krótkie wprowadzenie

JAG
ETI PG

by Arkadiusz Hiler

Kto używa?

- GridGain - DSL for cloud computing,
- Twitter – backend + nowe elementy infrastruktury,
- Forsquare – popularny serwis – geolokacja,
- The Guardian w Open Platform,
- Apache Camel – DSL for routing rules,
- Sony Pictures.

Co mamy?

- Lift,
- Play!,
- Specs, ScalaCheck, ScalaTest, SUnit,
- Eclipse Plugin (3.5, do 3.6 nightly builds),
- tonę innych rzeczy o których nie wiem.

Charakterystka

- Łączy aspekty programowania imperatywnego i funkcjonalnego.
- Bezbolesny model współbieżności (concurrency) oparty o aktorów.
- Projektowana pod Java Platform. Łatwa integracja z istniejącymi rozwiązaniami (statyczne typowanie, podział na klasy, …).
- Łączenie i rozszerzanie kodu w obie stron.
- Interpreter (skrypt + interaktywny shell).

Porównanie z Java

- w pełni obiektowa,
- przeciążanie operatorów,
- „wielodziedziczenie” (traits, model mixinowy),
- funkcje anonimowe,
- minimalistyczny rdzeń języka,
- rozszerzalna - smalltalkowe podejście,
- gęstszy kod.

Specyfika Języka

Brak typów primitywnych widzianych dla programisty. Pod maską kryje się typ primitywny opakowywany w razie potrzeby – nie ma tu utraty wydajności. Tylko referencje.

Tuple do zwracania wielu wartości (jak w Pythonie):

`(1, 2, 3)`
Odwołujemy się do elementu: `tuple._2`

Rolę **void** pełni **Unit**.
Zamiast **this** używamy **self**.

Metoda (funkcja) jeśli coś zwraca to zwraca wartość ostatniego wyrażenia w ciele (nie trzeba używać return).

Domyślnie importowana jest zawartość pakietu java.lang

Wynikiem przypisania jest Unit, więc a = b = c nie działa tak jak w Javie.

== jest final i porównuje wartości (wywołuje equals()), eq() od referencji

Referencje

Referencje:

```
val frame: JFrame = new JFrame("Hello world!")
```

Dozwolony zapis w stylu:

```
val button = new JButton("Klick Me!")
```

Tam gdzie to możliwe kompilator nas wyręcza i odgaduje typ z kontekstu.

```
val – constant ref (final)
var – można zmieniać to na co wskazuje
```

Multiprzypisanie:

```
val (b1, b2, b3) = tupleReturningMeth
```

Inicjowanie domyślnymi wartościami:

```
val st: String = _
```

Czytelność:

```
val fancyref: GenericClassWithLongName[Type1, Type2[Type3]]
```

Czemu tak? To co najpopularniejsze, nie zawsze najlepsze.

Więcej kodu, mniej znaków

Opcjonalne średniki (\n jest wystarczającym separatorem):

```
frame.getContentPane.add(button)
frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE)
frame.pack
```

Notacja infiksowa:

Wywołanie metody przyjmującej jeden parametr zamiast:

```
a1.meth(a2)
```

Możemy zapisać:

```
a1 meth a2
```

Stąd już tylko krok do naturalnego przeciążania operatorów.

Przy większej ilości operatorów trzeba używać (), kropka jest opcjonalna.

Klasy, obiekty i metody

Klasa:

```
class SampleClass { kod... }
new SampleClass //nawiasy potrzebne, tylko jeśli wywołujemy konstruktor
                  przyjmując parametry
```

Brak elementów statycznych (nieobiektowe), zamiast tego singleton:

```
object SampleClass { kod... }
```

Pełna obiektowość. Wzorzec singletona na poziomie języka. Instancja tworzona podczas pierwszego użycia i pilnowana przez kompilator. Odwołujemy się po nazwie obiektu. Singleton i klasa mogą współistnieć „companion object”, dostęp do prywatnych pól, itp.

Wszystko jest domyślnie **public.**
Znaczenie **protected** jak w C++.
SampleClass(params) → SampleClass.apply(params)

Metody:

```
def method { kod... } //metoda bezargumentowa
//parametry, będzie coś zwracać (=), kompilator zgaduje:
def meth(arg1: Int) = { kod... }
//co zwraca podano w prost, tu para:
override def meth(arg1: Int): (Int, String) = { kod... }
```

Klasy, obiekty i metody

Konstruktor domyślny w definicji klasy:

```
class SampleClass(var zm: Int, val st: Int) {
    //kod wykonywany przy konstrukcji klasy bezpośrednio tutaj...
}
```

To co podajemy dostajemy jako pola:
val – final, dostajemy getter w konwncji scinstance.st
var – getter jak wyżej, setter scinstance.zm =

Można w stylu JavaBeans:
`@scala.reflect.BeanProperty var x: Int`

settery i getery, pola domyślnie są private

Własny, dodatkowy konstruktor:

```
class SampleClass(var zm: String, val st: Integer) {
    private var str: String = _ //dostaniemy prywatny geter i seter
    //kod domyślnego konstruktora bezpośrednio tu
    //poniżej kod konstruktora „dodatkowego”
    def this(zm: Int, st: Int, strin: String ) {
        this (zm, st)
        str = strin
    }
}
```

Dziedziczenie

Jak dziedziczyć:

```
class Sam(var zm: Int, val st: Int) { }
class Dsc(override var zm: Int, override val st: Int, val az: Int) extends
      Sam(zm, st) { }
```

Tylko główny konstruktor może przekazywać parametry do konstruktora klasy bazowej.
Po **Any** dziedziczą wszystkie klasy (analogicznie do **Object**).

Traits:

```
trait Worker {
    val name: String; //nie inicjujemy, abstract
    def work = print(name + "pracuje") //można nadpisać
}

class Constuctor extends Human with Worker
```

Dostajemy typ referencji.

Mieszanie na poziomie instancji:
```
new Worm("joe") with Worker
```

Ewaluacja od prawej do lewej with (override metod + łańcuch wywołań super).

Funkcje anonimowe

Funkcje anonimowe:

```
(parametry) => { ciało }
```

Przypiszmy:

```
val f = (i: Int) => { i * 2 }
f(3) //i wykonajmy...
```

Jak przekazywać do funkcji?

```
def functional(f1: (Int, Float) => Int)) { //ładna definicja
    f1(0, 0.1)
}
```

Partialy applied functions:

```
val paf = func(first_param, _ : Type)

paf(t1) //func(first_param, t1)
paf(t2) //func(first_param, t2)
```

Styl funkcyjny

Programowanie funkcyjne:

```
val list = List(1,2,3,4,5)
```
na dobry początek, teraz stwórzmy listę zawierającą dwukrotność tych elementów

```
val l1 = list.map( (i) => i * 2 )
```
używając pełnej składni przekazujemy funkcję anonimową

```
val l2 = list.map(_ * 2)
```
_ jest parametrem domyślnym, każde jego użycie reprezentuje kolejny parametr

```
val l3 = list.map(2*)
```
jako parametr podejmy zwykłą, istniejącą funkcję (metodę) * z obiektu 2

Znajdź maksimum? Nic prostrzego korzystając z automatycznego podawania parametrów:
```
val max = list.foldLeft(0) { Math.max }
```

```
Skrótowe formy zapisu (to samo co wyżej):
val max = (0 /: list) { Math.max }
```

Pattern Matching

Miodniejszy switch statment. Pierwszy pasujący od góry.

```
for (a <- args) a match {
  case "-v" | "-verbose" =>
    verbose = true
  case x =>
    println("Unknown option: '" + x + "'")
}
```

```
case _ - wildcard
```

```
case (x, y) => printf("%d %d", x, y)
```

```
case List("first", "second", oth @ _*) //_* - pozostałe, tu nazwane oth
```

```
case x: Int if (x > 9000) => print("over nine thousand!") //guard
```

matchowanie XMLa

case classes

extractors

regexps

To tylko początek.

Pytania?

Źródła

- http://www.scala-lang.org/
- „Programming Scala” Venkat Subramaniam (Pragmatric Bookshelf)
- http://programming-scala.labs.oreilly.com/
- http://www.scala-lang.org/docu/files/ScalaTutorial.pdf