View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Universidad Distrital de la ciudad de Bogotá: Open Journal Systems

UNIVERSIDAD DISTRITAL FRANCISCO JOSÉ DE CALDAS

Redes de Ingeniería

http://revistas.udistrital.edu.co/ojs/index.php/redes/index
http://dx.doi.org/10.14483/udistrital.jour.redes.2015.2.a01

REDES DE INGENIERÍA [Rompiendo las barreras del conocimiento]

ARTÍCULO DE INVESTIGACIÓN

El futuro de Apple: Swift *versus* Objective-C

The Future of Apple: Swift *Versus* Objective-C

Cristian González García[1] Jordán Pascual Espada[2] B. Cristina Pelayo G-Bustelo[3]
Juan Manuel Cueva Lovelle[4]

Para citar este artículo: González, C., Pascual, J., Pelayo, B. y Cueva, J. (2015). El futuro de Apple: Swift *versus* Objective-C. *Revista Redes de Ingeniería*. 6(2), 6-16.

Recibido: 20-mayo-2015 / Aprobado: 09-noviembre-2015

Resumen

Hace unos meses Apple presentó un nuevo lenguaje de programación para sus plataformas: Swift. Con Swift, Apple pretende atraer a los programadores de los lenguajes de programación basados en la sintaxis de C++ y darles una mayor abstracción, que con Objective-C, para que sea más fácil programar para las plataformas de Apple. Por estas razones, se hace necesario contrastar lo pretendido por Apple y realizar un estudio del lenguaje de programación a fin de contrastar su objetivo. Para ello, se hicieron dos evaluaciones, una cualitativa y otra cuantitativa, con el propósito de verificar en qué medida Swift es un avance respecto a Objective-C.

Palabras clave: lenguaje de programación, programación funcional, programación informática, programación orientada a objetos, software.

Abstract

Few months ago, Apple presented a new programming language: Swift. With Swift, Apple pretends to attract the programmers of the programming languages based on C++ syntax and gives them a higher abstraction than with Objective-C for being easier to programme to Apple's platforms. For these reasons, it is necessary to contrast what is intended by Apple and do a study of the programming language to ascertain their goal. For this purpose, we did two evaluations, firstly a qualitative evaluation and after, a quantitative evaluation to verify in how much Swift is an advance with respect to Objective-C.

Keywords: computer languages, computer programming, functional programming, object oriented programming, programming, software.

1. Ingeniero técnico en Informática de Sistemas; máster en Ingeniería Web, Escuela de Ingeniería Informática, (Universidad de Oviedo, España); doctorando en Ingeniería Web, Redes Sociales y Dispositivos Móviles. Trabaja en el grupo de investigación "Ingeniería Dirigida por Modelos MDE-RG", de la Universidad de Oviedo desde 2012. Contacto:gonzalezgarciacristian@hotmail.com
2. Doctor en ingeniería Informática, Universidad de Oviedo; máster en ingeniería Web; profesor en la Universidad de Oviedo, ha impartido clases y seminarios en diversas empresas y universidades (Universidad Internacional de la Rioja y Universidad Pontificia de Salamanca); ha participado en diversos proyectos de investigación de ámbito nacional y publicado varios libros y más de veinte artículos científicos en revistas internacionales. Contacto: jordansoy@gmail.com
3. Profesora e investigadora del Departamento de Informática de la Universidad de Oviedo; Doctora en Ingeniería Informática, Universidad de Oviedo. Contacto: crispelayo@uniovi.com
4. Ingeniero de Minas, Escuela Técnica de Ingenieros de Minas (Universidad de Oviedo, España); doctor, Universidad Politécnica de Madrid (1990); desde 1985 es profesor del área de Lenguajes y Sistemas del Departamento de Informática en la Universidad de Oviedo (España); miembro con derecho a voto de ACM e IEEE. Contacto: cueva@uniovi.com

INTRODUCCIÓN

Swift es el nuevo lenguaje de programación creado por Apple para las nuevas versiones de los sistemas operativos de sus plataformas: iOS y OS X [1]. Fue lanzado al público el 6 de Junio de 2014, siendo esta primera versión Swift 1.0 GM la analizada en este artículo. La intención de Apple es ofrecer a los programadores un lenguaje mucho más sencillo, fácil, rápido de programar y amigable que Objetive-C [2] para facilitar el desarrollo de aplicaciones para las plataformas de Apple [1], a pesar de su estado beta.

Swift es mucho más sencillo de aprender que Objective-C, pues se inspira en los lenguajes modernos como C++11, JavaScript, C#, Java, F#, Haskell, Go, Scala, etc. Olvidando la sintaxis basada en C, así como la introducida en la creación de Objective-C. Por ejemplo, no hay uso de punteros y tiene un gestor de memoria, lo que simplifica mucho el uso de lenguaje por parte de los desarrolladores. Esto se debe a que la sintaxis de Objective-C apenas evolucionó con el paso de los años y actualmente es muy diferente a la del resto de lenguajes que fueron apareciendo, al contrario de Swift, que se basa en ellos. Esto da como resultado que Swift posea una sintaxis más limpia, fácil de aprender, sencilla y familiar a costa de ser más restrictiva. Además, Swift incorpora nuevas funcionalidades y da soporte a la programación funcional [1].

Una característica del nuevo ecosistema creado por Apple integrándole Swift es la retrocompatibilidad. Permite tanto importar código de Objective-C en Swift como importar código Swift en Objective-C. Para esto, crearon un sistema de ayuda de importación de código en su Entorno de Desarrollo Integrado (IDE, por sus siglas en inglés) Xcode. Esto permite reutilizar el código existente en Frameworks y APIs, así como con los conocimientos para resolver determinados problemas usando algoritmos conocidos. No obstante, ya tiene varias librerías de su antecesor traducidas a código nativo Swift [3].

Debido a la aparición de este nuevo lenguaje se hace indispensable su estudio para comprobar si las afirmaciones de Apple sobre Swift son ciertas. Además, se estudiará si Swift es un lenguaje de programación adaptado a los nuevos tiempos y si puede facilitar el desarrollo de las aplicaciones para las plataformas de Apple [1].

El artículo está dividido de la siguiente manera: en *Cambios internos* se discuten los diferentes cambios internos introducidos en Swift frente a Objective-C. En la sección *Cambios notables* se explica las nuevas características de las variables y las clases. La sección *Paradigma funcional* hablará de ciertos cambios realizados en el lenguaje para incorporar el paradigma funcional. En *Innovación* se realiza una breve explicación de las novedades notables incorporadas a Swift. Posteriormente, en *Evaluación y discusión* se realiza un breve estudio de la sintaxis entre Objective-C y Swift. Por último, la sección de *Conclusiones*.

CAMBIOS INTERNOS

Swift y Objective-C utilizan el mismo compilador, el Low Level Virtual Machine (LLVM, por sus siglas en inglés) creado en la Universidad de Illinois en el año 2000 y que está programado en C++ [4], [5]. Dicho compilador transforma el código Swift en código nativo optimizado para el hardware para el que se desarrolla (Mac, iPhone o iPad) [2].

Sin embargo, no todo es igual e introducen nuevos cambios en la sintaxis para hacerla más parecida a los lenguajes más utilizados [6], [7], simplificarla y añadirle azúcar sintáctico. También incorpora cambios internos frente a Objective-C, como la ausencia de pre-procesador, cambios internos en las colecciones, cambios en el “goto” y en las Clases. Estas similitudes y diferencias serán vistas en esta sección.

ISSN: 2248–762X • Vol. 6, No. 2 (jul-dic 2015). pp. 6-16

Sintaxis

Al igual que en otros lenguajes de programación (JavaScript, Ruby), Swift permite el uso opcional del carácter "punto y coma" (;) al final de una línea. Además, ahora utilizan como operador de acceso el punto ("."), al igual que hacen otros muchos lenguajes en lugar de "[" y "]" como en Objective-C. No obstante, ahora obliga a acotar con llaves "{" y "}" en las estructuras de control, para así evitar problemas de programación, pues antes, en algunos casos, estas eran opcionales.

Con estos cambios, Apple pretende hacer un lenguaje más legible y fácil para los desarrolladores, además de conseguir reducir el número de errores que puedan cometer cuando programen.

Colecciones

A diferencia de Objective-C, que posee tres colecciones (Array, Set y Dictionary), Swift solo incorpora dos (Array y Dictionary). Además, en Swift, se encuentran implementadas con estructuras (struct), lo que difiere respecto a NSArray y NSDictionary de Foundation que están implementadas con clases.

Por ello, en Swift, cuando estas se asignan a una constante, una variable o se pasan como argumento en tiempo de ejecución a una función, internamente se hace una copia de las estructuras. Lo anterior contrasta con Objective-C, pues este pasa una referencia a la instancia existente.

Labeled Statments

Una de las diferencias entre Swift y Objective-C es la instrucción "goto". En Objective-C se podía usar para ir a cualquier punto de la ejecución del programa siempre que estuviese en el mismo ámbito. En Swift esto cambia, pues deja de existir la palabra reservada "goto" e introducen lo que llaman las Labeled Statements [2]. La idea es parecida al "goto", pero para ámbitos más pequeños, como ocurre en C#, Java y PHP: moverse a una etiqueta dentro de los bucles y el switch.

Clases, Structs y Enums

La creación de una clase o "struct" (figura 1) es muy parecida a la sintaxis de los nuevos lenguajes de programación como C++, Java o C#. Además, cambia la separación a un archivo con extensión ".h", el cual incluye las cabeceras de los métodos como hacía Objective-C por la definición completa con el cuerpo como hacen los lenguajes antes citados. Así, en Swift, se define todo el código de una clase o estructura en el mismo fichero.

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

Figura 1. Creación de una Estructura en Swift.

En el ejemplo de una clase de la figura 2, lo primero que se ve es el constructor. Este utiliza un método por defecto llamado "init". Mantiene la misma palabra reservada y un estilo parecido Objective-C. No obstante, hay que especificar a qué atributo se le asigna el valor. A continuación se puede observar los "getters" y "setters". Como se ve, la declaración es muy similar a como se hace en C#. El siguiente paso contiene la función. La cual necesita incluir la palabra reservada "func" al principio y el tipo de retorno al final "-> String".

Debajo de la declaración de la clase está el ejemplo de cómo se instancia un objeto y cómo se trabaja después con este. Para acceder al mismo objeto, sigue utilizando la palabra self, pero deja de lado el operador "->" y facilita el uso, utilizando siempre el operador ".", como ocurre, por ejemplo, en Python, C# y Java. En consecuencia, al reducir la complejidad al eliminar los punteros, se simplificó el acceso a variables y el uso de operadores a utilizar. Esto hace que ahora para instanciar un objeto no haya que utilizar "alloc" y baste con llamar

al constructor deseado. A diferencia de muchos de los lenguajes de programación actuales basados en C++, se omitió el uso de la palabra reservada "new" delante, siguiendo así el estilo de otros lenguajes actuales que no lo utilizan como son Boo, Python y Scala, entre otros.

```
// Clase en Swift
class Person {

    init(name:String){
        self.name = name
    }

    var name: String {
        get {
            return "John Doe"
        }
        set {
            self.name = newValue
        }
    }

    func description() -> String {
        return "My name is \(name)."
    }
}

var person = Person(name:"Cris")
person.name = "Cristian González García"
var description = person.description()
```

Figura 2. Creación de una clase en Swift con sus componentes más habituales

La herencia en Swift se sigue haciendo igual, pero en esta ahora hay que especificar con la palabra reservada "override" cuando se sobrescribe un método, al igual que ocurre en otros lenguajes. Esto sucede también cuando se invoca un método del padre. Se usa la misma palabra reservada, "super", pero ahora se hace uso del operador "."

Swift también permite el evitar que una variable, método o clase sea sobrescrito si a este se le incluye, delante del método padre la palabra reservada "final". Como se puede observar, sigue un sistema similar al de C++ y C#.

Por otro lado, los "enum" siguen funcionando igual que en Objective-C. La diferencia está en que ahora tienen una sintaxis más limpia y parecida a una clase. No obstante, Swift incorporó más potencia a los "enum" al permitir definir códigos de barra, códigos QR y valores en bruto, también conocidos como Raw, de una forma más sencilla [2].

Para dotar de más potencia a todo esto, Apple mejoró lo conocido en Objective-C como "categories" y creó las "extensions" [2]. Por medio de la cual se puede añadir una nueva funcionalidad a una clase, "struct" o "enum" existente a las que no se tiene acceso al código. Cabe resaltar que no permiten sobrescribir las funciones ya existentes.

CAMBIOS NOTABLES

Swift introdujo diversos cambios de cómo se ha de programar, además de nuevas características. Introdujo cambios en las variables, así como nuevos tipos y cambiaron cierto funcionamiento de las clases. Todo esto se explicará más en detalle en esta sección.

Variables

Swift es mucho más restrictivo con el tipado de las variables para así evitar código inseguro [2]. Obliga a que las variables estén inicializadas antes de ser usadas y a que se especifique explícitamente si es una variable (var) o una constante (let), utilizando estas palabras reservadas. También chequea posibles desbordamientos en arrays y enteros y maneja la memoria de forma automática, usando Automatic Reference Counting (ARC) [2].

Otro de los cambios es que Swift permite al desarrollador especificar explícitamente el tipo o dejar que el compilador lo infiera. No obstante, es fuertemente tipado, es decir, una variable no puede cambiar su tipo, así, si esta fue creada de un tipo específico, no permitirá al desarrollador cambiarle el tipo. Lo que sí permite es realizar una conversión explícitamente al resultado de una operación a otro tipo antes de que el valor sea asignado. Es

ISSN: 2248–762X • Vol. 6, No. 2 (jul-dic 2015). pp. 6-16

decir, si se hace una operación entre dos números reales, se puede convertir su resultado a un número entero para así poder almacenarlo en una variable de tipo entero.

Nuevos tipos de variable

Swift incorpora tres nuevos tipos de variable: tuplas, "Optionals" y "Lazy properties". Con la incorporación de estas, consigue otorgar de una mayor flexibilidad y facilidad al desarrollador.

Mediante el uso de tuplas se consigue agrupar múltiples valores en un único componente. Por esto, uno de sus usos es en el retorno de una función para permitir la devolución de múltiples parámetros. La figura 3 contiene un ejemplo de tuplas.

```
let http404Error = (404, "Not Found")
var num = http404Error.0
var message = http404Error.1
```

Figura 3. Tuplas.

Los "Optional" son una clase de variable nueva en Swift que no poseen ni C ni Objective-C. Se usan para asignar un tipo cuando un valor puede ser de diferente tipo o nulo [2]. De esta manera, si una conversión no puede ser realizada, la variable tomará el valor nulo: "nil". Para declarar una variable de tipo opcional hay que poner el signo de cierre de interrogación ("?") detrás del tipo. En la figura 4 se muestra un ejemplo en el que se asigna una cadena de texto que contiene un número a una constante entera. Si la conversión de tipo fallase, la variable tomaría el valor "nil".

```
let number: String = "0"
//let nilNumber: Int = number.toInt() // Error
let nilNumber: Int? = number.toInt()
```

Figura 4. Optionals.

Swift incorpora lo que se conoce como "Lazy Property" [2]. Esto es una propiedad de una clase o estructura en la que no se calcula su valor inicial en tanto es usada por primera vez. Hasta entonces, carece de valor. Esta es una novedad en la plataforma de Apple, pues Objective-C carecía de ella. Su uso es similar al existente en otros lenguajes como son C# y Python. Para crear una propiedad "lazy" es necesario poner delante de la declaración: "lazy". Se muestra un ejemplo en la figura 5.

```
class Figure{
    var height = 240;
    var width = 480;
    lazy var are = String("Hello Cruel World")
}
```

Figura 5. Lazy Property.

Clases

Swift añadió un tipo de constructor nuevo, el "convenience initializer". Este constructor de conveniencia es opcional y es llamado siempre antes que el constructor principal. Con esto, pretenden hacer unos constructores de clase más sencillos para el desarrollador, ya que este podrá crear en los "convenience initializers" las variables que tengan una inicialización fija y, en los constructores normales, las variables que tengan una inicialización dependiente de los parámetros.

Para crear un "convenience initializers" hay que crear un constructor como el ya existe, pero con la palabra reservada "convenience" delante (figura 6). De esta manera, se consigue tener un código más limpio y legible al tener un solo constructor común y diferentes constructores de conveniencia según determinadas situaciones.

El otro cambio viene por parte de los destructores. Debido a que Swift incorpora el gestor de memoria de Objective-C, ARC, no hace falta crear destructores para liberar memoria como ocurría en Objective-C y ocurre en C y C++. No obstante, Swift incorpora los "Deinitialization" [2].

Un "Deinitialization" es un método que se llama inmediatamente cuando la instancia es liberada.

Nunca puede ser llamado explícitamente por el desarrollador. Con esta opción se provee al desarrollador de un mecanismo extra para realizar una limpieza adicional de los recursos o realizar acciones bajo ciertos casos. Un ejemplo es cuando la clase trabaja con ficheros. Cuando esta se libera, el fichero ha de ser cerrado, cosa que ARC no puede inferir [2]. Una solución es que el desarrollador cree un "Deinitialization" que cierre el fichero. Para crear un "Deinitialization" solo hace falta crear un método con la palabra reservada "deinit" (figura 7).

```
class Food {
    var name: String
    var weight: Int

    init(name: String, weight: Int) {
        self.name = name
        self.weight = weight
    }

    convenience init() {
        self.init(name: "[Unnamed]", weight: 500)
    }

    convenience init(name:String) {
        self.init(name: "[Unnamed2]", weight: 5)
    }

    convenience init(weight:Int) {
        self.init(name: "[Unnamed3]", weight: weight)
    }
}

var food = Food()
var food2 = Food(name: "Chips")
var food3 = Food(weight: 50)
var food4 = Food(name: "Meat", weight: 5)
```

Figura 6. Ejemplo de diversos Convenience Initializers.

```
deinit {
    println("\(name) has disappeared")
}
```

Figura 7. Ejemplo de un destructor.

PARADIGMA FUNCIONAL

Una de las grandes novedades en Swift fue la incorporación de un tercer paradigma de programación, el paradigma funcional. Para ello, Apple modificó el funcionamiento de las funciones y métodos a fin de incorporales retorno múltiple y soporten otras características de la programación funcional, como las "Closure". Acerca de esto se hablará en esta sección.

Funciones

Swift permite el paso de funciones, lo que se conoce como funciones lambda o funciones anónimas en el paradigma de programación funcional. Por ello, es posible declarar funciones que aceptan funciones (figura 8), funciones que devuelven funciones (figura 8), retorno de múltiples parámetros utilizando una tupla (figura 8) y el anidamiento de funciones, pretendiendo evitar la creación de funciones largas y complejas.

```
func addWithFunction(array:[Int], f:([Int]) -> Int)
    -> (a:Int, b:Int){
        return (f(array), array.count)
}
```

Figura 8. Ejemplo de una función que recibe una función y tiene retorno de múltiples parámetros, en el que uno de ellos es una función.

Closures

Swift incorpora las "closures" como parte del paradigma funcional. Las "closures" permiten evaluar una función en un entorno conteniendo una o más variable dependiendo de otro. Un ejemplo sería pasar una función a la función "map", que se puede ejecutar sobre un array, para que así la función que se pasa sea ejecutada en cada elemento del "array" por la función "map".

Swift permite crear las "Closures" de otras formas más simplificadas: retorno implícito, que implica no especificar el retorno y que automáticamente devuelva el resultado de la operación; la posibilidad de utilizar los "Shorthand Argument Names", argumentos por defecto que crea Swift automáticamente para trabajar con los parámetros que reciba la clausura sin necesidad de declararlos; incluir solo el operador deseado (<, >, +, -,…), en el caso

de que la función reciba solo dos parámetros y devuelva el resultado.

INNOVACIÓN

En esta sección se explicarán las novedades que trae Swift respecto a lo que proporcionaba Objective-C. La más notable es el Playground, un sandbox para programar. Se comentarán también los nuevos operadores. Por último, se explicará la generecidad introducida en el nuevo lenguaje de Apple y los cambios en el Switch.

Playground

Swift incorpora una nueva tecnología en el IDE Xcode 6, el Playground [1], [2]. El Playground es un compilador en tiempo real que ejecuta el código escrito en él como si de un lenguaje de script se tratase. Ofrece el valor de la ejecución en diferentes partes del programa (asignaciones, operaciones, retornos), pues se autoejecuta cuando se realizan cambios sin necesidad de tener que lanzar la aplicación. En resumen, el Playground es una consola de ejecución del lenguaje, como las existentes en Bash, PHP, Python y Ruby, entre otros, pero integrada en un IDE con editor de texto y utilidades incorporadas. En la siguiente figura 9 se puede observar esto.

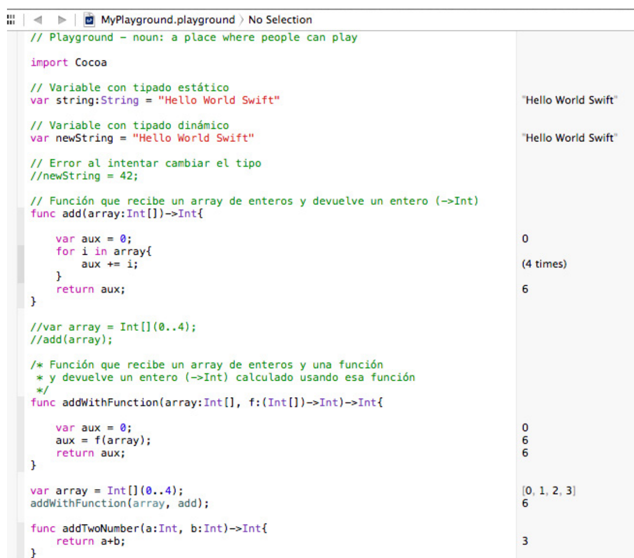

Figura 9. Ejemplo del Playground.

Lo anterior permite que se puedan probar algoritmos de una manera rápida, fácil, eficaz y aislada, pues otorga la posibilidad de encapsular el algoritmo en un sandbox limpio y sin relación con la aplicación a desarrollar. Lo cual implica que se puedan evitar errores colaterales [2]. Además, cuenta con otras utilidades gráficas para la visualización de los diferentes resultados durante la ejecución de un bucle (figura 10).

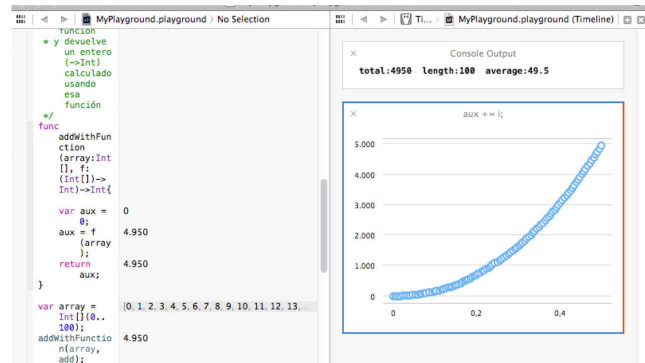

Figura 10. Ejemplo de las utilidades del Playground.

Nuevos funcionalidades de los operadores

Swift incorpora todos los operadores unarios (++, --, !), binarios (+, -, *, /) , ternarios (a:b?c), lógicos (!, &&, ||, true, false) y de asignación que posee C. No obstante, añadió tres nuevas funcionalidades a los operadores. Estas son los "Range Operators", "Overflow Operators" y los "Custom Operators".

Hay dos tipos de "Range Operator": el "Closed Range Operator" y el "Half-Open Range Operator". No obstante, su funcionamiento es muy similar y difieren mínimamente, pero nos permiten definir rápidamente la creación de una lista o "array" o crear iteración sobre ellos.

El Closed Range Operator se define con tres puntos (a…b). Esto define un rango que va desde "a" hasta "b" y que incluye ambos. Se usa para iterar sobre un rango de valores en el que ambos han de ser incluidos. Un ejemplo de definición de un "array" que vaya de 0 a 13 es: [0…13].

Por el otro lado, el Half-Open Range Operator (a..<b) define un rango que va desde "a" hasta "b" y que no incluye b. Un ejemplo de definición de un "array" que vaya de 0 a 12 es: [0..<13].

La segunda funcionalidad que añadió Swift es el "overflow" de los operadores aritméticos (+, -, *, /), pues en Swift, por defecto, estos no tienen "overflow [2]. Así, si se desea activar, hay que añadir el signo "&" antes del operador deseado.

Por ejemplo, el operador "&+" tendría "overflow" mientras que el operador "+" no. Así, en la primera variable del ejemplo obtendríamos 127, la segunda haría romper el programa y la tercera devolvería -128.

En el caso de que se desee "underflow", se debe utilizar "&-". En el ejemplo podemos ver que en la primera variable obtendríamos -128, la segunda haría romper el programa y la tercera devolvería -126.

Swift también permite controlar las divisiones por cero. Para ello, en caso de que se quiera obtener un cero en una división, se debe incluir el signo "&" antes del signo de división: "&/" y "&%". De esta manera, la primera variable del ejemplo devolvería un error y la segunda devolvería 0.

En l multiplicación es "&*", ocurre el mismo caso que en los anteriores.

La tercera funcionalidad incorporada en los operadores de Swift permite definir nuevos operadores usando los signos aritméticos existentes en el lenguaje [2]. En Objective-C no era posible, mientras que en C++ sí. Para crear un operador es necesario declarar su cabecera e implementarlo. Existen tres formas de implementar los operadores: prefijo, infijo o postfijo (figura 11).

```
operator prefix +/++-* {}

@prefix @assignment func +/++-* (inout newValue: Int) -> Int {
    newValue += 4
    return newValue
}

var addFour = 5
var newOperator = +/++-*addFour
```

Figura 11. Custom operator.

Generics

Swift incorpora los "Generics", conocidos en otros lenguajes de programación, C++ y Java, como "Templates". En Objective-C, para poder tener esta funcionalidad, era necesario implementar esa parte del código programándolo en C++. Ello permite hacer, nativamente en Swift, todo lo que conlleva e otros lenguajes el uso de los "Templates" utilizando para ello el tipo genérico "<T>".

Switch

El "switch" en Swift tiene mejoras respecto al existente en Objective-C [2]. Omite el uso de la sentencia "break", siendo ahora opcional usarla. Ahora, automáticamente rompe la secuencia de ejecución cuando se termina el "case" para conseguir que sea más fácil usarlo y poder evitar posibles errores que el desarrollador pueda cometer al olvidarse incorporar dicha palabra reservada, figura 12. No obstante, si se desea crear la programación contraria, que salte de un "case" a otro, hay que utilizar la nueva palabra reservada "fallthrough". Dicha funcionalidad es idéntica al funcionamiento de los "switch" en C#.

Otra mejora es que permite introducir varias comprobaciones en el mismo case separadas por un coma (","), usar "Range Operators" y tuplas.

La combinación del uso de "Range Operators" y tuplas en el Switch permite al desarrollador evaluar funciones matemáticas de una forma sencilla y rápida.

Esta nueva versión del "switch" incorpora la cláusula "where", que es opcional, y da la opción de añadir un chequeo adicional al caso que especifiquemos.

```
var letter:Character = "a"

switch letter{
    case "a", "A", "b":
        println("Hello")
    case "d":
        println("Bye")
    default:
        println("GoodBye")
}
```

Figura 12. Switch en Swift.

EVALUACIÓN Y DISCUSIÓN

Para corroborar lo anteriormente explicado, decidimos realizar una misma implementación en los ambos lenguajes: Swift 1.0 GM y Objective-C. Se implementó un analizador sintáctico de XML utilizando la librería estándar de XMLParser de Foundation.

En esta prueba se creó el método que inicializa el analizador sintáctico y se sobrescribieron los métodos "didStartElement" y "foundCharacters". El primero es llamado cuando se analiza un nuevo nodo. El segundo, "foundCharacters", es llamado cuando se encuentra texto dentro de un nodo.

A continuación se muestran las implementaciones en Objective-C y en Swift de los tres métodos, así como el análisis sintáctico y el número de caracteres, incluyendo espacios que necesita cada implementación. En ellas se llamó exactamente igual a las variables y se utilizó la forma nativa de cada lenguaje para acceder a los atributos de los métodos de la librería. El formateo del código es el realizado por el Xcode por defecto.

En la figura 13 se puede ver la implementación en Objective-C del método de inicialización y en la figura 14 la implementación en Swift. Como se observa, el método en Objective-C recibe un puntero. Además, cuando inicializa el analizador sintáctico de XML hace falta reservar memoria (alloc). También se puede ver un cambio de nombre de los métodos de la librería y el cambio de sintaxis en los atributos booleanos.

```
-(void)beginParsing:(NSURL *)xmlURL
{
    parser = [[NSXMLParser alloc] initWithContentsOfURL:xmlURL];
    [parser setDelegate:self];

    [parser setShouldProcessNamespaces:NO];
    [parser setShouldReportNamespacePrefixes:NO];
    [parser setShouldResolveExternalEntities:NO];

    [parser parse]
}
```

Figura 13. beginParsing en Objective-C.

```
func beginParsing(xmlUrl :NSURL)
{
    parser = NSXMLParser(contentsOfURL: xmlUrl)
    parser.delegate = self

    parser.shouldProcessNamespaces = false
    parser.shouldReportNamespacePrefixes = false
    parser.shouldResolveExternalEntities = false

    parser.parse()
}
```

Figura 14. beginParsing en Swift.

A continuación se muestra la implementación del método "didStartElement" en Objective-C (figura 15) y en Swift (figura 16). Como ocurría en el ejemplo anterior, en Objective-C hace falta recibir punteros, lo que implica que sea necesario copiar el objeto utilizando el método "copy" para así no modificar el objeto original. Después, como ocurría antes, hace falta reservar memoria para diferentes objetos de forma explícita, así como, en el comparador, asignar una "@" previamente a la cadena de texto.

```
- (void)parser:(NSXMLParser *)parser didStartElement:(NSString *)elementName namespaceURI:(NSString *)
    namespaceURI qualifiedName:(NSString *)qualifiedName attributes:(NSDictionary *)attributeDict
{
    element = [[elementName copy] stringByTrimmingCharactersInSet:[NSCharacterSet
        whitespaceAndNewlineCharacterSet]];

    if ([element isEqualToString: @"views"]) {
        elements = [[NSMutableDictionary alloc] init];
        views = [[Views alloc] init];
    }
}
```

Figura 15. Función didStartElement en Objective-C.

```
func parser(parser: NSXMLParser!,didStartElement elementName: String!, namespaceURI: String!,
    qualifiedName : String!, attributes attributeDict: NSDictionary!)
{
    element = elementName.stringByTrimmingCharactersInSet(NSCharacterSet.
        whitespaceAndNewlineCharacterSet())

    if (element as NSString).isEqualToString("views") {
        elements = NSMutableDictionary.alloc()
        elements = [:]
        views = Views()
    }
}
```

Figura 16. Función didStartElement en Swift.

Por último, se puede ver la implementación del método "foundCharacters". En la figura 17 se muestra en Objective-C y en la figura 18 en Swift. En esta comparativa vemos cómo se repiten cosas de los dos anteriores ejemplos.

```
- (void)parser:(NSXMLParser *)parser foundCharacters:(NSString *)string
{
    NSString* cad = [string stringByTrimmingCharactersInSet:[NSCharacterSet
        whitespaceAndNewlineCharacterSet]];

    if(![cad isEqual: @""] && [element isEqualToString: @"button"]) {
        [button setText: cad];
    }
}
```

Figura 17. Implementación en Objective-C de foundCharacters.

```
func parser(parser: NSXMLParser!, foundCharacters string: String!)
{
    var cad: String = string.stringByTrimmingCharactersInSet(NSCharacterSet.
        whitespaceAndNewlineCharacterSet())

    if cad != "" && element.isEqualToString("button") {
        button!.setText(cad)
    }
}
```

Figura 18. Implementación en Swift de foundCharacters.

En la tabla 1 se muestra los caracteres que se necesitó escribir en cada lenguaje para implementar los diferentes métodos, la diferencia entre Objective-C y Swift y el porcentaje de menos código necesaria al programar en Swift.

Tabla 1. Tabla comparativa entre Objective-C y Swift.

	Objective-C	Swift	Diferencia	%
beginParsing	256	226	-30	11,27%
didStartElement	401	360	-46	11.38%
foundCharacters	250	232	-18	10.77%

Como se observa en la tabla superior, en todos los casos Objective-C necesita de mucho más código para realizar la misma tarea, exactamente, entre un 10.77% y un 11.27%.

CONCLUSIONES

Como se observó, Objective-C utiliza los "[" y "]" mientras que Swift el operador "." para acceder a los métodos. Esto consigue una disminución en el uso de caracteres y, utilizando menos, dar la misma información al usuario, tal vez incluso de manera más visible. Un caso de facilidad para cometer menos errores es el no tener que reservar memoria de forma explícita, como ocurre en Objective-C. Además, renombraron métodos para acortar su nombre, pero sin perder expresividad y eliminaron la "@" que debe ir delante de las cadenas de texto. Todo esto hace que en Objective-C se requiera escribir más de un 10% de código que en Swift, lo que puede conseguir que un desarrollador cometa menos fallos.

Sobre la base de esto, se puede decir que Apple consiguió crear un lenguaje moderno con todo lo que poseen y tienen desde hace varios años otros lenguajes de programación como son C#, Java, JavaScript, Python, Ruby y Scala, entre otros. Además incorporaron la programación funcional, algo que están adoptando los lenguajes de programación para ofrecer más posibilidades a los programadores. También incorporaron nuevas posibilidades de cara a la facilidad y eficacia de programar, como los cambios en la sintaxis, las mejoras en diferentes estructuras de datos, el nuevo "Switch", las clases, variables, funciones, operadores y la eliminación de los punteros.

En resumen, Swift es un lenguaje de programación con la abstracción y funcionalidades requeridas por la época actual, y, en ciertos casos, mejorando lo existente, como es el caso del "Switch". Además, crearon un lenguaje nuevo para su ecosistema, pues Swift no es un competidor de Objective-C,

ni su evolución, sino un lenguaje de programación dispuesto a convivir con él y dar a los desarrolladores otra opción.

FINANCIAMIENTO

Este trabajo fue realizado por el grupo de investigación "Ingeniería Dirigida por Modelos MDE-RG", de la Universidad de Oviedo, en el contrato No. FUO-EM-086-14 del Proyecto de investigación "Proyecto Visio". El proyecto fue financiado parcialmente por Zed Worldwide S.A.

REFERENCIAS

[1] Apple Inc., "Swift", 2015 [en línea] Consultado el 30 de noviembre de 2015, disponible en: https://developer.apple.com/swift/

[2] Apple Inc., *The Swift Programming Language*. 2014.

[3]@adamjleonard, @thinkclay, and @cesar_devers, "Swift Toolbox", 2014 [en línea]. Consultado el 30 de noviembre de 2015, disponible en: http://www.swifttoolbox.io/

[4] "LLVM," 2000. [en línea]. Consultado el 30 de noviembre de 2015, disponible en: http://llvm.org/

[5] C. A. Lattner, "LLVM : An Infrastructure for Multi-Stage Optimization," University of Illinois, 2002.

[6] TIOBE Software BV, "TIOBE Index", 2014 [en línea]. Consultado el 30 de noviembre de 2015, disponible en: http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html

[7] C. Zapponi, "GitHub", 2014 [en línea]. Consultado el 30 de noviembre de 2015, disponible en: http://githut.info/