

# Základy Scaly – 2. část

Radek Miček

2016

# Case classy – úvod


- Stručný zápis tříd, jejichž úkolem je držet data předaná v konstruktoru:

```
case class RpcConfig(server: String, timeout: Int)
```

- Pro case classy kompilátor mj. implementuje:
  - Metodu **apply**, jenž lze použít pro vytváření instancí:

```
val config = RpcConfig("http://mapy.cz", 30)
```

Metoda **apply** má pojmenované parametry, což se hodí, když je jich mnoho.

```
val config2 = RpcConfig(server = "http://mapy.cz", timeout = 30)
```

- Metodu **copy**:

```
val devConfig = config.copy(server = "http://mapy.dev")
```

Instance case class jsou standardně neměnné, metoda **copy** vytvoří novou instanci, která se od té původní liší jen v předaných parametrech.

# Case classy – úvod (2)

- Pro case classy kompilátor mj. implementuje:

- Vlastnost pro každý parametr konstruktoru:

```
println(s"Server je: ${config.server}")
```

- Podporu pro pattern matching:

```
val RpcConfig(s, t) = config
```

```
println(s"Server je: $s")
```

- Strukturální rovnost a hašování.

- Instance lze použít v množinách (**Set**) nebo jako klíče map (**Map**).

- Metodu **toString**.

# Case classy v hierarchii

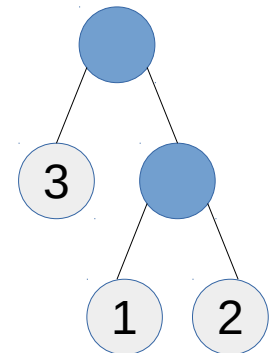
- Case classy mohou být součástí větší hierarchie tříd.
  - Nemělo by se z nich dědit (pokud si hierarchii tříd představíme jako strom, budou case classy v listech).
- Pomocí case class můžeme reprezentovat stromové struktury:

```
sealed trait Tree  
case class Fork(left: Tree, right: Tree) extends Tree  
case class Leaf(data: Int) extends Tree
```

Neprázdný úplný binární strom.

```
Fork(  
  Leaf(3),  
  Fork(Leaf(1), Leaf(2))  
)
```

Příklad instance stromu.



# Příklad: Dotazy na statserver

- Reprezentace dotazů na statserver:
  - Dotaz: pohlaví, okres, kombinace dotazů pomocí logických spojek AND, NOT.

**sealed** říká, že přímé podtřídy mohou být deklarovány pouze ve stejném souboru.

```
sealed trait Query
case class SexEq(sex: Sex) extends Query
case class DistrictEq(district: String) extends Query
case class And(a: Query, b: Query) extends Query
case class Not(query: Query) extends Query
```

```
sealed trait Sex
case object Male extends Sex
case object Female extends Sex
```

**Male** a **Female** jsou singleton objekty (mají jednu instanci v celém programu).

```
And(
  Not(DistrictEq("CZ0101")), // CZ0101 je Praha 1 podle NUTS 4
  SexEq(Female)
)
```


Ženy, jenž nejsou z Prahy 1.

# Příklad: Dotazy na statserver (2)

- Funkce lze definovat pomocí pattern matchingu:

```
val q = And(  
  Not(DistrictEq("CZ0101")), // CZ0101 je Praha 1 podle NUTS 4  
  SexEq(Female)  
)
```

```
case class Person(sex: Sex, district: String) {  
  def matches(q: Query): Boolean = q match {  
    case SexEq(s) => s == sex  
    case DistrictEq(d) => d == district  
    case And(a, b) => (this matches a) && (this matches b)  
    case Not(q) => !(this matches q)  
  }  
}
```

 Zastíní parametr **q**.

Funkce **matches** vrátí,  
zda člověk vyhovuje dotazu.

```
// Vrátí: List(Person(Female,CZ0201))  
List(Person(Male, "CZ0101"), Person(Female, "CZ0201"))  
  .filter(_ matches q)
```

# Příklad: Dotazy na statserver (3)

- Přidáme OR:

```
case class Or(a: Query, b: Query) extends Query
```

Díky tomu, že **Query** je deklarován jako **sealed**, ví kompilátor, jaké podtřídy **Query** má, a může nás varovat, když zjistí, že jsme nějaký dotaz zapomněli zpracovat.

- Od kompilátoru dostaneme varování:

Warning:(řádek, sloupec) match may not be exhaustive.

It would fail on the following input: Or(\_, \_)

```
def matches(q: Query): Boolean = q match {  
    ^
```

- Je třeba upravit i funkci matches:

```
def matches(q: Query): Boolean = q match {  
  case SexEq(s) => s == sex  
  case DistrictEq(d) => d == district  
  case And(a, b) => (this matches a) && (this matches b)  
  case Or(a, b) => (this matches a) || (this matches b)  
  case Not(q) => !(this matches q)  
}
```

# Příklad: Aritmetické výrazy

- Reprezentace aritmetických výrazů:

```
sealed trait Expr
case class Const(i: Int) extends Expr
case class Var(x: String) extends Expr
case class Add(a: Expr, b: Expr) extends Expr
```

```
// (3 + 7) + (x + 0)
Add(Add(Const(3), Const(7)), Add(Var("x"), Const(0)))
```



# Příklad: Aritmetické výrazy (2)

- Funkce **eval** na vyhodnocení výrazů:

```
def eval(e: Expr, vars: Map[String, Int]): Int = e match {  
  case Const(i) => i  
  case Var(x) => vars(x)  
  case Add(a, b) => eval(a, vars) + eval(b, vars)  
}  
  
// (3 + 7) + (x + 0)  
val e = Add(Add(Const(3), Const(7)), Add(Var("x"), Const(0)))  
  
eval(e, Map("x" -> 1))
```

# Kvíz: Aritmetické výrazy

- Co je špatného na funkci **simplifyBad**, jenž zjednodušuje aritmetické výrazy?

```
sealed trait Expr
case class Const(i: Int) extends Expr
case class Var(x: String) extends Expr
case class Add(a: Expr, b: Expr) extends Expr

def simplifyBad(e: Expr): Expr = e match {
  case Const(_) | Var(_) => e
  case Add(Const(i), Const(j)) => Const(i + j)
  case Add(x, Const(0)) => simplifyBad(x)
  case Add(Const(0), x) => simplifyBad(x)
}
```

# Kvíz: Aritmetické výrazy (2)

- Co je špatného na funkci **simplifyBad**, jenž zjednodušuje aritmetické výrazy?

```
sealed trait Expr
case class Const(i: Int) extends Expr
case class Var(x: String) extends Expr
case class Add(a: Expr, b: Expr) extends Expr

def simplifyBad(e: Expr): Expr = e match {
  case Const(_) | Var(_) => e
  case Add(Const(i), Const(j)) => Const(i + j)
  case Add(x, Const(0)) => simplifyBad(x)
  case Add(Const(0), x) => simplifyBad(x)
}
```

Nápověda: Na vstupu **Add(Var("x"), Var("y"))** (tj.  $x + y$ ) **simplifyBad** vyhodí výjimku.

# Kvíz: Aritmetické výrazy (3)

- Odpověď: **simplifyBad** neumí zpracovat některé výrazy (kompilátor by ohlásil varování).

```
sealed trait Expr
case class Const(i: Int) extends Expr
case class Var(x: String) extends Expr
case class Add(a: Expr, b: Expr) extends Expr

def simplifyBad(e: Expr): Expr = e match {
  case Const(_) | Var(_) => e
  case Add(Const(i), Const(j)) => Const(i + j)
  case Add(x, Const(0)) => simplifyBad(x)
  case Add(Const(0), x) => simplifyBad(x)
}
```

Nápověda: Na vstupu **Add(Var("x"), Var("y"))** (tj.  $x + y$ ) **simplifyBad** vyhodí výjimku.

# Kvíz: Aritmetické výrazy (4)

- Funkce pro zjednodušování výrazů:

```
def simplify(e: Expr): Expr = e match {  
  case Const(_) | Var(_) => e  
  case Add(Const(i), Const(j)) => Const(i + j)  
  case Add(x, Const(0)) => simplify(x)  
  case Add(Const(0), x) => simplify(x)  
  case Add(x, y) => /* doplň */  
}
```

A, B, C nebo D?

A `simplify(Add(x, y))`

B `val sx = simplify(x)`  
`val sy = simplify(y)`  
`Add(sx, sy)`

C `val sx = simplify(x)`  
`val sy = simplify(y)`  
`if (x != sx || y != sy)`  
    `simplify(Add(sx, sy))`  
`else e`

D `val sx = simplify(x)`  
`val sy = simplify(y)`  
`simplify(Add(sx, sy))`

# Kvíz: Aritmetické výrazy (5)

- Funkce pro zjednodušování výrazů:

```
def simplify(e: Expr): Expr = e match {  
  case Const(_) | Var(_) => e  
  case Add(Const(i), Const(j)) => Const(i + j)  
  case Add(x, Const(0)) => simplify(x)  
  case Add(Const(0), x) => simplify(x)  
  case Add(x, y) => /* doplň */  
}
```

Na vstupu  $x + y$  neskončí.

$(1 + 2) + 3$  zjednoduší pouze na  $3 + 3$ .

A ~~`simplify(Add(x, y))`~~

B ~~`val sx = simplify(x)  
val sy = simplify(y)  
Add(sx, sy)`~~

C `val sx = simplify(x)  
val sy = simplify(y)  
if (x != sx || y != sy)  
 simplify(Add(sx, sy))  
else e`

D ~~`val sx = simplify(x)  
val sy = simplify(y)  
simplify(Add(sx, sy))`~~

Na vstupu  $x + y$  neskončí.

# Vracení chyb

- Alternativou k výjimkám je signalizovat chybu pomocí návratové hodnoty.
- Standardní knihovna obsahuje typy **Either** a **Option**.

# Either

- Buď (**Either**) je vše v pořádku (**Right**) nebo nastala chyba (**Left**):

```
sealed trait Either[+L, +R]
case class Right[R](r: R) extends Either[Nothing, R]
case class Left[L](l: L) extends Either[L, Nothing]
```

Při chybě bude  
vrácen řetězec, jinak  
hodnota typu **T**.

```
// Vrací první prvek seznamu nebo chybové hlášení.
def prvniPrvek[T](xs: List[T]): Either[String, T] =
  xs match {
    case List() => Left("Seznam je prázdný - nemá první prvek!")
    case List(x, _) => Right(x)
  }
```

```
// Implementace pomocí výjimek (pro srovnání).
def prvniPrvekExn[T](xs: List[T]): T =
  xs match {
    case List() => sys.error("Seznam je prázdný - nemá první prvek!")
    case List(x, _) => x
  }
```

Vyhodí **RuntimeException**.



# Option


**Option** se také používá jako náhrada za **null**.

- Typ **Option** použijeme, pokud nás nezajímají podrobnosti o chybě:

**None** = žádný výsledek se nepodařilo spočítat.  
**Some** = nějaký výsledek se podařilo spočítat.

```
sealed trait Option[+T]  
case class Some[T](a: T) extends Option[T]  
case object None extends Option[Nothing]
```

Ve skutečnosti bychom použili **xs.headOption**.



```
def prvniPrvek[T](xs: List[T]): Option[T] =  
  xs match {  
    case List() => None  
    case List(x, _) => Some(x)  
  }
```

# Výhody Option/Either

- Z typu je vidět, že funkce může selhat.
  - Scala nemá checked exceptions.
- Rychlé, když nastává mnoho chyb.
  - Vyhození + chycení výjimky je pomalejší než vrácení + otestování **None** (pod JVM).
- Funkce pro kombinování hodnot typu **Option[T]**:

```
for {  
  x <- xs.headOption  
  y <- ys.headOption  
  z <- zs.headOption  
} yield x + y + z
```

Pokud jsou všechny 3 sekvence neprázdné, vrátí součet prvních prvků (zabalený uvnitř **Some**), jinak vrátí **None**.

# Implicitní konverze

- Speciální funkce, které kompilátor automaticky doplní do kódu, aby kód prošel typovou kontrolou.
  - Aplikace: „Přidávání“ metod ke třídám.

# „Přidávání“ metod ke třídám

- **String** (třída z Javy) nemá metodu **toInt**, přesto můžeme psát:

```
"5".toInt
```

- Aby se výraz přeložil, doplní kompilátor volání funkce **augmentString**:

```
augmentString("5").toInt
```

- **augmentString** zabalí řetězec do třídy **StringOps**, která již metodu **toInt** má.

**StringOps** má celou řadu užitečných metod: **lines** (rozdělí řetězec na řádky), **r** (převéde řetězec na regulární výraz), **\*** (jako v Pythonu), metody známé ze sekvencí (**head**, **drop**, **take**, **reverse**, ...), ...

# „Přidání“ metody isOdd

- Třídě **Int** „přidáme“ metodu **isOdd** určující, zda je číslo liché:

Napřed si připravíme třídu, jenž má potřebné metody.

```
case class IntExtensions(i: Int) {  
  def isOdd = i % 2 != 0  
}
```

Poté vytvoříme funkci, která **Int** zabalí do naší třídy.

Klíčové slovo **implicit** umožňuje kompilátoru použít funkci jako implicitní konverzi.

```
implicit def augmentInt(i: Int) = IntExtensions(i)
```

3.isOdd


4.isOdd

← Ve skutečnosti se volá **augmentInt(4).isOdd**.

# „Přidání“ metody isOdd (2)

- Je zvykem psát:


Implicitní třída – definice třídy a implicitní konverze v jednom.



```
implicit class IntExtensions(val i: Int) extends AnyVal {  
  def isOdd = i % 2 != 0  
}
```

3.isOdd

4.isOdd



Hodnotová třída – kompilátor ji eliminuje (po kompilaci nebude třída existovat).

# Implicitní parametry

- Scala umí doplnit chybějící argumenty.
  - Doplňuje se dle typu (ne dle jména parametru):

Implicitní parametry.

```
def myFunction(a: Int)(implicit b: Int, c: String) =  
  println(s"a = $a, b = $b, c = $c")
```

```
implicit val x = 3  
implicit val y = "Ahoj"
```

Jako argumenty budou doplňovány pouze implicitní hodnoty (bez klíčového slova **implicit** to nebude fungovat).

`myFunction(3)`

Dělají totéž.

`myFunction(3)(x, y)`

# K čemu je to dobré?

- V knihovnách bývá třídící funkce, která jako parametr bere funkci pro porovnávání:

```
def sort[T](xs: List[T], compare: (T, T) => Int): List[T]
```

- S pomocí implicitních parametrů může kompilátor třídící funkci předávat automaticky:

Místo obyčejné funkce `(T, T) => Int` používáme typ `Compare[T]`, neboť není vhodné používat běžné typy (např. funkce, `String`, `Int`, ...) jako implicitní hodnoty.

```
case class Compare[T](compare: (T, T) => Int)
```

Uvnitř funkce `sort` můžeme volat např. `cmp.compare(xs.head, xs.last)`.

```
def sort[T](xs: List[T])(implicit cmp: Compare[T]): List[T] = ???
```

```
implicit val compareInt = Compare[Int]((i, j) => i - j)
```

```
sort(List(4, 2, 1, 3))
```



# Knihovný typ Ordering[T]

- Pro porovnávání obsahuje standardní knihovna typ **Ordering[T]**.
- Pro třídění mají sekvence metodu **sorted**:

*List*(4, 2, 1, 3).sorted

Dělají totéž.

*List*(4, 2, 1, 3).sorted(*Ordering*.Int)

# Implementace Ordering[T]

- Instanci **Ordering[T]** můžeme vytvořit i pro vlastní typ:

```
case class Person(name: String, salary: Double)
```

```
object Person {  
  implicit val personOrdering = Ordering.by((p: Person) => p.salary)  
}
```

Uspořádání dle platu.

```
List(  
  Person("Franz", 50),  
  Person("Kafka", 40),  
  Person("Felice", 45)  
).sorted
```

Kompilátor automaticky doplní argument **personOrdering**.

# Serializace do JSONu

```
type Json = String
```

```
case class JsonWriter[A](toJson: A => Json)
```

```
object JsonWriter {  
  implicit val intInstance = JsonWriter[Int](_.toString)  
  implicit val stringInstance = JsonWriter[String]("\"" + _ + "\"")  
  implicit def listInstance[A](implicit writer: JsonWriter[A]) =  
    ↑ JsonWriter[List[A]](_.map(writer.toJson).mkString("[", ", ", "]"))  
}
```

Funkce pro konstrukci implicitních hodnot. Máme-li **A**, jenž jde zapisovat do JSONu, pak díky této funkci půjde do JSONu zapisovat i **List[A]**.

```
def saveToDb[A](a: A)(implicit writer: JsonWriter[A]) = {  
  val json = writer.toJson(a)  
  println(s"Saving to database: $json.")  
}
```

```
saveToDb(5)  
saveToDb("Text")  
saveToDb(List(1, 2))  
saveToDb(List(List(1, 2), List(3))) ←
```

Kompilátor přidá argument  
**listInstance(listInstance(intInstance))**  
typu **JsonWriter[List[List[Int]]**.

# Kvíz: Implicitní parametry

- Jaký implicitní argument kompilátor doplní?

```
List(  
  ("Franz", 50),  
  ("Kafka", 40),  
  ("Felice", 45)  
).sorted
```

A *Ordering*.String

B *Ordering*.Int

C *Ordering*.Tuple2(  
 *Ordering*.String,  
 *Ordering*.Int  
)

D *Ordering*.Tuple3(  
 *Ordering*.String,  
 *Ordering*.Int,  
 *Ordering*.Double  
)

# Kvíz: Implicitní parametry (2)

- Je třeba doplnit argument typu **Ordering[(String, Int)]**.

```
List(  
  ("Franz", 50),  
  ("Kafka", 40),  
  ("Felice", 45)  
).sorted
```

Typ **Ordering[String]**.

Typ **Ordering[Int]**.

A ~~Ordering.String~~

B ~~Ordering.Int~~

**Tuple2** je funkce.

Typ **Ordering[(String, Int, Double)]**.

C `Ordering.Tuple2(  
 Ordering.String,  
 Ordering.Int  
)`

D ~~`Ordering.Tuple3(  
 Ordering.String,  
 Ordering.Int,  
 Ordering.Double  
)`~~

# Implicity místo rozhraní

- Trik, který jsme si ukázali s `Ordering[T]` nebo `JsonWriter[T]`, představuje alternativu k rozhraním.
- Místo, aby typ `T` implementoval rozhraní, stačí udělat implicitní hodnotu typu `Ordering[T]` nebo `JsonWriter[T]`.
  - Obecně implicity mohou nahradit ta rozhraní, kde implementace nepotřebuje do třídy přidat data.
- Výhody oproti rozhraním:
  - Kvůli implementaci není třeba měnit třídu.
  - Implementace rozhraní může být jen jedna, implicitních hodnot mnoho (to, jaká se použije, lze řešit importy).

Typům `Ordering[T]` a `JsonWriter[T]` se někdy říká typové třídy. Implicitní hodnoty `Ordering.String` nebo `JsonWriter.intInstance` jsou instance zmíněných typových tříd. Terminologie pochází z Haskellu.

# Kvíz: Riziko implicitů

- Implicitní konverze a implicitní parametry mohou zamlžit význam programu.
- Příklad:
  - Jak funguje následující výraz? Co vrací?

`Map(8 -> "osm").map(_._1)`

Nemůže vrátit hodnotu typu `Map`, protože `map` nevrací dvojice.

- Odpověď v konzoli Scaly:

```
import scala.reflect.runtime.universe._
```

```
showCode(reify { Map(8 -> "osm").map(_._1) }.tree)
```

## Kvíz: Riziko implicitů (2)

- Implicitní konverze a implicitní parametry mohou zamlžit význam programu.
- Příklad – řešení:
  - Jak funguje následující výraz? Co vrací?

`Map(8 -> "osm").map(_._1)`

Typ hodnoty je `Iterable[Int]`,  
hodnota je však `List(8)`.

- Odpověď v konzoli Scaly:

```
import scala.reflect.runtime.universe._
```

```
showCode(reify { Map(8 -> "osm").map(_._1) }.tree)
```

```
Predef.Map.apply(Predef.ArrowAssoc(8).->("osm")).map(((x$1) => x$1._1))(Iterable.canBuildFrom).
```



# Riziko implicitů

- Implicitní konverze a implicitní parametry mohou zamlžit význam programu.
- Definice implicitní hodnoty nebo její import může změnit význam programu.
  - A to aniž by se kód, jehož význam se změnil, objevil v diffu.

# Konec druhé části

- Otázky?