Podstawy programowania w języku Scala

Elementy programowania obiektowego (I)

Roman Dębski

Instytut Informatyki, AGH

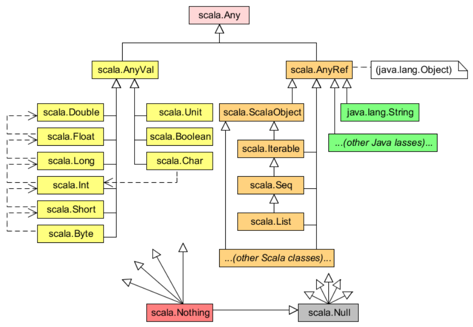
9 marca 2022

AGH

Plan wykładu

Plan wykładu

Hierarchia klas (typów) w Scali



Plan wykładu

Definicja klasy w Scali

```
class C1(val i: Int) {
  println("Inside primary constructor (1)")

  val attr1: Int = 3 * i + 4

  println("Inside primary constructor (2)")

  def m1(iP: Int) = 2 * iP + attr1

  println("Inside primary constructor (3)")
}
```

```
$ scala
scala> new C1(3)
Inside primary constructor (1)
Inside primary constructor (2)
Inside primary constructor (3)
```

Specyfikatory dostępu (pierwsza wzmianka)

Scala

```
class A {
  val a1: String = "a3"
  private val a2: Int = 42
  protected var a3: Double = -0.5

  def m1(m1P1: Int): Int = 42 * m1P1 + a2
  private def m2() = 5 * a2
  protected def m3() = a3
}
```

Java

```
public class A {
  private final java.lang.String a1;
  private final int a2;
  private double a3;
  public java.lang.String a1();
  private int a2();
  public double a3();
  public void a3_$eq(double);
  public int m1(int);
  private int m2();
  public double m3();
  public A();
}
```

domyślnie: Java - package, Scala - public

Parametry konstruktora głównego* (I)

```
class A(val i: Int)
```

```
public class A {
  private final int i;
  public int i();
  public A(int);
}
```

```
class A(var i: Int)
```

```
public class A {
  private int i;
  public int i();
  public void i_$eq(int); // i_=...
  public A(int);
}
```

*primary constructor

Parametry konstruktora głównego (II)

```
class A(i: Int, d1: Double) {
  def m1 = d1
}
```

```
public class A {
  private final double d1;
  public double m1();
  public A(int, double);
}
```

```
class A(iP: Int,
        dP: Double,
        private val sP: String) {
  private val i = iP
  val d = dP
}
```

```
public class A {
  private final java.lang.String sP;
  private final int i;
  private final double d;
  private java.lang.String sP();
  private int i();
  public double d();
  public A(int, double, java.lang.String);
}
```

Własne wersje metod dostępowych

```
class Person(private var _age: Int) {
  def age = {
    println("age getter working")
    _age }
  def age_= (newAge: Int) = {
    _age = newAge
    println("age setter working")
  }
}
object Appl { // objects are discussed later
  def main(args: Array[String]) {
    val p = new Person(10)
    println(p.age)
    p.age = 15 // p.age_=(15)
    println(p.age)
  }
}
```

```
public class Person {
  private int _age;
  private int _age();
  private void _age_$eq(int);
  public int age();
  public void age_$eq(int);
  public Person(int);
}
```

```
$ scala Appl
age getter working
10
age setter working
age getter working
15
```

Konstruktor prywatny

```
class A private (val i: Int,
                 var d: Double,
                 s: String) {
  def m = s
}
```

```
public class A {
  private final int i;
  private double d;
  private final java.lang.String s;
  public int i();
  public double d();
  public void d_$eq(double);
  public java.lang.String m();
  private A(int, double, java.lang.String);
}
```

Konstruktory pomocnicze* (I)

```
class A(val i: Int) {
  def this() = {
    this(0)
    println("Some more code...")
  }

  def this(d: Double) = {
    this(d.toInt)
    println("Some more code...")
  }
}
```

```
public class A {
  private final int i;
  public int i();
  public A(int);
  public A();
  public A(double);
}
```

*auxiliary constructors

Konstruktory pomocnicze (II)

```
class A private (val i: Int) {
  def this() = {
    this(0)
    println("Some more code...")
  }
}
```

```
public class A {
  private final int i;
  public int i();
  private A(int);
  public A();
}
```

```
class A private (val i: Int) {
  private def this() = {
    this(0)
    println("Some more code...")
  }
}
```

```
public class A {
  private final int i;
  public int i();
  private A(int);
  private A();
}
```

Konstruktory pomocnicze (III)

Zamiast wielu konstruktorów pomocniczych

```
class Person(val name: String = "", val age: Int = 0)
```

Wywołanie konstruktora klasy nadrzędnej (napisanej w Javie)

```
class Square(x: Int, y: Int, width: Int) extends
  java.awt.Rectangle(x, y, width, width)
```

@BeanProperty

```
import scala.beans.BeanProperty

class JavaBeanC(@BeanProperty var field: Int) {
  def m1 = 2 * field
}
```

```
public class JavaBeanC {
  private int field;
  public int field();
  public void field_$eq(int);
  public void setField(int);
  public int m1();
  public int getField();
  public JavaBeanC(int);
}
```

?

```
public class C1 {
  private final int a1;
  private double a2;
  public int getA1() { return a1; }
  public double getA2() { return a2; }
  public void setA2(double a2) {
    this.a2 = a2;
  }
  public int m1(int x) { return 2 * x; }
  public C1(int a1) { this(a1, 0); }
  public C1(int a1, double a2) {
          this.a1 = a1;
          this.a2 = a2;
  }
}
```

```
class C1(a1: Int, a2: Double) {
  def this(a1: Int) = { this(a1, 0) }
  def m1(x: Int) = {2 * x}
} // a)
```

```
class C1(val a1: Int, var a2: Double) {
  def this(a1: Int) = { this(a1, 0.0) }
  def m1(x: Int) { 2 * x }
} // b)
```

```
class C1(val a1: Int, var a2: Double) {
  def this(a1: Int) = { this(a1, 0) }
  def m1(x: Int) = 2 * x
} // c)
```

Plan wykładu

Kompilacja obiektu* w Scali

```
object O1 { val iVal = 10; private var dVar: Double = _
            def m1 = iVal * dVar
            private def m2(p1: Int) = p1 * iVal }
```

```
public final class O1$ {
  public static final O1$ MODULE$;
  private final int iVal;
  private double dVar;
  public static {};
  public int iVal();
  private double dVar();
  private void dVar_$eq(double);
  public double m1();
  private int m2(int);
  private O1$();
} // O1$.class
```

```
public final class O1 {
  public static double m1();
   Code:
      0: getstatic     #16 //Field O1$.MODULE$:LO1$;
      3: invokevirtual #18 //Method O1$.m1:()D
      6: dreturn
  public static int iVal();
   Code:
      0: getstatic     #16 //Field O1$.MODULE$:LO1$;
      3: invokevirtual #22 //Method O1$.iVal:()I
      6: ireturn
} // O1.class
```

* scala object vs. java singleton

Parametry konstruktora obiektu ???

```
object O2(iP: Int) {
  def m1 = iP
} // O2.scala
```

```
$ scalac O2.scala
O2.scala:1: error: traits or objects may not have parameters
object O2(iP: Int) {
         ^
one error found
```

Obiekt "towarzyszący" (companion object)

```
class C1(val i: Int) {
  def mC1 = i * C1.pmOC1(2 * i) // private!
  private def pmC1(a: Int) = i * a
}
object C1 {
  def mOC1(a: Double, inst: C1) =
    a * inst.pmC1(3) * inst.i
  private def pmOC1(b: Int) = 5 * b
}
object Appl {
  def main(args: Array[String]) {
    val c1Inst = new C1(2)
    print("c1Inst.mC1 = " + c1Inst.mC1)
    println(", C1.mOC1 = " +
             C1.mOC1(2.5, c1Inst))
  }
}
```

```
public class C1 {
  private final int i;
  public static double mOC1(double,C1);
  public int i(); public int mC1();
  public int C1$$pmC1(int); public C1(int);
  public C1(int); }
```

```
public final class C1$ {
  public static final C1$ MODULE$;
  public static {};
  public double mOC1(double, C1);
  public int C1$$pmOC1(int);
  private C1$(); }
```

```
$ scala Appl
c1Inst.mC1 = 40, C1.mOC1 = 30.0
```

Metoda apply() [por. rodzina wzorców "Factory"]

```
class C1 private (val i: Int) {}
object C1 {
  def apply(i: Int = 1) = new C1(i)
  def apply(d: Double) = new C1(d.toInt)
}
object Appl {
  def main(args: Array[String]) {
    println(C1(3).i)
    println(C1().i)
    println(C1(4.7).i)
  }
}
```

```
$ scala Appl
3
1
4
```

```
public class C1 {
  private final int i;
  public static int apply$default$1();
  public static C1 apply(double);
  public static C1 apply(int);
  public int i();
  public C1(int);
}
```

```
public final class C1$ {
  public static final C1$ MODULE$;
  public static {};
  public C1 apply(int);
  public C1 apply(double);
  public int apply$default$1();
  private C1$();
}
```

Obiekt aplikacji (+App trait)

```
object Main {
  def main(args: Array[String])
    println("Aloha!")
  }
}
```

```
object Main extends App {
  println("Aloha!")
}
```

```
object Main {
  def main(args: Array[Any]) {
    println("Aloha!")
  }
} //*
```

```
$ scalac Main.scala # version *
Main.scala:1: warning: Main has a main method with parameter type Array[String],
 but Main will not be a runnable program.
 Reason: main method must have exact signature (Array[String])Unit
object Main {
      ^
one warning found
```

Plan wykładu

Klasy abstrakcyjne: atrybuty i metody abstrakcyjne

```
class AbstrC(val i: Int) {
  private val theAnswer = 42
  def absrM1(d: Double): Double
}
```

```
$ scalac AbstrC.scala
AbstrC.scala:1: error: class AbstrC needs to be abstract,
since method absrM1 is not defined
class AbstrC(val i: Int) {
    ^ one error found
```

```
class AbstrC(val i: Int) {
  val theAnswer: Int
  def absrM1(d: Double): Double =
    d * theAnswer }
```

```
$ scalac class.scala
... error: class AbstrC needs to be abstract
since value theAnswer is not defined
...
```

```
abstract class AbstrC(val i: Int) {
  val theAnswer: Int
  def absrM1(d: Double): Double
}
```

```
public abstract class AbstrC {
  private final int i; public int i();
  public abstract int theAnswer();
  public abstract double absrM1(double);
  public AbstrC(int); }
```

Dziedziczenie: redefinicja (overriding) metod

```
abstract class AbstrC1(val i: Int) {
  val theAnswer: Int
  def m1(d: Double): Double = d * theAnswer }

abstract class AbstrC2(i: Int) extends AbstrC1(i) {
  override def m1(d: Double) = theAnswer.toDouble }

class ConrC1(i: Int) extends AbstrC2(3 * i) {
  override val theAnswer: Int = 42 }

class ConrC2 extends AbstrC2(2) {
  override val theAnswer: Int = 44
  override def m1(d:Double): Double = d + theAnswer }
```

```
public abstract class AbstrC2 extends AbstrC1 {
  public double m1(double);
  public AbstrC2(int); }
```

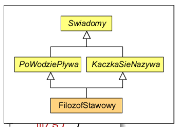
```
public abstract class AbstrC1 {
  private final int i; public int i();
  public abstract int theAnswer();
  public double m1(double);
  public AbstrC1(int); }
```

```
public class ConrC1 extends AbstrC2 {
  private final int theAnswer;
  public int theAnswer();
  public ConrC1(int); }
```

```
public class ConrC2 extends AbstrC2 {
  private final int theAnswer;
  public int theAnswer();
  public double m1(double);
  public ConrC2(); }
```

Plan wykładu

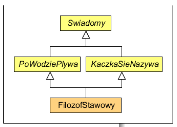
Traits* & mix-ins: przykład (1/3)



```
trait Swiadomy { def istotaBytu = "" }

trait PoWodziePlywa extends Swiadomy {
  private val vMax = 0.4 // m/s
  def plywaj() = println("Plywam, plywam (z predkoscia " + vMax
  override def istotaBytu = "Po wodzie plywam. " + super.istotaBytu
}

trait KaczkaSieNazywa extends Swiadomy {
  override def istotaBytu = "Kaczka sie nazywam. " + super.istotaBytu
}

class FilozofStawowy extends PoWodziePlywa with KaczkaSieNazywa {
  override def istotaBytu = "Plywam, wiec jestem! (" + super.istotaBytu + ")"
}
```

* vs. Java 8 default methods (motivation, state, linearization,...)

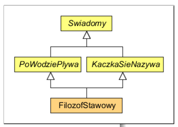
Traits & mix-ins: przykład (2/3)



```
class StworzenieBoze

object Appl {
  def main(agrs: Array[String]) {
    val folozof = new FilozofStawowy
    println(folozof.istotaBytu)
    folozof.plywaj()
    val chibaRiba = new StworzenieBoze with PoWodziePlywa with KaczkaSieNazywa
    println(chibaRiba.istotaBytu)
    val nieRiba = new StworzenieBoze with KaczkaSieNazywa with PoWodziePlywa
    println(nieRiba.istotaBytu) }}
```

```
$ scala Appl
Plywam, wiec jestem! (Kaczka sie nazywam. Po wodzie plywam. )
Plywam, plywam (z predkoscia 0.4 m/s)
Kaczka sie nazywam. Po wodzie plywam.
Po wodzie plywam. Kaczka sie nazywam.
```

Traits & mix-ins: przykład (3/3) [new Class with T1 with T2 → kod JVM]



```
public final class Appl$$anon$1 extends StworzenieBoze
  implements PoWodziePlywa,KaczkaSieNazywa {
  private final double PoWodziePlywa$$vMax;
  public java.lang.String KaczkaSieNazywa$$super$istotaBytu();
  public java.lang.String istotaBytu();
  public double PoWodziePlywa$$vMax();
  public java.lang.String PoWodziePlywa$$super$istotaBytu();
  public void PoWodziePlywa$_setter_$PoWodziePlywa$$vMax_$eq(double);
  public void plywaj();
  public Appl$$anon$1();
}
```

```
public final class Appl$$anon$2 extends StworzenieBoze
  implements KaczkaSieNazywa, PoWodziePlywa {
  private final double PoWodziePlywa$$vMax;
  // ...
}
```

(Re)definicja "operatorów"

```
trait Int2DVec {
  val x1: Int
  val x2: Int
  def +(other: Int2DVec): Int2DVec
  def unary_- : Int2DVec
}
```

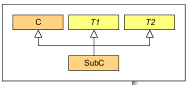
```
object Appl {
  def main(args: Array[String]) {
    val v1 = Int2DVec(1, 2)
    val v2 = Int2DVec(11, 22)
    println("v2-v1="+(v2+(-v1)))
  }}
```

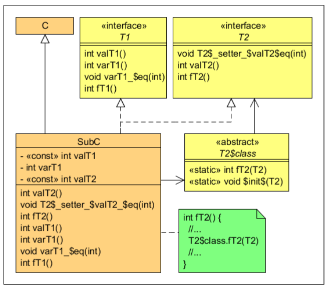
```
object Int2DVec {
  def apply(x1: Int, x2: Int): Int2DVec =
    new Int2DVecImpl(x1, x2)

  private class Int2DVecImpl(val x1: Int, val x2: Int)
    extends Int2DVec {
      override def +(other: Int2DVec): Int2DVec =
        Int2DVec(x1 + other.x1, x2 + other.x2)
      override def unary_- : Int2DVec =
        new Int2DVecImpl(-x1, -x2)
      override def toString() =
        "(" + x1.toString + "," + x2.toString + ")"
  }}
```

```
$ scala Appl
v2 - v1 = (10,20)
```

Traits → JVM, przykład (1/3)

```
trait T1 {
  val valT1: Int
  var varT1: Int
  def fT1: Int
}
trait T2 {
  val valT2 = 2
  def fT2 = 10 * valT2
}
class C(val a1: Int) {
  def fC = a1 * a1
}
class SubC(a1:Int)
extends C(a1) with T1 with T2 {
  override val valT1 = 3
  override var varT1 = 4
  override def fT1 = valT1 * varT1
}
```





Traits → JVM, przykład (2/3)

```
trait T2 {
  val valT2 = 2
  def fT2 = 10 * valT2
}
```

```
public interface T2 {
  public abstract void T2$_setter_$valT2_$eq(int);
  public abstract int valT2();
  public abstract int fT2();
}

public abstract class T2$class {
  public static int fT2(T2);
  public static void $init$(T2);
}
```

```
trait T1 {
  val valT1: Int
  var varT1: Int
  def fT1: Int
}
```

```
public interface T1 {
  public abstract int valT1();
  public abstract int varT1();
  public abstract void varT1_$eq(int);
  public abstract int fT1();
}
```

Traits → JVM, przykład (3/3)

```
trait T1 {
  val valT1: Int
  var varT1: Int
  def fT1: Int
}
trait T2 {
  val valT2 = 2
  def fT2 = 10 * valT2
}
class C(val a1: Int) {
  def fC = a1 * a1
}
class SubC(a1:Int)
extends C(a1) with T1 with T2 {
  override val valT1 = 3
  override var varT1 = 4
  override def fT1 = valT1 * varT1
}
```

```
public class SubC extends C implements T1,T2 {
  private final int valT1;
  private int varT1;
  private final int valT2;
  public int valT2();
  public void T2$_setter_$valT2_$eq(int);
  public int fT2();
  public int valT1();
  public int varT1();
  public void varT1_$eq(int);
  public int fT1();
  public SubC(int);
}
```

Plan wykładu

Typy strukturalne (vs. duck typing*)

```
trait Quackable {
  def quack(): Unit
}

class Duck extends Quackable {
  override def quack() =
    println("Quack, quack,...")
}

class J23 {
  def quack() = {
    print("Quack, quack... ")
    println("J-23 znowu nadaje!")
  }
}
```

```
object Appl {
  def quack(duckLike: {def quack(): Unit}) = {
    import scala.language.reflectiveCalls
    duckLike.quack()
  }

  def main(agrs: Array[String]) {
    quack(new Duck)
    quack(new J23)
  }
}
```

```
$ scala Appl
Quack, quack,...
Quack, quack... J-23 znowu nadaje!
```

*jeśli chodzi jak kaczka i kwacze jak kaczka, to musi być kaczką

nie dynamiczne, a statyczne typowanie (deklaracja duckLike: {def quack(): Unit} → błędy w czasie kompilacji, a nie wykonania)