Scala & Spark

PTT18/19

Prof. Dr. Ralf Lämmel
Msc. Johannes Härtel
Msc. Marcel Heinz
What is Scala?

- Scala is a **general purpose** programming language.
- Scala provides support for **functional programming**.
- Scala has a strong **static type system**.
- Scala source code is compiled to Java bytecode that runs on the **JVM**.
- Scala provides **language interoperability** with Java.

This is hello world:

```scala
object Main {
  def main(args: Array[String]): Unit = 
    println("Hello, Scala developer!")
}
```
Trending Scala Projects

[Logos of akka, kafka, play, Spark, finagle]
Trending Scala Projects

Message-driven Applications

- akka
- kafka
- play
- Spark
- finagle

(C) 2018, SoftLang Team, University of Koblenz-Landau
Trending Scala Projects

Message-driven Applications

A Distributed Streaming Platform

akka

kafka

play

Spark

finagle

(C) 2018, SoftLang Team, University of Koblenz-Landau
Trending Scala Projects

- Message-driven Applications
- A Distributed Streaming Platform
- High Velocity Web Framework
Trending Scala Projects

Message-driven Applications
- akka

High Velocity Web Framework
- Play

A Distributed Streaming Platform
- Apache Kafka®

Lightning-fast Unified Analytics Engine
- Spark

(C) 2018, SoftLang Team, University of Koblenz-Landau
Trending Scala Projects

- Message-driven Applications
  - akka
- High Velocity Web Framework
  - play
- A Distributed Streaming Platform
  - kafka
- Lightning-fast Unified Analytics Engine
  - Spark
- Extensible RPC System
  - finagle

(C) 2018, SoftLang Team, University of Koblenz-Landau
Scala

def products = orders.flatMap(o => o.products)

Java

public List<Product> getProducts()
{
    List<Product> products = new ArrayList<Product>();
    for (Order order : orders)
    {
        products.addAll(order.getProducts());
    }
    return products;
}
Context

IDEs, SBT and JVM.
Context: IDEs
IntelliJ or Eclipse provide an interactive development environment for Scala.
Scala comes with the Scala Build Tool (SBT) written in Scala using a DSL that also supports dependency management.

```
name := "org.softlang.initialscala"

val buildSettings = Defaults.coreDefaultSettings ++ Seq(
  scalaVersion := "2.11.11"
)

lazy val core = project
  .settings(
    buildSettings,
    libraryDependencies +++= Seq(
      "commons-io" % "commons-io" % "2.5",
      "org.apache.spark" % "spark-core_2.11" % "2.3.1",
      "org.apache.spark" % "spark-mllib_2.11" % "2.3.1",
      "org.apache.spark" % "spark-hive" % "2.3.1",
      "org.apache.hadoop" % "hadoop-client" % "2.7.2"
    )
  )
```
Scala compiles to Java bytecode that runs on the JVM. Calling Scala from Java looks funny (see this decompiled scala class).

```scala
class Person (var name: String, var age: Int)
```

```java
public class Person extends java.lang.Object implements scala.ScalaObject{
    public java.lang.String name();
    public void name_$eq(java.lang.String);
    public int age();
    public void age_$eq(int);
    public Person(java.lang.String, int);
}
```
Basics

[scdoc]
https://docs.scala-lang.org/
Basics: Expressions and Values

Expressions are computable statements. The keyword ‘val’ defines values that name results of expressions. They do not need to be recomputed and they cannot be reassigned.

```plaintext
// An simple expression as statement.
1 + 1

// Printing an expression.
println(1 + 1)

// Assignment of a values.
val x = 1 + 1

// Assignment of a values.
val x : Int = 1 + 1

// Re-assigning a value does not work.
x = 2
```

(C) 2018, SoftLang Team, University of Koblenz-Landau
Basics: Variables

The keyword ‘var’ defines Variables that can be declared like values. Variables can be reassigned to a different expression.

```plaintext
// A variable declaration.
var y: Int = 1 + 1

// Printing 2 into the console.
println(y)  // 2

// Reassignment of such declaration.
y = y + 1

// Printing 3 into the console.
println(y)  // 3
```
Basics: Blocks

Expressions can be surrounded by a Block with '{' and '}'. The result of the last expression in this block is the result of the overall block.

```scala
// Printing a Block.
println(
  val x = 1 + 1
  x + 1
)
```

3
Basics: Functions

Functions are expressions that take parameters. To the left of keyword ‘=>’, a list declares available parameters and to the right an expression involving those parameters.

```scala
// Definition and application of an anonymous function.
((x: Int) => x + 1) (\v1\ = 4)

// Naming such function.
val addOne = (x: Int) => x + 1
println(addOne(1))
```

2
Basics: Methods

Methods look and behave very similar to functions. The keyword ‘def’ is followed by a name, multiple parameter lists, an optional return type, and a body.

```python
// A method adding two integers.
def add(x: Int, y: Int): Int = x + y
println(add(1, 1))

// A method with two parameter lists.
def addThenMultiply(x: Int, y: Int)(multiplier: Int): Int =
    (x + y) * multiplier

println(addThenMultiply(1, 2)(multiplier = 3))
val fkt: Int => Int = addThenMultiply(1, 2)
println(fkt(2))

// A method with no parameters.
def someNumber = 12
println("This is a some number " + someNumber)
```
Basics: Classes

The keyword ‘class’ defines classes taking a list of constructor parameters. Methods with the singleton ‘Unit’ return type carry no information and are called because of its side-effects.

```scala
// A class definition with constructor parameters.
class Greeter(prefix: String, suffix: String) {
  def greet(name: String): Unit =
    println(prefix + name + suffix)
}

// Creating and instance of such class and
// printing 'Hello, Scala developer!
val greeter = new Greeter( prefix = "Hello, ", suffix = "!")
greeter.greet( name = "Scala developer")
```
The prefix ‘case’ distinguishes case classes from classes. Case classes are immutable and can be compared by value.

```scala
// A case class definition
case class Point(x: Int, y: Int){
   // ...
}

// Creating instances without the 'new' keyword.
val point1 = Point(1,2)
val point2 = Point(1,2)

// Case classes can be compared by value.
println(point1 == point2)
```
Basics: Objects

Objects are singleton instances of their own definition.

```java
// A singleton factory for unique ids.
object IdFactory {
    private var counter = 0

    def create(): Int = {
        counter += 1
        counter
    }
}

// A new id is printed.
println(IdFactory.create())
```
Basics: Name Arguments
Comparable to Python you can pass the method arguments by name.

```python
def printName(first: String, last: String): Unit = {
  println(first + " " + last)
}
// Always print 'John Smith'.
printName("John", "Smith")
printName(first = "John", last = "Smith")
printName(last = "Smith", first = "John")
```
Basics: For Comprehension

An enumerator contains either a generator which introduces new variables, or it is a filter. The yield expression is executed for every generated binding of the variables.

```scala
val userBase = List(User("Travis", 28),
                     User("Kelly", 33),
                     User("Jennifer", 44),
                     User("Dennis", 23))

val twentySomethings = for (user <- userBase if user.age >= 20 && user.age < 30)
                        yield user.name

twentySomethings.foreach(name => println(name))
```

- Travis
- Dennis
Basics: Main Method

The Java Virtual Machine requires a main method to be named ‘main’ as an entry point of the program. It takes an array of strings as arguments.

```scala
object Main {
    def main(args: Array[String]): Unit =
    println("Hello, Scala developer!")
}
```
Best Practices

[twbp]
http://twitter.github.io/effectivescala/
‘While highly effective, Scala is also a large language, and our experiences have taught us to practice great care in its application. What are its pitfalls? Which features do we embrace, which do we eschew? When do we employ “purely functional style”, and when do we avoid it?’ [twbp]
Best Practices: Optional

Using the ‘Optional’ container provides a safe alternative to the use of ‘null’.

```scala
var username: Option[String] = None
username = Some("foobar")
```

```scala
username.isDefined
```

(C) 2018, SoftLang Team, University of Koblenz-Landau
Best Practices: Destructuring

Destructure tuples or case classes during the binding instead of accessing its properties using the methods ‘_1’ or ‘_2’.

```
val tuple = ('a', 1)
val first = tuple._1
val second = tuple._2
```

```
val (first, second) = tuple
```
Best Practices: Destructuring & Matching

Combine pattern matching with such destructuring.

```python
animal match {
  case dog: Dog => "dog (%s)".format(dog.breed)
  case _ => animal.species
}
```

```python
animal match {
  case Dog(breed) => "dog (%s)".format(breed)
  case other => other.species
}
```
Best Practices: Matching
Use pattern matching whenever applicable but collapse it.
Best Practices: Mutable Collections

Prefer using immutable collections. If referencing to mutable Collections, use the ‘mutable’ namespace explicitly.

```scala
import scala.collection.mutable._
val set = Set[Int]()
set.add(1)
```

```scala
import scala.collection.mutable
val set = mutable.Set[Int]()
set.add(1)
```
Best Practices: Collection Construction

Use the default constructors for collection type. This style separates the semantics of the collection from its implementation and allows compiler optimization.

Use the default constructors for collection type. This style separates the semantics of the collection from its implementation and allows compiler optimization.

```scala
val seq = List(1, 2, 3)
val set = HashSet(1, 2, 3)
val map = HashMap(1 -> "one", 2 -> "two", 3 -> "three")
```

Use the default constructors for collection type. This style separates the semantics of the collection from its implementation and allows compiler optimization.

```scala
val seq = Seq(1, 2, 3)
val set = Set(1, 2, 3)
val map = Map(1 -> "one", 2 -> "two", 3 -> "three")
```
Best Practices: Java Collections

Use the converters to interoperate with the Java collection types.

```scala
val list = new java.util.ArrayList[Int]() for (x <- Seq(1, 2, 3, 4)) list.add(x)

var buffer = Seq[Int]() for (i <- 0 to list.size()) buffer = buffer ++ Seq(list.get(i))

import scala.collection.JavaConverters._
val list: java.util.List[Int] = Seq(1, 2, 3, 4).asJava
val buffer: scala.collection.mutable.Buffer[Int] = list.asScala
```
Best Practices: Implicit Conversion

Implicits should be used sparingly, for instance in case of a library extension ("pimp my library" pattern).

```scala
implicit class SetWithPrintLn[T](xs: Set[T]) {
  def println(): Unit = for (x <- xs) {
    System.out.println(x)
  }
}

val set = Set("e1","e2","e3")

set.println()
```
Best Practices: Return

Use ‘return’ to enhance readability but not as you would in an imperative programming language.

```python
def suffix(i: Int): String = {
    if (i == 1) return "st"
    else if (i == 2) return "nd"
    else if (i == 3) return "rd"
    else return "th"
}
```

- Use simple return statements instead of if-else conditions.

```python
def suffix(i: Int): String = {
    if (i == 1) "st"
    else if (i == 2) "nd"
    else if (i == 3) "rd"
    else "th"
}
```

- Simplify return values directly.

```python
def suffix(i: Int): String = i match {
    case 1 => "st"
    case 2 => "nd"
    case 3 => "rd"
    case _ => "th"
}
```

- Use pattern matching for concise return logic.
Best Practices: Style

Keep track of all the intermediate results that are only implied.

```scala
val votes = Seq(("scala", 1), ("java", 4), ("scala", 10),
                 ("scala", 1), ("python", 10))
val orderedVotes = votes
  .groupBy(_.1)
  .map { case (which, counts) =>
    (which, counts.foldLeft(0)(_ + _.2))
  }.toSeq
  .sortBy(_.2)
  .reverse

val votesByLang = votes.groupBy { case (lang, _) => lang }
val sumByLang = votesByLang.map { case (lang, counts) =>
  val countsOnly = counts.map { case (_, count) => count }
  (lang, countsOnly.sum)
}
val orderedVotes = sumByLang.toSeq
  .sortBy { case (_, count) => count }
  .reverse
```
Best Practices: FlatMap

High order functions like ‘map’ or ‘flatMap’ are also available in nontraditional collections such as Future and Option. Using ‘for’ translates into the former.

```scala
var host: Option[String] = None
var port: Option[Int] = None

def connect: Option[InetSocketAddress] = host.flatMap { h =>
  port.map { p =>
    new InetSocketAddress(h, p)
  }
}

def connect: Option[InetSocketAddress] = for {
  h <- host
  p <- port
  yield new InetSocketAddress(h, p)
}
```
Best Practices: ADTs
Using case classes to encode ADTs together with pattern matching. This results in code that is “obviously correct”.

```scala
sealed trait Tree[T]

case class Node[T](left: Tree[T], right: Tree[T]) extends Tree[T]

case class Leaf[T](value: T) extends Tree[T]

def findMin[T <: Ordered[T]](tree: Tree[T]): T = tree match {
  case Node(left, right) => Seq(findMin(left), findMin(right)).min
  case Leaf(value) => value
}
```
Distributing Scala’s high level functions.
Spark: A simple Task.
Counting the words of some Lorem Ipsum.

```
val counts = textFile.flatMap(line => line.split(\s+))
  .map(word => (word, 1))
  .reduceByKey(_ + _)
  .sortBy(_._2, ascending = false)

for ((word, count) <- counts.take(num = 10)) println(count + " " + word)
```
Spark: Distributing and Fetching Data

A spark session is created (this time a local one with 16 cores). The data is processed using the provided API in the RDD class (resilient distributed dataset).

```scala
val sc = SparkSession
  .builder()
  .config(new SparkConf().setMaster("local[16]"))
  .getOrCreate()

val textFile = sc.sparkContext.textFile("data/loremipsum.txt")
val counts = textFile.flatMap(line => line.split(" "))
  .map(word => (word, 1))
  .reduceByKey(_ + _)
  .sortBy(_._2, ascending = false)

for ((word, count) <- counts.take(num = 10)) println(count + " " + word)
```

Distribute the data

Fetch back the data
Spark: Infrastructure
Spark serializes the functions and sends them to the workers. Further it provides 4 mechanisms to exchange data, i.e., parallelize, broadcast, collect and accumulate.
Spark: Partitions

The Lorem Ipsum is split into several partitions that can be processed in isolation; hence, on different nodes.
Spark: Partitions

Reading the lines of a local file into a resilient distributed dataset (RDD) with three partitions.
Spark: Partitions

Splitting the lines with ‘flatMap’ into words. This can be done within the same partition as there is no dependency between the different sentences.

code: `textFile.flatMap(line => line.split(\" \"))`
Spark: Partitions

Tuples of word and number are formed that can later be summed up. Now there are plenty of tuples with the same key and the value ‘1’.
Spark: Partitions

‘reduceByKey’ aggregates the values of all tuples with the same key. Unfortunately this requires records to switch between partitions causing network traffic.
Spark: Partitions

Sorting of the records also requires movement data between partitions. This time the shuffling is not based on the hash of the key.

```scala
.partition(_.2, ascending = false)
```
Spark: Benefits

The data and processing can be distributed over several nodes of a cluster. The partitions can be (i) recomputed when needed, (ii) persistent on the hard drive if the computation is expensive or (iii) kept in memory if it is accessed frequently.
Spark: A complex Task (LOC)

Question: ‘Who contributed the most LOC in the evolution of a Git repository?'
Spark: A complex Task (LOC)

Some values needed during the analysis.

```scala
// An target project and an access to its repository.
val project = "libgdx/libgdx"
val git = Gitobject(project)

// The list of commits (for libgdx around 12000).
val commits = git.headCommits()

// Data that is needed all over the analysis.
val min = commits.map(git.time).min
val max = commits.map(git.time).max
val length = max - min

// An accumulator called 'Analysed Objects' to keep track
val accumulator = SparkUtils.sc.longAccumulator(name = "Analysed Objects")
```

- A target project and some access wrapper for the repository.
- First and last commit time. Values needed and send with the instructions.
- A way to keep track of how many objects have been processed.
Spark: A complex Task (LOC)
Send the commits to the worker nodes forming 16 partitions (by calling ‘parallelize’). Fatten this records in that they represent all ‘objects’ in a tree of a particular commit (like splitting the Lorem Ipsum sentence into words).

```scala
// The RDD of (path,object,commit) records.
val objectCommits: RDD[((String, String), String)] =
SparkUtils.sc.parallelize(commits, numSlices = 16)
 .flatMap(commit => Gitobject(project).tree(commit))
 .collect { case (path, obj) if path.endsWith(".java") => ((path, obj), commit) }
```

22,443,734 records

((.../collision/btDispatcher.java,
60ad8e527688ac799d7f21c52e6a79c62e78bde2),0273c2288d39777528ece72ffa47c0b8790b42ab)
((.../ApplicationConfiguration.java,
da9aa59e2daed4642a9a953d914daa0e94f05ee0),0273c2288d39777528ece72ffa47c0b8790b42ab)
((.../ShadowMappingTest.java,
19ce36397ca0088ddd9a0ef6de3546fdcd4be4b29),0273c2288d39777528ece72ffa47c0b8790b42ab)
((.../ShortSoundTest.java,
05a780eb16f84e0fab565cb4ae6ea4e21d830818),0273c2288d39777528ece72ffa47c0b8790b42ab)
Spark: A complex Task (LOC)

We don’t want to analysis the same object twice so we group by object and path. We increase the number of partitions to 256 during the required shuffle step.

```scala
// The pair RDD of ((path,object),commits) records.
val objects: RDD[((String, String), Iterable[String])] =
    objectCommits.groupByKey(numPartitions = 256)
```

39.754 records
Spark: A complex Task (LOC)

Now GIT ‘blame’ can be applied on each object.

```scala
// The RDD of ((author, mail), path, metric) records.
val authorPathBlames: RDD[((String, String), String, Double)] =
  objects.flatMap { case ((path, obj), commitsOnObj) =>
    // Get a thread specific access to the repository since this runs parallel.
    val git = GitObject(project)
    // Inform the accumulator for logging,
    accumulator.add(1)
    // The time this object is valid in the overall
    val factor = commitsOnObj.map { commit =>
      git.parents(commit).headOption.map(git.time)
        .map(t => (git.time(commit) - t).toDouble / length.toDouble).getOrElse(0.0)
    }.sum
    // Run blame.
    git.blame(path, obj).map { case (author, metric) => (author, path, metric.toDouble * factor) }
}.persist(StorageLevel.MEMORY_AND_DISK)
```

We cannot serialize the repository wrapper ‘git’ in the instruction so we create a new one on each worker.

Increase object conter

154,565 records
Spark: A complex Task (LOC)

We sum up the values by author ignoring the path.

```scala
// The RDD of ((author,mail),metric) records.
val authorBlames = authorPathBlames.map { case (author, _, value) =>
  (author, value)
}.reduceByKey(_ + _)
```
Spark: A complex Task (LOC)

All stages, i.e., a set of parallel tasks with one task per partition can be supervised in the Web UI.

```scala
val authorBlames = authorPathBlames.map {
  case (author, _, value) =>
    (author, value)
}.reduceByKey(_ + _)
```
Spark: A Comparison

There is also a Spark API available for Python and Java, however, Spark is implemented in Scala.

```python
text_file = sc.textFile("hdfs://...")
counts = text_file.flatMap(lambda line: line.split(" "))
    .map(lambda word: (word, 1))
    .reduceByKey(lambda a, b: a + b)
counts.saveAsTextFile("hdfs://...")
```

```java
JavaRDD<String> textFile = sc.textFile("hdfs://...");
JavaPairRDD<String, Integer> counts = textFile
    .flatMap(s -> Arrays.asList(s.split(" ")).iterator())
    .mapToPair(word -> new Tuple2<>(word, 1))
    .reduceByKey((a, b) -> a + b);
counts.saveAsTextFile("hdfs://...");
```
Spark: A Comparison
The Java 8 native solution is to use parallel Streams which is also based on chunking. However, the partitions are not distributed over a cluster.

```java
public Stream<Employee> filterE(Department d) {
    Stream<Employee> stream1 = d.getEmployees() .parallelStream()
        .filter(e -> e.getSalary() < 2000);
    Stream<Employee> stream2 = d.getSubdepts() .parallelStream()
        .map(sub -> filterE(sub))
        .reduce(Stream.of(), Stream::concat);

    return Stream.concat(stream1, stream2);
}
```
References

- [scdoc] https://docs.scala-lang.org/
- [ep3] https://gist.github.com/calincru/cea751f050883581730093e93eaf2723
- [tp] https://www.tutorialspoint.com/scala
- [spark] https://spark.apache.org/examples.html
- [spark2] https://spark.apache.org/docs/latest/