## Obiektowo-funkcyjny język programowania

Zbyszek Skowron

4 czerwca 2007

# Wprowadzenie

**Scala** jest obiektowo-funkcyjnym jezykiem programowania zaprojektowanym przez **Martina Odersky'ego**.

**Inne projekty Odersky'ego:**

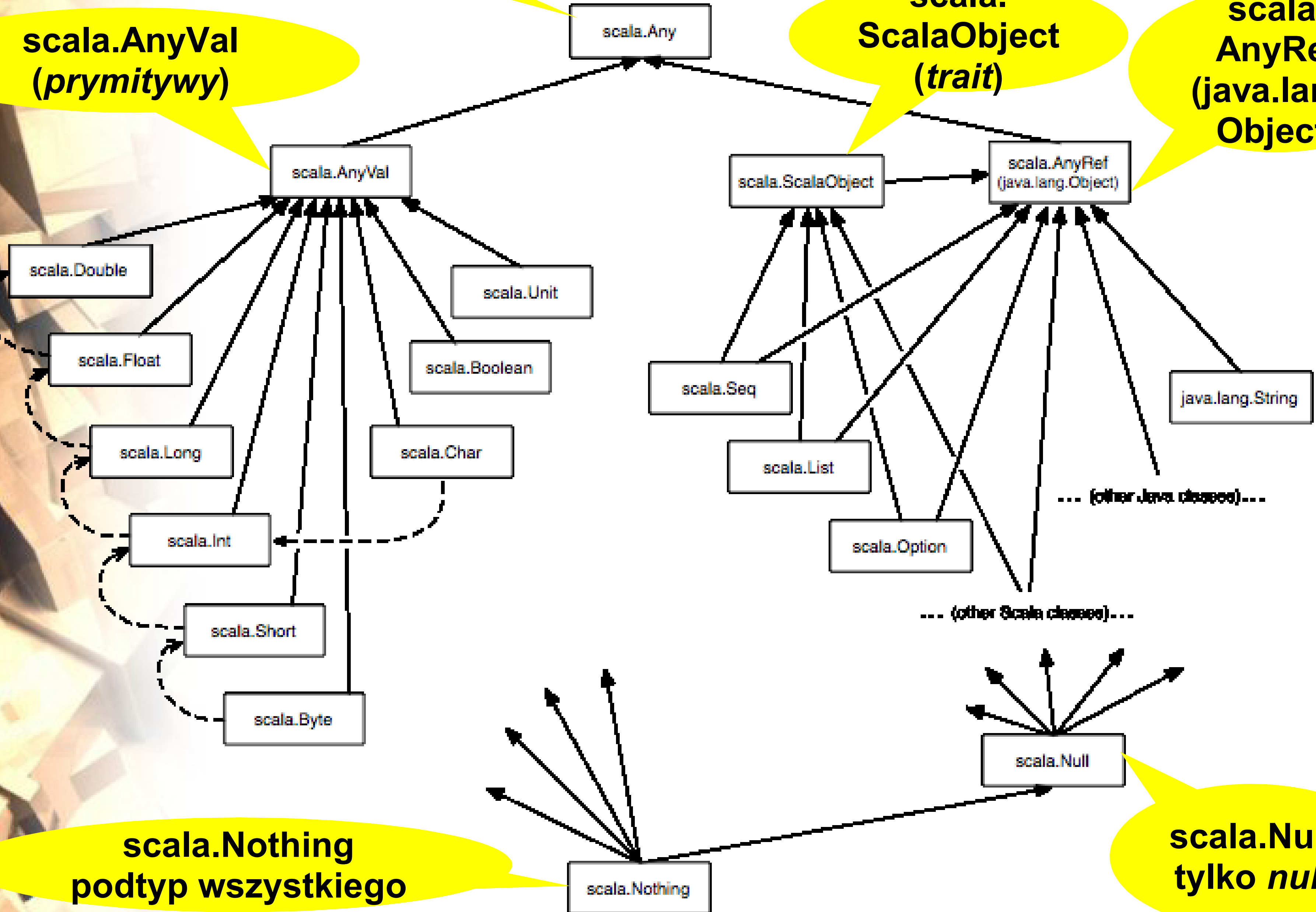
- Pizza (Java + generyki, wsk. do funkcji, typy algebraiczne)
- GJ (pierwsza wersja generyków Javy)
- Generyki Javy
- Funnel

# Scala

- W Scali:
  - wszystko jest obiektem, włącznie z:
    - wartościami liczbowymi,
    - funkcjami,
  - istnieją pojęcia:
    - klasy,
    - cechy (ang. *trait*).

- Obiekty są tworzone jako instancje klas.
- Scala jest statycznie typowana.
- Może być używana zarówno w środowisku JRE jak i .NET

# Hierarchia klas



# Hierarchia klas cd.

- W Scali mamy pojedyncze dziedziczenie po **klasach** i wielodziedziczenie po **cechach**.
- **Cechy** (*traits*) to analogi interfejsów, tyle że mogą być częściowo zaimplementowane.
- Każda klasa napisana w Scali domyślnie zawiera cechę **scala.ScalaObject**.
- Klasy Javy nie zawierają cechy **scala.ScalaObject**.
- Klasy dziedziczące po **scala.AnyVal** są predefiniowane. Po **scala.AnyVal** nie można dziedziczyć.
- **null** jest tylko dla typów referencyjnych.

# Program w Scali

```
import Predef._

object App {
   def main(args: Array[String]) {
      val pt = new Point(1, 2)
      println(pt)
      pt.move(10, 10)
      println(pt)
   }
}

object App2 extends Application {
   val pt = new Point(1, 2)
   println(pt)
   pt.move(10, 10)
   println(pt)
}
```

Domyślny import

Obiekt singletonowy

Punkt wejścia dla programu

Konstrukcja obiektu

Predef.println()

val - wartość
var - zmienna

Klasa Application zawiera pustą metodę main()

# Skróty notacyjne

Implementacja: tablica Javy

```
val greetStrings = new Array[String](3)
var s = greetStrings(0)
var t = greetStrings.apply(0)
greetStrings(0) = "Hello"
greetStrings.update(0, "Hello")

for(i <- 0 to 2)
    print(greetStrings(i))
for(i <- 0.to(2))
    print(greetStrings(i))

val oneTwo = List(1, 2)  // factory method
val threeFour = 3 :: 4 :: Nil
val oneTwoThreeFour = oneTwo ::: threeFour
oneTwo.count(s => s == 4)
oneTwo.foreach(println)
oneTwo.reverse
```

Obiekt singletonowy z metodą apply()

Funkcja anonimowa

# Elementy obiektowe w Scali

# Deklaracja klasy

```
package test;

class Point(xc: Int, yc: Int) {

  var x: Int = xc
  var y: Int = yc

  def this() = this(1,2)

  def move(dx: Int, dy: Int) {
    x = x + dx
    y = y + dy
  }

  override def toString(): String =
    "(" + x + ", " + y + ")";

  println("Ala ma kota")
}
```

Deklaracja pakietu

Główny konstruktor

Dodatkowy konstruktor

Nowa metoda (zwraca Unit)

Przedefiniowanie metody z nadklasy

Instrukcje w ciele klasy są wywoływane w czasie konstrukcji

# Dziedziczenie

```
class ColorPoint(u: Int, val v: Int, var color: String)
                                        extends Point(u, v) {

  override def equals(pt: Any): Boolean =
    pt.isInstanceOf[ColorPoint] &&
    (pt.asInstanceOf[ColorPoint].color == color)

  def << (col: String) = color = col

  this << "Zielono-" + color

  val theU = u

  def foo(pt: ColorPoint) = u == x || this == pt
}
```

Parametr konstruktora

Parametry dostępne na zewnątrz jako wartość i zmienna

Konstrukcja nadklasy

Przeciążanie operatorów

Parametry konstruktora wszędzie dostępne

Wywołanie equals()

# Przedefiniowywanie metod

```
class Upper
class Middle extends Upper
class Lower extends Middle

class Base {
  def getObj(o: Middle): Middle = new Middle
}

class Derived extends Base {

  override def getObj(o: Middle): Lower = new Lower
  //override def getObj(o: Upper): Middle = new Middle

  override def toString() = "Pod" + super.toString()

  def getAla(surname: String) = "Ala " + surname
  def getAla(age: Int) = "Ala, lat " + age
}
```

Covariant Return Type

Contravariant Parameter Type - błąd

Metody przeciążone

# Cechy (*traits, mixins*)

Deklaracja cechy (brak konstruktora)

Metoda abstrakcyjna

Metoda konkretna

Dziedziczenie

```
trait Similarity {
  def isSimilar(x: Any): Boolean

  def isNotSimilar(x: Any): Boolean = !isSimilar(x)
}

class Thing(val name: String) extends Similarity {
  def isSimilar(obj: Any) =
    obj.isInstanceOf[Thing] &&
    (name startsWith obj.asInstanceOf[Thing].name)
}

object TraitsTest extends Application {
  val ala = new Thing("Ala")
  val alastor = new Thing("Alastor")
  println(alastor.isSimilar(ala))     // == true
  println(alastor.isNotSimilar(ala)) // == false
}
```

Wywołanie wmieszanej metody

# Cechy - przykład

```
abstract class AbsIterator(foo:Int) {
  type T
  def hasNext: Boolean
  def next: T
}

trait RichIterator extends AbsIterator {
  def foreach(f: T => Unit): Unit =
    while (hasNext) f(next)
}

class StringIterator(s: String) extends AbsIterator(1) {
  type T = Char
  private var i = 0
  def hasNext = i < s.length()
  def next = { val ch = s.charAt(i); i += 1; ch }
}

class Iter extends StringIterator("Ala") with RichIterator
val iter = new Iter
iter.foreach(System.out.println)
```

Typ abstrakcyjny

Cecha rozszerzająca klasę - brak konstruktora

Konstruktor

Typ konkretny

Zmiksowanie klasy i cechy

Wywołanie wmieszanej metody

# Klasy abstrakcyjne i mixiny

- W **klasach abstrakcyjnych** mogą występować:
  - **metody** abstrakcyjne,
  - **wartości** abstrakcyjne,
  - **zmienne** abstrakcyjne,
  - **typy** abstrakcyjne.

```
abstract class Person {
  type Napis
  def fullName(): Napis
  val name: Napis
  var surname: Napis
}
```

- **Cechy** mogą dziedziczyć po cechach i po klasach:

```
class Base
trait MixinBase
trait Mixin extends Base with MixinBase
trait AnotherMixin
class Derived extends Mixin with AnotherMixin
```

# Klasy zagnieżdżone

Klasy zagnieżdżone są nieco inne niż w Javie:

```
class Graph {
  class Node
  def connect(m: Node, n: Node) {}
  def newNode: Node = new Node
  def connect2(m: Graph#Node, n: Graph#Node) {}
  def newNode2: Graph#Node = new Node
}

object NestedTest extends Application {
  val g = new Graph
  val h = new Graph
  val w = g;
  g.connect(g.newNode, g.newNode);
  //g.connect(g.newNode, w.newNode);
  //g.connect(g.newNode, h.newNode);
  g.connect2(g.newNode, g.newNode2);
  g.connect2(g.newNode, h.newNode);
}
```

Typy g.Node i h.Node są różne

Typy Graph#Node są takie same

# Typy sparametryzowane

# Typy sparametryzowane

- W Scali można tworzyć **typy sparametryzowane** innymi typami.
- Podobnie można parametryzować metody.
- Parametry typowe giną po kompilacji.

Dostępne mechanizmy:

| Scala | Java |
|---|---|
| `[Derived <: Base]` | `<Derived extends Base>` |
| `[Base >: Derived]` | `<Base super Derived>` |
| `[+T]` | brak |
| `[-T]` | brak |

**Oznaczenie pozycji w jakiej może występować typ: +kowariantnej, -kontrawariantnej**

**Można to określić w miejscu użycia**

# Prosty przykład

Typ parametryczny

```
class Container[A](a: A) {

  def getObj[B >: A]():B = a

  def getOtherObj[B <: Upper](b: B) = b;

  def getSome[B >: Lower <: Upper]():B =
    new Lower
}
```

Metoda polimorficzna

Typ ograniczony z obu stron

# Stos niemodyfikowalny

Określa sposób dziedziczenia

Klasa anonimowa

```
class Stack[+A] {
  def push[B >: A](elem: B): Stack[B] = new Stack[B] {
    override def top: B = elem
    override def pop: Stack[B] = Stack.this
    override def toString() = elem.toString() + " " +
                              Stack.this.toString()
  }
  def top: A = error("no element on stack")
  def pop: Stack[A] = error("no element on stack")
  override def toString() = ""
}

object VariancesTest extends Application {
  var s: Stack[Any] = null
  s = new Stack().push("hello"); // Stack[String]
  s = s.push(new Object())       // Stack[AnyRef]
  s = s.push(7)                  // Stack[Any]
  Console.println(s)
}
```

Stack[Derived] <: Stack[Base]

# Klasy abstrakcyjne i generyczne

```
abstract class Buffer {
  type T
  val element: T
}

abstract class SeqBuffer extends Buffer {
  type U
  type T <: Seq[U]
  def length = element.length
}
```

Zawężenie typu abstrakcyjnego

```
abstract class Buffer[+T] {
  val element: T
}

abstract class SeqBuffer[U, +T <: Seq[U]]
    extends Buffer[T] {
  def length = element.length
}
```

Zawężenie parametru typowego

# Ręczne typowanie `this`'a

Ręczne typowanie this'a

```
trait Comparable[t <: Comparable[t]] requires t {
  def < (that: t): boolean
  def <=(that: t): boolean =
    this < that || this == that
```

...bo tu jest potrzebny t, a nie Comparable[t]

```
  def > (that: t): boolean = that < this
  def >=(that: t): boolean = that <= this
}
```

# Ręczne typowanie `this`'a

```
abstract class Graph {
  type Node <: NodeIntf
  trait NodeIntf { def connectWith(node:Node) }

  def connect(start: Node, end: Node)


  class NodeImpl requires Node extends NodeIntf {
    def connectWith(node: Node) =
      connect(this, node)
  }
}

class ConcreteGraph extends Graph {
  class Node extends NodeImpl { def nop() {} }
  def connect(start: Node, end: Node) {}
}
```

Ręczne typowanie this'a

...bo tu jest potrzebny Node, a nie NodeImpl

# Elementy funkcyjne w Scali

# Funkcje w Scali

W Scali wszystkie funkcje są obiektami.

Typ funkcyjny: `(T1, ..., Tn) => R`

Czyli inaczej:

```
trait Functionn[-T1, . . ., -Tn,+R] {
  def apply(x1: T1,  . . ., xn: Tn): R
}
```

Typ funkcyjny jest kowariantny ze względu na typ wyniku i kontrawariantny ze względu na typy parametrów:

```
class Upper
class Lower extends Upper

def func(a: Upper, b: Upper): Lower = new Lower
var f: (Lower, Lower) => Upper = func
```

# Parametry funkcji

Funkcje mogą mieć wieloczłonowe listy parametrów i być częściowo aplikowane:

```
def add(a: Int)(b: Int) = a + b    //(Int)(Int) => Int
val inc = add(1) _                 // Int => Int
val dwa = inc(1)
```

**Częściowa aplikacja funkcji**

W Scali istnieje też przekazywanie parametrów *przez nazwę*:

**Przekazanie przez nazwę**

```
def whileLoop(cond: => Boolean)(body: => Unit): Unit =
  if (cond) { body ; whileLoop(cond)(body) } else {}
```

**Wyliczane przy każdym użyciu**

```
whileLoop(count > 0) { count -= 1 }
```

# Funkcje wyższego rzędu

```
object FunTest extends Application {

  class Printer {
    def layout[A](x: A) =
      println("Cześć " + x.toString() + "!")

    def print[A](printer: A => Unit, args: A*) =
      args.foreach(printer)
  }

  val p = new Printer
  p.print(p.layout, "Ala", "Ela", "Ula")

}
```

Funkcja wyższego rzędu

Automatyczne rzutowanie z metody na funkcję

# Funkcje anonimowe

W Scali istnieje kilka sposobów na zdefiniowanie funkcji anonimowej:

```
(x: Int) => x + 1

(x: Int, y: Int) => x + y

() => { println("Ala ma kota") }

new Function1[Int, Int] {
    def apply(x: Int): Int = x + 1
}

var k : Int => String =
{ case 1 => "Ala" case 2 => "ma" case _ => "kota" }
```

# Domknięcia

Domknięcie zwraca wartość ostatniego wyrażenia

Typ: `=> Int`

```
var count = 0;
for(i <- 0 to 3)
  println( {count = count + 1; count} )
```

Ten `return` kończy funkcję `f()`

```
def f() : (Object => Object) = {
  return { d:Object => return null };
}

try {
  f()("Ala") //scala.runtime.NonLocalReturnException
}
catch {
  case ex => println("Ex: " + ex)
  case _ => println("Never")
}
```

Wywołanie rzuci wyjątek

# Typy algebraiczne

W Scali istnieje skrócona notacja dla tworzenia klas wariantowych (*case classes*):

- można tworzyć bez **new**: `a = Some(5)`
- domyślnie dodawany modyfikator **val** dla parametrów konstruktora
- domyślnie przedefiniowywane metody **equals()** i **hashCode()** - porównanie strukturalne
- **toString()** zwraca napis `Klasa(parametry)`

```
case class Some(a: Int)
Some(5) == Some(5)                    //prawda
Some(5).toString() == "Some(5)"       //prawda
Some(5).a == 5                        //prawda
```

# Typy algebraiczne: przykład

```
class Expr
case class Var(x: String) extends Expr
case class Apply(f: Expr, e: Expr) extends Expr
case class Lambda(x: String, e: Expr) extends Expr

case class Value(e: Expr, env: Env)
type Env = String => Value

def eval(e: Expr, env: Env): Value =
  e match {
    case Var(x) =>
        env(x)
    case Apply(f, g) =>
        val Value(Lambda (x, e1), env1) = eval(f, env)
        val v = eval(g, env)
        eval (e1, (y => if (y == x) v else env1(y)))
    case Lambda(_, _) =>
        Value(e, env)
}
```

Klasy jako wzorce

Przypisanie na wzorzec

# Dopasowanie wzorców

```
val Some(x) = f()
```

```
val x = f() match { case Some(x) => x }
```

```
val x :: xs = List(1,2,3);
```

Konstruktor pary

```
val xpair = List(1,2,3) match
                        { case x :: xs => (x, xs) }
val x = xpair._1
val xs = xpair._2
```

Predykat

```
val x: Any = 5
x match {
  case 1 if false  => 1
  case (1,2,3)     => "Ala"
  case x: String   => Some(x)
  case h :: Nil    => h
  case _           => "Other"
}
```

Sprawdzanie wartości, struktury, typu...

# Ekstraktory

Sposób dopasowania wzorca do obiektu można określić pisząc ekstraktor: `unapply()`

```
object Twice {
  def apply(x: Int) = x * 2

  def unapply(z: Int) =
    if (z%2 == 0) Some(z/2) else None
}


val a = Twice(21)

a match {
  case Twice(n) => Console.println(n)
}
```

Ekstraktor, zwraca Option[A]

Stworzenie obiektu

Dopasowanie obiektu

Istnieje również możliwość dopasowywania kolekcji poprzez: `unapplySeq()`

# Część IV

# Inne ciekawe własności

# Parametry domyślne

- Funkcja w Scali może mieć **parametry domyślne**.
- Kandydatami na uzupełnienie parametrów domyślnych są **obiekty domyślne**, dostępne bezpośrednio w bieżącym zasięgu.

```
val dbl = (x:Int) => 2*x
implicit val id = (x:Int) => x

def morphSum(xs: Int*)
            (implicit morph: Int => Int) =
{
  var sum = 0
  xs.foreach( i => { sum = sum + morph(i) } )
  sum
}

println( morphSum(1,2,3)(dbl) )
println( morphSum(1,2,3) )
```

Ostatni człon listy parametrów może przyjmować wartości domyślne

Domyślny argument: id()

# Domyślne konwersje

Scala pozwala na definiowanie **domyślnych konwersji** (*views*) dla typów.

```
class FancyString(s: String) {
  def *(rep: Int) = {
    var cnt = rep
    var out = ""
    while(cnt > 0) { cnt = cnt - 1; out += s }
    out
  }
  def maKota = s + " ma kota"
}

object FancyTest extends Application {

  implicit def fancyWrapper(s: String) =
    new FancyString(s)

  println( "Ala" * 5 )
  println( "Ala" maKota )
}
```

implicit
S => T

Domyślna
konwersja

# Pętle `for`

Pętle `for` są tylko lukrem syntaktycznym do wywołań metod `map()`, `flatMap()`, `filter()` i `foreach()`.

```
for(i <- 1 until 10;
    j <- 1 until i if (i+j == 10) )
    yield (i, j)

(1 until 10)
  .flatMap {
     case i => (1 until i)
       .filter { j => (i+j == 10) }
       .map    { j => (i, j) }
  }
```

# Pakiety

Pakiety w Scali nie są tożsame z katalogami.

W jednym pliku może być zdefiniowanych wiele klas.

```
package p {

  package q {
    class Ela
  }
  class Ala {
    import q.Ela
    new Ela
  }
  class Ula
  package q { class Ola }
}

import p.Ala
import p._
import p.q.Ola, p.q._
import p.{Ala => Alicja, Ula => _, _ }
```

Importy lokalne

Zmiana nazwy importowanej klasy

# Kontrola dostępu

Kontrola dostępu do elementów klas w Scali jest dużo bardziej rozbudowana niż w Javie:

- **private** - klasowy private
- **private[this]** - obiektowy private
- **private[Cls]** - dla klasy Cls i singletona Cls
- **protected**, **protected[this]**, **protected[Cls]**

Kwalifikatory:

- **final** - dla metod i klas
- **sealed** - dla klas (jak **final**, ale można dziedziczyć w tym samym pliku źródłowym)

# Współpraca z JVM

- Klasy Scali są konwertowane na klasy Javy.
- Cechy są zamieniane na interfejsy.
- Metody, o ile to możliwe, zachowują swoje nazwy.
- String przechodzi na java.lang.String.
- Tablice są implementowane jako tablice Javy.

```
abstract class Test {
  var x : Int; // <=> abstrakcyjne akcesory
  // public abstract void x_$eq(int arg0)
  // public abstract int x()

  var y : String = ""; // <=> zmienna i akcesory
  // private String y;
  // public void y_$eq(String arg0)
  // public String y()
}
```

# Wtyczki

- Wtyczka dodająca obsługę Scali do **Eclipse**'a:
  - ma podświetlanie składni (wolne...),
  - nie wspomaga współpracy z Javą,
  - nie ma podpowiadania kodu,
  - ma błędy.
- Wtyczka do **IntelliJ IDEA** nie jest lepsza:
  - mimo że podpowiada kod,
  - to nie instaluje się na najnowszej **IDEI**.

Strona domowa Scali, dystrybucje i dokumentacja:

http://www.scala-lang.org