

Základy Scaly

Radek Miček

2016

Programovací jazyk Scala

- Kompilovaný jazyk.
- Platformy: JVM, JavaScript (pomocí ScalaJS).
- Podporuje:
 - OOP
 - i FP.
- Statický typový systém
(silnější než Java, C#, F#, C++).

První program

- Zkompiluje se na třídu **HelloWorld** se statickou metodou **main**:

Typ **Unit** se používá místo typu **void**.


Jedná se o typ, jenž má jedinou hodnotu **()**.

```
object HelloWorld {  
  
    def main(args: Array[String]): Unit = {  
        println("Hello, world!")  
    }  
  
}
```

Středníky nejsou třeba,
pokud není více příkazů na jednom řádku
(podrobněji viz §1.2 Newline Characters).

Kompilace a spuštění programu

- Pro sestavování projektů se používá SBT (Scala Build Tool).
- Soubory ***.sbt** (typicky **build.sbt**) a **project/*.scala** určují, jak projekt sestavit.
- Užitečné příkazy:
 - **compile** (kompilace programu).
 - **~compile** (průběžné kompilování programu).
 - **run** (spuštění programu).
 - **test** (spuštění všech testů programu).
 - **testOnly** (spuštění vybraných testů).
 - Př.: **testOnly org.example.*Slow org.example.MyTest1**.
 - **console** (spustí interpret Scaly; třídy projektu budou k dispozici),
 - **reload** (znovu načte soubory určující, jak sestavit projekt).



Cesty jsou relativní vzhledem k adresáři s projektem.

Proměnné

- Proměnné se definují pomocí **val** a **var**:

Typ proměnné **Int** se automaticky odvodí z přiřazované hodnoty **3**.

```
val x = 3
```

```
// x += 3
```

Proměnné definované pomocí **val** jsou neměnné (nejde do nich přiřadit jinou hodnotu).

```
var y = 5
```

```
y += 3
```

```
val l: Long = 7
```

Proměnné můžeme přiřadit typ explicitně pomocí dvojtečky.

Řetězce

- Řetězce se uzavírají do dvojitých uvozovek:

"dvojité uvozovky \" mohou být escapovány \\""

K escapování speciálních znaků se standardně používá zpětné lomítko.

""víceřádkový řetězec můžeme uzavřít do ztrojených dvojitých uvozovek""

Odsazení druhého řádku bude součástí řetězce (druhý řádek bude začínat mezerami).

Tato uvozovka bude součástí řetězce.

""odsazení druhého a třetího
| řádku není součástí řetězce
| ""stripMargin

Pokud chceme mít jednotlivé řádky odsazené, ale nechceme, aby odsazení bylo součástí řetězce použijeme funkci **stripMargin**.

stripMargin uřízne prefix každého řádku tvořený bílými znaky, jenž jsou následovány ořítkem |.

Interpolace řetězců

s umožňuje použití dolaru.

- Standardní interpolátory jsou **s**, **raw** a **f**:

```
val name = "Lucie"
```

```
s"Ahoj $name"
```

```
s"1 + 1 = ${1 + 1}"
```

Interpolátory umožňují do řetězců vkládat hodnoty jiných výrazů pomocí dolaru.

Dolar se escapuje zdvojením.

raw se liší od **r** v Pythonu tím že umožňuje vkládat hodnoty výrazů pomocí **\$**.

Metoda **r** vytvoří regulární výraz z řetězce.

```
val yearRegex = raw"\d{4}".r
```

```
val height = 1.7
```

```
f"$name%s měří $height%2.2f metru"
```


Interpolátor **f** podporuje formátování jako funkce **printf** z C.

Obsah řetězce bude:
Lucie měří 1.70 metru.

if, while, do-while, for

- **if**, **while** a **do-while** se chovají jako v Javě.
 - Scala nemá **break** a **continue**.
- **for** je odlišný od Javy:


until je obyčejná metoda na **Int**tech.



```
for (i <- 1 until 3)
  println(i)
```

```
for {
  i <- 1 to 5
  j <- i to 5
  k = i + j
  if k % 2 == 0
} {
  println(i, j)
}
```

S pomocí **yield** lze vygenerovat i sekvenci.



```
val squares: IndexedSeq[Int] =
  for (i <- 1 to 5)
    yield i * i
```


match – úvod

- **match** můžeme chápat jako lepší **switch** z Javy:

```
val i = 4
```

match vrátí hodnotu, na rozdíl od **switch** z Javy.

```
val desc = i match {  
  case 0 => "nulá"  
  case 1 => "jedna"  
  case j if j > 0 && j < 6 => "mezi 2 a 5 (včetně)"  
  case _ => "něco jiného"  
}
```

Vzor podtržítka odpovídá libovolné hodnotě.

Hodnota proměnné **i** je postupně srovnána se vzory mezi **case** a **=>**.

- Pokud se najde vzor, který odpovídá hodnotě, bude vyhodnocen kód napravo od příslušné **=>**. Takto získaná hodnota pak bude výsledkem celého výrazu **i match { /* ... */ }**.
- Pokud hodnota proměnné **i** žádnému vzoru neodpovídá, bude vyhozena výjimka **MatchError**.

Anglicky se tomuto procesu říká pattern matching.

match – testování typů

- **match** umožňuje testovat typ:

Any je nadtyp všech typů a nadtřída každé třídy (jako **Object** z Javy).

```
val a: Any = /* ... */
```

Pokud má hodnota v **a** typ **Int**, uložíme ji do proměnné **i** typu **Int**.

```
a match {
```

To děláme proto, abychom hodnotu mohli porovnat s **0**. Pro porovnání nejde použít **a**, neboť má typ **Any**, jenž nepodporuje **>**.

```
  case i: Int if i > 0 => println("kladné celé číslo")
  case _: Int         => println("celé číslo")
  case null           => println("null")
  case _              => println("něco jiného")
```

```
}
```

Pozor, generické typy (kromě polí) takto jednoduše testovat nejde. Typové parametry totiž existují pouze během kompilace. Řešením je použít **TypeTag**.

Podrobnosti:

- [Pattern matching on generic type in Scala](#)
- a [Scala: What is a TypeTag and how do I use it?](#)

match – regulární výrazy

Díky metodám `unapplySeq` ze třídy `scala.util.matching.Regex`.

- **match** podporuje regulární výrazy:

\$ je třeba escapovat.

```
val czechPhoneNumber = raw"^((?:\+|00)420)?(\d{9})$$".r
```

```
" +420123456789" match {  
  case czechPhoneNumber(n) => println(s"České číslo $n")  
  case _ => ()  
}
```

Jediná hodnota typu `Unit`.

Do proměnné `n` se uloží číslo bez předvolby.

Obecně: Jeden parametr pro každou capturing skupinu z regulárního výrazu.

match – regulární výrazy (2)

- Žádná capturing skupina:

```
val reasonableEmail = "^.*@seznam.cz$".r
```

```
"foo@seznam.cz" match {  
  case reasonableEmail() => println("Dobrý výběr!")  
  case _ => println("?")  
}
```

Prázdné závorky jsou klíčové, bez nich by si kompilátor myslel, že `reasonableEmail` je proměnná a celý řetězec `"foo@seznam.cz"` by do ní uložil.

match – vnořování vzorů

- Vzory je možné vnořovat:

```
val reasonableEmail = "^.*@seznam.cz$".r
```

Název položky.

Popis položky.

```
val itemWithDesc = "^([^\s]+) - (.*)$".r
```

```
"foo@seznam.cz - nejlepší adresa" match {
```

```
  case itemWithDesc(e @ reasonableEmail(), d) =>  
    s"Email '$e' s popisem '$d'"
```

```
  case itemWithDesc("xyz", _) =>  
    "Položka se jménem xyz"
```

```
  case _ => "?"
```

```
}
```

Název položky srovnáme se vzorem `reasonableEmail()`.

`e @` slouží k uložení emailu do proměnné `e`.

try-catch-finally

- Při chytání výjimek se používá pattern matching:

```
try {  
    //new Array(-1)  
    5 / 0  
}  
catch {  
    case _: NegativeArraySizeException =>  
        println("Chyba při vytváření pole")  
  
    case e: ArithmeticException if e.getMessage == "/ by zero" =>  
        println("Dělení 0")  
}  
finally {  
    /* ... */  
}
```

Místo **equals** se ve Scala používá **==** a místo **hashCode** se používá **##**.

== se chová jinak než **equals** na boxovaných primitivních typech: **(new java.lang.Integer(1) == new java.lang.Long(1))**.

je párový k **==** (např. není vhodné používat **==** s **hashCode**).

Na rozdíl od konstrukce **match** nevyhodí **catch** výjimku **MatchError**, pokud chytaná výjimka neodpovídá žádnému vzoru.

Scala nemá checked exceptions jako Java.

try-with-resources

- Scala nemá analogickou konstrukci jako je **try-with-resources** v Javě nebo **with** v Pythonu.
- Místo toho lze použít knihovnu **scala-arm** (ARM = Automatic Resource Management):

```
import resource.managed
```

```
for {  
  input <- managed(new java.io.FileInputStream("test.txt"))  
  output <- managed(new java.io.FileOutputStream("test2.txt"))  
} {  
  /* Zde použijeme input a output. */  
}
```

for není jen pro tvorbu cyklů.

Metody

- Musí být umístěny uvnitř objektu nebo třídy:

Scala nemá statické metody. Místo nich má tzv. singleton objekty. Singleton objekt je třída, jenž má právě jednu instanci.

```
object math {  
  def square(x: Int) = x * x  
}
```

Pokud se tělo metody skládá pouze z jednoho výrazu, není třeba ho psát do složených závorek.

(Tělo metody **main** na úvodním slajdu by nemuselo být ve složených závorkách).

Metoda vrací hodnotu posledního výrazu (není třeba před tento výraz psát **return**).

Typ návratové hodnoty je třeba explicitně udat pouze u rekurzivních metod. Typy parametrů jsou povinné.

```
case class Person(name: String, age: Int) {  
  
  def greet() = println(s"Ahoj $name")  
  
}
```

Prázdné závorky u metod bez parametrů není třeba psát. Nicméně bývá zvykem je psát u metod s vedlejšími efekty.

Metody (2)

- Metody s jedním parametrem můžeme volat bez tečky s argumentem mimo závorky:

`1.+(1)` ←

Plus je obyčejná metoda a k jejímu zavolání můžeme použít tečkovou notaci.

`1 + 1`

Scala umožňuje pojmenovat metody jako operátory. Například `<*>` nebo `>>=` nebo `|@|` jsou platné názvy metod.

```
object math {  
  def square(x: Int) = x * x  
}
```

`math.square(5)`

`math square 5` ←

`square` můžeme volat stejně jako `+`, tj. bez tečky a bez závorek kolem jediného argumentu.

Metoda apply

- apply** se chová jako **__call__** z Pythonu:

```
object hello {  
  def apply(name: String) = println(s"Ahoj $name")  
}
```

`hello("Lucie")`

Stejně jako `hello.apply("Lucie")`.

`hello.apply("Marie")`

Symbol je jako **String**. Pro porovnávání symbolů stačí referenční rovnost, neboť je zaručeno, že stejné symboly jsou reprezentovány stejnou instancí.

Metodou **name** můžeme symbol převést na **String**.
Například `'foo.name'`.

```
val items: Seq[Symbol] = Seq('foo, 'bar, 'baz)
```

`items(2)`

apply se používá i k indexování polí a jiných kolekcí.

`items.apply(0)`

Scala k indexování nepoužívá hranaté závorky.
Hranaté závorky slouží pro zápis typových parametrů a pro řízení přístupu – například `private[this]`.

Další speciální metody

```
class SpecialMethods {
```

update umožňuje změnit prvek s určitým indexem.

```
def update(idx: Int, s: String): Unit = println(s"update($idx, $s)")
```

```
def foo_=(newValue: String): Unit = println(s"foo_=( $newValue)")
```

Setter vlastnosti **foo**.

```
def foo = "xyz"
```

Getter vlastnosti **foo**. Nejedná se o speciální metodu. Bez getteru však nefunguje syntaktický cukr pro setter, tj. setter by nešlo volat **sp.foo = "Petra"**, ale muselo by se použít klasické **sp.foo_=("Petra")** nebo **sp foo_ = "Petra"**.

```
val sp = new SpecialMethods
```

```
sp(0) = "Lucie"
```

```
sp.update(0, "Lucie")
```

```
sp.foo = "Petra"
```

```
sp.foo_=("Petra")
```

Metody jako argumenty

- Parametrem metody může být metoda:

Parametr **action** je metoda/funkce s parametrem typu **Int** a návratovou hodnotou typu **Unit**.



```
def repeat(times: Int, action: Int => Unit) =  
  for (i <- 0 until times) action(i)
```

```
repeat(3, println)
```

Nevyhodnocené výrazy jako argumenty

- Umožňují vytvářet funkce, které se chovají jako řídicí konstrukce:

=> zajistí, že se argument nevyhodnotí před zavoláním funkce **catchAll**.

```
def catchAll(action: => Unit): Unit =  
  try action  
  catch {  
    case e: Exception =>  
      /* Zde můžeme výjimku e zalogovat. */  
  }
```

```
catchAll {  
  println(s"3 / 2 = ${3 / 2}")  
  println(s"3 / 1 = ${3 / 1}")  
  println(s"3 / 0 = ${3 / 0}")  
}
```

Kompilátor zabalí argument do funkce, a tuto funkci předá **catchAll**.

Každé použití parametru **action**, uvnitř **catchAll** způsobí zavolání této funkce.

Metody s více seznamy parametrů

- Metoda/funkce může mít 0 nebo více seznamů parametrů:

```
def repeat(times: Int)(action: Int => Unit) =  
  for (i <- 0 until times) action(i)
```

```
repeat(3)(println)
```

Seznam argumentů obsahující jediný argument můžeme uzavřít do složených závorek.

```
repeat { 3 } { println }
```

```
repeat(3) { i =>  
  println(i)  
  println(i * i)  
}
```

Anonymní funkce, tzv. lambda funkce, s parametrem *i*.

```
repeat {  
  val x = 3  
  x * x  
} (println)
```

Uzavření argumentu do složených závorek se nečastěji uplatňuje v případě lambda funkcí, jejichž tělo se skládá z více než jednoho výrazu.

Lambda funkce

- Několik možností, jak lambdy definovat:

V Pythonu: `map(lambda i: i * 2, [1, 2, 3])`.

```
Seq(1, 2, 3).map(i => i * 2)  
Seq(1, 2, 3).map((i : Int) => i * 2)
```

```
Seq(1, 2, 3).map { i => i * 2 }  
Seq(1, 2, 3).map { (i : Int) => i * 2 }
```

```
Seq(1, 2, 3).map(_ * 2)  
Seq(1, 2, 3).map { _ * 2 }
```

1. podtržítko zastupuje 1. parametr.
2. podtržítko by zastoupilo 2. parametr.

```
Seq(1, 2, 3).map { case i => i * 2 }
```

K definici funkce lze použít pattern matching.
V tomto případě nejde použít kulaté závorky.

Lambda funkce s více parametry

Co dělá `foldLeft`: `Seq(a, b, c).foldLeft(zero)(f) == f(f(f(zero, a), b), c)`.

Pro odečítání: `Seq(1, 2, 3).foldLeft(0)(_ - _) == ((0 - 1) - 2) - 3`.

Výraz `((0 - 1) - 2) - 3` je uzávorkovaný doleva.

Naopak `foldRight` by výraz uzávorkovalo doprava `1 - (2 - (3 - 0))`.

```
Seq(1, 2, 3).foldLeft(0)((i, j) => i + j)
Seq(1, 2, 3).foldLeft(0) { (i, j) => i + j }
```

```
Seq(1, 2, 3).foldLeft(0)((i: Int, j: Int) => i + j)
Seq(1, 2, 3).foldLeft(0) { (i: Int, j) => i + j }
```

```
Seq(1, 2, 3).foldLeft(0)(_ + _)
Seq(1, 2, 3).foldLeft(0) { _ + _ }
```

Některé parametry mohou mít typy, jiné nemusí.

```
Seq(1, 2, 3).foldLeft(0) { case (i, j) => i + j }
```


Lambda funkce a pattern matching

- Lze definovat lambdu s více alternativami:

```
Seq(5, "ahoj", Map("jedna" -> 1, "dvě" -> 2)).map {
```

```
  case x: Int => x
```

```
  case s: String => s.length
```

```
  case m: Map[_ , _] => m.size
```

```
  case _ => -1
```

```
}
```

Lambda počítá velikost objektů v sekvenci.

Typové parametry nejde pomocí pattern matchingu testovat.

Pro hodnoty jiných typů než **Int**, **String** a **Map** (nechceme vyhazovat **MatchError**).

Pokud chceme hodnoty jiných typů přeskočit, lze použít funkci **collect** místo **map** a odstranit poslední alternativu. Výsledkem bude parciální funkce.

n-tice

Standardně nejde n-ticemi iterovat nebo je indexovat (jako např. v Pythonu).

Oboje lze však doimplementovat – viz například knihovna [shapeless](#).

- n-tice nejsou kolekce.

```
val quadruple = (1, "a", true, 77L)
```

Vytvoření čtveřice.

```
quadruple._2 == "a"
```

Prvky n-tic lze získat pomocí metod `_1`, `_2`, `_3`,

Obecně: `n`-tý prvek lze získat pomocí metody `_n`.

```
val pair = 1 -> "a"
```

Dvojice lze vytvářet pomocí metody `->`. Zápis `1 -> "a"` je totožný se zápisem `(1, "a")`.

Zápis se šipkou je přehlednější uvnitř závorek.

```
val (x, _, _, y) = quadruple
```

n-tice můžeme rozebrat pomocí pattern matchingu. Pattern matching lze použít i při deklaraci proměnných.

```
quadruple match {  
  case (x, _, _, y) => /* ... */  
}
```

Kolekce

- Scala má bohatou knihovnu kolecí.
 - Více viz [dokumentace](#).
- Některé kolekce mají strukturální rovnost:

```
List("a", "b", "c") == List("a", "b", new String("c"))
```

```
Map(1 -> "x", 2 -> "y") == Map(2 -> "y", 1 -> "x")
```



Obě rovnosti platí.

Nejčastěji používané kolekce

- Sekvence (zachovávají pořadí):
 - Neměnné: **Seq**, **Vector**, **List**, **Queue**
 - Měnitelné: **Seq**, **Array**, **ArrayBuffer**, **ListBuffer**, **Queue**, **Stack**
- Množiny:
 - Neměnné i měnitelné: **Set**, **SortedSet**
- Mapy:
 - Neměnné i měnitelné: **Map**, **SortedMap**

Neměnné kolekce jsou v balíčku **scala.collection.immutable**.
Měnitelné kolekce jsou v balíčku **scala.collection.mutable**.

Seq

Vytvoří neměnnou sekvenci s prvky 1, 2, 3, 4.

- Neměnná sekvence **Seq** je implementována jako **List**.
- Měnitelná sekvence **Seq** je implementována jako **ArrayBuffer**.

val xs = Seq(1, 2, 3, 4)

xs(1) ← Indexování. Indexovatelné kolekce se chovají jako funkce.

xs :+ 5 ← Operátor :+ slouží pro přidávání na konec sekvencí. :+ vrátí novou kolekci, původní kolekce zůstane beze změn.

0 +: xs ← Operátor +: slouží pro přidávání na začátek kolekce. Volání 0 +: xs se chová jako xs.+:(0). +: rovněž vrací novou kolekci a původní kolekci nemění.

Pokud název metody končí dvojtečkou a metoda je volána bez tečky, pak je volána na svém druhém argumentu. Například a ::: b je totéž jako b.:::(a). Operátory končící dvojtečkou jsou asociativní zprava doleva. Například zápis a ::: b ::: c ::: d je ekvivalentní se zápisem a ::: (b ::: (c ::: d)).

xs ++ Seq(5, 6) ← Zřetězení dvou kolekcí. Původní kolekce zůstanou nezměněny.

xs + "!" ← Převede kolekci na řetězec a na jeho konec přidá vykřičník. Vrací řetězec: List(1, 2, 3, 4)!.

Sekvence – pattern matching

- Idea představená na regulárních výrazech funguje i se sekvencemi **Seq**, **List**, **Queue**, **Array** a dalšími:

```
Seq(1, 2, 3, 4) match {  
  case Seq() => "prázdná sekvence"  
  case Seq(x) => s"jediný: $x"  
  case Seq(x, x2, xs @ _*) => s"první: $x; druhý $x2, zbylé $xs"  
}
```

Vzor `_*` odpovídá 0 nebo více prvkům. Poslední alternativa tedy odpovídá sekvencím se dvěma a více prvky.

List

- Jednosměrný spojový seznam (neměnný).

List('A', 'B', 'C', 'D')

'A' :: 'B' :: 'C' :: 'D' :: *Nil*

'A' :: 'B' :: 'C' :: *List*('D')

'A' :: 'B' :: *List*('C', 'D')

Různé způsoby konstrukce spojového seznamu.

:: přidá prvek na začátek spojového seznamu. Volání $x :: xs$ zkonstruuje nový spojový seznam přidáním prvku x na začátek seznamu xs . Toto volání nijak nemění seznam xs , se seznamem xs lze dále pracovat.

Nil je prázdný spojový seznam. Též lze psát *List()*.

U spojových seznamů $++$ volá :: (pokud má být výsledkem spojový seznam).

List(1, 2) ::: *List*(3, 4) ::: *List*(5, 6)

::: zřetězí spojové seznamy. Na rozdíl od $++$ je asociativní zprava doleva (lepší časová složitost při zřetězování více seznamů).

List – časová složitost

- Přidání na konec seznamu xs pomocí $:+$ vykoná $O(xs.length)$ kroků.
- Indexování: Nalezení i -tého prvku v seznamu vykoná $O(i)$ kroků.
- Zřetězení dvou seznamů $xs ::: ys$ vykoná $O(xs.length)$ kroků.
 - Na začátek ys se postupně pomocí $::$ přidá poslední prvek xs , předposlední prvek xs , ..., první prvek xs .

Při zřetězování více seznamů je lepší uzávorkování doprava než uzávorkování doleva.

Například pro seznamy xs , ys , zs délek 5, 4, 3 bude třeba na zřetězení $(xs ::: ys) ::: zs$ cca 14 kroků: Na zřetězení xs a ys je třeba cca 5 kroků a dostaneme seznam $(xs ::: ys)$ s 9 prvky. Na zřetězení $(xs ::: ys) ::: zs$ bude tedy třeba cca 9 kroků. Dohromady $5 + 9 = 14$ kroků. Naopak pro zřetězení $xs ::: (ys ::: zs)$ bude třeba cca $4 + 5 = 9$ kroků.

List – pattern matching

- Pattern matching jde používat i s `::`

Zřetězení dvou seznamů. Definice rekurzí podle prvního seznamu.

Pokud bude seznam `a` dlouhý, například `List.range(1, 10000)`, dojde k přetečení zásobníku.

`def append[T](a: List[T], b: List[T]): List[T] = a match {`

Místo `Nil` by šlo použít vzor `List()`.

`case Nil => b`

`case x :: xs => x :: append(xs, b)`

Šlo by použít vzor `List(x, xs @ _*)`.

`}`

Vector

- Neměnná pole.
- Zřetězení pomocí `++` má konstantní časovou složitost.
- Indexování má rovněž konstantní časovou složitost.

ArrayBuffer

- Jako seznam z Pythonu nebo **ArrayList** z Javy.
 - Lze tam přidávat a odebírat prvky.
- Oproti neměnným kolekcím jsou k dispozici: **+=**, **++=**, **-=**, **--=**.
 - **:+**, **+:**, **++**, **-**, **--** stále nemění kolekci.

ListBuffer

- Jako **StringBuffer** z Javy, ale pro **List**.
 - Umožňuje rychle konstruovat **List** přidáváním prvků na konec.

```
val b = ListBuffer[Int]()
```

```
b += 1
```

```
b += 2
```

```
b ++= Set(3, 18) & Set(3)
```

```
val x = b.toList // List(1, 2, 3)
```

```
b += 4
```

```
val y = b.toList // List(1, 2, 3, 4)
```

Map

- Jako **Map** z Javy nebo slovník z Pythonu.

```
val age = Map("Zdeněk" -> 20, "Radek" -> 6, "Pavel" -> 15)
```

```
age("Radek") // 6
```

Mapu lze zkonstruovat i ze sekvence dvojic:
`Seq("Zdeněk" -> 20, "Radek" -> 6, "Pavel" -> 15).toMap.`

```
age.getOrElse("Rudolf", 17) // 17
```

```
List("Radek", "Pavel").map(age) // List(6, 15)
```

Nejen sekvence se chovají jako funkce, i **Map** se chová jako funkce.

Použití kolekcí z Javy

Import zajistí, že na kolekcích ze Scaly bude k dispozici metoda **asJava** a na kolekcích z Javy bude metoda **asScala**.

Dokumentace traitů **DecorateAsJava** resp. **DecorateAsScala** říká, jakou kolekci **asJava** resp. **asScala** vrátí.



```
import scala.collection.convert.decorateAll._
```

```
val xs = new java.util.ArrayList[Int]()  
xs.add(1)  
xs.asScala
```

```
List(1).asJava
```

None místo null

Díky `Option[T]` typy v programu jasně říkají, kde hodnota může chybět a kde ne.

- Ve Scala není zvykem používat `null`.
- Může-li hodnota typu `T` chybět, použije se typ `Option[T]`:
 - `Option[Int]` má například hodnoty `Some(1)`, `Some(2)`, `Some(55)`, `None`.

Hodnoty typu `T` jsou zabaleny uvnitř `Some`.

`None` značí, že hodnota není k dispozici.

- Typ `Option[T]` podporuje pattern matching:

```
Map("Zdeněk" -> 20, "Radek" -> 6).get("Zdeněk") match {  
  case None => "Věk Zdeňka neznáme."  
  case Some(age) => s"Zdeňkovi je $age."  
}
```

Konec první části

- Otázky?
- Příště:
 - Třídy (speciálně **case class** a pattern matching).
 - Objekty.
 - Traits.
 - Imports.
 - Implicitní konverze.
 - Implicitní definice, implicitní argumenty, návrhový vzor „typová třída“.