Mark A. Lassoff
& Tom Stachowitz

Podstawy języka Swift

Programowanie aplikacji dla **platformy iOS**

learntoprogram.tv

Helion

Tytuł oryginału: Swift Fundamentals: The Language of iOS Development

Tłumaczenie: Robert Górczyński
Projekt okładki: Studio Gravite / Olsztyn
Obarek, Pokoński, Pazdrijowski, Zaprucki

ISBN: 978-83-283-1912-7

Wydawnictwo HELION
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)

Pliki z przykładami omawianymi w książce można znaleźć pod adresem:
ftp://ftp.helion.pl/przyklady/pjswif.zip

SPIS TREŚCI

Rozdział 7.

Więcej konstrukcji języka Swift

W rozdziale:

- Dowiesz się, jak używać rozszerzeń w celu zwiększenia użyteczności klas i typów danych.
- Poznasz mechanizm przeciążania operatorów i przekonasz się, że to narzędzie, które można wykorzystać do bardzo wielu celów.
- Nauczysz się wykorzystywać funkcje generyczne i zobaczysz, że ich stosowanie może ograniczyć konieczność powielania kodu i zapewnić mu większą czytelność.
- Dowiesz się, jak można używać emotikonów w języku Swift.

7.1. Rozszerzenia

Jak dotąd używaliśmy elementów wbudowanych w języku Swift lub też tworzyliśmy własne konstrukcje. Na przykład za każdym razem, gdy korzystaliśmy ze zmiennej w postaci liczby rzeczywistej, stosowaliśmy wbudowany typ `Int`. W przypadku własnych konstrukcji, takich jak klasy, prototypy i typy wyliczeniowe, mieliśmy możliwość zdefiniowania wymaganej funkcjonalności. Jednak w trakcie programowania zdarzają się sytuacje, w których nie ma możliwości bezpośredniej modyfikacji pierwotnego kodu lub też wbudowane elementy mają zostać użyte do wykonania znacznie poważniejszych zadań niż te, do których zostały zaprojektowane.

Podczas zwiększania użyteczności istniejącego fragmentu kodu mówimy o rozszerzeniu jego możliwości poza wbudowane. Dokładnie tym są **rozszerzenia** w języku Swift: to metody egzemplarza dodane do istniejących klas, struktur lub typów wyliczeniowych.

Aby zademonstrować rozszerzenie, rozbudujemy jeden z podstawowych typów danych i tym samym zwiększymy jego funkcjonalność. W Xcode utwórz nowy plik typu *playground*, po czym umieść w nim następujący fragment kodu:

```
extension Double
{
   // Miejsce na kod rozszerzający funkcjonalność.
}
```

W powyższym fragmencie kodu użyliśmy nowego słowa kluczowego **`extension`** do utworzenia metody egzemplarza, która będzie dodana do klasy `Double`. Nową metodę egzemplarza możemy wywołać, ponieważ istnieje jedynie w programie, w którym została zdefiniowana.

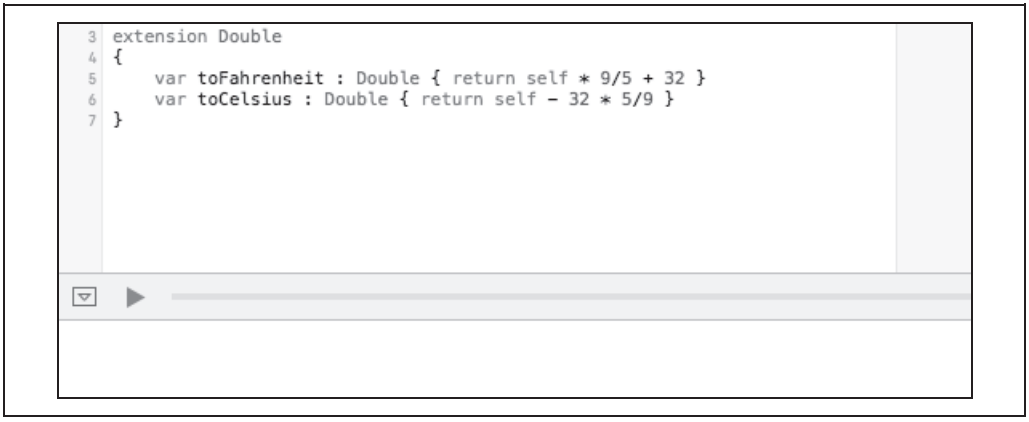
Rozszerzenie

Właściwości obliczane

W naszym rozszerzeniu zdefiniujemy kilka nowych właściwości typu `Double` za pomocą techniki, która nie została jeszcze omówiona. Do pobrania wartości wspomnianych właściwości wykorzystamy **właściwości obliczane**. Właściwości obliczane nie przechowują wartości, ale zwracają wynik obliczeń. Kod przeznaczony do ustawiania i pobierania wartości właściwości obliczanych nosi nazwę odpowiednio *setter* i *getter*. W omawianym przykładzie użyjemy *właściwości obliczanej przeznaczonej jedynie do odczytu*. Te właściwości mogą być tylko odczytywane, nie można przypisać im wartości i mają jedynie blok *getter*. Dokładne wyjaśnienie właściwości obliczanych wykracza poza zakres tematyczny tej książki, ale wyczerpującą dokumentację na ten temat znajdziesz w witrynie Apple Developer.

W przygotowanym szkielecie rozszerzenia usuń komentarz, a następnie wprowadź przedstawiony poniżej fragment kodu (patrz rysunek 7.1):

```
var toFahrenheit : Double { return self * 9/5 + 32 }
var toCelsius : Double { return self – 32 * 5/9 }
```



Rysunek 7.1. Rozszerzenie typu danych Double za pomocą właściwości obliczanych

W powyższym kodzie znajduje się wiele znanych Ci już elementów, choć jednocześnie pojawiły się nowe idee. Zadeklarowane zostały właściwości, mają przypisane nazwy oraz jasno określone typy. Jednak każda właściwość zamiast wartości ma polecenie `return` wraz z blokiem kodu. To blok kodu typu *getter* odpowiedzialny za obliczenie wartości właściwości.

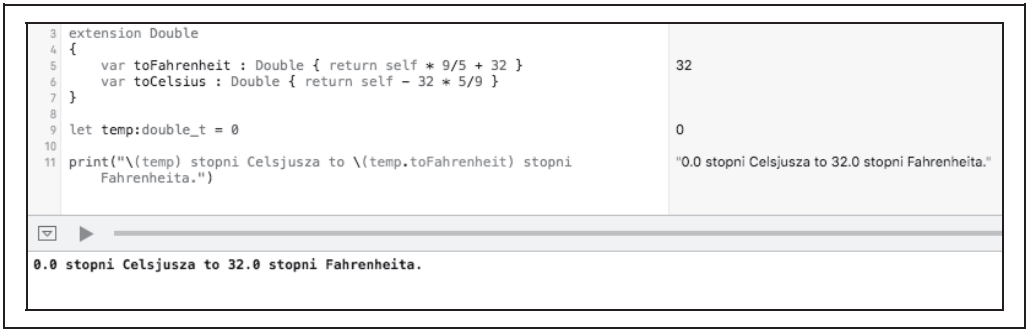
self

W omawianym fragmencie kodu właściwości `toFahrenheit` przypisywana jest wartość zwrócona przez wyrażenie znajdujące się w bloku kodu. Słowo kluczowe **`self`** odwołuje się do wartości `Double` wywołującej właściwość `toFahrenheit` i jest używane w wyrażeniu w celu obliczenia temperatury w skali Fahrenheita. Następnie tej samej składni używamy do obliczenia temperatury Fahrenheita w skali Celsjusza.

Po zdefiniowaniu rozszerzeń warto uzyskać do nich dostęp, aby upewnić się, że działają prawidłowo. Rozpoczynamy od znanej wartości, czyli zera stopni w skali Celsjusza. Po kodzie rozszerzenia wprowadź następujący fragment kodu:

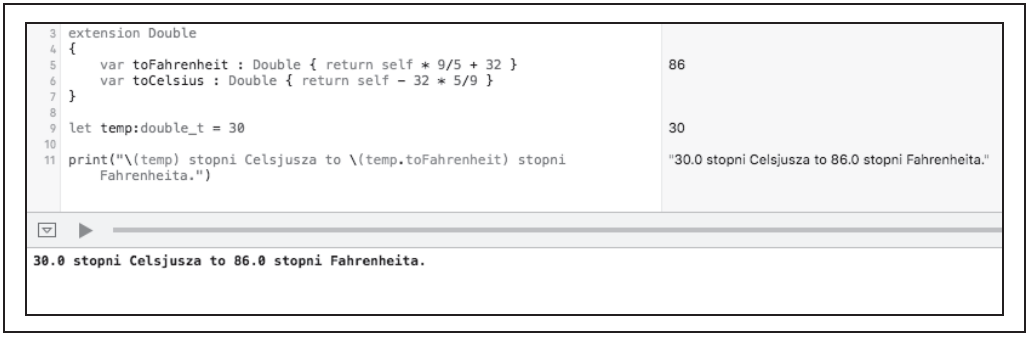
```
let temp:double_t = 0
print("\(temp) stopni Celsjusza to \(temp.toFahrenheit) stopni Fahrenheita.")
```

Powyżej utworzyliśmy stałą typu `Double`. Następnie zastosowaliśmy ją w interpolacji ciągu tekstowego w celu wyświetlenia wartości tej stałej typu `Double` o właściwości obliczonej `Double.toFahrenheit()`, za pomocą której rozszerzyliśmy typ danych `Double`. Otrzymane dane wyjściowe pokazałem na rysunku 7.2.



Rysunek 7.2. Przygotowana przez nas właściwość obliczona zwraca po wywołaniu odpowiedni wynik

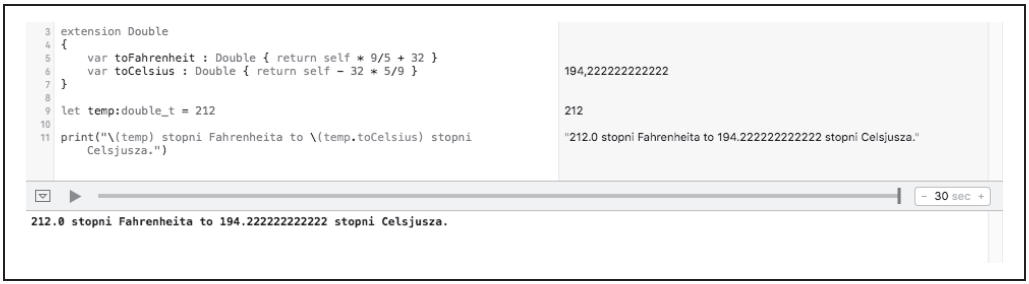
Zmień wartość `temp` na 30, a następnie spójrz na wygenerowane dane wyjściowe (patrz rysunek 7.3).



Rysunek 7.3. Weryfikacja implementacji właściwości obliczanej

Ponownie zmień wartość `temp`, tym razem na `212`, i zmodyfikuj kod, aby wprowadzona wartość została przekonwertowana na temperaturę w skali Celsjusza. Powinieneś otrzymać wynik pokazany na rysunku 7.4.

```
print("\(temp) stopni Fahrenheita to \(temp.toCelsius) stopni Celsjusza.")
```



Rysunek 7.4. Za pomocą rozszerzenia typu danych Double przeprowadzamy konwersję temperatury w skali Fahrenheita na temperaturę w skali Celsjusza

Jak możesz zobaczyć w powyższych fragmentach kodu, rozszerzenia pozwalają na implementację pewnych nowych metod i właściwości w istniejących typach danych oraz klasach projektu. W następnym podrozdziale przeanalizujemy podobną technikę zwiększania funkcjonalności operatorów.

PYTANIA KONTROLNE

1) Co to jest rozszerzenie?
   a) Konstrukcja pozwalająca na dodanie funkcjonalności do kodu istniejących elementów.
   b) Sposób na zwiększenie precyzji typów danych `Float` i `Double`.
   c) Sposób na zapewnienie funkcji dodatkowego czasu potrzebnego na zwrócenie wartości.
   d) Metoda modyfikacji istniejących klas.

2) Jak przedstawia się poprawna składnia rozszerzenia?
   a) `func:extension Double`.
   b) `extension Double`.
   c) `extension T:Double`.
   d) `extension (Double)`.

3) Rozszerzenie spowoduje modyfikację pierwotnego kodu w rozszerzanym elemencie.
   a) Prawda.
   b) Fałsz.

4) Na co pozwalają nam właściwości obliczane?
   a) Na użycie predefiniowanych wartości domyślnych.
   b) Na użycie przekazanej wartości.
   c) Na przypisanie wartości zgodnej z obliczanym blokiem kodu.
   d) Na żadne z powyższych.

7.2. PRZECIĄŻANIE OPERATORA

W poprzednim podrozdziale dowiedziałeś się, jak używać rozszerzeń w celu dodania funkcjonalności do już istniejących klas lub typów danych. W tym podrozdziale zobaczysz, jak uzyskać podobny efekt, ale w przypadku operatorów. Wykorzystamy do tego tak zwane **przeciążanie operatora**. Jest to technika dodawania funkcjonalności do operatorów i zwiększania ich możliwości zwłaszcza podczas pracy z wygenerowanymi przez użytkownika typami wyliczeniowymi, klasami i strukturami.

Przeciążanie operatora

Struktura

O **strukturach** pokrótce wspomniałem już we wcześniejszej części książki. Struktury w języku Swift są przeznaczone do hermetyzacji niewielkich zbiorów prostych wartości. Struktury można porównać do klas, ponieważ mogą zawierać właściwości i metody. W przedstawionym poniżej kodzie utworzymy strukturę przeznaczoną do przechowywania danych punktów niezbędnych do utworzenia trójwymiarowego wektora, następnie utworzymy dwa wektory i na koniec spróbujemy je dodać za pomocą standardowego operatora dodawania. Zdefiniowanie trójwymiarowego wektora wymaga trzech współrzędnych x, y i z, dodawanie dwóch wektorów odbywa się natomiast przez dodanie ich odpowiednich współrzędnych x, y i z. Utwórz nowy plik typu *playground* w Xcode, a następnie wprowadź w nim przedstawiony poniżej fragment kodu.

```
struct Vector
{
  var x = 0
  var y = 0
  var z = 0
}

var v1 = Vector(x:5, y:8, z:5)
var v2 = Vector(x:9, y:4, z:2)

print(v1+v2)
```

W powyższym kodzie za pomocą słowa kluczowego **`struct`** utworzyliśmy strukturę. Składnia w definicji struktury jest podobna do stosowanej w klasach. Deklaracje trzech właściwości definiujących wektor zostały przygotowane za pomocą słowa kluczowego `var`. Ponieważ wszystkie właściwości mają przypisane wartości początkowe, nie ma konieczności użycia metody inicjalizującej.

Struktura

Po definicji struktury utworzyliśmy dwie zmienne typu `Vector`, a następnie przypisaliśmy im zdefiniowane wcześniej właściwości. Kolejnym krokiem jest próba wyświetlenia wyniku operacji dodania obu wektorów. Wynik wykonania powyższego kodu pokazałem na rysunku 7.5.

Rysunek 7.5. Standardowy operator dodawania nie jest w stanie dodać przygotowanych przez nas struktur Vector

Wygenerowany przez powyższy fragment kodu komunikat błędu informuje, że Swift nie potrafi dodać dwóch struktur `Vector`. W celu zdefiniowania funkcjonalności pozwalającej na dodawanie struktur `Vector` konieczne jest przeciążenie operatora dodawania. W pliku typu *playground* po definicji struktury wprowadź poniższy fragment kodu:

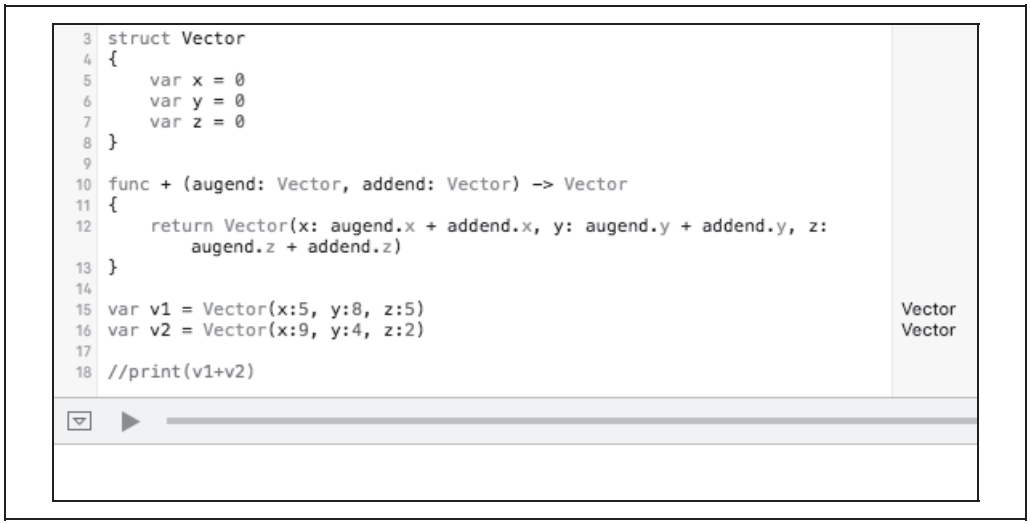
```
func + (augend: Vector, addend: Vector) -> Vector
{
  return Vector(x: augend.x + addend.x, y: augend.y + addend.y, z: augend.z + addend.z)
}
```

W powyższym kodzie użyliśmy słowa kluczowego `func` oraz operatora przeznaczonego do przeciążenia. Następnie zdefiniowaliśmy argumenty przekazywane do przeciążenia (dwie zdefiniowane wcześniej struktury `Vector`) i na końcu wskazaliśmy, że wartość zwrotna ma być typu `Vector`. Wewnątrz bloku kodu wskazujemy językowi Swift, jak przedstawia się wartość zwrotna typu `Vector`. W omawianym przykładzie zdefiniowaliśmy, że zwracane wartości `x`, `y` i `z` typu `Vector` mają być sumami wartości `x`, `y` i `z` przekazanych danych typu `Vector`. Kod powinien wyglądać tak, jak pokazałem na rysunku 7.6.

Skoro wskazaliśmy operatorowi dodawania, jak obsługiwać struktury typu `Vector`, za pomocą poniższego fragmentu kodu możemy przetestować przeciążony operator:

```
var v3 = v1 + v2
print("x:\(v3.x) y:\(v3.y) z:\(v3.z)")
```

Utworzonej zmiennej `v3` przypisujemy sumę `v1` i `v2`, a dodawanie jest przeprowadzane za pomocą zdefiniowanego przeciążonego operatora. Podczas wyświetlania poszczególnych właściwości `v3` powinieneś zobaczyć, że zostały poprawne obliczone zgodnie z wyrażeniem dodawania `Vector`. Otrzymane dane wyjściowe porównaj z pokazanymi na rysunku 7.7.



Rysunek 7.6. Przeciążenie operatora pozwala na dodanie funkcjonalności do operatorów w języku Swift

Rysunek 7.7. Po utworzeniu przeciążonego operatora dla struktury Vector język Swift zyskuje możliwość przeprowadzania operacji dodawania dwóch wektorów

Przeciążanie operatorów pozwala na większą elastyczność podczas tworzenia aplikacji, ponieważ zyskujesz możliwość wykonania większej liczby zadań za pomocą standardowych operatorów. Tym samym struktury, klasy, typy wyliczeniowe oraz inne niestandardowe typy stają się jeszcze bardziej wszechstronne.

PYTANIA KONTROLNE

1) Przeciążanie operatora pozwala na trwałe modyfikowanie operatorów w języku Swift.
   a) Prawda.
   b) Fałsz.

2) Jak przedstawia się poprawna składnia przeciążania operatora?
   a) `func - (myVarA: Int, myVarB: Int) -> Int.`
   b) `overload func - (myVarA: Int, myVarB: Int) -> Int.`
   c) `func:overload - (myVarA: Int, myVarB: Int) -> Int.`
   d) `func - (myVarA, myVarB) -> Int.`

3) Do czego przede wszystkim służą struktury?
   a) Do zdefiniowania abstrakcyjnych reprezentacji obiektów.
   b) Do przechowywania podobnych grup elementów w sposób zapewniający bezpieczeństwo typu.
   c) Do hermetyzacji małych zbiorów prostych wartości.
   d) Są alternatywą dla klas.

4) Struktury mogą zawierać właściwości i metody.
   a) Prawda.
   b) Fałsz.

7.3. FUNKCJE GENERYCZNE

Podczas tworzenia funkcji i metod wyraźnie deklarujemy typy przekazywanych im argumentów. Jednym z celów efektywnego programowania jest uniknięcie powielania tego samego kodu. Jednak czasami w trakcie pracy z różnymi typami danych powielenie kodu wydaje się nieuniknione. Kiedy potrzebujesz funkcji przeprowadzającej te same obliczenia na różnych typach danych, wówczas tworzysz wiele funkcji różniących się zaledwie nazwą i składnią definicji. Problem związany z powtarzaniem tego samego kodu można rozwiązać za pomocą funkcji generycznych.

Funkcje generyczne

Funkcje generyczne w języku Swift to funkcje, które mogą działać ze wszystkimi zgodnymi z nimi typami danych. W ten sposób pozwalają na uniknięcie powielania kodu i jednocześnie zwiększają jego czytelność. Rozważ sytuację, gdy potrzebna jest funkcja sprawdzająca równość dwóch argumentów i zwracająca wartość typu boolowskiego. Kod funkcji będzie dokładnie taki sam, niezależnie od sprawdzanego typu danych. Jedyna różnica będzie występowała w deklaracji funkcji. W nowym pliku typu *playground* wprowadź przedstawiony poniżej fragment kodu:

```
func isEqualInt(a:Int, b:Int) -> Bool
{
  return a == b
}

print(isEqualInt(3, b: 3))
print(isEqualInt(3.4, b: 3.8))
```

W powyższym fragmencie kodu sprawdzamy, czy dwie wartości typu `Int` są takie same. Dwa wywołania `print()` pokazują wyniki użycia funkcji `isEqualInt()` najpierw na wartościach typu `Int`, a później `Double`. Wynik wykonania tego fragmentu kodu pokazałem na rysunku 7.8.

Rysunek 7.8. Przygotowana funkcja isEqualInt() działa jedynie z wartościami typu Int i generuje błąd w przypadku otrzymania wartości typu Double. Dla obu wymienionych typów danych kod funkcji będzie dokładnie taki sam

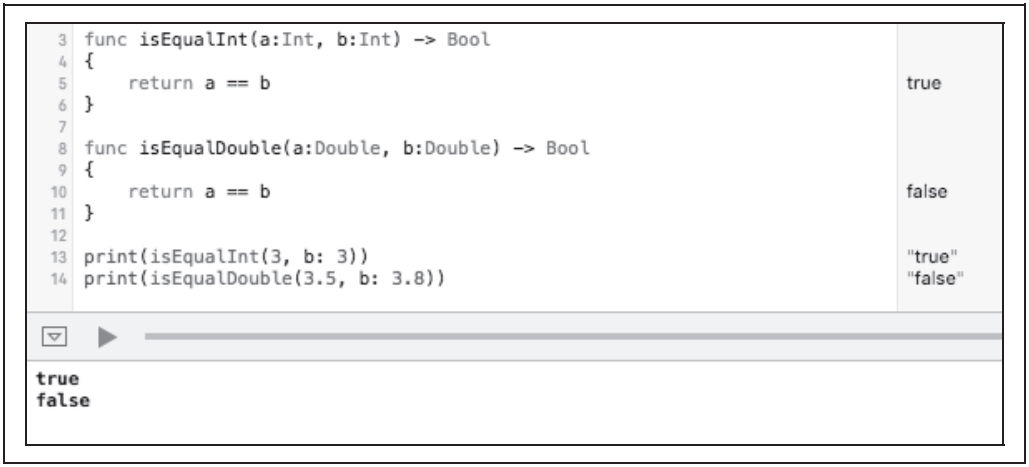
Funkcja zwraca poprawny wynik dla wartości typu `Int`, a dla wartości typu `Double`, jak pokazałem na rysunku 7.8, generuje komunikat błędu. Aby uzyskać tę samą funkcjonalność dla typu `Double`, konieczne jest dodanie kolejnej funkcji:

```
func isEqualDouble(a:Double, b:Double) -> Bool
{
  return a == b
}

print(isEqualInt(3, b: 3))
print(isEqualDouble(3.5, b: 3.8))
```

Po wprowadzeniu kodu drugiej funkcji otrzymasz wynik pokazany na rysunku 7.9.

Jak możesz zobaczyć, otrzymaliśmy poprawny wynik dla wartości typu `Double`. Niestety odbyło się to kosztem utworzenia zupełnie nowej funkcji w większości składającej się z tego samego kodu, który znajduje się w pierwszej funkcji. Aby rozwiązać problem związany z powielaniem kodu, możemy użyć funkcji generycznej pobierającej argument dowolnego typu danych. Utwórz funkcję generyczną o następującym kodzie:



Rysunek 7.9. Utworzenie nowej funkcji dla typu danych Double pozwala na zwrócenie poprawnej wartości, choć jednocześnie oznacza konieczność wprowadzenia dwukrotnie większej ilości kodu

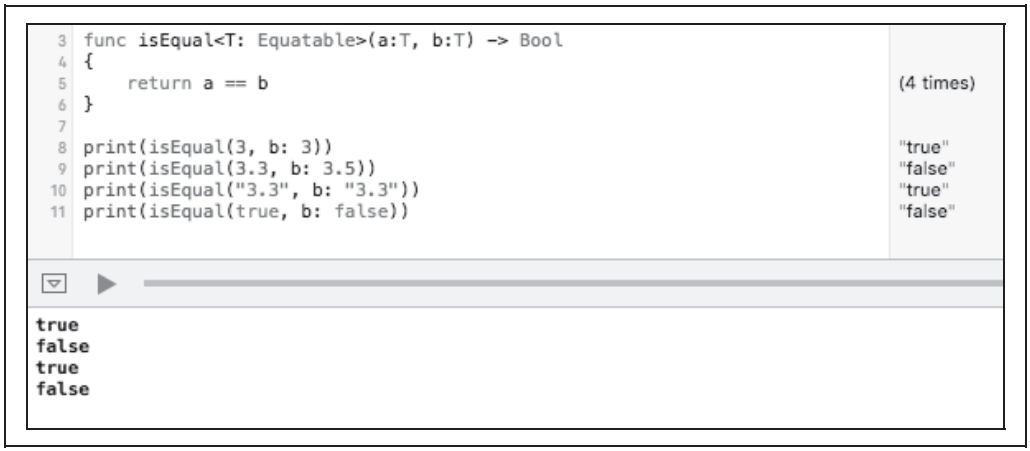
```
func isEqual<T: Equatable>(a:T, b:T) -> Bool
{
  return a == b
}

print(isEqual(3, b: 3))
print(isEqual(3.3, b: 3.5))
print(isEqual("3.3", b: "3.3"))
print(isEqual(true, b: false))
```

W powyższym fragmencie kodu po nazwie funkcji zdefiniowaliśmy, że typ argumentów funkcji to `T`, o ile są równe (`Equatable`). Słowo kluczowe `Equatable` oznacza, że wszystkie argumenty są tego samego typu. Następnie w definicji argumentów zadeklarowaliśmy `T` jako typ wszystkich argumentów. Pozostała część funkcji jest dokładnie taka sama jak w przedstawionych wcześniej przykładach. Oczekiwane wyniki działania nowej wersji funkcji `isEqual()` pokazałem na rysunku 7.10.

W wywołaniach `print()` przetestowaliśmy funkcję wraz z wartościami typów `Int`, `Double`, `String` i `Boolean`. We wszystkich przypadkach funkcja zwraca prawidłowy wynik, co eliminuje konieczność czterokrotnego tworzenia tego samego kodu.

W ostatnim podrozdziale zajmiemy się jednym z zabawniejszych aspektów języka Swift, czyli użyciem emotikonów w kodzie.



Rysunek 7.10. Funkcje generyczne pozwalają na wyeliminowanie znacznej ilości powielanego kodu

PYTANIA KONTROLNE

1) Funkcje generyczne pozwalają na uniknięcie powielania kodu.
   a) Prawda.
   b) Fałsz.

2) Jak przedstawia się prawidłowa składnia funkcji generycznej?
   a) `func myFunc (myVarA: T, myVarB: T).`
   b) `func myFunc:<T> (myVarA: T, myVarB: T).`
   c) `func myFunc<T: Equatable> (myVarA: T, myVarB: T).`
   d) `generic myFunc (myVarA, myVarB).`

3) Funkcje generyczne to dobry wybór, gdy trzeba znać dokładny typ argumentów w funkcji.
   a) Prawda.
   b) Fałsz.

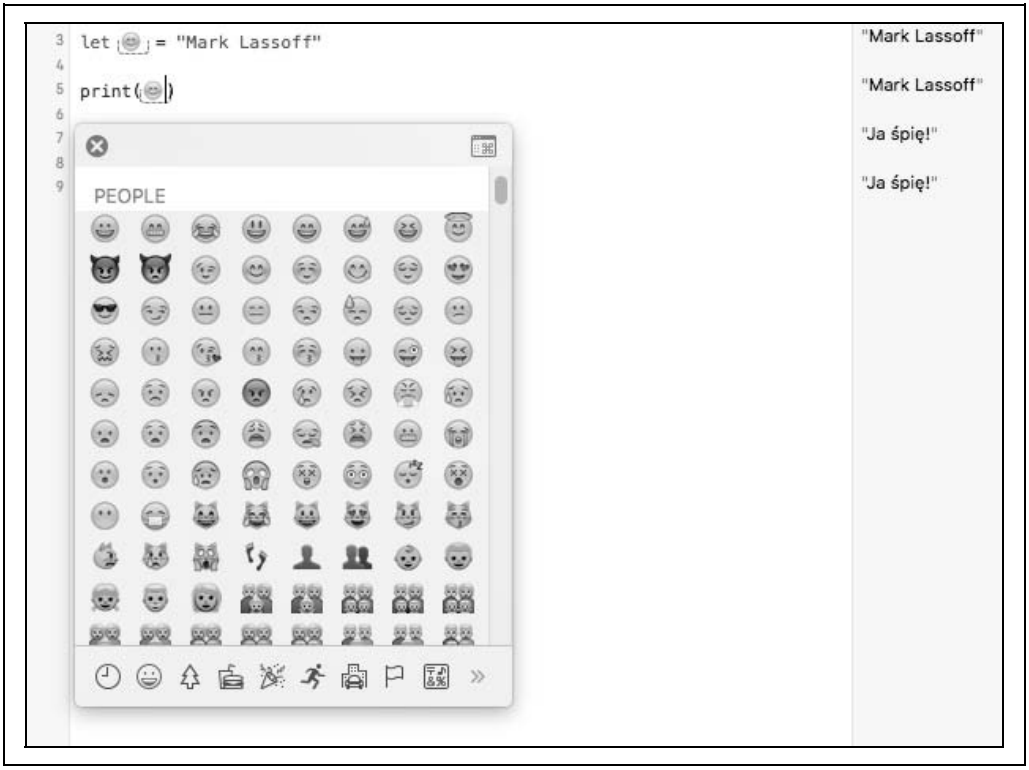
4) Które z poniższych zdań NIE przedstawia zalety funkcji generycznych?
   a) Mniejsza ilość powielonego kodu.
   b) Większa czytelność kodu.
   c) Łatwiejszy proces usuwania błędów.
   d) Dodanie funkcjonalności do istniejących klas.

7.4. EMOTIKONY

Emotikony (ang. *emoji*) to małe obrazy przeznaczone do przedstawiania idei lub koncepcji. Pierwotnie były stosowane w japońskich komunikatach tekstowych. Nazwa pochodzi od japońskiego *e* oznaczającego obraz i *moji* oznaczającego znak. Ostatnio w standardzie Unicode

wprowadzono setki emotikonów, co pozwala na ich zastosowanie w wielu różnych miejscach. W języku Swift emotikonów można używać dla nazw zmiennych, klas, typów wyliczeniowych, struktur oraz innych konstrukcji.

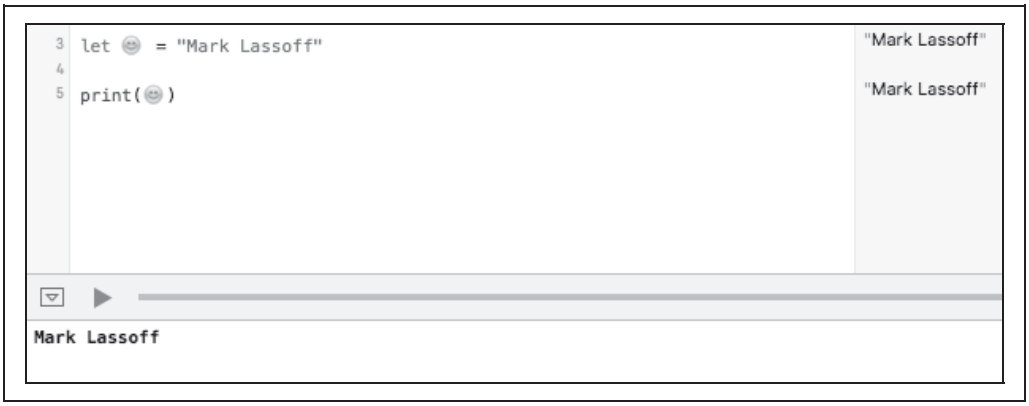
W Xcode utwórz nowy plik typu *playground*, a następnie zadeklaruj w nim stałą przeznaczoną do przechowywania imienia i nazwiska. Zamiast nazwy tekstowej dla stałej użyj emotikona. Aby uzyskać dostęp do menu emotikonów w języku Swift, naciśnij kombinację klawiszy *Spacja+Ctrl+Command*. Na ekranie zostanie wyświetlone menu (patrz rysunek 7.11), w którym możesz wybrać emotikon do użycia.



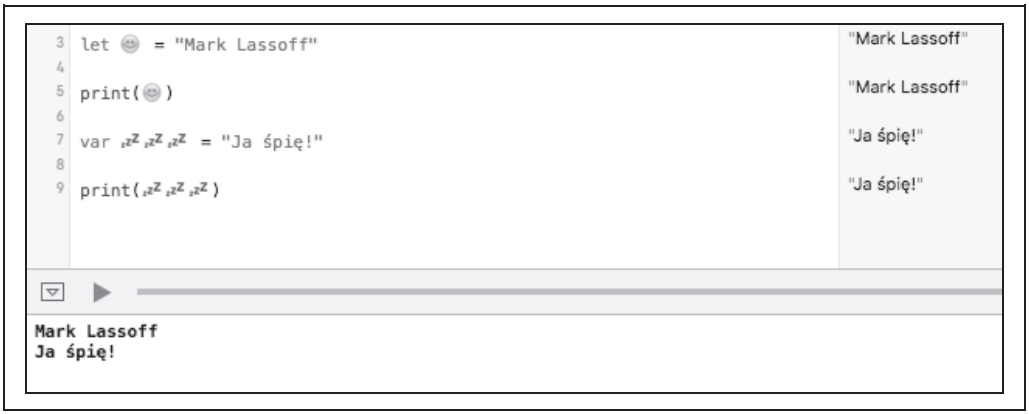
Rysunek 7.11. Menu emotikonów w języku Swift

Po wyborze emotikona język Swift uznaje go za całkowicie poprawną nazwę zmiennej. W omawianym przykładzie zdecydowałem się na uśmiechniętą buźkę jako nazwę stałej. Następnie za pomocą wywołania `print()` wyświetliliśmy wartość tej stałej (patrz rysunek 7.12).

Istnieje również możliwość użycia wielu emotikonów do przedstawienia elementu. Na rysunku 7.13 możesz zobaczyć, jak wykorzystałem trzy emotikony do przedstawienia ciągu tekstowego `Ja Śpię!`.



Rysunek 7.12. Swift współpracuje z emotikonami tak jak z innymi nazwami elementów



Rysunek 7.13. Użycie wielu emotikonów do przedstawienia zmiennej

Emotikony to niewielki, uprzyjemniający pracę dodatek do języka programowania Swift, ale nie stanowią one najbardziej efektywnych nazw dla elementów. Mimo wszystko mogą dostarczyć nieco radości podczas tworzenia oprogramowania w języku Swift.

PYTANIA KONTROLNE

1) Nazwa emotikony (*emoji*) pochodzi z języka japońskiego, w którym *e* oznacza obraz, a *moji* oznacza znak.
   a) Prawda.
   b) Fałsz.

2) W nazwach których elementów można używać emotikonów? Wybierz wszystkie poprawne odpowiedzi.

a) W nazwach zmiennych.
b) W nazwach stałych.
c) W nazwach klas.
d) W nazwach struktur.

3) Intensywne użycie emotikonów prowadzi do uzyskania czytelnego, łatwego w konserwacji kodu.
a) Prawda.
b) Fałsz.

4) Za pomocą jakiej kombinacji klawiszy można uzyskać dostęp do menu emotikonów?
a) *Ctrl+Alt+Delete*.
b) *Ctrl+Command+Enter*.
c) *Ctrl+Command+Space*.
d) *Ctrl+Command+Alt*.

ĆWICZENIA DO ROZDZIAŁU 7.

1) Utwórz nowy plik typu *playground* o nazwie *SLA_Lab7*. W edytorze tekstów usuń cały niepotrzebny kod i pozostaw jedynie niezbędne polecenie `import`.

2) Utwórz wielowierszowy komentarz z Twoim imieniem, bieżącą datą oraz nazwą ćwiczenia (`SLA_Lab7`).

3) W pliku typu *playground* wprowadź następujący fragment kodu:

```
import UIKit

func feetToInchesInt(feet:Int) -> Double
{
   return Double(feet*12)
}

func feetToInchesDouble(feet:Double) -> Double
{
   return feet*12
}

func feetToInchesFloat(feet:Float) -> Float
{
   return feet*12
}

func InchesToFeetInt(inches:Int) -> Double
{
   return Double(inches/12)
}

func InchestToFeetDouble(inches:Double) -> Double
{
```

```
    return inches/12
}

func InchestToFeetFloat(inches:Double) -> Double
{
    return inches/12
}

print(feetToInchesInt(177))
print(feetToInchesFloat(164.55))
print(InchesToFeetInt(1000))
```

1) Wykorzystując zdobytą w rozdziale wiedzę o funkcjach generycznych, funkcjonalność wszystkich przedstawionych powyżej metod hermetyzuj w dwóch metodach generycznych. Zmień wywołania funkcji na końcu kodu w taki sposób, aby były używane opracowane przez Ciebie metody generyczne.

PODSUMOWANIE

W tym rozdziale poznałeś kolejne użyteczne funkcje języka programowania Swift. Dowiedziałeś się, jak rozszerzenia pozwalają na zwiększenie użyteczności wbudowanych elementów i klas. Przeanalizowaliśmy temat przeciążania operatorów, aby stały się jeszcze bardziej użyteczne i lepiej dopasowane do budowanych programów. Wykorzystaliśmy funkcje generyczne do utworzenia funkcji, które mogą obsługiwać dowolne typy danych, co umożliwia wyeliminowanie powielania kodu oraz pozwala na tworzenie jeszcze bardziej czytelnego kodu źródłowego. Na końcu rozdziału spojrzeliśmy na emotikony, które dodają nieco uśmiechu do tworzonych aplikacji.

W kolejnym i zarazem ostatnim rozdziale książki zdobytą dotąd wiedzę wykorzystasz do utworzenia aplikacji.

SKOROWIDZ

I

J

K

L

M

O

P

R

S

Ś

T

U

W

X

Z

Język Swift, uważany za następcę Objective-C, bardzo szybko stał się popularny wśród programistów tworzących aplikacje na platformę iOS. A to z wielu przyczyn: Swift w porównaniu ze swoim poprzednikiem charakteryzuje się dużo bardziej zwięzłą i przejrzystą składnią, a kod napisany w tym języku jest znacznie krótszy i czytelniejszy. Ponadto łączy wiele elementów takich języków, jak JavaScript, Python, Ruby czy C#, dzięki czemu posługujące się nimi osoby nie powinny mieć trudności z opanowaniem Swifta.

Swift jest językiem kompilowanym o dużej wydajności, z kompilatorem typu LLVM (ang. Low Level Virtual Machine). Oferuje liczne funkcje wspomagające programowanie, przy tym jest elastyczny, jeśli chodzi o typy danych. Dzięki swojej strukturze umożliwia wykorzystywanie paradygmatu programowania funkcyjnego. Autor tej książki, uznany autorytet w dziedzinie programowania, w przystępny sposób przedstawił podstawy programowania w Swifcie i przygotował zestaw utrwalających tę wiedzę ćwiczeń i przykładów.

Z racji tego, że Swift jest nowym i bardzo obiecującym językiem programowania, umiejętność tworzenia w nim aplikacji może wkrótce stać się Twoim atutem — zacznij naukę już dziś!

Dzięki tej książce:

- bez problemów rozpoczniesz pracę ze Swiftem
- poznasz typy zmiennych
- opanujesz polecenia i pętle
- nauczysz się tworzyć i uaktualniać tablice oraz słownik
- poznasz rodzaje oraz metody wykorzystywania funkcji, klas i protokołów
- stworzysz interfejs aplikacji na iOS 8 z użyciem Swifta

Helion

38935 numer katalogowy

księgarnia internetowa
http://helion.pl

zamówienia telefoniczne
0 801 339900
0 601 339900

Sprawdź najnowsze promocje:
http://helion.pl/promocje
Książki najchętniej czytane:
http://helion.pl/bestsellery
Zamów informacje o nowościach:
http://helion.pl/nowosci

Helion SA
ul. Kościuszki 1c, 44-100 Gliwice
tel.: 32 230 98 63
e-mail: helion@helion.pl
http://helion.pl

ISBN 978-83-283-1912-7

Informatyka w najlepszym wydaniu

cena: 49,00 zł