Vikash Sharma

Scala

Nauka programowania

Helion

Packt›

Tytuł oryginału: Learning Scala Programming

Tłumaczenie: Robert Górczyński

ISBN: 978-83-283-4796-0

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/scalan
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Printed in Poland.

Spis treści

4

ROZDZIAŁ

Podział programu na funkcje

Kod programowania zorientowanego obiektowo zrozumiemy, hermetyzując jego ruchome części. Kod programowania funkcyjnego zrozumiemy, minimalizując liczbę ruchomych części.

— Michael Feathers

Scala jako język łączący różne paradygmaty programowania motywuje do wielokrotnego użycia tego samego kodu i jednocześnie oczekuje stosowania koncepcji stojących za programowaniem funkcyjnym. Oznacza to, że program powinien zostać podzielony na mniejsze fragmenty odpowiedzialne za wykonywanie starannie zdefiniowanych zadań. Konstrukcją pozwalającą na osiągnięcie tego celu jest funkcja. Funkcja to po prostu konstrukcja logiczna wykonująca określone zadanie, której można użyć wielokrotnie, jeśli trzeba.

Istnieją określone sposoby na pojawienie się funkcji w programie i użycie ich w nim. Mamy wiele powodów, dla których warto korzystać z funkcji w programie. Doskonale przygotowana funkcja wraz z dokładną liczbą niezbędnych argumentów, ze starannie określonym zasięgiem i podanymi kwestiami prywatności powoduje, że kod prezentuje się świetnie. Co więcej, funkcje nadają także większy sens programowi. Na podstawie wymagań programu można zastosować pewne określone strategie, wykorzystując przy tym zmiany syntaktyczne. Przykładowo funkcja będzie wykonana wtedy i tylko wtedy, gdy jest niezbędna. Inną strategią jest wczytywanie z opóźnieniem (ang. *lazy loading*) oznaczające, że wyrażenie będzie obliczone dopiero w trakcie pierwszej operacji dostępu do niego. Przedstawię teraz wprowadzenie do funkcji w Scali. Oto krótka lista tematów poruszonych w rozdziale:

- składnia funkcji,
- wywoływanie funkcji,
- literał funkcji i funkcja anonimowa,

- strategie wywoływania funkcji,
- funkcja częściowa.

Na początku pokażę, jak można utworzyć funkcję w Scali.

Składnia funkcji

Do utworzenia funkcji w Scali jest używane słowo kluczowe `def`, po którym trzeba podać nazwę funkcji i argumenty dostarczające jej dane wejściowe. Spójrz na przedstawioną tutaj ogólną składnię konstrukcji pozwalającej na utworzenie funkcji.

```
modifikatory...
def nazwa_funkcji(arg1: typ_arg1, arg2: typ_arg2,...): typ_wartości_zwrotnej = ???
```

Pokazałem tutaj składnię ogólnej sygnatury funkcji w Scali. Na początku znajdują się modyfikatory funkcji. To po prostu właściwości zdefiniowane dla funkcji. Modyfikatory mogą przybierać różne postaci, oto kilka z nich:

- adnotacja,
- modyfikator nadpisania,
- modyfikator dostępu, na przykład `private` i tak dalej,
- słowo kluczowe `final`.

Zalecaną praktyką jest używanie modyfikatorów wedle potrzeb, w podanej ich kolejności. Po podaniu modyfikatorów należy użyć słowa kluczowego `def` wraz z nazwą tworzonej funkcji. Później wymienia się parametry funkcji w następującej postaci: nazwa parametru wraz z dwukropkiem i jego typ. To nieco inaczej niż w Javie, w której kolejność jest dokładnie odwrotna. Na końcu trzeba określić typ wartości zwrotnej funkcji. Istnieje możliwość pominięcia typu wartości zwrotnej, w takim przypadku zostanie on ustalony automatycznie przez Scalę. Gdy pominiemy kilka przypadków szczególnych, przedstawione tutaj podejście sprawdza się doskonale. Aby zapewnić przejrzystość programu, można stosować praktykę, zgodnie z którą typ wartości zwrotnej stanowi część sygnatury funkcji. Po zadeklarowaniu funkcji kolejnym krokiem jest umieszczenie poleceń w jej bloku. Spójrz na przedstawiony tutaj przykład.

```
def compareIntegers(value1: Int, value2: Int): Int = if (value1 == value2) 0 else
↪if (value1 > value2) 1 else -1
```

Mamy tutaj przykład definiujący funkcję oczekującą dwóch wartości w postaci liczb całkowitych, porównującą je, a następnie zwracającą wynik też w postaci liczby całkowitej. Blok funkcji również jest prosty — podane wartości są porównywane. Jeżeli są równe, wartością zwrotną funkcji jest `0`. Jeżeli pierwsza wartość jest większa, funkcja zwróci `1`. Jeśli natomiast druga wartość jest większa, funkcja zwróci `-1`. W omawianym tutaj przypadku nie zostały użyte żadne modyfikatory. Scala domyślnie traktuje funkcję jako publiczną, a to oznacza możliwość uzyskania dostępu do niej z poziomu każdej klasy oraz nadpisania funkcji.

Gdy dokładnie przyjrzysz się tej funkcji, dostrzeżesz, że została zdefiniowana w miejscu. Funkcja ta jest zdefiniowana bezpośrednio z dwóch powodów. Oto one:

- uproszczenie kodu źródłowego i zapewnienie mu większej czytelności,
- definicja jest na tyle mała, że zmieściła się w jednym wierszu.

Zalecaną praktyką jest definiowanie funkcji w pokazany tutaj sposób, gdy jej sygnatura wraz z definicją ma maksymalnie około 30 znaków. Jeżeli ma więcej i nadal pozostaje zwięzła, definicję można umieścić w następnym wierszu, tak jak pokazałem w kolejnym fragmencie kodu.

```
def compareIntegersV1(value1: Int, value2: Int): Int = if (value1 == value2) 0
↪else if (value1 > value2) 1 else -1
```

Wybór należy do Ciebie: możesz zdecydować się na większą czytelność kodu lub na zdefiniowanie funkcji w miejscu. Jeżeli blok funkcji zawiera wiele wierszy kodu, warto umieścić go w nawiasie klamrowym.

```
def compareIntegersV2(value1: Int, value2: Int): Int = {
  println(s" Wykonywanie funkcji V2")
  if (value1 == value2) 0 else if (value1 > value2) 1 else -1
}
```

W tej definicji funkcji znajdują się dwa polecenia. Pierwsze wyświetla komunikat, natomiast drugie przeprowadza operację porównania. Dlatego też zostały one umieszczone w nawiasie klamrowym. Spójrz teraz na cały program.

```
object FunctionSyntax extends App{
 /*
  * Funkcja porównuje dwie liczby całkowite.
  * @param value1 Int
  * @param value2 Int
  * wartość zwrotna typu Int
  * 1  jeśli value1 > value2
  * 0  jeśli value1 = value2
  * -1 jeśli value1 < value2
  */
  def compareIntegers(value1: Int, value2: Int): Int = if (value1 == value2) 0
  ↪else if (value1 > value2) 1 else -1

  def compareIntegersV1(value1: Int, value2: Int): Int = {
    if (value1 == value2) 0 else if (value1 > value2) 1 else -1
   }

  def compareIntegersV2(value1: Int, value2: Int): Int =
    if (value1 == value2) 0 else if (value1 > value2) 1 else -1

  println(compareIntegers(1, 2))
  println(compareIntegersV1(2, 1))
  println(compareIntegersV2(2, 2))
}
```

Oto dane wyjściowe wygenerowane przez przedstawiony program:

```
-1
1
0
```

Podczas definiowania bloku funkcji ostatnie polecenie jest odpowiedzialne za zwrot wartości przez tę funkcję. W omawianym przykładzie następuje wykonanie wyrażenia `if-else`, wynikiem wyrażenia jest liczba całkowita, która będzie typem wartości zwrotnej funkcji `compareIntegers()`.

Zagnieżdżanie funkcji

Gdy tylko istnieje możliwość opakowania logiki, z reguły następuje przekształcenie tego fragmentu kodu na funkcję. Jeśli zostanie to zrobione zbyt wiele razy, może doprowadzić do zaśmiecenia kodu. Podczas podziału funkcji na mniejsze jednostki pomocnicze zwykle nadajemy im bardzo podobne nazwy. Spójrz na pokazany tutaj przykład.

```
object FunctionSyntaxOne extends App {

  def compareIntegersV4(value1: Int, value2: Int): String = {
    println("Wykonywanie funkcji V4")
    val result = if (value1 == value2) 0 else if (value1 > value2) 1 else -1
    giveAMeaningFullResult(result, value1, value2)
  }

  private def giveAMeaningFullResult(result: Int, value1: Int, value2: Int) =
  ↪result match {
    case 0 => "Wartości są równe"
    case -1 => s"Wartość $value1 jest mniejsza niż $value2"
    case 1 => s"Wartość $value1 jest większa niż $value2"
    case _ => "Nie można przeprowadzić operacji"
  }

  println(compareIntegersV4(2,1))
}
```

Oto dane wyjściowe wygenerowane przez przedstawiony program:

```
Wykonywanie funkcji V4
Wartość 2 jest większa niż 1
```

W tym programie została zdefiniowana funkcja `compareIntegersV4()`, w której po porównaniu dwóch liczb całkowitych następuje wywołanie funkcji pomocniczej `giveAMeaningFullResult()` wraz z wynikiem i dwiema wartościami. Działanie funkcji pomocniczej polega na wygenerowaniu czytelnego dla człowieka ciągu tekstowego na podstawie wyniku działania funkcji `compareIntegersV4()`. Ten kod działa świetnie, ale jeśli dokładnie się mu przyjrzysz i zastanowisz nad tym rozwiązaniem, dojdziesz do wniosku, że metoda prywatna ma znaczenie tylko dla funkcji `compareIntegersV4()`. Dlatego też byłoby znacznie lepiej, gdyby definicja `giveAMeaningFullResult()` została umieszczona wewnątrz funkcji `compareIntegersV4()`. Przeprowadzimy więc refaktoryzację kodu w taki sposób, aby funkcja pomocnicza została zagnieżdżona w `compareIntegersV5()`.

```
object FunctionSyntaxTwo extends App {

  def compareIntegersV5(value1: Int, value2: Int): String = {
    println("Wykonywanie funkcji V5")

    def giveAMeaningFullResult(result: Int) = result match {
      case 0 => "Wartości są równe"
      case -1 => s"Wartość $value1 jest mniejsza niż $value2"
      case 1 => s"Wartość $value1 jest większa niż $value2"
      case _ => "Nie można przeprowadzić operacji"
    }

    val result = if (value1 == value2) 0 else if (value1 > value2) 1 else -1
    giveAMeaningFullResult(result)
  }

  println(compareIntegersV5(2,1))
}
```

Oto dane wyjściowe wygenerowane przez przedstawiony program:

```
Wykonywanie funkcji V5
Wartość 2 jest większa niż 1
```

Jak możesz zobaczyć w tym programie, zdefiniowaliśmy funkcję zagnieżdżoną `giveAMean` ↪`ingFullResult()`, która również została nieco zmodyfikowana. Po modyfikacji oczekuje ona tylko jednego parametru w postaci liczby całkowitej i na jego podstawie generuje odpowiedni ciąg tekstowy. Możliwe jest uzyskanie dostępu do wszystkich zmiennych znajdujących się w funkcji nadrzędnej. Dlatego też można było pominąć przekazywanie wartości `value1` i `value2` do zagnieżdżonej funkcji pomocniczej. Dzięki temu kod źródłowy stał się znacznie czytelniejszy. Funkcję można wywołać bezpośrednio wraz z argumentami, tutaj `2` i `1`. Istnieje wiele różnych sposobów na wywołanie funkcji. Zanim przejdziemy dalej, warto je poznać.

Wywołanie funkcji

Funkcję można wywołać, aby wykonała zadanie, dla którego została zdefiniowana. Podczas wywoływania funkcji następuje przekazanie jej argumentów działających w charakterze danych wejściowych tej funkcji. Można to zrobić na wiele sposobów: podać zmienną liczbę argumentów, podać nazwę argumentu lub określić wartość domyślną przeznaczoną do użycia wtedy, gdy podczas wywołania funkcji nie zostanie przekazany argument. Rozważmy na przykład sytuację, w której nie ma pewności dotyczącej liczby argumentów przekazywanych funkcji, choć znany jest ich typ.

Przekazywanie zmiennej liczby argumentów

Powinieneś pamiętać, że w poprzednim rozdziale już spotkałeś funkcję pobierającą zmienną liczbę argumentów i przeprowadzającą pewne operacje.

```
/*
 * Wydruk stron dokumentu o podanych numerach.
 */
def printPages(doc: Document, indexes: Int*) = for(index <- indexes if index <=
↪doc.numOfPages) print(index)
```

Metoda pobiera numery stron, a następnie drukuje je z dokumentu przekazanego jako pierwszy parametr. Tutaj parametr `indexes` jest określany mianem **vararg**. Oznacza to możliwość przekazania dowolnej liczby argumentów podanego typu, którym w omawianym programie jest `Int`. Podczas wywoływania tej funkcji należy przekazać dowolną liczbę argumentów typu `Int`. Tę możliwość już wypróbowaliśmy. Zastanówmy się teraz nad funkcją matematyczną oczekującą pewnej ilości liczb całkowitych i zwracającą ich wartość średnią. Jak taka funkcja mogłaby zostać zdefiniowana?

To może być sygnatura wraz ze słowem kluczowym `def`, nazwą i parametrami lub tylko jednym parametrem *vararg*.

```
def average(numbers: Int*): Double = ???
```

Skoro mamy sygnaturę funkcji `average()`, można przejść do zdefiniowania jej bloku.

```
object FunctionCalls extends App {

  def average(numbers: Int*) : Double = numbers.foldLeft(0)((a, c) => a + c) /
  ↪numbers.length

  def averageV1(numbers: Int*) : Double = numbers.sum / numbers.length

  println(average(2,2))
  println(average(1,2,3))
  println(averageV1(1,2,3))
}
```

Oto dane wyjściowe wygenerowane przez przedstawioną funkcję:

```
2.0
2.0
2.0
```

Spójrz na pierwsze wywołanie funkcji `average()` — oczekuje zmiennej liczby argumentów typu `Int`. W tym przypadku została wywołana z argumentami `2` i `2`. Całkowita liczba argumentów wyniosła 2. Do przeprowadzenia operacji można dostarczyć dowolną liczbę argumentów. W bloku funkcji została użyta operacja `fold()` odpowiedzialna za obliczenie sumy wszystkich przekazanych liczb. Więcej informacji na temat sposobu działania operacji `fold()` znajdziesz w następnym rozdziale, przy okazji omawiania funkcji działających wraz z kolekcjami. Teraz wystarczy jedynie wiedzieć, że ta operacja przeprowadza iterację przez

wszystkie elementy kolekcji i wykonuje pewne działania wraz z dostarczonym argumentem, którym tutaj jest 0. Funkcja jest wywoływana z różną liczbą argumentów. W ten sam sposób można zdefiniować funkcję do obsługi różnej liczby argumentów dowolnych typów. Funkcję trzeba będzie odpowiednio wywołać. Jedynym wymaganiem jest to, że parametr *vararg* musi być podany jako ostatni na liście parametrów w sygnaturze funkcji.

```
def averageV1(numbers: Int*, wrongArgument: Int): Double = numbers.sum /
↪numbers.length
```

W pokazanej tutaj sygnaturze `numbers` przedstawia parametr o zmiennej liczbie przyjmowanych argumentów. Powinien być więc zadeklarowany jako ostatni i dlatego umieszczenie po nim parametru `wrongArgument` powoduje wygenerowanie *błędu w trakcie kompilacji*.

Wywołanie funkcji wraz z wartością domyślną parametru

W trakcie deklarowania funkcji dozwolone jest dostarczenie wartości domyślnej dla parametru. Jeżeli się na to zdecydujesz, możesz uniknąć przekazywania argumentu dla tego parametru w trakcie wywoływania funkcji. Pokażę takie rozwiązanie na przykładzie. Mogłeś to zobaczyć w przykładzie funkcji porównującej dwie liczby całkowite. Teraz drugiemu parametrowi zostanie przypisana wartość domyślna wynosząca 10.

```
def compareIntegersV6(value1: Int, value2: Int = 10): String = {
  println("Wykonywanie funkcji V6")

  def giveAMeaningFullResult(result: Int) = result match {
    case 0 => "Wartości są równe"
    case -1 => s"Wartość $value1 jest mniejsza niż $value2"
    case 1 => s"Wartość $value1 jest większa niż $value2"
    case _ => "Nie można przeprowadzić operacji"
  }

  val result = if (value1 == value2) 0 else if (value1 > value2) 1 else -1
  giveAMeaningFullResult(result)
}

println(compareIntegersV6(12))
```

Oto dane wyjściowe wygenerowane przez przedstawiony program:

```
Wykonywanie funkcji V6
Wartość 12 jest większa niż 10
```

Podczas deklarowania funkcji `compareIntegersV6()` jej parametr `value2` otrzymał wartość domyślną wynoszącą 10. Następnie funkcja została wywołana z tylko jednym argumentem.

```
compareIntegersV6(12)
```

Podczas wywołania funkcji był przekazany tylko jeden argument — 12 — dostarczający wartość dla parametru `value1`. W takiej sytuacji kompilator Scali szuka wartości dołączonej do drugiego parametru. W omawianym przykładzie kompilator był w stanie ustalić, że drugi parametr ma zdefiniowaną wartość domyślną 10. Dlatego też funkcja została wywołana wraz

z tymi dwoma wartościami. Zdefiniowanie wartości domyślnej i jej użycie będzie działało tylko wtedy, gdy kompilator Scali ma możliwość ustalenia wartości. W przypadku niejasności nie pozwoli na wywołanie funkcji. Spójrz na przedstawiony tutaj przykład.

```
def compareIntegersV6(value1: Int = 10, value2: Int) = ???
```

Spróbujmy teraz w następujący sposób wywołać funkcję `compareIntegersV6()`:

```
println(compareIntegersV6(12)) // Kompilator nie pozwoli na wywołanie.
```

Jeśli spróbujesz wywołać funkcję w pokazany sposób, kompilator Scali wygeneruje komunikat błędu, ponieważ z powodu kolejności argumentów nie potrafi dołączyć wartości `12` do parametru `value2`. Jeżeli w jakikolwiek sposób będzie można poinformować kompilator, że przykazywany argument powinien być dołączony do parametru o nazwie `value2`, funkcja zadziała. Rozwiązaniem jest przekazywanie argumentów za pomocą nazw podczas wywołania funkcji.

Wywoływanie funkcji wraz z nazwanymi argumentami

Podczas wywoływania funkcji można bezpośrednio używać nazw jej argumentów. Zapewnia to dowolność w kolejności podawania przekazywanych argumentów. Spójrz na przykład takiego rozwiązania.

```
def compareIntegersV6(value1: Int = 10, value2: Int): String = {
  println("Wykonywanie funkcji V6")

  def giveAMeaningFullResult(result: Int) = result match {
    case 0 => "Wartości są równe"
    case -1 => s"Wartość $value1 jest mniejsza niż $value2"
    case 1 => s"Wartość $value1 jest większa niż $value2"
    case _ => "Nie można przeprowadzić operacji"
  }

  val result = if (value1 == value2) 0 else if (value1 > value2) 1 else -1
  giveAMeaningFullResult(result)
}

println(compareIntegersV6(value2 = 12))
```

Oto dane wyjściowe wygenerowane przez przedstawiony program:

```
Wykonywanie funkcji V6
Wartość 10 jest mniejsza niż 12
```

Rozwiązanie to działa z prostego względu: kompilatorowi Scali trzeba zapewnić możliwość ustalenia wartości przeznaczonych do dołączenia parametrom. Poza tym zyskujemy dowolność w kolejności przekazywania argumentów, niezależnie od kolejności, w jakiej zostały podane w sygnaturze funkcji. Dlatego też tę funkcję można wywołać w następujący sposób:

```
println(compareIntegersV6(value2 = 12, value1 = 10))
```

Oto dane wyjściowe wygenerowane przez przedstawione wywołanie:

```
Wykonywanie funkcji V6
Wartość 10 jest mniejsza niż 12
```

W ten sposób dowiedziałeś się, że istnieje więcej niż tylko jeden sposób na zdefiniowanie i wywołanie funkcji. Dobrą wiadomością jest również to, że masz możliwość przekazywania funkcji w postaci literału. Nazywamy je wówczas literałami funkcji. W kolejnym podrozdziale nieco dokładniej omówię temat literału funkcji.

Literał funkcji

Funkcję można przekazać innej, wykorzystując do tego postać literału. Pokażę to na przykładzie tej samej funkcji `compareIntegers()`, której używaliśmy wcześniej w tym rozdziale.

```
def compareIntegersV6(value1: Int = 10, value2: Int): Int = ???
```

Wiemy, na czym ma polegać działanie naszej funkcji: ma to być pobranie danych wejściowych w postaci dwóch liczb całkowitych i zwrot liczby całkowitej określającej wynik operacji porównania. Jeżeli spojrzysz na abstrakcyjną formę funkcji, zobaczysz, że przedstawia się ona następująco:

```
(value1: Int, value2: Int) => Int
```

Oznacza to, że funkcja oczekuje podania dwóch liczb całkowitych i zwraca także liczbę całkowitą — wymagania pozostały więc niezmienione. W tej abstrakcyjnej postaci elementy po lewej stronie operatora przedstawiają dane wejściowe, natomiast po prawej stronie dane wyjściowe funkcji. Można powiedzieć, że taka postać to **literał funkcji**. Dlatego też można go przypisać dowolnej zmiennej:

```
val compareFuncLiteral = (value1: Int, value2: Int) => if (value1 == value2) 0
↪else if (value1 > value2) 1 else -1
```

Przypomnij sobie z poprzedniego rozdziału, że obiekt `PagePrinter` zawierał funkcję o nazwie `print()` pobierającą numer strony i drukującą ją.

```
private def print(index: Int) = println(s"Wydruk strony nr $index.")
```

Gdy spojrzysz na tę funkcję, zobaczysz, że pobiera liczbę całkowitą i drukuje podane strony. Tak więc funkcja ma następującą postać:

```
(index: Int) => Unit
```

Słowo kluczowe `Unit` przedstawia tutaj literał, który nie zwraca żadnej wartości. Rozważmy teraz sytuację, w której trzeba poinformować drukarkę, czy strona ma zostać wydrukowana w kolorze, czy ma pozostać czarno-biała. Omawiany kod można poddać refaktoryzacji, aby zapewnić obsługę użycia literału funkcji.

```
object ColorPrinter extends App {

  def printPages(doc: Document, lastIndex: Int, print: (Int) => Unit) = {
    if(lastIndex <= doc.numOfPages) for(i <- 1 to lastIndex) print(i)
  }

  val colorPrint = (index: Int) => println(s"Wydruk kolorowej strony nr $index.")
  val simplePrint = (index: Int) => println(s"Wydruk czarno-białej strony nr $index.")

  println("---------Metoda V1-----------")
  printPages(Document(15, "DOCX"), 5, colorPrint)

   println("---------Metoda V2-----------")
   printPages(Document(15, "DOCX"), 2, simplePrint)
}

case class Document(numOfPages: Int, typeOfDoc: String)
```

Oto dane wyjściowe wygenerowane przez przedstawiony program:

```
---------Metoda V1-----------
Wydruk kolorowej strony nr 1
Wydruk kolorowej strony nr 2
Wydruk kolorowej strony nr 3
Wydruk kolorowej strony nr 4
Wydruk kolorowej strony nr 5
---------Metoda V2-----------
Wydruk czarno-białej strony nr 1
Wydruk czarno-białej strony nr 2
```

Po przeprowadzeniu refaktoryzacji metody `printPages()` pobiera ona teraz *literał funkcji*, który przedstawia postać wspomnianej wcześniej funkcji `print()`. Zaprezentowałem dwie postaci funkcji `print()`: pierwszą przeznaczoną do wydruku stron w *kolorze*, zaś drugą do wydruku stron *czarno-białych*. Dzięki temu można wywołać tę samą funkcję `printPages()` i przekazać jej niezbędny *literał funkcji*. Trzeba jedynie poinformować funkcję `printPages()` o formie przekazywanej jej funkcji, a następnie w trakcie wywołania `printPages()` przekazać *literał funkcji* w podanej wcześniej postaci.

W Scali używa się również *literałów funkcji* w konstrukcjach domyślnych. Jednym z przykładów może tutaj być funkcja `filter()` współpracująca z kolekcjami. Funkcja ta oczekuje predykatu sprawdzającego warunek i udzielającego odpowiedzi w postaci wartości boolowskiej na podstawie filtrowanych elementów listy lub kolekcji.

```
scala> val names = List("Alicja","Aleksy","Bartek","Celina","Aleksander")
names: List[String] = List(Alicja, Aleksy, Bartek, Celina, Aleksander)

scala> val nameStartsWithA = names.filter((name) => name.startsWith("A"))
nameStartsWithA: List[String] = List(Alicja, Aleksy, Aleksander)
```

Fragment, w którym następuje sprawdzenie, czy imię rozpoczyna się na literę *A*, jest *literałem funkcji*.

```
(name) => name.startsWith("A")
```

Kompilator Scali wymaga informacji dodatkowych tylko wtedy, gdy musi przeprowadzić inferencję typu. Pozostałe fragmenty będące tylko składnią dodatkową można pominąć. Dlatego też wcześniej przedstawiony kod źródłowy można zapisać w następującej postaci:

```
scala> val nameStartsWithA = names.filter(_.startsWith("A"))
nameStartsWithA: List[String] = List(Alicja, Aleksy, Aleksander)
```

W pokazanym tutaj fragmencie kodu pominąłem nazwę parametru i użyłem składni *miejsca zarezerwowanego*. Co można zrobić w sytuacji, gdy literał funkcji ma zostać przekazany jako argument i wykonany tylko w razie potrzeby? Przykładem może być predykat wykonywany, gdy dana funkcjonalność jest aktywna. W takim przypadku można przekazać parametr jako nazwany. Scala oferuje tę możliwość w postaci parametrów *wywoływanych po nazwie*. Parametry te są używane z opóźnieniem, czyli dopiero wtedy, gdy trzeba lub w trakcie pierwszego ich wywołania. W następnym podrozdziale przedstawię wybrane strategie wywoływania funkcji.

Strategie wywoływania funkcji

Gdy funkcja ma pewne zdefiniowane parametry, w trakcie wywołania oczekuje przekazania argumentów. Teraz już wiesz o możliwości przekazania *literału funkcji* wykonywanego w trakcie wywołania funkcji lub pierwszego użycia literału. Scala pozwala również na *wywoływanie po wartości* i *wywoływanie po nazwie*. Przeanalizuję je dokładnie w kolejnych punktach.

Wywoływanie po nazwie

Wywołanie po nazwie to strategia, w której zastąpienie literału następuje w miejscu wywołania funkcji. Literał zostaje wówczas wykonany i następuje obliczenie jego wartości. Spróbuję to wyjaśnić na prostym przykładzie. Powrócę do naszej przykładowej aplikacji `ColorPrinter` — przyjmuję założenie, że istnieje w niej funkcja boolowska sprawdzająca, czy drukarka jest włączona. Oto sposób, na jaki można przeprowadzić refaktoryzację funkcji.

```
def printPages(doc: Document, lastIndex: Int, print: (Int) => Unit, isPrinterOn:
↪() => Boolean) = {
  if(lastIndex <= doc.numOfPages && isPrinterOn()) for(i <- 1 to lastIndex) print(i)
}
```

Aby wywołać funkcję, można użyć następującego polecenia:

```
printPages(Document(15, "DOCX"), 16, colorPrint, () => !printerSwitch)
```

Jednak z pokazanym tutaj podejściem wiążą się dwa problemy. Oto pierwszy: z powodu użycia konstrukcji `() => wyrażenie` ten kod wygląda dziwnie jak na literał funkcji zwracającej wartość boolowską. A to drugi: chcemy obliczyć wartość wyrażenia dopiero w chwili jego użycia. Dlatego też trzeba wprowadzić małą zmianę w sygnaturze funkcji `printPages()`.

```
object ColorPrinter extends App {

  val printerSwitch = false

  def printPages(doc: Document, lastIndex: Int, print: (Int) => Unit, isPrinterOn:
  ↪=> Boolean) = {
    if(lastIndex <= doc.numOfPages && isPrinterOn) for(i <- 1 to lastIndex) print(i)
  }

  val colorPrint = (index: Int) => {
    println(s"Wydruk kolorowej strony nr $index.")
  }

  println("---------Metoda V1-----------")
  printPages(Document(15, "DOCX"), 2, colorPrint, !printerSwitch)
}

case class Document(numOfPages: Int, typeOfDoc: String)
```

Oto dane wyjściowe wygenerowane przez przedstawiony program:

```
---------Metoda V1-----------
Wydruk kolorowej strony nr 1
Wydruk kolorowej strony nr 2
```

Jeżeli przyjrzysz się dokładnie temu kodu źródłowemu, zauważysz usunięcie nawiasu `()` i dodanie operatora `=>` w sygnaturze funkcji. To spowodowało, iż wskazaliśmy, że to jest parametr wywoływany *po nazwie* i wykonywany tylko po wywołaniu. Dlatego też teraz można użyć następującego wywołania funkcji:

```
printPages(Document(15, "DOCX"), 2, colorPrint, !printerSwitch)
```

Ostatnim argumentem tego wywołania jest wyrażenie boolowskie. Ponieważ nasza funkcja jest typem wywoływanym *po nazwie*, obliczenie wartości wyrażenia nastąpi dopiero po faktycznym wywołaniu funkcji.

Wywołanie po wartości

Wywołanie po wartości jest prostą i dość często stosowaną strategią, w której wyrażenie jest obliczane, a wynik zostaje dołączony do parametru. W miejscu użycia parametru zostaje on po prostu zastąpiony obliczoną wartością. Dotychczas miałeś okazję zobaczyć wiele przykładów tej strategii.

```
def compareIntegers(value1: Int, value2: Int): Int =
       if (value1 == value2) 0 else if (value1 > value2) 1 else -1

compareIntegers(10, 8)
```

Wywołania tej funkcji są przykładami strategii *wywołania po wartości*. Wartość jest po prostu przekazywana jako argument, który następnie zostaje zastąpiony przez wartość parametru.

Taka strategia daje wiele możliwości w zakresie wywołania funkcji. Ponadto obliczenie wartości wyrażenia dopiero wtedy, gdy ta wartość jest niezbędna, to cecha charakterystyczna języków programowania funkcyjnego. Nosi ona nazwę *obliczania z opóźnieniem*. Więcej informacji na ten temat znajdziesz w rozdziale 9. przy okazji omawiania oferujących potężne możliwości konstrukcji programowania funkcyjnego.

Programowanie funkcyjne obsługuje możliwość tworzenia funkcji, które są prawidłowe, i wykorzystania ich do pobierania danych wejściowych zamiast wygenerowania błędu. Scala oferuje również obsługę tak zwanych funkcji częściowych.

Funkcja częściowa

Funkcja częściowa nie nadaje się do użycia w przypadku każdych danych wejściowych. Jest definiowana jako przeznaczona do obsługi określonego zbioru parametrów danych wejściowych. Aby to lepiej zrozumieć, najpierw spójrz na przykład funkcji częściowej.

```
scala> val oneToFirst: PartialFunction[Int, String] = {
     | case 1 => "Jeden"
     | }
oneToFirst: PartialFunction[Int, String] = <function1>

scala> println(oneToFirst(1))
Jeden
```

W przedstawionym tutaj fragmencie kodu najpierw zdefiniowałem funkcję częściową o nazwie `oneToFirst()` wraz z typami parametrów, którymi są `Int` i `String`. W Scali `PartialFunction` to cecha zdefiniowana w następujący sposób:

```
trait PartialFunction[-A, +B] extends (A) => B
```

Cecha wskazuje, że oczekiwane są dwa parametry `A` i `B`, które stają się typami danych wejściowych i wyjściowych w naszej funkcji częściowej. W omawianym przypadku funkcja częściowa po prostu oczekuje wartości `1` i zwraca ciąg tekstowy wyrażający tę wartość. Dlatego też próba jej wywołania wraz z wartością `1` działa doskonale, natomiast po przekazaniu innego argumentu, na przykład `2`, następuje wygenerowanie błędu `MatchError`.

```
scala> println(oneToFirst(2))
scala.MatchError: 2 (of class java.lang.Integer)
  at scala.PartialFunction$$anon$1.apply(PartialFunction.scala:254)
  at scala.PartialFunction$$anon$1.apply(PartialFunction.scala:252)
  at $anonfun$1.applyOrElse(<console>:12)
  at $anonfun$1.applyOrElse(<console>:11)
  at scala.runtime.AbstractPartialFunction.apply(AbstractPartialFunction.scala:34)
```

Omawiana funkcja częściowa jest odpowiednia tylko dla jednej wartości, tutaj 1, a dla innych już nie. Aby zagwarantować, że funkcja częściowa nie spowoduje zgłoszenia błędu, można za pomocą metody `isDefinedAt()` wcześniej sprawdzić, czy dana wartość jest odpowiednia dla funkcji.

```
scala> oneToFirst.isDefinedAt(1)
res3: Boolean = true

scala> oneToFirst.isDefinedAt(2)
res4: Boolean = false
```

W przypadku wartości obsługiwanych przez funkcję częściową metoda `isDefinedAt()` zwraca wartość `true`, natomiast dla pozostałych wartości zwraca `false`. Funkcja częściowa może być również złożona. W takiej sytuacji cecha `PartialFunction` definiuje dwie metody, czyli `orElse()` i `andThen()`.

```
object PartialFunctions extends App {

  val isPrimeEligible: PartialFunction[Item, Boolean] = {
    case item => item.isPrimeEligible
  }

  val amountMoreThan500: PartialFunction[Item, Boolean] = {
    case item => item.price > 500.0
  }

  val freeDeliverable = isPrimeEligible orElse amountMoreThan500

  def deliveryCharge(item: Item): Double = if(freeDeliverable(item)) 0 else 50

  println(deliveryCharge(Item("1", "Klawiatura ABC", 490.0, false)))
}

case class Item(id: String, name: String, price: Double, isPrimeEligible: Boolean)
```

Oto dane wyjściowe wygenerowane przez przedstawiony program:

```
50.0
```

W tym fragmencie kodu zdefiniowałem funkcje częściowe o nazwach `isPrimeEligle()` i `amountMoreThan500()`. Następnie przygotowałem kolejną funkcję częściową za pomocą metody `orElse()` sprawdzającej, czy dany produkt kwalifikuje się do bezpłatnej wysyłki. Funkcje częściowe pozwalają na przygotowywanie i definiowanie funkcji przeznaczonych do określonego zadania wykonywanego na zbiorze wartości. Ponadto funkcja częściowa zapewnia możliwość zdefiniowania logiki oddzielnie od danego zbioru wartości danych wejściowych, gdyż opiera się na pewnych różnicach. Trzeba koniecznie pamiętać, że funkcja częściowa działa tylko jako jeden operand. Stąd przybiera postać funkcji jednoargumentowej i do programisty należy odpowiedzialność za sprawdzenie, czy określona wartość jest odpowiednia dla danej funkcji.

Podsumowanie

Czas na krótkie podsumowanie tego, czego nauczyłeś się w tym rozdziale. Przedstawiłem krótkie wprowadzenie do bardzo ważnej koncepcji w języku Scala, czyli *funkcji*. Na początku poznałeś składnię definicji funkcji. Warto w tym miejscu przypomnieć o możliwości zagnieżdżania funkcji, dzięki czemu kod staje się czytelniejszy. Dowiedziałeś się, jak można wywoływać funkcję na różne sposoby, na przykład ze zmienną liczbą argumentów, wraz z wartością domyślną parametru lub z argumentem nazwanym. Następnie przeszedłem do literałów funkcji w Scali. Dalej omówiłem różne strategie wywoływania funkcji w Scali, między innymi *wywołanie po nazwie* i *wywołanie po wartości*. Na końcu omówiłem inną ważną koncepcję w Scali, czyli *funkcję częściową*.

Poznanie wszystkich omówionych dotąd koncepcji niewątpliwie zwiększyło Twoje umiejętności w zakresie tworzenia i zrozumienia fragmentów kodu Scali. W dalszych rozdziałach będę kontynuował omawianie kolejnych koncepcji. W następnym rozdziale poznasz wiele kolekcji oferowanych przez Scalę i różne metody pozwalające na pracę z tymi kolekcjami.

Skorowidz

G

H

I

J

K

L

Ł

M

N

O

P

R

S

Ś

T

W

Z

Scala: nowoczesność i prostota w każdej skali!

Scala jest językiem programowania ogólnego przeznaczenia, który łączy cechy języków funkcyjnych i obiektowych. Jego twórcy postawili sobie za cel skalowalność napisanych w nim aplikacji — stąd wzięła się nazwa Scala. Scala jest oparta na wirtualnej maszynie Javy, umożliwia programowanie funkcyjne i oferuje bogaty wybór typów. Dzięki tym cechom można tworzyć kod mniej podatny na występowanie błędów w trakcie działania programu. Po uzyskaniu pewnej biegłości w kodowaniu praca z tym językiem staje się prawdziwą przyjemnością, nawet podczas tworzenia frameworków czy bibliotek.

Ten przystępny podręcznik przeznaczono dla programistów, którzy chcą poznać język Scala, aby wykorzystywać go do pisania współbieżnych, skalowalnych i reaktywnych aplikacji. Przedstawiono tu podstawy niezbędne do rozpoczęcia kodowania w Scali: składnię języka, podstawowe typy danych, literały czy zmienne. Następnie omówiono struktury danych w Scali i sposoby korzystania z funkcji wyższego rzędu. Zaprezentowano również takie koncepcje jak dopasowanie wzorca, klasy przypadku oraz zagadnienia związane z programowaniem funkcyjnym i programowaniem zorientowanym obiektowo. Opisano techniki programowania asynchronicznego i reaktywnego. Znalazło się tu także obszerne wprowadzenie do frameworka Akka.

W tej książce między innymi:

- paradygmaty programowania a korzystanie ze Scali
- kolekcje modyfikowalne i niemodyfikowalne
- koncepcja domniemania i praca z wyjątkami
- programowanie równoległe, asynchroniczne i reaktywne
- programowanie sterowane testami w Scali

Vikash Sharma — urodził się w Indiach. Jest zapalonym programistą i gorącym propagatorem idei open source. Uważa, że zachowanie prostoty podczas projektowania oprogramowania pomaga w tworzeniu przejrzystego i łatwego w późniejszej obsłudze kodu. Poświęcił dużo czasu na implementowanie kodu w Scali; z myślą o innych programistach przygotował kurs wideo programowania w tym języku. Obecnie pracuje jako programista w SAP Labs.

ISBN 978-83-283-4796-0
9 788328 347960
Cena: 67,00 zł
Packt