Joshua Suereth D.

Scala

od podszewki

POZNAJ | WYKORZYSTAJ POTĘGĘ PROGRAMOWANIA FUNKCYJNEGO!

Helion

Tytuł oryginału: Scala in Depth

Tłumaczenie: Justyna Walkowska

Projekt okładki: Anna Mitka
Materiały graficzne na okładce zostały wykorzystane za zgodą Shutterstock Images LLC.

ISBN: 978-83-246-5188-7

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)

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

Printed in Poland.

Spis treści

5 Domniemane wartości i widoki podstawą ekspresywnego kodu

W tym rozdziale:

- wprowadzenie do domniemanych parametrów i widoków,
- mechanizm odnajdywania wartości domniemanych,
- wykorzystanie domniemanych konwersji do rozszerzania klas,
- ograniczanie zakresu.

System domniemanych parametrów i konwersji w Scali pozwala kompilatorowi na przetwarzanie kodu na bazie dobrze zdefiniowanego mechanizmu wyszukiwania. Programista może pominąć część informacji, a kompilator spróbuje wywnioskować je w czasie kompilacji. Takie wnioskowanie jest możliwe w jednej z dwóch sytuacji:

- przy wywołaniu metody lub konstruktora bez podania któregoś z parametrów,
- przy domniemanej konwersji pomiędzy typami (domniemanym widoku) — dotyczy to także obiektu, na którym jest wywoływana metoda.

W obu przypadkach kompilator stosuje zestaw reguł w celu pozyskania brakujących informacji, by możliwa była kompilacja kodu. Możliwość pominięcia parametrów jest niesamowicie przydatna. Często wykorzystują ją biblioteki Scali. Bardziej kontrowersyjne,

lub wręcz niebezpieczne, jest zmienianie typu przez kompilator w celu umożliwienia poprawnej kompilacji.

System domniemań to jeden z największych atutów Scali. Rozsądnie stosowany może radykalnie zmniejszyć rozmiar Twojego kodu. Może także zostać wykorzystany do eleganckiego wymuszenia ograniczeń projektowych.

Spójrzmy najpierw na domniemane parametry.

5.1. Słowo kluczowe implicit

Scala udostępnia słowo kluczowe `implicit` (z ang. domniemany, niejawny), które można stosować na dwa sposoby: podczas definiowania metod lub zmiennych albo na liście parametrów metody. Użyte w definicji metody lub zmiennej słowo to informuje kompilator o tym, że dana metoda lub zmienna może zostać wykorzystana podczas wnioskowania na temat domniemań. Wyszukiwanie wartości domniemanych jest przeprowadzane, gdy kompilator zauważa, że w kodzie brakuje pewnej informacji. Jeśli słowo `implicit` zostanie użyte na początku listy parametrów pewnej metody, kompilator przyjmie, że ta lista może nie zostać podana i że konieczne może się okazać odgadnięcie parametrów na podstawie reguł.

Przeanalizujmy mechanizm wnioskowania na przykładzie metody z brakującą listą parametrów:

```
scala> def findAnInt(implicit x : Int) = x
findAnInt: (implicit x: Int)Int
```

Metoda `findAnInt` (znajdź liczbę całkowitą) deklaruje jeden parametr `x` typu `Int`. Zwróci ona bez zmian każdą przekazaną do niej wartość. Lista parametrów została oznaczona słowem `implicit`, co oznacza, że nie jest konieczne jej podawanie. Jeśli opuścimy listę parametrów, kompilator poszuka zmiennej typu `Int` w zakresie domniemanym. Oto przykład wywołania tej metody:

```
scala> findAnInt
<console>:7: error: could not find implicit value for parameter x: Int
       findAnInt
       ^
```

Metoda `findAnInt` została wywołana bez listy parametrów. Kompilator skarży się, że nie jest w stanie odnaleźć wartości domniemanej parametru `x`. Dostarczmy mu taką wartość:

```
scala> implicit val test = 5
test: Int = 5
```

Wartość `test` została zdefiniowana z użyciem słowa `implicit`. Oznacza to, że może ona zostać uwzględniona we wnioskowaniu na temat domniemań. Skoro działamy w środowisku REPL, zmienna `test` będzie dostępna aż do końca naszej sesji. Oto co się wydarzy, gdy wywołamy `findAnInt`:

```
scala> findAnInt
res3: Int = 5
```

Tym razem wywołanie się powiedzie — metoda zwróci wartość zmiennej `test`. Kompilatorowi udało się odnaleźć brakujące fragmenty układanki. Oczywiście jeśli chcemy, możemy wywołać funkcję, przekazując jej parametr:

```
scala> findAnInt(2)
res4: Int = 2
```

Ponieważ tym razem nie brakuje parametru, kompilator nie rozpoczyna procesu wyszukiwania wartości na podstawie reguł. Zapamiętaj, że domniemane parametry zawsze można jawnie podać. Wrócimy do tego w podrozdziale 5.6.

W celu zrozumienia sposobu, w jaki kompilator określa, czy zmienna może zostać uwzględniona w procesie wyszukiwania wartości domniemanych, trzeba wgryźć się trochę w to, jak są obsługiwane identyfikatory i zakresy.

5.1.1. Identyfikatory (dygresja)

Zanim zagłębimy się w szczegóły mechanizmu wnioskowania, warto zrozumieć, w jaki sposób kompilator rozpoznaje identyfikatory w danym zakresie. Ta sekcja jest oparta na rozdziale 2. specyfikacji języka Scala[1]. Zachęcam Cię do przeczytania specyfikacji, gdy już zapoznasz się z podstawami. Identyfikatory odgrywają kluczową rolę podczas wybierania zmiennych domniemanych, dlatego poświęćmy im nieco uwagi.

W specyfikacji pojawia się słowo *entity* (z ang. encja, byt), obejmujące znaczeniem: typy, wartości, zmienne oraz klasy. Są to podstawowe elementy używane do budowania programów. Odwołujemy się do nich za pomocą identyfikatorów czy też nazw. Mówimy wówczas o wiązaniu (ang. *binding*) pomiędzy identyfikatorem a daną encją. Rozważ następujący fragment kodu:

```
class Foo {
  def val x = 5
}
```

Sama encja `Foo` to klasa zawierająca metodę `x`. Powiązaliśmy ją z identyfikatorem `Foo`. Jeśli zadeklarujemy tę klasę lokalnie wewnątrz REPL, będziemy mogli utworzyć jej instancję, ponieważ nazwa i encja zostały lokalnie powiązane:

```
scala> val y = new Foo
y: Foo = Foo@33262bf4
```

Możemy utworzyć nową zmienną o nazwie `y` i typie `Foo`, odwołując się do nazwy `Foo`. Powtórzę: jest tak dlatego, że klasa `Foo` została zdefiniowana lokalnie wewnątrz REPL i lokalnie powiązano ją z nazwą `Foo`. Skomplikujmy nieco sprawy, umieszczając `Foo` wewnątrz pakietu.

```
package test;

class Foo {
  val x = 5
}
```

Klasa `Foo` należy teraz do pakietu `test`. Jeśli spróbujemy odwołać się do niej w REPL za pomocą nazwy `Foo`, poniesiemy klęskę:

```
scala> new Foo
<console>:7: error: not found: type Foo
       new Foo
```

Utworzenie nowej instancji `Foo` nie powiodło się, ponieważ w naszym zakresie nazwa `Foo` nie została powiązana z żadną encją. Klasa `Foo` znajduje się w pakiecie `test`. Aby się do

[1] *http://www.scala-lang.org/docu/files/ScalaReference.pdf*.

niej odwołać, musimy albo skorzystać z nazwy `test.Foo`, albo powiązać nazwę `Foo` z klasą `test.Foo` w aktualnym zakresie. Druga z wspomnianych opcji jest dostępna dzięki słowu kluczowemu `import`:

```
scala> import test.Foo
import test.Foo

scala> new Foo
res3: test.Foo = test.Foo@60e1e567
```

Instrukcja `import` pobiera encję `test.Foo` i wiąże ją z nazwą `Foo` w zakresie lokalnym. Dzięki temu możliwe staje się utworzenie instancji `test.Foo` za pomocą wywołania `new Foo`. Podobnie działa instrukcja `import` w Javie i `using` w C++. Mechanizm dostępny w Scali jest jednak nieco bardziej elastyczny.

Instrukcję `import` można zastosować w dowolnym miejscu pliku źródłowego, gdzie stworzy ona wiązanie jedynie w zakresie lokalnym. Dzięki temu mamy kontrolę nad tym, gdzie są używane zaimportowane nazwy. Ta funkcjonalność pozwala dodatkowo na ograniczenie zakresu domniemanych widoków i zmiennych. Więcej na ten temat znajdziesz w podrozdziale 5.4.

Jednym z przejawów elastyczności mechanizmu wiązania encji w Scali jest możliwość wykorzystania arbitralnych nazw. W Javie czy w C# jest możliwe jedynie przeniesienie do bieżącego zakresu nazwy zdefiniowanej w innym zakresie bądź pakiecie. Klasę `test.Foo` moglibyśmy zaimportować lokalnie jako `Foo`. Natomiast instrukcja `import` w Scali pozwala na zdefiniowanie nowej nazwy przy użyciu składni `{OryginalneWiązanie => NoweWiązanie}`. Zaimportujmy naszą encję `test.Foo`, nadając jej nową nazwę:

```
scala> import test.{Foo=>Bar}
import test.{Foo=>Bar}

scala> new Bar
res1: test.Foo = test.Foo@596b753
```

Pierwsza instrukcja `import` wiąże klasę `test.Foo` z nazwą `Bar` w bieżącym zakresie. W następnej linii tworzymy nową instancję `test.Foo`, wywołując `new Bar`. Mechanizm ten pozwala na uniknięcie konfliktów nazw podczas importowania encji z różnych pakietów. Dobrym przykładem są `java.util.List` i `scala.List`. W celu uniknięcia nieporozumień w kodzie współpracującym z kodem Javy często stosuje się konstrukcję `import java.util.{List=>JList}`.

Zmiana nazwy pakietu

Instrukcję `import` w Scali można zastosować także w celu zmiany nazwy pakietu. Przydaje się to podczas korzystania z bibliotek Javy. Sam, gdy korzystam z pakietu `java.io`, często zaczynam od następującego kodu:

```
import java.{io=>jio}
def someMethod( input : jio.InputStream ) = ...
```

Wiązanie pozwala na nadanie encji określonej nazwy w konkretnym zakresie. Istotne jest tutaj zrozumienie, czym jest zakres i jakie wiązania można w nim znaleźć.

5.1.2. Zakres i wiązania

Zakres to leksykalna granica, wewnątrz której są dostępne wiązania. Zakresem może być ciało klasy, ciało metody, anonimowy blok kodu… Zasadniczo za każdym razem, gdy używasz nawiasów klamrowych, tworzysz w ich wnętrzu nowy zakres.

W Scali jest możliwe zagnieżdżanie zakresów — jeden zakres może wystąpić wewnątrz drugiego. W zagnieżdżonym zakresie są dostępne wiązania z zakresu szerszego. Możliwa jest zatem następująca operacja:

```
class Foo(x : Int) {
  def tmp = {
    x
  }
}
```

Konstruktor klasy `Foo` pobiera parametr `x`. Następnie definiujemy zagnieżdżoną metodę `tmp`. Parametry konstruktora są dostępne z jej wnętrza — możemy odwołać się do identyfikatora `x`. Zakres zagnieżdżony ma dostęp do wiązań w zakresie-rodzicu, ale możliwe jest też tworzenie wiązań, które je przesłonią. Metoda `tmp` może utworzyć nowe wiązanie o nazwie `x`. Wówczas `x` nie będzie już identyfikatorem prowadzącym do parametru rodzica. Zobaczmy:

```
scala> class Foo(x : Int) {
     |   def tmp = {
     |     val x = 2
     |     x
     |   }
     | }
defined class Foo
```

Klasa `Foo` ma definicję taką samą jak wcześniej, jednak metoda `tmp` w zakresie zagnieżdżonym definiuje zmienną o nazwie `x`. Nowe wiązanie *przesłania* parametr konstruktora `x`. W wyniku tego lokalnie widoczne jest tylko nowe wiązanie, a parametr konstruktora jest niedostępny — a przynajmniej nie przy użyciu nazwy `x`. W Scali wiązania o wyższym priorytecie przesłaniają te o niższym w tym samym zakresie. Ponadto wiązania o wyższym lub tym samym priorytecie przesłaniają wiązania zdefiniowane w zakresie zewnętrznym.

Priorytety wiązań w Scali są następujące:

1. Najwyższy priorytet mają definicje lub deklaracje lokalne, odziedziczone lub udostępnione poprzez klauzulę pakietową w tym samym pliku, w którym pojawia się definicja.
2. Następne w kolejności są encje jawnie zaimportowane.
3. Dalej mamy encje zaimportowane z użyciem symboli wieloznacznych (`import foo._`).
4. Najniższy priorytet mają definicje udostępniane poprzez klauzulę pakietową znajdującą się poza plikiem, w którym pojawia się definicja.

Przesłanianie wiązań

W Scali wiązanie przesłania wiązania o niższym priorytecie w tym samym zakresie. Ponadto wiązanie przesłania wiązania o tym samym lub niższym priorytecie z zakresu zewnętrznego. Dzięki temu możemy napisać:

```
class Foo(x : Int) {
  def tmp = {
    val x = 2
    x
  }
}
```

Metoda `tmp` będzie zwracała wartość `2`.

Sprawdźmy priorytety na konkretnym przykładzie. Zacznijmy od zdefiniowania pakietu `test` i obiektu `x` wewnątrz pliku źródłowego, który nazwiemy *externalbindings.scala* (listing 5.1).

Listing 5.1. Plik externalbindings.scala z wiązaniami zewnętrznymi

```
package test;

object x {
  override def toString = "Zewnętrznie powiązany obiekt x w pakiecie test"
}
```

Plik definiuje pakiet `test` oraz zawarty w nim obiekt `x`. Obiekt `x` przesłania metodę `toString`, dzięki czemu łatwo go rozpoznać. Zgodnie z przedstawionymi wcześniej regułami obiekt `x` powinien mieć najniższy możliwy priorytet wiązania. Stwórzmy teraz plik, który to sprawdzi (listing 5.2).

Listing 5.2. Test wiązania w tym samym pakiecie

```
package test;

object Test {
  def main(args : Array[String]) : Unit = {
    testSamePackage()       // Ten sam pakiet
    testWildcardImport()    // Import poprzez symbol wieloznaczny
    testExplicitImport()    // Jawny import
    testInlineDefinition() // Definicja w miejscu[JW1]
  }
  ...
}
```

Zaczynamy od deklaracji, zgodnie z którą treść pliku przynależy do tego samego pakietu co nasza wcześniejsza definicja. Następnie definiujemy metodę `main` wywołującą cztery metody testowe, po jednej dla każdej reguły określającej priorytety wiązań. Zacznijmy od zdefiniowania pierwszej z metod:

```
def testSamePackage() {
  println(x)
}
```

Metoda wypisuje encję o nazwie `x`. Ponieważ obiekt `Test` został zdefiniowany wewnątrz pakietu `test`, stworzony wcześniej obiekt `x` jest dostępny i to on zostanie przekazany metodzie `println`. Oto dowód:

```
scala> test.Test.testSamePackage()
Zewnętrznie powiązany obiekt x w pakiecie test
```

Wywołanie metody `testSamePackage` generuje łańcuch znaków związany z obiektem `x`. Spójrzmy teraz, co się zmieni, gdy do zaimportowania obiektu wykorzystamy symbol wieloznaczny (listing 5.3).

Listing 5.3. Import z użyciem symbolu wieloznacznego

```
object Wildcard {
  def x = "Import x poprzez symbol wieloznaczny"
}

def testWildcardImport() {
  import Wildcard._
  println(x)
}
```

Obiekt `Wildcard` przechowuje encję `x`. Encja `x` to metoda, która zwraca łańcuch znaków `"Import x poprzez symbol wieloznaczny"`. Metoda `testWildcardImport` najpierw wywołuje `import Wildcard._`. Dzięki temu wywołaniu nazwy i encje z obiektu `Wildcard` zostaną powiązane w bieżącym zakresie. Ponieważ importowanie za pomocą symbolu wieloznacznego ma wyższy priorytet niż zasoby dostępne w tym samym pakiecie, ale innym pliku źródłowym, encja `Wildcard.x` zostanie użyta zamiast `test.x`. Możemy to sprawdzić, wywołując funkcję `testWildcardImport`:

```
scala> test.Test.testWildcardImport()
Wildcard Import x
```

Wywołanie metody `testWildcardImport` powoduje wyświetlenie napisu `"Import x poprzez symbol wieloznaczny"` — właśnie tego spodziewaliśmy się, znając priorytety wiązań. Sprawy staną się bardziej interesujące, gdy do przykładu dorzucimy jeszcze jawne importowanie elementów (listing 5.4).

Listing 5.4. Jawny import

```
object Explicit {
  def x = "Jawny import x"
}
def testExplicitImport() {
  import Explicit.x
  import Wildcard._
  println(x)
}
```

Obiekt `Explicit` stanowi przestrzeń nazw dla kolejnej encji `x`. Metoda `testExplicitImport` najpierw importuje tę encję bezpośrednio, a potem importuje jeszcze zawartość encji obiektu `Wildcard`, korzystając z symbolu wieloznacznego. Chociaż import z użyciem symbolu wieloznacznego jest drugi w kolejności, działają tu reguły określające priorytety wiązań. Metoda zwróci wartość `x` z obiektu `Explicit`. Sprawdźmy:

```
scala> test.Test.testExplicitImport()
Jawny import x
```

Zgodnie z oczekiwaniami zwrócona została wartość `Explicit.x`. Widoczna tu reguła określająca priorytety wiązań ma duże znaczenie w kontekście wyszukiwania wartości domniemanych, do którego przejdziemy w sekcji 5.1.3.

Ostatnia reguła dotyczy deklaracji lokalnych. Zmieńmy metodę `testExplicitImport` tak, by definiowała lokalne wiązanie dla nazwy `x` (listing 5.5).

Listing 5.5. Definicja lokalna

```
def testInlineDefinition() {
  val x = "Lokalna definicja x"
  import Explicit.x
  import Wildcard._
  println(x)
}
```

Pierwsza linia metody `testInlineDefinition` to deklaracja zmiennej lokalnej `x`. Następnie linie w sposób jawny bądź domniemany (z wykorzystaniem symbolu wieloznacznego) importują wiązania `x` z obiektów `Explicit` i `Wildcard`, pokazanych wcześniej. W ostatniej linii drukujemy wynik, by przekonać się, które wiązanie zwyciężyło.

```
scala> test.Test.testInlineDefinition()
Lokalna definicja x
```

Ponownie, mimo że instrukcje `import` pojawiają się po instrukcji `val x`, wybór jest oparty na priorytecie, a nie na kolejności deklaracji.

Wiązania nieprzesłaniane

Jest możliwe stworzenie w tym samym zakresie dwóch wiązań o tej samej nazwie. W takim wypadku kompilator ostrzeże o dwuznaczności nazw. Oto przykład zapożyczony bezpośrednio ze specyfikacji języka Scala:

```
scala> {
 | val x = 1;
 | {
 | import test.x;
 | x
 | }
 | }
<console>:11: error: reference to x is ambiguous; it is both defined in
            value res7 and imported subsequently by import test.x
x
^
```

Zmienna `x` jest tu wiązana w zakresie zewnętrznym. Jest ona także importowana z pakietu `test` w zakresie zagnieżdżonym. Żadne z wiązań nie przesłania drugiego. Zmienna `x` z zakresu zewnętrznego nie może przesłonić zmiennej w zakresie zagnieżdżonym, a zaimportowana zmienna `x` również nie ma odpowiedniego priorytetu, by przesłonić tę w zakresie zewnętrznym.

Skąd taki nacisk na sposób rozwikływania nazw przez kompilator? Otóż wnioskowanie na temat domniemań jest ściśle powiązane z wnioskowaniem na temat nazw. Zawiłe reguły określające priorytety nazw mają znaczenie także w przypadku domniemań. Przyjrzyjmy się teraz, jak postępuje kompilator, napotykając niepełną deklarację.

5.1.3. Wyszukiwanie wartości domniemanych

Specyfikacja języka Scala deklaruje dwie reguły związane z wyszukiwaniem encji oznaczonych jako domniemane:

- Wiązanie encji domniemanej jest dostępne na stronie wyszukiwania bez prefiksu — to znaczy nie jako `foo.x`, tylko jako `x`.
- Jeśli pierwsza reguła nie prowadzi do rozwiązania problemu, to wszystkie składowe obiektu oznaczone jako `implicit` należą do domniemanego zakresu związanego z typem parametru domniemanego.

Pierwsza reguła ściśle łączy się z regułami wiązania przedstawionymi w poprzedniej sekcji. Druga jest nieco bardziej złożona. Przyjrzymy się jej dokładnie w sekcji 5.1.4.

Na początek wróćmy do przedstawionego już wcześniej przykładu wyszukiwania wartości domniemanych:

```
scala> def findAnInt(implicit x : Int) = x
findAnInt: (implicit x: Int)Int

scala> implicit val test = 5
test: Int = 5
```

Metoda `findAnInt` została zadeklarowana z oznaczoną jako `implicit` listą parametrów składającą się z jednej wartości typu całkowitego. Następnie definiujemy wartość `val test`, także oznaczoną jako `implicit`. Dzięki temu identyfikator, `test`, jest dostępny w zakresie lokalnym bez prefiksu. Jeśli w REPL wpiszemy `test`, otrzymamy wartość `5`. Jeśli wywołamy metodę, pisząc `findAnInt`, kompilator przepisze ją jako `findAnInt(test)`. Podczas wyszukiwania są wykorzystywane reguły wiązania, które zostały przeze mnie opisane wcześniej.

Druga reguła domniemanego wyszukiwania jest używana, gdy kompilator nie może znaleźć żadnej wartości domniemanej, stosując pierwszą z reguł. W takim wypadku kompilator spróbuje odnaleźć domniemane zmienne zdefiniowane wewnątrz dowolnego obiektu w domniemanym zakresie typu, którego szuka. **Domniemany zakres typu** definiuje się jako wszystkie moduły towarzyszące powiązane z danym typem. Oznacza to, że jeśli kompilator szuka parametru metody `def foo (implicit param : Foo)`, to parametr musi być zgodny z typem `Foo`. Jeśli pierwsza reguła nie zwróci wartości typu `Foo`, to kompilator sprawdzi *domniemany zakres* `Foo`. Domniemany zakres `Foo` to obiekt towarzyszący `Foo`.

Przeanalizuj kod na listingu 5.6.

Listing 5.6. Obiekt towarzyszący a wyszukiwanie zmiennych domniemanych

```
scala> object holder {
     | trait Foo
     | object Foo {
     |   implicit val x = new Foo {
```

```
     |     override def toString = "Obiekt towarzyszący Foo"
     |   }
     | }
     | }
defined module holder

scala> import holder.Foo
import holder.Foo

scala> def method(implicit foo : Foo) = println(foo)
method: (implicit foo: holder.Foo)Unit

scala> method
Obiekt towarzyszący Foo
```

Obiekt `holder` jest nam potrzebny do zdefiniowania cechy i obiektu towarzyszącego wewnątrz sesji REPL, podobnie jak robiliśmy to w sekcji 2.1.2. Wewnątrz niego definiujemy cechę `Foo` oraz obiekt towarzyszący `Foo`. Obiekt towarzyszący definiuje składową `x` typu `Foo`, udostępnianą na potrzeby wnioskowania na temat domniemań. Następnie importujemy typ `Foo` z obiektu `holder` do zakresu bieżącego. Ten krok nie jest wymagany, wykonujemy go w celu uproszczenia definicji metody. Następnie definiujemy metodę `method`. Pobiera ona domniemany parametr typu `Foo`.

Jeśli wywołamy metodę z pustą listą argumentów, kompilator użyje zdefiniowanej w obiekcie towarzyszącym zmiennej `implicit val x`.

Jako że zakres domniemany jest sprawdzany w drugiej kolejności, możemy wykorzystać go do przechowywania wartości domyślnych, a jednocześnie umożliwić użytkownikowi importowanie własnych wartości, jeśli jest mu to potrzebne. Poświęcimy temu zagadnieniu więcej uwagi w podrozdziale 7.2.

Jak wspominałem wcześniej, zakresem domniemanym dla typu `T` jest zbiór obiektów towarzyszących dla wszystkich typów powiązanych z typem `T` — czyli istnieje zbiór typów powiązanych z `T`. Wszystkie obiekty towarzyszące tym typom są przeszukiwane podczas wnioskowania na temat domniemań. Według specyfikacji języka typ powiązany z klasą `T` to każda klasa będąca klasą bazową pewnej części typu `T`. Poniższa lista przedstawia istniejące części typu `T`.

- Wszystkie podtypy `T` są częściami `T`. Jeśli typ `T` został zdefiniowany jako `A with B with C`, to `A`, `B` i `C` wszystkie są częściami `T`, zatem ich obiekty towarzyszące zostaną przeszukane, gdy konieczne będzie znalezienie domniemanej wartości typu `T`.
- Jeśli `T` ma parametry, to wszystkie parametry typu i ich części należą do zbioru części `T`. Przykładowo wyszukiwanie domniemanej wartości dla typu `List[String]` sprawdzi obiekt towarzyszący `List` i obiekt towarzyszący `String`.
- Jeśli `T` jest typem singletonowym `p.type`, to części typu `p` należą również do zbioru części typu `T`. Oznacza to, że jeśli typ `T` został zdefiniowany wewnątrz obiektu, to sam ten obiekt zostanie przeszukany pod kątem wartości domniemanych. Więcej na temat typów singletonowych znajdziesz w sekcji 6.1.1.
- Jeśli `T` jest projekcją typu `S#T`, to części `S` są także częściami `T`. Oznacza to, że jeśli typ `T` został zdefiniowany wewnątrz klasy lub cechy, to obiekty towarzyszące tej klasie lub cesze zostaną przeszukane pod kątem wartości domniemanych. Więcej na temat projekcji typów znajdziesz w sekcji 6.1.1.

Zakres domniemany typu obejmuje wiele różnych lokalizacji i zapewnia dużą elastyczność, jeśli chodzi o dostarczanie wartości domniemanych.

Przeanalizujmy teraz co ciekawsze aspekty zakresu domniemanego.

ZAKRES DOMNIEMANY POPRZEZ PARAMETRY TYPU

Zgodnie ze specyfikacją języka Scala zakres domniemany typu obejmuje wszystkie obiekty towarzyszące wszystkich typów i podtypów zawartych w parametrach typu. Oznacza to na przykład, że możemy zdefiniować wartość domniemaną dla `List[Foo]`, podając ją w obiekcie towarzyszącym `Foo`. Oto przykład:

```
scala> object holder {
     |   trait Foo
     |   object Foo {
     |     implicit val list = List(new Foo{})
     |   }
     | }
defined module holder

scala> implicitly[List[holder.Foo]]
res0: List[holder.Foo] = List(holder$Foo$$anon$1@2ed4a1d3)
```

Obiekt `holder` służy nam, tradycyjnie, do stworzenia obiektów stowarzyszonych wewnątrz REPL. Zawiera on cechę `Foo` i jej obiekt towarzyszący. Obiekt towarzyszący zawiera definicję `List[Foo]` oznaczoną słowem `implicit`. W następnej linii wywołujemy funkcję Scali o nazwie `implicitly`. Pozwoli ona na wyszukanie typu w aktualnym zakresie domniemanym. Definicja tej funkcji to `def implicitly[T](implicit arg : T) = arg`. Parametr typu `T` pozwala nam wykorzystać ją niezależnie od tego, jakiego typu encji szukamy. Więcej o parametrach typów powiem w podrozdziale 6.2. Wywołanie `implicitly` na typie `List[holder.Foo]` zwróci listę zdefiniowaną w obiekcie towarzyszącym `Foo`.

Mechanizm ten służy do implementacji *cech typów*, nazywanych też *klasami typów*. Cechy typów to abstrakcyjne interfejsy wykorzystujące parametry typu, które można implementować przy użyciu dowolnych typów. Przykładowo możemy zdefiniować cechę `BinaryFormat[T]`. Następnie można zaimplementować ją dla danego typu, definiując w ten sposób jego serializację do postaci binarnej. Oto przykład takiego interfejsu:

```
trait BinaryFormat[T] {
  def asBinary(entity: T) : Array[Byte]
}
```

Cecha `BinaryFormat` definiuje jedną metodę, `asBinary`. Pobiera ona instancję typu zgodnego z parametrem typu i zwraca tablicę bajtów reprezentującą przekazany parametr. Kod, który ma za zadanie przeprowadzić serializację i zapisać obiekt na dysku, może odszukać cechę typu `BinaryFormat` za pośrednictwem mechanizmu domniemań. Możemy dodać implementację dla naszego typu `Foo`, stosując słowo `implicit` w obiekcie towarzyszącym `Foo`:

```
trait Foo {}
object Foo {
  implicit lazy val binaryFormat = new BinaryFormat[Foo] {
    def asBinary(entity: Foo) = "zserializowaneFoo".getBytes
  }
}
```

Cecha `Foo` jest pusta. Jej obiekt towarzyszący ma składową `implicit val` przechowującą implementację `BinaryFormat`. Teraz gdy kod wymagający `BinaryFormat` widzi typ `Foo`, może w sposób domniemany odszukać `BinaryFormat`. Szczegóły tego mechanizmu i tej techniki projektowania zostaną dokładniej omówione w podrozdziale 7.2.

Domniemane wyszukiwanie na bazie parametrów typu i cech pozwala na uzyskanie eleganckiego kodu. Inny sposób dostarczania i odnajdywania argumentów domniemanych jest oparty na typach zagnieżdżonych.

ZAKRES DOMNIEMANY POPRZEZ TYPY ZAGNIEŻDŻONE

Zakres domniemany obejmuje także obiekty towarzyszące z zakresów zewnętrznych, jeśli typ został zdefiniowany w zakresie zagnieżdżonym. Pozwala nam to na stworzenie zestawu podręcznych zmiennych domniemanych dla typu w zakresie zewnętrznym. Oto przykład:

```
scala> object Foo {
     | trait Bar
     | implicit def newBar = new Bar {
     |   override def toString = "Implicit Bar"
     | }
     | }
defined module Foo

scala> implicitly[Foo.Bar]
res0: Foo.Bar = Implicit Bar
```

Obiekt `Foo` to typ zewnętrzny. Wewnątrz niego zdefiniowana została cecha `Bar`. Obiekt `Foo` dodatkowo zawiera opisaną jako `implicit` metodę tworzącą instancję cechy `Bar`. Po wywołaniu `implicitly[Foo.Bar]` wartość domniemana zostanie odnaleziona w zewnętrznej klasie `Foo`. Ta technika jest bardzo podobna do umieszczania składowych domniemanych bezpośrednio w obiekcie towarzyszącym. Definiowanie domniemanych składowych dla typów zagnieżdżonych przydaje się, gdy zakres zewnętrzny ma kilka podtypów. Technikę tę możemy stosować, jeśli nie jest możliwe utworzenie zmiennej domniemanej w obiekcie towarzyszącym.

Obiekty towarzyszące w Scali nie mogą być oznaczane jako `implicit`. Domniemane encje związane z typem obiektu, jeśli mają być dostępne w zakresie domniemanym, muszą być dostarczone w zakresie zewnętrznym. Oto przykład:

```
scala> object Foo {
     |   object Bar { override def toString = "Bar" }
     |   implicit def b : Bar.type = Bar
     | }
defined module Foo

scala> implicitly[Foo.Bar.type]
res1: Foo.Bar.type = Bar
```

Obiekt `Bar` jest zagnieżdżony wewnątrz obiektu `Foo`. Obiekt `Foo` definiuje domniemaną składową zwracającą `Bar.type`. Dzięki takiej definicji wywołanie `implicitly[Foo.`
`↪Bar.type]` zwróci obiekt `Bar`. W ten sposób jest możliwe definiowanie domniemanych obiektów.

Kolejny przypadek zagnieżdżania, który może zdziwić osoby nieprzyzwyczajone do niego, to obiekty pakietowe. Począwszy od wersji 2.8, obiekty mogą być definiowane jako obiekty pakietowe. Obiekt pakietowy to obiekt zdefiniowany z użyciem słowa kluczowego `package`. Konwencja w Scali nakazuje umieszczać wszystkie obiekty pakietowe w pliku o nazwie `package.scala` w katalogu odpowiadającym nazwie pakietu.

Każda klasa zdefiniowana wewnątrz pakietu jest w nim zagnieżdżona. Wszelkie encje domniemane zdefiniowane w obiekcie pakietowym będą dostępne w zakresie domniemanym wszystkich typów zdefiniowanych wewnątrz pakietu. Dzięki temu można składować wartości domniemane w wygodnej lokalizacji, bez potrzeby tworzenia obiektów towarzyszących dla każdego typu w pakiecie. Pokazuje to następujący przykład:

```
package object foo {
  implicit def foo = new Foo
}

package foo {
  class Foo {
    override def toString = "FOO!"
  }
}
```

Obiekt pakietowy `foo` zawiera jedno pole `implicit`, zwracające nową instancję klasy `Foo`. Następnie definiujemy klasę `Foo` wewnątrz pakietu `foo`. W Scali pakiety mogą być definiowane w wielu plikach, które w końcu zostaną zagregowane i utworzą jeden kompletny pakiet. Jeden pakiet może mieć tylko jeden obiekt pakietowy, niezależnie od tego, na ile plików został podzielony pakiet. Klasa `Foo` przesłania metodę `toString` — jej implementacja wypisuje łańcuch `"FOO!"`. Skompilujmy pakiet `foo` i przetestujmy go w REPL:

```
scala> implicitly[foo.Foo]
res0: foo.Foo = FOO!
```

Nie musieliśmy importować obiektu pakietowego ani jego składowych. Kompilator sam odnalazł domniemaną wartość obiektu `foo.Foo`. W Scali często można natknąć się na zestaw definicji domniemanych encji wewnątrz obiektu pakietowego danej biblioteki. Z reguły obiekt pakietowy zawiera także domniemane widoki, służące do konwersji typów.

5.2. Wzmacnianie klas za pomocą domniemanych widoków

Domniemany widok to automatyczna konwersja z jednego typu na drugi w celu spełnienia warunków stawianych przez wyrażenie. Definicja domniemanego widoku ma następującą ogólną postać:

```
implicit def <nazwaKonwersji>(<nazwaArgumentu> : TypOryginalny) : TypWidoku
```

Powyższa konwersja w sposób domniemany przekształca wartość typu `TypOryginalny` na wartość typu `TypWidoku`, jeśli jest dostępna w zakresie domniemanym.

Przeanalizujmy prosty przykład, w którym podejmiemy próbę konwersji zmiennej typu całkowitego na łańcuch znaków:

```
scala> def foo(msg : String) = println(msg)
foo: (msg: String)Unit

scala> foo(5)
```

```
<console>:7: error: type mismatch;
  found : Int(5)
  required: String
       foo(5)
```

Metoda `foo` pobiera wartość `String` i wypisuje ją w konsoli. Wywołanie `foo` z użyciem wartości `5` kończy się błędem, ponieważ typy nie są zgodne. Domniemany widok jest w stanie umożliwić to wywołanie:

```
scala> implicit def intToString(x : Int) = x.toString
intToString: (x: Int)java.lang.String

scala> foo(5)
5
```

Metoda `intToString` została zdefiniowana jako `implicit`. Pobiera ona wartość typu `Int` i zwraca `String`. Metoda ta jest domniemanym widokiem, często opisywanym jako `Int => String`. Teraz gdy wywołamy metodę `foo` z wartością `5`, wypisze ona łańcuch `"5"`. Kompilator wykryje, że typy nie są zgodne, a także że istnieje widok, który może rozwiązać problem.

Domniemane widoki wykorzystuje się w dwóch sytuacjach:

- Wyrażenie nie pasuje do typu oczekiwanego przez kompilator. Wtedy kompilator poszuka domniemanego widoku, który pozwoli przekształcić wartość do oczekiwanej postaci. Przykład to przekazanie zmiennej typu `Int` do funkcji oczekującej wartości `String`. Wówczas jest wymagane, by w zakresie istniał domniemany widok `String => Int`.
- Użyta została selekcja `e.t`, przy czym typ `e` nie ma składowej `t`. Kompilator wyszuka domniemany widok, który zastosuje do `e` i którego typ zwracany zawiera składową `t`. Dla przykładu, jeśli spróbujemy wywołać metodę `foo` na wartości `String`, kompilator wyszuka domniemany widok, który umożliwi kompilację wyrażenia. Wyrażenie `"foo".foo()` wymagałoby domniemanego widoku wyglądającego mniej więcej tak:

  ```
  implicit def stringToFoo(x : String) = new { def foo() : Unit = println("foo") }
  ```

Domniemane widoki wykorzystują ten sam zakres domniemany co domniemane parametry. Jednak gdy kompilator sprawdza możliwości przekształcenia typu, wyszukuje według typu źródłowego, a nie docelowego. Przykład:

```
scala> object test {
     | trait Foo
     | trait Bar
     | object Foo {
     |   implicit def fooToBar(foo : Foo) = new Bar {}
     | }
     | }
defined module test

scala> import test._
import test._
```

Obiekt `test` jest kontenerem, który pozwala nam na utworzenie obiektu towarzyszącego w ramach sesji REPL. Zawiera on cechy `Foo` oraz `Bar`, a także obiekt towarzyszący `Foo`. Obiekt towarzyszący `Foo` obejmuje domniemany widok przekształcający `Foo` na `Bar`. Pamiętaj, że gdy kompilator szuka domniemanych widoków, to typ źródłowy definiuje domniemany zakres. Oznacza to, że domniemane widoki zdefiniowane w obiekcie towarzyszącym `Foo` zostaną przeszukane tylko podczas próby konwersji z `Foo` na inny typ. Na potrzeby testów zdefiniujmy metodę oczekującą typu `Bar`:

```
scala> def bar(x : Bar) = println("bar")
bar: (x: test.Bar)Unit
```

Metoda `bar` pobiera obiekt `Bar` i wypisuje łańcuch znaków `"bar"`. Spróbujmy wywołać ją z argumentem typu `Foo` i zobaczmy, co się stanie:

```
scala> val x = new Foo {}
x: java.lang.Object with test.Foo = $anon$1@15e565bd

scala> bar(x)
bar
```

Wartość `x` jest typu `Foo`. Wyrażenie `bar(x)` zmusza kompilator do wyszukania domniemanego widoku. Ponieważ typ zmiennej `x` to `Foo`, kompilator szuka wśród typów powiązanych z `Foo`. Wreszcie znajduje widok `fooToBar` i dodaje odpowiednią transformację, dzięki czemu kompilacja kończy się sukcesem.

Mechanizm domniemanych konwersji pozwala nam na dopasowywanie do siebie różnych bibliotek, a także na dodawanie do istniejących typów naszych własnych metod pomocniczych. Adaptacja bibliotek Javy do postaci, w której dobrze współpracują z biblioteką standardową Scali, to dosyć częsta praktyka. Dla przykładu biblioteka standardowa definiuje moduł `scala.collection.JavaConversions`, usprawniający współpracę bibliotek do obsługi kolekcji w obu językach. Moduł ten jest zestawem domniemanych widoków, które można zaimportować do zakresu bieżącego w celu umożliwienia domniemanych konwersji pomiędzy kolekcjami w Javie i w Scali, przez co możliwe staje się także „dodawanie" metod do kolekcji Javy. Adaptacja bibliotek Javy czy wszelkich innych zewnętrznych bibliotek za pomocą domniemanych widoków to popularna praktyka w Scali. Przeanalizujmy odpowiedni przykład.

Będziemy chcieli opakować pakiet `java.security` tak, by korzystanie z niego w Scali było wygodniejsze. Chodzi nam zwłaszcza o uproszczenie zadania uruchamiania uprzywilejowanego kodu za pomocą `java.security.AccessController`. Klasa `AccessController` (kontroler dostępu) zawiera statyczną metodę `doPrivileged` (wykonaj uprzywilejowane), która pozwala na uruchamianie kodu w uprzywilejowanym stanie uprawnień. Metoda `doPrivileged` ma dwa warianty. Pierwszy przyznaje kodowi uprawnienia z bieżącego kontekstu, drugi pobiera obiekt `AccessControlContext` (kontekst kontroli dostępu), w którym są zdefiniowane uprawnienia do przyznania. Metoda `doPrivileged` pobiera argument typu `PrivilegedExceptionAction` (uprzywilejowana akcja), który jest cechą definiującą jedną metodę: `run` (uruchom). Cecha ta przypomina cechę Scali `Function0`, a my chcielibyśmy móc użyć funkcji anonimowej podczas wywoływania metody `doPrivileged`.

Stwórzmy domniemany widok przekształcający typ `Function0` do postaci metody `doPrivileged`:

```
object ScalaSecurityImplicits {
  implicit def functionToPrivilegedAction[A](func : Function0[A]) =
    new PrivilegedAction[A] {
      override def run() = func()
    }
}
```

Zdefiniowaliśmy obiekt `ScalaSecurityImplicits` zawierający widok domniemany. Widok `functionToPrivilegedAction` pobiera `Function0` i zwraca nowy obiekt `Privileged ↪Action`, którego metoda `run` wywołuje funkcję. Skorzystajmy z tego widoku:

```
scala> import ScalaSecurityImplicits._
import ScalaSecurityImplicits._

scala> AccessController.doPrivileged( () =>
     | println("Ma przywileje"))
Ma przywileje
```

Pierwsza instrukcja importuje domniemany widok do zakresu. Następnie wywołujemy metodę `doPrivileged`, przekazując jej anonimową funkcję `() => println("Ma przywileje")`. Po raz kolejny kompilator wykrywa, że funkcja anonimowa nie pasuje do oczekiwanego typu. Rozpoczyna wówczas wyszukiwanie i odnajduje domniemany widok zdefiniowany w `ScalaSecurityImplicits`. Tę samą technikę można wykorzystać do opakowywania obiektów Javy w obiekty Scali.

Często pisze się klasy opakowujące istniejące biblioteki Javy tak, by korzystały z bardziej zaawansowanych konstrukcji Scali. Domniemane konwersje Scali można zastosować w celu przekształcania typu oryginalnego w opakowany i z powrotem. Dla przykładu dodajmy kilka wygodnych metod do klasy `java.io.File`.

Zaczniemy od wprowadzenia specjalnej notacji — operator / będzie tworzył nowe pliki w danym katalogu. Stwórzmy klasę opakowującą, która wprowadzi ten operator:

```
class FileWrapper(val file: java.io.File) {
  def /(next : String) = new FileWrapper(new java.io.File(file, next))
  override def toString = file.getCanonicalPath
}
```

Klasa `FileWrapper` w konstruktorze pobiera obiekt `java.io.File`. Definiuje ona nową metodę /, która pobiera `String` i zwraca nowy obiekt `FileWrapper`. Nowy obiekt jest powiązany z plikiem o nazwie przekazanej metodzie /, wewnątrz katalogu związanego z oryginalnym plikiem. Na przykład jeśli oryginalny `FileWrapper` o nazwie `file` był związany z katalogiem `/tmp`, to wyrażenie `file / "mylog.txt"` zwróci nowy obiekt `FileWrapper` powiązany z plikiem `/tmp/mylog.txt`. Chcemy skorzystać z domniemanych widoków do automatycznej konwersji pomiędzy `java.io.File` i `FileWrapper`. Zacznijmy od dodania domniemanego widoku do obiektu towarzyszącego `FileWrapper`:

```
object FileWrapper {
  implicit def wrap(file : java.io.File) = new FileWrapper(file)
}
```

Obiekt towarzyszący `FileWrapper` definiuje jedną metodę, `wrap`, pobierającą `java.io.File` i zwracającą `FileWrapper`. Przetestujmy go teraz w sesji REPL:

```
scala> import FileWrapper.wrap
import FileWrapper.wrap
```

```
scala> val cur = new java.io.File(".")
cur: java.io.File = .

scala> cur / "temp.txt"
res0: FileWrapper = .../temp.txt
```

Pierwsza linia to import domniemanego widoku do naszego zakresu. Druga linia tworzy nowy obiekt `java.io.File`, przekazując konstruktorowi parametr `"."`. Ostatnia linia to wywołanie metody / na zmiennej `cur` typu `java.io.File`. Kompilator nie znajdzie tej metody w `java.io.File`, spróbuje więc wyszukać odpowiedni widok domniemany, umożliwiający kompilację. Po znalezieniu metody `wrap` kompilator opakuje `java.io.File` w `FileWrapper` i wywoła metodę /. W wyniku tego zostanie zwrócony obiekt `FileWrapper`.

Przedstawiony tu mechanizm stanowi doskonały sposób dodawania metod do istniejących klas Javy czy też do klas z różnych zewnętrznych bibliotek. Tworzenie obiektu opakowującego może wpłynąć na wydajność, jednak optymalizator HotSpot ma szansę na zminimalizowanie tego problemu. Piszę „ma szansę", ponieważ nie mamy gwarancji, że usunie on alokację obiektu opakowującego, jednak kilka niewielkich testów potwierdziło, że to robi. Jak zwykle lepiej jest przeprowadzić profilowanie aplikacji w celu wykrycia problematycznych fragmentów, niż zakładać, że optymalizator zrobi wszystko za nas.

Z metodą / wiąże się pewien problem. Zwraca ona nowy obiekt `FileWrapper`. Oznacza to, że nie możemy przekazać jej wyniku bezpośrednio do metody oczekującej zwykłego obiektu `java.io.File`. Moglibyśmy zmienić ją, by zwracała `java.io.File`, ale Scala oferuje jeszcze inne rozwiązanie. Gdy przekażemy `FileWrapper` do metody oczekującej `java.io.File`, kompilator rozpocznie wyszukiwanie odpowiedniego widoku. Jak już wspomniałem, przeszukany zostanie także obiekt towarzyszący typowi `FileWrapper`. Dodajmy do niego domniemany widok `unwrap` (rozpakuj) i zobaczmy, czy zadziała:

```
object FileWrapper {
  implicit def wrap(file : java.io.File) = new FileWrapper(file)
  implicit def unwrap(wrapper : FileWrapper) = wrapper.file
}
```

Obiekt towarzyszący `FileWrapper` ma teraz dwie metody: `wrap` i `unwrap`. Metoda `unwrap` pobiera instancję `FileWrapper` i zwraca odpakowany typ `java.io.File`. Przetestujmy ją teraz w REPL.

```
scala> import test.FileWrapper.wrap
import test.FileWrapper.wrap

scala> val cur = new java.io.File(".")
cur: java.io.File = .

scala> def useFile(file : java.io.File) = println(file.getCanonicalPath)
useFile: (file: java.io.File)Unit

scala> useFile(cur / "temp.txt")
/home/jsuereth/projects/book/scala-in-depth/chapter5/wrappers/temp.txt
```

Pierwsza linia importuje widok domniemany `wrap`. Następna konstruuje obiekt `java.io.File` wskazujący na bieżący katalog. Trzecia linia definiuje metodę `useFile`. Metoda ta oczekuje wejścia typu `java.io.File`, którego ścieżkę wypisze w konsoli.

Ostatnia linia to wywołanie metody `useFile` z argumentem w postaci wyrażenia `cur / "temp.txt"`. Kompilator, jak zwykle, na widok metody `/` rozpocznie wyszukiwanie odpowiedniego widoku domniemanego do przeprowadzenia konwersji. Wynikiem konwersji będzie `FileWrapper`, ale metoda `useFile` oczekuje typu `java.io.File`. Kompilator przeprowadzi kolejne wyszukiwanie za pomocą typu `Function1[java.io.File, FileWrapper]`. W ten sposób znajdzie widok domniemany `unwrap` w obiekcie towarzyszącym `FileWrapper`. Wszystkie typy się zgadzają, zatem kompilacja kończy się sukcesem. W czasie wykonania pojawi się oczekiwana wartość typu `String`.

Zwróć uwagę na to, że do wywołania widoku `unwrap` nie jest nam potrzebna instrukcja `import`, wymagana w przypadku metody `wrap`. Jest tak dlatego, że obiekt `wrap` był potrzebny w sytuacji, gdy kompilator nie znał typu wymaganego do spełnienia wyrażenia `cur / "temp.txt"`, dlatego sprawdzał tylko lokalne widoki, jako że `java.io.File` nie ma obiektu towarzyszącego. Opisany tu mechanizm pozwala na stworzenie obiektu opakowującego z dodatkowymi metodami, który jest w stanie niemalże niezauważalnie przekształcać obiekty z i do postaci opakowanej.

Zachowaj ostrożność podczas dodawania funkcjonalności do istniejących klas za pomocą domniemanych widoków. Ten mechanizm sprawia, że trudno jest zauważyć konflikt nazw pomiędzy różnymi domniemanymi widokami typu. Na dodatek pociąga on za sobą pewne nakłady wydajnościowe, z którymi niekoniecznie poradzi sobie optymalizator HotSpot. Wreszcie programistom niekorzystającym z nowoczesnego środowiska programistycznego nie jest łatwo ocenić, które domniemane widoki są wykorzystywane w danym bloku kodu.

Reguła 13

Unikaj domniemanych widoków

Domniemane widoki to najbardziej nadużywana funkcjonalność Scali. Mimo że w wielu sytuacjach ich wprowadzenie może się wydawać dobrym pomysłem, w większości z nich Scala oferuje lepsze alternatywy. Zbyt duża liczba takich widoków z pewnością utrudni nowemu programiście wdrożenie się w kod. Widoki są przydatne, jednak ich stosowanie należy ograniczyć do przypadków, w których rzeczywiście są najlepszym rozwiązaniem.

Domniemane widoki w Scali pozwalają użytkownikowi dopasować istniejące API do swoich potrzeb. W połączeniu z obiektami opakowującymi i obiektami towarzyszącymi widoki są w stanie radykalnie zmniejszyć nakład pracy niezbędny do zintegrowania bibliotek z podobnymi, ale nie takimi samymi interfejsami, a także pozwalają na dodawanie nowych funkcjonalności do istniejących bibliotek. Domniemane widoki są kluczem do pisania ekspresywnego kodu, jednak należy się z nimi obchodzić ostrożnie.

Kolejnym elementem powiązanym z pojęciem domniemań są parametry domyślne.

5.3. Parametry domniemane i domyślne

Argumenty domniemane to mechanizm pozwalający na uniknięcie redundantnego specyfikowania parametrów. Argumenty domniemane świetnie uzupełniają się z parametrami domyślnymi. Jeśli nie podano parametru i nie odnaleziono dla niego wartości domniemanej, zostanie wykorzystana wartość domyślna. W ten sposób możemy tworzyć parametry domyślne, które użytkownik może pominąć, ale zawsze ma możliwość ich określenia.

Jako przykład zaimplementujmy zestaw metod wykonujących obliczenia na macierzach. Będą one korzystały z wątków w celu zrównoleglenia obliczeń. Jako projektant biblioteki nie wiesz jednak, gdzie metody te będą wywoływane. Mogą zostać uruchomione w kontekście, w którym nie wolno korzystać z wielowątkowości, a może mają już swoją własną kolejkę zadań. Chcemy umożliwić użytkownikowi określenie sposobu korzystania z wątków, ale chcemy także zapewnić tryb domyślny.

Zacznijmy od definicji klasy `Matrix` (macierz) na listingu 5.7.

Listing 5.7. Prosta klasa Matrix

```
class Matrix(private val repr : Array[Array[Double]]) {
  def row(idx : Int) : Seq[Double] = {
    repr(idx)
  }
  def col(idx : Int) : Seq[Double] = {
    repr.foldLeft(ArrayBuffer[Double]()) {
      (buffer, currentRow) =>
        buffer.append(currentRow(idx))
        buffer
    } toArray
  }
  lazy val rowRank = repr.size
  lazy val colRank = if(rowRank > 0) repr(0).size else 0
  override def toString = "Macierz" + repr.foldLeft("") {
    (msg, row) => msg + row.mkString("\n|", " | ", "|")
  }
}
```

Klasa `Matrix` pobiera tablicę wartości typu `Double` i zapewnia dwie podobne metody: `row` i `col`. Pobierają one wartość indeksu i zwracają tablicę wartości z danego wiersza (`row`) lub kolumny (`col`). Klasa `Matrix` zawiera także wartości `rowRank` i `colRank`, zwracające odpowiednio liczbę wierszy i kolumn. Wreszcie metoda `toString` wyświetla przyjazną reprezentację danych w macierzy.

Klasa `Matrix` jest gotowa do zrównoleglenia. Zacznijmy od definicji przeznaczonego do tego interfejsu:

```
trait ThreadStrategy {
  def execute[A](func : Function0[A]) : Function0[A]
}
```

Interfejs `ThreadStrategy` (strategia wątków) definiuje jedną metodę, `execute` (wykonaj). Pobiera ona funkcję, która zwraca wartość typu `A`. Zwraca wartość tego samego typu: funkcję zwracającą wartość `A`. Zwrócona funkcja powinna zwrócić tę samą wartość co funkcja przekazana, ale może ona zablokować aktualny wątek do momentu, w którym jej wartość zostanie wyznaczona w osobnym wątku. Zaimplementujmy naszą usługę obliczeń na macierzach, korzystając z interfejsu `ThreadStrategy`:

```
object MatrixUtils {
  def multiply(a: Matrix,
               b: Matrix)(
               implicit threading: ThreadStrategy): MatrixN = {
    ...
  }
}
```

Obiekt `MatrixUtils` zawiera metodę `multiply` (mnóż). Pobiera ona dwie macierze, zakładając, że mają one odpowiednie wymiary, i zwraca nową macierz, będącą wynikiem mnożenia dwóch macierzy przekazanych jako parametry. Mnożenie macierzy polega na mnożeniu wartości z wierszy pierwszej macierzy przez wartości z kolumn drugiej i dodawaniu iloczynów. Takie mnożenie i sumowanie musi zostać osobno przeprowadzone dla każdego elementu wynikowej macierzy. Prostym sposobem zrównoleglenia obliczeń jest wyliczenie każdej wartości w osobnym wątku. Algorytm metody `Matrix ↪Utils.multiply` jest prosty:

- utwórz bufor do przechowywania wyników,
- stwórz domknięcie, które wyznaczy pojedynczą wartość dla pary wiersz – kolumna i umieść ją w buforze,
- wyślij stworzone w ten sposób domknięcia do `ThreadStrategy`,
- wywołaj funkcje zwrócone przez `ThreadStrategy`, by upewnić się, że ich wykonanie dobiegło końca,
- opakuj bufor w klasę `Matrix` i go zwróć.

Zacznijmy od utworzenia bufora:

```
def multiply(a: Matrix,
             b: Matrix)(
             implicit threading : ThreadStrategy): Matrix = {
  assert(a.colRank == b.rowRank)
  val buffer = new Array[Array[Double]](a.rowRank)
  for ( i <- 0 until a.rowRank ) {
    buffer(i) = new Array[Double](b.colRank)
  }
  ...
}
```

Początkowa instrukcja `assert` została dodana w celu upewnienia się, że wymiary macierzy pozwalają na ich mnożenie. Żeby operacja ta była możliwa, liczba kolumn w pierwszej macierzy musi być równa liczbie wierszy w drugiej. Następnie tworzymy tablicę tablic, którą wykorzystamy jako bufor. Wynikowa macierz będzie miała tę samą liczbę wierszy co macierz `a` i tę samą liczbę kolumn co macierz `b`. Gdy bufor jest już gotowy, możemy przystąpić do utworzenia zbioru domknięć, które wyliczą poszczególne wartości i umieszczą je w buforze (listing 5.8).

Listing 5.8. Mnożenie macierzy

```
def multiply(a: Matrix,
             b: Matrix)(
             implicit threading : ThreadStrategy): Matrix = {
  ...
  def computeValue(row : Int, col : Int) : Unit = {
    val pairwiseElements =
      a.row(row).zip(b.col(col))
    val products =
      for((x,y) <- pairwiseElements)
         yield x*y
    val result = products.sum
    buffer(row)(col) = result
  }
...
```

Metoda pomocnicza `computeValue` (wylicz wartość) pobiera numer wiersza i kolumny, a następnie wylicza wartość odpowiadającą tym elementom. Pierwszy krok to dopasowanie parami kolejnych elementów z wiersza `a` i kolumny `b`. Scala oferuje tu funkcję `zip`, która pobiera dwie kolekcje i dopasowuje do siebie ich elementy. Następnie sparowane elementy są mnożone, w wyniku czego powstaje lista ich iloczynów. Wreszcie lista ta jest sumowana. Wynik obliczeń jest wstawiany w miejsce w buforze odpowiadające danemu wierszowi i danej kolumnie. Kolejną rzeczą, jaką musimy zrobić, jest skonstruowanie na podstawie tej metody funkcji, która będzie wyznaczała wartość dla pary wiersz – kolumna i przekazanie tej funkcji do odpowiedniej strategii:

```
val computations = for {
  i <- 0 until a.rowRank
  j <- 0 until b.colRank
} yield threading.execute { () => computeValue(i,j) }
```

Pętla `for` przechodzi przez każdy wiersz i każdą kolumnę w macierzy wynikowej i przekazuje funkcję do metody `execute` w `ThreadStrategy`. Składnia `() =>` jest stosowana podczas tworzenia obiektów funkcji anonimowych, które nie pobierają argumentów, wymaganych przez typ `Function0`. Po przekazaniu pracy wątkom, a przed zwróceniem wyników, metoda `multiply` musi „upewnić się", że praca została wykonana. Robi to, wywołując każdą metodę zwróconą przez `ThreadStrategy`:

```
def multiply(a: Matrix,
             b: Matrix)(
             implicit threading : ThreadStrategy) : Matrix = {
  ...
  computations.foreach(_())
  new Matrix(buffer)
}
```

Ostatnia część metody sprawdza, czy wszystkie obliczenia rzeczywiście zostały wykonane, i zwraca obiekt `Matrix` zbudowany na podstawie bufora. Przetestujemy kod w sesji REPL, ale najpierw musimy zaimplementować interfejs `ThreadStrategy`. Stwórzmy prostą wersję, która wykonuje całość pracy w jednym wątku:

```
object SameThreadStrategy extends ThreadStrategy {
  def execute[A](func : Function0[A]) = func
}
```

Strategia `SameThreadStrategy` sprowadza się do wykonania wszystkich obliczeń w jednym wątku — zwracana jest dokładnie ta sama funkcja, która została przekazana metodzie `execute`. Przetestujmy metodę `multiply` w sesji REPL:

```
scala> implicit val ts = sameThreadStrategy
ts: ThreadStrategy.sameThreadStrategy.type = ...

scala> val x = new Matrix(Array(Array(1,2,3), Array(4,5,6)))
x: library.Matrix =
Macierz
|1.0 | 2.0 | 3.0|
|4.0 | 5.0 | 6.0|

scala> val y = new Matrix(Array(Array(1), Array(1), Array(1)))
y: library.Matrix =
```

```
Macierz
|1.0|
|1.0|
|1.0|

scala> MatrixService.multiply(x,y)
res0: library.Matrix =
Macierz
|6.0|
|15.0|
```

W pierwszej linii tworzymy domniemaną strategię `ThreadStrategy`, której będziemy używać we wszystkich pozostałych przykładach. Następnie konstruujemy dwie macierze i mnożymy je przez siebie. Macierz o wymiarach 2×3 przemnożona przez macierz o wymiarach 3×1 da wynik o wymiarach 2×1, zgodnie z oczekiwaniami. Wygląda na to, że w ramach jednego wątku wszystko działa jak trzeba, zatem przejdźmy do wersji wielowątkowej (listing 5.9).

Listing 5.9. Strategia współbieżna

```
import java.util.concurrent.{Callable, Executors}

  object ThreadPoolStrategy extends ThreadStrategy {
    val pool = Executors.newFixedThreadPool(
                java.lang.Runtime.getRuntime.availableProcessors)
  def execute[A](func : Function0[A] ) = {
    val future = pool.submit(new Callable[A] {
      def call() : A = {
        Console.println("Wykonanie funkcji w wątku: " +
                        Thread.currentThread.getName)
        func()
      }
    })
    () => future.get()
  }
}
```

Tym razem implementacja `ThreadPoolStrategy` tworzy pulę wątków, korzystając z biblioteki `java.util.concurrent.Executors`. Liczba wątków w puli odpowiada liczbie dostępnych procesorów. Metoda `execute` pobiera przekazaną jej funkcję i tworzy anonimową instancję `Callable`. Interfejs `Callable` służy właśnie do przekazywania zadań do wykonania przez wątki z puli. Zwracany jest obiekt typu `Future`, dzięki któremu jest możliwe określenie, czy praca została już wykonana. Ostatnia linia `execute` zwraca anonimowe domknięcie, które wywoła metodę `get` na obiekcie `future`. To wywołanie zablokuje program do momentu, aż oryginalna funkcja zakończy działanie i zwróci wynik. Za każdym razem, gdy wewnątrz `Callable` jest wykonywana funkcja, wypisana zostanie informacja o tym, który wątek za nią odpowiada. Wypróbujmy nasz kod w sesji REPL:

```
scala> implicit val ts = ThreadPoolStrategy
ts: ThreadStrategy.ThreadPoolStrategy.type = ...

scala> val x = new Matrix(Array(Array(1,2,3), Array(4,5,6)))
x: library.Matrix =
```

```
Macierz
|1.0 | 2.0 | 3.0|
|4.0 | 5.0 | 6.0|

scala> val y = new Matrix(Array(Array(1), Array(1), Array(1)))
y: library.Matrix =
Macierz
|1.0|
|1.0|
|1.0|

scala> MatrixUtils.multiply(x,y)
Wykonanie funkcji w wątku: pool-2-thread-1
Wykonanie funkcji w wątku: pool-2-thread-2
res0: library.Matrix =
Macierz
|6.0|
|15.0|
```

W pierwszej linii tworzymy strategię `ThreadPoolStrategy` i oznaczamy ją jako `implicit`. Zmienne `x` i `y` to macierze o wymiarach 2×3 i 3×1. Metoda `MatrixService.multiply` wypisuje teraz dwie linie, co oznacza, że obliczenia są wykonywane w różnych wątkach. Wynikowa macierz zawiera poprawne wyniki, tak samo jak wcześniej.

A co by się stało, gdybyśmy chcieli zapewnić domyślną strategię wątków dla użytkowników biblioteki, którą mogliby jednak nadpisać wedle potrzeb? Możemy skorzystać z mechanizmu parametrów domyślnych. Parametr domyślny zostanie użyty, gdy w zakresie domniemanym nie będzie dostępna odpowiednia wartość, zatem użytkownicy będą mogli nadpisać zakres domyślny, importując lub tworząc własną strategię `ThreadStrategy`. Użytkownicy mogą także przesłonić zachowanie pojedynczej metody, jawnie przekazując `ThreadStrategy`. Zmieńmy sygnaturę metody `MatrixService.multiply`:

```
def multiply(a: Matrix, b: Matrix)(
            implicit threading: ThreadStrategy = SameThreadStrategy
           ) : Matrix = {
  ...
}
```

Metoda `multiply` za domyślną strategię uznaje teraz `SameThreadStrategy`. Korzystając z biblioteki, nie musimy już określać własnej strategii:

```
scala> val x = new Matrix(Array(Array(1,2,3), Array(4,5,6)))
x: library.Matrix =
Macierz
|1.0 | 2.0 | 3.0|
|4.0 | 5.0 | 6.0|

scala> val y = new Matrix(Array(Array(1), Array(1), Array(1)))
y: library.Matrix =
Macierz
|1.0|
|1.0|
|1.0|

scala> MatrixService.multiply(x,y)
res0: library.Matrix =
```

```
Macierz
|6.0|
|15.0|
```

Inaczej niż w przypadku zwykłych parametrów domyślnych, domniemana lista parametrów z wartościami domyślnymi nie musi być oznaczona dodatkowymi nawiasami `()`. Elegancja parametrów domniemanych została połączona z użytecznością parametrów domyślnych. Nadal możemy normalnie korzystać z parametrów domniemanych:

```
scala> implicit val ts = ThreadPoolStrategy
ts: ThreadStrategy.ThreadPoolStrategy.type = ...

scala> MatrixUtils.multiply(x,y)
Wykonanie funkcji w wątku: pool-2-thread-1
Wykonanie funkcji w wątku: pool-2-thread-2
res1: library.Matrix =
Macierz
|6.0|
|15.0|
```

Pierwsza linia tworzy w sposób domniemany dostępną strategię wątków. Od tej chwili wywołanie `MatrixService.multiply` będzie stosowało strategię `ThreadPoolStrategy`. Użytkownicy usługi `MatrixService` mogą dzięki temu sami decydować, kiedy zrównoleglać wykonywane przez nią obliczenia. Mogą dostarczyć domniemany obiekt strategii w danym zakresie lub po prostu wywołać metodę z odpowiednim parametrem `ThreadStrategy`.

Technika tworzenia domniemanej wartości w zakresie obliczeniowym to przejaw wzorca Strategia użytego w odpowiednim miejscu i w odpowiedni sposób. Wzorzec ten ma zastosowanie wtedy, gdy fragment kodu musi wykonać pewną operację, lecz pewne elementy zachowania — „strategia wykonania" — mogą zostać zmienione. Przykładem takiego zachowania jest obiekt `ThreadPoolStrategy` przekazywany do metod biblioteki `MatrixUtils`. Ta sama strategia może zostać wykorzystana w wielu innych miejscach systemu. Jest to kolejny obok dziedziczenia (omówionego w podrozdziale 4.3) sposób składania komponentów odpowiedzialnych za zachowanie aplikacji.

Kolejny dobry przykład zastosowania parametrów domniemanych i domyślnych to odczyt linii z pliku. W ogólnym przypadku użytkowników nie interesuje, czy linia jest zakończona sekwencją `\r`, `\n`, czy `\r\n`. Biblioteka programistyczna powinna jednak obsługiwać wszystkie warianty. Można zaprojektować kod tak, by użytkownik mógł nieobowiązkowo podać znak końca linii, ale domyślną wartością byłoby „bez różnicy".

Parametry domniemane pozwalają uniknąć nadmiarowego, powtarzającego się kodu. Należy jednak pamiętać o zachowaniu ostrożności — temu zagadnieniu jest poświęcony następny podrozdział.

5.4. Ograniczanie zakresu encji domniemanych

Najważniejszym wymaganiem podczas programowania z użyciem domniemań jest głębokie rozumienie tego, co dzieje się w danym bloku kodu. Programista może ułatwić sobie zadanie, ograniczając liczbę miejsc, które musi sprawdzić w poszukiwaniu dostępnych encji domniemanych. Oto ich możliwe lokalizacje:

- obiekty towarzyszące wszelkich typów powiązanych, w tym obiektów pakietowych,

- obiekt `scala.Predef`,
- wszelkie elementy zaimportowane do bieżącego zakresu.

Jak już widzieliśmy w sekcji 1.1.3, Scala w poszukiwaniu zmiennych domniemanych sprawdzi obiekty towarzyszące wszelkich typów powiązanych. To zachowanie należy do rdzenia języka. Obiekty towarzyszące i pakietowe powinny być uznawane za część API klasy. Ucząc się nowej biblioteki, zawsze sprawdzaj obiekty towarzyszące i obiekty pakietowe pod kątem domniemanych konwersji, z których będziesz mógł korzystać.

Reguła 14

Ograniczaj zakres domniemań

Ponieważ konflikty encji domniemanych wymagają zaawansowanego przekazywania argumentów i konwersji, najbezpieczniej jest ich unikać. Z tego powodu najlepiej ograniczać liczbę encji domniemanych w zakresie i udostępniać je w sposób, który można ukryć lub nadpisać.

Na początku każdego skompilowanego pliku Scali pojawia się domyślna klauzula `import scala.Predef._`. Obiekt `Predef` zawiera wiele użytecznych przekształceń, w tym te pozwalające na dodawanie metod do typu `java.lang.String`, dzięki czemu wspiera on metody wymagane w specyfikacji języka. Zawiera także przekształcenia pomiędzy obiektami opakowującymi typy proste w Javie i odpowiadającymi im typami zunifikowanymi w Scali. Dla przykładu w `scala.Predef` istnieje domyślna konwersja `java.lang.Integer => scala.Int`. Podczas programowania w Scali warto znać przekształcenia dostępne w tym obiekcie.

Ostatnia możliwa lokalizacja encji domniemanych to instrukcje `import` w kodzie źródłowym. Zaimportowane encje domniemane dość trudno jest wytropić, trudno także je dokumentować. Ponieważ jest to jedyny przypadek domniemań wymagający instrukcji `import` w każdym pliku źródłowym, poświęcimy mu najwięcej uwagi.

5.4.1. Przygotowywanie encji domniemanych do zaimportowania

Podczas tworzenia nowego widoku lub parametru domniemanego, który w przyszłości ma być jawnie importowany, upewnij się, że są spełnione następujące warunki:

- Domniemany widok lub parametr nie jest w konflikcie z inną domniemaną encją.
- Nazwa domniemanego widoku lub parametru nie jest w konflikcie z niczym w obiekcie `scala.Predef`.
- Domniemany widok lub parametr jest odkrywalny, to znaczy użytkownik biblioteki lub modułu powinien być w stanie zlokalizować domniemaną encję i zrozumieć jej przeznaczenie.

Ponieważ Scala wyszukuje encje domniemane w dostępnym zakresie, konflikt pomiędzy dwiema domniemanymi definicjami może prowadzić do powstania problemów. Konflikty takie bywają trudne do wykrycia, ponieważ widoki i parametry mogą być definiowane w dowolnym zakresie i importowane. Obiekt `scala.Predef` w sposób domniemany importuje całą swoją zawartość do każdego pliku Scali, dlatego konflikty z jego składowymi szybko się uwidaczniają. Zobaczmy, co się stanie w przypadku konfliktu:

```
object Time {
  case class TimeRange(start : Long, end : Long)
  implicit def longWrapper(start : Long) = new {
```

```
    def to(end : Long) = TimeRange(start, end)
  }
}
```

Powyższy kod definiuje obiekt `Time` (czas) zawierający klasę `TimeRange` (przedział czasowy). Mamy też domniemaną konwersję na typie `Long`, dodającą do niego metodę `to`. Za pomocą tej metody możesz konstruować przedziały czasowe.

Domniemana konwersja `longWrapper` jest w konflikcie z encją `scala.Predef.long`
`↪Wrapper`, która między innymi oferuje widok domyślny także zawierający metodę `to`. Ta metoda `to` zwraca obiekt `Range` (przedział), który może zostać użyty w pętli `for`. Wyobraź sobie scenariusz, w którym ktoś stosuje naszą domniemaną konwersję w celu definiowania przedziałów czasowych, ale później chce odwołać się do oryginalnego widoku zdefiniowanego w `Predef`, ponieważ ma zamiar napisać pętlę `for`. Jednym z rozwiązań jest zaimportowanie widoku z `Predef` z wyższym priorytetem w węższym zakresie — tylko tam, gdzie jest potrzebny. Taki kod nie jest zbyt czytelny, co widać na listingu 5.10.

Listing 5.10. Priorytety i zakresy

```
object Test {
  println(1L to 10L)
  import Time._
  println(1L to 10L)
  def x() = {
    import scala.Predef.longWrapper
    println(1L to 10L)
    def y() = {
      import Time.longWrapper
      println(1L to 10L)
    }
    y()
  }
  x()
}
```

Obiekt `Test` natychmiast po swojej definicji wypisuje w konsoli wyrażenie `(1L to 10L)`. Następnie importujemy domniemane encje z `Time` i jeszcze raz wypisujemy wynik wyrażenia. Dalej, w zagnieżdżonym zakresie, importujemy `longWrapper` z `Predef` i ponownie wypisujemy wynik na wyjściu. Na koniec, jeszcze głębiej, importujemy `longWrapper` z `Time` i wypisujemy wynik. Oto co pojawi się na wyjściu:

```
scala> Test
NumericRange(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
TimeRange(1,10)
NumericRange(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
TimeRange(1,10)
res0: Test.type = Test$@2d34ab9b
```

Pierwsza wypisana linia typu `NumericRange` (zakres liczbowy) to wynik wyrażenia `(1L to 10L)` przed jakąkolwiek instrukcją `import`. Dalej mamy wynik `TimeRange`, zwrócony dzięki zaimportowaniu domniemanego widoku z `Time`. Dalej pojawia się `NumericRange`, związany z zakresem zagnieżdżonym w metodzie `x()`, a ostatni wynik `TimeRange` to wynik z najbardziej zagnieżdżonej metody `y()`. Gdyby obiekt `Test` zawierał więcej takiego kodu

i kod ten nie mieściłby się w jednym oknie, trudno byłoby przewidzieć, jaki będzie wynik wyrażenia `(1L to 10L)` w danym miejscu. Unikaj tego typu zawikłanych sytuacji. Najlepiej wystrzegać się konfliktów w definicji domniemanych widoków, jednak nie zawsze jest to proste. W trudnych sytuacjach można zdecydować, że jedna z konwersji będzie domniemana, natomiast inne będą wywoływane tradycyjnie.

Projektowanie odkrywalnych domniemanych encji zwiększa czytelność kodu, ponieważ nowemu programiście łatwiej jest zrozumieć, co dzieje się w danym fragmencie kodu i co powinno się w nim dziać. Znaczenie odkrywalnych encji rośnie podczas pracy w zespole. W społeczności Scali panuje ogólna zgoda na ograniczenie importowalnych encji domniemanych do jednego z dwóch miejsc:

- obiektów pakietowych,
- obiektów singletonowych z postfiksowymi (przyrostkowymi) widokami domniemanymi.

Obiekty pakietowe są doskonałym miejscem do składowania encji domniemanych, ponieważ i tak są one w zakresie domniemanym dla typów zdefiniowanych wewnątrz pakietu. Użytkownicy powinni szukać w obiekcie pakietowym encji domniemanych związanych z pakietem. Umieszczenie w obiekcie pakietowym domniemanych encji wymagających jawnego importu zwiększy ich szanse na to, że zostaną zauważone przez użytkownika. Podczas korzystania z obiektu pakietowego do przechowywania encji domniemanych zawsze dokumentuj, czy wymagają one jawnych importów.

Lepszym rozwiązaniem niż dokumentowanie jawnych importów domniemanych encji jest całkowita rezygnacja z instrukcji `import`.

5.4.2. Parametry i widoki domniemane bez podatku od importu

Parametry i widoki domniemane świetnie sobie radzą bez instrukcji `import`. Ich drugorzędne reguły wyszukiwania, sprawdzające obiekty towarzyszące typów powiązanych, pozwalają na definiowanie domniemanych konwersji i wartości, które nie wymagają używania instrukcji `import`. Przy odrobinie kreatywności jest możliwe stworzenie ekspresywnych bibliotek, które w pełni wykorzystują siłę domniemań bez potrzeby importowania. Przeanalizujemy to zadanie na przykładzie, za który posłuży nam biblioteka do reprezentacji liczb zespolonych.

Liczby zespolone to liczby, które składają się z części rzeczywistej i urojonej. Część urojona jest mnożona przez pierwiastek kwadratowy z –1, znany także jako *i* (lub *j* w dziedzinie elektrotechniki). W Scali łatwo zamodelować taką liczbę za pomocą tzw. klasy wzorcowej (`case class`) — prostej klasy będącej kontenerem na wartości.

```
package complexmath
case class ComplexNumber(real : Double, imaginary : Double)
```

Klasa `ComplexNumber` definiuje część rzeczywistą jako pole `real` typu `Double`. Część urojona to pole `imaginary`, także typu `Double`. Klasa reprezentuje liczby zespolone przy użyciu arytmetyki zmiennoprzecinkowej w poszczególnych częściach. Liczby zespolone można dodawać i mnożyć, stwórzmy więc przeznaczone do tego metody (listing 5.11).

Listing 5.11. Klasa ComplexNumber reprezentująca liczbę zespoloną

```
package complexmath

case class ComplexNumber(real : Double, imaginary : Double) {
  def *(other : ComplexNumber) =
    ComplexNumber( (real*other.real) - (imaginary * other.imaginary),
                   (real*other.imaginary) + (imaginary * other.real) )
  def +(other : ComplexNumber) =
    ComplexNumber( real + other.real, imaginary + other.imaginary )
}
```

Dodawanie (+) polega na dodaniu do siebie osobno części rzeczywistej i urojonej dwóch liczb. Mnożenie (*) jest nieco bardziej złożone. Definiuje się je w następujący sposób:

- Część rzeczywista iloczynu dwóch liczb zespolonych to iloczyn ich komponentów rzeczywistych pomniejszony o iloczyn ich części urojonych: `(real*other.real) - (imaginary * other.imaginary)`.
- Część urojona iloczynu dwóch liczb zespolonych to suma iloczynów części rzeczywistej jednej liczby z częścią urojoną drugiej: `(real*other.imaginary) + (imaginary * other.real)`.

Klasa `ComplexNumber` wspiera teraz dodawanie i mnożenie. Zobaczmy ją w akcji:

```
scala> ComplexNumber(1,0) * ComplexNumber(0,1)
res0: imath.ComplexNumber = ComplexNumber(0.0,1.0)

scala> ComplexNumber(1,0) + ComplexNumber(0,1)
res1: imath.ComplexNumber = ComplexNumber(1.0,1.0)
```

Pierwsza linia mnoży liczbę rzeczywistą z liczbą urojoną — wynikiem jest liczba urojona. Druga linia dodaje do siebie liczbę rzeczywistą i urojoną, tworząc liczbę zespoloną. Operatory + i * działają zgodnie z oczekiwaniami, jednak wywoływanie metody wytwórczej `ComplexNumber` jest trochę męczące. Można to uprościć, stosując nową notację dla liczb zespolonych.

W matematyce liczby zespolone najczęściej przedstawia się jako sumę części rzeczywistej i urojonej. Liczba `ComplexNumber(1.0,1.0)` zostałaby zapisana jako `1.0 + 1.0*i`, gdzie `i` to jednostka urojona, odpowiadająca pierwiastkowi kwadratowemu z –1. Taka notacja byłaby optymalną składnią dla biblioteki zajmującej się liczbami zespolonymi. Zdefiniujmy symbol i powiążmy go z pierwiastkiem kwadratowym z –1.

```
package object complexmath {
  val i = ComplexNumber(0.0,1.0)
}
```

Zdefiniowaliśmy w ten sposób wartość `val i` w obiekcie pakietowym `complexmath`. Nazwa `i` staje się dostępna w całym pakiecie, możliwe jest także jej bezpośrednie importowanie. Za jej pomocą można konstruować liczby zespolone z ich części rzeczywistej i urojonej. Ciągle jednak brakuje pewnego elementu, co pokazuje następująca sesja REPL:

```
scala> i * 1.0
<console>:9: error: type mismatch;
  found : Double(1.0)
  required: ComplexNumber
       i * 1.0
```

Próba pomnożenia naszej liczby urojonej przez wartość typu `Double` kończy się niepowodzeniem, ponieważ typ `ComplexNumber` definiuje mnożenie jedynie dla zmiennych typu `ComplexNumber`. W matematyce możliwe jest mnożenie liczb rzeczywistych przez zespolone, ponieważ na liczbę rzeczywistą można spojrzeć jak na liczbę zespoloną bez części urojonej. Tę właściwość liczb rzeczywistych można emulować w Scali za pomocą domniemanej konwersji z `Double` na `ComplexNumber`:

```
package object complexmath {
  implicit def realToComplex(r : Double) = new ComplexNumber(r, 0.0)
  val i = ComplexNumber(0.0, 1.0)
}
```

Obiekt pakietowy `complexmath` zawiera teraz także definicję wartości `i` oraz domniemanej konwersji z `Double` na `ComplexNumber` o nazwie `realToComplex`. Chcielibyśmy ograniczyć zastosowanie tej konwersji do przypadków, w których jest ona absolutnie konieczna. Spróbujmy zastosować pakiet `complexmath` bez jawnego importowania żadnych konwersji:

```
scala> import complexmath.i
import complexmath.i

scala> val x = i*5.0 + 1.0
x: complexmath.ComplexNumber = ComplexNumber(1.0,5.0)
```

Wartość `val x` została zadeklarowana za pomocą wyrażenia `i*5 + 1` i ma typ `ComplexNumber`. Część rzeczywista to `1.0`. a część urojona `5.0`. Zwróć uwagę, że tylko nazwa `i` została zaimportowana z `complexmath`. Pozostałe domniemane konwersje są wywoływane z obiektu `i`, gdy tylko kompilator napotyka wyrażenie `i*5`. O wartości `i` wiadomo, że jest liczbą zespoloną `ComplexNumber` i że definiuje metodę *, która wymaga drugiej wartości typu `ComplexNumber`. Literał `5.0` nie jest typu `ComplexNumber`, tylko `Double`. Kompilator rozpoczyna zatem wyszukiwanie domniemanej konwersji `Double => complexmath.ComplexNumber`, znajdując wreszcie konwersję `realToComplex` w obiekcie pakietowym. Następnie kompilator napotyka wyrażenie `(... : ComplexNumber) + 1.0`. Znajduje wtedy metodę + zdefiniowaną w `ComplexNumber`, która akceptuje drugi obiekt `ComplexNumber`. Wartość `1.0` ma typ `Double`, a nie `ComplexNumber`, zatem znowu rozpocznie się wyszukiwanie domniemanej konwersji `Double => ComplexNumber`. Oczywiście poszukiwania kończą się sukcesem, dzięki czemu ostatecznie jest zwracany wynik `ComplexNumber(1.0, 5.0)`.

Zauważ, że to wartość `i` powoduje uruchomienie obliczeń na liczbach zespolonych. Gdy tylko pojawia się liczba zespolona, kompilator znajduje odpowiednie konwersje, pozwalające na kompilację wyrażeń. Składnia jest elegancka i zwięzła, nie musieliśmy także importować żadnych konwersji. Minusem jest to, że w tej sytuacji jest konieczne odwołanie się na samym początku do wartości `i`, by została stworzona pierwsza liczba typu `ComplexNumber`. Zobaczmy, co się stanie, gdy `i` pojawi się pod koniec wyrażenia:

```
scala> val x = 1.0 + 5.0*i
<console>:6: error: overloaded method value * with alternatives:
  (Double)Double <and>
  (Float)Float <and>
  (Long)Long <and>
  (Int)Int <and>
  (Char)Int <and>
  (Short)Int <and>
  (Byte)Int
cannot be applied to (complexmath.ComplexNumber)
       val x = 1 + 5*i
```

Kompilator narzeka, ponieważ nie może znaleźć metody + zdefiniowanej w typie `Double`, która pobierałaby argument typu `ComplexNumber`. Ten problem można rozwiązać, importując domniemany widok `Double => ComplexNumber` do naszego zakresu:

```
scala> import complexmath.realToComplex
import complexmath.realToComplex

scala> val x = 1.0 + 5.0*i
x: complexmath.ComplexNumber = ComplexNumber(1.0,5.0)
```

Najpierw importujemy widok `realToComplex`. Teraz wyrażenie `1 + 5*i` daje oczekiwany wynik `ComplexNumber(1.0,5.0)`. Minusem jest to, że w zakresie typu `Double` pojawił się dodatkowy domniemany widok. Może to spowodować kłopoty, gdy zostaną zdefiniowane inne domniemane widoki o metodach podobnych do `ComplexNumber`. Zdefiniujmy nową domniemaną konwersję, która doda do typu `Double` metodę `imaginary`.

```
scala> implicit def doubleToReal(x : Double) = new {
     | def real = "Rzeczywista(" + x + ")"
     | }
doubleToReal: (x: Double)java.lang.Object{def real: java.lang.String}

scala> 5.0 real
<console>:10: error: type mismatch;
 found : Double
 required: ?{val real: ?}
Note that implicit conversions are not applicable
 because they are ambiguous:
 both method doubleToReal in object $iw of type
   (x: Double)java.lang.Object{def real: java.lang.String}
 and method realToComplex in package complexmath of type
   (r: Double)complexmath.ComplexNumber
 are possible conversion functions from
   Double to ?{val real: ?}
       5.0 real
```

Pierwsza instrukcja definiuje domniemany widok dla typu `Double`, który dodaje nowy typ zawierający metodę `real`. Metoda `real` zwraca wartość `Double` w postaci łańcucha znaków `String`. Kolejna linia to próba wywołania metody `real`, zakończona niepowodzeniem. Kompilator informuje o znalezieniu niejednoznacznych domniemanych konwersji. Problem polega na tym, że typ `ComplexNumber` również definiuje metodę `real`, zatem domniemana konwersja pomiędzy typami `Double => ComplexNumber` kłóci się z konwersją domniemaną `doubleToReal`. Konfliktu można uniknąć, rezygnując z importowania konwersji `Double => ComplexNumber`:

```
scala> import complexmath.i
import complexmath.i

scala> implicit def doubleToReal(x : Double) = new {
     | def real = " Rzeczywista(" + x + ")"
     | }
doubleToReal: (x: Double)java.lang.Object{def real: java.lang.String}

scala> 5.0 real
res0: java.lang.String = Rzeczywista(5.0)
```

Rozpoczynamy tu nową sesję REPL, w której importujemy jedynie `complexmath.i`. Kolejna instrukcja redefiniuje konwersję `doubleToReal`. Teraz wyrażenie `5.0 real` kompiluje się poprawnie, ponieważ nie występuje konflikt.

Takie konstrukcje pozwalają na tworzenie ekspresywnego kodu bez niebezpieczeństwa konfliktu pomiędzy domniemanymi przekształceniami. Można tu zaproponować następujący wzorzec:

- Zdefiniuj podstawowe abstrakcje dla biblioteki, takie jak klasa `ComplexNumber`.
- Zdefiniuj domniemane konwersje niezbędne do powstania ekspresywnego kodu w jednym z typów powiązanych konwersją. Konwersja `Double => ComplexNumber` została zdefiniowana w obiekcie pakietowym `complexmath`, powiązanym z typem `ComplexNumber`, dzięki czemu jest odkrywalna w kodzie korzystającym z typu `ComplexNumber`.
- Zdefiniuj *punkt wejścia* do biblioteki, na podstawie którego ujednoznaczniane będą domniemane konwersje. W przypadku biblioteki `complexmath` punktem wejścia jest wartość `i`.
- W niektórych sytuacjach nadal jest konieczne jawne zaimportowanie widoku. W bibliotece `complexmath` punkt wejścia `i` pozwala na konstruowanie pewnych typów wyrażeń, jednak inne typy, choć intuicyjnie wydają się poprawne, nie zadziałają. Przykładowo `(i * 5.0 + 1.0)` jest akceptowane, a `(1.0 + 5.0*i)` nie. W tej sytuacji można zaimportować konwersję z dobrze znanej lokalizacji. W `complexmath` tę lokalizację stanowi obiekt pakietowy.

Trzymając się powyższych wytycznych, będziesz w stanie tworzyć API, które będą nie tylko ekspresywne, ale także odkrywalne.

5.5. Podsumowanie

Ten rozdział był poświęcony kwestii domniemanych wartości i widoków oraz mechanizmowi ich wyszukiwania. Wartości domniemane wykorzystuje się do przekazywania parametrów wywołaniom metod. Domniemane widoki służą do konwersji pomiędzy typami oraz do wywoływania metod na zmiennych, których oryginalny typ na to nie pozwala. Wyszukiwanie domniemanych parametrów i widoków jest oparte na tym samym mechanizmie. Proces odnajdywania domniemanej encji przebiega dwuetapowo. Najpierw jest podejmowana próba odnalezienia encji, które nie mają prefiksu w bieżącym zakresie. Drugi etap to sprawdzenie obiektów towarzyszących typom powiązanym. Domniemane encje pozwalają na rozszerzanie istniejących klas. Dodatkowo można je połączyć z parametrami domyślnymi w celu uproszczenia wywołań metod i powiązania zachowania z zakresem domniemanej wartości.

Najistotniejsze jest to, że domniemania to potężne narzędzie, które powinno być stosowane rozsądnie. Kluczem do sukcesu jest ograniczanie zakresu domniemanych encji i definiowanie ich w dobrze znanych lub łatwo odkrywalnych lokalizacjach. Można osiągnąć ten cel, zapewniając jednoznaczne punkty wejścia dla domniemanych konwersji oraz ekspresywne API. Domniemane encje w bardzo ciekawy sposób łączą się z systemem typów Scali. Wrócimy do tego tematu w rozdziale 7., na razie zajmijmy się samym systemem typów.

Skorowidz

E

F

G

H

I

J

K

L

Ł

M

N

O

P

R

S

Ś

T

U

W

Z

Programuj w języku Scala!

Scala to słowo, które ostatnio nie schodzi z ust programistów Javy. Pod tą nazwą kryje się język łączący światy programowania funkcyjnego i obiektowego. Jego ogromną zaletą jest działanie oparte na wirtualnej maszynie Javy. Pozwala to między innymi na bezproblemową komunikację i współdzielenie kodu między obomą językami. James Gosling, twórca Javy, zapytany o to, jakiego języka oprogramowania działającego w ten sposób użyłby obecnie (gdyby nie mógł wykorzystać Javy), odparł bez zastanowienia: „Scali!". To chyba najlepiej dowodzi, że ten język jest wart Twojego czasu!

Dzięki tej książce opanujesz Scalę szybko i bezboleśnie, więc będziesz mógł wykorzystać jej zalety już w najbliższym projekcie. W trakcie lektury poznasz składnię, fundamentalne zasady tworzenia oprogramowania w Scali oraz konwencje kodowania w tym języku. Z kolejnych rozdziałów dowiesz się, czym są niejawne widoki, jakie typy danych masz do dyspozycji i jakie są ich ograniczenia. Co jeszcze? Integracja Scali z Javą — to bardzo istotny temat. Masz pole do popisu! Ponadto poznasz wzorce stosowane w programowaniu funkcyjnym.

Słowo wstępne do tej niezwykłej książki napisał sam Martin Odersky — twórca języka Scala! Niniejsza książka jest najlepszym kompendium wiedzy na temat programowania w tym języku. Musisz ją mieć!

Poznaj:

- konwencje obowiązujące w Scali
- składnię języka
- najlepsze wzorce projektowe stosowane w programowaniu funkcyjnym
- potencjał języka Scala!

ISBN 978-83-246-5188-7

9 788324 651887