

Spis treści

1. Wstęp	3
2. Podobieństwa do innych języków programowania	5
2.1. Podstawowe cechy	5
2.2. Typy generyczne	5
2.3. Typy wartościowe i referencyjne	6
2.4. Domknięcia jako typy pierwszoklasowe	7
2.5. Leniwość	7
2.6. Elementy zaczerpnięte z języków funkcyjnych	7
2.7. Zwięzłość składni	8
2.8. Rozszerzenia typów	8

Rozdział 1.

Wstęp

...

Rozdział 2.

Podobieństwa do innych języków programowania

2.1. Podstawowe cechy

Swift to wieloparadygmataowy język programowania łączący pomysły znane z innych popularnych języków, takich jak: Objective-C, C#, Rust, Haskell czy Ruby. Podobnie jak C#, pozwala na tworzenie struktur (value type/typów wartościowych?), klas (reference type/typów referencyjnych?) i typów wyliczeniowych. Wspiera również dziedziczenie (ale nie wielokrotne), definiowanie protokołów (odpowiednik interfejsów z C# czy Java) oraz polimorfizm parametryczny (typy generyczne), nie pozwala natomiast na definiowanie klas abstrakcyjnych, zachęcając tym samym programistów do szerokiego stosowania interfejsów.

TODO: Tu przydałby się jeszcze jeden akapit o mniej znaczących feature'ach

2.2. Typy generyczne

Swift wspiera dwie podstawowe koncepcje generyczności:

- klasy, struktury, typy wyliczeniowe oraz funkcje z parametrami typu (ang. *generics*)
- protokoły z powiązаныmi typami (ang. *associated types*)

Klasy (struktury, funkcje) ze zmiennymi typu to pomysł dobrze znany z większości popularnych języków pozwalających na programowania obiektowe, takich jak C# czy Java. W momencie definiowania klasy programista ma możliwość zdefiniowania zmiennych przebiegających przestrzeń typów używanych w definiowanej klasie. Dodatkowo Swift oferuje kilka bardziej zaawansowanych mechanizmów związanych ze zmiennymi typu:

TODO: Ujednolicić nazewnictwo

- możliwość dodania ograniczeń na typy, po których przebiega zmienna, np. zmienna może być tylko typem implementującym dany protokół lub dziedziczącym po danej klasie
- automatyczna inferencja typów parametrów generycznych
- możliwość nadawania aliasów funkcjom i typom generycznym

TODO: Coś o implementacji?

W odróżnieniu od klas, struktur i funkcji, protokoły nie wspierają generycznych parametrów typu. Zamiast tego, protokoły posiadają mechanizm typów powiązanych (ang. *associated types*), wzorowany na znanym np. ze Scali mechanizmie abstrakcyjnych pól typu (ang. *abstract type members*). Pozwala on na zdefiniowanie w protokole zmiennej typu, która zostanie ukonkretniona dopiero przez klasę implementującą dany protokół. Główną zaletą tego rozwiązania jest ukrycie typu podstawionego pod zmienną typu przed programistą używającym klasy implementującej dany protokół - typ podstawiony pod zmienną jest częścią implementacji i nie musi być jawnie podawany podczas tworzenie obiektu implementującego protokół.

2.3. Typy wartościowe i referencyjne

Podobnie jak w języku C#, typy w Swifcie można podzielić na dwie grupy:

- typy wartościowe (ang. *value types*)
- typy referencyjne (ang. *reference types*)

Typy wartościowe to typy, które tworzą nowe instancje obiektów podczas przypisywania do zmiennej lub przekazywania do funkcji. Innymi słowy, każda instancja posiada swoją własną kopię danych, obiekty takie nie dzielą ze sobą stanu, przez co są łatwiejsze w zrozumieniu i bezpieczniejsze przy pracy z wieloma wątkami. Jeśli zmienna typu wartościowego zostanie zadeklarowana jako stała, cały obiekt, łącznie ze wszystkimi polami nie może zostać zmieniony. Typami wartościowymi w Swifcie są:

- struktury
- typy wyliczeniowe
- krotki

Typy referencyjne to typy, których obiekty dzielą pomiędzy sobą te same dane, a podczas przypisywania lub przekazywania do funkcji tworzona jest tylko nowa referencja do tych samych danych. Zmienne typu referencyjnego zadeklarowane jako stałe zapewniają jedynie stałość referencji, jednak dane przypisane do zmiennej mogą być bez dowolnie zmieniane. Typami referencyjnymi w Swiftcie są tylko klasy.

2.4. Domknięcia jako typy pierwszoklasowe

Podobnie jak w językach funkcyjnych i w większości nowoczesnych języków programowania obiektowego, domknięcia w Swiftcie są typem pierwszoklasowym (ang. *first-class citizen*), tzn:

- mogą być przechowywane w zmiennych i stanowić elementu struktur danych
- mogą być podawane jako parametry wywołania funkcji i metod
- mogą być zwracane przez funkcje i metody

2.5. Leniwość

Swift jest domyślnie językiem z ewaluacją gorliwą, autorzy zaimplementowali jednak dwa rozwiązania pozwalające w podstawowym stopniu na wspieranie leniwych obliczeń. Po pierwsze, w Swiftcie, podobnie jak w C#, istnieje możliwość leniwej inicjalizacji obiektów. O ile jednak w C# mechanizm ten polega na użyciu klasy *Lazy* z biblioteki standardowej, o tyle w Swiftcie jest on zaszyty w samym języku - służy do tego słowo kluczowe *lazy*. Drugim rozwiązaniem są leniwe struktury danych, których implementacja opiera się na znanych również z języka C# czy Java generatorach.

2.6. Elementy zaczerpnięte z języków funkcyjnych

Pomimo tego, że Swift był projektowany głównie jako język programowanie obiektowego, jego twórcy skupili dużą część swojej uwagi na elementach powiązanych z programowaniem funkcyjnym, które mogłyby pomóc programistom pisać bezpieczniejszy i bardziej czytelny kod obiektowy. Najważniejsze z nich to:

- Typy wyliczeniowe z typami powiązanymi (ang. *associated types*), które wprowadzają do Swifta koncept podobny do konstruktorów typów z Haskella. Dzięki tym strukturom danych, programista może w funkcyjny deklarować nowe typy.
- Dzięki zwężłej i eleganckiej składni oraz potraktowaniu domknięć na równi klasami i strukturami, Swift oferuje bardzo dobre wsparcie dla funkcji wyższego

rzędu. Najbardziej jest to widoczne w implementacji kolekcji danych, które już w bibliotece standardowej posiadają najczęściej używane funkcje służące do manipulowania nimi, takie jak `filter`, `map`, `reduce` czy `flatMap`.

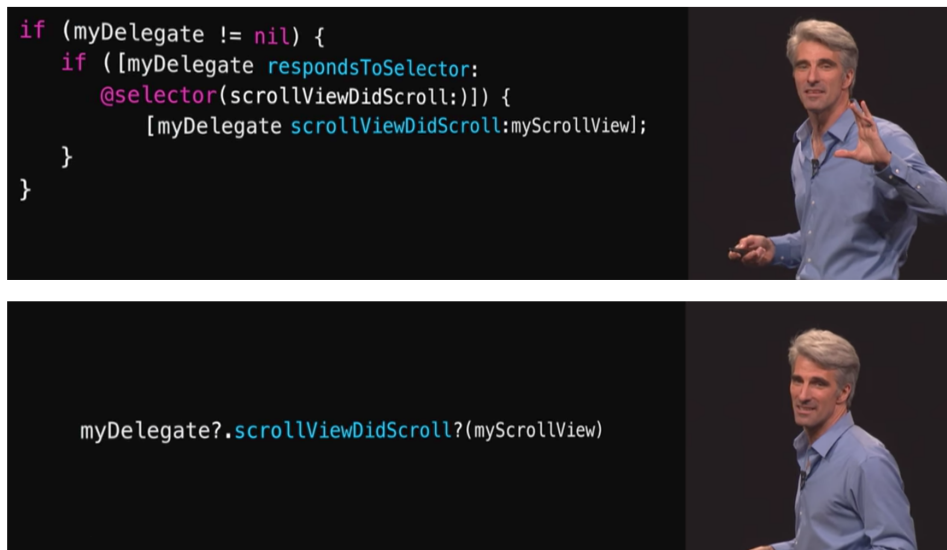
- Autorzy Swifta postawili bardzo duży nacisk na niemutowalność (*TODO: lepsze słowo?*) danych, co przejawia się w całej składni języka, np. dostępne jest słowo odrębne słowo kluczowe `let` służące do deklarowania stałych, parametry przekazywane do funkcji są domyślnie stałymi, użycie typów wartościowych jest preferowane nad użyciem klas (również w bibliotece standardowej) itp.
- Swift posiada zaawansowany mechanizm *pattern matchingu*, który można wykorzystać do dopasowywania typów wyliczeniowych, krotek i wyrażeń. Tak jak w wielu językach funkcyjnych, *pattern matching* w Swiftcie jest wyczerpujący (ang. *exhaustive*), co oznacza, że każda wartość, która może pojawić się podczas dopasowywania musi zostać obsłużona.
- Aby uniknąć problemów z wartością `nil`, w języku Swift każda zmienna musi zostać zainicjalizowana już w momencie deklaracji. Jeśli programista chce celowo stworzyć zmienną mogącą przyjmować `nil` jako wartość, powinien użyć typu `Optional<T>`, który w swojej konstrukcji jest bardzo podobny do monady `Maybe` znanej z Haskella. Istnieje nawet mechanizm zwany *optional chaining*, który zachowuje się tak, jak operacja `>>=` dla monady `Maybe`.

2.7. Zwięzłość składni

Jednym z największych problemów podczas programowania w Objective-C była słaba czytelność kodu i bardzo rozwlekła składnia. Dlatego podczas projektowania Swifta jego autorzy bardzo mocno wzorowali się na językach bardzo zwięzłych i łatwych do czytania, takich jak Python czy Ruby. Zrezygnowano z plików nagłówkowych, wprowadzono dużo cukru syntaktycznego dla najczęściej stosowanych konstrukcji (jak np. operator `T?` dla typu `Optional<T>`), wprowadzono domyślną inferencję typów. Rysunek 2.1 pokazuje różnice pomiędzy kodem napisanym w Objective-C, a takim samym kodem w Swift.

2.8. Rozszerzenia typów

Jedną z rzadziej spotykanych w językach statycznie typowanych językach programowania funkcjonalności jest możliwość rozszerzania istniejących już typów. Co prawda już w Objective-C programista miał możliwość stworzenia kategorii (ang. *category*), ale pozwalała ona tylko na dodawanie nowych funkcji, nie można było natomiast definiować nowych właściwości, konstruktorów ani typów zagnieżdżonych.



Rysunek 2..1: Craig Federighi wskazuje różnice pomiędzy czytelnością kodu Objective-C (na górze) i Swift (na dole). *WWDC Keynote 2014*

Dlatego w Swifcie zostały zaimplementowane rozszerzenia (ang. *extensions*), które pozwalają na:

- dodawanie nowych właściwości obliczanych (ang. *computed properties*)
- definiowanie nowych metod instancji i metody typu
- definiowanie nowych inicjalizatorów
- definiowanie i używanie typów zagnieżdżonych
- implementowanie metod protokołów

10ROZDZIAŁ 2. PODOBIENSTWA DO INNYCH JĘZYKÓW PROGRAMOWANIA

```
// struktura - typ wartościowy
struct UserInfo {
    let name: String
    let identifier: Int
}

// typ wyliczeniowy
enum ScreenResolution {
    case SD
    case HD
    case FullHD
}

// protokół - obiekt definiujący interfejs klasy/struktury go implementującej
protocol UserInfoDrawer {
    func drawUserInfo(info: UserInfo) -> Void
}

// klasa - typ referencyjny
class TerminalUserInfoDrawer: UserInfoDrawer {
    var screenResolution: ScreenResolution = .FullHD

    func drawUserInfo(info: UserInfo) -> Void {
        // .. implementacja wyświetlania informacji o użytkowniku
    }
}
```

Listing 1: Przykładowe definicje podstawowych obiektów w Swift: struktury, klasy, protokołu i typu wyliczeniowego

```
class Stack<T> {
    var stack: Array<T> = []
    func push(object: T) {
        stack.append(object)
    }
    func pop() -> T {
        return stack.removeLast()
    }
}

class PopManyStack<T> : Stack<T> {
    func popN(n: Int) -> [T] {
        let lastN = stack.suffix(n)
        stack.removeLast(n)
        return Array(lastN)
    }
}
```

Listing 2: Przykład klasy generycznej i klasy pochodnej w Swift

```

import UIKit

protocol ViewDecorator {
    associatedtype ViewType: UIView
    func decorate(view: ViewType)
}

class ImageViewDecorator: ViewDecorator {
    typealias ViewType = UIImageView

    func decorate(view: UIImageView) {
        view.backgroundColor = UIColor.black
        view.layer.cornerRadius = 5.0
        // ... więcej ustawień
    }
}

class LabelDecorator: ViewDecorator {
    typealias ViewType = UILabel

    internal func decorate(view: UILabel) {
        view.font = UIFont.systemFont(ofSize: 20.0)
        // ...
    }
}

let decorator1 = ImageViewDecorator() // nie trzeba podawać typu generycznego,
                                     // implementacja klasy wskazuje, czym jest ViewType
let decorator2 = LabelDecorator()
decorator1.decorate(view: UIImageView()) // system typów pilnuje,
                                     // aby view było typu UIImageView
decorator2.decorate(view: UILabel())

```

Listing 3: Przykład protokołu z typem powiązonym w Swift

```

// domknięcie przyjmujące dwa obiekty typu Int i zwracające obiekt typu Int
let addTwoInts: ((Int, Int) -> Int) = { (a, b) in a + b }

// wywołanie domknięcia przypisanego do zmiennej 'addTwoInts'
let result = addTwoInts(5, 10)

```

Listing 4: Przykład użycia domknięcia w Swift

```

typealias BigInt = Int

class DataContainer {
    // zmienna tworzona leniwie, dopiero podczas jej pierwszego użycia
    lazy var bigData = BigInt()

    // .. reszta ciała klasy
}

let container = DataContainer()
print(container.bigData) // obiekt bigData zostanie utworzony dopiero w tym momencie

// sekwencja liczb Fibbonacciego generowana leniwie
class Fibonacci: LazySequenceProtocol {
    public func makeIterator() -> FibonacciIterator {
        return FibonacciIterator()
    }
}

// generator liczb Fibbonacciego
class FibonacciIterator: IteratorProtocol {
    private var first = 0
    private var second = 1

    public func next() -> Int? {
        let next = first + second
        first = second
        second = next
        return next
    }
}

let evenFibonacci = Fibonacci().filter { $0 % 2 == 0 }
var iterator = evenFibonacci.makeIterator()

for i in 1...5 {
    print(iterator.next()!)
}

```

Listing 5: Przykład deklaracji leniwej zmiennej i leniwej sekwencji w Swift

```

indirect enum Tree<Value> {
    case Empty
    case Node(Tree<Value>, Value, Tree<Value>)
}

let intTree = Tree.Node(
    .Node(.Empty, 1, .Empty),
    2,
    .Empty
)

```

Listing 6: Implementacja drzewa binarnego w Swift

```
// Implementacja typu Optional z biblioteki standardowej
public enum Optional<Wrapped> {
    case none
    case some(Wrapped)
}

// Monada Maybe w Haskellu
// data Maybe a = Nothing | Just a

struct Address {
    let city: String? // String? to cukier syntaktyczny dla typu Optional<String>
    let postalCode: String?
}

struct User {
    let name: String
    let address: Address?
}

let sampleUser: User? = User(
    name: "Jan Kowalski",
    address: Address(city: "Wrocław", postalCode: "50-500")
)

// Przykład użycia optional chaining - stała city ma typ Optional<String>
let city = sampleUser?.address?.city
```

Listing 7: Typ Optional i mechanizm *optional chaining*