Breve revisión del lenguaje de desarrollo Swift

Por: Tomás Balderas Contreras

Resumen

Swift es un lenguaje de desarrollo orientado a objetos moderno, flexible y robusto que incorpora abstracciones y elementos sintácticos que han probado ser de gran utilidad. El objetivo del presente documento es describir los elementos y las construcciones del lenguaje Swift más relevantes, con la intención de hacer más comprensible una posterior lectura a profundidad.

Introducción

Swift es un lenguaje de programación para desarrollar aplicaciones nativas para los ambientes operativos OS X e iOS. Apple Inc. comenzó a desarrollar Swift en 2010 con la finalidad de reemplazar, eventualmente, al lenguaje Objective-C como herramienta principal de desarrollo. Swift ha sido utilizado desde mediados de 2014 dentro del ambiente XCode, exclusivamente, aunque hay planes de que funcione en ambientes operativos distintos a iOS y OS X.

La intención del presente reporte es describir los elementos constitutivos del lenguaje utilizando ejemplos simples y prácticos. Se asume que la persona que lee éste manuscrito está familiarizada con los conceptos básicos del paradigma de orientación a objetos y cuenta con experiencia en el uso de algunos de los lenguajes Smalltalk, Java, Objective-C, C# o C++.

Clases, estructuras y enumeraciones

Las *clases* y las *estructuras* permiten definir nuevos tipos de datos. Estos elementos del lenguaje encapsulan *propiedades* (variables de instancia) y operaciones (*métodos*), indican cómo inicializar nuevas instancias y cómo limpiar la información en instancias existentes antes de eliminarlas; además, las clases y las estructuras pueden implementar *protocolos* (interfaces). Es posible definir relaciones de generalización entre clases mediante el mecanismo de herencia, lo cual no aplica para las estructuras. Las instancias de estructuras se almacenan por valor, mientras que las instancias de las clases se almacenan por referencia.

La Figura 1 muestra las declaraciones de la clase `Employee` y de la estructura `PersonalInformation`, empleadas, respectivamente, para describir empleados en una línea aérea y sus correspondientes datos personales. Cada uno de estos elementos declara algunas propiedades cuyos tipos se indican explícitamente. La clase `Employee` especifica un método para inicializar las propiedades de sus instancias (`init()`) y un método que sobrecarga el operador de igualdad (`==`) para definir un criterio de comparación entre instancias de la clase. La sobrecarga del operador de igualdad es necesaria para que la clase pueda implementar adecuadamente los protocolos `Equatable` y `Hashable`, la cual es una condición necesaria para construir conjuntos de instancias de `Employee`, como se discutirá más adelante.

Una *enumeración* agrupa, bajo un mismo identificador, un conjunto de valores relacionados denominados *miembros*. En el lenguaje C cada miembro está enumerado consecutivamente por un valor entero comenzando en cero. Swift da la opción de asignar un valor arbitrario (*valor bruto*) a cada miembro de una enumeración o de no asignar valores a ningún miembro. La enumeración `Gender` en la Figura 1 define dos miembros (`Male` y

Female), utilizados para designar género, a los que no asigna valores brutos. Una definición alternativa que asigna valores brutos a los miembros es la siguiente:

```
enum Gender: String {
    case Male = "Male"
    case Female = "Female"
}
```

Note que Swift permite al desarrollador asignar valores arbitrarios y no necesariamente enteros a los miembros de una enumeración, lo que demuestra la flexibilidad del lenguaje. En éste caso es necesario especificar el tipo de los valores brutos después de dos puntos (:) en la declaración del nombre de la enumeración.

```
import UIKit

enum Gender: Printable {
 case Female
 case Male
 var description: String {
  switch self {
   case .Male: return "Male"
   case .Female: return "Female"
  }
 }
}
```

```
struct PersonalInformation {
 var name: String
 var address: String
 var dateOfBirth: NSDate
 var age: Int {
  get {
   var today = NSDate()
   var aTimeInterval =
                today.timeIntervalSinceDate(dateOfBirth)
   if aTimeInterval >= 0 {
    return Int(aTimeInterval / 3.1536e7)
   } else {
    return 0
   }
  }
 }
 var gender: Gender
 var SSN: String
}
```

```
class Employee: Equatable, Hashable {
 var ID: Int
 var personalInformation: PersonalInformation
 var wage: Double
 init(ID: Int, personalInformation: PersonalInformation, wage: Double) {
  self.ID = ID
  self.personalInformation = personalInformation
  self.wage = wage
 }
 var description: String {
  return "My name is \(personalInformation.name), ID: \(ID), and work at this airline"
 }
 var hashValue: Int {
        get {
            return self.ID
        }
 }
}
func ==(lhs: Employee, rhs: Employee) -> Bool {
    return lhs.ID == rhs.ID
}
```

Figura 1. Definición de clases, estructuras y enumeraciones en Swift.

Otra característica de las enumeraciones en Swift es que los miembros de una enumeración pueden tener *datos asociados*, que es un concepto distinto al de los valores brutos. Los miembros de la siguiente enumeración denotan diferentes sistemas de coordenadas y, demás, pueden almacenar coordenadas utilizando *tuplas*:

```
enum Coordinate {
    case Cartesian(Float, Float)
    case Polar(Float, Int)
}

var aCoordinate = Coordinate.Cartesian(123.0, 180.0)
aCoordinate = .Polar(120.0, 45)
```

La variable `aCoordinate` es una instancia de la enumeración `Coordinate` que inicialmente denota una coordenada cartesiana ubicada en el punto (123,180) y posteriormente se modifica para indicar una coordenada polar localizada en el punto (120,45º).

Variables y constantes

Swift permite declarar *variables* y *constantes* en cualquier parte de un archivo de código, ya sea dentro de funciones, métodos y bloques de código o fuera de la definición de cualquier construcción del lenguaje. En Swift se utiliza la palabra reservada `var` para declarar variables, cuyo valor puede cambiar en el tiempo, y la palabra reservada `let` para declarar constantes, cuyo valor no puede cambiar después de haber sido establecido. Las siguientes sentencias que declaran la variable `aGender` y la constante `anotherGender` se pueden escribir en cualquier lugar de un archivo de código después de la definición de la estructura `Gender`:

```
var aGender: Gender = .Female
aGender = .Male
let anotherGender: Gender = .Male
anotherGender = .Female  // Esta sentencia genera un error de compilación
```

Swift es un lenguaje con tipificación estricta que ordena que todas las variables y constantes deben pertenecer a un tipo de datos. En las sentencias anteriores la variable `aGender` y la constante `anotherGender` se declaran explícitamente como instancias de la enumeración `Gender`, por lo que se les puede asignar los valores `.Female` y `.Male` sin especificar que pertenecen a dicha enumeración. Note que en Swift las sentencias que ocupan diferentes líneas de texto no necesitan separarse con punto y coma (;). Sin embargo, cuando dos sentencias ocupan la misma línea de texto entonces sí es necesario delimitarlas.

El lenguaje Swift permite que el compilador *infiera el tipo* de una variable o constante de acuerdo al contexto. Como ejemplo considere la siguiente sentencia de declaración y asignación:

```
var aGender = Gender.Female
aGender = .Male
```

En este caso, el compilador determina que el valor a la derecha del operador de asignación (=) es una instancia de la enumeración `Gender` e infiere que la variable `aGender` también debe ser una instancia de la enumeración `Gender`. Ahora, considere las siguientes sentencias:

```
var aString = "Hello World!"
let anInteger = 6152
```

Aquí se tienen una variable y una constante que no están tipificadas explícitamente pero cuyos tipos se infieren a partir de los tipos de los valores a la derecha del operador de asignación, que son `String` e `Int`, respectivamente.

Los tipos en Swift se clasifican en *tipos por valor* y *tipos por referencia*. Las variables con tipo por valor almacenan todo su contenido en el espacio en memoria reservado para ellas, por lo que la asignación de una variable a otra crea una copia del espacio en memoria de la primera variable. Los tipos numéricos, cadenas y estructuras son tipos por valor. Las variables con tipo por referencia son apuntadores al espacio en memoria que almacena su contenido, por lo que la asignación de una variable a otra establece un nuevo apuntador al mismo espacio en memoria. Todas las clases definidas en Swift son tipos por referencia.

Propiedades en clases y estructuras

La sintaxis para definir propiedades dentro de una clase o estructura es idéntica a la sintaxis utilizada para declarar variables y constantes, como se muestra en la Figura 1. Swift permite declarar *propiedades variables* y *propiedades constantes* que se almacenan en el espacio de direcciones en memoria de la clase o estructura. Para éste tipo de propiedades también aplica el mecanismo de inferencia de tipos y la sintaxis para inicialización, la cual es muy importante cuando se declaran clases. Las propiedades declaradas por una clase se deben inicializar explícitamente, a menos que dicha clase defina un método `init()` para tal propósito, como es el caso de la clase `Employee` en la Figura 1.

Swift permite declarar *propiedades calculadas*, las cuales se listan en la declaración de una clase o estructura y tienen un tipo asociado pero no se almacenan en memoria. La Figura 1 muestra que la propiedad `age`, de tipo entero, en la estructura `PersonalInformation` es una propiedad calculada, así como la propiedad `hashValue` en la clase `Employee`. El concepto de propiedad calculada es similar al concepto de *propiedad derivada,* utilizado ampliamente en la especificación del lenguaje UML (Unified Modeling Language)

Cuando se realiza la lectura de una una propiedad calculada se ejecuta el código en la sección `get` de su declaración, el cual utiliza los valores de otras propiedades de la clase o estructura para calcular el valor actual de la propiedad. Cuando se escribe a una propiedad calculada se ejecuta el código en la sección `set` de su declaración, el cual asigna nuevos valores a otras propiedades de la clase o estructura para mantener la coherencia entre todas las propiedades. Cada vez que se lee la propiedad `age` en `PersonalInformation`, su valor se calcula a partir del año actual y del valor de la propiedad `dateOfBirth`, lo cual hace innecesario almacenar y actualizar una variable para la edad del empleado.

Funciones y cerraduras

El lenguaje Swift permite definir funciones en cualquier lugar de un archivo de código, siempre y cuando sea después de haber declarado los tipos de datos de sus parámetros y variables. Además de describir las sentencias que conforman el cuerpo de una función, Swift permite definir *tipos función* para declarar variables que tienen funciones como valores. Se puede establecer cierto paralelismo entre un tipo función y un apuntador a función en el lenguaje C; sin embargo, como el concepto de tipo función está a un nivel de abstracción más alto que el de apuntador a función, la sintaxis para declarar tipos función es más elegante, simple y clara.

La Figura 2 declara la función `getBound()` que regresa el valor máximo o mínimo en un arreglo de valores enteros dependiendo de la función que recibe como entrada a través del parámetro `test`. A continuación se definen las funciones `greaterThan()` y `lessThan()` que comparan sus parámetros, regresan un valor de tipo `Bool` (verdadero o falso) y se usan como entrada a `getBound()` posteriormente. Después, se asigna la función greaterThan() a la variable `aFunction` de tipo función `(Int, Int) -> Bool`. A continuación se invoca a `getBound()`, con el arreglo a inspeccionar y la función que realiza la comparación en la variable `aFunction` como parámetros, para encontrar el valor máximo en el arreglo de entrada. Finalmente, las últimas dos sentencias invocan nuevamente a `getBound()` para obtener el valor mínimo en el arreglo.

Además de permitir que las funciones acepten funciones o variables de tipo función como argumento, Swift también permite que una función regrese un valor de tipo función. La función `chooseFunction()` en la Figura 3 verifica el valor de su único parámetro y entonces decide cuál de las dos funciones definidas en la Figura 2 debe regresar. Note que el

tipo función devuelto por la función `chooseFunction()` es el mismo que el del parámetro `test` de la función `getBound()`, es decir `(Int, Int) -> Bool`.

```
func getBound(anArray: [Int], test: (Int, Int) -> Bool) -> Int {
    var bound = anArray[0]
    for item in anArray[1..<anArray.count] {
        if test(item, bound) {
            bound = item
        }
    }
    return bound
}

func greaterThan(a: Int, b: Int) -> Bool {
    return a > b
}

func lessThan(a: Int, b: Int) -> Bool {
    return a < b
}

let anArray = [52, -234, 177, 23, -545, 16, -8, 23, 3, -1]
var aFunction = greaterThan
var bound = getBound(anArray, aFunction)    // bound = 177

aFunction = lessThan
bound = getBound(anArray, aFunction)        // bound = -545
```

Figura 2. Funciones que aceptan valores de tipo función en Swift.

```
func chooseFunction(select: Int) -> (Int, Int) -> Bool {
    return select == 1 ? greaterThan : lessThan
}

aFunction = chooseFunction(2)
aFunction(4, 5)                             // true
```

Figura 3. Funciones que regresan valores de tipo función.

Otra característica relevante del leguaje Swift es la declaración de funciones que regresan más de un valor, lo cual es posible gracias al concepto de tupla, mencionado anteriormente. También es posible pasar cualquier tipo de tupla a una función como parámetro, declarar tuplas como variables dentro de funciones y declarar tuplas cuyos elementos sean funciones, es decir elementos de tipo función. Como ejemplo, considere las siguientes sentencias:

```
func rearrange(input: (Int, Int, Int)) -> (Int, Int, Int) {
    return (input.2, input.1, input.0)
}

var aTuple = (23, 45, 67)
var anotherTuple = rearrange(aTuple)      // anotherTuple = (67, 45, 23)
```

La función rearrange() recibe como entrada una tupla con tres elementos de tipo entero, intercambia el primer elemento y el último elemento en dicha tupla y regresa la tupla resultante. Las tuplas pueden tener cualquier número de elementos de cualquier tipo de datos proporcionado por Swift o definido por el desarrollador.

Otro elemento del lenguaje Swift interesante y útil es la *cerradura*, que denota un bloque de código auto-contenido que puede ser asignado a una variable, enviado como argumento a alguna función e invocado para que se ejecute. Se puede establecer un

paralelismo entre éste concepto y el bloque de código en Smalltalk, que también puede ser manipulado e invocado por diferentes construcciones del lenguaje. Como ejemplo simple de la declaración de una cerradura y su uso considere las siguientes sentencias que transforman un valor entero en su representación como cadena:

```
var aClosure = { (input: Int) -> String in return String(input) }
var aString = aClosure(234)   // aString = ‘‘234’’
```

La sintaxis completa de la cerradura establece el número y los tipos de los parámetros, el tipo del valor de retorno y las sentencias que la conforman. La cerradura consta de un único parámetro entero y una única sentencia que crea una instancia de `String` a partir del parámetro y la regresa. Es posible simplificar la sintaxis de la cerradura aprovechando la inferencia de tipos, como se muestra en las siguientes sentencias:

```
aClosure = { (input: Int) in String(input) }
aString = aClosure(123)       // aString = ‘‘123’’
```

En este caso, debido a que sólo hay una sentencia, el compilador infiere que la instancia de `String` creada a partir del parámetro entero es el valor de retorno de la cerradura. Tal inferencia permite que el compilador también determine que el tipo del valor de retorno es `String`, por lo que se hace innecesario declararlo explícitamente.

Métodos

Los *métodos instancia* son funciones declaradas por una clase, estructura o enumeración que se invocan por las instancias de tales tipos, pueden modificar las propiedades de las instancias e implementan la funcionalidad que se espera del tipo. Los *métodos tipo* se invocan por el tipo mismo y pueden modificar sus *propiedades tipo*, las cuales están vinculadas al tipo, no a sus instancias, por lo que sólo existe una sola copia de estas propiedades.

La Figura 4 muestra la declaración de la clase `Airline` que complementa las clases declaradas en la Figura 1. La clase declara la propiedad `corporateInformation` para almacenar información general de la empresa como su razón social, su dirección postal, la URL de su portal en Internet e información bursátil. Debido a que los datos almacenados en esta propiedad no varían entre instancias de `Airline`, la variable se declara como propiedad tipo mediante el modificador `static` y se inicializa explícitamente al declarar la clase. El método `displayCorporateInformation()` es un método tipo que despliega la información almacenada por la propiedad `corporateInformation` y se invoca enviando un mensaje directamente a la clase `Airline`, como se muestra en la Figura 4.

La clase también declara el método `init()`, que se invoca al crear cada instancia de `Airline` y cuyo propósito es inicializar las propiedades de la instancia. Swift permite declarar diferentes variantes del método `init()` dentro de una misma clase y establece una secuencia en la ejecución de los métodos de inicialización de una clase y su superclase. En el caso de la clase `Airline`, que no tiene superclase, el método `init()` únicamente asigna un valor a la propiedad `employees` y termina su ejecución.

```
class Airline {
    static var corporateInformation = CorporateInformation(
        name: "Bad Airlines",
        address: "Bad Street 666",
        webSiteURL: "www.bad.com",
        sharePrice: 0.00003,
        volume: 10
    )
```

```
    var employees: Set<Employee>
    var numberOfEmployees: Int {
        get {
            return employees.count
        }
    }
    init() {
        employees = []
    }
    func addEmployee(employee: Employee) {
        employees.insert(employee)
    }
    static func displayCorporateInformation() {
        println("\(self.corporateInformation.name) at
                 \(self.corporateInformation.address),
                 URL: \(self.corporateInformation.webSiteURL)")
    }
}

Airline.displayCorporateInformation()
```

Figura 4. Declaración de la clase Airline.

Arreglos, diccionarios y conjuntos

Swift permite agrupar valores mediante tres diferentes variedades de colecciones: *arreglos*, *conjuntos* y *diccionarios*. Los arreglos son colecciones de valores indizados donde es posible almacenar una o más ocurrencias del mismo valor. Los conjuntos son colecciones no indizadas de valores que son todos distintos. Los diccionarios pueden almacenar parejas cuyos elementos son una llave y su valor correspondiente.

Para mostrar algunas de las ventajas del uso de arreglos en Swift considere las siguientes sentencias que realizan varias operaciones sobre un arreglo de números enteros:

```
var anArray = [65, -123, 873, 29, -45, 704, 234, 2, -5]
anArray[0]      // 65
anArray[6...8]  // [234, 2, -5]
anArray[2..<6]  // [873, 29, -45, 704]
anArray.count   // 9
anArray.append(-45)
anArray.count   // 10
anArray = anArray.filter { $0 > 0 }  // [65, 873, 29, 704, 234, 2]
anArray = anArray + [867, 34, 3]
                            // [65, 873, 29, 704, 234, 2, 867, 34, 3]
```

Después de la sentencia de asignación hay tres sentencias que acceden y regresan, respectivamente, el primer elemento del arreglo, el sub-arreglo delimitado por el séptimo y el noveno elementos (índices 6 a 8) del arreglo inicial y el sub-arreglo delimitado por el tercero y el sexto elementos (índices 2 a 5) del arreglo inicial. A continuación se verifica el número de elementos en el arreglo antes y después de añadir un valor entero al final del mismo y se corrobora la precisión de los valores obtenidos. La penúltima sentencia elimina los valores negativos en el arreglo inicial aplicando una cerradura sobre cada elemento y descartando los que no cumplen la condición en la cerradura. Finalmente, la última sentencia añade tres elementos más al final del arreglo utilizando el operador de adición, que opera sobre arreglos.

La clase `Airline` declarada en la Figura 4 almacena la información de sus empleados en un conjunto para garantizar que no hay más de una ocurrencia de cada instancia de la clase `Employee` de la Figura 1. La clase utiliza el método `insert()` para anexar una nueva instancia al conjunto y la propiedad `count` para obtener el número de elementos en el conjunto. Para poder construir conjuntos a partir de instancias de `Employee` es necesario

que dicha clase siga estrictamente las reglas establecidas por los protocolos `Equatable` y `Hashable`. Primero, la clase debe implementar un criterio que determine la igualdad entre instancias de `Employee` mediante la sobrecarga del operador de comparación. Segundo, la clase debe proporcionar un valor a utilizar como llave (`hashValue`) en la tabla hash que almacena el conjunto. En nuestro caso se establece que dos instancias de `Employee` son iguales cuando tienen el mismo valor en su propiedad `ID` y que el campo `ID` es la llave a utilizar para almacenar instancias en la tabla hash.

Los diccionarios en Swift se emplean para representar tablas en las que es necesario buscar un valor requerido a partir de un valor llave. Todos los elementos en un diccionario son pares del tipo [*llave*:*valor*], donde el elemento *llave* se utiliza para acceder al elemento correspondiente en la tabla hash que almacena el diccionario y el elemento *valor* puede ser de cualquier tipo. Como consecuencia de esta definición, la siguiente declaración de diccionario es válida:

```
var anotherDictionary: [String:(Int, Int)]
```

mientras que la siguiente sentencia genera un error de compilación:

```
var anotherDictionary: [(Int, Int):String]
```

La primera declaración es válida porque las instancias del tipo String funcionan sin problemas como llaves para una tabla hash, mientras que una tupla no sirve directamente como una llave.

Con la intención de revisar algunos rasgos de los diccionarios en Swift considere las siguientes sentencias que asignan y manipulan un diccionario cuyos elementos tienen una cadena como llave y un valor de tipo entero:

```
var aDictionary = ["Mercury":1, "Venus":2, "Earth":3, "Mars":4]
aDictionary["Mars"]           // 4
aDictionary["Jupiter"] = 5
aDictionary["Saturn"] = 6
aDictionary.count             // 6
aDictionary.description
   // "[Mars: 4, Venus: 2, Earth: 3, Jupiter: 5, Saturn: 6, Mercury: 1]"
aDictionary.removeValueForKey("Earth")
aDictionary.count             // 5
```

La primera sentencia declara e inicializa un diccionario con cuatro pares no ordenados de datos. La segunda sentencia accede al diccionario para buscar el valor entero correspondiente a la llave “Mars” y lo regresa. La tercera y cuarta sentencias ingresan dos nuevos pares de datos al diccionario, lo que incrementa el número total de elementos en el diccionario a seis. La sexta sentencia regresa una representación del contenido del diccionario en forma de una cadena, lo que permite observar que los elementos no necesariamente se almacenan en el orden en el que fueron definidos o anexados. La séptima sentencia elimina el par cuya llave tiene el valor “Earth”, lo que reduce el número de elementos en el diccionario a cinco.

Opcionales

Uno de los conceptos fundamentales en el lenguaje Swift es el de los *opcionales* pues se puede encontrar en otras construcciones del lenguaje y en muchas clases en la biblioteca de Swift. El concepto de opcional se puede aplicar a cualquier tipo proporcionado por Swift y a cualquier

tipo definido por el desarrollador, independientemente de si es una clase, estructura, enumeración, tupla o colección.

El concepto de opcional se aplica a un tipo de datos X cuando existe el riesgo de que una operación no pueda producir un valor de tipo X como se espera. Por ejemplo, es posible que la función que convierte cadenas en valores enteros transforme "456" en 456, pero no es posible que transforme la cadena "abc" en un valor entero. En este caso es conveniente que la función regrese un opcional entero (`Int?`) en vez de un entero (`Int`), de manera que cuando la función pueda transformar su argumento el resultado se considere existente y se interprete como un valor entero, mientras que cuando no sea posible hacer la transformación el resultado se considere inexistente, lo que se denota con la palabra reservada `nil`.

En un momento de suspicacia se podría pensar que los opcionales son una forma rebuscada de apuntadores. Establecer un paralelismo entre apuntadores a un tipo en lenguaje C y opcionales de un tipo en Swift es natural y se refuerza cuando se construyen estructuras compuestas de nodos enlazados, en donde se utilizan opcionales para establecer los vínculos. Sin embargo, hay que resaltar que no son el mismo concepto y no se pretende que los opcionales sean el sustituto de los apuntadores en Swift.

Como se indicó anteriormente, el opcional de un tipo se declara colocando un signo de interrogación (?) después del nombre del tipo. Para ilustrar el uso del opcional considere una función que calcula recursivamente la suma de los primeros *n* enteros y regresa un opcional entero, lo cual es necesario porque hay casos en los que la suma no tiene sentido ($n \leq 0$). El código de la función es el siguiente:

```
func sum(n: Int) -> Int? {
    if n > 0 {
        if n == 1 {
            return 1
        } else {
            return n + sum(n - 1)!
        }
    } else {
        return nil
    }
}

if var anInt = sum(10) {
    println("Result: \(anInt)")    // "Result: 55"
} else {
    println("Result: nil")
}
```

Cuando el valor del parámetro es menor o igual a cero la función regresa `nil`, pues es el caso en el que se considera que el resultado no existe. Cuando el valor del parámetro es mayor que cero la función verifica si se cumple el caso base o si es necesario realizar el cálculo recursivo. El signo de exclamación (!) colocado inmediatamente después de la llamada recursiva es necesario para indicar que el opcional entero devuelto por la llamada recursiva se debe interpretar como un valor entero, pues se sabe que tal valor existe. La estructura `if` que sigue a la función recursiva expresa una *unión opcional*, la cual asigna el resultado de `sum(10)` a la variable `anInt` y compara tal variable con `nil` en un solo paso. Si la variable es `nil` entonces se ejecuta la sentencia en el bloque `else`, de otra forma se muestra el resultado de la variable entera `anInt`, como ocurre en nuestro ejemplo.

Comentarios finales.

Aunque se describieron algunos rasgos relevantes del lenguaje Swift a lo largo del presente reporte, la revisión no fue de ningún modo exhaustiva y aún queda mucho por discutir sobre éste lenguaje de desarrollo. La persona interesada puede consultar la documentación oficial de Swift para profundizar en su estudio y puede utilizar las herramientas de software disponibles para practicar. Los ejemplos en el libro y en este reporte se pueden verificar utilizando "playgrounds" en el ambiente de desarrollo XCode. También, se recomienda consultar la información referente al desarrollo de aplicaciones utilizando Swift y los kits de desarrollo en OS X e iOS, así como a la compatibilidad entre Swift y Objective-C.

Referencias

1. The Swift Programming Language. Apple Inc. 2014.
2. Using Swift with Cocoa and Objective-C. Apple Inc. 2014.
3. Swift homepage. URL: https://developer.apple.com/swift/

Agradecimientos

El autor desea expresar su más profundo agradecimiento al M.C. Ricardo Ruíz Rodríguez por sus valiosos comentarios y acertadas observaciones. Cualquier error que subsista es responsabilidad exclusiva del autor.

Semblanza del autor

Tomás Balderas recibió los títulos de doctor y maestro en Ciencias Computacionales por el INAOE en 2012 y 2004, respectivamente, y de licenciado en Ciencias de la Computación por la BUAP en el año 2000. Cuenta con experiencia académica e industrial, y se ha desempeñado en distintas universidades y centros de investigación en México, así como en empresas como Intel Tecnología de México y Virtutech AB. Actualmente es ingeniero de software y consultor independiente
LinkedIn: https://mx.linkedin.com/in/tomasbalderas