ROZDZIAŁ 5

Tworzenie funkcji wyższego rzędu w języku Scala

Venkat Subramaniam

W rozdziale 3, *Scala i styl funkcyjny* (s. 15), omawialiśmy funkcje wyższego rzędu w programowaniu funkcyjnym, a w rozdziale 4, *Praca z kolekcjami języka Scala* (s. 23), przyjrzeliśmy się funkcjom wyższego rzędu w API kolekcji Scali. W tym rozdziale dowiemy się, jak pisać własne funkcje wyższego rzędu.

Funkcje wyższego rzędu mogą akceptować inne funkcje jako parametry, mogą zwracać funkcje, mogą pozwolić na tworzenie funkcji wewnątrz funkcji. W języku Scala funkcje te mogą być przekazywane i są nazywane *wartościami funkcyjnymi*. Wiemy, że w programowaniu obiektowym klasy (lub obiekty) abstrahują od zachowania danych i je enkapsulują. Wartości funkcji także enkapsulują zachowanie i abstrahują od niego, ale zamiast trzymać się stanu, mogą pomóc w jego przekształceniu. Popatrzmy na dwa przykłady, gdzie wartości funkcyjne stają się przydatne.

Tworzenie funkcji wyższego rzędu

Kontynuując nasz przykład giełdowy, napiszemy funkcję, która zsumuje ceny podane w kolekcji. Wydaje się, że wystarczy prosta iteracja, więc piszemy:

```
val prices = List(10, 20, 15, 30, 45, 25, 82)

def totalAllPrices(prices : List[Int]) = {
  prices.foldLeft(0) { (total, price) =>
    total + price
  }
}
```

W funkcji `totalAllPrices` metoda `foldLeft` związana z listą jest używana do obliczenia sumy w stylu funkcyjnym z pełną niemutowalnością. Wartość

funkcji przekazujemy do metody `foldLeft`. Ta wartość funkcji akceptuje dwa parametry i zwraca ich sumę. Metoda `foldLeft` wywołuje wartość funkcji tyle razy, ile jest elementów na liście. Za pierwszym razem `total` i `price` są powiązane z wartością 0 (przekazywaną jako parametr do metody `foldLeft`) oraz odpowiednio pierwszy element na liście. Przy drugim wywołaniu `total` jest powiązane z sumą zwracaną z poprzedniego wywołania wartości funkcji, `price` zaś jest powiązana z drugim elementem z kolekcji. Funkcja `foldLeft` iteruje tę sekwencję wywołań dla pozostałych elementów z kolekcji.

Przećwiczmy naszą funkcję `totalAllPrices`, aby zobaczyć wynik.

```
println("Total of prices is " + totalAllPrices(prices))
//Total of prices is 227
```

Zanim możemy zadeklarować, że zostało to zrobione, otrzymujemy prośbę o napisanie jeszcze jednej funkcji, która zsumuje tylko ceny wyższe od podanej wartości. Można po prostu ponownie użyć większej części kodu z małej funkcji, którą właśnie napisaliśmy. Patrząc na zegarek (mamy umówione spotkanie), mówimy sobie „Oto dlaczego Bóg stworzył kopiuj i wklej”. W rezultacie otrzymujemy taką funkcję:

```
def totalOfPricesOverValue(prices : List[Int], value : Int) {
  prices.foldLeft(0) { (total, price) =>
    if (price > value) total + price else total
  }
}
```

Niestety, zapotrzebowanie na własności wydaje się dziś niewyczerpane i za każdym razem jesteśmy proszeni o jeszcze jedną funkcję, tym razem do zsumowania tylko tych cen, które są niższe od podanej wartości. Wiemy, że kopiowanie i wklejanie kodu jest niemoralne, ale na razie się na to decydujemy, refaktorując go, aby stał się lepszy zaraz po spotkaniu, na które musimy iść.

Tuż po spotkaniu przyglądamy się kolejnej wersji swojego kodu:

```
val prices = List(10, 20, 15, 30, 45, 25, 82)

def totalAllPrices(prices : List[Int]) = {
  prices.foldLeft(0) { (total, price) =>
    total + price
  }
}

def totalOfPricesOverValue(prices : List[Int], value : Int) = {
  prices.foldLeft(0) { (total, price) =>
    if (price > value) total + price else total
  }
}
```

```
def totalOfPricesUnderValue(prices : List[Int], value : Int) {
  prices.foldLeft(0) { (total, price) =>
    if (price < value) total + price else total
  }
}
```

Sprawdźmy go:

```
println("Total of prices is " + totalAllPrices(prices))
//Suma cen wynosi 227

println("Total of prices over 40 is " +
 totalOfPricesOverValue(prices, 40))
// Suma cen powyżej 40 wynosi 127

println("Total of prices under 40 is " +
 totalOfPricesUnderValue(prices, 40))
// Suma cen poniżej 40 wynosi 100
```

Mamy najlepsze intencje, aby program działał i robił to lepiej, ale chcemy go szybko refaktoryzować, aby usunąć duplikacje, zanim ktoś oskarży nas o to, że kodem tym ujawniamy naszą ciemną stronę. Mamy do ocalenia wartości funkcji.

Jeśli lekko zmodyfikujemy pierwszą funkcję do postaci *`if (true) total + price`*, zauważymy, że jedyną różnicą między treścią trzech funkcji jest wyrażenie warunkowe w instrukcji `if`. Możemy wyodrębnić ten warunek jako wartość funkcji.

Wyodrębniona wartość funkcji przyjmie `Int` jako parametr i zwróci `Boolean`. Możemy to wyrazić w postaci odwzorowania lub przekształcenia z `Int` na `Boolean` lub jako `selector : Int => Boolean`. Podobnie jak `prices : List[Int]` reprezentuje ceny (`prices`) odniesienia typu `List[Int]`, `selector : Int => Boolean` reprezentuje selektor typu `function value`, który akceptuje `Int` i zwraca `Boolean`.

Możemy teraz zastąpić trzy poprzednie funkcje za pomocą jednej funkcji:

```
def totalPrices(prices : List[Int],
  selector : Int => Boolean) = {

  prices.foldLeft(0) { (total, price) =>
    if (selector(price)) total + price else total
  }
}
```

Funkcja `totalPrices` akceptuje kolekcję i wartość funkcji jako parametr. W funkcji, w warunku `if`, możemy wywołać wartość funkcji z ceną jako parametrem. Jeśli wartość funkcji selektora da wynik true, możemy dodać cenę do sumy, a w przeciwnym przypadku ignorujemy cenę.