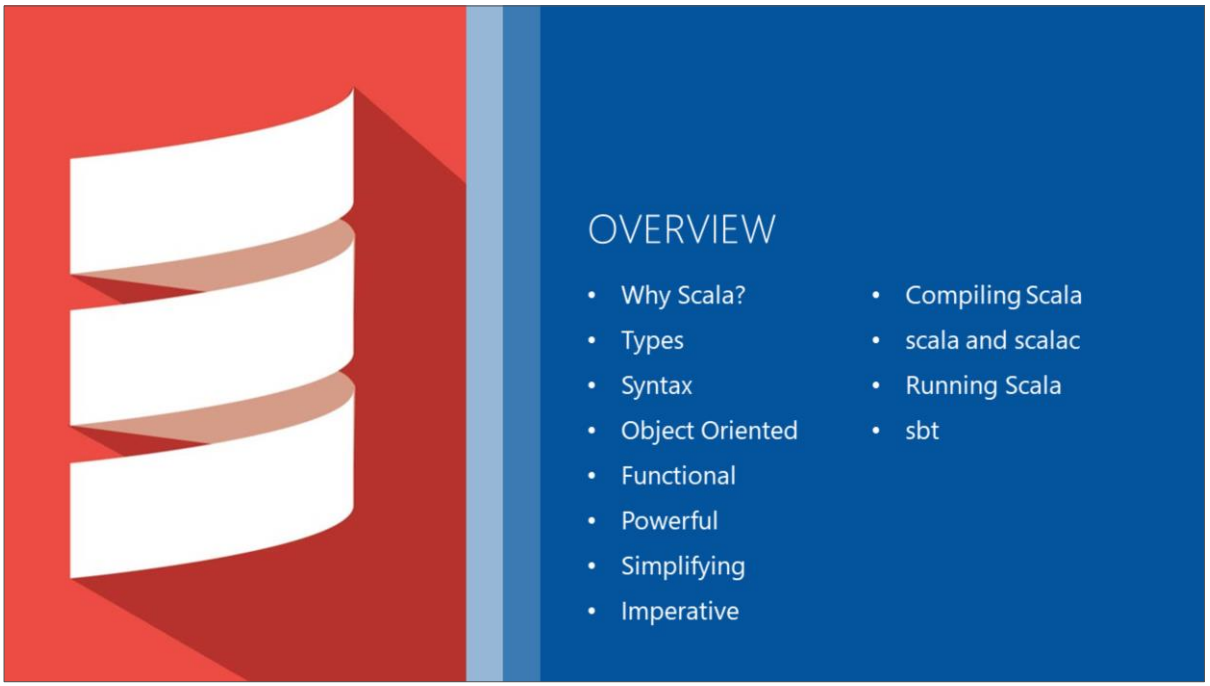
Overview

SCALA PROGRAMMING
A DATA SCIENCE AND MACHINE LEARNING COURSE



This chapter is a high level overview of the major motivations behind the design of scala, and why it is used.

Since many of these ideas will be new, and they are treated quickly here, this is a difficult chapter to understand fully on first read. The takeaways are the themes and sense of direction that these provide.

Why Scala?

- Martin Odersky in 2004
  - functional language for jvm
  - inspire by Haskell
  - practical
  - pragmatism between functional and OO
- A powerful language
  - killer libraries
  - BigData
  - engineering glue

3

Martin Odersky in 2004 decided jvm needed a functional language

inspire by Haskell

intention of making Haskell-like functional programming practical

Resume java libraries, performance of jvm

Combine object orientation with functional programming for a pragmatic compromise

A powerful language with "killer libraries"

especially relevant in BigData applications

As an engineering glue language

Modernizing replacement for java

Java - ;

Types

- rules
  - of the source
- statically inferred
  - by the complier

```
val name = "Michael"
var age = 27


//in full
val fname: String = "Michael" : String
var fage: Int = 27 : Int


// functions as in -> out
def describe(name: String, age: Int) =
  s"${name} (${age})"

println(describe(name, age))
```

statically typed

Types are rules that apply to the construction of the text of the program

object-functional language

types statically inferred

All rules are present and well-defined at compile time

Types often do not need to be written in-file but can be inferred

Manifest typing

Aside: Type Systems

- Types
  - Rules
  - Properties
  - Sets
- Weak
- Strong
- Latent
- Manifest
- Dynamic
- Static

5

Types = Information about values (= Rules = Properties = Sets = …)

Weak (ly enforced rules)

Strong (ly enforced rules)

Latent – not there/written

Manifest – written down

Dynamic – types are tags on objects/values

Static -- types are rules applied to the text of the program (source)

Weak dynamic latent = JavaScript

Weak static manifest = c

Strong static latent = Haskell

Strong static manifest = java

Strong dynamic latent = python

Weak dynamic manifest = php

Strong static manifest (a little bit latent) = scala

Syntax

- Compositional
- Expressions over statements
  - calculative phrases
  - typed
  - type equivalent

```
def describe(name: String, age: Int) =
  s"${name} (${age})"

val name = "Michael"
val age = 27

println(
  describe(name, age) + " is " + (
    if (age >= 18) {
      "allowed"
    } else {
      "not allowed"
    }
  ))



// OUTPUT :

scala>

Michael (27) is allowed
```

Syntax is compositional, piece of it can be cut and pasted in many compatible locations

A compatible location is determined by type

In imperative languages the if/else construct is a statement, nothing further than be done to it

Fn(if(test) { doA(); } else { doB(); })

is a syntax error in java but not in scala

Syntax is mostly made of expressions

Phrases which calculate values

The entire phrase has the same type as the value it makes

So the phrase can be cut/pasted into any place of the same type

Eg. Into the argument of a method

Object Oriented

- Values are objects
- Objects are structured
  - Class Reference
  - State
  - Uniqueness
- Namespaces
  - obj.method()
- Impure Methods

```
class Human(var age: Int) {
  def grow_older() {
    age += 1
  }

  def describe() {
    println(age)
  }
}

val me = new Human(27)

me.grow_older()       // actions
me.describe()         // ugly!

// OUTPUT :

scala>


defined class Human
me: Human = Human@65cc5252

scala> 28
```

Scala is also fundamentally object oriented

Every value is an object – a data structure which comprises a reference to a template it was made from, its remembered data, and a unique identifier

The basic term of the language is still therefore

obj.method()

And methods may still have no meaningful return value

Therefore act impurely by appealing to some form of revision to memory or input-outpu

Functional

- Functional Programming
- Pure Functional Programming
  - output = f(g(h(i(j(k(input)))))
  - cake = bake(mix(prep(ingredients)))
- Too explicit
  - require simplifying syntax
  - learning challenge

```
val people = List(
  Map("id" -> 1, "name" -> "Sherlock",
    "address" -> "London, United Kingdom"),

  Map("id" -> 2, "name" -> "Jefferson",
    "address" -> "Virginia, United States")
)

println(
 people map { _("name") } mkString ","
)


// OUTPUT :

scala>


Sherlock, Jefferson
```

Pure functional programming is a paradigm

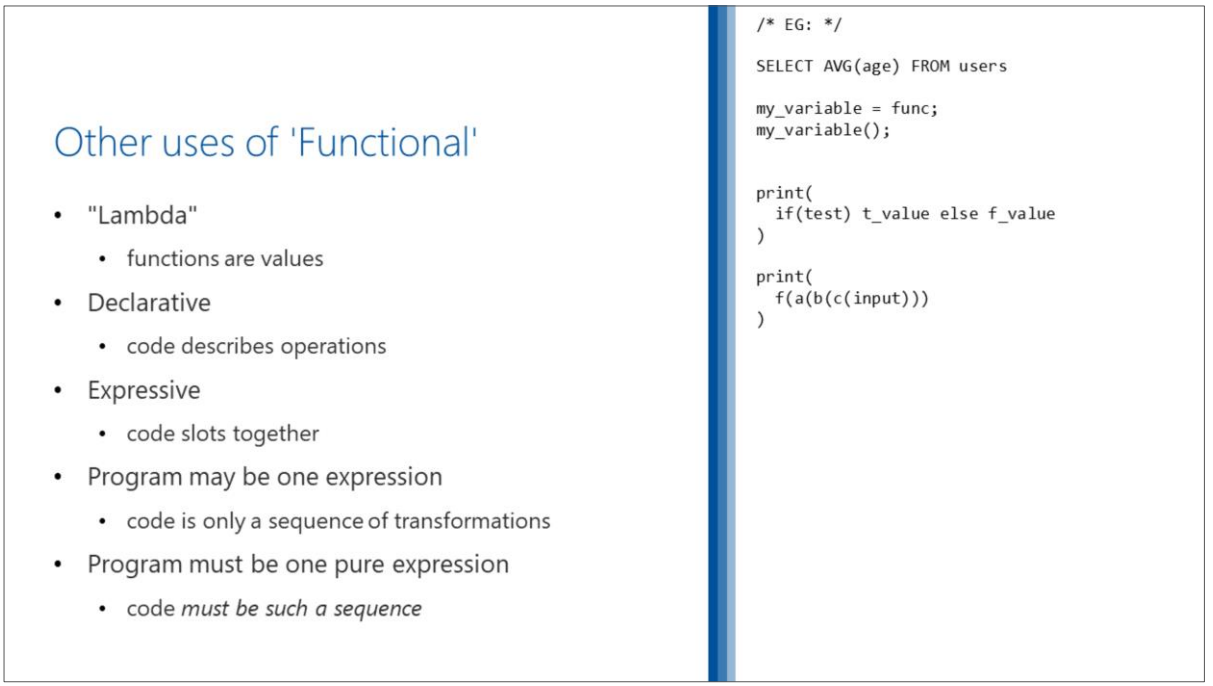
- Wherein all programs are phrases as translations
- From old values to new values
- Without changing any values in-place

Pure programs therefore have the form

- output = f(g(h(i(j(k(input)))))
- eg., cake = bake(mix(prep(ingredients)))

This is often laboriously explicit

- Scala provides powerful simplifying syntax
- The power of this syntax is the learning challenge
- The underlying program is always just a calculation



The term functional programming is very broad it might refer to languages where

syntax declarative

eg., SELECT AVG(age) FROM users

Functions are values

ie., my_variable = fn;

Programs may be reduced to a single expression

Output = f(a(b(c(input)))

Syntax is mostly kinds of expression

eg. if(test) t_value else f_value

Syntax is only has expressions

Eg., print() does not exist and everything is value-returning

Programs must be a single expression

Scala is functional in all but the last sense

Powerful

- Libraries
  - Play
  - Akka
  - Spark
  - Data Engineering
- Equal access to jvm
  - different expressive power
  - versioning

```
val svc = io.Source.fromFile(

"""C:\Windows\System32\drivers\etc\services"""
)

val inUsePorts = (for (
  line <- svc.getLines;
  port <- """\d+""".r.findFirstIn(line)
) yield port.toInt).toSet

val freePorts =
 Set(1 to 200: _*) &~ inUsePorts



// OUTPUT :
scala>

freePorts: Set[Int] = Set(110, 196, 74, 109,
149, 71, 172, 81, 72, 146, 195)
```

Many popular libraries are written in scala for its expressive power

Play – widely used web framework

Akka – widely used concurrency framework

Spark – widely used machine learning framework

Access to this ecosystem gives scala an advantage in the data engineering sector

Any jvm language, eg. java has equal "access" but its syntax will not express the native ideas of these libraries clearly

Simplifying

- java – ;
  - type inference
  - class definition
- Optimizing Reasoning
  - execution
  - debugging, refactoring, iterating, design
  - cogitating vs keyboarding

```
class Person(
  val n: String, val a: Int
) {
  def describe() = s"${n} (${a})"
}

println(
  new Person("Sherlock", 27).describe()
)
```

```
// OUTPUT :

scala>

defined class Person

Sherlock (27)
```

Scala can be used as a "java without semicolons" and in such a use case becomes merely a simplified syntax for writing java

Scala's syntax condenses more information into fewer lines and dispense with inessential information

Eg. Type hinting, the class definition syntax

The vast majority of time spent on a programming is in the cognition of the programmer

Reasoning about the execution of the program, debugging, refactoring, iterating and considering its design

The net number of keystrokes on a 6mo project is quite small!

Scala aims to improve programmer reasoning within a jvm environment by encouraging a pure functional style

In which, by eliminating side effects, reasoning is easier

Imperative

- Sequences of statements
- Memory Efficient
  - Revising in-place vs creating
- Pure interfaces
  - impure implementations

```
var total: Int = 0
var i: Int = 0
var people = List("Me", "You")

while ( i < people.length ) {
 total += people(i).length
 i += 1
}

println(total)
```

Along side functional and declarative features scala is also imperative

Programs may be written as a sequence of statements

This style is sometimes more memory effcicient

Revising the state of a object can be more efficient than building up a new one

Scala allows you to fall-back to an imperative form if this is suitable for your problem

However, wherever possible, provide a pure interface

Eg. Modify state within a method, but ensure that this is merely to return a value

Compiling Scala

- Entry Point
  - main()
    - commandline args
  - App
    - scripty
- object for namespacing

```
//$ scalac myapp.scala && scala MyApp
object MyApp extends App {
  println("Hello World!")
}


//$ scalac main.scala
//$ javap HelloWorld.class
object HelloWorld {
  def main(a: Array[String]): Unit =
    println("Hello World!")

}
```

All compiled programs must have an entry point

The body of an object which extends App

Or in a main method

For now, consider the object keyword to introduce an object (like any other) that is used for the purpose of grouping, ie., like a package or a namespace

In the first form the body of the object's definition is the whole program and reads like a script

This is done with complier magic related to the App trait

Planned to be decorated and removed in future versions

In the second form the method accepts an array of strings, the command line arguments and returns Unit (roughly equivalent to void)

scala and scalac

- scalac
  - compile .scala to .class
- scala command
  - compiled .class
  - script .sc
  - REPL

```
$ scalac -help
Usage: scalac <options> <source files>
where possible standard options include:

$ scala -help
Usage: scala <options>
[<script|class|object|jar> <arguments>]
   or  scala -help
All options to scalac (see scalac -help) are
also allowed.

$ scala
Welcome to Scala 2.11.8 (Java HotSpot(TM) 64-
Bit Server VM, Java 1.8.0_111).
Type in expressions for evaluation. Or try
:help.
```

The scala command runs scala code

Either code compiled into .class files with scala

Or a .sc script file to be run line-by-line

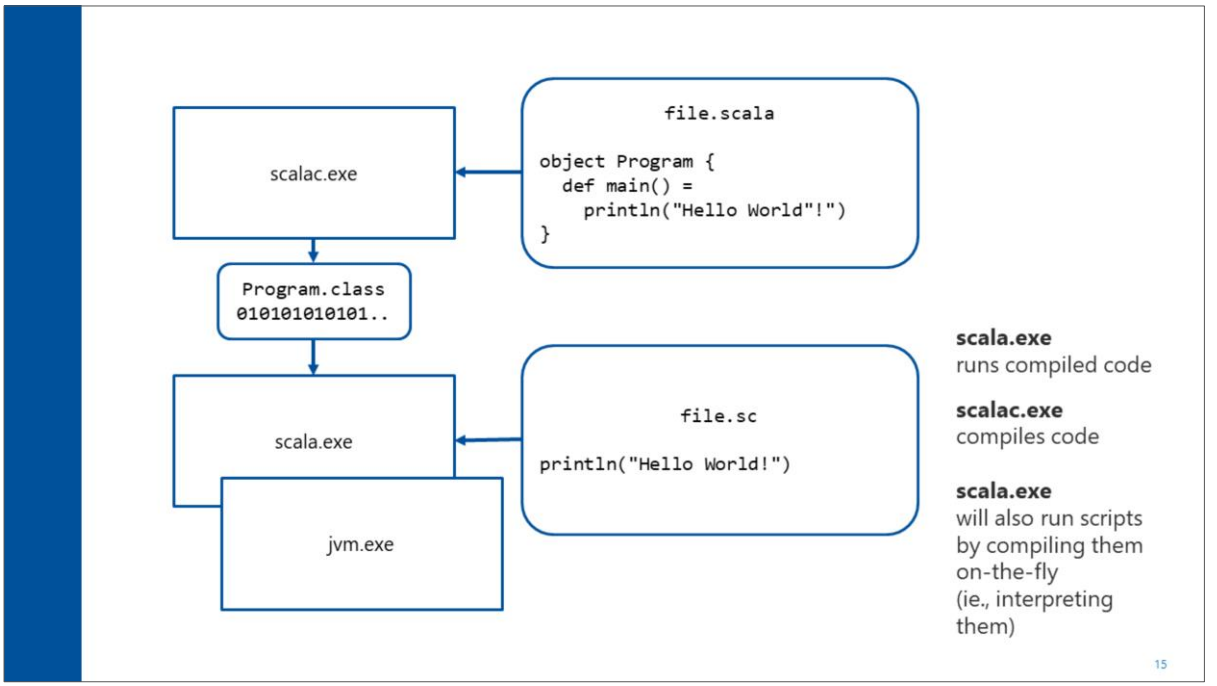
Or, running the command without a file, opens up the REPL (“console”) which will be considered latere

The scalac command compiles scala programs into .class files the jvm supports

The scala command runs these files

To run a class file it must contain a main method

Use the name of the file without the class extension with the scala command



.exe added to emphasize we're talking about the programs. On linux this would simply be ./scala

Running Scala

- REPL
  - :help
  - :paste
- REPL first
- REPL
  - println() implied

```
val name = "Michael"
var age = 27

println(name + " is " + age)

if (age >= 18) {
  println("You're allowed!")
} else {
  println("You're not allowed!")
}

def describe(name: String, age: Int) =
  s"${name} (${age})"

println( describe(name, age) + " is " + (
    if (age >= 18) "allowed in"
    else "not allowed in")
)


// OUTPUT :

scala> age: Int = 27
scala> Michael is 27
scala> You're allowed!
scala> Michael (27)
```

Scala code can be run without compiling an entire program

This is often the best way to learn, experiment and debug scala

The REPL (read eval print loop) or console runs commands line by line or in :paste mode whole snippets

Type :help to get console-specific help

The REPL can introduce one confusion:

The replies the REPL gives are not part of your scala program

When compiling a program println() or other IO commands are explicitly required to get data out of memory and on to a screen

In a REPL commands both store data in-memory and then print helpful information

A REPL-reply typically contains type information (:)

These lines will not appear in a compiled program

sbt

- building
  - vs. by hand
  - vs. precompiling libraries
- Simple Build Tool
  - build files in scala
  - tasks
  - console with libraries
  - IDE

```
/* EG: */

$ sbt
> help
> tasks
> tasks -V > compile > test
> clean
> ~test
> console
> run
> show x
> show scalaVersion
> eclipse > exit


val json4sNative = "org.json4s" %% "json4s-
native" % "latest.integration"

lazy val root = (
 project in file(".")
).settings(
  name := "hello",
  version := "1.0",
  libraryDependencies += json4sNative
)

scalaVersion := "2.12.1"
```

In practice, scala developers are not compiling code by issuing scalac commands manually

This would require pre-compiling lots of libraries and coordinating complex build information

To simplify the process The Simple Build Tool (sbt) was created

Sbt uses build files defined in scala

sbt integrates with IDEs or can be used from the command line

Where it defines a series of tasks:

Running help, tests, compiling, running, and the console

Sbt provides a scala console that has pre-loaded all the libraries defined in the build configuration which is useful for debugging a project and trying out libraries

Exercise