Samouczek Scali
dla programistów Javy

Wersja 1.3
3 listopada 2009

Michel Schinz, Philipp Haller – przełożył Grzegorz Balcerek

PROGRAMMING METHODS LABORATORY
EPFL
SWITZERLAND

1 Wstęp

Ten dokument daje szybkie wprowadzenie do języka Scala i jego kompilatora. Jest przeznaczony dla osób, które mają już jakieś doświadczenie programistyczne i chcą mieć przegląd tego, co mogą zrobić w Scali. Zakładana jest podstawowa wiedza o programowaniu obiektowym, zwłaszcza w Javie.

2 Pierwszy przykład

Jako pierwszego przykładu użyjemy standardowego programu *Witaj świecie*. Nie jest on bardzo fascynujący, ale pozwala w prosty sposób zademonstrować sposób użycia narzędzi Scali bez posiadania zbyt wielkiej wiedzy o języku. Oto jak on wygląda:

```
object HelloWorld {
  def main(args: Array[String]) {
    println("Hello, world!")
  }
}
```

Struktura tego programu powinna być znajoma programistom języka Java: składa się z jednej metody o nazwie `main`, która pobiera argumenty wiersza poleceń, tablicę łańcuchów znakowych, jako parametr; ciało tej metody składa się z pojedynczego wywołania predefiniowanej metody `println` z przyjaznym powitaniem jako argumentem. Metoda `main` nie zwraca żadnej wartości (metoda jest procedurą). Wobec tego nie jest konieczne deklarowanie typu rezultatu.

Mniej znana dla programistów języka Java jest deklaracja **`object`** zawierająca metodę `main`. Taka deklaracja wprowadza coś, co jest powszechnie znane jako *obiekt singleton*, to jest klasę z pojedynczą instancją. Tak więc powyższa deklaracja deklaruje zarówno klasę o nazwie `HelloWorld`, jak i instancję tej klasy, również nazywaną `HelloWorld`. Ta instancja jest tworzona na żądanie, kiedy jest użyta po raz pierwszy.

Uważny czytelnik mógł zauważyć, że metoda `main` nie jest zadeklarowana jako `static`. Jest tak dlatego, że składowe statyczne (metody lub pola) nie istnieją w Scali. Zamiast definiowania składowych statycznych, programista Scali deklaruje te składowe w obiektach singletonach.

2.1 Kompilowanie przykładu

W celu skompilowania przykładu, używamy `scalac`, kompilatora Scali. `scalac` działa jak większość kompilatorów: pobiera plik źródłowy jako argument, być może jakieś opcje, oraz produkuje jeden lub kilka plików obiektowych. Wytworzone przez niego pliki obiektowe są standardowymi plikami klas Javy.

Jeśli zapiszemy powyższy program w pliku o nazwie `HelloWorld.scala`, możemy

skompilować go wydając następującą komendę (znak większości '>' reprezentuje znak zachęty powłoki i nie powinien być wpisywany):

```
> scalac HelloWorld.scala
```

Zostanie wygenerowane kilka plików klas w bieżącym katalogu. Jeden z nich będzie się nazywał `HelloWorld.class` i zawiera klasę, która może być wykonana bezpośrednio za pomocą komendy `scala`, jak pokazuje następna sekcja.

2.2 Uruchamianie przykładu

Po skompilowaniu, program w Scali może być uruchamiany przy pomocy komendy `scala`. Używa się jej podobnie do komendy `java` używanej do uruchamiania programów w Javie i przyjmuje ona te same opcje. Powyższy przykład może być uruchomiony przy pomocy następującej komendy, która produkuje oczekiwane dane wyjściowe:

```
> scala -classpath . HelloWorld

Hello, world!
```

3 Interakcja z językiem Java

Jedną z silnych stron Scali jest to, że interakcja z kodem języka Java jest w nim bardzo łatwa. Wszystkie klasy z pakietu `java.lang` są domyślnie zaimportowane, podczas gdy inne muszą być zaimportowane jawnie.

Spójrzmy na demonstrujący to przykład. Chcemy otrzymać i sformatować bieżącą datę, zgodnie z konwencją używaną w określonym kraju, powiedzmy we Francji (inne regiony, takie jak francuskojęzyczna część Szwajcarii, używają tej samej konwencji).

Biblioteki klas Javy definiują klasy narzędziowe o dużych możliwościach, takie jak `Date` i `DateFormat`. Ponieważ Scala bezproblemowo współdziała z Javą, nie ma potrzeby implementowania odpowiednich klas w bibliotece klas Scali–możemy po prostu zaimportować klasy odpowiednich pakietów Javy:

```
import java.util.{Date, Locale}
import java.text.DateFormat
import java.text.DateFormat._

object FrenchDate {
  def main(args: Array[String]) {
    val now = new Date
    val df = getDateInstance(LONG, Locale.FRANCE)
```

```
      println(df format now)
    }
  }
```

Polecenie importu Scali wygląda bardzo podobnie do swojego odpowiednika w Javie, jednak posiada większe możliwości. Wiele klas może być zaimportowanych z tego samego pakietu poprzez zawarcie ich w nawiasach klamrowych, jak w pierwszym wierszu. Kolejną różnicą jest to, że gdy importowane są wszystkie nazwy z pakietu lub klasy, używa się znaku podkreślenia (_) zamiast gwiazdki (*). Jest tak dlatego, że gwiazdka jest prawidłowym identyfikatorem Scali (np. nazwą metody), jak zobaczymy później.

Polecenie importu w trzecim wierszu importuje zatem wszystkie składowe klasy `DateFormat`. W ten sposób statyczna metoda `getDateInstance` i statyczne pole `LONG` stają się widoczne bezpośrednio.

Wewnątrz metody `main`, najpierw tworzymy instancję klasy Javy `Date`, która domyślnie zawiera datę bieżącą. Następnie definiujemy format daty, używając statycznej metody `getDateInstance`, zaimportowanej wcześniej. W końcu drukujemy datę bieżącą sformatowaną według zlokalizowanej instancji `DateFormat`. Ten ostatni wiersz pokazuje interesującą właściwość składni Scali. Metody pobierające jeden argument mogą być używane przy użyciu notacji infiksowej. To znaczy, wyrażenie

```
df format now
```

jest tylko innym, trochę mniej rozwlekłym, sposobem zapisu wyrażenia

```
df.format(now)
```

Może się to wydawać tylko mało istotnym detalem składniowym, ale ma to ważne konsekwencje, z których jedna zostanie zbadana w następnej sekcji.

Na zakończenie tej sekcji o integracji z Javą powinno być wspomniane, że jest również możliwe dziedziczenie z klas Javy i implementacja interfejsów Javy bezpośrednio w Scali.

4 Wszystko jest obiektem

Scala jest czystym językiem zorientowanym obiektowo w tym sensie, że *wszystko* jest obiektem, włącznie z liczbami czy funkcjami. Różni się w tym od Javy, ponieważ Java odróżnia typy proste (takie jak `boolean` i `int`) od typów referencyjnych i nie daje możliwości manipulowania funkcjami jak wartościami.

4.1 Liczby są obiektami

Jako że liczby są obiektami, mają również metody. I faktycznie, wyrażenie arytmetyczne takie jak to:

```
1 + 2 * 3 / x
```

składa się wyłącznie z wywołań metod, bowiem jest ono równoważne następującemu wyrażeniu, jak widzieliśmy w poprzedniej sekcji:

```
(1).+(((2).*(3))./(x))
```

To oznacza również, że +, * itd. są ważnymi identyfikatorami w Scali.

Nawiasy naokoło liczb w drugiej wersji są potrzebne, bo lekser Scali używa reguły najdłuższego dopasowania dla symboli. Tak więc rozbiłby następujące wyrażenie:

```
1.+(2)
```

na symbole `1.`, `+` i `2`. Powodem dla którego takie rozbicie na symbole jest wybierane jest to, że `1.` jest dłuższym prawidłowym dopasowaniem niż `1`. Symbol `1.` jest interpretowany jako literał `1.0`, co robi z niego `Double`, a nie `Int`. Zapisanie wyrażenia jako:

```
(1).+(2)
```

zapobiega przed interpretacją `1` jako `Double`.

4.2 Funkcje są obiektami

Być może bardziej zaskakujące dla programisty Javy jest to, że funkcje są również obiektami w Scali. Jest więc możliwe przekazywanie funkcji jako argumentów, przechowywanie ich w zmiennych i zwracanie ich z innych funkcji. Ta umiejętność manipulowania funkcjami jak wartościami jest jednym z kamieni węgielnych bardzo interesującego paradygmatu programistycznego zwanego *programowaniem funkcyjnym*.

Jako bardzo prosty przykład tego, dlaczego może być użyteczne użycie funkcji jako wartości, rozważmy funkcję czasomierza, której celem jest wykonanie jakiejś akcji co sekundę. Jak przekazać jej akcję do wykonania? Całkiem logicznie, jako funkcję. Ten bardzo prosty rodzaj przekazywania funkcji powinien być znany wielu programistom: jest on często używany w kodzie interfejsu użytkownika do rejestracji funkcji zwrotnych, które są wywoływane gdy zachodzi jakieś zdarzenie.

W poniższym programie funkcja czasomierza nazywa się `oncePerSecond` i pobiera funkcję zwrotną jako argument. Typ tej funkcji jest zapisywany `() => Unit` i jest typem wszystkich funkcji, które nie pobierają argumentów i niczego nie zwracają (typ `Unit` jest podobny do `void` w C/C++). Główna funkcja tego programu po prostu

woła tę funkcję czasomierza z funkcją zwrotną, która drukuje zdanie na terminalu. Innymi słowy, ten program bez końca drukuje zdanie “time flies like an arrow” co sekundę.

```
object Timer {
  def oncePerSecond(callback: () => Unit) {
    while (true) { callback(); Thread sleep 1000 }
  }
  def timeFlies() {
    println("time flies like an arrow...")
  }
  def main(args: Array[String]) {
    oncePerSecond(timeFlies)
  }
}
```

Zauważ, że w celu wydrukowania łańcucha znaków, użyliśmy predefiniowanej metody `println` zamiast użycia tej z `System.out`.

4.2.1 Funkcje anonimowe

Chociaż ten program jest łatwy do zrozumienia, może być trochę udoskonalony. Przede wszystkim zauważ, że funkcja `timeFlies` jest zdefiniowana tylko po to, aby zostać później przekazana do funkcji `oncePerSecond`. Konieczność nazwania tej funkcji, która jest użyta tylko raz, mogłaby się wydawać niepotrzebna i byłoby rzeczywiście miło móc skonstruować tę funkcję dokładnie w momencie, gdy jest przekazywana do `oncePerSecond`. Jest to możliwe w języku Scala przy użyciu *funkcji anonimowych*, które są dokładnie tym: funkcjami bez nazwy. Poprawiona wersja naszego programu czasomierza, z użyciem funkcji anonimowej zamiast `timeFlies`, wygląda w ten sposób:

```
object TimerAnonymous {
  def oncePerSecond(callback: () => Unit) {
    while (true) { callback(); Thread sleep 1000 }
  }
  def main(args: Array[String]) {
    oncePerSecond(() =>
      println("time flies like an arrow..."))
  }
}
```

Obecność funkcji anonimowej w tym przykładzie jest ujawniona przez strzałkę w prawo ‘=>’, która oddziela listę argumentów funkcji od jej ciała. W tym przykładzie lista argumentów jest pusta, czego świadectwem jest pusta para nawiasów po lewej stronie strzałki. Ciało funkcji jest takie samo, jak to z `timeFlies` wyżej.

5 Klasy

Jak widzieliśmy wyżej, Scala jest językiem zorientowanym obiektowo i jako taki posiada pojęcie klasy (dla całości obrazu warto zauważyć, że niektóre języki zorientowane obiektowo nie mają pojęcia klasy, ale Scala nie jest jednym z nich). Klasy w Scali są deklarowane przy użyciu składni zbliżonej do składni Javy. Jedną ważną różnicą jest to, że klasy w Scali mogą mieć parametry. Jest to zilustrowane następującą definicją liczb zespolonych.

```
class Complex(real: Double, imaginary: Double) {
  def re() = real
  def im() = imaginary
}
```

Ta klasa liczb zespolonych zawiera dwa argumenty, które są rzeczywistą (`real`) i urojoną (`imaginary`) częścią liczby zespolonej. Te argumenty muszą być podane podczas tworzenia instancji klasy `Complex`, następująco: **`new`** `Complex(1.5, 2.3)`. Klasa zawiera dwie metody, nazwane `re` i `im`, które dają dostęp do tych dwóch części.

Należy odnotować, że typ rezultatu tych dwóch metod nie jest jawnie podany. Będzie on wywnioskowany automatycznie przez kompilator, który patrzy na prawą stronę tych metod i wnioskuje, że obie zwracają wartość typu `Double`.

Kompilator nie zawsze jest zdolny wywnioskować typy jak robi to tutaj i nie ma niestety prostej reguły, która pozwalałaby stwierdziż dokładnie kiedy będzie, a kiedy nie. W praktyce zwykle nie jest to problem, jako że kompilator skarży się, gdy nie jest w stanie wywnioskować typu nie podanego jawnie. Początkujący programiści Scali powinni, stosując to jako prostą regułę, próbować omijać deklaracje typów, które wydają się być łatwe do wywnioskowania z kontekstu i zobaczyć czy kompilator się z tym zgodzi. Po jakimś czasie programista powinien mieć dobre wyczucie kiedy omijać typy, a kiedy jawnie je specyfikować.

5.1 Metody bez argumentów

Mały problem z metodami `re` i `im` jest taki, że aby je wywołać, trzeba umieszczać pustą parę nawiasów po ich nazwie, jak pokazuje następujący przykład:

```
object ComplexNumbers {
  def main(args: Array[String]) {
    val c = new Complex(1.2, 3.4)
    println("imaginary part: " + c.im())
  }
}
```

Byłoby milej móc mieć dostęp do rzeczywistej i urojonej części jak gdyby były to pola, bez umieszczania pustej pary nawiasów. Jest to zupełnie wykonalne w Scali po prostu poprzez zdefiniowanie ich jako metod *bez argumentów*. Takie metody

różnią się od metod z zerową liczbą argumentów tym, że nie mają nawiasów po swojej nazwie ani w ich definicji, ani przy ich użyciu. Nasza klasa `Complex` może być przepisana następująco:

```
class Complex(real: Double, imaginary: Double) {
  def re = real
  def im = imaginary
}
```

5.2 Dziedziczenie i nadpisywanie

Wszystkie klasy w Scali dziedziczą z nadklasy. Kiedy żadna nadklasa nie jest wyspecyfikowana, tak jak w przykładzie klasy `Complex` z poprzedniej sekcji, domyślnie użyta jest `scala.AnyRef`.

W Scali jest możliwe nadpisanie metod dziedziczonych z nadklasy. Jest jednak obowiązkowe podkreślenie faktu, że metoda nadpisuje inną, przy użyciu modyfikatora **`override`**, w celu uniknięcia przypadkowego nadpisania. Jako przykład, nasza klasa `Complex` może być powiększona o przedefiniowanie metody `toString`, dziedziczonej z `Object`.

```
class Complex(real: Double, imaginary: Double) {
  def re = real
  def im = imaginary
  override def toString() =
    "" + re + (if (im < 0) "" else "+") + im + "i"
}
```

6 Klasy przypadków i dopasowanie wzorców

Rodzajem struktury danych, która często występuje w programach jest drzewo. Na przykład interpretery i kompilatory zwykle wewnętrznie reprezentują programy jako drzewa; dokumenty XML są drzewami; a także różne rodzaje pojemników są oparte na drzewach, jak drzewa czerwono-czarne.

Sprawdzimy teraz jak takie drzewa są reprezentowane i przetwarzane w Scali przy pomocy małego programu kalkulatora. Celem tego programu jest przetwarzanie bardzo prostych wyrażeń arytmetycznych złożonych z sum, stałych całkowitych i zmiennych. Dwoma przykładami takich wyrażeń są $1+2$ i $(x+x)+(7+y)$.

Najpierw musimy podjąć decyzję co do reprezentacji takich wyrażeń. Najbardziej naturalną jest drzewo, którego węzłami są operacje (tutaj dodawanie), a liśćmi wartości (tutaj stałe i zmienne).

W Javie takie drzewo byłoby reprezentowane przy użyciu abstrakcyjnej nadklasy dla drzew oraz jednej konkretnej podklasy dla węzła lub liścia. W języku programowa-

nia funkcyjnego użyto by w tym samym celu algebraicznego typu danych. Scala dostarcza pojęcie *klas przypadków* (ang. **case class**), które jest pojęciem pośrednim. Oto jak mogą być one użyte do zdefiniowania typu drzew w naszym przykładzie:

```
abstract class Tree
case class Sum(l: Tree, r: Tree) extends Tree
case class Var(n: String) extends Tree
case class Const(v: Int) extends Tree
```

Fakt, że klasy `Sum`, `Var` i `Const` są zadeklarowane jako klasy przypadku oznacza, że różnią się one od standardowych klas pod kilkoma względami:

- słowo kluczowe **new** nie jest obowiązkowe aby stworzyć instancje tych klas (tj. można napisać `Const(5)` zamiast **new** `Const(5)`),
- funkcje dostępowe są automatycznie definiowane dla parametrów konstruktora (tj. jest możliwy dostęp do wartości parametru `v` konstruktora jakieś instancji `c` klasy `Const` poprzez napisanie po prostu `c.v`),
- są zapewniane domyślne implementacje metod `equals` i `hashCode`, które działają na *strukturze* instancji, a nie na ich tożsamości,
- jest zapewniona domyślna definicja metody `toString`, która drukuje wartość w "formie źródłowej" (np. drzewo dla wyrażenia $x + 1$ jest drukowane jako `Sum(Var(x),Const(1))`),
- instancje tych klas mogą być dekomponowane przy użyciu *dopasowania wzorców*, jak zobaczymy poniżej.

Teraz, kiedy zdefiniowaliśmy typ danych do reprezentowania naszych wyrażeń arytmetycznych, możemy zacząć definiować operacje do ich przetwarzania. Zaczniemy od funkcji, która da rozwiązanie wyrażenia w jakimś *środowisku*. Celem środowiska jest danie wartości zmiennym. Na przykład, wyrażenie $x + 1$ ewaluowane w środowisku, które przypisuje wartość 5 do zmiennej x, zapisane jako $\{x \rightarrow 5\}$, daje w rezultacie 6.

Musimy więc znaleźć sposób reprezentowania środowisk. Moglibyśmy oczywiście użyć jakiejś skojarzeniowej struktury danych, jak tablica haszująca, ale możemy również bezpośrednio użyć funkcji! Środowisko to w rzeczywistości nic więcej niż funkcja, która przypisuje nazwie (zmiennej) wartość. Powyższe środowisko $\{x \rightarrow 5\}$ może być prosto zapisane w Scali w następujący sposób:

```
{ case "x" => 5 }
```

Ta notacja definiuje funkcję, która gdy poda się jej łańcuch znaków `"x"` jako argument, zwraca liczbę całkowitą 5 i wywołuje wyjątek w innym przypadku.

Przed napisaniem funkcji ewaluacyjnej, nadajmy nazwę typowi środowisk. Moglibyśmy oczywiście zawsze używać typu `String => Int` dla środowisk, ale jeśli wpro-

wadzimy nazwę tego typu, uprości to program i ułatwi przyszłe zmiany. W Scali jest to realizowane za pomocą następującej notacji:

```
type Environment = String => Int
```

Od tego momentu, typ `Environment` może być użyty jako alias typu funkcji ze `String` do `Int`.

Możemy teraz zdefiniować funkcję ewaluacyjną. Koncepcyjnie jest to bardzo proste: wartość sumy dwóch wyrażeń jest po prostu sumą wartości tych wyrażeń; wartość zmiennej jest otrzymywana bezpośrednio ze środowiska; a wartość stałej jest tą stałą. Wyrażenie tego w Scali nie jest wiele trudniejsze:

```
def eval(t: Tree, env: Environment): Int = t match {
  case Sum(l, r) => eval(l, env) + eval(r, env)
  case Var(n)    => env(n)
  case Const(v)  => v
}
```

Ta funkcja ewaluacyjna działa wykonując *dopasowanie wzorców* na drzewie `t`. Intuicyjnie znaczenie powyższej definicji powinno być jasne:

1. na początku sprawdza ona, czy drzewo `t` jest sumą (`Sum`) i jeśli jest, wiąże lewe poddrzewo z nową zmienną nazwaną `l`, a prawe poddrzewo ze zmienną nazwaną `r`, a następnie kontynuuje ewaluując wyrażenia po strzałce; to wyrażenie może (i robi to) użyć zmiennych związanych przez wzorzec pojawiający się po lewej stronie strzałki, tj. `l` i `r`,
2. jeśli pierwsze sprawdzenie nie udaje się, to jest jeśli drzewo nie jest sumą (`Sum`), funkcja kontynuuje i sprawdza czy `t` jest zmienną (`Var`); jeśli jest, wiąże nazwę zawartą w węźle `Var` ze zmienną `n` i kontynuuje z wyrażeniem po prawej stronie,
3. jeśli drugie sprawdzenie również zawodzi, to jest jeśli `t` nie jest ani sumą (`Sum`), ani zmienną (`Var`), funkcja sprawdza czy jest to stała (`Const`) i jeśli tak jest, wiąże wartość zawartą w węźle `Const` ze zmienną `v` i przechodzi do prawej strony,
4. w końcu, jeśli wszystkie sprawdzenia zawodzą, wywoływany jest wyjątek, aby zasygnalizować porażkę wyrażenia dopasowania wzorca; może się to zdarzyć tutaj tylko, gdyby zostało zadeklarowanych więcej podklas klasy `Tree`.

Widzimy, że podstawową myślą dopasowania wzorców jest próba dopasowania wartości do szeregu wzorców i jak tylko wzorzec pasuje, wyciągnięcie i nazwanie różnych części wartości, a w końcu ewaluacja jakiegoś kodu, który zwykle używa tych nazwanych części.

Wytrawny programista zorientowany obiektowo mógłby się zastanawiać, dlaczego nie zdefiniowaliśmy `eval` jako *metody* klasy `Tree` i jej podklas. Rzeczywiście, mogli-

byśmy tak zrobić, jako że Scala zezwala na definicje metod w klasach przypadków tak jak w normalnych klasach. Decyzja, czy użyć dopasowania wzorców, czy metod jest wobec tego kwestią smaku, ale ma również ważne implikacje związane z rozszerzalnością:

- kiedy używamy metod, łatwo jest dodać nowy rodzaj węzła, jako że może to być zrobione po prostu poprzez zdefiniowanie dla niego podklasy klasy `Tree`; z drugiej strony, dodanie nowej operacji przetwarzania drzewa jest nużące, jako że wymaga modyfikacji wszystkich podklas klasy `Tree`,
- kiedy używamy dopasowania wzorców, sytuacja jest odwrotna: dodanie nowego rodzaju węzła wymaga modyfikacji wszystkich funkcji, które wykonują dopasowania wzorców na drzewie, aby wzięły pod uwagę nowy węzeł; z drugiej strony, dodanie nowej operacji jest łatwe, po prostu przez zdefiniowanie jej jako niezależnej funkcji.

Aby dalej zgłębić dopasowanie wzorców, zdefiniujmy inną operację na wyrażeniach arytmetycznych: symboliczne obliczanie pochodnej. Czytelnik może pamiętać następujące reguły dotyczące tej operacji:

1. pochodna sumy jest sumą pochodnych,
2. pochodna jakiejś zmiennej v jest równa jeden, jeśli v jest zmienną względem której jest obliczana pochodna i zero w przeciwnym przypadku,
3. pochodna stałej jest równa zero.

Te reguły mogą być przetłumaczone prawie dosłownie na kod w Scali, uzyskując następującą definicję:

```
def derive(t: Tree, v: String): Tree = t match {
  case Sum(l, r) => Sum(derive(l, v), derive(r, v))
  case Var(n) if (v == n) => Const(1)
  case _ => Const(0)
}
```

Ta funkcja wprowadza dwa nowe pojęcia związane z dopasowaniem wzorców. Po pierwsze, wyrażenie **case** dla zmiennych ma *strażnika*, wyrażenie występujące po słowie kluczowym **if**. Ten strażnik zapobiega temu, aby dopasowanie wzorców powiodło się, chyba że jego wyrażenie jest prawdziwe. Tutaj jest użyty w celu zapewnienia, że zwracamy stałą 1 tylko jeśli nazwa zmiennej, której pochodną obliczamy jest taka sama, jak zmienna względem której obliczamy pochodną. Drugą nową cechą dopasowania wzorców tu użytą jest *wieloznacznik*, zapisywany _, który jest wzorcem dopasowującym dowolną wartość, bez nadawania jej nazwy.

Nie zgłębiliśmy jeszcze wszystkich możliwości dopasowania wzorców, ale zatrzymamy się tutaj w celu uzyskania minimalistycznego podejścia tego dokumentu.

Ciągle chcemy zobaczyć jak powyższe dwie funkcje zachowują się w rzeczywistym przykładzie. W tym celu napiszmy prostą funkcję `main`, która wykonuje kilka operacji na wyrażeniu $(x + x) + (7 + y)$: najpierw oblicza jej wartość w środowisku $\{x \to 5, y \to 7\}$, potem oblicza jej pochodną względem x, a potem y.

```
def main(args: Array[String]) {
  val exp: Tree = Sum(Sum(Var("x"),Var("x")),Sum(Const(7),Var("y")))
  val env: Environment = { case "x" => 5 case "y" => 7 }
  println("Expression: " + exp)
  println("Evaluation with x=5, y=7: " + eval(exp, env))
  println("Derivative relative to x:\n " + derive(exp, "x"))
  println("Derivative relative to y:\n " + derive(exp, "y"))
}
```

Wykonując ten program, dostajemy spodziewane dane wyjściowe:

```
Expression: Sum(Sum(Var(x),Var(x)),Sum(Const(7),Var(y)))
Evaluation with x=5, y=7: 24
Derivative relative to x:
 Sum(Sum(Const(1),Const(1)),Sum(Const(0),Const(0)))
Derivative relative to y:
 Sum(Sum(Const(0),Const(0)),Sum(Const(0),Const(1)))
```

Sprawdzając dane wyjściowe widzimy, że rezultat pochodnej powinien być uproszczony przed prezentacją użytkownikowi. Zdefiniowanie prostej funkcji upraszczającej przy użyciu dopasowania wzorców jest interesującym (ale zaskakująco skomplikowanym) problemem, zostawionym jako ćwiczenie dla czytelnika.

7 Cechy

Poza dziedziczeniem kodu z nadklasy, klasa Scali może również zaimportować kod z jednej lub wielu *cech* (ang. trait).

Być może najłatwiejszym sposobem dla programisty Javy żeby zrozumieć czym są cechy jest spojrzenie na nie jak na interfejsy, które mogą również zawierać kod. W Scali, kiedy klasa dziedziczy z cechy, implementuje interfejs tej cechy oraz dziedziczy cały kod zawarty w cesze.

Aby zobaczyć użyteczność cech, spójrzmy na klasyczny przykład: uporządkowane obiekty. Często jest przydatne mieć możliwość porównania obiektów danej klasy pomiędzy sobą, na przykład aby je posortować. W Javie obiekty, które są porównywalne implementują interfejs `Comparable`. W Scali możemy to zrobić trochę lepiej niż w Javie, definiując cechę równoważną klasie `Comparable` jako cechę, którą nazwiemy `Ord`.

Podczas porównywania obiektów, przydatnych może być sześć różnych predykatów: mniejszy, mniejszy lub równy, równy, różny, większy lub równy i większy. Jednak definiowanie ich wszystkich jest drobiazgowe, szczególnie dlatego, że cztery z tych sześciu może być wyrażone przy użyciu pozostałych dwóch. To jest, mając dane predykaty równy i mniejszy (na przykład), możliwe jest wyrażenie pozostałych. W Scali, wszystkie te spostrzeżenia mogą być ładnie ujęte przez następującą deklarację cechy:

```
trait Ord {
  def < (that: Any): Boolean
  def <=(that: Any): Boolean = (this < that) || (this == that)
  def > (that: Any): Boolean = !(this <= that)
  def >=(that: Any): Boolean = !(this < that)
}
```

Ta definicja tworzy zarówno nowy typ nazwany `Ord`, który gra tą samą rolę jak interfejs `Comparable` Javy, jak i domyślne implementacje trzech predykatów za pomocą czwartego, abstrakcyjnego. Predykaty równości i różności nie pojawiają się tutaj, ponieważ są domyślnie obecne we wszystkich obiektach.

Typ `Any`, który jest użyty powyżej, jest typem, który jest nadtypem wszystkich innych typów w Scali. Można go widzieć jako bardziej ogólną wersję typu `Object` Javy, ponieważ jest on również nadtypem typów prostych, takich jak `Int`, `Float` itd.

Aby dodać do obiektów klasy możliwość ich porównywania wystarczy więc zdefiniować predykaty, które testują równość i podrzędność i wmieszać powyższą klasę `Ord`. Jako przykład zdefiniujmy klasę `Date`, reprezentującą daty w kalendarzu gregoriańskim. Takie daty są złożone z dnia, miesiąca i roku, które będziemy reprezentować jako liczby całkowite. Rozpoczynamy zatem definicję klasy `Date` następująco:

```
class Date(y: Int, m: Int, d: Int) extends Ord {
  def year = y
  def month = m
  def day = d

  override def toString(): String = year + "-" + month + "-" + day
```

Ważna jest tutaj deklaracja **`extends`** `Ord`, która występuje po nazwie klasy i parametrach. Deklaruje ona, że klasa `Date` dziedziczy z cechy `Ord`.

Następnie przedefiniowujemy metodę `equals`, odziedziczoną z `Object`, aby poprawnie porównywała daty poprzez porównanie ich poszczególnych pól. Domyślna implementacja `equals` nie nadaje się do użycia, ponieważ, tak jak w Javie, porównuje obiekty fizycznie. Dochodzimy do następującej definicji:

```
  override def equals(that: Any): Boolean =
    that.isInstanceOf[Date] && {
      val o = that.asInstanceOf[Date]
```

```
    o.day == day && o.month == month && o.year == year
  }
```

Ta metoda używa predefiniowanych metod `isInstanceOf` i `asInstanceOf`. Pierwsza, `isInstanceOf`, odpowiada operatorowi `instanceof` Javy i zwraca true wtedy i tylko wtedy, gdy obiekt na którym jest zastosowana jest instancją danego typu. Druga, `asInstanceOf`, odpowiada operatorowi rzutowania języka Java: jeśli obiekt jest instancją danego typu, jest widziany jako taki, a w przeciwnym przypadku wyrzucany jest wyjątek `ClassCastException`.

Wreszcie ostatnią metodą do zdefiniowania jest predykat, który testuje podrzędność, w sposób następujący. Używa ona innej predefiniowanej metody, `error`, która wyrzuca wyjątek z podanym komunikatem błędu.

```
def <(that: Any): Boolean = {
  if (!that.isInstanceOf[Date])
    error("cannot compare " + that + " and a Date")

  val o = that.asInstanceOf[Date]
  (year < o.year) ||
  (year == o.year && (month < o.month ||
                      (month == o.month && day < o.day)))
}
```

To kończy definicję klasy `Date`. Instancje tej klasy mogą być widziane bądź jako daty, bądź jako porównywalne obiekty. Co więcej, wszystkie one definiują sześć wspomnianych wyżej predykatów porównywania: `equals` i `<`, ponieważ znajdują się one bezpośrednio w definicji klasy `Date`, a pozostałe ponieważ są one odziedziczone z cechy `Ord`.

Cechy są oczywiście użyteczne w innych sytuacjach niż te pokazane tutaj, ale długie dyskutowanie ich zastosowań jest poza zakresem tego dokumentu.

8 Generyczność

Ostatnią cechą charakterystyczną Scali, którą zbadamy w tym samouczku, jest generyczność. Programiści Javy powinni być doskonale świadomi problemów związanych z brakiem generyczności w ich języku, wadą którą zajęto się w Javie 1.5.

Generyczność to możliwość pisania kodu parametryzowanego typami. Na przykład, programista piszący bibliotekę dla list łączonych staje przed problemem decyzji jaki typ nadać elementom listy. Ponieważ ta lista jest przeznaczona do bycia używaną w wielu różnych kontekstach, nie jest możliwe zdecydowanie, że typem elementów ma być, powiedzmy, `Int`. Byłoby to zupełnie arbitralne i zbyt restrykcyjne.

Programiści Javy uciekają się do użycia klasy `Object`, która jest nadtypem wszystkich obiektów. Jednak to rozwiązanie jest dalekie od bycia idealnym, ponieważ nie działa

dla typów prostych (`int`, `long`, `float` itd.) oraz powoduje, że wiele dynamicznych rzutowań typu musi być wstawionych przez programistę.

Aby rozwiązać ten problem, Scala umożliwia zdefiniowanie klas (i metod) generycznych. Zbadajmy to na przykładzie najprostszej możliwej klasy-pojemnika: referencji, która może być albo pusta, albo wskazywać na obiekt jakiegoś typu.

```
class Reference[T] {
  private var contents: T = _

  def set(value: T) { contents = value }
  def get: T = contents
}
```

Klasa `Reference` jest parametryzowana typem, nazwanym `T`, który jest typem jej elementów. Ten typ jest użyty w ciele klasy jako typ zmiennej `contents`, argument metody `set` oraz typ rezultatu metody `get`.

Powyższa próbka kodu wprowadza zmienne w Scali, co nie powinno wymagać dalszych wyjaśnień. Jest jednak interesujące zobaczyć, że początkowa wartość nadana tej zmiennej to `_`, co reprezentuje wartość domyślną. Ta wartość domyślna to 0 dla typów numerycznych, **`false`** dla typu `Boolean`, `()` dla typu `Unit` i **`null`** dla wszystkich typów obiektowych.

Aby użyć tej klasy `Reference`, trzeba określić którego typu użyć jako parametru typu `T`, to jest typu elementu zawartego w komórce. Na przykład, aby utworzyć i użyć komórkę zawierającą liczbę całkowitą, można by napisać co następuje:

```
object IntegerReference {
  def main(args: Array[String]) {
    val cell = new Reference[Int]
    cell.set(13)
    println("Reference contains the half of " + (cell.get * 2))
  }
}
```

Jak można zobaczyć w tym przykładzie, nie jest niezbędne rzutowanie wartości zwracanej przez metodę `get` przed użyciem jej jako liczby całkowitej. Nie jest również możliwe przechowanie niczego innego poza liczbą całkowitą w tej szczególnej komórce, ponieważ była ona zadeklarowana jako przechowująca liczbę całkowitą.

9 Wniosek

Ten dokument dał szybki przegląd języka Scala i zaprezentował trochę podstawowych przykładów. Zainteresowany czytelnik może kontynuować poprzez czytanie dokumentu towarzyszącego *Scala By Example*, który zawiera o wiele bardziej za-

awansowane przykłady oraz w razie potrzeby sięgać do *Scala Language Specification*.