Další jazyky kompilovatelné do Java byte-code

Přehled

- Scala
  - objektový a funkcionální jazyk
- Closure
  - funkcionální jazyk
    - dialekt Lispu
- Groovy
  - „skritpovací“ jazyk
- Kotlin
  - „nová“ Java
- Jython
  - Java implementace jazyka Python
- JRuby
  - Java implementace jazyka Ruby

Scala

- SCAlable LAnguage
- mix objektového a funkcionálního jazyka
- staticky typovaný
- překládaný do byte-code
  - existovala i verze pro .NET
- používá se
  - Twitter napsán ve Scale
    - od r. 2009, původně byl v Ruby
  - LinkedIn
  - ...

Scala

- používání
  - překladač scalac
  - spouštění scala -classpath . Třída

- proměnné
  - var identifikator[:typ] = hodnota
- konstanty
  - val identifikator[:typ] = hodnota

- datové typy
  - třídy Int, Double, Boolean,...
  - při překladu se použijí primitivní Java typy

Scala

- metody
  - def max(x:Int, y:Int):Int = if(x>y)x else y
    - příkazy není nutno oddělovat středníkem
    - pokud má tělo jeden příkaz není nutno používat složené závorky
- pole
  - val pole:Array[Typ] = new Array[Typ](velikost)
  - použití
    pole(0) = 5
- seznam
  - val seznam = List(1,2,3)
  - neměnný, všechny prvky musejí být stejného typu
  - Nil – prázdný řetězec
  - metody pro práci se seznamy
    - head, tail, filter, sort, count, reverse,...

Scala

- n-tice
  - val hrac = (“Novak“, 68)
  - mohou obsahovat prvky různých typů
  - neměnné
  - přístup k položkám
    - hrac._1 hrac._2
- funkce jsou „first class“ elementy
  - funkce je objekt, lze přiřadit do proměnné
  - existují anonymní funkce
  - př.
    - funkce(i) = {x => Math.pow(x,i)}
    - seznam.filter(x => x > 4)
    - seznam.exists(x => x == 4)
    - seznam.sort((x,y) => x < y)

Scala

- třídy
  - **object** ExampleClass {
    def foo() { … }
    }
  - object ~ singleton
    - od třídy existuje jediná instance
    - vše uvnitř se chová jako static
      - klíčové slovo static neexistuje
  - class Complex(re:Double, im:Double) {
    …
    }
  - pouze jednoduchá dědičnost
    - ale existují traits

Scala

- trait
  - částečně implementovaná třída
  - trait se „přimíchá“ (mix-in) do třídy
    - class A extends T
  - lze přidat více traits
    - class B extends A with T1 with T2 with …

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

Scala

- generické datové typy

```
class Moje[T] {
…
}

val m1 = new Moje[Int]
val m2 = new Moje[Double]
```

- lze mít i generické metody
  - def foo[T](i:Int) …

Scala

- volná syntax
  - středník nepovinný
  - metody lze používat jako infix oprátory
    - "%d apples".format(num)
    - "%d apples" format num
  - složené závorky lze použít místo obyčejných při volání metod
    - breakable { ... if (...) break() ... }
  - „placeholder“ v anonymních funkcích
    - list map { x => sqrt(x) }
    - list map { sqrt(_) }
    - list map sqrt
  - …
- snadná tvorba „nových“ jazyků

Groovy

- objektový jazyk, dynamický
- dynamicky kompilovaný do byte-code
- syntaxe podobná Javě a Ruby
  - většina Java kódu je syntakticky správný Groovy kód
- vznik 2003
  - v roce v 2009 původní autor Groovy napsal „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.“

Groovy

- příklady

```
class Song{
  length
  name
}

class Book{
  name
  author
}

def doSomething(thing){
  println "going to do something with a thing
                                named = " + thing.name
}

mySong = new Song(length:90, name:"Burning Down the
House")

anotherSomething = doSomething
```

Groovy

- používání closures

```
class Dog{
  action

  train(){
    action.call()
  }
}

sit = { println "Sit, Sit! Sit! Good dog"}
down = { println "Down! DOWN!" }

myDog = new Dog(action:sit)
myDog.train()  // prints Sit, Sit! Sit! Good dog

mollie = new Dog(action:down)
mollie.train() // prints Down! DOWN!
```

Groovy

- kolekce

```
aCollect = [5, 9, 2, 2, 4, 5, 6]
println aCollect.join(' - ')
                    // prints 5 - 9 - 2 - 2 - 4 - 5 - 6
println aCollect.count(2)     // prints 2
println aCollect.sort()
                    // prints [2, 2, 4, 5, 5, 6, 9]
```

- „maps“

```
myMap = ["name" : "Groovy", "date" : new Date()]
println myMap["date"]
println myMap.date
```

- „ranges“

```
myRange = 29...32
myInclusiveRange = 2..5
println myRange.size() // prints 3
println myRange[0]   // prints 29
println myRange.contains(32) //prints false
println myInclusiveRange.contains(5) // prints true
```

Groovy

- GroovyBeans
  - obdoba JavaBeans

- transformace AST (Abstract Syntax Tree)
  - pomocí anotací

```
@ToString
class Person {
    String firstName
    String lastName
}
```

    - přidá metodu toString()

Clojure

- funkcionální jazyk
- dialekt Lispu

- příklady

```
(println "Hello, world!")

(javax.swing.JOptionPane/showMessageDialog nil
"Hello World" )
```

JAVA

Kotlin

Přehled

- od 2011
- 2016 – verze 1.0
- vyvíjeno JetBrains
- 2017 – Android podpora

- cíl

  an industrial-strength object-oriented language, and a "better language" than Java

- interoperabilita s Javou

- podpora v IntelliJ, Eclipse, Maven, Gradle,...

Hello world

- podpora pro top-level funkce

- main

```
fun main(args: Array<String>) {
  val scope = "world"
  println("Hello, $scope!")
}
```

- středník je volitelný

- kotlinc HelloWorld.kt
- java -cp .:kotlin-rutime.jar HelloWorldKt

Proměnné

- var result: String
  - proměnná
- val message: String = "Hello world"
  - konstanta

- inference typů

  val message = "Hello world"

  fun plusOne(x: Int) = x + 1

- nullable proměnné

  var str: String? = null

Základní typy

- Long 64 bit
- Int 32 bit
- Short 16 bit
- Byte 8 bit
- Double 64 bit
- Float 32 bit
- Boolean

- třídy
  - pokud lze, při překladu se mapují na Java primitivní typy

Char a String

- Char
  - jednoduché uvozovky
  - znaky
  - není považován za číslo (jako v Javě)

- String
  - uvozovky
  - „raw“ řetězce – nejsou třeba „escape“ znaky
    """raw string here"""

- řetězcové šablony (templates)

```
val name = "John"
println("Hello $name")
println("The name has ${name.length} chars")
```

Balíčky

- `package cz.cuni.mff.kotlin`

- `import cz.cuni.mff.kotlin.AClass`
- `import cz.cuni.mff.kotlin.*`

- není `import static` jako v Javě
  - použije se jen `import`

- `import cz.cuni.mff.kotlin.AClass as AnotherClass`

- není nutné organizovat balíčky do adresářů
  - ale je to doporučené

Pole

- třída `Array`

- vytvoření pole
  `arrayOf()`

- `val array = arrayOf(1, 2, 3)`
- `val asc = Array(5, { i -> (i * i) })`

- přístup k elementům [ ]
  - ve skutečnosti metody `get()` a `set()`

- pole pro primitivní typy
  - `ByteArray, ShortArray, IntArray`

Ranges

- interval hodnot

```
• val aToZ = "a".."z"
• val oneToNine = 1..9

• val isTrue = "c" in aToZ
• val isFalse = 11 in oneToNine

• val countingDown = 100.downTo(0)
• val rangeTo = 10.rangeTo(20)

• val oneToFifty = 1..50
• val oddNumbers = oneToFifty.step(2)
```

Výjimky

- pouze „unchecked“ výjimky
  - tj. nemusejí se deklarovat ani odchytávat
- ošetření jako v Javě
  - `try/catch/finally`

Řízení programu

- **`if-else`**

```
var max: Int
if (a > b) {
    max = a
} else {
    max = b
}
```

- **`if`** lze použít jako výraz
  - ternární výraz

```
val max = if (a > b) a else b
```

Řízení programu

```
val max = if (a > b) {
    print("Choose a")
    a
} else {
    print("Choose b")
    b
}
```

- při použití jako výraz je **`else`** větev povinná

Řízení programu

- **`when`** – náhrada za **`switch`**

- ```
  when (x) {
      1 -> print("x == 1")
      2 -> print("x == 2")
      else -> {
          print("x is neither 1 nor 2")
      }
  }
  ```

- ```
  when (x) {
      0, 1 -> print("x == 0 or x == 1")
      else -> print("otherwise")
  }
  ```

Řízení programu

- when (x) {
    in 1..10 -> print("x is in the range")
    in validNumbers -> print("x is valid")
    !in 10..20 -> print("x is outside the range")
    else -> print("none of the above")
  }

- fun hasPrefix(x: Any) = when(x) {
    is String -> x.startsWith("prefix")
    else -> false
  }

- when {
    x.isOdd() -> print("x is odd")
    x.isEven() -> print("x is even")
    else -> print("x is funny")
  }

Řízení programu

- for (item in collection) {
  print(item)
  }

- for (i in 1..3) {
  println(i)
  }

- for (i in array.indices) {
  println(array[i])
  }

Řízení programu

- while (x > 0) {
  x--
  }

- do {
  val y = retrieveData()
  } while (y != null)

Řízení programu

- break, continue
  - jako obvykle

- s návěštími

```
loop@ for (i in 1..100) {
    for (j in 1..100) {
        if (...) break@loop
    }
}
```

Přetypování

- smartcast

```
fun printStringLength(any: Any) {
    if (any is String) {
        println(any.length)
    }
}

fun isEmptyString(any: Any): Boolean {
    return any is String && any.length == 0
}
```

- explicitní přetypování

```
fun length(any: Any): Int {
    val string = any as String
    return string.length
}
```

Hierarchie typů

- Any
  - nadtyp všeho
    ~ Java Object

- Unit
    ~ Java void
  - je to typ
  - singleton

- Nothing
  - podtyp všeho

Třídy

Primární konstruktor

- class Person constructor(firstName: String) {
  }

- class InitOrderDemo(name: String) {
    val firstProperty = name

    init {
        println("First initializer block that prints ${name}")
    }

    val secondProperty = name.length

    init {
        println("Second initializer block that prints ${name.length}")
    }
  }

Primární konstruktor nemá tělo
Použijí se init bloky

Třídy

- class Person(val firstName: String, val lastName: String, var age: Int) {
  // ...
  }

Přímá deklarace a inicializace vlastností

- class Person {
  constructor(parent: Person) {
  parent.children.add(this)
  }
  }

Sekundární konstruktor
Lze jich mít více

- class Person(val name: String) {
  constructor(name: String, parent: Person) : this(name) {
  parent.children.add(this)
  }
  }

Volání jiného konstruktoru

Třídy a objekty

- instanciace
  - val invoice = Invoice()
  - val customer = Customer("Joe Smith")

Není **new**

- nejsou statické metody
- náhrada ~ companion object
  - ```
    class MyClass {
        companion object Factory {
            fun create(): MyClass = MyClass()
        }
    }
    ```
  - val instance = MyClass.create()
    ```

Dědičnost

Implicitně nelze dědit

- open class Base(p: Int)
- class Derived(p: Int) : Base(p)

Volání konstruktoru předka

- class MyView : View {
  constructor(ctx: Context) : super(ctx)
  constructor(ctx: Context, attrs: AttributeSet) : super(ctx, attrs)
  }

Implicitně nelze předefinovat

- open class Base {
  open fun v() {}
  fun nv() {}
  }
  class Derived() : Base() {
  override fun v() {}
  }

- open class AnotherDerived() :
  Base() {
  final override fun v() {}
  }

Předefinování lze opět zakázat

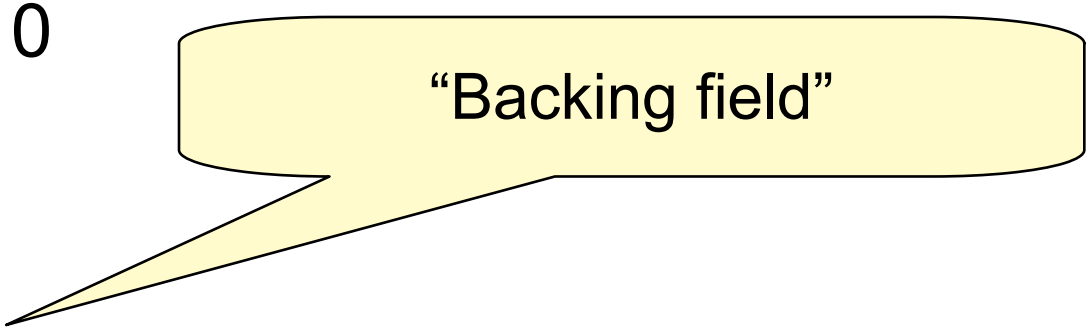
Třídy: properties

- class Address {
  var name: String = ...
  var street: String = ...
  }

- nejsou to atributy jako v Javě

- val isEmpty: Boolean
  get() = this.size == 0

- var counter = 0
  set(value) {
  if (value >= 0) field = value
  }



Třídy: properties

- properties mohou být předefinovány
  - jako metody

- open class Foo {
  open val x: Int get() { ... }
  }

- class Bar1 : Foo() {
  override val x: Int = ...
  }

Třídy: metody

- metody ~ member functions
  - vše jako normální funkce

Funkce

- implicitní hodnoty argumentů

```
fun read(b: Array<Byte>, off: Int = 0, len: Int = b.size) {
  ...
}
```

- infix funkce

```
infix fun Int.shl(x: Int): Int {
    // ...
}

1 shl 2    // is the same as   1.shl(2)
```

Lokální funkce

```
fun dfs(graph: Graph) {
    val visited = HashSet<Vertex>()
    fun dfs(current: Vertex) {
        if (!visited.add(current)) return
        for (v in current.neighbors)
            dfs(v)
    }

    dfs(graph.vertices[0])
}
```

Vnořené/vnitřní třídy

- class Outer {
  private val bar: Int = 1
  class Nested {
  fun foo() = 2
  }
  }

- class Outer {
  private val bar: Int = 1
  inner class Inner {
  fun foo() = bar
  }
  }

Interface

- interface MyInterface {
  fun bar()
  fun foo() {
  // optional body
  }
  }

- class Child : MyInterface {
  override fun bar() {
  // body
  }
  }

Interface

- interface MyInterface {
  val prop: Int // abstract

  val propertyWithImplementation: String
  get() = "foo"

  fun foo() {
  print(prop)
  }
  }

- class Child : MyInterface {
  override val prop: Int = 29
  }

Interface

- interface A {
  fun foo() { print("A") }
  fun bar()
  }

- interface B {
  fun foo() { print("B") }
  fun bar() { print("bar") }
  }

- class C : A {
  override fun bar() { print("bar") }
  }

- class D : A, B {
  override fun foo() {
  super<A>.foo()
  super<B>.foo()
  }

  override fun bar() {
  super<B>.bar()
  }
  }

Extension metody

- fun MutableList<Int>.swap(index1: Int, index2: Int) {
  val tmp = this[index1] // 'this' corresponds to the list
  this[index1] = this[index2]
  this[index2] = tmp
  }

- extension metody se určují staticky

```
open class C
class D: C()

fun C.foo() = "c"
fun D.foo() = "d"

fun printFoo(c: C) {
    println(c.foo())
}

printFoo(D())   // prints out c
```

Datové třídy

- data class User(val name: String, val age: Int)
  - kompilátor automaticky generuje
    - equals()/hashCode()
    - toString() vracející "User(name=John, age=42)";
    - componentN()
    - copy()

- fun copy(name: String = this.name, age: Int = this.age) = User(name, age)

Generické typy

- class Box<T>(t: T) {
  var value = t
  }

- nejsou „wild-cards“ jako v Javě

```
interface Source<out T> {
    fun nextT(): T
}

fun demo(strs: Source<String>) {
    val objects: Source<Any> = strs
    // ...
}
```

```
interface Comparable<in T> {
    operator fun compareTo(other: T): Int
}

fun demo(x: Comparable<Number>) {
    x.compareTo(1.0)
    val y: Comparable<Double> = x
}
```

Singleton

- object DataProviderManager {
  fun registerDataProvider(provider: DataProvider) {
  // ...
  }

  val allDataProviders: Collection<DataProvider>
  get() = // ...
  }

- DataProviderManager.registerDataProvider(...)

Přetěžování operátorů

- a + b a.plus(b)
- a - b a.minus(b)
- a * b a.times(b)
- a / b a.div(b)
- a % b a.rem(b), a.mod(b) (deprecated)
- a..b a.rangeTo(b)
- …

- data class Counter(val dayIndex: Int) {
    operator fun plus(increment: Int): Counter {
        return Counter(dayIndex + increment)
    }
}

...

- ...