Funkcje i zamknięcia

W tym momencie możesz pisać dość złożone programy, które potrafią podejmować decyzje i powtarzać sekwencje instrukcji. Możesz również przechowywać dane dla swoich programów, używając typów kolekcji. W miarę jak programy, które piszesz, stają się coraz większe i bardziej złożone, zrozumienie ich działania będzie coraz trudniejsze. Aby ułatwić zrozumienie dużych programów, Swift umożliwia tworzenie funkcji, które pozwalają łączyć ze sobą kilka instrukcji i wykonywać je przez wywołanie jednej nazwy. Możesz także tworzyć domknięcia, co pozwala łączyć ze sobą kilka instrukcji bez nazwy i przypisywać je do stałej lub zmiennej. Pod koniec tego rozdziału poznasz funkcje, funkcje zagnieżdżone, funkcje jako typy zwracane, funkcje jako argumenty i instrukcję guard. Dowiesz się również, jak tworzyć i używać zamknięć. Omówione zostaną następujące tematy:

• Zrozumienie funkcji

• Zrozumienie zamknięć

Zrozumienie funkcji

Funkcje są przydatne do hermetyzacji szeregu instrukcji, które wspólnie wykonują określone zadanie, na przykład:

• Naliczanie 10% opłaty serwisowej za posiłek w restauracji.

• Obliczanie miesięcznej opłaty za samochód, który chcesz kupić.

Oto jak wygląda funkcja:

```
func nazwa_funkcji(parametr1: typ_parametru, ...) -> typ powrotu

{

kod

}
```

Każda funkcja ma opisową nazwę. Możesz zdefiniować jedną lub więcej wartości, które funkcja przyjmuje jako dane wejściowe, zwanych parametrami. Możesz także zdefiniować, co funkcja wygeneruje po zakończeniu, co nazywa się jej typem zwracanym. Zarówno parametry, jak i typy zwracane są opcjonalne. „Wywołujesz” nazwę funkcji, aby ją wykonać. Tak wygląda wywołanie funkcji:

```
functionName(parametr1: argument1, …)
```

Podajesz wartości wejściowe (znane jako argumenty), które pasują do typu parametrów funkcji. Zobaczmy, jak możesz utworzyć funkcję do obliczania opłaty za usługę w następnej sekcji.

Tworzenie funkcji

W swojej najprostszej postaci funkcja wykonuje tylko niektóre instrukcje i nie ma żadnych parametrów ani typów zwracanych. Zobaczysz, jak to działa, pisząc funkcję obliczania opłaty za usługę za posiłek. Opłata za obsługę powinna wynosić 10% kosztu posiłku. Dodaj następujący kod do swojego placu zabaw, aby utworzyć i wywołać tę funkcję, a następnie kliknij przycisk Odtwórz/Zatrzymaj, aby ją uruchomić:

```
func serviceCharge() {

let mealCost = 50
```

```
let serviceCharge = mealCost / 10

print("Service charge is \(serviceCharge)")

}

serviceCharge()
```

Właśnie stworzyłeś bardzo prostą funkcję o nazwie serviceCharge(). Jedyne, co robi, to obliczanie 10% opłaty za usługę za posiłek kosztujący 50 USD, czyli 50 / 10, zwracając 5. Następnie wywołujesz tę funkcję, używając jej nazwy. Zobaczysz, że opłata za usługę wynosi 5 w obszarze debugowania. Ta funkcja nie jest zbyt użyteczna, ponieważ mealCost zawsze wynosi 50 za każdym wywołaniem tej funkcji, a wynik 5 jest drukowany tylko w obszarze Debug i nie może być użyty w innym miejscu programu. Dodajmy do tej funkcji kilka parametrów i typ zwracany, aby była bardziej użyteczna. Zmodyfikuj swój kod, jak pokazano:

```
func serviceCharge(mealCost: Int) -> Int {

return mealCost / 10

}

let serviceChargeAmount = serviceCharge(mealCost: 50)

print(serviceChargeAmount)
```

To jest dużo lepsze. Teraz możesz ustawić koszt posiłku po wywołaniu funkcji serviceCharge(mealCost:), a wynik można przypisać do zmiennej lub stałej. Wygląda to jednak trochę niezręcznie. Powinieneś spróbować, aby podpisy funkcji w Swift były odczytywane jak zdanie angielskie, ponieważ jest to uważane za najlepszą praktykę. Zobaczmy, jak to zrobić w następnej sekcji, w której użyjesz niestandardowych etykiet, aby Twoja funkcja była bardziej zbliżona do języka angielskiego i łatwiejsza do zrozumienia.

Korzystanie z niestandardowych etykiet argumentów

Zauważ, że funkcja serviceCharge(mealCost:) nie jest zbyt podobna do angielskiego. Możesz dodać niestandardową etykietę do parametru, aby ułatwić zrozumienie funkcji. Zmodyfikuj swój kod, jak pokazano:

```
func serviceCharge(forMealPrice mealCost: Int) -> Int {

return mealCost / 10

}

let serviceChargeAmount = serviceCharge(forMealPrice: 50)

print(serviceChargeAmount)
```

Funkcja działa dokładnie tak samo jak poprzednio, ale do jej wywołania używasz serviceCharge(forMealPrice:). Brzmi to bardziej jak angielski i ułatwia zorientowanie się, co robi funkcja. W następnej sekcji dowiesz się, jak używać kilku mniejszych funkcji w ciałach innych funkcji, które są znane jako funkcje zagnieżdżone. Korzystanie z funkcji zagnieżdżonych.

Możliwe jest posiadanie funkcji w treści innej funkcji i są one nazywane funkcjami zagnieżdżonymi. Funkcja zagnieżdżona może używać zmiennych funkcji otaczającej. Zobaczmy, jak działają

zagnieżdżone funkcje, pisząc funkcję do obliczania miesięcznych spłat pożyczki. Wpisz i uruchom następujący kod:

```
func calculateMonthlyPayments(carPrice: Double, downPayment:

Double, interestRate: Double, paymentTerm: Double) -> Double {

func loanAmount() -> Double {

return carPrice - downPayment

}

func totalInterest() -> Double {

Understanding functions 85

return interestRate * paymentTerm

}

func numberOfMonths() -> Double {

return paymentTerm * 12

}

return ((loanAmount() + ( loanAmount() *

totalInterest() / 100 )) / numberOfMonths())

}

calculateMonthlyPayments(carPrice: 50000, downPayment: 5000,

interestRate: 3.5, paymentTerm: 7.0)
```

Tutaj znajdują się trzy funkcje w ramach metody obliczaniaPłatnościMiesięcznych (carPrice:downPayment:interestRate:paymentTerm:). Przyjrzyjmy się im:

• Pierwsza zagnieżdżona funkcja, loanAmount(), oblicza całkowitą kwotę pożyczki, odejmując downPayment od carPrice. Zwraca 50000 - 5000 = 45000.

• Druga funkcja zagnieżdżona, totalInterest(), oblicza łączną kwotę odsetek naliczonych w okresie płatności poprzez pomnożenie stopy procentowej przez paymentTerm. Zwraca 3,5 * 7 = 24,5.

• Trzecia funkcja zagnieżdżona numberOfMonths() oblicza całkowitą liczbę miesięcy w okresie płatności, mnożąc paymentTerm przez 12. Zwraca 7 * 12 = 84.

Zauważ, że wszystkie trzy funkcje zagnieżdżone używają zmiennych funkcji otaczającej. Zwrócona wartość to (45000 + (45000*24,5/100))/84=666.96, co jest kwotą, którą trzeba płacić miesięcznie przez 7 lat, aby kupić ten samochód. Jak widać, funkcje w Swift są podobne do funkcji w innych językach, ale mają fajną funkcję. Funkcje są typami pierwszej klasy w Swift, więc mogą być używane jako parametry i typy zwracane. Zobaczmy, jak to się robi w następnej sekcji.

Używanie funkcji jako typów zwracanych

Funkcja może zwrócić inną funkcję jako typ zwracany. Wpisz i uruchom następujący kod, aby utworzyć funkcję, która generuje wartość dla Pi:

```
func makePi() -> (() -> Double) {

func generatePi() -> Double {

return 22.0 / 7.0

}

return generatePi

}

let pi = makePi()

print(pi())
```

Typem zwracanym przez funkcję makePi() jest funkcja, która nie ma parametrów, a typem zwracanym jest Double. generatePi() to funkcja, która nie ma parametrów, a typem zwracanym jest Double i będzie funkcją, która zostanie zwrócona. Tak więc pi zostanie przypisane generatePi() i zwróci 22.0/7,0 po wywołaniu. 3.142857142857143 zostanie wydrukowane w obszarze Debug. Zobaczmy, jak funkcja może być użyta jako parametr innej funkcji w następnej sekcji.

Używanie funkcji jako parametrów

Funkcja może przyjąć funkcję jako parametr. Wpisz i uruchom następujący kod, aby utworzyć funkcję, która określa, czy liczba spełniająca określony warunek istnieje na liście liczb:

```
func isThereAMatch(listOfNumbers: [Int], condition: (Int) ->

Bool) -> Bool {

for item in listOfNumbers {

if condition(item) {

return true

}

}

return false

}

func oddNumber(number: Int) -> Bool {

return (number % 2) > 0

}

var numbersList = [2, 4, 6, 7]

isThereAMatch(listOfNumbers: numbersList, condition: oddNumber)
```

isThereAMatch(listOfNumbers:condition:) ma dwa parametry; tablica liczb całkowitych i funkcja. Funkcja podana jako argument musi przyjmować wartość całkowitą i zwracać wartość logiczną. nieparzysta(liczba:) przyjmuje liczbę całkowitą i zwraca prawdę, jeśli liczba jest nieparzysta, co oznacza, że może być argumentem dla drugiego parametru. Jako argument dla pierwszego parametru używana

jest tablica numbersList, która zawiera liczbę nieparzystą. Ponieważ numbersList zawiera liczbę nieparzystą, funkcja isThereAMatch(listOfNumbers:condition:) zwróci true po wywołaniu. W następnej sekcji zobaczysz, jak wykonać wczesne wyjście z funkcji, jeśli użyte argumenty nie są odpowiednie.

Używanie instrukcji guard do wcześniejszego wyjścia z funkcji

Jeśli coś jest nie tak z danymi wejściowymi, warto mieć możliwość wcześniejszego wyjścia z funkcji. Załóżmy, że potrzebujesz funkcji do użycia w terminalu zakupów online. Ta funkcja obliczy pozostałe saldo karty debetowej lub kredytowej, gdy coś kupisz. Cena przedmiotu, który chcesz kupić, jest wprowadzana w polu tekstowym. Wartość w polu tekstowym jest konwertowana na liczbę całkowitą, dzięki czemu można obliczyć pozostałe saldo karty. Wpisz i uruchom następujący kod:

```
func buySomething(itemValueEntered itemValueField: String,

cardBalance: Int) -> Int {

guard let itemValue = Int(itemValueField) else {

print("error in item value")

return cardBalance

}

let remainingBalance = cardBalance - itemValue

return remainingBalance

}

print(buySomething(itemValueEntered: "10", cardBalance: 50))

print(buySomething(itemValueEntered: "blue", cardBalance: 50))
```

Powinieneś zobaczyć ten wynik w obszarze Debug:

```
40

error in item value

50
```

Zobaczmy, jak działa ta funkcja. Pierwszy wiersz w treści funkcji to instrukcja strażnika. To sprawdza, czy warunek jest prawdziwy; jeśli nie, wychodzi z funkcji. Tutaj służy do sprawdzenia i zobaczenia, czy użytkownik wprowadził prawidłową cenę w terminalu zakupów online. Jeśli tak, wartość można z powodzeniem przekonwertować na liczbę całkowitą i obliczyć pozostałe saldo karty. W przeciwnym razie wykonywana jest klauzula else w instrukcji guard. Komunikat o błędzie jest drukowany w obszarze debugowania i zwracane jest niezmienione saldo karty. W przypadku print(buySomething(itemValueEntered: "10", cardBalance: 50)) cena przedmiotu jest pomyślnie odliczana od salda karty, a 40 jest zwracanych. W przypadku print(buySomething(itemValueEntered: "blue", cardBalance: 50)) warunek instrukcji guard kończy się niepowodzeniem i wykonywana jest jej klauzula else, co skutkuje wydrukowaniem komunikatu o błędzie w obszarze debugowania i zwróceniem 50. Wiesz już, jak tworzyć i używać funkcji. Zobaczyłeś również, jak używać niestandardowych etykiet argumentów, funkcji zagnieżdżonych, funkcji jako parametrów lub typów zwracanych oraz instrukcji guard. Spójrzmy teraz na zamknięcia. Podobnie jak funkcje, domknięcia

pozwalają łączyć ze sobą kilka instrukcji, ale domknięcia nie mają nazw i mogą być przypisane do stałej lub zmiennej. Zobaczysz, jak działają w następnej sekcji.

Zrozumienie zamknięć

Zamknięcie, podobnie jak funkcja, zawiera sekwencję instrukcji i może przyjmować argumenty i zwracać wartości. Zamknięcia nie mają jednak nazw. Sekwencja instrukcji w zamknięciu jest otoczona nawiasami klamrowymi ({ }), a słowo kluczowe in oddziela argumenty i zwracany typ od treści zamknięcia. Zamknięcia mogą być przypisane do stałej lub zmiennej, więc są przydatne, jeśli musisz je przekazywać w swoim programie. Załóżmy na przykład, że masz aplikację, która pobiera plik z Internetu i po zakończeniu pobierania musisz coś zrobić z plikiem. Możesz umieścić listę instrukcji przetwarzania pliku w zamknięciu i uruchomić go po zakończeniu pobierania pliku.

Napiszesz teraz zamknięcie, które stosuje obliczenia na każdym elemencie tablicy liczb. Dodaj następujący kod do swojego placu zabaw i kliknij przycisk Graj/Zatrzymaj, aby go uruchomić:

```
var numbersArray = [2, 4, 6, 7]

let myClosure = { (number: Int) -> Int in

let result = number * number

return result

}
```

To przypisuje do myClosure zamknięcie, które oblicza potęgę liczby dwójki. Funkcja map() następnie stosuje to zamknięcie do każdego elementu w numbersArray. Każdy element jest mnożony przez siebie iw polu Wyniki pojawia się [4, 16, 36, 49]. Możliwe jest pisanie zamknięć w bardziej zwięzły sposób, a zobaczysz, jak to zrobić w w następnej sekcji.

Uproszczone zamknięcia

Jedną z rzeczy, z którymi mają problemy nowi programiści, jest bardzo zwięzły sposób, w jaki doświadczeni programiści Swift używają do pisania zamknięć. Rozważmy kod pokazany w poniższym przykładzie:

```
var testNumbers = [2, 4, 6, 7]

let mappedTestNumbers = testNumbers.map({ (number: Int)

-> Int in

let result = number * number

return result

})

print(mappedTestNumbers)
```

Tutaj masz testNumbers, tablicę liczb, i używasz funkcji map(_:) do mapowania zamknięcia do każdego elementu tablicy po kolei. Kod w zamknięciu sam mnoży liczbę, generując kwadrat tej liczby. Wynik [4, 16, 36, 49] jest następnie drukowany w obszarze Debug. Jak zobaczysz, kod zamknięcia można napisać bardziej zwięźle. Gdy typ zamknięcia jest już znany, możesz usunąć typ parametru, typ zwracany lub oba. Zamknięcia pojedynczych instrukcji niejawnie zwracają wartość swojej jedynej instrukcji, co

oznacza, że możesz również usunąć instrukcję return. Możesz więc napisać zamknięcie w następujący sposób:

```
let mappedTestNumbers = testNumbers.map({ number in

number * number

})
```

Gdy zamknięcie jest jedynym argumentem funkcji, możesz pominąć nawiasy otaczające zamknięcie w następujący sposób:

```
let mappedTestNumbers = testNumbers.map { number in

number * number

}
```

Możesz odwoływać się do parametrów za pomocą liczby wyrażającej ich względną pozycję na liście argumentów zamiast nazwy, w następujący sposób:

```
let mappedTestNumbers = testNumbers.map { $0 * $0 }
```

Tak więc zamknięcie teraz jest rzeczywiście bardzo zwięzłe, ale będzie trudne do zrozumienia dla nowych programistów. Zapraszam do pisania zamknięć w sposób, który Ci odpowiada. Wiesz już, jak tworzyć i używać domknięć oraz jak je pisać bardziej zwięźle. Świetny!

Podsumowanie

Nauczyłeś się, jak grupować wyrażenia w funkcje. Nauczyłeś się używać niestandardowych etykiet argumentów, funkcji wewnątrz innych funkcji, funkcji jako typów zwracanych i funkcji jako parametrów. Będzie to przydatne później, gdy będziesz musiał wykonać to samo zadanie w różnych punktach programu. Nauczyłeś się także tworzyć zamknięcia. Będzie to przydatne, gdy musisz przekazywać bloki kodu w swoim programie. W następnym rozdziale zapoznasz się z klasami, strukturami i wyliczeniami. Klasy i struktury pozwalają na tworzenie złożonych obiektów, które mogą przechowywać stan i zachowanie, a wyliczenia mogą służyć do ograniczania wartości, które można przypisać do zmiennej lub stałej, zmniejszając prawdopodobieństwo błędu.