Rafał Kocisz

Swift: rewolucja czy ewolucja?

Na tegorocznym WWDC Apple zafundowało wszystkim swoim developerom sporą niespodziankę w postaci... nowego języka programowania! Swift, bo o tym właśnie języku tu mowa, to dla wielu programistów duży znak zapytania. W niniejszym artykule postaram się przybliżyć czytelnikowi ten temat oraz udzielić odpowiedzi na pytanie postawione w tytule: czy Swift należy postrzegać w kategoriach rewolucji, czy po prostu mamy do czynienia z nieuchronną ewolucją...

SZCZYPTA HISTORII

O tym, jak dużą niespodzianką jest Swift, szczególnie dla doświadczonych programistów wytwarzających oprogramowanie dedykowane systemom pod znaku jabłuszka, świadczyć może to, że przez ostatnie dwadzieścia lat korzystali oni tylko i wyłącznie z języka Objective-C. Język ten, zaprojektowany w 1983 roku przez Brad'a Cox'a oraz Tom'a Love'a, został wykorzystany do zaprogramowania systemu operacyjnego NeXTstep, którego następcami są OS X oraz iOS.

Póki co nic nie wskazywało na jakiekolwiek zmiany w tej materii. Co więcej, język ten przeszedł stosunkowo niedawno dość poważny lifting (Objective-C 2.0), co mogło jedynie utwierdzić jego użytkowników w przekonaniu o jego monopolistycznej pozycji w swoim segmencie rynku. A tutaj na początku czerwca 2014 roku - taka niespodzianka.

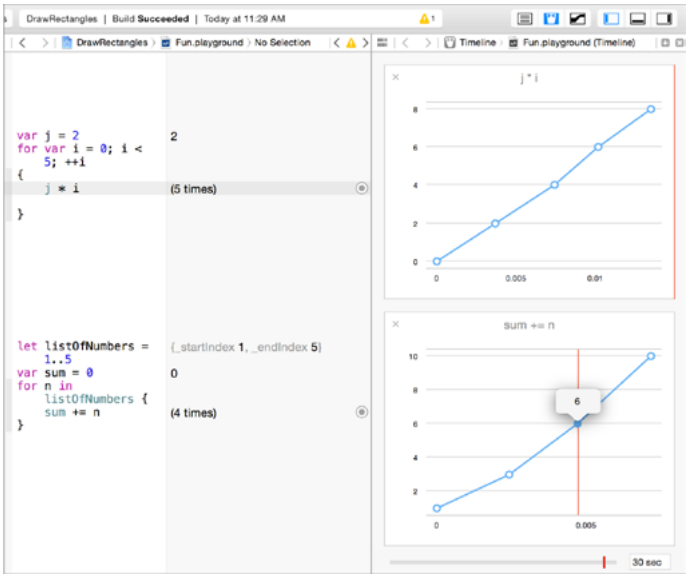
SPOJRZENIE Z LOTU PTAKA

Czym więc jest Swift? Apple w następujący sposób podsumowuje ten język:

Swift to innowacyjny, nowy język programowania dla Cocoa oraz Cocoa Touch. Cechuje się dużym poziomem interakcji, a pisanie w nim programów jest przyjemnością. Składnia języka jest bardzo zwięzła, a jednocześnie pełna wyrazu. Aplikacje pisane w tym języku działają szybko jak błyskawica. Swift jest z miejsca gotowy do użycia w Twoim kolejnym projekcie pod iOS lub OS X, bądź przy rozszerzaniu istniejącej aplikacji, bo kod napisany w tym języku działa ramię w ramię z Objective-C.

W tym krótkim opisie podkreślone są cechy tego języka, przyjrzyjmy im się bliżej:

- *Innowacyjność*. Pod tym hasłem kryje się zestaw nowoczesnych mechanizmów języka takich jak: domknięcia leksykalne (ang. *closures*), krotki (ang. *tuples*), typy generyczne (ang. *generics*), klasy oraz elementy zapożyczone z języków funkcjonalnych, np. *map* czy *filter*. W tym ujęciu Swift wpisuje się w kanon współczesnych, interpretowanych języków programowania takiej klasy jak Python, Ruby, Groovy.
- *Interakcja*. Pod tym hasłem kryje się interaktywna powłoka języka (tzw. *REPL*), która umożliwia programiście interakcję z kodem w czasie działania aplikacji. REPL nie jest w zasadzie niczym nowym, programiści korzystający z innych języków interpretowanych używają tego mechanizmu od lat, jednakże w uniwersum Apple jest zdecydowanie powiew świeżości. Poza tym programiści iOS oraz OS X w ramach nowej wersji sztandarowego IDE marki Apple (Xcode) mają do dyspozycji interaktywne narzędzie Playgrounds (Rysunek 1), które pozwala między innymi wizualizować działanie budowanych algorytmów, tworzyć w locie testy jednostkowe oraz łatwo eksperymentować z nowymi API.
- *Wydajność*. Apple na każdym kroku podkreśla, że Swift pod kątem wydajności nie ustępuje językowi Objective-C, przede wszystkim dzięki temu, iż jest on w locie przekształcany do zoptymalizowanego kodu natywnego. Faktem pozostaje, że komentarze pojawiające się ze strony programistów eksperymentujących z językiem, w kontekście wydajności aplikacji pisanych w Swift, nie są aż tak optymistyczne jak chciałoby Apple...
- *Dostępność*. Swift'a może zacząć z miejsca używać każdy, kto ma konto developerskie Apple. W takim wypadku wystarczy pobrać i zainstalować sobie paczkę ze środowiskiem Xcode 6 Beta. Oprócz tego Apple oferuje całkiem pokaźny zestaw materiałów do nauki tego języka.



Rysunek 1. Narzędzie Playgrounds w akcji

PRZEGLĄD MOŻLIWOŚCI JĘZYKA

Rzućmy okiem na Swift od strony praktycznej. Na początek oczywiście szybkie spojrzenie na program typu *Hello, World!* (patrz: Listing 1).

Listing 1. Program typu Hello, World! w Swift

```
println("Hello, World!")
```

Pierwsze wnioski, które nasuwają się po analizie tego krótkiego programu, są następujące:

- aplikację typu *Hello, World!* w języku Swift da się napisać w jednej linii!
- Swift nie wymaga stosowania średników jako separatorów instrukcji,
- w Swift korzystamy z nawiasów okrągłych przy wywoływaniu funkcji, niejako w opozycji do nawiasów kwadratowych używanych przy wywoływaniu metod w Objective-C.

Programiści korzystający ze Swift będą się musieli pożegnać z rozróżnieniem pomiędzy plikami nagłówkowymi (*.h*) a źródłowymi (*.m*). Programy pisane w nowym języku Apple umieszczane są w pojedynczych plikach tekstowych opatrzonych rozszerzeniem *.swift*.

Nie będzie też znaków plus (+) oraz minus (-), służących do rozróżniania zwykłych metod od metod statycznych w Objective-C. Składnia definicji klasy została w Swift bardzo uproszczona. Na Listingu 2 przedstawione są dwie proste definicje klas: Shape oraz Square prezentujące takie mechanizmy języka jak konstruktory, dziedziczenie czy przeciążanie funkcji wirtualnych.

Listing 2. Proste definicje klas w Swift

```
class Shape
{
  var numberOfSides: Int = 0
  var name: String

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

  func toString() -> String
  {
     return "A shape with \(numberOfSides) sides."
  }
}

class Square: Shape
{
  var sideLength: Double

  init(sideLength: Double, name: String)
  {
    self.sideLength = sideLength
    super.init(name: name)
    numberOfSides = 4
  }

  func area() ->  Double
  {
    return sideLength * sideLength
  }

  override func toString() -> String
  {
    return "A square with sides of length \(sideLength)."
  }
}

let test = Square(sideLength: 5.2, name: "my test square")

test.area()
test.toString()
```

Bardzo miłym akcentem jest wsparcie dla mechanizmu właściwości (ang. *properties*) powiązanych z akcesorami (`get/set`), znanego z takich języków programowania jak C# czy AcionScript 3.0. Na Listingu 2a pokazana jest prosta definicja klasy korzystająca z tego udogodnienia.

Listing 2. Przykład użycia mechanizmu właściwości w Swift

```
class EquilateralTriangle: Shape
{
  var sideLength: Double = 0.0

  init(sideLength: Double, name: String)
  {
    self.sideLength = sideLength
    super.init(name: name)
    numberOfSides = 3
  }

  var perimeter: Double
  {
    get
    {
      return 3.0 * sideLength
    }
    set
    {
      sideLength = newValue / 3.0
    }
  }

  override func toString() -> String
  {
    return "An equilateral triagle with "
      "sides of length \(sideLength)."
  }
}
```

Dużą zmianą w stosunku do Objective-C jest niewątpliwie wprowadzenie mechanizmu typów uogólnionych (ang. *generics*). Dla przykładu, klasa `NSArray` z Objective-C może w zasadzie przechowywać obiekty dowolnego typu (przy czym wszystkie one muszą dziedziczyć po protokole `NSObject`). Rozwiązanie to, pomimo stwarzania pozoru dużej elastyczności, w praktyce jest bardzo mało odporne na błędy, przede wszystkim ze względu na fakt, iż kompilator nie jest w stanie wykryć pewnych niezgodności, które objawiają się dopiero w czasie wykonania programu. Typy uogólnione w Swift, czyli mechanizm zbliżony do szablonów języka C++, pozwalają tworzyć definicje klas parametryzowane typami. W tym układzie niezgodność typów jest wykrywana już na etapie kompilacji.

Listing 3 pokazuje przykład użycia typów uogólnionych w Swift.

Listing 3. Typy uogólnione w Swift

```
struct IntPair
{
  let a: Int!
  let b: Int!

  init(a: Int, b: Int)
  {
    self.a = a
    self.b = b
  }

  func equal() -> Bool {
    return a == b
  }
}

let intPair = IntPair(a: 5, b: 10)

intPair.a // 5
intPair.b // 10
intPair.equal() // false

struct Pair<T: Equatable>
{
  let a: T!
  let b: T!

  init(a: T, b: T)
  {
    self.a = a
    self.b = b
  }

  func equal() -> Bool
  {
    return a == b
  }
}

let pair = Pair(a: 5, b: 10)

pair.a // 5
pair.b // 10
pair.equal() // false

let floatPair = Pair(a: 3.14159, b: 2.0)

floatPair.a // 3.14159
floatPair.b // 2.0
floatPair.equal() // false
```

Skoro już mówimy o typach, trzeba jasno powiedzieć, że Swift jest językiem silnie typowanym, co można uznać za duży krok do przodu w dziedzinie odporności na błędy. Ceną za silne typowanie jest zazwyczaj bardziej skomplikowana składnia (patrz: język C++). Projektanci Swift podjęli próbę zmierzenia się z tym problemem, wprowadzając do swojego języka mechanizm inferencji typów, czyli technikę stosowaną w językach statycznie typizowanych, która zwalnia programistę z obowiązku specyfikowania typów, przerzucając obowiązek ich identyfikacji na kompilator. Przykład zastosowania tego mechanizmu w Swift pokazany jest na Listingach 4a oraz 4b.

Listing 4a. Tworzenie instancji obiektu w Objective-C

```
Person *me = [[Person alloc] initWithName:@"Rafal Kocisz"];
[me sayHello];
Listing 4b. Inferencja typów w Swift
var me = Person(name:"Rafal Kocisz")
me.sayHello()
```

Listingi te pokazują dwa przypadki tworzenia instancji obiektu klasy `Person`; jeden w Objective-C, zaś drugi w Swift. Jak widać, w drugim przypadku kompilator sam jest w stanie wydedukować typ obiektu, co w rezultacie skutkuje znacznym uproszczeniem składni.

Dużo pozytywnych zmian pojawia się w Swift w związku z obsługą napisów. W nowym języku od Apple napisy są pełnoprawnymi obiektami, można je łatwo porównywać za pomocą operatora == czy łączyć dzięki zastosowaniu operatorów + oraz +=. W Swift nie istnieje również rozróżnienie pomiędzy napisami zmiennymi (ang. *mutable*) oraz niezmiennymi (ang. *immutable*), które występowało w Objective-C. Niejeden programista ucieszy się też z faktu, że napisy dostępne w ramach języka Swift domyślnie obsługują pełny zestaw znaków Unicode! Co ciekawe, znaków tego rodzaju można używać również w nazwach identyfikatorów. Biorąc pod uwagę to, jak niewygodna jest obsługa napisów w języku Objective-C, opisane wyżej zmiany będą zapewne bardzo miłe programistom wytwarzającym aplikacje pod system iOS oraz OS X.

Warto też wspomnieć o znacznym usprawnieniu instrukcji `switch`, która w przypadku Swift'a - w odróżnieniu od Objective-C - obsługuje zarówno napisy, jak i założone obiekty. Poza tym, instrukcja `switch` w nowym języku od Apple domyślnie wykonuje skok do kolejnej instrukcji tuż po wykonaniu kodu umieszczonego w dopasowanej sekcji `case`. W tym układzie słowo kluczowe `break` (którego używanie wiąże się niejednokrotnie z powstawaniem trudnych do wyśledzenia błędów) przestaje być potrzebne. Dla tych, którzy mimo wszystko chcieliby mieć możliwość wywołania kilku sekcji `case` pod rząd, Swift oferuje słowo kluczowe `fallthrough`, za pomocą którego można tego rodzaju efekt uzyskać. Na Listingu 5 przedstawiony jest przykład użycia instrukcji `switch` w języku Swift.

Listing 5. Przykład użycia instrukcji switch w języku Swift

```
let vegetable = "red pepper"

switch vegetable
{
case "celery":
  let vegetableComment = "Add some raisins and make ants on a log."

case "cucumber", "watercress":
  let vegetableComment = "That would make a good tea sandwich."

case let x where x.hasSuffix("pepper"):
  let vegetableComment = "Is it a spicy \(x)?"

default:
  let vegetableComment = "Everything tastes good in soup."
}
```

Koniec końców, autorom Swift należy się duża pochwała za dodanie do tego języka elementów programowania funkcjonalnego. Przede wszystkim, w Swift funkcje są pełnoprawnymi obiektami, które można przekazywać jako parametry, przechowywać w kontenerach itd. Swift oferuje również domknięcia leksykalne (ang. *closures*), mechanizm podobny nieco do bloków (ang. *blocks*) z Objective-C 2.0, jednakże bardziej potężny: porównywalny z wyrażeniami lambda rodem z języka C#. Listing 6 zawiera kilka przykładów użycia opisanych wyżej mechanizmów w kodzie Swift (między innymi: funkcja, która zwraca funkcję; funkcja, która przyjmuje inną funkcję jako parametr, a także praktyczne zastosowanie domknięcia leksykalnego).

Listing 6. Elementy programowania funkcjonalnego w języku Swift

```
func makeIncrementer() -> (Int -> Int)
{
  func addOne(number: Int) -> Int
  {
    return 1 + number
  }

  return addOne
}

var increment = makeIncrementer()

increment(7)

func hasAnyMatches(
  list: Int[],
  condition: Int -> Bool) -> Bool
{
  for item in list
  {
    if condition(item)
    {
      return true
    }
  }

  return false
}

func lessThanTen(number: Int) -> Bool
{
  return number < 10
}

var numbers = [20, 19, 7, 12]

hasAnyMatches(numbers, lessThanTen)

numbers.map(
  {
    (number: Int) -> Int in
    let result = 3 * number
    return result
  })
```

Na tym będziemy kończyć nasz pobieżny przegląd możliwości języka. Celowo użyłem tutaj słowa „pobieżny", jako że moim zamiarem było przede wszystkim przybliżenie czytelnikowi istoty języka Swift; pod takim też kątem starałem się dobrać prezentowane przykłady. W dalszej części artykułu postaram się odpowiedzieć na kilka pytań, które mogły pojawić się w Twojej głowie w związku z pojawieniem się nowego języka do Apple.

QUO VADIS OBJECTIVE-C?

Póki co, drogi przyjacielu, nigdzie się nie wybieram! ;) - taką odpowiedź usłyszelibyśmy zapewne od języka Objective-C, gdyby umiał on mówić. Objective-C mówić oczywiście nie potrafi, jednakże umiejętność tę posiadły rzesze programistów, którzy na co dzień korzystają z tego języka. Wyobrażasz sobie ich reakcję w sytuacji, gdyby miał on z dnia na dzień zniknąć, wyparować? Taki eksperyment myślowy polecam wszystkim tym, którzy prowadzą zagorzałe dyskusje na temat najbliższej przyszłości Objective-C.

Trochę trudniej jest odpowiedzieć na inne pytanie: „którego języka w tym układzie warto się uczyć?". Generalnie, wydaje się, że opracowując Swift, Apple chciał z jednej strony obniżyć barierę wejścia dla tych, którzy dopiero uczą się programować, zaś z drugiej strony - zrobić ukłon w kierunku młodszej generacji programistów wychowanych na takich językach jak Python, Ruby czy C#. Dla tych programistów przesiadka na Swift będzie czymś zupełnie naturalnym.

Co z kolei ze starszą generacją programistów (do której sam się już niestety w pewnym stopniu zaliczam ;))? Dla tych ludzi, zakorzenionych w językach pokroju C oraz C++, Swift będzie prawdopodobnie nieco mniej atrakcyjny. Osobom tego pokroju zapewne łatwiej byłoby nadal używać Objective-C, który jest przecież nadzbiorem klasycznego C.

Trudno przewidywać dalsze ruchy Apple w zakresie wsparcia dla języków programowania; podejrzewam, że przed nami jest dość ciekawy okres przejściowy, czas, w którym Apple będzie starać się przesunąć środek ciężkości zainteresowania programistów w kierunku Swift'a. Jednakże jestem głęboko przekonany, że Objective-C jeszcze długo pozostanie z nami.

REWOLUCJA CZY EWOLUCJA?

Biorąc pod uwagę to wszystko, co opisałem powyżej, moja odpowiedź na to pytanie może być tylko jedna: zdecydowanie ewolucja!

No bo zastanówmy się: w dziedzinie języków programowania Swift przecież żadną rewolucję nie jest. Jest to oczywiście wysokiej klasy nowoczesny język programowania, ale nie wprowadza nic takiego, co można by uznać za

jakąś rewelację. Gdyby Apple zdecydował się wprowadzić np. własny dialekt Lispu jako swój nowy, oficjalny język programowania, to być może byłbym skłonny uznać taki ruch za rewolucyjny... Swoją drogą, szkoda, że się na to nie zdecydowali... Trudno mówić również o rewolucji w kontekście jakichś gwałtownych zmian (Objective-C póki co pozostaje z nami).

W mojej opinii Swift'a należy postrzegać jako nowe, fajne narzędzie, które dostaliśmy w prezencie od firmy Apple, i podchodzić to tego tak, jak należy podchodzić do narzędzia: to znaczy w sposób pragmatyczny, a nie emocjonalny.

Bardzo spodobał mi się pewien komentarz na jednym z forów, na którym rozgorzała dyskusja dotycząca przyszłości Swift'a oraz Objective-C. Autor komentarza zauważył, że w gruncie rzeczy kwestia języka, z którego w danym momencie korzystamy, jest w dużym stopniu drugorzędna. Prawdziwa trudność (a zarazem sztuka) związana z programowaniem leży w umiejętności konstruowania algorytmów, projektowania interfejsów, klas itd. Z kolei w przypadku programowania systemów iOS oraz OS X znacznie ważniejszym (i trudniejszym) zadaniem w stosunku do opanowania języka jest poznanie oraz zrozumienie olbrzymiej biblioteki klas dostępnych w ramach frameworków Cocoa oraz Cocoa Touch.

W tym ujęciu, tym, którzy pytają: *czego warto się dziś uczyć,* odpowiem tak: uczcie się programować aplikacje pod systemy iOS oraz OS X, zgłębiajcie API Cocoa oraz Cocoa Touch. A czy będziecie używać do tego celu języka Objective-C, czy może Swift'a, wydaje mi się naprawdę kwestią drugorzędną. Wybierzcie sobie po prostu ten język, który lepiej Wam pasuje! :)

Rafał Kocisz rafal.kocisz@gmail.com

Rafał od dziesięciu lat pracuje w branży związanej z produkcją oprogramowania. Jego zawodowe zainteresowania skupiają się przede wszystkim na nowoczesnych technologiach mobilnych oraz na programowaniu gier. Rafał pracuje aktualnie jako Techniczny Koordynator Projektu w firmie BLStream.