An Introduction to Scala for Spark programming

Nastaran Fatemi

Slides of this course are prepared based on the book “Programming in Scala” and its presentations by Martin Odersky
What’s Scala

• Scala is a statically typed, object-oriented programming language that blends imperative and functional programming styles.

• It is designed to integrate easily with applications that run on modern virtual machines, primarily the Java virtual machine (JVM).

• Scala was developed starting in 2003 by Martin Odersky's group at EPFL.

• It is used for big-iron projects in industry.
Some adoption vectors:

- Web platforms
- Trading platforms
- Financial modeling
- Simulation
- Big Data Analytics

Fast to first product, scalable afterwards
Scala is a Unifier

Agile, with lightweight syntax

Object-Oriented → Scala → Functional

Parallel

Sequential

Safe and performant, with strong static typing
“If I were to pick a language to use today other than Java, it would be Scala.”
- James Gosling, creator of Java

“Scala, it must be stated, is the current heir apparent to the Java throne. No other language on the JVM seems as capable of being a "replacement for Java" as Scala, and the momentum behind Scala is now unquestionable. While Scala is not a dynamic language, it has many of the characteristics of popular dynamic languages, through its rich and flexible type system, its sparse and clean syntax, and its marriage of functional and object paradigms.”

- Charles Nutter, creator of JRuby

“I can honestly say if someone had shown me the Programming in Scala book by Martin Odersky, Lex Spoon & Bill Venners back in 2003 I'd probably have never created Groovy.”

- James Strachan, creator of Groovy.
What makes Scala scalable?

• Many factors: strong typing, inference, little boilerplate,…
• But mainly, its tight integration of functional and object-oriented programming

**Functional programming:**

Makes it easy to build interesting things from simple parts, using

- immutable datatypes
- closures and higher-order functions,
- generic typing

**Object-oriented programming:**

Makes it easy to adapt and extend complex systems, using

- subtyping and inheritance,
- dynamic configurations,
- classes as partial abstractions.
The Philosophy behind Scala

Put productivity and creativity back in the hands of developers.

“Joy is underrated as a metric for a language’s potential success in a development organization.” a3lx@twitter

- address professional developers
- trust them & value their expertise
- (don’t tell them how they should do their job)
Scala Basics
Overview

In this course you will learn about

- Scala REPL
- Variable and method definitions
- Scala syntax
- Scala’s type hierarchy
- How to construct functional objects
- Collections: sequences
- Function values and higher-order functions
- For loops and for expressions
- Collections: sets and maps
Download and install Scala

• Installation :
  - Go to http://www.scala-lang.org/downloads and follow the directions for your platform.
  - Once you download an archive file, create a directory wherever you prefer and unzip (or untar, etc.) the archive file in that empty directory.
  - Among the subdirectories created will be the bin directory that contains the Scala executables, including the compiler and interpreter.
  - To make Scala convenient to use, add the full pathname of the bin directory to your PATH environment variable.

You can also use Scala via plug-ins for Eclipse and IntelliJ downloadable from the same link given above.
Using the Scala interpreter

The easiest way to get started with Scala is by using the Scala interpreter, which is an interactive “shell” for writing Scala expressions and programs.

The interactive shell for Scala is simply called scala.

```
scala> 1 + 2
res0: Int = 3

scala> res0 * 2
res1: Int = 6

scala> println("Hello, world!")
Hello, world!
```
Variables

Two forms:

val immutable variable
var reassignable variable

scala> val msg = "hello world!"
msg: String = hello world!

scala> var greeting = "hi!"
greeting: String = hi!

scala> greeting = "hi there!"
greeting: String = hi there!
Expressions

Languages like C and Java distinguish between *expressions* which return a result and *statements* which don’t.

Scala does not: *every* statement is an expression that returns a value.

```
scala> if (msg contains 'a') msg else "no a"
res3: String = no a

scala> try { msg } finally { println("done") }
done
res4: String = hello world!

scala> { val x = 2; x * x }
res5: Int = 4
```
Functions

- Basic form:
  ```python
def max(x: Int, y: Int): Int = {
  if (x < y) y else x
}
```

- Result type is required only for recursive functions.
- Right hand side may be simple expression without `{ ... }`
- Short form:
  ```python
def max(x: Int, y: Int) = 
  if (x < y) y else x
```

  ```python
def square(x: Double) = x * x
```
Recursion

• Recursive functions need an explicit return type

```python
def power(x: Double, n: Int): Double =
    if (n == 0) 1.0
    else if (n % 2 == 0) square(power(x, n / 2))
    else x * power(x, n - 1)
```

```python
def findIndex(str: String, chr: Char, from: Int): Int =
    if (str.charAt(from) == chr) from
    else findIndex(str, chr, from + 1)
```
The Unit Type

• Question: What’s the type of the expression `println("hi")`?
• Try it out!

```scala
scala> val x = println("hi")
hi
x: Unit = ()
```

• Scala uses `Unit` as the type of expressions that are executed only for their side-effects.
• `Unit` has a value, written `()`.
• `Unit` corresponds roughly to `void` in Java.
Procedures

• Procedures are functions that return Unit.

```scala
scala> def sayHi(): Unit = println("hi!")
sayHi: ()Unit
```

• They have an alternative syntax, where the parameter list is immediately followed by a block, without return type or =.

```scala
scala> def sayHo { println("ho!") }
sayHo: Unit
```
Scala cheat sheet (1): Definitions

**Scala method definitions:**

```scala
def fun(x: Int): Int = {
  result
}
```

or

```scala
def fun(x: Int) = result
```

```scala
def fun = result
```

**Scala variable definitions:**

```scala
var x: Int = expression
val x: String = expression
```

or

```scala
var x = expression
val x = expression
```

**Java method definition:**

```java
int fun(int x) {
  return result;
}
```

(no parameterless methods)

**Java variable definitions:**

```java
int x = expression
final String x = expression
```
Scala method calls:

```
obj.meth(arg)
or obj meth arg
```

Scala choice expressions:

```
if (cond) expr1 else expr2
```

```
expr match {
  case pat₁ => expr₁
  ....
  case patₙ => exprₙ
}
```

Java method call:

```
obj.meth(arg)
(no operator overloading)
```

Java choice expressions, stats:

```
cond ? expr₁ : expr₂
```

```
if (cond) return expr₁;
else return expr₂;
```

```
switch (expr) {
  case pat₁ : return expr₁;
  ...
  case patₙ : return exprₙ;
} // statement only
```
Scala cheat sheet (3): Objects and Classes

Scala Class and Object

```scala
class Sample(x: Int) {
    def instMeth(y: Int) = x + y
}

object Sample {
    def staticMeth(x: Int, y: Int) = x * y
}
```

Java Class with static

```java
class Sample {
    final int x;
    Sample(int x) {
        this.x = x
    }

    int instMeth(int y) {
        return x + y;
    }

    static int staticMeth(int x, int y) {
        return x * y;
    }
}
```
Scala cheat sheet (4): Traits

Scala Trait

```scala
trait T {
  def absMeth(x: String): String
  def concreteMeth(x: String) = x + field
  var field = "!
}
```

Scala mixin composition:

```scala```

Java Interface

```java
interface T {
  String absMeth(String x)
  // (no concrete methods)
  // (no fields)
}
```

Java extension + implementation:

```java
class C extends Super with T
```
Scala cheat sheet (5): Packages and Imports

Scala Package Clause

```scala
package org.project.module

or package org.project
package module
```

Scala Import

```scala
import collection.mutable.Map
import collection.mutable._
import collection.mutable.{
  Map => mMap
}
```

Java Package Clause

```java
package org.project.module;
```

Java import

```java
import collection.mutable.Map;
import collection.mutable.*;

(no import renaming)
```
Functional Objects

<table>
<thead>
<tr>
<th>Conventional Wisdom</th>
<th>Scala</th>
</tr>
</thead>
<tbody>
<tr>
<td>Objects have</td>
<td>identity (definable)</td>
</tr>
<tr>
<td></td>
<td>state (maybe)</td>
</tr>
<tr>
<td></td>
<td>behavior</td>
</tr>
</tbody>
</table>

Objects without state are called *immutable*. Such objects are ubiquitous: strings, numbers, polynomials, time-series functions, financial contracts, ...
Example: Rational Numbers

• Writing a class for rational numbers.
The Naked Class

class Rational (n: Int, d: Int)
extends AnyRef {
}

Superclass (optional)

Class parameters (optional)

Class body (optional)
Adding toString

- So far, new Rational displays strangely (something like Rational@12ab3)
- You can change the way objects display by overriding the toString method, which is defined in Rational’s superclass java.lang.Object, a.k.a AnyRef.

```
override def toString = n+"/"+d
```

*override* required, because concrete method in AnyRef is replaced
Checking Preconditions

- The denominator of a rational number should be greater than zero.
- It’s best to check this when Rationals are constructed, thereby establishing a useful class invariant:

  \[
  \text{require}(d > 0)
  \]

- Require is a pre-defined method in Predef. It throws an IllegalArgumentException if the condition is false.
- You can use a two argument version:

  \[
  \text{require}(d > 0, "denominator of Rational must be greater than zero")
  \]
Auxiliary Constructors

• Unlike Java, Scala classes take parameters directly.

```scala
class Rational(n: Int, d: Int)
```

defines implicitly a two parameter constructor (called the primary constructor).

• Sometimes, you want more than one constructor.
• For instance for class Rational, you might want a constructor that takes only one nominator parameter and assumes 1 for the denominator.
• You can do this with an auxiliary constructor:

```scala
def this(n: Int) = this(n, 1)
```

• Every auxiliary constructor must call a preceding constructor as its first action.
Defining Fields

• Fields of a class are defined like variables:
  - `val` for immutable fields
  - `var` for reassignable fields

• Example:

  ```scala
  private val g = gcd(n, d)
  val numer = n / g
  val denom = d / g
  ```
Hiding fields and methods

- Class members are hidden to the outside with `private`.
- There’s also `protected` (as in Java).
- There’s no “public” modifier – all members without `private` or `protected` modifiers are public.
Binary Operations

• Here’s a method to add two rational numbers.
  ```scala
  def add (that: Rational) = new Rational(
    this.numer * that.denom + that.numer * this.denom,
    this.denom * that.denom)
  ```

• Notes:
  – Return type can be omitted, and is inferred.
  – Body can be a single expression or a block `{ ... }
  – If it’s a block `{ ... }`, the last expression is returned.
  – With these conventions, explicit returns are rare in Scala.
Mathematical Notation

• With add defined, we can write
  
  ```scala
  val r = new Rational(1, 3)
  r.add(r)
  ```

• But why is it `add` for Rationals, but `+` for Ints and Floats?

• In Scala, there’s no need for this, because of two conventions:
  1. `+` is a legal identifier name just like `add`.
  2. A binary operation `a op b` is the same as a method call `a.op(b)`

• So you can alternatively define:
  
  ```scala
  def + (that: Rational) = new Rational(...)  
  ```
Forms of identifiers in Scala

Scala knows four forms of identifiers.

• **Alphanumeric:** A letter, followed by a sequence of letters or digits.
  – _ counts as a letter (but single _ is reserved)
  – $ is reserved for system use.

• **Operator:** One or more symbolic characters such as +, -, %, #, !

• **Mixed:** alpha_

• **Quoted:** any character sequence in backquotes: `yield`
Question: When you write
\[ x + y \times z \]
How does the compiler “know” that \(*\) binds stronger than \(+\)?

Scala arranges precedence of identifiers according to their first letter.

(all other special characters) \(^{\text{highest}}\)
- \(*\) / \%
- + -
- :
- = !
- < >
- &
- ^

| (all letters) \(^{\text{lowest}}\)
| (all assignment operators such as +=, -=, ...)
Ass ociativity

In Scala, every operation is a method call.

\[
an + b \quad \text{is exactly the same as} \quad a.+(b)
\]

So operators resolve to method calls of their left operand.

There's one exception to this rule:
If the operator ends with a "::", it resolves to a method call of its right operand.

So
\[
\text{x :: xs} \quad \text{is the same as} \quad \text{xs.::(x)}
\]

This also extends to associativity.

\[
a + b + c \quad \text{is} \quad (a + b) + c
\]

But
\[
x :: y :: z \quad \text{is} \quad x :: (y :: z)
\]
Mixed Arithmetic

Challenge:

How to make

```scala
val x = new Rational(1, 2)
val y = x + 1
```

work?

What about

``` scala
val z = 1 + x
```
Scala’s Type Hierarchy
Top Types: Any, AnyRef, AnyVal

Any

The base type of all types

Methods:

  ==, !=, equals
  #*, hashCode
  toString
  asInstanceOf
  isInstanceOf

AnyRef

The base type of all reference types, alias of java.lang.Object

Methods:

  eq, ne

AnyVal

The base types of all value types
The Nothing Type

- Nothing is a type without any values.
- Why is that useful?
  - To signal abnormal termination:
    ```java
to see Error() has type Nothing
  - As an element of empty collections.
- The two meanings hang together: Taking an element of a
  List[Nothing] has type Nothing, and will not terminate normally.
The Null Type

- The null value also has a type in Scala; it is called `Null`.

- `Null` is a subtype of every reference type in Scala, but it is not compatible with value types.

```scala
scala> val x = null
x: Null = null

scala> val x: String = null
x: String = null

scala> val x: Int = null
<console>:7: error: type mismatch; found   : Null(null)
         required: Int
         val x: Int = null
   ^
```
Scala collections: Mutable and Immutable

Scala collections systematically distinguish between mutable and immutable collections. All collection classes are found in the package scala.collection

A mutable collection can be updated or extended in place. This means you can change, add, or remove elements of a collection as a side effect. They are found in the package scala.collection.mutable

Immutable collections, by contrast, never change. You have still operations that simulate additions, removals, or updates, but those operations will in each case return a new collection and leave the old collection unchanged. They are found in the package scala.collection.immutable
Scala Immutable collection
Scala Mutable collection
Lists

The List is one of the most important data types in Scala. Here are some examples of Lists.

```
val fruit = List("apples", "oranges", "pears")
val nums = List(1, 2, 3, 4)
val diag3 =
  List(
    List(1, 0, 0),
    List(0, 1, 0),
    List(0, 0, 1)
  )
val empty = List()
```
Sequences

Lists in Scala are just one implementation of the general abstraction of sequences. Other sequence types are `Array`, `ArrayBuffer`, `String` and `Vector`.

```scala
val fruit = List("apples", "oranges", "pears")
val nums = ArrayBuffer(1, 2, 3, 4)
val diag3 =
  Array(
    Array(1, 0, 0),
    Array(0, 1, 0),
    Array(0, 0, 1)
  )
val empty = Vector()
val v = Vector(1, 2, 3)
```
Vector versus List

• Lists are very efficient when the algorithm processing them is careful to only process their heads.

• Accessing, adding, and removing the head of a list takes only constant time, whereas accessing or modifying elements later in the list takes time linear in the depth into the list.

• Vector is a collection type that addresses the inefficiency for random access on lists. Vectors allow accessing any element of the list in “effectively” constant time.
Sequence Types

List[T], Vector[T], etc are the type of sequences with elements of type T. They are parameterized types - in Java it would be List<T>, Vector<T>. Here are the previous definitions again, with types given.

```scala
val fruit: List[String] = List("apples", "oranges", "pears")
val nums: ArrayBuffer[Int] = ArrayBuffer(1, 2, 3, 4)
val diag3: Array[Array[Int]] =
  Array(
    Array(1, 0, 0),
    Array(0, 1, 0),
    Array(0, 0, 1)
  )
val empty: Vector[Nothing] = Vector()
```
Sequence Type Hierarchy

All* sequence types are subtypes of type Seq[T].
* Except for Array[T], String which, coming from Java, can only be implicitly convertible to Seq[T]

So we can also type-annotate as follows:

``` scala
val fruit: Seq[String] = List("apples", "oranges", "pears")
val nums: Seq[Int] = ArrayBuffer(1, 2, 3, 4)
val diag3: Seq[Array[Int]] =
  Array(
    Array(1, 0, 0),
    Array(0, 1, 0),
    Array(0, 0, 1)
  )
val empty: Seq[Nothing] = Vector()
```
Functions on Sequences

- `xs.isEmpty` is sequence empty?
- `xs.length` length
- `xs.head, xs.last` first / last element
- `xs.tail, xs.init` all elements except first / last
- `xs take n` first n elements
- `xs drop n` all elements except first n
- `xs slice (start, end)` same as `xs.drop(start).take(end-start)`
- `xs splitAt n` split into `(xs.take(n), xs.drop(n))`
- `xs.reverse` reversal
- `xs(n), xs.apply(n)` n’th element (indices start at 0)
- `xs contains x` does `xs` contain an element equal to `x`?
- `xs ++ ys` concatenation
- `xs zip ys` a sequence of pairs of corresponding elements from `xs` and `ys`. 
Even more functions on sequences

xs.zipWithIndex    Zips a sequence with its indices (starting from 0)
xs.iterator       An iterator yielding list elements one by one.
xs.unzip           Split a sequence of pairs into two sequences.
xs.flatten         Concatenates a sequence of sequences into a single sequence
xs.sum             The sum of all elements of a sequence of numeric values
xs.max             The maximal element of a sequence of numeric values
xs.min             The minimal element of a sequence of numeric values
xs.mkString(start, sep, end), xs mkString sep
                    Assemble elements in string.
xs.toArray         Conversions
xs.toList
xs.toStream
xs.toSet
xs.toMap
1) Find the last but one element of a list.

Example:
```
scala> penultimate(List(1, 1, 2, 3, 5, 8))
res0: Int = 5
```

2) Find out whether a list is a palindrome.

Example:
```
scala> isPalindrome(List(1, 2, 3, 2, 1))
res0: Boolean = true
```

3) Remove the Kth element from a list. Return the list and the removed element in a Tuple. Elements are numbered from 0.

Example:
```
scala> removeAt(1, List('a', 'b', 'c', 'd'))
res0: (List[Symbol], Char) = (List(a, c, d), b)
```