Spis treści

Rozdział 1.

Wstęp

...

Rozdział 2.

Podobieństwa do innych języków programowania

2.1. Podstawowe cechy

Swift to wieloparadygmatowy 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 (typów wartościowych), klas (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.

2.2. Typy generyczne

Swift wspiera dwie podstawowe koncepcje generyczności:

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

Klasy (struktury, funkcje) ze zmiennymi typu to pomysł dobrze znany z większości popularnych języków programowania obiektowego, 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:

- 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 paremetrów generycznych
- możliwość nadawania aliasów funkcjom i typom generycznym

Przykład użycia typów generycznych ilustruje Listing 1.

```
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 1: Przykład klasy generycznej i klasy pochodnej w Swift

W odróżnieniu od klas, struktur i funkcji, protokoły nie wspierają generycznych paremetrów 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. Przykład użycia protokołu z parametrami typu prezentuje Listing 2.

```
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 2: Przykład protokołu z typem powiązanym w Swift

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 Swifcie są tylko klasy.

```
// 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 3: Przykładowe definicje podstwowych obiektów w Swift: struktury, klasy, protokołu i typu wyliczeniowego

2.4. Domknięcia jako typy pierwszoklasowe

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

- mogą być przechowywane w zmiennych i stanowić elementy struktur danych
- mogą być podawane jako parametry wywołania funkcji i metod

- mogą byc zwracane przez funkcje i metody

```
// 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

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 Swifcie, 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 Swifcie jest on zaszyty w samym języku - służy do tego słowo kluczowe *lazy*. Drugim rozwiązaniem są leniwe struktury danych, których implementacja opera się na znanych również z języka C# czy Java generatorach.

```
typealias BigData = Int

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

   // .. reszta ciała klasy
}

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

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

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

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

```
        second = next
        return next
    }
}

let evenFibbonacci = Fibbonacci().filter { $0 % 2 == 0 }
var iterator = evenFibbonacci.makeIterator()

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

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

2.6. Elementy zaczerpnięte z języków funkcyjnych

Pomimo tego, że Swift był projektowany głównie jako język programowania 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 wartościami powiązanymi (ang. *associated values*), które wprowadzają do Swifta koncept podobny do konstruktorów typów z Haskella. Dzięki tym strukturom danych, programista może w funkcyjny deklarować nowe typy (zob. Listing 6).

```
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

- Dzięki zwięzłej i eleganckiej składni oraz potraktowaniu domknięć na równi klasami i stukturami, Swift oferuje bardzo dobre wsparcie dla funkcji wyższego rzędu. Funkcje wyższego rzędu są też często używane w bibliotece standardowej, np. kolekcje danych 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 Swifcie jest wyczerpujący (ang. *exhaustive*), co oznacza, że każda wartość, która może pojawić się podczas dopasowywania musi zostać obsłużona.

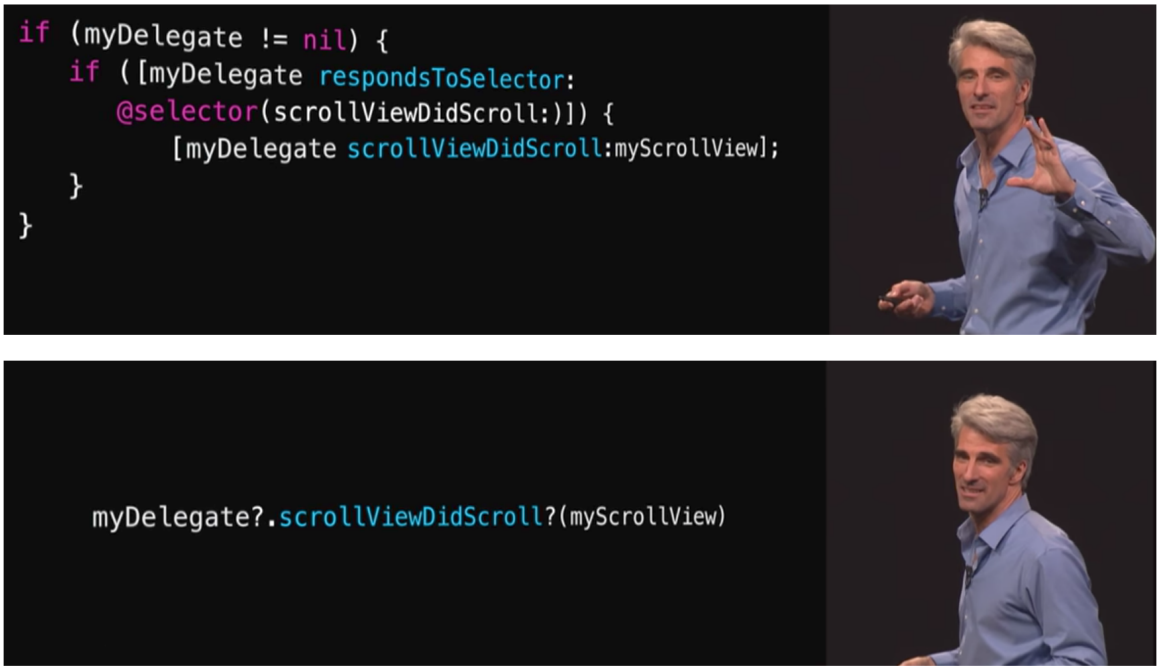
- Aby uniknąć problemów z wartością `nil`, w języku Swift każda zmienna musi zostać zainicializowana już w momencie deklaracji. Jeśli programista chce celowo stworzyć zmienną mogącą przyjmować wartość `nil`, powinien użyć typu `Optional<T>`, który w swojej konstrukcji jest bardzo podobny do monady `Maybe` znanej z Haskella. Istnieje nawet mechanizm zwany *optional chaning*, który zachowuje się tak, jak operacja `>>=` dla monady `Maybe`.

```
// 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*

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 iżynierowie Apple mocno wzorowali się na językach znanych ze swojej zwięzłości i łatwości 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.



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

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 moliwość stworzenia kategorii (ang. *category*), ale pozwalała ona tylko na dodawanie nowych funkcji, nie można było natomiast definiować nowych właściwości, konstruktorów ani typów zagnieżdżonych. Dlatego w Swifcie zotały zaimplementowane rozszerzenia (ang. *extensions*), które pozwalają na:

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

```
// Rozszerzenie dla typu String z biblioteki standardowej
extension String {
    func isEmpty() ->Bool {
        return self != ""
    }
}
```

Listing 8: Przykład rozszerzenia w Swift

Rozdział 3.

Podobieństwa i różnice pomiędzy Swiftem i Objective-C

TODO: *Dwa słowa o samym Objective-C?*

3.1. Podobieństwa

3.1.1. Swift i Objective-C jako języki programowania obiektowego

Zarówno Swift, jak i Objective-C są głównie językami programowania obiektowego. Oba języki pozwalają na definiowanie własnych typów (klasy i typy wyliczeniowe w Objective-C, struktury, klasy oraz typy wyliczeniowe w Swifcie), wspierają dziedziczenie i polimorfizm. Dzięki modyfikatorom dostępu umożliwiają również enkapsulację implementacji i opisywanie zachowań za pomocą abstrakcyjnych typów danych (protokoły w Objective-C, interfejsy w Swifcie).

```
#import <Foundation/Foundation.h>

// Definicja typu Book
@interface Book: NSObject
@property (copy, nonatomic, readonly) NSString *title;
@property (copy, nonatomic, readonly) NSString *author;
@property (copy, nonatomic, readonly) NSNumber *numPages;
- (instancetype)initWithTitle:(NSString *)title author:(NSString *)author
  numPages:(NSNumber *)numPages;
@end

@implementation Book
- (instancetype)initWithTitle:(NSString *)title author:(NSString *)author
  numPages:(NSNumber *)numPages {
    if (self = [super init]) {
        _title = title;
        _author = author;
```

```
        _numPages = numPages;
    }
    return self;
}
@end

// Definicja interfejsu BookPrinter
@protocol BookPrinter
- (void)printBook:(Book *)user;
@end

// Definicja klasy ConsoleBookPrinter implementującego protokół BookPrinter
@interface ConsoleBookPrinter: NSObject <BookPrinter>
@end

@implementation ConsoleBookPrinter
- (void)printBook:(Book *)book {
    NSLog(@"------------------------------");
    NSLog(@"> Tytul: %@", book.title);
    NSLog(@"> Autor: %@", book.author);
    NSLog(@"> Ilosc stron: %@", book.numPages);
    NSLog(@"------------------------------");
}
@end

int main (int argc, const char * argv[])
{
    Book* book = [[Book alloc] initWithTitle: @"Sztuka programowania"
                                      author: @"Donald Knuth"
                                    numPages: @(2338)];
    id<BookPrinter> printer = [ConsoleBookPrinter new];
    [printer printBook: book];
    return 0;
}
```

Listing 9: Przykład kodu obiektowego w Objective-C

```
struct Book {
    let title: String
    let author: String
    let numPages: Int
}

protocol BookPrinter {
    func printBook(_ book: Book)
}

class ConsoleBookPrinter: BookPrinter {
    func printBook(_ book: Book) {
        print("------------------------------");
        print("> Tytul: \(book.title)");
```

```
        print("> Autor: \(book.author)");
        print("> Ilosc stron: \(book.numPages)");
        print("------------------------------");
    }
}

let book = Book(title: "Sztuka programowania", author: "Donald Knuth", numPages: 2338)
let printer: BookPrinter = ConsoleBookPrinter()
printer.printBook(book)
```

Listing 10: Analogiczny kod napisany w Swifcie

3.1.2. Zarządzanie pamięcią

W początkowych wersjach systemu OSX i iOS zarządzanie pamięcią było w pełni manualne - co prawda obiekty (a raczej wskaźniki do nich) posiadały liczniki referencji, jednak programista musiał sam zadbać o zarządzanie tymi licznikami. Przełom nastąpił w roku 2011, kiedy to do Objective-C zostało dodane automatyczne zliczanie referencji (ang. *automatic reference counting*, w skrócie: ARC).

Automatyczne zliczanie referencji to jedna z najprostszych metod zarządzania pamięcią, odciążajaca programistę z obowiązku jawnego inkrementowania i dekrementowania liczników referencji. Użycie ARC w Objective-C powoduje wygenerowanie kodu, który zwiększa licznik referencji w momencie, gdy nowa referencja do obiektu zostaje utworzona (np. inicjalizacja, przypisanie, przekazania obiektu w parametrze) oraz zmniejsza go, w momencie usunięcia referencji. Dzięki temu programista nie musi manualnie używać funkcji `retain` i `release`, jak to miało miejsce w poprzednich wersjach Objective-C. Pozwala to na wyłapanie wielu błędów, takich jak: wycieki pamięci, wielokrotne zwalnianie pamięci czy odwoływanie się do wcześniej zwolnionej pamięci. Jednocześnie, użycie ARC nie wprowadza niedeterminizmu, co ma miejsce w przypadku użycia automatycznego odśmiecania pamięci (ang. *garbage collector*) oraz ma znikomy wpływ na wydajność działania aplikacji.

ARC jest również metodą zarządzania pamięcią zaimplementowaną w Swifcie. Główną różnicą jest jednak sposób implementacji. W Objective-C, ARC jest rozszerzeniem języka, operającym się głównie na preprocesorze i generowaniu kodu odpowiedzialnego za zliczanie referencji. W Swifcie natomiast, ARC jest jego podstawową cechą, posiadającą odrębną składnię i wparcie ze strony środowiska uruchomieniowego.