Podstawy programowania w języku Scala

Wprowadzenie

Roman Dębski

Instytut Informatyki, AGH

4 marca 2022

Plan wykładu

Plan wykładu

Organizacja zajęć

Punkty do oceny końcowej

- aktywne uczestnictwo w zajęciach (lab), $o \in [0, 42]$
- kartkówki/quizy, $k \in [0, 36]$
- projekt, $p \in [0, 22]$

Suma punktów $sp = o + k + p$, $sp \in [0, 100]$

sp	ocena
$[50, 60)$	3.0
$[60, 70)$	3.5
$[70, 80)$	4.0
$[80, 90)$	4.5
$[90, 100]$	5.0

Kartkówki/quizy: zaliczenie materiału z poprzedniego laboratorium[1].

Projekty: w parach (praca zespołowa), tematy uzgadniane z prowadzącym zajęcia lab, prezentacje podczas ostatnich zajęć.

[1] obecność na wykładzie NIE jest obowiązkowa

Plan wykładu

"Narodziny" Scali

Martin Odersky
http://lampwww.epfl.ch/~odersky/

——— Original Message ———
Subject: [TYPES] ANNOUNCEMENT: The Scala Programming Language
Date: 20 Jan 2004 17:13:59 +0100
From: Martin Odersky
To: [The Types Forum]

We'd like to announce availability of the first implementation of the Scala programming language. Scala smoothly integrates object-oriented and functional programming. It is designed to express common programming patterns in a concise, elegant, and type-safe way. Scala introduces several innovative language constructs. For instance:

- Abstract types and mixin composition unify ideas from object and module systems.
- Pattern matching over class hierarchies unifies functional and object-oriented data access. It greatly simplifies the processing of XML trees.
- A flexible syntax and type system enables the construction of advanced libraries and new domain specific languages. At the same time, Scala is compatible with java. java libraries and frameworks can be used without glue code or additional declarations.

The current implementation of Scala runs on java VM. It requires JDK 1.4 and can run on Windows, MacOS, Linux, Solaris, and most other operating systems. .net version of Scala is currently under development.
For further information and downloads, please visit: scala.epfl.ch

Martin Odersky and the Scala team

Wybrane opinie na temat Scali

"If I were to pick a language today other than Java, it would be Scala."

— James Gosling, creator of Java

"I can honestly say if someone had shown me the Programming in Scala book by Martin Odersky, Lex Spoon & Bill Venners back in 2003 I'd probably have never created Groovy."

— James Strachan, creator of Groovy

"No other language on the JVM seems as capable of being a 'replacement for Java' as Scala, and the momentum behind Scala is now unquestionable."

— Charles Nutter, co-creator of JRuby

I jeszcze dwa cytaty...

"Research results from the psychology of programming indicate that expertise in programming is far more strongly related to the number of different programming styles understood by an individual than it is the number of years of experience in programming."

— Timothy Budd

"[...] great care has been spent in the Scala design to make functional constructs, imperative constructs, and objects all play well together. I think postfunctional is a good term for that blend."

— Martin Odersky

Krótka charakterystyka języka Scala

Scala to język:

- obiektowo-funkcyjny[2],
- kompilowany do kodu bajtowego JVM,
- statycznie (silnie) typowany[3] z inferencją (wnioskowaniem) typu,
- rozszerzalny (ze względu na typy danych i struktury kontrolne),
- skalowalny (krótkie skrypty → duże systemy),
- ...

Łączy wybrane cechy wielu języków, np. Java/C#, C/C++, Smalltalk, Ruby, Algol, Simula, Pascal, Modula-2, Eiffel, ML (OCaml, F#), Haskell, Erlang,...

[2] Scala jest językiem w pełni obiektowym (każda wartość jest obiektem, każda operacja jest wywołaniem metody, por. Java)
[3] typy są ustalane/sprawdzane na etapie kompilacji, zmienne i wartości mają typ; błędy kompilacji zamiast błędów czasu wykonania; automatyczna refaktoryzacja w IDE

Rozszerzalność Scali - przykład

```
def repeatWhile(condition: => Boolean)
               (loopBody: => Unit): Unit = {
  if (condition) {
    loopBody
    repeatWhile(condition)(loopBody)
  }
}
```

```
var i = 0
repeatWhile (i <= 3) {
  println(i)
  i += 1
}
```


Zwięzłość Scali - przykład

Java

```
class C {
  private int idx;
  private String name;
  public C(int index, String name) {
    this.idx = index;
    this.name = name;
  }
}
```

Scala

```
class C(idx: Int, name: String)
```

"But you must understand, young hobbit, it takes a long time to say anything in old Entish" —Treebeard

Możliwości Scali - przykład z dynamicznym wywołaniem metod

```
import scala.language.dynamics

class DynClass(dynMethodsMap: Map[String, Function0[Unit]]) extends Dynamic {
  def applyDynamic(methodName: String)() = dynMethodsMap(methodName)()
}

val dynClassInst = new DynClass(Map("dynM1" -> (() => println("dynM1")),
                                    "dynM2" -> (() => println("dynM2"))))

dynClassInst.applyDynamic("dynM1") // -> dynM1

dynClassInst.dynM1() // -> dynM1
dynClassInst.dynM2() // -> dynM2
```


Plan wykładu

Read-Evaluate-Print-Loop (REPL)

```
$ scala
Welcome to Scala version 2.11.7 (Java HotSpot(TM) 64-Bit Server VM, Java...)
Type in expressions to have them evaluated.
Type :help for more information.

scala>
```

```
scala> println("Imagination is the only weapon in the war against reality.")
Imagination is the only weapon in the war against reality.
```

```
scala> 2+2
res0: Int = 4
```

```
scala> 2 * (
     | 3 +
     | 4)
res1: Int = 14
```

REPL vs. *Scala Worksheets* (IntelliJ IDEA lub ScalaIDE)

REPL - :paste mode

```
scala> :paste
// Entering paste mode (ctrl-D to finish)
println("There is no curse in Elvish, Entish,")
println("or the tongues of Men for this treachery!")
// Exiting paste mode, now interpreting.

There is no curse in Elvish, Entish,
or the tongues of Men for this treachery!

scala>
```


REPL - pomoc kontekstowa ([TAB], 2 x [TAB])

```
scala> "abc".
+                    concat              intern              replace
asInstanceOf         contains            isEmpty             replaceAll
charAt               contentEquals       isInstanceOf        replaceFirst
codePointAt          endsWith            lastIndexOf         split
codePointBefore      equalsIgnoreCase    length              startsWith
codePointCount       getBytes            matches             subSequence
compareTo            getChars            offsetByCodePoints  substring
compareToIgnoreCase  indexOf             regionMatches       toCharArray

scala> "abc".
```

REPL - opcja *Xprint:typer*

```
$ scala -Xprint:typer
Welcome to Scala version ...
scala> [[syntax trees at end of          typer]] // <init>
package <empty> {
  class $repl_$init extends scala.AnyRef { /*...*/ }}
[[syntax trees at end of                 typer]] // <console>
package $line1 {
  object $eval extends scala.AnyRef { /*...*/ }}
[[syntax trees at end of                 typer]] // <console>
package $line2 {
  object $read extends scala.AnyRef { /*...*/ }}
[[syntax trees at end of                 typer]] // <console>
package $line2 {
  object $eval extends scala.AnyRef { /*...*/ }}

scala>
```


REPL - opcja *Xprint:typer*, przykład

```
scala> def abs(x: Int) = if (x >= 0) x else -x
[[syntax trees at end of                    typer]] // <console>
package $line3 {
object $read extends scala.AnyRef { /* ... */
 object $iw extends scala.AnyRef { /* ... */
   object $iw extends scala.AnyRef { /* ... */
     def abs(x: Int): Int = if (x.>=(0)) x else x.unary_- }}}}
[[syntax trees at end of                    typer]] // <console>
package $line3 {
object $eval extends scala.AnyRef { /* ... */
 /* ... */ lazy def $print: String = {
   $eval.this.$print = { /* ... */
     "abs: (x: Int)Int\n"
   };
   $eval.this.$print }}}
abs: (x: Int)Int
scala>
```


REPL - opcja *Xprint:typer*, przykład, c.d.

```
scala> val x1 = abs(-15)
[[syntax trees at end of                    typer]] // <console>
package $line4 {
object $read extends scala.AnyRef { /* ... */
 object $iw extends scala.AnyRef { /* ... */
   object $iw extends scala.AnyRef { /* ... */
     private[this] val x1: Int = $line3.$read.$iw.$iw.abs(-15);
     /* ... */ def x1: Int = $iw.this.x1 }}}}
[[syntax trees at end of                    typer]] // <console>
package $line4 {
object $eval extends scala.AnyRef { /* ... */
 /* ... */ lazy def $print: String = {
   $eval.this.$print = { /* ... */
     /* ... */
     "x1: Int = ".+(/* ... */.replStringOf($line4./* ... */.x1, /* ... */)) };
   $eval.this.$print }}}
x1: Int = 15
```


Plan wykładu

Skrypty w Scali - dwa warianty uruchamiania

```
$ scala file_name.scala
```

```
$ ls
script.scala
$ cat script.scala
println("Hasta la vista, baby")
$ scala script.scala
Hasta la vista, baby
```

```
scala> :load file_name.scala
```

```
$ scala
Welcome to Scala version...
scala> :load script.scala
Loading script.scala...
Hasta la vista, baby

scala>
```

por. *Opowieści Skrypty*

Plan wykładu

Podstawowe typy danych w Scali

- Byte (8), Short (16), Int (32), Long (64),
- Float (32), Double (64),
- Char (16),
- Boolean,
- Unit,
- (String).

Plan wykładu

Zmienne: **val** vs. **var**

`val` - wartość niezmienna* (single-assignment variable))

```
scala> val value1: Int = 3
value1: Int = 3
scala> value1 = 4
<console>:8:
    error: reassignment to val
    value1 = 4
```

`var` - "zwykła zmienna"

```
scala> var var1: Int = 3
var1: Int = 3
scala> var1 = 4
var1: Int = 4
```

Inferencja typu

```
scala> val x1 = 3
x1: Int = 3
```

* `val` ~ `const` w C++, `final` w Javie

"Always start with val when declaring variables in Scala and change to var when it's absolutely necessary"

Incjalizacja zmiennych w Scali

```
scala> val x1: Int
<console>:7: error: only classes can have declared but undefined members
val x1: Int
 ^
scala> val x2: Double
<console>:7: error: only classes can have declared but undefined members
val x2: Double
 ^
scala> val x3: Int = _
<console>:7: error: unbound placeholder parameter
val x3: Int = _
              ^
scala> var x3: Int = _
x3: Int = 0
```


Przykładowe :) nazwy zmiennych w Scali

```
scala> val ^|^ = 3
^|^: Int = 3
scala> val *** = 5
***: Int = 5
scala> *** + ^|^
res1: Int = 8
```

```
scala> val ==> = "r-arrow"
==>: String = r-arrow
scala> val s = "abc" + ==>
s: String = abc-arrow
```

```
scala> val `żółta gżegżółka` = 5
żółta gżegżółka: Int = 5
```


Plan wykładu

Instrukcje vs. wyrażenia

Języki imperatywne
wykonywanie instrukcji

Języki funkcyjne
obliczanie wartości wyrażeń

źródła: en.wikipedia.org, www.felienne.com

Wyrażenia w Scali

```
scala> var x: Int = _
x: Int = 0
scala> val xr1 = (x = 2)
xr1: Unit = ()
scala> val xr2 = (x += 2)
xr2: Unit = ()
scala> x
res3: Int = 4
scala> val x3 = x + 2
x3: Int = 6
```

```
scala> x++
<console>:9: error:
 value ++ is not a member of Int
  x++
  ^
scala> val x4 = x += 1
x4: Unit = ()
scala> val x5 = (x += 1)
x5: Unit = ()
scala> x5
// <-- ()
scala>
```


Blok

```
scala> { val x = 10; 2 * x + 1 }
res1: Int = 21
```

```
scala> val b1 = {
     |     println("The answer to the Ultimate Question of Everything "); 42
     |   }
The answer to the Ultimate Question of Everything
b1: Int = 42
```

```
scala> val b2 = {
     |   val x = 42;
     |   println("The answer to the Ultimate Question of Everything is " + x)
     | }
The answer to the Ultimate Question of Everything is 42
b2: Unit = ()
```


Wyrażenie warunkowe ("if else")

```
scala> val i = -1
i: Int = -1
scala> val absV = if (i >= 0) i else -i
absV: Int = 1
```

```
scala> i
res1: Int = -1
scala> val halfIf1 = if (i > 0) i
halfIf1: AnyVal = ()
scala> val halfIf2 = if (i > 0) i else ()
halfIf2: AnyVal = ()
```

Por. ternary operator: (cond) ? valTrue : valFalse

Pętla "while"

```
var i = 0
while (i < 5) {
  println(i)
  i += 1
}
```

```
scala> var i = 0
i: Int = 0
scala> val wlV = while(i < 3) {println(i); i += 1}
0
1
2
wlV: Unit = ()
```

Uwaga: `while` i `do-while` to pętle, a nie wyrażenia. W Scali nie ma instrukcji `break` i `continue`.

Pętla "do-while"

```
var i = 0
do {
 println(i)
 i += 1
} while (i < 3)
```

```
scala> i = 0
scala> val dwlV = do {
     |   println(i)
     |   i += 1
     | } while (i < 3)
0
1
2
dwlV: Unit = ()
```

```
scala> i = 0
scala> val dwlV2 = do {
     |   println(i)
     |   i += 1
     |   42 // <-- !
     | } while (i < 3)
0
1
2
dwlV2: Unit = () // <-- !
```


Pętla "for"

`for (seq) ...`

seq - przedzielona średnikami sekwencja: generatorów, definicji i filtrów
`generator: patern <- expression`

```
for (i <- 1 to 10; x = 2 * i + 1; if (x % 3 == 0)) println(i, x)
```

=

```
var i = 0
while (i <= 10) {
 val x = 2 * i + 1
 if (x % 3 == 0) println(i,x)
 i += 1
}
```


Pętla "for", przykłady

```
scala> for (i <- 1 to 2) println(i)
1
2
```

```
scala> for (i <- 1 until 2) println(i)
1
```

```
scala> for ((k,v) <- Map(1 -> "I", 2 -> "II")) println(k,v)
(1,I)
(2,II)
```

```
scala> for {i <- 1 until 3
     | j <- i until 4 * i; if ((i * j) % 3 == 0)
     | k <- 1 until 5; if (j < k)} println(i, j, k)
(1,3,4)
(2,3,4)
```

Uwaga: `(1 to 2 = 1.to(2)) != (1 until 2 = 1.until(2))`; :type 1 to 2 -> ...Range.Inclusive; :type 1 until 2 -> ...Range

Wyrażenie "for-yield" [for comprehension]

```
for (seq) yield expression
```

```
scala> for (i <- 1 to 3) yield(i)
res4: scala.collection.immutable.IndexedSeq[Int] = Vector(1, 2, 3)
```

```
scala> val xsa = for (i <- (1 to 3).toArray) yield(i) // def toArray: Array[A]
xsa: Array[Int] = Array(1, 2, 3)
scala> val xsl = for (i <- (1 to 3).toList) yield(i) // def toList: scala.List[A]
xsl: List[Int] = List(1, 2, 3)
```

por. list comprehensions

?

```
val r = for (i <- 1 until 5) { 2 * i + 1 }
```

a)
```
r = Vector(3, 5, 7, 9)
```

b)
```
r = // sth else
```

c)
```
r = Vector(3, 5, 7, 9, 11)
```

```
val r = for (i <- (0 until 5).toList; x = i + 1; if (x % 2 == 0)) yield (2 * i)
```

a)
```
r = List(0, 4, 8)
```

b)
```
r = Vector(0, 4, 8)
```

c)
```
r = List(2, 6)
```


Plan wykładu

Metoda != funkcja

Metoda

```
scala> def abs(x: Double): Double = if (x >= 0) x else -x
abs: (x: Double)Double
```

Funkcja

```
scala> val absAsFunction: Double => Double = abs
absAsFunction: Double => Double = <function1>
scala> val absAsFunction = abs _
absAsFunction: Double => Double = <function1>
scala> val x = absAsFunction.apply(-3)
x: Double = 3.0
```

Uwagi: a) funkcje będą omawiane podczas wykładu 4. b) `absAsFunction.toString()` -> absFunToString: String = <function1>

Typ wyniku funkcji

Inferencja typu wyniku

```
scala> def abs(x: Double): Double = if (x >= 0) x else -x
abs: (x: Double)Double
scala> def abs(x: Double) = if (x >= 0) x else -x
abs: (x: Double)Double
```

Inferencja typu dla funkcji rekurencyjnej

```
scala> def factorial (n: Int) = if (n <= 0) 1 else n * factorial(n-1)
<console>:8: error: recursive method factorial needs result type
    def factorial (n: Int) = if (n <= 0) 1 else n * factorial(n-1)
scala> def factorial (n: Int): Int = if (n <= 0) 1 else n * factorial(n-1)
factorial: (n: Int)Int
```


"Procedury" - funkcje/metody zwracające Unit

```
scala> def printlnArg(a: Int): Unit = {println(a)}
printlnArg: (a: Int)Unit
scala> def printlnArg(a: Int): Unit = println(a)
printlnArg: (a: Int)Unit
scala> def printlnArg(a: Int) = println(a)
printlnArg: (a: Int)Unit
scala> def printlnArg(a: Int) {println(a)}
printlnArg: (a: Int)Unit
scala> def printlnArg(a: Int) println(a)
<console>:1: error: '=' expected but identifier found.
     def printlnArg(a: Int) println(a)
```


No return

Problemy z użyciem `return` ...

```
scala> def max(a: Int, b: Int) = { if(a > b) return a else return b }
<console>:9: error: method max has return statement; needs result type
      def max(a: Int, b: Int) = { if(a > b) return a else return b }
                                                ^
<console>:9: error: method max has return statement; needs result type
      def max(a: Int, b: Int) = { if(a > b) return a else return b }
scala> def max(a: Int, b: Int): Int = { if(a > b) return a else return b }
max: (a: Int, b: Int)Int
```

... znikają, gdy programujemy funkcyjnie

```
scala> def max(a: Int, b: Int) = if (a > b) a else b
max: (a: Int, b: Int)Int
```

Uwaga: instrukcja `return` - programowanie imperatywne

Funkcje o zmiennej liczbie parametrów (variadic functions)

```
def printAll(args: Int*) {for (arg <- args) println(arg)}
```

```
scala> printAll(3,4,5)
3
4
5
```

Ograniczenie parametru-*

```
scala> def f2(ints: Int*, theLast: Int) = {
     |     for (arg <- ints) println(arg); println(theLast)
     |   }
<console>:7: error: *-parameter must come last
      def f2(ints: Int*, theLast: Int) = {
```


Funkcje o zmiennej liczbie parametrów – "splat" i typ Any

```
scala> printAll(1 to 5)
<console>:9: error: type mismatch;
found   : ...Range.Inclusive
required: Int ... printAll(1 to 5)
```

:_* "splat"

```
scala> printAll(3 to 4:_*)
3
4
```

```
scala> printAll(1, 2, 3.5)
<console>:9: error: type mismatch;
found   : Double(3.5)
required: Int
            printAll(1, 2, 3.5)
```

Typ `Any`

```
scala> def printAll(args: Any*)
     |   {for (arg <- args) println(arg)}
printAll: (args: Any*)Unit
scala> printAll(1,2.5,"Aloha!",(2+2==5))
1
2.5
Aloha!
false
```


Wartości domyślne parametrów. Argumenty nazwane (vs. pozycyjne)

```
def printName(name: String = "John", surname: String = "Doe")
{ println(name + " " + surname) }
```

```
scala> printName()
John Doe
scala> printName(surname = "Kowalsky")
John Kowalsky
scala> printName(surname = "Nawasky", name = "Frank")
Frank Nawasky
scala> printName("Nawasky", "Frank")
Nawasky Frank
```


Metody bezparametrowe

```
def f() = 2 * 2
```

```
scala> f()
res53: Int = 4
scala> f
res54: Int = 4
scala> def f1 = 2 * 2
f1: Int
scala> f1
res55: Int = 4
scala> f1()
<console>:9: error: Int does not take parameters f1()
```