# Scala Introduction

Jens Grassel

May 20, 2019

Wegtam GmbH

# Environment and Tooling

# Java Virtual Machine

Scala runs on the JVM, but not only there!

- Scala.js for Javascript endpoints
- Scala Native for LLVM endpoints

# Build Tools

- multiple build tools are available
  - SBT
  - Mill
  - CBT
  - Fury
- SBT has the biggest ecosystem and is widely used
- stick to SBT but maybe checkout Mill ;-)

# SBT Setup

SBT is usually setup on a per project basis via the `build.sbt` file. However here are some nice settings for the global configuration, put it into `~/.sbt/1.0/global.sbt`

```
// Prevent Strg+C from killing sbt.
cancelable in Global := true
// Coloured console.
initialize ~= (_ ⇒
  if (ConsoleLogger.formatEnabled)
    sys.props("scala.color") = "true"
)
```

# SBT Plugins

Some useful global plugins can be enabled by adding them to the file `~/.sbt/1.0/plugins/plugins.sbt`

```
addSbtPlugin("net.virtual-void" % "sbt-dependency-graph" % "0.9.2")
addSbtPlugin("com.timushev.sbt" % "sbt-updates" % "0.4.0")
```

# Development Environment

There are several options available but beginners should maybe stick to a full blown IDE which in the case of Scala means IntelliJ IDEA with the Scala plugin.
For the confident:

- VS-Code with scala-metals
- Vim or Neovim with vim-scala plugin and optionally scala-metals
- Emacs
- Atom
- possibly others...

# Basics

# Values

**Table 1:** Values types in Scala

| Type | Notes |
|---|---|
| Immutable | Data structures which cannot be changed. |
| Mutable | Data structures which can be changed. |
| Val | A variable that cannot be re-assigned. |
| Var | A variable that can be re-assigned (changed). |

# Values II

**Table 2:** When to use which type?

| Definition | Notes |
|---|---|
| Immutable Val | The recommended way to go. |
| Immutable Var | Okay if used in a local scope. |
| Mutable Val | Try not to use this but there may be applications.[1] |
| Mutable Var | Never ever do this! |

[1] However **if** you do this then **never** pass the value around!

# Recursion

As Scala provides tail recursion you should try to use it if possible.

**But...**

Please not simply for the sake of using it!

Evaluate if it is necessary.[2]

**Remember**

Readability and maintainability trump obscure performance gains every time!

[2] Some algorithm are hard or impossible to do with tail recursion!

# Functions

Functions are there to make your life easier!

- Use HOF[3]
- Use Currying
- Use polymorphism

For additional benefit try to stick with **pure** functions. A pure function:

- is total (an output for every input)
- is free of side effects
- its output does only depend on its input

[3] Higher Order Functions

# Implicits

# What are implicits?

An **implicit** value can be used by the compiler to pass it to any function which depends on an implicit parameter of the same type.

# Use cases

- reduce boiler plate (implicit parameters)
- extend existing types with custom functions (wrapper classes)
- convert types implicitly (Do not do this!)

# Best practices

- always specify the type of implicit `val` or `def`
- do not to use implicits for simple datatypes (primitives)
- stick to the naming conventions e.g. `FooOps` when extending `Foo`
- put extension wrappers into `syntax` objects
- Do not use implicit conversions!

# Objects, Classes and Traits

# Objects

- the most simple container format in Scala
- basically singletons (in Java world)
- no constructor and **no type parameters**[4]
- can extend one class and one or more traits
- start out with objects and *upgrade* later if needed

**About mixing in traits...**

Try to avoid mixing in a lot of traits into your objects. See the infamous *Cake Pattern* which will lead to tight coupling and other issues.

[4] This will be become important later.

# Classes

Classes are like objects but have more features:

- a constructor
- can take type parameters
- can have a companion object

Use classes if

- you want to make your code more generic (type parameters)
- you don't want to pass a dependency to each function (use it in the constructor)

# Traits

Traits define abstract interfaces

- no constructor
- can take type parameters

Use traits to

- model sum types with `sealed traits`
- define modules
- define type classes

# Type Classes

# Use cases

# Encodings

# Laws

# Useful libraries for functional programming

# Cats

Cats provides a core library and additional modules for functional programming in Scala.

- Documentation at the website: https://typelevel.org/cats/
- Book "Scala with Cats" (Noel Welsh and Dave Gurnell)
- several sub modules
  - Cats-Effect (IO Monad for Scala)
  - Cats Tagless (A library of utilities for tagless final algebras)
  - Kittens (Automatic type class derivation)
  - a lot of others...

# Refined

Refined types allow you to add more constraints to types.

**Example**

```
type NES = String Refined NonEmpty
val password:  NES = ???
```

- Documentation at the website: https://github.com/fthomas/refined
- integration with Cats
- integration with ScalaCheck
- can add significant compile time overhead

# ScalaCheck

ScalaCheck brings property based testing to Scala.

- Documentation at the website: http://www.scalacheck.org/
- integration with ScalaTest

**Note**

Even when (mis)used for generators[5] only it brings significant improvements for testing.

[5] Generators provide instances for datatypes as input for testing.