Functional Programming in Scala

A gentle intro

Scala Training, 2016

Outline

Course Contents

- Intro
- OOP Basics & Testing
- Functional Programming Basics & Collections
- Comprehensions, "operators"
- Traits and inheritance
- Pattern matching
- Idiomatic types: *Try[T]*, *Option[T]*
- Advanced FP
- Type system magic
- Implicits & behind the scenes

Why Scala

- Java is popular and all...

Why Scala

- Java is popular and all...
- runs on Billions (b) of computers
  - regular PCs
  - mobile
  - servers
  - ...

Why Scala

- Java is popular and all...
- runs on Billions (b) of computers
  - regular PCs
  - mobile
  - servers
  - ...
- "write once, run everywhere"

Why Scala

- Java is popular and all...
- runs on Billions (b) of computers
  - regular PCs
  - mobile
  - servers
  - ...
- "write once, run everywhere"
- relatively easy to pick up

Why Scala

- Java is popular and all...
- runs on Billions (b) of computers
  - regular PCs
  - mobile
  - servers
  - ...
- "write once, run everywhere"
- relatively easy to pick up
- statically typed, guards against exceptions/errors

Why Scala

- Java is popular and all...
- runs on Billions (b) of computers
  - regular PCs
  - mobile
  - servers
  - ...
- "write once, run everywhere"
- relatively easy to pick up
- statically typed, guards against exceptions/errors

Q: Statically typed?

Why Scala (2)

BUT:

- a gazillion lines of boilerplate
  - getMyAwesomeField(), generics, ...

Why Scala (2)

BUT:

- a gazillion lines of boilerplate
  - getMyAwesomeField(), generics, ...
- heavyweight frameworks
  - JEE, anyone?

Why Scala (2)

BUT:

- a gazillion lines of boilerplate
  - getMyAwesomeField(), generics, ...
- heavyweight frameworks
  - JEE, anyone?
- guards against exceptions/errors are good...

Why Scala (2)

BUT:

- a gazillion lines of boilerplate
  - getMyAwesomeField(), generics, ...
- heavyweight frameworks
  - JEE, anyone?
- guards against exceptions/errors are good...
- ...but being paranoid is not

Why Scala (2)

BUT:

- a gazillion lines of boilerplate
  - getMyAwesomeField(), generics, ...
- heavyweight frameworks
  - JEE, anyone?
- guards against exceptions/errors are good...
- ...but being paranoid is not

Q: Checked vs unchecked exceptions?

Why Scala (3)

So why Scala:

- Lightweight syntax, do more in less time
- **Statically typed, with type inference**
- Functional *and* OO
- Interoperable with Java
  - compiles to bytecode (more on that later)

Why Scala (3)

So why Scala:

- Lightweight syntax, do more in less time
- **Statically typed, with type inference**
- Functional *and* OO
- Interoperable with Java
  - compiles to bytecode (more on that later)

Q:
What's in a functional language?

Why Scala (3)

So why Scala:

- Lightweight syntax, do more in less time
- **Statically typed, with type inference**
- Functional *and* OO
- Interoperable with Java
  - compiles to bytecode (more on that later)

Q:
What's in a functional language?
Bytecode?

Short history

- 90s - making Java better
- 2001 - making a better Java - Scala was born
- 2003 - first experimental release
- 2005 - Scala 2.0
- now - Scala 2.11, Scala 2.12-M4

The REPL

- REPL (Read Eval Print Loop) = an interactive shell for Scala
- can be started from the command line
- compiles and evaluates Scala code immediately
- very helpful for experiment-driven development

```
$ scala
Welcome to Scala version 2.11.7 (Java HotSpot(TM)
    64-Bit Server VM, Java 1.8.0_45).
Type in expressions to have them evaluated.
Type :help for more information.

scala> "Hello, World"
res0: String = Hello, World
```

Declaring variables

```
val x = 2
```

Declaring variables

```
val x = 2
```

At this point, Scala knows (@ compile time):

- x is an `Int` and
- has the value 2

but...

```
x = 3
```

Declaring variables

```
val x = 2
```

At this point, Scala knows (@ compile time):

- x is an `Int` and
- has the value 2

but...

```
x = 3
```

- no!
- `val`s cannot be reassigned => immutable values

Declaring variables (2)

Reassignable variables: var

```
var x = 2
//...
x = 3 // allowed!
```

- allows imperative programming
- mutable variables
- immutable: `val`

Declaring variables (2)

Reassignable variables: var

```
var x = 2
//...
x = 3 // allowed!
```

- allows imperative programming
- mutable variables
- immutable: `val`

Q:
What's imperative programming?

Declaring variables (2)

Reassignable variables: var

```
var x = 2
//...
x = 3 // allowed!
```

- allows imperative programming
- mutable variables
- immutable: `val`

Q:
What's imperative programming?
Mutable? Immutable? Immutable object?

Variable types and type inference, basic

```
scala> val message = "Hello, world!"
message: String = Hello, world!
```

The compiler infers the type in the right-hand side!

Variable types and type inference, basic

```
scala> val message = "Hello, world!"
message: String = Hello, world!
```

The compiler infers the type in the right-hand side!

Other examples:

- inferring the result type of an operation

```
scala> val x = 2
scala> val y = x + "items"
y: String = 2 items
```

Variable types and type inference, basic (2)

- inferring a function type

```
scala> def succ(x: Int)= x + 1
succ: (x: Int)Int
```

Variable types and type inference, basic (2)

- inferring a function type

```
scala> def succ(x: Int)= x + 1
succ: (x: Int)Int
```

The type inferrer is *smart*

- type unification
- co/contra-variance detection

Variable types and type inference, basic (2)

- inferring a function type

```
scala> def succ(x: Int)= x + 1
succ: (x: Int)Int
```

The type inferrer is *smart*

- type unification
- co/contra-variance detection
- ... and much more ...

Variable types and type inference, basic (2)

- inferring a function type

```
scala> def succ(x: Int)= x + 1
succ: (x: Int)Int
```

The type inferrer is *smart*

- type unification
- co/contra-variance detection
- ... and much more ...
- ... later

Variable types and type inference, basic (3)

- you can also *specify* the type...
  - polymorphically, as in Java:

```
scala> val x: Object = "com"
x: Object = com
```

Variable types and type inference, basic (3)

- you can also *specify* the type...
  - polymorphically, as in Java:

```
scala> val x: Object = "com"
x: Object = com
```

- ... but it has to be correct

```
scala> val y: Int = "a string"
error: type mismatch
```

Variable types and type inference, basic (3)

- you can also *specify* the type...
  - polymorphically, as in Java:

```
scala> val x: Object = "com"
x: Object = com
```

- ... but it has to be correct

```
scala> val y: Int = "a string"
error: type mismatch
```

- tip: when using the above syntax, always put the ':' after the field name and keep a white space before the type

Expressions

- **Q:** instructions vs expressions?

Expressions

- **Q:** instructions vs expressions?
  - instructions get executed
  - expressions are evaluated

Expressions

- **Q:** instructions vs expressions?
  - instructions get executed
  - expressions are evaluated
- Most constructs in Scala are expressions
  - if-statements
  - for comprehensions
  - try/catches
  - blocks of code
  - ...

Expressions

- **Q:** instructions vs expressions?
  - instructions get executed
  - expressions are evaluated
- Most constructs in Scala are expressions
  - if-statements
  - for comprehensions
  - try/catches
  - blocks of code
  - ...
- There are also imperative elements (instructions)
  - while-loops
  - var reassignments

Expressions - examples

- basic computations

```
scala> val x = 1 + 2
x: Int = 3
```

Expressions - examples

- basic computations

```
scala> val x = 1 + 2
x: Int = 3
```

- if-statements
  - the "else" branch is not mandatory

```
scala> val y = if (x > 3) 3 else 5
y: Int = 5
```

Expressions - examples

- basic computations

```
scala> val x = 1 + 2
x: Int = 3
```

- if-statements
  - the "else" branch is not mandatory

```
scala> val y = if (x > 3) 3 else 5
y: Int = 5
```

- function callbacks

```
scala> val z = succ(y)
z: Int = 6
```

Expressions - examples (2)

Blocks of code are expressions

- value = evaluation of the last expression

```
def f(x: Int) = {
  val y = x + 2
  succ(y) // this is the "value" of this block
}
```

Expressions - examples (2)

Blocks of code are expressions

- value = evaluation of the last expression

```
def f(x: Int) = {
  val y = x + 2
  succ(y) // this is the "value" of this block
}
```

```
def g(x: Int) = {
  if (x > 1) 3
  "hello"
}
```

Q: what is g(2)?

Expressions - examples (3)

Multi-line expressions

- as in Java
- advice: leave dangling operators

```
val x = 1 +
  2 // x: Int = 3
```

Expressions - examples (3)

Multi-line expressions

- as in Java
- advice: leave dangling operators

```
val x = 1 +
  2 // x: Int = 3
```

Function values (lambdas)

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

Functions

```
def add(x: Int, y: Int): Int = x + y
```

- the last expression in the function body is the return expression
- avoid using `return` explicitly
- functions are called just like in Java:

```
add(1, 2) // the call is made within the same scope
obj.add(1, 2) // the call is made on an object
```

Functions(2)

- function arguments are evaluated from left to right
- the function arguments are reduced to values
- there are two evaluation strategies:
  - call-by-name
  - call-by-value
- call-by-value evaluates every function argument *exactly once*
- call-by-name leaves params *unevaluated until used*

Functions (3) - recursion

```
scala> def factorial(n: Int): Int =
  | if (n <= 0) 1 else n * factorial(n-1)
factorial: (n: Int)Int
```

- Q: stack-recursion? Tail-recursion?

Functions (3) - recursion

```
scala> def factorial(n: Int): Int =
  | if (n <= 0) 1 else n * factorial(n-1)
factorial: (n: Int)Int
```

- Q: stack-recursion? Tail-recursion?
- stack-recursion can cause stack overflow - be careful
- a tail recursive function can be seen as an iterative process
- note: tail-calls
- `@tailrec` as a compiler indication - we'll use this later

Functions (4) - higher order functions

Functions are first-class citizens:

- functions can be passed as parameters
- functions can be returned as a result
- anonymous functions (*lambdas*)

Functions (4) - higher order functions

Functions are first-class citizens:

- functions can be passed as parameters
- functions can be returned as a result
- anonymous functions (*lambdas*)

Functions can return other functions or receive functions as params

Functions (4) - higher order functions

Functions are first-class citizens:

- functions can be passed as parameters
- functions can be returned as a result
- anonymous functions (*lambdas*)

Functions can return other functions or receive functions as params

- hofs = higher order functions

Functions (4) - higher order functions

Functions are first-class citizens:

- functions can be passed as parameters
- functions can be returned as a result
- anonymous functions (*lambdas*)

Functions can return other functions or receive functions as params

- hofs = higher order functions
- critical to functional programming!

Functions (4) - higher order functions

Functions are first-class citizens:

- functions can be passed as parameters
- functions can be returned as a result
- anonymous functions (*lambdas*)

Functions can return other functions or receive functions as params

- hofs = higher order functions
- critical to functional programming!

```
def increment(x: Int): Int = x + 1

// the map method is a hof
List(1,2,3).map(increment) // List(2,3,4)
```

Functions (5) - currying

```
def sum(f: Int => Int): (Int, Int) => Int = {
  // notice helper function definition below
  def sumF(a: Int, b: Int): Int =
    if (a > b) 0 else f(a) + sumF(a + 1, b)

  sumF // return value
}
```

Q1: What does `sumF` do?

Functions (5) - currying

```
def sum(f: Int => Int): (Int, Int) => Int = {
  // notice helper function definition below
  def sumF(a: Int, b: Int): Int =
    if (a > b) 0 else f(a) + sumF(a + 1, b)

  sumF // return value
}
```

Q1: What does `sumF` do?

```
val sumFactorials = sum(factorial)
```

Q2: What *type* is `sumFactorials`?

Functions (5) - currying

```
def sum(f: Int => Int): (Int, Int) => Int = {
  // notice helper function definition below
  def sumF(a: Int, b: Int): Int =
    if (a > b) 0 else f(a) + sumF(a + 1, b)

  sumF // return value
}
```

Q1: What does `sumF` do?

```
val sumFactorials = sum(factorial)
```

Q2: What *type* is `sumFactorials`?

Q3: How do you call `sumFactorials`?

Functions (5) - currying

```
def sum(f: Int => Int): (Int, Int) => Int = {
  // notice helper function definition below
  def sumF(a: Int, b: Int): Int =
    if (a > b) 0 else f(a) + sumF(a + 1, b)

  sumF // return value
}
```

Q1: What does `sumF` do?

```
val sumFactorials = sum(factorial)
```

Q2: What *type* is `sumFactorials`?

Q3: How do you call `sumFactorials`?

```
sumFactorials(1,4)
```

Functions (5) - currying

```
def sum(f: Int => Int): (Int, Int) => Int = {
  // notice helper function definition below
  def sumF(a: Int, b: Int): Int =
    if (a > b) 0 else f(a) + sumF(a + 1, b)

  sumF // return value
}
```

Q1: What does `sumF` do?

```
val sumFactorials = sum(factorial)
```

Q2: What *type* is `sumFactorials`?

Q3: How do you call `sumFactorials`?

```
sumFactorials(1,4)
```

Final: get the same result *without* using a `val`!

Functions (5) - currying

```
def sum(f: Int => Int): (Int, Int) => Int = {
  // notice helper function definition below
  def sumF(a: Int, b: Int): Int =
    if (a > b) 0 else f(a) + sumF(a + 1, b)

  sumF // return value
}
```

Q1: What does `sumF` do?

```
val sumFactorials = sum(factorial)
```

Q2: What *type* is `sumFactorials`?

Q3: How do you call `sumFactorials`?

```
sumFactorials(1,4)
```

Final: get the same result *without* using a `val`!

```
sum(factorial)(1,4) // hello, curry!
```

Lightweight syntax

- no need for semicolons
- single-line expressions don't need curly braces
- no need for return statements
  - we work with expressions
- no need for type specification on variables
  - type inference!
- looooots of sugar (more on that later)
  - functions as operators
  - singletons
  - for-comprehensions
  - pattern matching
  - implicit
  - many many more...

Hello, Scala

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

- think Java-style, analogous to
  public static void main(String[] av)
- why no *static*?
- what else is different?

Practice!

http://workshop.rosedu.org/2016/sesiuni/scala/lab1