlyWelcome = "Hello!"

print("The current value of friendlyWelcome is \(friendlyWelcome)")
```

Lectura por teclado

```
var altura = Double(readLine()!) ?? 0.0
```

Valores numéricos y lógicos

Enteros

- Enteros con signo: Int
- Enteros sin signo: UInt
- Existen Int8, Int16, Int32 e Int64 (y las versiones sin signo)
- Al usar Int, internamente la longitud cambia a Int32 o Int64 dependiendo de la plataforma (32 o 64 bits)

Coma flotante

- Precisión simple: Float (32 bits, 6 dígitos decimales de precisión)
- Precisión doble: Double (64 bits, 15 dígitos decimales de precisión)

Inferencia de tipos

```
let meaningOfLife = 42

let pi = 3.14159

let anotherPi = 3 + 0.14159
```

Literales numéricos

	Literal	Notación	Ejemplo
Enteros	Decimal	Sin prefijo	1_000_000
	Binario	0b	0b1011
	Octal	0o	0o34
	Hexadecimal	0x	0xF3A
Coma flotante	Decimal	e	1.25e4
	Hexadecimal	p	0xAp3

Conversiones de tipo

```
let three = 3
let pointOneFourOneFiveNine = 0.14159
let pi = Double(three) + pointOneFourOneFiveNine


let integerPi = Int(pi)  // Se trunca el valor
```

Valores lógicos

```
let orangesAreOrange = true

let turnipsAreDelicious = false
```

Cadenas de texto

Cadenas de texto

```
let someString = "Some string literal value"
```

Multiline String

```
let quotation = """
The White Rabbit put on his spectacles.  "Where shall I begin,
please your Majesty?" he asked.

"Begin at the beginning," the King said gravely, "and go on
till you come to the end; then stop."
"""
```

Caracteres especiales

```
let wiseWords = "\"Imagination is more important than knowledge\" - Einstein"
// "Imagination is more important than knowledge" - Einstein

let dollarSign = "\u{24}"        // $,  Unicode scalar U+0024
let blackHeart = "\u{2665}"      // ♥,  Unicode scalar U+2665

let sparklingHeart = "\u{1F496}" // 💖, Unicode scalar U+1F496
```

Cadena vacía

```
var emptyString = ""               // empty string literal
var anotherEmptyString = String()  // initializer syntax

if emptyString.isEmpty {
    print("Nothing to see here")
}


var noEstaVacia: String  // No está inicializada
```

Características de los String

- Si se declaran con var son mutables, si se declaran con let, no
- Se pueden concatenar con + y +=
- Se pueden recorrer los caracteres individuales con un for-in
- Son tipos por valor, se copian al pasarlos a funciones o asignarlos a otras variables

Recorrer los caracteres de un String

```
for character in "Dog!🐶" {
    print(character)
}
```

Concatenación de Strings

```
let string1 = "hello"
let string2 = " there"
var welcome = string1 + string2

var instruction = "look over"
instruction += string2

let exclamationMark: Character = "!"
welcome.append(exclamationMark)
```

Interpolación de Strings

```
let multiplier = 3
let message = "\(multiplier) times 2.5 is \(Double(multiplier) * 2.5)"
```

Recuento de caracteres

```
let unusualMenagerie = "Koala 🐨, Snail 🐌, Penguin 🐧, Dromedary 🐪"
print("unusualMenagerie has \(unusualMenagerie.count) characters")
```

Subcadenas

```
let greeting = "Hello, world!"
let index = greeting.firstIndex(of: ",") ?? greeting.endIndex
let beginning = greeting[..<index]
// beginning is "Hello"

// Convert the result to a String for long-term storage.
let newString = String(beginning)
```

Subcadenas

```
let name = "The Grapes of Wrath"

// Get range based on the string index.
let r = name.index(name.startIndex, offsetBy: 4)..<name.endIndex

// Access substring from range.
let result = name[r]
print(result)
```

Subcadenas

```
// This is the input string.
let s = "one two three"

// Get range 4 places from the start, and 6 from the end.
let r = s.index(s.startIndex, offsetBy: 4)..<s.index(s.endIndex, offsetBy:
-6)

// Access the string by the range.
let substring = s[r]
print(substring)
```

Subcadenas

```
let value = "bird, lizard and fish"

// Get range of all characters past the first 6.
let range = value.index(value.startIndex, offsetBy: 6)..<value.endIndex

// Access the substring.
let substring = value[range]
print(substring)
```

Comparar Strings

- Se pueden comparar directamente con el operador ==
- Disponen de hasPrefix(_:) y hasSuffix(_:) para comparar el principio o el final de la cadena

Igualdad de Strings

```
let quotation = "We're a lot alike, you and I."
let sameQuotation = "We're a lot alike, you and I."

if quotation == sameQuotation {
    print("These two strings are considered equal")
}
```

Tuplas

Tuplas

- Agrupan múltiples valores en uno
- Los valores pueden ser de cualquier tipo
- No tienen que ser del mismo tipo
- Permiten que una función devuelva varios valores agrupados
- Para agrupaciones complejas, hay que usar estructuras o clases, no tuplas

Tuplas

```
let http404Error = (404, "Not Found")

let (statusCode, statusMessage) = http404Error

print("The status code is \(statusCode)")
print("The status message is \(statusMessage)")
```

Variables opcionales

Optionals

- Permiten definir variables que pueden o no tener valor
- Se crean añadiendo ? al tipo de dato de la variable

Optionals

```
let possibleNumber = "123"

let convertedNumber = Int(possibleNumber)
```

Valor nil

```
var serverResponseCode: Int? = 404
serverResponseCode = nil

var surveyAnswer: String?   // nil
```

Extraer el valor de un opcional

- Imprescindible: Int no es lo mismo que Int?
- Forced unwrapping: usando !
- Optional binding: para extraer el valor en un if o while
- Optional chaining: usando ?, cuando trabajemos con propiedades de estructuras o clases

Forced unwrapping

```
let possibleNumber = "123"
let convertedNumber = Int(possibleNumber)

if convertedNumber != nil {
    print("\(possibleNumber) has an integer value of \(convertedNumber!)")
} else {
    print("\(possibleNumber) could not be converted to an integer")
}
```

Optional binding

```
let possibleNumber = "123"

if let actualNumber = Int(possibleNumber) {
    print("\(possibleNumber) has an integer value of \(actualNumber)")
} else {
    print("\(possibleNumber) could not be converted to an integer")
}
```

Opcionales implícitos

- Se declaran con ! en vez de ? en el tipo de dato
- No necesitan de ! para acceder al valor, pero si no tienen valor disparar un error en tiempo de ejecución
- Se usan en la inicialización de clases con referencias unowned

Opcionales implícitos

```
let possibleString: String? = "An optional string."
let forcedString: String = possibleString!

let assumedString: String! = "An implicitly unwrapped optional string."
let implicitString: String = assumedString
```

Operador de coalescencia nil

- Se utiliza con opcionales mediante ??
- Permite extraer el valor del opcional o si vale nil, un valor por defecto
- a ?? b es una abreviatura de a != nil ? a! : b

Operador de coalescencia nil

```
let defaultColorName = "red"
var userDefinedColorName: String?   // nil

var colorNameToUse = userDefinedColorName ?? defaultColorName
```

Operador de coalescencia nil

```
userDefinedColorName = "green"

colorNameToUse = userDefinedColorName ?? defaultColorName
```

Operadores: asignación y aritméticos

Operador de asignación

- Copia el contenido de la parte derecha en la parque izquierda
- Descompone los valores de las tuplas en variables individuales
- No devuelve valor
- Hay versiones compuestas, como +=

Operador de asignación

```
let b = 10
var a = 5
a = b

let (x, y) = (1, 2)
```

Operador de asignación

```
var i = 6
var j = i = 5  // Error

if j=5 {  // Error

}
```

Operadores aritméticos

Operador	Operación
+	Suma
-	Resta
*	Multiplicación
/	División
%	Resto de la división
-i	Menos unario (cambio de signo)
+i	Más unario (no afecta al valor)

Operadores aritméticos

- No soportan overflow o underflow, se produce un error de tiempo de ejecución.

```
var potentialOverflow = Int16.max
// potentialOverflow equals 32767, which is the maximum value an Int16
potentialOverflow += 1
// this causes an error
```

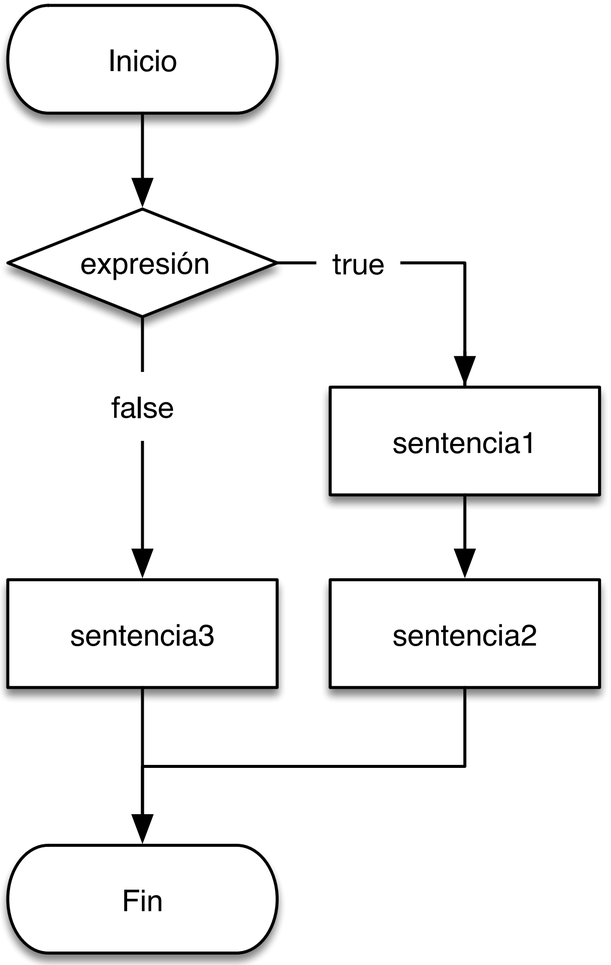
- Hay versiones con overflow, como &+
- La división y el resto entre 0 también provocan un error
- No existen los operadores ++ o --

Estructuras de control

Alternativa simple: if

Alternativa simple: if

```
if expresión {
    sentencia1
    sentencia2
}
else {
    sentencia3
}
```



Alternativa simple: if

```
var temperatureInFahrenheit = 30

if temperatureInFahrenheit <= 32 {
    print("It's very cold. Consider wearing a scarf.")
}
```

Alternativa simple: if

```
temperatureInFahrenheit = 90

if temperatureInFahrenheit <= 32 {
    print("It's very cold. Consider wearing a scarf.")
} else if temperatureInFahrenheit >= 86 {
    print("It's really warm. Don't forget to wear sunscreen.")
} else {
    print("It's not that cold. Wear a t-shirt.")
}
```

Operadores: relacionales y lógicos

Operadores relacionales

Operador	Operación
==	Igual (compara contenido)
!=	Distinto
>	Mayor que
<	Menor que
>=	Mayor o igual que
<=	Menor o igual que
===	Idéntico (compara referencias)
!==	No idéntico
`c ? a:b`	Si c, entonces a. Si no c, entonces b.

Operadores relacionales

```
1 == 1   // true because 1 is equal to 1
2 != 1   // true because 2 is not equal to 1
2 > 1    // true because 2 is greater than 1
1 < 2    // true because 1 is less than 2
1 >= 1   // true because 1 is greater than or equal to 1
2 <= 1   // false because 2 is not less than or equal to 1
```

Operadores relacionales

```
let name = "world"
if name == "world" {
    print("hello, world")
} else {
    print("I'm sorry \(name), but I don't recognize you")
}
// Prints "hello, world", because name is indeed equal to "world".
```

Comparar tuplas

```
(1, "zebra") < (2, "apple")    // true because 1 is less than 2; "zebra" and "apple" are not compared
(3, "apple") < (3, "bird")     // true because 3 is equal to 3, and "apple" is less than "bird"
(4, "dog") == (4, "dog")       // true because 4 is equal to 4, and "dog" is equal to "dog"

("blue", -1) < ("purple", 1)         // OK, evaluates to true
("blue", false) < ("purple", true)  // Error because < can't compare Boolean values
```

Operador ternario

```
let contentHeight = 40
let hasHeader = true
let rowHeight = contentHeight + (hasHeader ? 50 : 20)
// rowHeight is equal to 90
```

Operadores lógicos

Operador	Operación
!	Negación lógica, NOT
&&	Conjunción lógica, AND
ll	Disyunción lógica, OR

Operador NOT (!)

```
let allowedEntry = false
if !allowedEntry {
    print("ACCESS DENIED")
}
// Prints "ACCESS DENIED"
```

Operador AND (&&)

```
let enteredDoorCode = true
let passedRetinaScan = false
if enteredDoorCode && passedRetinaScan {
    print("Welcome!")
} else {
    print("ACCESS DENIED")
}
// Prints "ACCESS DENIED"
```

Operador OR (||)

```
let hasDoorKey = false
let knowsOverridePassword = true
if hasDoorKey || knowsOverridePassword {
    print("Welcome!")
} else {
    print("ACCESS DENIED")
}
// Prints "Welcome!"
```

Combinar operadores lógicos

```
let enteredDoorCode = true
let passedRetinaScan = false
let hasDoorKey = false
let knowsOverridePassword = true

if enteredDoorCode && passedRetinaScan || hasDoorKey || knowsOverridePassword {
    print("Welcome!")
} else {
    print("ACCESS DENIED")
}
// Prints "Welcome!"
```

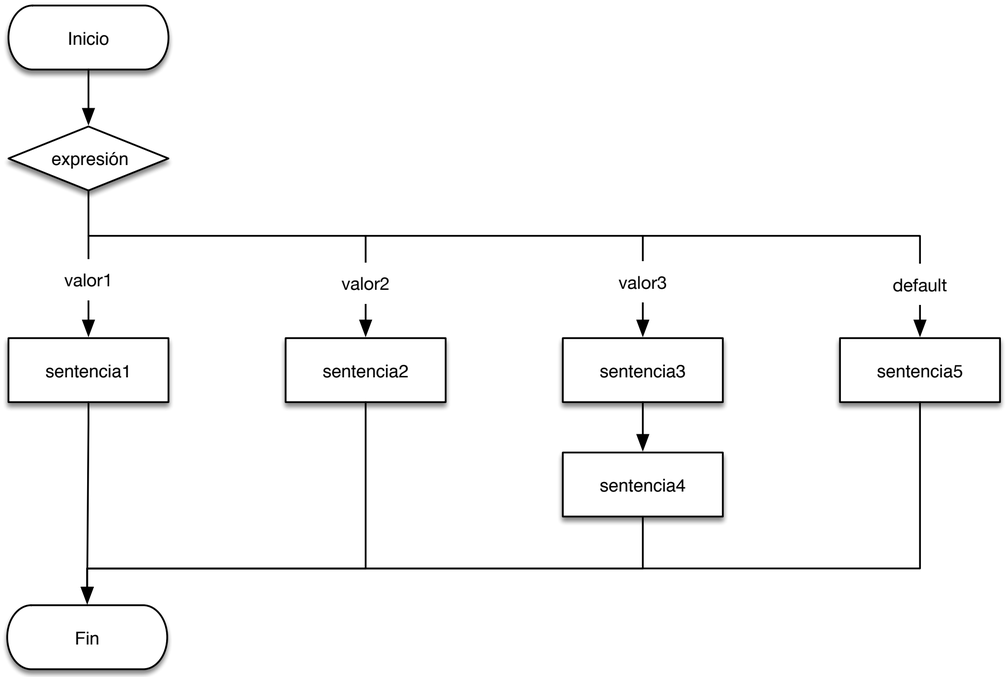
Paréntesis explícitos

```
let enteredDoorCode = true
let passedRetinaScan = false
let hasDoorKey = false
let knowsOverridePassword = true

if (enteredDoorCode && passedRetinaScan) || hasDoorKey || knowsOverridePassword {
    print("Welcome!")
} else {
    print("ACCESS DENIED")
}
// Prints "Welcome!"
```

Alternativa múltiple: switch

Alternativa múltiple: switch

```
switch variable {
case valor:
    sentencia
    sentencia
    ...
case valor:
    sentencia
    ...
default:
    sentencia
}
```



Alternativa múltiple: switch

```
let someCharacter: Character = "z"

switch someCharacter {
case "a":
    print("The first letter of the alphabet")
case "z":
    print("The last letter of the alphabet")
default:
    print("Some other character")
}
```

Alternativa múltiple: switch

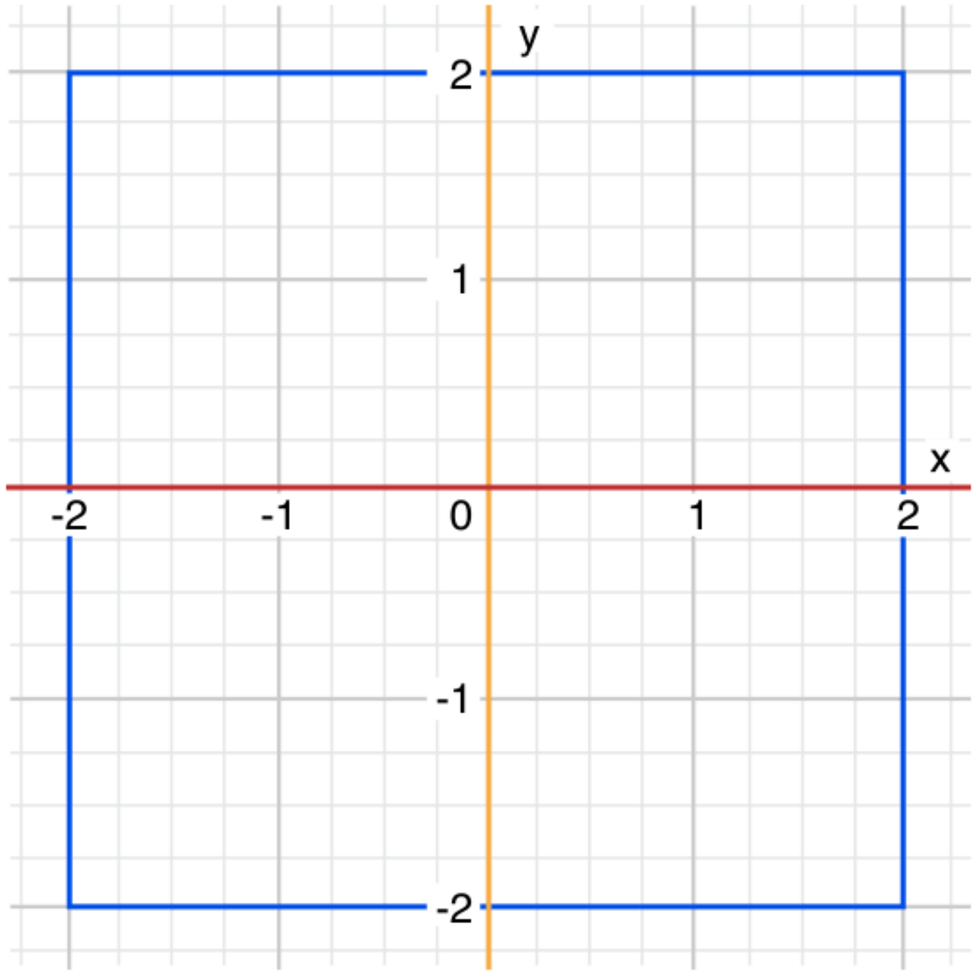
- A diferencia de en C o Java, no hace falta break en cada caso
- No hay *fallthrough* automático
- No puede haber casos vacíos
- Debe evaluar todos los casos posibles o tener default
- Se puede afinar más la condición usando where
- Admite intervalos y tuplas

Switch con intervalos

```
let approximateCount = 62
let countedThings = "moons orbiting Saturn"
var naturalCount: String

switch approximateCount {
case 0:
    naturalCount = "no"
case 1..<5:
    naturalCount = "a few"
case 5..<12:
    naturalCount = "several"
case 12..<100:
    naturalCount = "dozens of"
case 100..<1000:
    naturalCount = "hundreds of"
default:
    naturalCount = "many"
}

print("There are \(naturalCount) \(countedThings).")
```

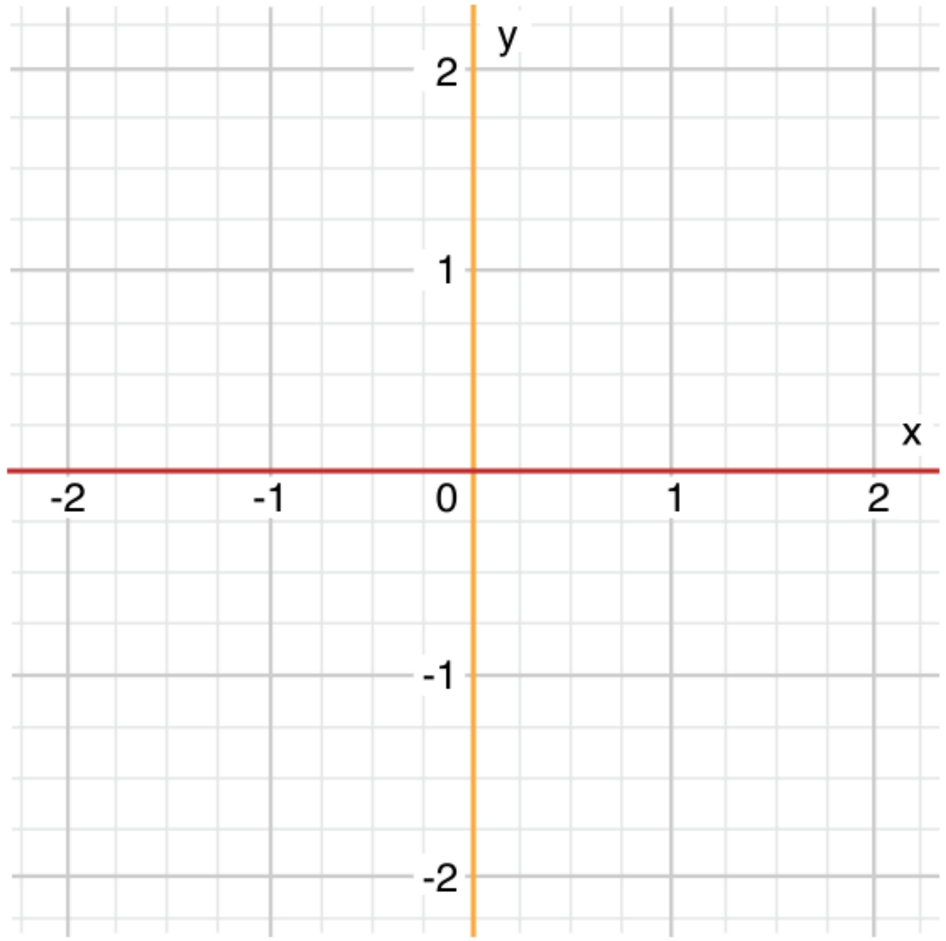
Switch con tuplas

```
let somePoint = (1, 1)

switch somePoint {
case (0, 0):
    print("(0, 0) is at the origin")
case (_, 0):
    print("(\(somePoint.0), 0) is on the x-axis")
case (0, _):
    print("(0, \(somePoint.1)) is on the y-axis")
case (-2...2, -2...2):
    print("(\(somePoint.0), \(somePoint.1)) is inside the
box")
default:
    print("(\(somePoint.0), \(somePoint.1)) is outside of the
box")
}
```



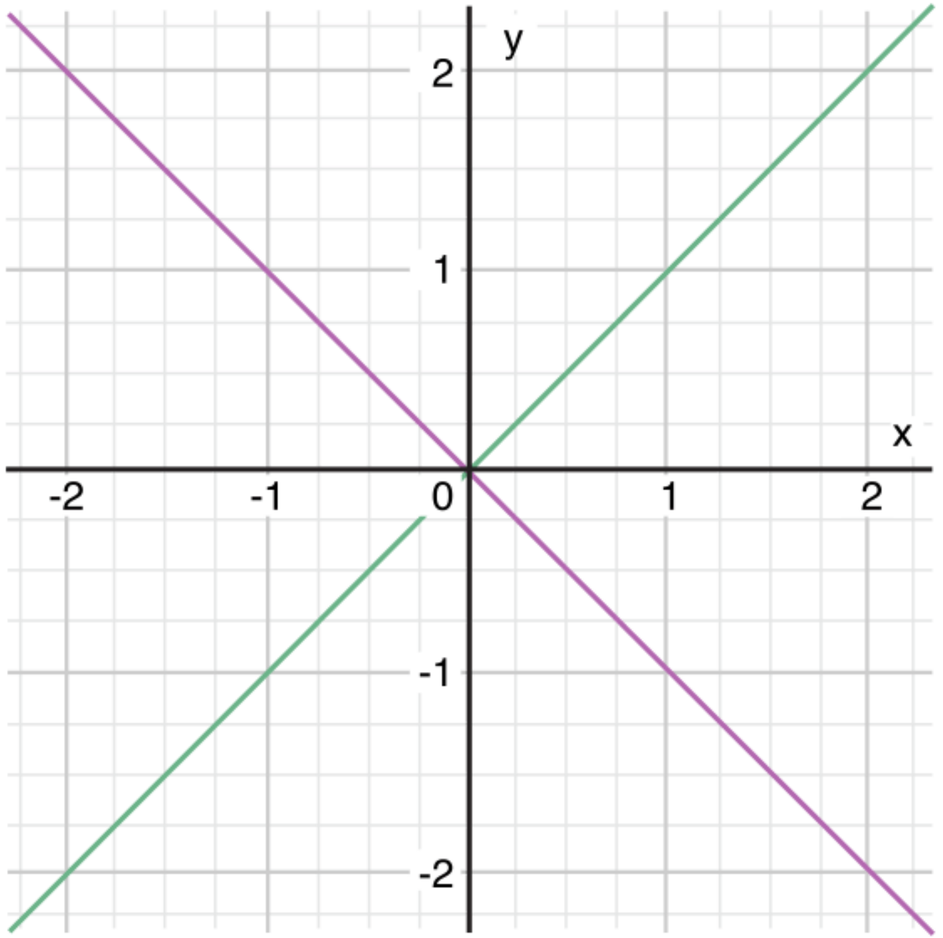
Value bindings

```
let anotherPoint = (2, 0)

switch anotherPoint {
case (let x, 0):
    print("on the x-axis with an x value of \(x)")
case (0, let y):
    print("on the y-axis with a y value of \(y)")
case let (x, y):
    print("somewhere else at (\(x), \(y))")
}
```



Switch con where

```
let yetAnotherPoint = (1, -1)

switch yetAnotherPoint {
case let (x, y) where x == y:
    print("(\(x), \(y)) is on the line x == y")
case let (x, y) where x == -y:
    print("(\(x), \(y)) is on the line x == -y")
case let (x, y):
    print("(\(x), \(y)) is just some arbitrary point")
}
```



Casos compuestos

```
let someCharacter: Character = "e"

switch someCharacter {
case "a", "e", "i", "o", "u":
    print("\(someCharacter) is a vowel")
case "b", "c", "d", "f", "g", "h", "j", "k", "l", "m",
     "n", "p", "q", "r", "s", "t", "v", "w", "x", "y", "z":
    print("\(someCharacter) is a consonant")
default:
    print("\(someCharacter) is not a vowel or a consonant")
}
```

Transferencia de control

- Se puede poner break en un caso para cortar la ejecución y forzar a que el switch termine
- El uso de break permite escribir casos vacíos en el switch (un comentario no basta, daría error)

Fallthrough

```
let integerToDescribe = 5

var description = "The number \(integerToDescribe) is"

switch integerToDescribe {
case 2, 3, 5, 7, 11, 13, 17, 19:
    description += " a prime number, and also"
    fallthrough
default:
    description += " an integer."
}

print(description)
```

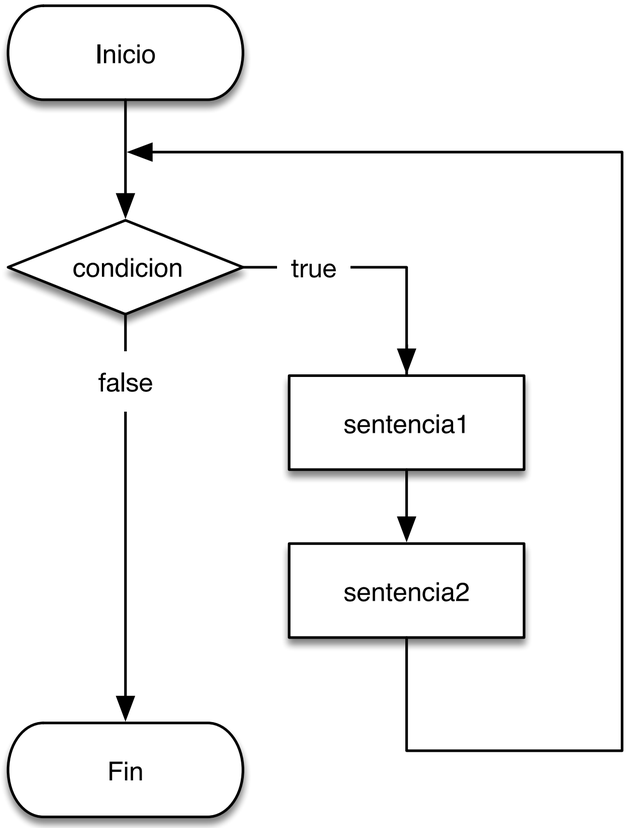
Repetitivas

Repetitivas

0 → n	1 → n	n
while	repeat-while	for-in
Puede que nunca se ejecute	Se ejecuta por lo menos una vez	Recorre los elementos de un intervalo o colección

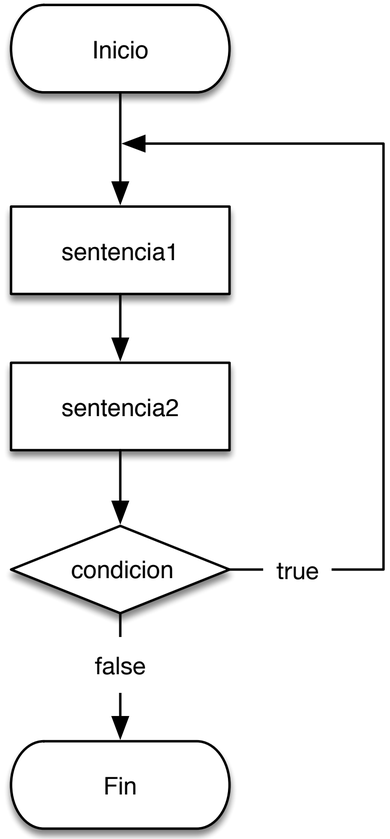
while

```
var i = 0

while i < 3 {
    print("W: El valor de i es: \(i)")
    i += 1
}
```



repeat-while

```
var j = 0

repeat {
    print("RW: El valor de j es: \(j)")
    j += 1
} while j < 3
```



for-in

```
for index in 1...5 {
    print("\(index) times 5 is \(index * 5)")
}
```

for-in

```
let base = 3
let power = 10

var answer = 1

for _ in 1...power {
    answer *= base
}

print("\(base) to the power of \(power) is \(answer)")
```

for-in

```
let names = ["Anna", "Alex", "Brian", "Jack"]
for name in names {
    print("Hello, \(name)!")
}
```

```
let numberOfLegs = ["spider": 8, "ant": 6, "cat": 4]
for (animalName, legCount) in numberOfLegs {
    print("\(animalName)s have \(legCount) legs")
}
```

Transferencia de control

- Se puede poner break dentro de un bucle para cortar la repetición actual y forzar a que el bucle termine
- Se puede utilizar continue dentro de un bucle para terminar la repetición actual y pasar a la siguiente
- Se pueden utilizar etiquetas para definir a quien afecta un posible break o continue

Operadores: rangos

Operadores de rango

Operador	Operación	Ejemplo	Valores
n...m	Rango cerrado	1...5	1, 2, 3, 4, 5
n..<m	Rango semicerrado	1..<5	1, 2, 3, 4
n... ...n	Rango cerrado por un lado	2... ...2	2, 3, 4, ... final comienzo ... 1, 2
..<n	Rango semicerrado por un lado	..<2	0, 1

Rango cerrado

```
for index in 1...5 {
    print("\(index) times 5 is \(index * 5)")
}
```

Rango semicerrado

```
let names = ["Anna", "Alex", "Brian", "Jack"]
let count = names.count

for i in 0..<count {
    print("Person \(i + 1) is called \(names[i])")
}
```

Rangos de un solo lado

```
for name in names[2...] {
    print(name)
}

for name in names[...2] {
    print(name)
}

for name in names[..<2] {
    print(name)
}
```

Estructuras de datos

Estructuras de datos

En programación, una estructura de datos es una forma particular de organizar datos en una computadora para que pueda ser utilizado de manera eficiente.

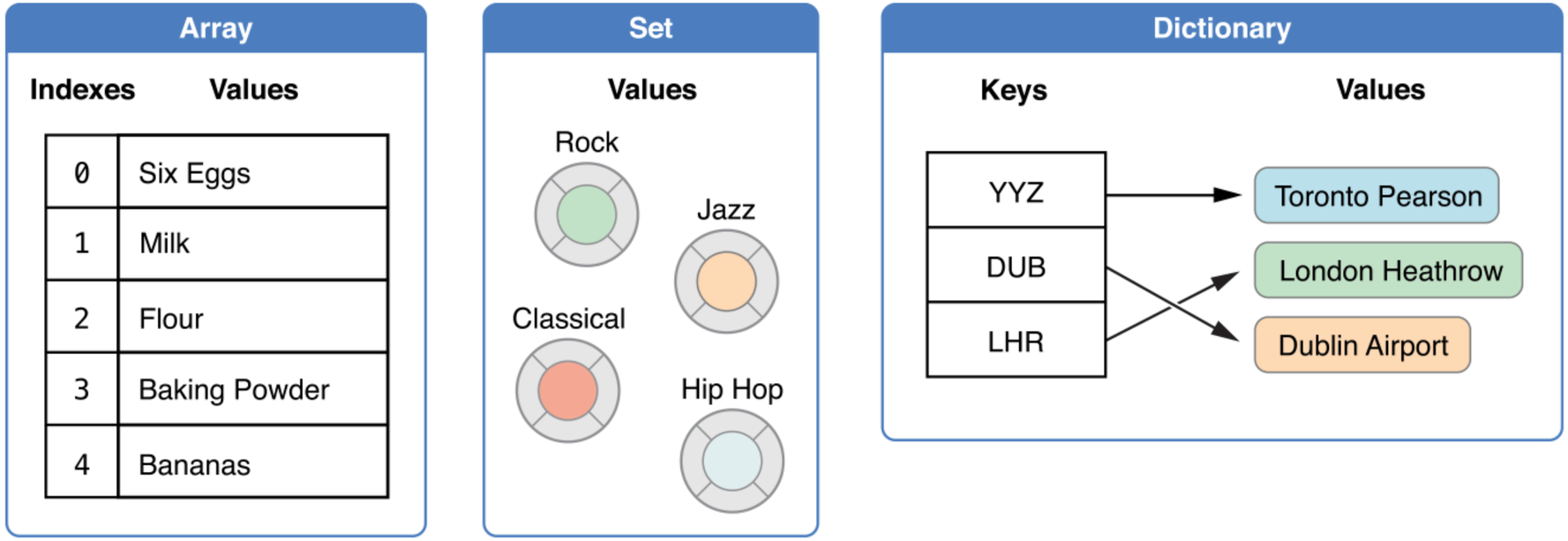
Diferentes tipos de estructuras de datos son adecuados para diferentes tipos de aplicaciones, y algunos son altamente especializados para tareas específicas.

https://es.wikipedia.org/wiki/Estructura_de_datos

Estructuras de datos

- Organización de datos
- Operaciones posibles

Colecciones en Swift



Arrays

En programación, una matriz o vector (llamado en inglés array) es una zona de almacenamiento continuo, que contiene una serie de elementos del mismo tipo, los elementos de la matriz.

https://es.wikipedia.org/wiki/Vector_(informática)

Arrays

```
var otherInts = Array<Int>()

var someInts = [Int]()  // Sintaxis preferida

someInts.append(3)

someInts = []

var threeDoubles = Array(repeating: 0.0, count: 3)
//var threeDoubles =[Double](repeating: 0.0, count: 3)


var anotherThreeDoubles = Array(repeating: 2.5, count: 3)

var sixDoubles = threeDoubles + anotherThreeDoubles
```

Arrays

```
var shoppingList: [String] = ["Eggs", "Milk"]

var firstItem = shoppingList[0]

shoppingList[0] = "Six eggs"

var cinemaShoppingList = ["Chocolate", "Popcorn"]
```

Características de los arrays

- Se numeran desde 0.
- Usan la inferencia de tipo si los inicializamos al crearlos
- El acceso a una posición inexistente provoca un error de tiempo de ejecución
- Si se declara con let no puede variar el número de elementos ni el contenido

Operaciones sobre arrays

- Se puede preguntar cuantos elementos hay con .count
- Se puede preguntar si está vacío con .isEmpty

Operaciones sobre arrays

```
print("The shopping list contains \(shoppingList.count) items.")


if shoppingList.isEmpty {
    print("The shopping list is empty.")
} else {
    print("The shopping list is not empty.")
}
```

Operaciones sobre arrays

- Se pueden añadir elementos con .append(_:)
- Se pueden concatenar arrays con el operador +=
- Se pueden insertar elementos con .insert(_:at:)
- Se puede eliminar elementos con .remove(at:)
- Se puede eliminar el último elemento con .removeLast()

Operaciones sobre arrays

```
shoppingList.append("Flour")
shoppingList += ["Baking Powder"]

shoppingList += ["Chocolate Spread", "Cheese", "Butter"]

shoppingList[4...6] = ["Bananas", "Apples"]

shoppingList.insert("Maple Syrup", at: 0)

let mapleSyrup = shoppingList.remove(at: 0)

let apples = shoppingList.removeLast()
```

Recorrer un array

```
for item in shoppingList {
    print(item)
}
```

Recorrer un array

```
for (index, value) in shoppingList.enumerated() {
    print("Item \(index + 1): \(value)")
}
```

Conjuntos

En informática, un conjunto es una colección (contenedor) de ciertos valores, sin ningún orden concreto ni valores repetidos.

https://es.wikipedia.org/wiki/Conjunto_(programación)

Conjuntos

- Almacenan elementos del mismo tipo
- No hay orden definido
- Se usa cuando no importa el orden o necesitamos que los elementos no se repitan
- Los valores tienen que ser “*hashable*”
- Si se declara con let no puede variar el número de elementos ni el contenido una vez inicializado

Hashable

- Valor numérico único para un objeto concreto
- Permite comparar objetos entre sí
- Los tipos básicos de Swift son *hashable*
- Si queremos que nuestros propios tipos lo sean tienen que implementar el protocolo Hashable

Conjuntos

```
var letters = Set<Character>() // Vacío

print("letters is of type Set<Character> with \(letters.count) items.")

letters.insert("a")

letters = []


var favoriteGenres: Set<String> = ["Rock", "Classical", "Hip hop"]

var myFavoriteGenres: Set = ["Rock", "Classical", "Hip hop”] // Forma más sencilla
```

Operaciones sobre conjuntos

- Se puede preguntar cuantos elementos hay con .count
- Se puede preguntar si está vacío con .isEmpty
- Se pueden añadir elementos con .insert(_:)
- Se pueden eliminar elementos con .remove(_:) o .removeAll()
- Se puede consultar si un elemento existe con .contains(_:)

Operaciones sobre conjuntos

```
print("I have \(favoriteGenres.count) favorite music genres.")


if favoriteGenres.isEmpty {
    print("As far as music goes, I'm not picky.")
} else {
    print("I have particular music preferences.")
}


favoriteGenres.insert("Jazz")
```

Operaciones sobre conjuntos

```
if let removedGenre = favoriteGenres.remove("Rock") {
    print("\(removedGenre)? I'm over it.")
} else {
    print("I never much cared for that.")
}
```

Operaciones sobre conjuntos

```
if favoriteGenres.contains("Funk") {
    print("I get up on the good foot.")
} else {
    print("It's too funky in here.")
}
```

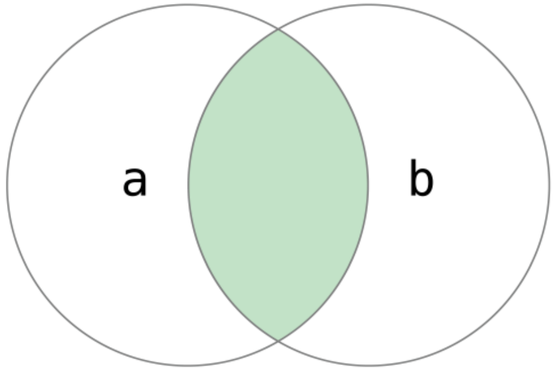
Recorrer un conjunto

```
for genre in favoriteGenres {
    print("\(genre)")
}
```

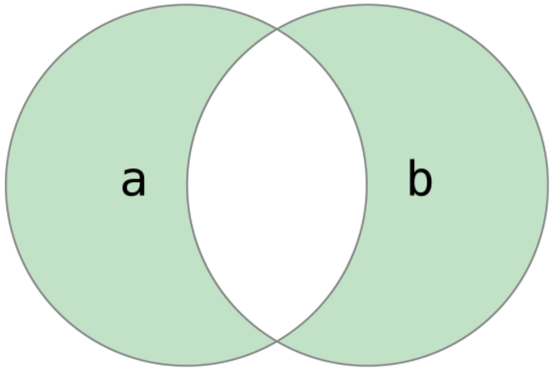
```
for genre in favoriteGenres.sorted() {
    print("\(genre)")
}
```

Operaciones entre conjuntos

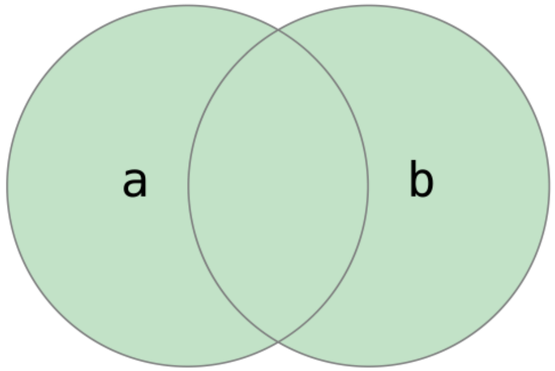
a.intersection(b)



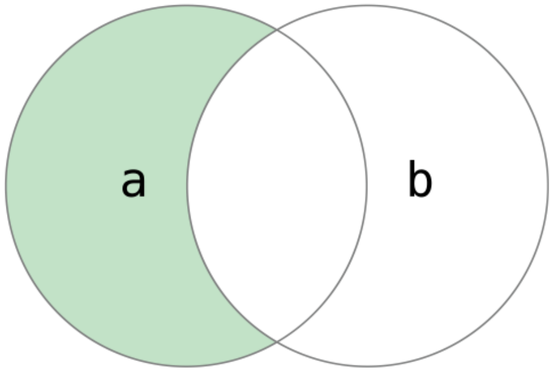
a.symmetricDifference(b)



a.union(b)



a.subtracting(b)



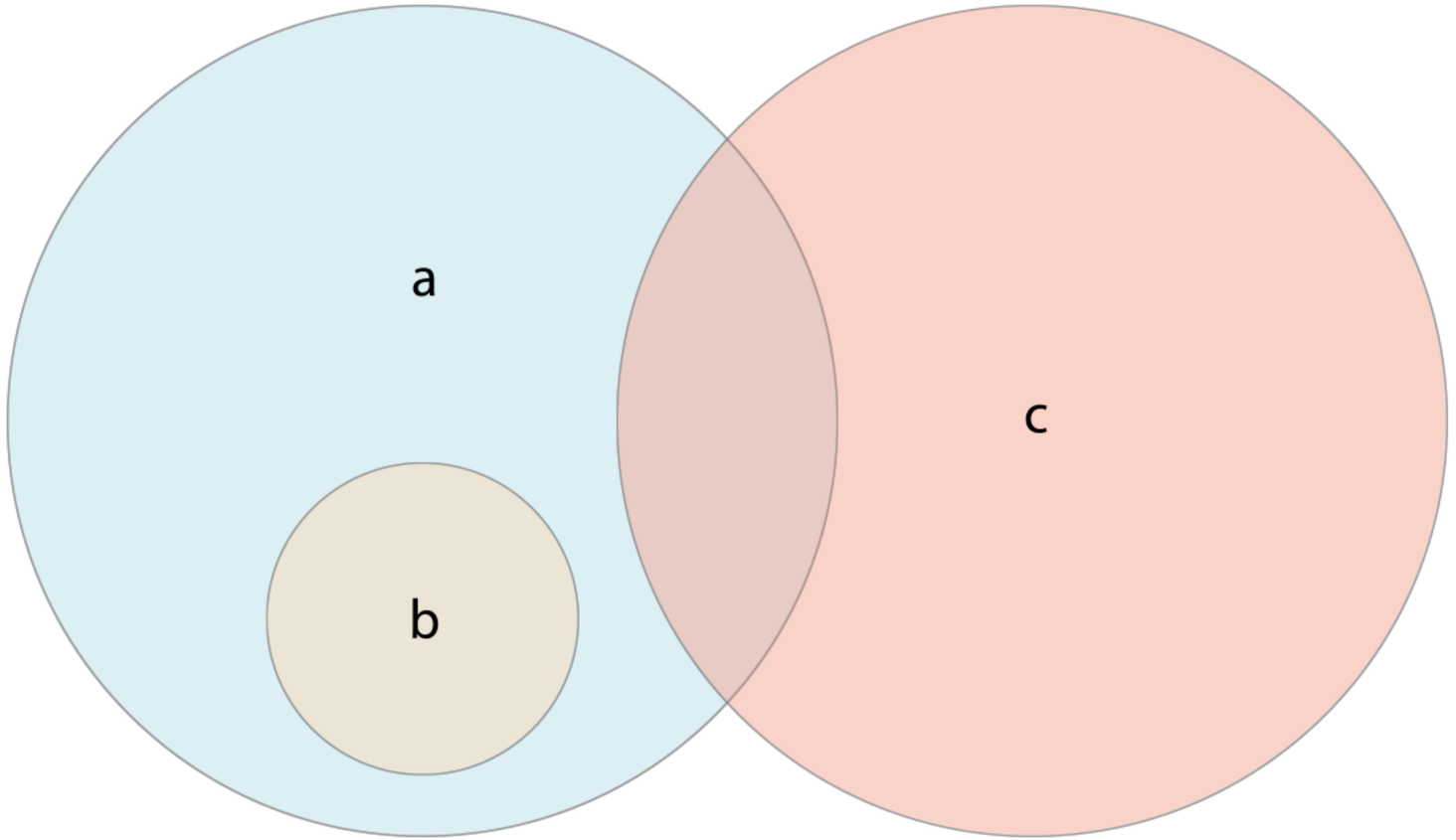
Operaciones entre conjuntos

- La operación intersection(_:) crea un nuevo conjunto que contiene sólo los valores comunes a los dos conjuntos
- La operación symmetricDifference(_:) crea un nuevo conjunto con los valores que no sean comunes a los dos conjuntos
- La operación union(_:) crea un nuevo conjunto con los valores comunes a los dos conjuntos
- La operación subtracting(_:) crea un nuevo conjunto con los valores que no estén en el conjunto especificado

Operaciones entre conjuntos

```
let oddDigits: Set = [1, 3, 5, 7, 9]
let evenDigits: Set = [0, 2, 4, 6, 8]
let singleDigitPrimeNumbers: Set = [2, 3, 5, 7]

oddDigits.union(evenDigits).sorted()
// [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

oddDigits.intersection(evenDigits).sorted()
// []

oddDigits.subtracting(singleDigitPrimeNumbers).sorted()
// [1, 9]

oddDigits.symmetricDifference(singleDigitPrimeNumbers).sorted()
// [1, 2, 9]
```

Pertenencia e igualdad de conjuntos



Pertenencia e igualdad de conjuntos

- El operador == permite comprobar si dos conjuntos contienen los mismos valores
- La operación isSubset(of:) determina si todos los valores de un conjunto están contenidos en otro
- La operación isSuperset(of:) determina si un conjunto contiene todos los valores de otro
- Las operaciones isStrictSubset(of:) o isStrictSuperset(of:) determinan si un conjunto es un subconjunto o superconjunto, pero no igual, a un conjunto dado
- La operación isDisjoint(with:) determina si dos conjuntos no tienen ningún valor el común

Pertenencia e igualdad de conjuntos

```
let houseAnimals: Set = ["🐶", "🐱"]
let farmAnimals: Set = ["🐮", "🐔", "🐑", "🐶", "🐱"]
let cityAnimals: Set = ["🐳", "🐭"]

houseAnimals.isSubset(of: farmAnimals)
// true

farmAnimals.isSuperset(of: houseAnimals)
// true

farmAnimals.isDisjoint(with: cityAnimals)
// true
```

Diccionarios

Diccionarios

```
var namesOfIntegers = [Int: String]()

namesOfIntegers[16] = "sixteen"

namesOfIntegers = [:]
```

Diccionarios

```
var airports: [String: String] = ["TYO": "Tokyo", "DUB": "Dublin"]

var moreAirports = ["TYO": "Tokyo", "DUB": "Dublin"]
```

Diccionarios

```
print("The dictionary of airports contains \(airports.count) items.")


if airports.isEmpty {
    print("The airports dictionary is empty.")
} else {
    print("The airports dictionary is not empty.")
}
```

Diccionarios

```
airports["LHR"] = "London"  // Añadir un elemento

airports["LHR"] = "London Heathrow"  // Actualizar el elemento

airports["APL"] = "Apple International"
airports["APL"] = nil  // Borrar un elemento
```

Características de los diccionarios

- Almacenan parejas de elementos clave-valor (*key-value*)
- El tipo de la clave tiene que ser "*hashable*" (los tipos básicos lo son)
- Si se declara con let no puede variar el número de elementos ni el contenido una vez inicializado

Operaciones sobre diccionarios

- Se puede preguntar cuantos elementos hay con .count
- Se puede preguntar si está vacío con .isEmpty
- Se puede modificar un valor con .updateValue(_:forKey:) que devuelve el valor antiguo como un opcional (la sintaxis con [], no)
- Se puede eliminar un valor con .removeValue(forKey:) que devuelve el valor antiguo como un opcional (la sintaxis con []=nil, no)

Operaciones sobre diccionarios

```
if let oldValue = airports.updateValue("Dublin Airport", forKey: "DUB") {
    print("The old value for DUB was \(oldValue).")
}
```

```
if let airportName = airports["DUB"] {
    print("The name of the airport is \(airportName).")
} else {
    print("That airport is not in the airports dictionary.")
}
```

Operaciones sobre diccionarios

```
if let removedValue = airports.removeValue(forKey: "DUB") {
    print("The removed airport's name is \(removedValue).")
} else {
    print("The airports dictionary does not contain a value for DUB.")
}
```

Recorrer un diccionario

```
for (airportCode, airportName) in airports {
    print("\(airportCode): \(airportName)")
}
```

Recorrer un diccionario

```
for airportCode in airports.keys {
    print("Airport code: \(airportCode)")
}
```

```
for airportName in airports.values {
    print("Airport name: \(airportName)")
}
```

Extraer los valores a arrays

```
let airportCodes = [String](airports.keys)

let airportNames = [String](airports.values)
```

Funciones y clausuras

Funciones

En computación, una subrutina o subprograma (también llamada procedimiento, función o rutina), como idea general, se presenta como un subalgoritmo que forma parte del algoritmo principal, el cual permite resolver una tarea específica.

https://es.wikipedia.org/wiki/Subrutina

Definición de una función

```
func greet(person: String) -> String {
    let greeting = "Hello, " + person + "!"
    return greeting
}
```

Llamada a la función

```
print(greet(person: "Anna"))

print(greet(person: "Brian"))
```

Características de las funciones

- Tienen nombre
- Disponen de una lista de parámetros
- Disponen de un valor de retorno (opcional)
- En la llamada se añaden argumentos que tienen que encajar con los parámetros de la función

Tipos de funciones

Funciones con parámetros

```
func greetAgain(person: String) -> String {
    return "Hello again, " + person + "!"
}

print(greetAgain(person: "Anna"))
```

Funciones sin parámetros

```
func sayHelloWorld() -> String {
    return "hello, world"
}

print(sayHelloWorld())
```

Funciones con múltiples parámetros

```
func greet(person: String, alreadyGreeted: Bool) -> String {
    if alreadyGreeted {
        return greetAgain(person: person)
    } else {
        return greet(person: person)
    }
}

print(greet(person: "Tim", alreadyGreeted: true))
```

Funciones sin valor de retorno

```
func greet(person: String) {
    print("Hello, \(person)!")
}

greet(person: "Dave")
```

Funciones con valor de retorno

```
func sayHelloWorld() -> String {
    return "hello, world"
}

print(sayHelloWorld())
```

Funciones con múltiples valores de retorno

```
func minMax(array: [Int]) -> (min: Int, max: Int) {
    var currentMin = array[0]
    var currentMax = array[0]
    for value in array[1..<array.count] {
        if value < currentMin {
            currentMin = value
        } else if value > currentMax {
            currentMax = value
        }
    }
    return (currentMin, currentMax)
}
```

Funciones con múltiples valores de retorno

```
let bounds = minMax(array: [8, -6, 2, 109, 3, 71])

print("min is \(bounds.min) and max is \(bounds.max)")
```

Parámetros y argumentos

Parámetros y argumentos

- Nombres de parámetros: para utilizar dentro de la función
- Etiquetas de argumentos: para usarlos al hacer la llamada a la función

Parámetros y argumentos

```
func someFunction(firstParameterName: Int, secondParameterName: Int) {
    // In the function body, firstParameterName and secondParameterName
    // refer to the argument values for the first and second parameters.
}

someFunction(firstParameterName: 1, secondParameterName: 2)
```

Parámetros y argumentos

- Por defecto, los parámetros usan su nombre como etiqueta de argumento
- Todos los parámetros tienen que tener nombres únicos

Etiquetas de argumentos explícitas

```
func someFunction(argumentLabel parameterName: Int) {
    // In the function body, parameterName refers to the argument value
    // for that parameter.
}
```

Etiquetas de argumentos explícitas

```
func greet(person: String, from hometown: String) -> String {
    return "Hello \(person)!  Glad you could visit from \(hometown)."
}

print(greet(person: "Bill", from: "Cupertino"))
// Prints "Hello Bill!  Glad you could visit from Cupertino.
```

Anular una etiqueta de argumento

```
func someFunction(_ firstParameterName: Int, secondParameterName: Int) {
    // In the function body, firstParameterName and secondParameterName
    // refer to the argument values for the first and second parameters.
}

someFunction(1, secondParameterName: 2)
```

Documentación de funciones

```
func someFunction(_ firstParameterName: Int, secondParameterName: Int) {
    // In the function body, firstParameterName and secondParameterName
    // refer to the argument values for the first and second parameters.
}

someFunction(1, secondParameterName: 2)


someFunction(_:secondParameterName:) // En la documentación aparece así
```

Parámetros por defecto

- Permiten fijar un valor para un parámetro si no se incluye en los argumentos de la llamada
- Es conveniente que estén al final de la lista de parámetros

Parámetros por defecto

```
func someFunction(parameterWithoutDefault: Int, parameterWithDefault: Int = 12) {
    // If you omit the second argument when calling this function, then
    // the value of parameterWithDefault is 12 inside the function body.
}

someFunction(parameterWithoutDefault: 3, parameterWithDefault: 6)
// parameterWithDefault is 6

someFunction(parameterWithoutDefault: 4)
// parameterWithDefault is 12
```

Parámetros indeterminados

- Son parámetros que permiten introducir múltiples valores
- Se declaran poniendo ... detrás del tipo de dato
- Los valores tienen que ser del mismo tipo
- Los valores llegan a la función como un array del tipo apropiado
- Sólo puede haber uno y tiene que ser siempre el último de la lista

Parámetros indeterminados

```
func arithmeticMean(_ numbers: Double...) -> Double {
    var total: Double = 0
    for number in numbers {
        total += number
    }
    return total / Double(numbers.count)
}

arithmeticMean(1, 2, 3, 4, 5)
// returns 3.0, which is the arithmetic mean of these five numbers

arithmeticMean(3, 8.25, 18.75)
// returns 10.0, which is the arithmetic mean of these three numbers
```

Parámetros InOut

- Son parámetros cuyo valor puede ser modificado por la función y persiste después de terminar esta (por defecto son constantes)
- Se generan marcando con inout el parámetro
- En la llamada, las variables que se pasan se marcan con &
- No se pueden pasar literales o constantes como parámetros
- No pueden tener valor por defecto ni ser indeterminados

Parámetros InOut

```
func swapTwoInts(_ a: inout Int, _ b: inout Int) {
    let temporaryA = a
    a = b
    b = temporaryA
}

var someInt = 3
var anotherInt = 107
swapTwoInts(&someInt, &anotherInt)

print("someInt is now \(someInt), and anotherInt is now \(anotherInt)")
// Prints "someInt is now 107, and anotherInt is now 3
```

Funciones como tipos de datos

Funciones como tipos de datos

- Toda función tiene tipo
- Está definido por los tipos de los parámetros y el tipo del valor de retorno

Tipo de dato de una función

```
func addTwoInts(_ a: Int, _ b: Int) -> Int {
    return a + b
}

func multiplyTwoInts(_ a: Int, _ b: Int) -> Int {
    return a * b
}

// (Int, Int) -> Int
```

Tipo de dato de una función

```
func printHelloWorld() {
    print("hello, world")
}

// () -> Void
```

Utilizar funciones como tipos

```
var mathFunction: (Int, Int) -> Int = addTwoInts
```

Utilizar funciones como tipos

```
func addTwoInts(_ a: Int, _ b: Int) -> Int {
    return a + b
}

var mathFunction: (Int, Int) -> Int = addTwoInts

print("Result: \(mathFunction(2, 3))")
```

Utilizar funciones como tipos

```
mathFunction = multiplyTwoInts
print("Result: \(mathFunction(2, 3))")
```

```
let anotherMathFunction = addTwoInts
```

Tipos de función como parámetros

- Podemos definir un parámetro de una función del tipo de otra función
- Permite que la implementación de la función varíe dependiendo de lo que le pasemos como parámetro (que será una función)

Tipos de función como parámetros

```
func printMathResult(_ mathFunction: (Int, Int) -> Int, _ a: Int, _ b: Int) {
    print("Result: \(mathFunction(a, b))")
}

printMathResult(addTwoInts, 3, 5)
// Prints "Result: 8

printMathResult(multiplyTwoInts, 3, 5)
// Prints "Result: 15
```

Tipos de función como valor de retorno

- Podemos definir el valor de retorno de una función del tipo de otra función
- Después de la —> de la función describimos el tipo de función

Tipos de función como valor de retorno

```
func stepForward(_ input: Int) -> Int {
    return input + 1
}

func stepBackward(_ input: Int) -> Int {
    return input - 1
}
```

Tipos de función como valor de retorno

```
func chooseStepFunction(backward: Bool) -> (Int) -> Int {
    return backward ? stepBackward : stepForward
}

var currentValue = 3
let moveNearerToZero = chooseStepFunction(backward: currentValue > 0)
// moveNearerToZero now refers to the stepBackward() function
```

Tipos de función como valor de retorno

```
print("Counting to zero:")
// Counting to zero:
while currentValue != 0 {
    print("\(currentValue)... ")
    currentValue = moveNearerToZero(currentValue)
}
print("zero!")
// 3...
// 2...
// 1...
// zero!
```

Clausuras

En Informática, una clausura es una función que es evaluada en un entorno conteniendo una o más variables dependientes de otro entorno. Cuando es llamada, la función puede acceder a estas variables.

https://es.wikipedia.org/wiki/Clausura_(informática)

Clausuras

- Bloques de funcionalidad autocontenidos
- Capturan referencias a las variables y constantes del ámbito en el que están definidas
- Las funciones son tipos especiales de clausuras
- Las clausuras son tipos por referencia cuando se asignan a variables o se pasan a funciones

Tipos de clausuras

- Funciones globales: clausuras con nombre y que no capturan variables
- Funciones anidadas: clausuras con nombre y que capturan las variables del ámbito de la función que las engloba
- Expresiones de clausura: clausuras sin nombre escritas en una notación simple que pueden capturar las variables de su entorno

Sintaxis de una clausura

```
{ (parameters) -> return_type in
    statements
}
```

Ejemplo: sorted(by:)

- La función sorted(by:) de la librería estándar de Swift ordena un array en función de un criterio expresado por una clausura
- sorted(by:) recibe como parámetros un array de elementos y una clausura que compara dos elementos y devuelve verdadero o falso

Ejemplo: sorted(by:)

```
let names = ["Chris", "Alex", "Ewa", "Barry", "Daniella"]

func backward(_ s1: String, _ s2: String) -> Bool {
    return s1 > s2
}

var reversedNames = names.sorted(by: backward)
// reversedNames is equal to ["Ewa", "Daniella", "Chris", "Barry", "Alex"]
```

Ejemplo: sorted(by:)

```
// Clausura
reversedNames = names.sorted(by: { (s1: String, s2: String) -> Bool in
    return s1 > s2
})


// En una línea
reversedNames = names.sorted(by: { (s1: String, s2: String) -> Bool in return s1 > s2 } )
```

Ejemplo: sorted(by:)

```
// Inferencia de tipos
reversedNames = names.sorted(by: { s1, s2 in return s1 > s2 } )

// Retorno implícito para expresiones de una sola línea
reversedNames = names.sorted(by: { s1, s2 in s1 > s2 } )
```

Ejemplo: sorted(by:)

```
// Nombres de parámetros abreviados
reversedNames = names.sorted(by: { $0 > $1 } )


// Función operador
reversedNames = names.sorted(by: >)
```

Clausuras posteriores

- Se pueden usar cuando la clausura es el último parámetro
- Se suelen usar si el código de la clausura es largo

Clausuras posteriores

```
func someFunctionThatTakesAClosure(closure: () -> Void) {
    // function body goes here
}

// Normal
someFunctionThatTakesAClosure(closure: {
    // closure's body goes here
})

// Clausura posterior
someFunctionThatTakesAClosure() {
    // trailing closure's body goes here
}
```

Clausuras posteriores

```
reversedNames = names.sorted(by: { $0 > $1 } )

reversedNames = names.sorted() { $0 > $1 }

reversedNames = names.sorted { $0 > $1 }
```

Clausuras posteriores

```
let digitNames = [
    0: "Zero", 1: "One", 2: "Two",   3: "Three", 4: "Four",
    5: "Five", 6: "Six", 7: "Seven", 8: "Eight", 9: "Nine"
]

let numbers = [16, 58, 510]

let strings = numbers.map {
    (number) -> String in
    var number = number
    var output = ""
    repeat {
        output = digitNames[number % 10]! + output
        number /= 10
    } while number > 0
    return output
}
// strings is inferred to be of type [String]
// its value is ["OneSix", "FiveEight", "FiveOneZero"
```

Captura de valores

- Una clausura **"captura" los valores del ámbito que la engloba**
- La clausura puede acceder y modificar esos valores

Captura de valores

```
func makeIncrementer(forIncrement amount: Int) -> () -> Int {
    var runningTotal = 0
    func incrementer() -> Int {
        runningTotal += amount
        return runningTotal
    }
    return incrementer
}
```

Captura de valores

```
let incrementByTen = makeIncrementer(forIncrement: 10)

incrementByTen()
// returns a value of 10
incrementByTen()
// returns a value of 20
incrementByTen()
// returns a value of 30
```

Captura de valores

```
let incrementBySeven = makeIncrementer(forIncrement: 7)

incrementBySeven()
// returns a value of 7

incrementByTen()
// returns a value of 40
```

Las clausuras son tipos por referencia

```
let alsoIncrementByTen = incrementByTen

alsoIncrementByTen()
// returns a value of 50
```

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
```

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)
```

Inicializador para clases

```
class VideoMode {
    var resolution = Resolution()
    var interlaced = false
    var frameRate = 0.0
    var name: String?
    init(resolution: Resolution = Resolution(), interlaced: Bool = false, frameRate:
Double = 0.0, name: String? = nil) {
        self.resolution = resolution
        self.interlaced = interlaced
        self.frameRate = frameRate
        self.name = name
    }
}
```

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 calculadas

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

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

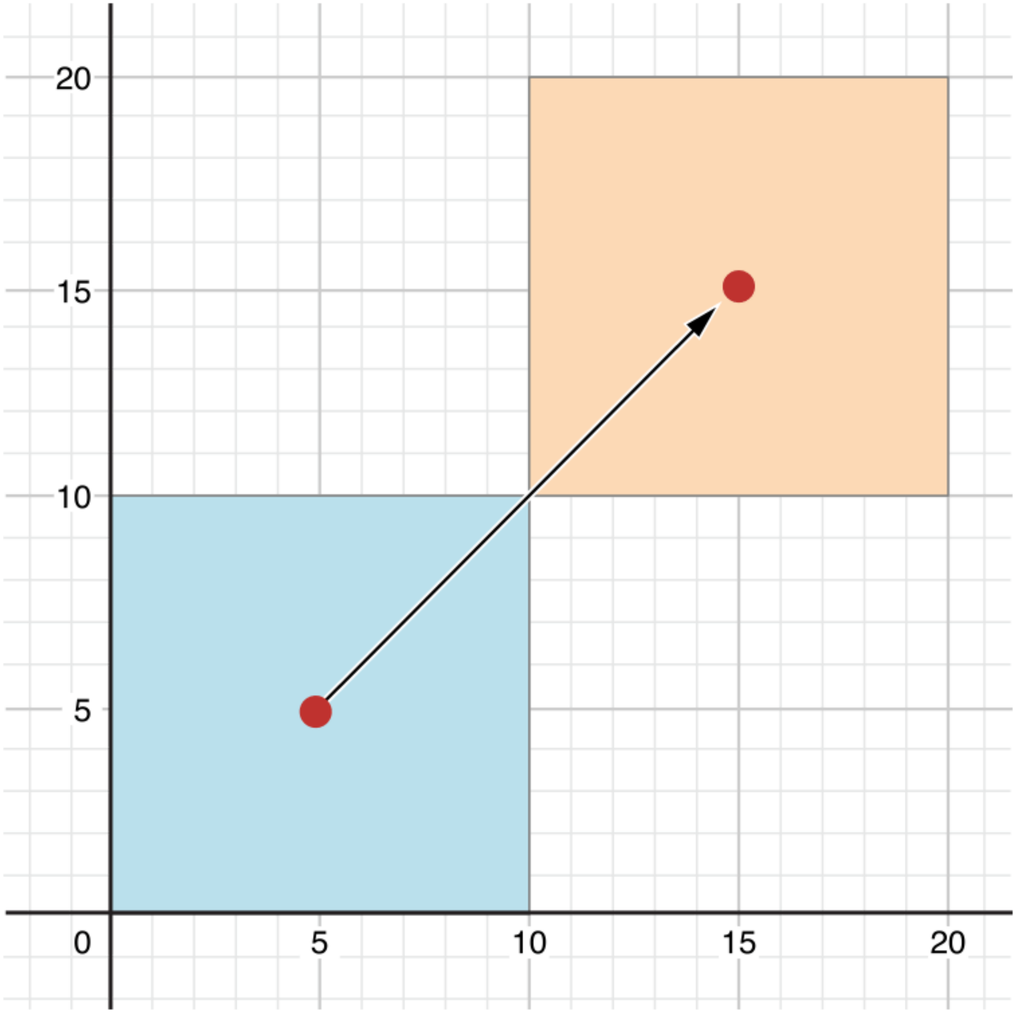
Propiedades calculadas

```
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"
```

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()
```

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

Extensiones y protocolos

Extensiones

Extensiones

- Permiten añadir funcionalidad a un tipo existente
- Cuando se define una extensión a un tipo, todas las instancias de ese tipo reciben la extensión, incluido las que se habían creado antes de la definición

Extensiones

```
extension SomeType {
    // new functionality to add to SomeType goes here
}

extension SomeType: SomeProtocol, AnotherProtocol {
    // implementation of protocol requirements goes here
}
```

Capacidades de las extensiones

- Añadir propiedades calculadas (no almacenadas ni observers)
- Definir métodos de instancia y tipo
- Añadir nuevos inicializadores
- Definir subíndices
- Definir y usar nuevos tipos anidados
- Hacer que un tipo existente adopte un protocolo

Añadir propiedades calculadas

```
extension Double {
    var km: Double { return self * 1_000.0 }
    var m: Double { return self }
    var cm: Double { return self / 100.0 }
    var mm: Double { return self / 1_000.0 }
    var ft: Double { return self / 3.28084 }
}

let oneInch = 25.4.mm
print("One inch is \(oneInch) meters")
// Prints "One inch is 0.0254 meters"

let threeFeet = 3.ft
print("Three feet is \(threeFeet) meters")
// Prints "Three feet is 0.914399970739201 meters"
```

Añadir propiedades calculadas

```
let aMarathon = 42.km + 195.m
print("A marathon is \(aMarathon) meters long")
// Prints "A marathon is 42195.0 meters long"
```

Protocolos

Protocolos

- Permiten definir un listado de métodos, propiedades y otros requisitos que se deben cumplir para garantizar cierta funcionalidad
- No proporcionan la implementación
- Pueden ser adoptados por una clase, estructura o enumeración
- Pueden requerir métodos de instancia o de tipo, propiedades de instancia, operadores y subíndices

Definición de un protocolo

```
protocol SomeProtocol {
    // protocol definition goes here
}
```

Adopción de un protocolo

```
struct SomeStructure: FirstProtocol, AnotherProtocol {
    // structure definition goes here
}

class SomeClass: SomeSuperclass, FirstProtocol, AnotherProtocol {
    // class definition goes here
}
```

Requerir propiedades

```
protocol SomeProtocol {
    var mustBeSettable: Int { get set }
    var doesNotNeedToBeSettable: Int { get }
}

protocol AnotherProtocol {
    static var someTypeProperty: Int { get set }
}
```

Requerir métodos

```
protocol RandomNumberGenerator {
    func random() -> Double
}

protocol SomeProtocol {
    static func someTypeMethod()
}
```