Tipos de datos avanzados

Tuplas

Una tupla es una lista de elementos separados por coma y entre paréntesis. Esta lista puede tener el nombre de los elementos y utilizar esos nombres para referirse al valor de algún elemento en particular. Los elementos en la tupla no tienen que ser del mismo tipo.

Un elemento consiste de un identificador seguido inmediatamente del símbolo dos puntos `:`.

Se puede utilizar una tupla como la devolución de una función de manera que pueda devolver valores de diferentes tipos.

```
let tuple = ("one", 2, "three")

// Values are read using index numbers starting at zero
print(tuple.0) // one
print(tuple.1) // 2
print(tuple.2) // three
```

Además, los valores de la tupla pueden ser nombrados cuando se declare la tupla.

```
let namedTuple = (first: 1, middle: "dos", last: 3)

// Values can be read with the named property
print(namedTuple.first)  // 1
print(namedTuple.middle) // dos

// And still with the index number
print(namedTuple.2)      // 3
```

Los tuplas pueden ser nombrados cuando son utilizados como una variable e inclusive tener la capacidad de tener valores opcionales en su interior.

```
var numbers: (optionalFirst: Int?, middle: String, last: Int)

//Later On
numbers = (nil, "dos", 3)

if let first = numbers.optionalFirst {
    print(first) // nil
}
print(numbers.middle) //"dos"
print(numbers.last) //3
```

Descomponiendo una tupla en variables individuales

Las tuplas pueden ser descompuestas en variables individuales mediante la siguiente sintaxis.

```
let myTuple = (name: "Some Name", age: 26)
let (first, second) = myTuple

print(first) // "Some Name"
print(second)  // 26
```

La sintaxis puede ser utilizada aún y cuando las propiedades de la tupla no hayan sido nombradas.

```
let unnamedTuple = ("uno", "dos")
let (one, two) = unnamedTuple

print(one) // "uno"
print(two) // "dos"
```

Es posible ignorar algunas propiedades específicas utilizando el símbolo `_`:

```
let longTuple = ("ichi", "ni", "san")
let (_, _, third) = longTuple

print(third) // "san"
```

Tuplas como resultado de una función

Una función puede devolver una tupla como resultado, esto mediante la siguiente sintaxis.

```
func tupleReturner() -> (Int, String) {
    return (3, "Hello")
}

let tuple = tupleReturner()

print(tuple.0) // 3
print(tuple.1) // "Hello"
```

Si se asignan nombres a los parámetros, estos pueden ser utilizados como un valor a devolver.

```
func tupleReturner() -> (anInteger: Int, aString: String) {
    return (3, "Hello")
}

let tuple = tupleReturner()
```

```
print(tuple.anInteger) // 3
print(tuple.aString) // "Hello"
```

Alias para una tupla.

Ocasionalmente puede utilizar más de una vez una tupla, lo cual puede volverse confuso sobre todo si la tupla no es simple.

```
import Foundation

// Define a circle tuple by its center point and radius
let unitCircle: (center: (x: CGFloat, y: CGFloat), radius: CGFloat) = ((0.0, 0.0),
1.0)

func doubleRadius(circle: (center: (x: CGFloat, y: CGFloat), radius: CGFloat)) ->
(center: (x: CGFloat, y: CGFloat), radius: CGFloat) {
    return (circle.center, circle.radius * 2.0)
}
```

Un tipo de dato `CGFloat` es una forma especializada de Float que permite almacenar datos de 32 ó 64 bits. Este tipo de dato es parte de Core Graphics y se encuentra en UIKit, Sprite Kit y otras bibliotecas de iOS.

Si se utiliza más de una vez la tupla, conviene usar `typealias` para nombrar la tupla.

```
// Define a circle tuple by its center point and radius
typealias Circle = (center: (x: CGFloat, y: CGFloat), radius: CGFloat)
let unitCircle: Circle = ((0.0, 0.0), 1)

func doubleRadius(circle: Circle) -> Circle {
    // Aliased tuples also have access to value labels in the original tuple type.
    return (circle.center, circle.radius * 2.0)
}

print(doubleRadius(circle: unitCircle))

let newCircle: Circle = ((1,3), 5)
print(doubleRadius(circle: newCircle))
```

Sin embargo, si utiliza esto frecuentemente quizá sea mejor que considere usar una estructura.

Intercambiar valores en una tupla

Si se desea intercambiar dos o más valores en una tupla, se puede hacer sin utilizar variables temporales.

Ejemplo con dos variables.

```
var a = "Marty McFly"
var b = "Emmett Brown"

(a, b) = (b, a)

print(a) // "Emmett Brown"
print(b) // "Marty McFly"
```

Ejemplo con cuatro variables

```
var a = 0
var b = 1
var c = 2
var d = 3

(a, b, c, d) = (d, c, b, a)

print(a, b, c, d) // 3, 2, 1, 0
```

Tuplas en switch

Uso de tuplas mediante un switch

```
let switchTuple = (firstCase: true, secondCase: false)

switch switchTuple {
    case (true, false):
        // do something
        print("TF")
    case (true, true):
        // do something
        print("TT")
    case (false, true):
        // do something
        print("FT")
    case (false, false):
        // do something
        print("FF")
}
```

Diccionarios

Los diccionarios son un conjunto no ordenado de llaves y valores. Los valores están relacionados a llaves únicas y deben ser del mismo tipo.

Para inicializar un diccionario se puede usar la sintáxis completa:

```
var books : Dictionary<Int, String> = Dictionary<Int, String>()
```

Adicionalmente hay formas más concisas de declarar un diccionario:

```
var books = [Int: String]()
// or
var books: [Int: String] = [:]
// or
var books: [Int: String]
```

Es posible declarar un diccionario y asignarle valores iniciales, separados con coma. Los tipos de datos pueden ser inferidos por el valor asignado, por lo que es posible no indicar el tipo explícitamente.

```
var books: [Int: String] = [1: "Book 1", 2: "Book 2"]
//books = [2: "Book 2", 1: "Book 1"]

var otherBooks = [3: "Book 3", 4: "Book 4"]
//otherBooks = [3: "Book 3", 4: "Book 4"]
```

Acceso a valores del diccionario

Un valor dentro del diccionario puede ser accesado mediante su llave correspondiente.

```
var books: [Int: String] = [1: "Book 1", 2: "Book 2"]
let bookName = books[1]
//bookName = "Book 1"
```

Los valores del diccionario pueden ser iterados mediante la propiedad `values`:

```
for book in books.values {
    print("Book Title: \(book)")
}
//output: Book Title: Book 2
//output: Book Title: Book 1
```

De manera similar, las llaves pueden ser iteradas mediante la propiedad `keys`:

```
for bookNumber in books.keys {
    print("Book number: \(bookNumber)")
}
// outputs:
```

```
// Book number: 1
// Book number: 2
```

Ademas, se puede iterar el diccionario y utilizar una tupla si se necesita tanto la llave como el valor de cada elemento:

```
for (bookTitle, bookNumber) in books {
    print("\(bookTitle) -> \(bookNumber)")
}
// output:
// 2 -> Book 2
// 1 -> Book 1
// 3 -> Book 3
```

Observe que a diferencia de un arreglo, los diccionarios son conjuntos no ordenados. Por ello es posible no obtener en la iteración los elementos en el orden que fueron guardados en el diccionario.

```
// Create a multilevel dictionary.
var myDictionary: [String:[Int:String]] =
    ["Toys":
        [1:"Car",2:"Truck"],
    "Interests":
        [1:"Science",2:"Math"]
    ]

if let v = myDictionary["Toys"]?[2] {
    print(v) // Outputs "Truck"
}

if let w = myDictionary["Interests"]?[1] {
    print(w) // Outputs "Science"
}
```

Considere que cuando se usan diccionarios es posible que una llave solicitada no exista en el diccionario. Por esa razón los diccionarios siempre devuelven opcionales. Debido a eso, en el ejemplo se tuvo que desenvolver el opcional devuelto por el diccionario.

Si está seguro de que la llave existe en el diccionario y desea obtener el valor opcional sin hacer un desenvolvimiento, puede entonces forzar la devolución.

```
print(myDictionary["Toys"]?[2]!)
```

Cambiar valores en un diccionario

```
var dict = ["name": "John", "surname": "Doe"]
```

```
// Set the element with key: 'name' to 'Jane'
dict["name"] = "Jane"

print(dict)
```

Obtener las llaves de un diccionario

Si desea obtener las llaves del diccionario, puede hacerlo con el operador `keys` y recibir el resultado como un arreglo. Esto es posible dado que las llaves tienen el mismo tipo de dato.

```
var dict = ["name": "John", "surname": "Doe"]
let allKeys = Array(dict.keys)
print(allKeys)
```

Modificar el diccionario

Si se desea cambiar un valor específico del diccionario, se puede hacer mediante su llave.

```
var books = [Int: String]()
//books = [:]

books[5] = "Book 5"
print(books)
//books = [5: "Book 5"]
```

El método updateValue cambia el valor inidicado en la posición de la llave y devuelve el valor previo.

```
books.updateValue("Book 7", forKey: 5)
//[5: "Book 7"]
print(previousValue)
print(books)
```

Para borrar valores y su respectiva llave se utiliza una sintáxis similar:

```
books[5] = nil

books[6] = "Deleting from Dictionaries"
print(books)
//books = [6: "Deleting from Dictionaries"]

let removedBook = books.removeValue(forKey:6)
print(removedBook)
//removedValue = "Deleting from Dictionaries"
```

```
print(books)
```

Fusionar dos diccionarios

Para fusionar dos diccionarios en uno solo, se puede hacer mediante el método `merge`. Lo que debe decidir es que ocurrirá con los valores de las llaves repetidas en ambos, si se que existen.

Si desea conservar los valores del primer diccionario, se le debe indicar a `merge` como haga la fusión `{(current, _) in current}`:

```
var dictionary1:[String:Int] = ["Mohan":75, "Raghu":82, "John":79]

var dictionary2:[String:Int] = ["Surya":91, "John":80, "Saranya":92]

dictionary1.merge(dictionary2){(current, _) in current}

print("dictionary1\n------------")
for (key, value) in dictionary1 {
    print("(\(key),\(value))")
}

print("\ndictionary2\n------------")
for (key, value) in dictionary2 {
    print("(\(key),\(value))")
}
```

Y si se desea conservar los valores del segundo diccionario, se le indica a `merge` que haga `{(current, _) in current}`:

```
var dictionary1:[String:Int] = ["Mohan":75, "Raghu":82, "John":79]

var dictionary2:[String:Int] = ["Surya":91, "John":80, "Saranya":92]

dictionary1.merge(dictionary2){(_, new) in new}

print("dictionary1\n------------")
for (key, value) in dictionary1 {
    print("(\(key),\(value))")
}

print("\ndictionary2\n------------")
for (key, value) in dictionary2 {
    print("(\(key),\(value))")
}
```

Ejemplo de un diccionario

Dado un diccionario que tiene almacenados arreglos de números diferenciados por su llave, encuentre el valor mayor de todos ellos.

```
let interestingNumbers = [
    "Prime": [2, 3, 5, 7, 11, 13],
    "Fibonacci": [1, 1, 2, 3, 5, 8],
    "Square": [1, 4, 9, 16, 25],
]
var largest = 0
var key:String = ""
for (_, numbers) in interestingNumbers {
    for number in numbers {
        if number > largest {
            largest = number
        }
    }
}
print(largest)
// Prints "25
```

Conjuntos

Un conjunto es una colección de valores únicos no ordenados. Los valores deben ser del mismo tipo.

La declaración de un conjunto es mediante la sintáxis:

```
var colors = Set<String>()
```

Se puede declarar un conjunto conocido de valores mediante la sintáxis de arreglo.

```
var favoriteColors: Set<String> = ["Red", "Blue", "Green", "Blue"]
// {"Blue", "Green", "Red"}
```

Operaciones básicas con conjuntos

Crear un conjuto con elementos iniciales.

```
// create a set of integer type
var studentID : Set = [112, 114, 116, 118, 115]

print("Student ID: \(studentID)")
```

Añadir elementos a un conjunto, se utiliza el método `insert()`.

```
var numbers: Set = [21, 34, 54, 12]

print("Initial Set: \(numbers)")

// using insert method
numbers.insert(32)

print("Updated Set: \(numbers)")
```

Eliminar elementos de un conjunto, se utiliza el método `remove()`.

```
var languages: Set = ["Swift", "Java", "Python"]

print("Initial Set: \(languages)")

// remove Java from a set
let removedValue = languages.remove("Java")

print("Set after remove(): \(languages)")
print("Value removed: \(removedValue)")
```

Para iterar un conjunto se puede utilizar el ciclo `for`.

```
let fruits: Set = ["Apple", "Peach", "Mango"]

print("Fruits:")

// for loop to access each fruits
for fruit in fruits {
  print(fruit)
}
```

Métodos adicionales para el manejo de conjuntos.

Método	**Descripción**
removeFirst()	Eliminar el primer elemento
removeAll()	Eliminar todos los elementos
sorted()	Ordena los elementos
forEach()	Realiza una acción a cada elemento
contains()	Busca el elemento indicado en el conjunto
randomElement()	Devuelve un elemento (tipo opcional) aleatorio
firstIndex()	Devuelve el índice del elemento dado

Método	Descripción
count()	Devuelve la canitdad de elemento en el conjunto

Operaciones avanzadas con conjuntos

Intersección

Se puede utilizar el método `intersection(_:)` para crear un nuevo conjunto que contenga todos los valores comúnes de dos conjuntos origen.

- Ejemplo 1.

```
let favoriteColors: Set = ["Red", "Blue", "Green"]
let newColors: Set = ["Purple", "Orange", "Green"]
let intersect = favoriteColors.intersection(newColors) // a AND b
// intersect = {"Green"}

print(intersect)
```

- Ejemplo 2.

```
var A : Set = ["a", "c", "d"]
var B : Set = ["c", "b", "e" ]
var C : Set = ["b", "c", "d"]

// compute intersection between A and B
print("A n B =", A.intersection(B))


// compute intersection between B and C
print("B n C =", B.intersection(C))
```

- Ejemplo 3.

```
// create a set that ranges from 1 to 4
var total = Set(1...10)

// compute intersection
print(total.intersection([5,10,15]))
```

Unión

Para realizar la unión entre dos conjuntos se puede usar el método union(_:), el cual creará un conjunto nuevo con los valores de ambos conjuntos sin repeticiones. * Ejemplo 1.

```
let favoriteColors: Set = ["Red", "Blue", "Green"]
```

```
let newColors: Set = ["Purple", "Orange", "Green"]

let allColors = favoriteColors.union(newColors)
print(allColors)
```

- Ejemplo 2.

```
var A : Set = ["a", "c", "d"]
var B : Set = ["c", "b", "e" ]
var C : Set = ["b", "c", "d"]

// compute union between A and B and C
let D = A.union(B).union(C)
print(D)
```

- Ejemplo 3.

```
// create a set that ranges from 1 to 15
var total = Set(1...10)

// compute union
print(total.union(Set(5...15)))
```

Diferencia

La diferencia entre dos conjuntos es el conjunto formado por los elementos del primer conjunto y que no se encuentran en el segundo. Se debe usar el método `subtracting(_:)`.

- Ejemplo 1.

```
let favoriteColors: Set = ["Red", "Blue", "Green"]
let newColors: Set = ["Purple", "Orange", "Green"]

let diffColors = favoriteColors.subtracting(newColors)
print(diffColors)
```

- Ejemplo 2.

```
var A : Set = ["a", "c", "d"]
var B : Set = ["c", "b", "e" ]
var C : Set = ["b", "c", "d"]

// compute subtracting between A and B and C
let D = A.subtracting(B).subtracting(C)
print(D)
```

- Ejemplo 3.

```
var total = Set(1...10)

// compute subtracting
print(total.subtracting(Set(5...15)))
```

Diferencia simétrica (Or exclusiva)

Para obtener un conjunto construido con los elementos que no existen en ambos conjuntos, se debe utilizar el método `symmetricDifference()`.

- Ejemplo 1.

```
let favoriteColors: Set = ["Red", "Blue", "Green"]
let newColors: Set = ["Purple", "Orange", "Green"]

let xorColors = favoriteColors.symmetricDifference(newColors)
print(xorColors)
```

- Ejemplo 2.

```
var A : Set = ["a", "c", "d"]
var B : Set = ["c", "b", "e" ]
var C : Set = ["b", "c", "d"]

// compute symmetric difference between A and B and C
let D = A.symmetricDifference(B).symmetricDifference(C)
print(D)
```

- Ejemplo 3.

```
// create a set that ranges from 1 to 4
var total = Set(1...10)

// compute symmetric difference
print(total.symmetricDifference(Set(5...15)))
```

Subconjunto de un conjunto

Para verificar si un conjunto es subconjuto de otro, se utiliza el método `isSubset()`.

```
// first set
let setA: Set = [1, 2, 3, 5, 4]
print("Set A: ",  setA)
```

```
// second set
let setB: Set = [1, 2]
print("Set B: ",  setB)

// check if setB is subset of setA or not
print("Subset: ", setB.isSubset(of: setA))
```

Verificar si dos conjuntos son iguales

Para determinar si dos conjuntos son iguales, basta con utilizar el operador `==`.

```
let setA: Set = [1, 3, 5]
let setB: Set = [3, 5, 1]

if setA == setB {
  print("Set A and Set B are equal")
}
else {
  print("Set A and Set B are different")
}
```

Crear un conjunto vacío

Para la creación de un conjunto vacío, basta con utilizar la sintáxis:

```
var emptySet = Set<Int>()
print("Set:", emptySet)
```

Estructuras

Las estructuras se utilizn para almacenar variables de diferentes tipo de dato.

Sintáxis de la definición de una estructura:

```
struct StructureName {
  // structure definition
}
```

Un ejemplo de una estructura llamada `Person` que contiene dos variables `name` y `age`, de tipo `String` e `Int` respectivamente. Las variables, llamadas propiedades en el contexto de la estructura, tienen valores iniciales.

```
struct Person {

  var name = ""
  var age = 0
```

```
}
```

La estrucutra es una plantilla de variables, para usarla se necesita crear una instancia. Esto se hace indicando el nombre de la estructura como un tipo de dato y el inicializador `()`.

```
struct Person {

  var name = " "
  var age = 0
}

// create instance of struct
var person1 = Person()
```

Para tener acceso a los valores a las propiedades de la estructura para lectura o escritura se debe usar el operador `.`.

```
// define a structure
struct Person {

// define two properties
 var name = ""
 var age = 0
}

// create instance of Person
var person1 = Person()

// access properties and assign new values
person1.age = 21
person1.name = "Rick"

print("Name: \(person1.name) and Age: \( person1.age) ")
```

Es posible crear múltiples instancias de una estructura.

```
// define a structure
struct Student {

// define a property
var studentID = 0
}

// instance of Person
var student1 = Student()

// access properties and assign new values
```

```
student1.studentID = 101

print("Student ID: \(student1.studentID)")

// another instance of Person
var student2 = Student()

// access properties and assign new values
student2.studentID = 102

print("Student ID: \(student2.studentID)")
```

Se puede inicializar una estructura a nivel de miembro, es decir, cuando se define se puede además asignarle valores iniciales.

```
struct Person {

// properties with no default values
var name: String
var age: Int
}

// instance of Person with memberwise initializer
var person1 = Person(name: "Kyle", age: 19)

print("Name: \(person1.name) and Age: \( person1.age)")
```

En Swift es posible definir funciones dentro de una estructura. Una función dentro de una estructura se llama método.

```
struct Car {

  var gear = 0

  // method inside struct
  func applyBrake() {
    print("Applying Hydraulic Brakes")
  }
}

// create an instance
var car1 = Car()

car1.gear = 5

print("Gear Number: \(car1.gear)")
// access method
car1.applyBrake()
```

Enums: Enumeraciones básicas

Un `enum` define un conjunto de valores fijos y relacionados entre si. Por ejemplo, se crea un enum Season y con cuatro valores enum.

```
enum Season {
    case spring
    case summer
    case autumn
    case winter
}
```

Una sinátix equivalente más sintética es la siguiente:

```
enum Season {
  case spring, summer, autumn, winter
}
```

Para crear variables tipo `enum`, donde dicha variable sólo podrá tomar los `valores enum` contenidos por la definición del `enum` correspondiente. Por ejemplo, se crea una variable enum llamada currentSeason que sólo puede tomar los valores enum `{spring, summer, autumn, winter}`.

```
var currentSeason: Season
```

Un ejemplo completo de la definición de un `enum`, sus `valores enum`, la declaración de una variable y asignación de un valor enum.

```
// define enum
enum Season {

  // define enum values
  case spring, summer, autumn, winter
}

// create enum variable
var currentSeason: Season

// assign value to enum variable
currentSeason = Season.summer

print("Current Season:", currentSeason)
```

Se puede utilizar un switch para el manejo de los `enum`.

```
enum PizzaSize {
```

```
  case small, medium, large
}

var size = PizzaSize.medium

switch(size) {
  case .small:
    print("I ordered a small size pizza.")

  case .medium:
    print("I ordered a medium size pizza.")

   case .large:
     print("I ordered a large size pizza.");
}
```

Observe que el los cases del switch no se hace una referencia al nombre del `enum` correspondiente, incluir el nombre es opcional. Esto ayuda a tener un código más limpio.

Es posible iterar sobre los enum cases mediante el ciclo for, es necesario utilizar el método `allCases`. Además es necesario agregar el parámetro `CaseIterable` al momento de definir el enum.

```
// conform Languages to caseIterable
enum Season: CaseIterable {
  case spring, summer, autumn, winter
}

// for loop to iterate over all cases
for currentSeason in Season.allCases {
  print(currentSeason)
}
```

En Swift es posible asignar valores específicos para cada `valor enum`, esto se llama valores *crudos*.

```
enum Size : Int {
  case small = 10
  case medium = 12
  ...
}
```

Por ejemplo, aquí los tamaños small, medium y large tiene asociado una medida numérica entera. Nótese que se debe añadir el método `rawValue` para tener acceso al valor crudo.

```
enum Size : Int {
  case small = 10
  case medium = 12
  case large = 14
```

```
}

// access raw value of python case
var result = Size.small.rawValue
print(result)
```

En Swift se puede incluir información adicional a cada caso enum, por ejemplo, dado el enum Laptop:

```
enum Laptop {

  // associate value
  case name(String)
  ...
}
```

En este enum, el valor name tiene asociado un tipo String que puede ser asignado posteriormente.

```
Laptop.name("Razer")
```

En este ejemplo se ha asociado el valor `Razer` al valor enum llamado `name`.

```
enum Laptop {

  // associate string value
  case name(String)

  // associate integer value
  case price (Int)
}

// pass string value to name
var brand = Laptop.name("Razer")
print(brand)

// pass integer value to price
var offer = Laptop.price(1599)
print(offer)
```

Referencias

Programación Swift