Jon Hoffman

Swift 4

Koduj jak mistrz

Wydanie IV

Tytuł oryginału: Mastering Swift 4 - Fourth Edition

Tłumaczenie: Robert Górczyński

ISBN: 978-83-283-4794-6

Helion SA
ul. Kościuszki 1c, 44-100 Gliwice
tel. 32 231 22 19, 32 230 98 63
e-mail: *helion@helion.pl*
WWW: *http://helion.pl* (księgarnia internetowa, katalog książek)

Drogi Czytelniku!
Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres
http://helion.pl/user/opinie/sw4km4
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Printed in Poland.

Spis treści

O autorze 11

O recenzencie technicznym 12

Wprowadzenie 13

Rozdział 1. Pierwsze kroki w języku Swift 17

Czym jest Swift? 18
Funkcje języka Swift 19
Plik typu playground 21
Rozpoczęcie pracy z plikiem typu playground 21
Typ pliku playground 24
Wyświetlanie obrazu w pliku playground 25
Tworzenie i wyświetlanie wykresu w pliku playground 28
Czym nie jest plik typu playground? 29
Składnia języka Swift 29
Komentarze 30
Średniki 32
Nawiasy okrągłe 33
Nawiasy klamrowe 33
Operator przypisania nie zwraca wartości 34
Białe znaki w konstrukcjach warunkowych i poleceniach przypisania są opcjonalne 35
Program wyświetlający komunikat Witaj, świecie! 35
Podsumowanie 37

Rozdział 2. Zmienne, stałe, ciągi tekstowe i operatory 39

Zmienne i stałe 40
Definiowanie zmiennych i stałych 41
Bezpieczeństwo typu 42
Inferencja typu 43
Jawne określenie typu 43
Typy liczbowe 44
Wartości boolowskie 48
Ciąg tekstowy 48
Zmienne typu opcjonalnego 52
Dołączanie wartości typu opcjonalnego 54
Łączenie wartości typu opcjonalnego 55
Typy wyliczeniowe 57
Operatory 61
Operator przypisania 61
Operatory porównania 61
Operatory arytmetyczne 62
Operator reszty z dzielenia 62
Złożone operatory przypisania 63
Trójargumentowy operator warunkowy 63
Operator logiczny NOT 63
Operator logiczny AND 64
Operator logiczny OR 64
Podsumowanie 64

Rozdział 3. Krotki i kolekcje 67

Typy kolekcji w Swifcie 67
Modyfikowalność 68
Tablica 68
Tworzenie oraz inicjalizacja tablicy 69
Uzyskanie dostępu do elementu tablicy 70
Zliczanie elementów tablicy 71
Czy tablica jest pusta? 72
Dodawanie elementu do tablicy 72
Wstawienie wartości do tablicy 73
Zastępowanie elementu tablicy 73
Usunięcie elementu z tablicy 73
Połączenie dwóch tablic 74
Pobranie podtablicy z tablicy 74
Wprowadzenie wielu zmian w tablicy 75
Algorytmy dla tablic 75
Iteracja przez tablicę 78
Słownik 79
Utworzenie oraz inicjalizacja słownika 79
Uzyskanie dostępu do wartości słownika 80
Zliczanie kluczy lub wartości w słowniku 80
Czy słownik jest pusty? 80
Uaktualnienie wartości klucza 81

Dodanie pary klucz-wartość 81
Usunięcie pary klucz-wartość 82
Zbiór 82
Inicjalizacja zbioru 82
Wstawianie elementów do zbioru 83
Określenie liczby elementów w zbiorze 83
Sprawdzenie, czy zbiór zawiera dany element 84
Iteracja przez zbiór 84
Usunięcie elementu zbioru 84
Operacje na zbiorze 84
Krotka 86
Podsumowanie 87

Rozdział 4. Funkcje programu i sterowanie przebiegiem ich działania 89

Czego nauczyłeś się dotąd z książki? 90
Nawias klamrowy 90
Nawias okrągły 90
Sterowanie przebiegiem działania programu 91
Konstrukcje warunkowe 91
Pętla for-in 94
Pętla while 96
Konstrukcja switch 97
Używanie bloków case i klauzul where w konstrukcjach warunkowych 101
Polecenia transferu kontroli 105
Funkcje 107
Funkcja z pojedynczym parametrem 107
Funkcja z wieloma parametrami 109
Zdefiniowanie wartości domyślnych parametrów 109
Zwrot wielu wartości przez funkcję 110
Zwrot wartości typu opcjonalnego 111
Dodawanie zewnętrznych nazw parametrów 112
Używanie parametrów wariadycznych 113
Parametr inout 114
Zebranie wszystkiego w całość 114
Podsumowanie 115

Rozdział 5. Klasy i struktury 117

Czym są klasy i struktury? 118
Podobieństwa między klasami i strukturami 118
Różnice między klasami i strukturami 118
Przekazywanie przez wartość kontra przez referencję 119
Utworzenie klasy lub struktury 120
Właściwość 120
Właściwość przechowywana 120
Właściwość obliczana 122
Obserwator właściwości 125
Metoda 126

Własna metoda inicjalizacyjna **128**
Wewnętrzne i zewnętrzne nazwy parametru metody inicjalizacyjnej 130
Metoda inicjalizacyjna, której działanie może zakończyć się niepowodzeniem 130
Kontrola dostępu **132**
Dziedziczenie **133**
Nadpisanie metody lub właściwości **135**
Nadpisywanie metody 136
Nadpisywanie właściwości 137
Uniemożliwianie nadpisywania 138
Protokoły **138**
Składnia protokołu **139**
Wymagania właściwości 139
Wymagania metody 140
Rozszerzenie **142**
Zarządzanie pamięcią **143**
Sposób działania mechanizmu ARC 143
Cykl silnych odwołań 145
Podsumowanie **149**

Rozdział 6. Protokoły i rozszerzenia protokołów **151**

Protokół jako typ danych **152**
Polimorfizm za pomocą protokołów **154**
Rzutowanie typu i protokół **154**
Rozszerzenie protokołu **156**
Czy trzeba używać protokołów? **163**
Biblioteka standardowa Swifta **164**
Podsumowanie **165**

Rozdział 7. Projekt oparty na protokołach **167**

Wymagania **168**
Projekt zorientowany obiektowo **168**
Projekt zorientowany na protokoły **174**
Dziedziczenie protokołu 174
Kompozycja protokołu 175
Programowanie zorientowane na protokoły 176
Używanie klauzuli where z protokołem 179
Struktura kontra klasa **180**
Struktura tablicy **181**
Podsumowanie **182**

Rozdział 8. Tworzenie bezpiecznego kodu za pomocą atrybutu available i obsługi błędów **183**

Natywna obsługa błędów **184**
Przedstawienie błędu 184
Zgłaszanie błędu 185
Przechwytywanie błędu 187
Atrybut available **191**
Podsumowanie **192**

Rozdział 9. Niestandardowe indeksy 193

Wprowadzenie do indeksów 194
Indeks w tablicy Swifta 194
Tworzenie i używanie niestandardowego indeksu 195
Niestandardowy indeks tylko do odczytu 196
Indeks obliczany 197
Wartość indeksu 197
Nazwa zewnętrzna dla indeksu 198
Indeks wielowymiarowy 198
Kiedy nie należy używać niestandardowego indeksu? 201
Podsumowanie 202

Rozdział 10. Typy opcjonalne 203

Wprowadzenie do typu opcjonalnego 203
Potrzeba istnienia typów opcjonalnych w Swifcie 205
Definiowanie wartości typu opcjonalnego 206
Używanie wartości typu opcjonalnego 206
Łączenie wartości typu opcjonalnego 211
Operator koalescencji nil 213
Podsumowanie 214

Rozdział 11. Typy generyczne 215

Wprowadzenie do typu generycznego 215
Funkcja generyczna 216
Typ generyczny 220
Indeks generyczny 223
Typ powiązany 224
Podsumowanie 226

Rozdział 12. Domknięcia 227

Wprowadzenie do domknięcia 227
Proste domknięcia 228
Skrócona składnia domknięcia 230
Używanie domknięcia wraz z algorytmem tablicy Swifta 233
Samodzielne domknięcia i wskazówki dotyczące dobrego stylu 237
Zmiana funkcjonalności 239
Wybór domknięcia na podstawie wyniku 242
Utworzenie cyklu silnych odwołań za pomocą domknięć 244
Podsumowanie 247

Rozdział 13. Połączenie Swifta i Objective-C 249

Połączenie Swifta i Objective-C 249
Kiedy łączyć kod Swifta i Objective-C? 250
Użycie Swifta i Objective-C w tym samym projekcie 251
Utworzenie projektu 251
Dodawanie pliku Swifta do projektu Objective-C 253
Plik Objective-C Bridging Header — część 1. 255

Dodawanie pliku Objective-C do projektu 256
Klasa Objective-C Messages 258
Plik Objective-C Bridging Header — część 2. 259
Klasa Swifta MessageBuilder — dostęp do kodu Objective-C z poziomu Swifta 259
Klasa Objective-C — dostęp do kodu Swifta z poziomu Objective-C 260
Podsumowanie 261

Rozdział 14. Programowanie równoległe i współbieżność 263

Równoległość i współbieżność 264
Grand Central Dispatch 265
Typ DoCalculations 266
Użycie typów Operation i OperationQueue 272
Podsumowanie 277

Rozdział 15. Formatowanie kodu Swifta i przewodnik po jego stylu 279

Czym jest styl programowania? 280
Twój styl programowania 281
Nie używaj średnika na końcu polecenia 281
Nie używaj nawiasu w konstrukcji warunkowej 281
Konwencja nazw 282
Komentarze 283
Użycie słowa kluczowego self 284
Stałe i zmienne 285
Typy opcjonalne 285
Użycie inferencji typu 286
Użycie skróconych deklaracji kolekcji 287
Użycie konstrukcji switch zamiast wielu poleceń if 287
Nie pozostawiaj w aplikacji kodu umieszczonego w komentarzu 287
Podsumowanie 288

Rozdział 16. Podstawowe biblioteki Swifta 289

System wczytywania adresów URL 290
URLSession 291
URLSessionConfiguration 291
URLSessionTask 291
URL 292
URLRequest 292
HTTPURLResponse 292
Usługa sieciowa typu REST 292
Wykonywanie żądania HTTP GET 293
Wykonywanie żądania HTTP POST 296
Formatter 298
DateFormatter 298
NumberFormatter 300
FileManager 301
Kodowanie i dekodowanie danych JSON 304
Użycie JSONEncoder 305
Użycie JSONDecoder 306
Podsumowanie 307

Rozdział 17. Wzorce projektowe w Swifcie **309**

Czym są wzorce projektowe? **310**
Wzorce konstrukcyjne **311**
Wzorzec singleton 312
Wzorzec budowniczego 315
Wzorce strukturalne **320**
Wzorzec mostu 320
Wzorzec fasady 324
Wzorzec pełnomocnika 327
Wzorce operacyjne **330**
Wzorzec polecenia 330
Wzorzec strategii 333
Podsumowanie **335**

Skorowidz **337**

ROZDZIAŁ 6

Protokoły i rozszerzenia protokołów

Oglądając pochodzącą z konferencji WWDC 2015 prezentację dotyczącą rozszerzeń protokołów i **programowania zorientowanego na protokoły** (ang. *protocol-oriented programming*), byłem bardzo sceptycznie nastawiony do koncepcji przedstawionych w tej prezentacji. Od bardzo dawna stosowałem **programowanie zorientowane obiektowo** i nie byłem przekonany, czy zgodnie z założeniami Apple nowy paradygmat programowania jest w stanie rozwiązać wszystkie problemy. Ponieważ nie należę do osób, u których sceptycyzm uniemożliwia działanie, przygotowałem nowy projekt powielający mój bieżący, ale kod zacząłem tworzyć, stosując zalecenia Apple dotyczące programowania zorientowanego na protokoły. W nowo tworzonym kodzie bardzo często korzystałem z rozszerzeń protokołów. Muszę przyznać, że byłem bardzo zaskoczony tym, iż nowy projekt okazał się znacznie bardziej przejrzysty niż oryginalny. Jestem przekonany, że obsługa rozszerzenia protokołów stanie się jedną z tych funkcji, która będzie odróżniała od siebie języki programowania. Wierzę, że podobne funkcje pojawią się wkrótce w innych najważniejszych językach programowania.

W rozdziale:

- dowiesz się, w jaki sposób protokoły są używane jako typy;
- zobaczysz, jak można zaimplementować polimorfizm w Swifcie, używając protokołów;
- zobaczysz, jak można używać rozszerzeń protokołów;
- dowiesz się, dlaczego miałbyś korzystać z rozszerzeń protokołów.

Wprawdzie rozszerzenia protokołów to w zasadzie lukier syntaktyczny, ale według mnie stanowią one jedną z najważniejszych funkcji, jaka została dodana do języka programowania Swift. Dzięki rozszerzeniom protokołów można dostarczać implementacje metod i właściwości dowolnemu typowi zgodnemu z protokołem. Aby naprawdę dobrze zrozumieć użyteczność protokołów i rozszerzeń protokołów, najpierw należy dokładnie poznać protokoły.

Choć w języku Swift z protokołami mogą być zgodne zarówno klasy, struktury, jak i typy wyliczeniowe, w tym rozdziale skoncentruję się wyłącznie na klasach i strukturach. Typy wyliczeniowe będą używane wtedy, gdy zajdzie potrzeba przedstawienia pewnej liczby przypadków. Wprawdzie istnieją sytuacje, w których typ wyliczeniowy powinien być zgodny z protokołem, ale zdarza się to bardzo rzadko. Po prostu zapamiętaj, że wszędzie tam, gdzie odwołuję się do klasy lub struktury, można użyć również typu wyliczeniowego.

Na początek przedstawię protokoły i wyjaśnię, dlaczego są one pełnoprawnymi typami w Swifcie.

Protokół jako typ danych

Wprawdzie protokół nie implementuje żadnej funkcjonalności, ale jest uznawany za pełnoprawny typ danych w języku programowania Swift i może być wykorzystywany w taki sam sposób jak każdy inny typ. Oznacza to możliwość użycia protokołu jako typu parametru lub wartości zwrotnej funkcji. Ponadto protokół można podać jako typ zmiennej, stałej lub kolekcji. Spójrz na kilka przykładów, w których zostanie wykorzystany protokół `PersonProtocol` zdefiniowany w następujący sposób:

```
protocol PersonProtocol {
  var firstName: String { get set }
  var lastName: String { get set }
  var birthDate: Date { get set }
  var profession: String { get }
  init(firstName: String,lastName: String, birthDate: Date)
}
```

W pierwszym przykładzie protokół zostanie wykorzystany jako typ parametru i wartości zwrotnej funkcji.

```
func updatePerson(person: PersonProtocol) -> PersonProtocol {
  // Miejsce na kod uaktualniający informacje o osobie i zwracający je.
}
```

W tym przykładzie funkcja `updatePerson()` akceptuje jeden parametr typu protokołu `PersonProtocol` i zwraca wartość również będącą typu wymienionego protokołu. W następnym przykładzie pokazałem, jak można użyć protokołu jako typu dla stałej, zmiennej lub właściwości.

```
var myPerson: PersonProtocol
```

W tym przykładzie utworzona została zmienna o nazwie myPerson i typie protokołu PersonProtocol. Protokół może być również używany jako typ elementu przeznaczonego do umieszczenia w kolekcji, takiej jak tablica, słownik lub zbiór.

```
var people: [PersonProtocol] = []
```

W omawianym przykładzie została utworzona tablica typu protokołu PersonProtocol. Choć ten protokół nie implementuje żadnej funkcjonalności, mimo wszystko może zostać użyty do określenia typu. Jednak protokół nie może zostać utworzony w taki sam sposób jak klasa lub struktura. Wynika to z tego, że protokół nie implementuje żadnej funkcjonalności. Dlatego też kompilator wygeneruje błąd podczas próby utworzenia egzemplarza protokołu PersonProtocol w następujący sposób:

```
var test = PersonProtocol(firstName: "Jon", lastName: "Hoffman",
birthDate:bDateProgrammer)
```

Gdy wymagany jest typ protokołu, można użyć egzemplarza dowolnego typu zgodnego z tym protokołem. Na przykład po zdefiniowaniu zmiennej typu protokołu PersonProtocol można jej użyć w dowolnej klasie lub strukturze zgodnej z tym protokołem. Na potrzeby kolejnego przykładu przyjmuję założenie o istnieniu dwóch typów SwiftProgrammer i FootballPlayer zgodnych z protokołem PersonProtocol.

```
var myPerson: PersonProtocol

myPerson = SwiftProgrammer(firstName: "Jon", lastName: "Hoffman",
                           birthDate: bDateProgrammer)
print("\(myPerson.firstName) \(myPerson.lastName)")

myPerson = FootballPlayer(firstName: "Dan", lastName: "Marino",
                          birthDate:bDatePlayer)
print("\(myPerson.firstName) \(myPerson.lastName)")
```

W omawianym fragmencie kodu najpierw utworzyłem zmienną myPerson typu protokołu PersonProtocol. Następnie przypisałem jej egzemplarz typu SwiftProgrammer oraz wyświetliłem wartości właściwości firstName i lastName. Dalej zmiennej myPerson przypisałem egzemplarz typu FootballPlayer i ponownie wyświetliłem wartości właściwości firstName i lastName. Warto w tym miejscu wspomnieć, że dla Swifta nie ma znaczenia, czy egzemplarz jest klasą, czy strukturą. Ważna jest jedynie zgodność typu z typem protokołu PersonProtocol.

Jak wcześniej zobaczyłeś, protokołu PersonProtocol można użyć jako typu dla tablicy. Oznacza to możliwość wypełnienia tablicy egzemplarzami dowolnego typu zgodnego z protokołem. Przypominam raz jeszcze, że nie ma żadnego znaczenia, czy typ jest klasą, czy strukturą, o ile jest zgodny z protokołem. PersonProtocol.

Polimorfizm za pomocą protokołów

We wcześniejszych przykładach można było dostrzec pewne formy polimorfizmu. Słowo polimorfizm (ang. *polymorphism*) pochodzi z języka greckiego, w którym *poly* oznacza wielkość, a *morphe* postać. W językach programowania polimorfizm to pojedyncze dziedziczenie wielu typów (wielu postaci). W przykładach przedstawionych dotąd w rozdziale pojedynczym interfejsem był protokół `PersonProtocol`, a wiele innych typów było z nim zgodnych.

Polimorfizm pozwala na pracę z wieloma typami w ujednolicony sposób. Aby to zilustrować, można rozbudować poprzedni przykład, w którym została utworzona tablica typów `PersonProtocol`, i przeprowadzić iterację przez jej elementy. Następnie dostęp do poszczególnych elementów tablicy odbywa się za pomocą właściwości i metod zdefiniowanych w protokole `PersonProtocol` niezależnie od rzeczywistego typu. Spójrz na kolejny przykład.

```
for person in people {
  print("\(person.firstName) \(person.lastName):\(person.profession)")
}
```

W rozdziale kilkakrotnie wspomniałem, że po zdefiniowaniu typu zmiennej, stałej, kolekcji itd. jako protokołu można używać dowolnego egzemplarza typu zgodnego z tym protokołem. To jest bardzo ważna koncepcja do zrozumienia i zarazem jedna z wielu cech, dzięki którym protokoły i rozszerzenia protokołów mają tak potężne możliwości.

Podczas użycia protokołu w celu uzyskania dostępu do egzemplarza, jak pokazałem w poprzednim przykładzie, można używać jedynie metod i właściwości zdefiniowanych w samym protokole. Jeżeli mają być używane metody i właściwości charakterystyczne dla poszczególnych typów, konieczne jest rzutowanie egzemplarza na ten typ.

Rzutowanie typu i protokół

Rzutowanie typu to sposób na sprawdzenie typu egzemplarza lub potraktowanie egzemplarza tak, jakby był określonego typu. W Swifcie słowo kluczowe `is` jest używane do sprawdzenia, czy egzemplarz jest podanego typu, natomiast słowo kluczowe `as` służy do potraktowania egzemplarza jako tego typu.

Na początek pokażę, jak można sprawdzić typ egzemplarza za pomocą słowa kluczowego `is`. Spójrz na następujący fragment kodu:

```
for person in people {
  if person is SwiftProgrammer {
    print("\(person.firstName) to programista Swifta.")
  }
}
```

W omawianym przykładzie konstrukcja warunkowa `if` została użyta do sprawdzenia, czy każdy element tablicy `people` jest egzemplarzem typu `SwiftProgrammer`. Jeżeli tak, w konsoli zostanie wyświetlony komunikat informujący, że dana osoba jest programistą Swifta. Wprawdzie jest to dobra metoda na sprawdzenie, czy masz do czynienia z egzemplarzem konkretnej klasy lub struktury, ale nie będzie zbyt efektywna, gdy trzeba będzie sprawdzić egzemplarz pod kątem wielu typów. W takich przypadkach znacznie efektywniejsze będzie użycie konstrukcji `switch`.

```
for person in people {
  switch person {
    case is SwiftProgrammer:
      print("\(person.firstName) to programista Swifta.")
    case is FootballPlayer:
      print("\(person.firstName) to sportowiec.")
    default:
      print("Nie wiadomo, czym się zajmuje \(person.firstName).")
  }
}
```

W omawianym fragmencie kodu pokazałem wykorzystanie konstrukcji `switch` do sprawdzenia typu egzemplarza wszystkich elementów tablicy. Do wykonania tej operacji w poszczególnych blokach `case` zostało użyte słowo kluczowe `is`, aby podjąć próbę dopasowania typu egzemplarza.

W rozdziale 4. pokazałem, jak można filtrować konstrukcje warunkowe za pomocą klauzuli `where`. Tę klauzulę można wykorzystać również wraz ze słowem kluczowym `is` do filtrowania tablicy, np.:

```
for person in people where person is SwiftProgrammer {
  print("\(person.firstName) to programista Swifta.")
}
```

Przechodzę teraz do rzutowania egzemplarza klasy lub struktury na określony typ. Do tego należy użyć słowa kluczowego `as`. Ponieważ rzutowanie może zakończyć się niepowodzeniem — jeśli egzemplarz nie jest określonego typu — więc słowo kluczowe `as` jest dostarczane w dwóch postaciach: `as?` i `as!`. W przypadku postaci `as?`, jeżeli rzutowanie zakończy się niepowodzeniem, wartością zwrotną będzie `nil`. Natomiast w przypadku postaci `as!` niepowodzenie rzutowania powoduje wygenerowanie błędu w trakcie działania aplikacji. Dlatego też zaleca się użycie `as?`, o ile nie ma absolutnej pewności dotyczącej typu egzemplarza lub jeśli sprawdzenie typu egzemplarza odbywa się przed przeprowadzeniem rzutowania.

> Wprawdzie w książce przedstawię przykłady rzutowania za pomocą słowa kluczowego `as!`, ale odradzam jego używanie w rzeczywistych projektach właśnie ze względu na możliwość wygenerowania błędów w trakcie działania aplikacji.

Spójrz na przykład pokazujący, jak można wykorzystać słowo kluczowe `as?` do rzutowania egzemplarza klasy lub struktury na określony typ.

```
for person in people {
  if let p = person as? SwiftProgrammer {
    print("\(person.firstName) to programista Swifta.")
  }
}
```

Ponieważ słowo kluczowe `as?` zwraca wartość typu opcjonalnego, zastosowany został mechanizm dołączania wartości typu opcjonalnego, jak pokazałem w przykładzie.

Po poznaniu podstaw związanych z protokołami można przejść do znacznie bardziej ekscytującej funkcji Swifta, czyli rozszerzenia protokołu.

Rozszerzenie protokołu

Rozszerzenie protokołu pozwala na rozbudowę protokołu mającą na celu dostarczenie implementacji metod i właściwości typom zgodnym z danym protokołem. Rozszerzenie protokołu pozwala również na przygotowanie często używanych implementacji wszystkim zgodnym typom, co eliminuje konieczność oddzielnego dostarczania implementacji poszczególnym typom lub tworzenia hierarchii klas. Wprawdzie rozszerzenie protokołu może nie wydawać się zbyt ekscytujące, ale gdy tylko poznasz jego potężne możliwości, zmienisz sposób myślenia dotyczący tworzenia kodu.

Na początek pokażę, jak można użyć rozszerzenia protokołu w bardzo prostym przykładzie. Pracę należy rozpocząć od zdefiniowania protokołu o nazwie `DogProtocol`:

```
protocol DogProtocol {
  var name: String { get set }
  var color: String { get set }
}
```

Mając ten protokół, wskazujesz, że każdy zgodny z nim typ musi zawierać dwie właściwości typu `String` o nazwach `name` i `color`. Kolejnym krokiem jest zdefiniowanie trzech typów zgodnych z tym protokołem. Nowym typom nadałem nazwy `JackRussel`, `WhiteLab` i `Mutt`, a ich definicje przedstawiają się następująco:

```
struct JackRussel: DogProtocol {
  var name: String
  var color: String
}

class WhiteLab: DogProtocol {
  var name: String
  var color: String
  init(name: String, color: String) {
    self.name = name
    self.color = color
  }
}
```

```
struct Mutt: DogProtocol {
  var name: String
  var color: String
}
```

Celowo utworzyłem typy `JackRussel` i `Mutt` jako struktury, a typ `WhiteLab` jako klasę, aby zaprezentować różnice między nimi oraz pokazać, że są traktowane dokładnie w taki sam sposób podczas pracy zarówno z protokołami, jak i rozszerzeniami protokołów.

Największa różnica widoczna w omawianym przykładzie polega na tym, że typ struktury dostarcza domyślną metodę inicjalizacyjną, natomiast klasa musi mieć jawnie zdefiniowaną metodę inicjalizacyjną, aby przypisać wartości początkowe właściwościom.

Przyjmuję założenie, że każdemu typowi zgodnemu z protokołem `DogProtocol` chcę dostarczyć metodę o nazwie `speak()`. Przed wprowadzeniem rozszerzenia protokołu wymagałoby to dodania definicji metody do protokołu na przykład w następujący sposób:

```
protocol DogProtocol {
  var name: String { get set }
  var color: String { get set }
  func speak() -> String
}
```

Gdy metoda jest zdefiniowana w protokole, wówczas trzeba dostarczyć jej implementację w każdym typie zgodnym z tym protokołem. W zależności od liczby typów zgodnych z danym protokołem implementacja nowej metody może wymagać sporo czasu i utworzenia dużej ilości kodu. W kolejnym fragmencie kodu pokazałem, jak może przedstawiać się przykładowa implementacja metody `speak()`:

```
struct JackRussel: DogProtocol {
  var name: String
  var color: String
  func speak() -> String {
    return "hau hau"
  }
}

class WhiteLab: DogProtocol {
  var name: String
  var color: String
  init(name: String, color: String) {self.name = nameself.color = color}
  func speak() -> String {
  return "hau hau"
  }
}

struct Mutt: DogProtocol {
  var name: String
  var color: String
  func speak() -> String {
    return "hau hau"
  }
}
```

Wprawdzie przedstawione tutaj rozwiązanie działa, ale na pewno nie zalicza się do szczególnie efektywnych, ponieważ po każdym uaktualnieniu protokołu konieczne będzie zmodyfikowanie również wszystkich zgodnych z nim typów. To oznacza dużą ilość powielonego kodu, jak pokazałem w omawianym przykładzie. Ponadto jeśli zajdzie potrzeba zmiany domyślnego sposobu działania metody `speak()`, wówczas trzeba będzie sprawdzić każdą implementację i zmienić tę metodę. W tym momencie do gry wchodzi rozszerzenie protokołu.

Dzięki rozszerzeniu protokołu można wyciągnąć z protokołu definicję metody `speak()` i zdefiniować ją wraz z domyślnym sposobem działania w rozszerzeniu protokołu.

> Jeżeli implementujesz metodę w rozszerzeniu protokołu, nie trzeba jej definiować w protokole.

W kolejnym fragmencie kodu pokazałem, jak można zdefiniować protokół i jego rozszerzenie.

```
protocol DogProtocol {
  var name: String { get set }
  var color: String { get set }
}

extension DogProtocol {
  func speak() -> String {
    return "hau hau"
  }
}
```

Na początku znajduje się definicja protokołu `DogProtocol` wraz z dwiema wcześniej użytymi właściwościami. Następnie zostało utworzone rozszerzenie protokołu zawierające domyślną implementację metody `speak()`. Mając tak przygotowany kod, nie trzeba już teraz dostarczać implementacji metody `speak()` we wszystkich typach zgodnych z protokołem `DogProtocol`, ponieważ będą one automatycznie otrzymywały implementację jako część protokołu.

Praktyczne zastosowanie takiego rozwiązania pokażę na przykładzie przywrócenia trzem typom zgodnym z protokołem `DogProtocol` ich początkowych implementacji. W takim przypadku metodę `speak()` powinny otrzymać dzięki rozszerzeniu protokołu.

```
struct JackRussel: DogProtocol {
  var name: String
  var color: String
}
class WhiteLab: DogProtocol {
  var name: String
  var color: String
  init(name: String, color: String) {
    self.name = name
    self.color = color
  }
}

struct Mutt: DogProtocol {
  var name: String
  var color: String
}
```

Zobacz teraz, jak przygotowane typy można zastosować w kodzie.

```
let dash = JackRussel(name: "Dash", color: "brązowy i biały")
let lily = WhiteLab(name: "Lily", color: "biały")
let maple = Mutt(name: "Buddy", color: "brązowy")
let dSpeak = dash.speak()   // Wartością zwrotną jest "hau hau".
let lSpeak = lily.speak()   // Wartością zwrotną jest "hau hau".
let bSpeak = maple.speak()    // Wartością zwrotną jest "hau hau".
```

Jak widać w omawianym przykładzie, zdefiniowanie metody `speak()` w rozszerzeniu protokołu powoduje jej automatyczne dodanie do wszystkich typów zgodnych z tym protokołem. Dlatego też tutaj metoda `speak()` umieszczona w rozszerzeniu protokołu może być uznawana za domyślną implementację metody, ponieważ można ją nadpisać w poszczególnych implementacjach typów. Na przykład metodę `speak()` można nadpisać w strukturze `Mutt`, jak pokazałem w kolejnym fragmencie kodu.

```
struct Mutt: DogProtocol {
  var name: String
  var color: String
  func speak() -> String {
    return "Jestem głodny"
  }
}
```

Po wywołaniu metody `speak()` egzemplarza `Mutt` wartością zwrotną będzie ciąg tekstowy `Jestem głodny`.

> W tym rozdziale definiowanym protokołom dodaję przyrostek `Protocol`. Zdecydowałem się na to, aby wyraźnie pokazać, kiedy mamy do czynienia z protokołami. To nie jest standardowe podejście w zakresie nadawania nazw typom. Przedstawiony nieco dalej w tekście przykład znacznie lepiej pokazuje, jak należy prawidłowo nadawać nazwy protokołom. Więcej informacji na temat konwencji nazw w języku Swift znajdziesz na stronie *https://swift.org/documentation/api-design-guidelines/#general-conventions*.

Zobaczyłeś już, jak można używać protokołów i ich rozszerzeń, przechodzę więc teraz do znacznie bardziej praktycznego przykładu. W wielu aplikacjach dostępnych na różnych platformach (iOS, Android, Windows) zachodzi potrzeba weryfikacji danych wejściowych wprowadzonych przez użytkownika. Taką weryfikację można bardzo łatwo przeprowadzić za pomocą wyrażeń regularnych. Jednak zwykle nie chcemy, aby różne wyrażenia regularne były porozrzucane w wielu miejscach kodu źródłowego. Ten problem można bardzo łatwo rozwiązać poprzez utworzenie oddzielnych klas lub struktur zawierających kod odpowiedzialny za przeprowadzenie weryfikacji danych wejściowych. Typy te trzeba jednak zorganizować w określony sposób, aby ułatwić ich użycie i późniejszą konserwację. Przed wprowadzeniem rozszerzenia protokołu w Swifcie trzeba było użyć protokołu do zdefiniowania wymagań dotyczących weryfikacji danych wejściowych, a następnie utworzyć zgodne z nim struktury dla każdej potrzebnej operacji sprawdzania danych. Spójrz na rozwiązanie, które było stosowane przed wprowadzeniem rozszerzenia protokołu.

Wyrażenie regularne to sekwencja znaków definiujących określony wzorzec. Ten wzorzec może być następnie używany do przeszukiwania ciągów tekstowych i sprawdzania, czy ciąg tekstowy jest dopasowany do wzorca lub czy zawiera dopasowanie wzorca. Większość języków programowania ma wbudowany pewien analizator składni wyrażeń regularnych. Jeżeli nie znasz wyrażeń regularnych, naprawdę warto poświęcić czas na ich opanowanie.

W kolejnym fragmencie kodu przedstawiłem protokół `TextValidating` definiujący wymagania dla każdego typu, który ma być używany podczas weryfikacji danych wejściowych.

```
protocol TextValidating {
  var regExMatchingString: String { get }
  var regExFindMatchString: String { get }
  var validationMessage: String { get }
  func validateString(str: String) -> Bool
  func getMatchingString(str: String) -> String?
}
```

Zgodnie z dokumentem umieszczonym na stronie *https://swift.org/documentation/api-design-guidelines/* protokół wskazujący na obecność czegoś powinien mieć nazwę w postaci rzeczownika w języku angielskim, natomiast protokół opisujący możliwości powinien mieć nazwę z przyrostkiem `-able`, `-ible` lub `-ing`. Mając to na uwadze, protokół otrzymał nazwę `TextValidating`.

W tym protokole zdefiniowałem trzy właściwości i dwie metody, które zgodny z nimi typ musi implementować. W kolejnych punktach przedstawiłem krótkie omówienie właściwości wymaganych przez protokół `TextValidating`.

- `regExMatchingString`. To jest ciąg tekstowy wyrażenia regularnego używanego do sprawdzenia, czy dane wejściowe składają się jedynie z poprawnych znaków.
- `regExFindMatchString`. To jest ciąg tekstowy wyrażenia regularnego używanego do pobrania z danych wejściowych nowego ciągu tekstowego zawierającego jedynie prawidłowe znaki. Ogólnie rzecz biorąc, to wyrażenie regularne jest używane wtedy, gdy zachodzi potrzeba sprawdzania danych wejściowych w czasie rzeczywistym podczas wprowadzania informacji przez użytkownika. Znajduje ono najdłuższy dopasowany prefiks danych wejściowych.
- `validationMessage`. To jest komunikat błędu, który zostanie wyświetlony, gdy ciąg tekstowy zawiera nieprawidłowe znaki.

Oto krótkie omówienie metod wymaganych przez protokół `TextValidating`:

- `validateString()`. Ta metoda zwraca wartość `true`, jeżeli ciąg tekstowy zawiera jedynie prawidłowe znaki. Aby znaleźć dopasowanie, ta metoda używa właściwości `regExMatchingString`.
- `getMatchingString()`. Ta metoda zwraca nowy ciąg tekstowy zawierający jedynie prawidłowe znaki. Ta metoda jest najczęściej używana, gdy zachodzi potrzeba sprawdzania danych wejściowych w czasie rzeczywistym podczas wprowadzania informacji przez użytkownika. Znajduje ona najdłuższy dopasowany prefiks danych wejściowych. Natomiast do pobrania nowego ciągu tekstowego ta metoda używa właściwości `regExFindMatchString`.

Przechodzę teraz do utworzenia struktury zgodnej z omówionym protokołem. Zadaniem przedstawionej tutaj struktury jest sprawdzenie, czy ciąg tekstowy danych wejściowych zawiera jedynie znaki alfanumeryczne.

```
struct AlphaValidation1: TextValidating {
  static let sharedInstance = AlphaValidation1()
  private init(){}
  let regExFindMatchString = "^[a-zA-Z]{0,10}"
  let validationMessage = "Dozwolone są jedynie litery."
  var regExMatchingString: String {
    get {
      return regExFindMatchString + "$"
    }
  }
  func validateString(str: String) -> Bool {
    if let _ = str.range(of: regExMatchingString,
                         options: .regularExpression) {
      return true
    } else {
      return false
    }
  }
  func getMatchingString(str: String) -> String? {
    if let newMatch = str.range(of: regExFindMatchString,
                                options:.regularExpression) {
      return str.substring(with:newMatch)
    } else {
      return nil
    }
  }
}
```

W przedstawionej implementacji `regExFindMatchString` i `validationMessage` to właściwości przechowywane, natomiast `regExMatchingString` to właściwość obliczana. Ta struktura implementuje również metody `validateString()` i `getMatchingString()`.

W rzeczywistym projekcie istniałoby wiele różnych typów zgodnych z tym protokołem przeznaczonych do weryfikacji odmiennych rodzajów danych wejściowych. Jak możesz zobaczyć na przykładzie struktury `AlphaValidation1`, przygotowanie każdego typu przeznaczonego do weryfikacji wymaga utworzenia znacznej ilości kodu. Ponadto duża część tego kodu będzie powielona w poszczególnych typach. Kod obu metod i właściwości `regExMatchingString` prawdopodobnie zostanie powtórzony w każdej klasie weryfikacji danych. Takie rozwiązanie jest dalekie od idealnego. Jeżeli jednak chciałbyś uniknąć tworzenia hierarchii klas wraz z superklasą zawierającą powtarzający się kod (zaleca się preferowanie typu przekazywanego przez wartość, a nie przez referencję), to przed wprowadzeniem rozszerzenia protokołu nie miałeś żadnego innego wyboru. Teraz pokażę, jak zaimplementować rozwiązanie oparte na rozszerzeniu protokołu.

W przypadku rozszerzeń protokołów należy zmienić sposób myślenia o kodzie. Największa różnica polega na tym, że nie trzeba i nie należy definiować wszystkiego w protokole. W przypadku standardowych protokołów wszystkie metody i właściwości, do których będziesz chciał mieć dostęp za pomocą interfejsu protokołu, zostaną zdefiniowane w tym protokole.

Mając do dyspozycji rozszerzenie protokołu, odradza się definiowanie w nim metody lub właściwości, jeśli ta definicja może być umieszczona w rozszerzeniu protokołu. Dlatego też w zmodyfikowanej wersji protokołu dotyczącego weryfikacji danych wejściowych `TextValidating` może zostać znacznie uproszczony do następującej postaci:

```
protocol TextValidating {
  var regExFindMatchString: String { get }
  var validationMessage: String { get }
}
```

W początkowej wersji protokołu `TextValidating` były zdefiniowane trzy właściwości i dwie metody. Jak możesz zobaczyć w zmodyfikowanej wersji, teraz protokół zawiera jedynie dwie właściwości. Po zdefiniowaniu protokołu `TextValidating` można przystąpić do przygotowania rozszerzenia tego protokołu.

```
extension TextValidating {
  var regExMatchingString: String {
    get {
      return regExFindMatchString + "$"
    }
  }
  func validateString(str: String) -> Bool {
    if let _ = str.range(of:regExMatchingString,
                         options:.regularExpression){
      return true
    } else {
      return false
    }
  }
  func getMatchingString(str: String) -> String? {
    if let newMatch = str.range(of:regExFindMatchString,
                                options:.regularExpression) {
      return str.substring(with: newMatch)
    } else {
      return nil
    }
  }
}
```

Rozszerzenie protokołu `TextValidating` zawiera dwie metody i właściwość, które wcześniej znajdowały się w pierwotnej wersji protokołu, a nie zostały uwzględnione w nowej. Mając utworzony protokół i jego rozszerzenie, można przystąpić do zdefiniowania typów odpowiedzialnych za weryfikację danych wejściowych. W kolejnym fragmencie kodu przedstawiłem trzy struktury używane do sprawdzenia tekstu wpisanego przez użytkownika.

```
struct AlphaValidation: TextValidating {
  static let sharedInstance = AlphaValidation()
   private init(){}
  let regExFindMatchString = "^[a-zA-Z]{0,10}"
  let validationMessage = "Dozwolone są jedynie litery."
}

struct AlphaNumericValidation: TextValidating {
```

```
  static let sharedInstance = AlphaNumericValidation()
  private init(){}
  let regExFindMatchString = "^[a-zA-Z0-9]{0,15}"
  let validationMessage = "Dozwolone są jedynie znaki alfanumeryczne."
}

struct DisplayNameValidation: TextValidating {
  static let sharedInstance = DisplayNameValidation()
  privateinit(){}
  let regExFindMatchString = "^[\\s?[a-zA-Z0-9\\-_\\s]]{0,15}"
  let validationMessage = "Dozwolone są jedynie znaki alfanumeryczne."
}
```

We wszystkich przedstawionych tutaj strukturach weryfikacji danych wejściowych zostały utworzone statyczne stałe i prywatne metody inicjalizacyjne, co pozwoli na użycie struktury jako wzorca singleton. Więcej informacji na temat wzorca projektowego singleton znajdziesz w rozdziale 17.

Po zdefiniowaniu wzorca singleton w poszczególnych typach trzeba przypisać wartości właściwościom `regExFindMatchString` i `validationMessage`. W ten sposób praktycznie zostanie wyeliminowany powielający się kod. Jedynym powtarzającym się kodem jest wzorzec singleton — nie zostanie on umieszczony w rozszerzeniu protokołu, aby nie wymagać stosowania tego wzorca projektowego we wszystkich typach zgodnych z danym protokołem.

Po wprowadzeniu omówionych zmian można zacząć używać nowych typów przeznaczonych do weryfikacji danych wejściowych użytkownika:

```
var testString = "abc123"

var alpha = AlphaValidation.sharedInstance
alpha.getMatchingString(str:testString)
alpha.validateString(str: testString)
```

W omówionym tutaj przykładzie został utworzony nowy ciąg tekstowy przeznaczony do sprawdzenia. Ponadto utworzyłem współdzielony egzemplarz typu `AlphaValidation`. Następnie metoda `getMatchingString()` zostanie użyta do pobrania najdłuższego prefiksu dopasowanego do ciągu tekstowego, którym w omawianym przykładzie jest `abc`. Później metoda `validateString()` sprawdza ciąg tekstowy, a ponieważ zawiera on cyfry, więc jej wartością zwrotną jest `false`.

Czy trzeba używać protokołów?

Czy trzeba korzystać z protokołów i ich rozszerzeń, gdy programista ma doświadczenie w programowaniu zorientowanym obiektowo? Krótka odpowiedź brzmi: nie, jednak stosowanie protokołów jest zalecane. W rozdziale 7. zobaczysz, dlaczego projekt oparty na protokołach oferuje potężne możliwości, i dowiesz się, dlaczego powinieneś preferować styl oparty na protokołach zamiast stylu programowania zorientowanego obiektowo. Dzięki zrozumieniu protokołów i opartego na nich projektu będziesz mógł jeszcze lepiej rozumieć bibliotekę standardową Swifta.

Biblioteka standardowa Swifta

Biblioteka standardowa Swifta definiuje podstawową warstwę funkcjonalności potrzebnej podczas tworzenia aplikacji w języku Swift. Wszystko to, czego dotąd używałem w książce, pochodzi właśnie z biblioteki standardowej Swifta. Znajdują się w niej definicje najważniejszych typów danych, takich jak `String`, `Int` i `Double`. Ponadto biblioteka standardowa definiuje kolekcje, typy opcjonalne, funkcje globalne i wszystkie protokoły, z którymi są zgodne te typy.

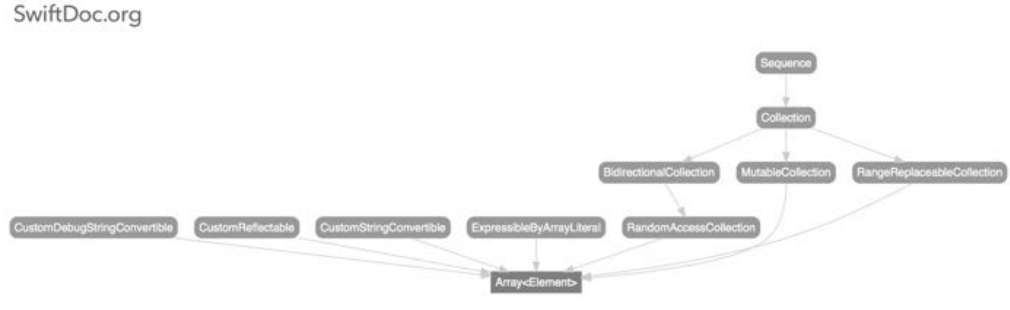
Jedną z najlepszych witryn zawierających wiele informacji na temat biblioteki standardowej Swifta jest *http://swiftdoc.org/*. Znajdziesz w niej omówienie wszystkich typów, protokołów, operatorów i funkcji globalnych tworzących bibliotekę standardową. Ta witryna zawiera również dokumentację dla tych komponentów.

Pokażę teraz, jak protokoły są używane w bibliotece standardowej. W tym celu posłużę się dokumentacją pochodzącą z witryny *http://swiftdoc.org/*. Gdy po raz pierwszy odwiedzisz tę witrynę, zobaczysz możliwą do przeszukiwania listę wszystkiego, co tworzy bibliotekę standardową Swifta. Dostępna jest również pełna lista wszystkich typów Swifta, z której można wybierać interesujące Cię pozycje. Spójrz na typ `Array`, klikając łącze o tej samej nazwie. W ten sposób przejdziesz na stronę zawierającą dokumentację wybranego typu.

Strony dokumentacji w witrynie *http://swiftdoc.org/* są niezwykle użyteczne i zawierają naprawdę wiele informacji na temat różnych typów tworzących bibliotekę standardową oraz przykłady ich użycia. W tym miejscu przyjmuję założenie, że interesuje Cię sekcja **Inheritance** na stronie typu `Array`, jak pokazałem na rysunku 6.1.

Rysunek 6.1. Sekcja Inheritance na stronie dokumentacji typu Array

Jak możesz zobaczyć, typ `Array` jest zgodny z dziesięcioma protokołami, choć to tylko wierzchołek góry lodowej. Jeżeli klikniesz łącze widoku hierarchii protokołów, otrzymasz pełną hierarchię protokołów, z którymi jest zgodny typ `Array` (patrz rysunek 6.2).



Rysunek 6.2. Hierarchia protokołów typu Array

Dzięki dotychczas przedstawionemu materiałowi nie powinieneś mieć problemów z rozszyfrowaniem tego diagramu. Możesz natomiast nie wiedzieć, dlaczego został ułożony w taki właśnie sposób. W następnym rozdziale dowiesz się, jak projektować aplikacje i frameworki, wykorzystując podejście oparte na protokołach. Na końcu następnego rozdziału nieco dokładniej przedstawię hierarchię protokołów.

Podsumowanie

W tym rozdziale dowiedziałeś się, że protokoły to pełnoprawne typy w Swifcie. Zobaczyłeś również, jak polimorfizm w Swifcie może zostać zaimplementowany za pomocą protokołów. Następnie dość dokładnie omówiłem rozszerzenia protokołów i przykłady ich użycia w Swifcie.

Protokoły i ich rozszerzenia są podstawą stosowanego przez Apple nowego paradygmatu programowania zorientowanego na protokołach. Ten nowy model programowania ma potencjał do zmiany sposobu tworzenia kodu źródłowego. Wprawdzie ten rozdział nie został poświęcony temu nowemu paradygmatowi, ale przedstawiony tutaj materiał pozwolił na solidne poznanie podstaw dotyczących protokołów i ich rozszerzeń. Ta wiedza będzie niezbędna podczas poznawania tego nowego modelu programowania.

W następnym rozdziale pokażę, jak można używać protokołów i ich rozszerzeń podczas projektowania aplikacji.

Skorowidz

A

ABI, application binary interface, 18
adapter, 320
adres URL, 290
algorytm
  filter(), 76
  forEach(), 77
  map(), 77
  sort(), 76
  sorted(), 76
algorytmy dla tablic, 75, 233
ARC, 143
atrybut available, 191

B

bezpieczeństwo typu, 42
białe znaki, 35
biblioteka standardowa, 164
biblioteki, 289
blok case, 101
błędy, 184
  przechwytywanie, 187
  zgłaszanie, 185
budowniczy, 311

C

ciąg tekstowy, 48
cykl silnych odwołań, 145, 244

D

DateFormatter, 298
definiowanie zmiennych, 41
deklaracje kolekcji, 287
dekorator, 320
dołączanie wartości, 54
  typu opcjonalnego, 285, 286, 305
domknięcia, 227
  cykl silnych odwołań, 244
  samodzielne, 237
  składnia, 230
  wybór, 242
  z algorytmem tablicy, 233
  zmiana funkcjonalności, 239
dostęp
  do elementu tablicy, 70
  do kodu Objective-C, 259
  do kodu Swifta, 260
  do wartości słownika, 80
  otwarty, 132
  prywatny, 132
  prywatny dla pliku, 132
  publiczny, 132
  wewnętrzny, 132
dziedziczenie, 118, 133
  protokołu, 174

F

fabryka abstrakcyjna, 311
fasada, 320
FIFO, 265
FileManager, 301

filtrowanie danych, 101, 102
formatowanie kodu, 279
Formatter, 298
framework
  Cocoa, 24
  UIKit, 24
funkcje, 107, 282
  generyczne, 216
  języka, 19
  parametry wariadyczne, 113
  wartości domyślne parametrów, 109
  z pojedynczym parametrem, 107
  z wieloma parametrami, 109
  zewnętrzne nazwy parametrów, 112
  zwrot wartości typu opcjonalnego, 111
  zwrot wielu wartości, 110

G

Grand Central Dispatch, 265

H

hierarchia
  klas, 169
  protokołów, 164
HTTPURLResponse, 292

I

implementacja wzorca
  budowniczego, 315
  fasady, 325
  mostu, 321
  pełnomocnika, 328
  polecenia, 331
  singleton, 313
  strategii, 333
indeksy, 118, 193
  generyczne, 223
  nazwa zewnętrzna, 198
  niestandardowe, 195, 201
  obliczane, 197
  tablic, 194
  tylko do odczytu, 196
  wielowymiarowe, 198
inferencja typu, 20, 43, 286
inicjalizacja
  słownika, 79
  tablicy, 69
  zbioru, 82
interfejs binarny aplikacji, ABI, 18
IP, internet protocol, 114
iteracja przez tablicę, 78
iterator, 330

J

jawne określenie typu, 43
JSON, 304
JSONDecoder, 306
JSONEncoder, 305

K

katalog Resources, 26
klasa, 118, 180
  BlockOperation, 273
  DateFormatter, 298
  FileManager, 301
  Formatter, 298
  HTTPURLResponse, 292
  MessageBuilder, 259
  Messages, 258
  NumberFormatter, 300
  Operation, 276
  URLRequest, 292
  URLSession, 291
  URLSessionConfiguration, 291
klauzula where, 101, 179
kodowanie danych JSON, 304
kolejka
  FIFO, 265
  główna, 266, 271
  szeregowa, 266, 269
  współbieżna, 266, 268
kolekcje, 67
komentarze, 30, 283
kompozycja protokołu, 175
kompozyt, 320
konstrukcja
  for-case, 102
  guard, 93
  if, 91
  if-case, 104
  if-else, 92
  switch, 20, 97, 287
konstrukcje warunkowe, 91, 101
kontrola dostępu, 132
konwencja nazw, 282
krotka, 20, 86, 210

L

liczby
  całkowite, 44
  podwójnej precyzji, 46
  zmiennoprzecinkowe, 46

Ł

łańcuch zobowiązań, 330
łączenie
  kodu, 250
  wartości, 55

M

mechanizm ARC, 143
mediator, 330
metoda, 118, 126, 282
  addOperation(), 274
  async(), 271
  asyncAfter(), 272
  sync(), 271
metody
  dealokujące, 118
  inicjalizacyjne, 118, 128, 130
  nadpisywanie, 136
  wytwórcze, 311
modyfikowalne kolekcje, 20
modyfikowalność, 68
most, 320

N

nadpisywanie
  metody, 136
  właściwości, 137
natywna obsługa błędów, 184
nawiasy
  klamrowe, 33, 90
  okrągłe, 33, 90
nil, 213
NumberFormatter, 300

O

Objective-C, 249, 251
obserwator, 330
  właściwości, 125
obsługa błędów, 184
odwiedzający, 330
okno
  pliku typu playground, 21, 22
  powitalne Xcode, 22
operacje na zbiorze, 84
operator
  koalescencji nil, 213
  AND, 64
  NOT, 63
  OR, 64
  reszty z dzielenia, 62
  warunkowy trójargumentowy, 63
operatory
  arytmetyczne, 62
  logiczne, 63
  porównania, 61
  przypisania, 34, 63

P

pamiątka, 330
pamięć, 143
panel nawigacyjny, 25
para klucz-wartość, 81
parametr inout, 114
parametry wariadyczne, 113
pełnomocnik, 320
pętla
  for-in, 84, 94
  repeat-while, 97
  while, 96
plik Bridging Header, 255, 259
pliki playground, 21, 29
  tworzenie wykresu, 28
  typ iOS, 24
  typ tvOS, 24
  wyświetlanie obrazu, 25
polecenie, 330
  break, 105
  continue, 105
  fallthrough, 106
polimorfizm, 154
poziomy dostępu, 132
programowanie
  równoległe, 263
  zorientowane na protokoły, 20, 151, 167, 174
  zorientowane obiektowo, 151, 168
projekt Objective-C, 251
  dodawanie pliku, 253, 256
  dostęp do kodu, 259
  klasa Messages, 258
  plik Bridging Header, 255, 259

protokoły, 20, 138, 151
  dziedziczenie, 174
  jako typ danych, 152
  klauzula where, 179
  kompozycja, 175, 179
  polimorfizm, 154
  projekt, 174
  rozszerzenie, 156
  składnia, 139
  typu Array, 164
  wymagania metody, 140
  wymagania właściwości, 139
protokół IP, 114
prototyp, 312
przeciążanie operatorów, 20
przekazywanie
  przez referencję, 119
  przez wartość, 119
pyłek, 320

R

REPL, read-evaluate-print-loop, 21
REST, 292
rozszerzenia, 118, 142
  protokołów, 151, 156
równoległość, 264
rzutowanie typu, 154

S

separator, 36
singleton, 312
składnia
  domknięcia, 20, 230
  języka, 29
  protokołu, 139
słownik, 79
  dostęp, 80
  inicjalizacja, 79
  para klucz-wartość, 80–82
słowo kluczowe self, 284
stałe, 40, 282, 285
stan, 330
sterowanie przebiegiem działania programu, 91
strategia, 330
struktura, 118, 180
  tablicy, 181
styl programowania, 280, 281
superklasa, 169
Swift, 18
system plików, 301

Ś

średnik, 32, 281

T

tablica, 68, 181
  algorytmy, 75
  dodawanie elementu, 72
  dostęp do elementu, 70
  indeksy, 194
  inicjalizacja, 69
  łączenie tablic, 74
  pobranie podtablicy, 74
  tworzenie, 69
  usunięcie elementu, 73
  wprowadzenie wielu zmian, 75
  wstawienie wartości, 73
  zastępowanie elementu, 73
  zliczanie elementów, 71
terminator, 36
trójargumentowy operator warunkowy, 63
tworzenie
  cyklu silnych odwołań, 244
  klasy, 120
  kolejki, 267
  niestandardowego indeksu, 195
  słownika, 79
  struktury, 120
  tworzenie wykresu, 28
typ, 118
  DoCalculations, 266
  JSONDecoder, 306
  JSONEncoder, 305
  Operation, 272
  OperationQueue, 272
  pliku playground, 24
typy
  generyczne, 20, 215, 220
  kolekcji, 67
  liczbowe, 44
  opcjonalne, 20, 203, 210, 285
  powiązane, 224
  własne, 282
  wyliczeniowe, 20, 57

U

URL, 290, 292
URLRequest, 292
URLSession, 291
URLSessionConfiguration, 291

URLSessionTask, 291
usługa sieciowa typu REST, 292
usprawnienia, 20
użycie
  inferencji typu, 286
  JSONDecoder, 306
  konstrukcji switch, 287
  nawiasu, 281
  skróconych deklaracji kolekcji, 287
  średnika, 281

W

wartości
  boolowskie, 48
  typu opcjonalnego, 206–211
  indeksu, 197
wcięcia, 283
wczytywanie adresów URL, 290
właściwości
  nadpisywanie, 137
  obliczane, 120, 122
  przechowywane, 120
współbieżność, 264
wykres
  tworzenie, 28
  wyświetlanie, 28
wyświetlanie
  obrazu, 25
  wykresu, 28
wzorce
  konstrukcyjne, 311
  operacyjne, 330
  projektowe, 309
  strukturalne, 320
wzorzec
  budowniczego, 315
  fasady, 324
  mostu, 320
  pełnomocnika, 327
  polecenia, 330
  singleton, 312
  strategii, 333

X

Xcode
  okno powitalne, 22
  opcje menu, 23

Z

zapis pliku, 254
zarządzanie pamięcią, 143
zbiór, 82
  inicjalizacja, 82
  określenie liczby elementów, 83
  operacje, 84
  usuwanie elementów, 84
  wstawianie elementów, 83
zmienne, 40, 282, 285
  typu opcjonalnego, 52

Ż

żądanie
  GET, 293
  POST, 296

Swift 4: programuj po mistrzowsku!

Historia Swifta rozpoczęła się w 2014 roku. Dziś jest najważniejszym językiem programowania dla platform macOS i iOS. Charakteryzuje się zwięzłą i przejrzystą składnią, jest przy tym wygodny i elastyczny, a jego nauka nie powinna sprawiać problemów nawet początkującym deweloperom. Od chwili jego powstania Apple co roku przedstawia nowe wydanie tego języka. Na konferencji WWDC w 2017 roku zaprezentowano wersję numer 4, w której wprowadzono sporo bardzo interesujących zmian. Każdy, kto chce pisać efektywne i bezpieczne aplikacje dla macOS i iOS, koniecznie powinien się z nimi zapoznać!

Ta książka jest praktycznym podręcznikiem efektywnego programowania w języku Swift 4. Znajdziesz tu wyjaśnienie jego podstaw, następnie poznasz nowe funkcje i nauczysz się z nich korzystać podczas tworzenia aplikacji. Poza dość zasadniczymi kwestiami przedstawiono tu również zagadnienia zaawansowane, takie jak łączenie w projekcie kodu Objective-C i Swift, wykorzystanie mechanizmu ARC, używanie domknięć i zastosowanie programowania równoległego. Bardzo ciekawymi tematami poruszonymi w książce są rozszerzenia protokołów, obsługa błędów, stosowanie wzorców projektowych i współbieżności. Poznasz potężne możliwości programowania zorientowanego na protokoły. Szybko nauczysz się pisać elastyczny i łatwy w zarządzaniu kod.

W tej książce między innymi:

- składnia i elementy języka Swift
- kontrola przepływu działania programu
- tworzenie bezpiecznego kodu i obsługa błędów
- typy opcjonalne, typy generyczne i domknięcia
- zasady pisania eleganckiego i czytelnego kodu
- podstawowe biblioteki Swifta i wzorce projektowe

Jon Hoffman jest wyjątkowo doświadczonym twórcą oprogramowania dla platformy iOS. Pracował też jako administrator systemu i sieci oraz administrator bezpieczeństwa. Napisał sporo aplikacji mobilnych dla Androida i Windowsa. Uwielbia wyzwania programistyczne, projekty z zakresu robotyki i druku 3D. Od kilku lat wraz z żoną i córkami dzieli inną pasję: taekwondo.

Helion
helion.pl
HELION SA
ul. Kościuszki 1c
44-100 Gliwice
tel.: 32 230 98 63
helion@helion.pl

KOD KORZYŚCI
Sięgnij po więcej!
ISBN 978-83-283-4794-6
9 788328 347946
INFORMATYKA W NAJLEPSZYM WYDANIU
Cena: 67,00 zł
Packt