

# An Introduction to Scala

...

BJ Kennedy  
@arcdrag

# ABOUT THIS PRESENTATION

Code examples are all intentionally written in a Java style. Scala code in the wild will often look quite a bit different.

I don't expect you to know Scala at the end of this presentation. I hope to make you interested enough in Scala to learn it.

I dislike tech presentations that pretend like a technology has no tradeoffs or downsides. You're going to hear many downsides. This doesn't mean I hate Scala.

When I say “This Scala concepts are roughly equal”, I mean “the concepts are close enough for the purposes of giving someone that wants to know the basics of a PL in a one hour session an idea of what I mean”.

# What is Scala?

- Hybrid Functional and Object Oriented Language
- Primarily targets the JVM, although ScalaJS exists.
- The most popular language on the JVM other than Java.\*
- Strongly Typed
- Everything is an expression.

\* According to some language metrics, Kotlin has recently surpassed Scala in popularity. However, Scala's userbase is almost entirely on the JVM, whereas Kotlin is split between DalvikVM and the JVM. <http://pypl.github.io/PYPL.html>

# Scala Philosophy (The Optimist's Perspective)

Your programming language should never limit you.

*"With great power comes great responsibility"* - Martin Odersky probably





# Scala Philosophy (The Pessimist's Perspective)

Pure functional or heavy use of mutable state and side effects?

Semicolons or inferred line ends?

Parens or brackets?

Spaces or dots?

Inferred types or explicit?

Implicit parameters or explicit?

Possibly Coming Soon: Python style significant whitespace vs. brackets?



# Let's Walk Through a Toy Scala Project

## Demo Time

(Don't worry if you're a person from the future, reading through these slides after my presentation. All code that I plan on walking through is available in the next few slides)

# Scala Case Classes

```
case class Farm(name: String, owner: String)
```

Similar to Record types available in Java14 preview. <https://openjdk.java.net/jeps/359>

The current closest thing in Java doesn't come close to fitting on a slide. Think about a simple DTO that has the following characteristics.

- Constructor
- All fields final / immutable
- Equals, hashCode, and toString overrides based on all values in the constructor
- Builder pattern implementation (the copy method in Scala)

# Scala Trait - You probably know this as an interface

```
trait FarmDAO {  
  |   def save(farm: Farm): Try[SaveResult]  
  |  
}
```

Similar to Java 8 interfaces with default methods.


Allows for multiple inheritance without the diamond problem due to strict rules about inheritance order.

As of Scala 2.12, traits compile directly into Java interfaces

# Scala classes vs. Java Classes

Scala Class  $\simeq$  Java class with no static methods

```
class PostgresFarmDAO(implicit sf: SessionFactory) extends FarmDAO {  
  def save(farm: Farm): Try[SaveResult] = withSession { session =>  
    session.save(farm)  
  }  
}
```



These are  
equivalent

withSession(\_.save(farm))

# The Loan Pattern

```
def withSession[T](f: Session => T)(implicit sf: SessionFactory): T = {  
  val s = sf.getSession()  
  try {  
    f(s)  
  } finally {  
    s.close()  
  }  
}
```

More advanced, but extremely powerful method of reducing boilerplate when you need an AOP style of approach.

# Sealed Algebraic Data Types (ADTs)

```
sealed trait ValidationResult  
case object Valid extends ValidationResult  
case object Invalid extends ValidationResult
```

A sealed trait is only allowed to be extended within the same file.

- Allows the compiler to enforce exhaustiveness checks during pattern matching.

The closest Java equivalent is enumerations, which are not nearly as powerful.

# Scala Object

```
object ValidationResult {  
  implicit def toValidationResult(b: Boolean): ValidationResult = {  
    if(b) Valid  
    else Invalid  
  }  
}
```

Singleton instance of a class similar to a Java class with the following characteristics.

- Finalized
- Private no-args constructor
- Only static methods and values.



# For Comprehensions

```
for {  
  validity <- Try(FarmValidation(farm))  
  saveResult <- farmDao.save(farm) if(isValid(validity))  
} yield saveResult
```

Allows iterating over multiple monadic types to build up some larger computation.

- Has “guards” that can prevent moving to the next step of the computation.
- Compiles to nested flatMap calls.
- If you go down the “purely functional” scala path, you’ll use this everywhere.

# Pattern matching

```
def isValid(validationResult: ValidationResult) = validationResult match {  
  case Valid => true  
  case Invalid => false  
}
```

A switch statement on steroids.

Adds ability to destructure product types (case classes and objects).

With certain compiler options, can enforce exhaustivity checks.

# The ??? operator

```
val flip: Int = ???  
val flop: String = ???  
val floo = ???
```

- Always typechecks
- Useful for iterative development using a *wishful thinking* workflow.
- Will throw a `NotImplementedException` at runtime if you don't implement it before shipping.

**WARNING: The next few slides are purely opinions. There is no objective data to back them up.**

# Things I've noticed Most Java Devs Find Weird

# Checked Exceptions

Scala has no “Checked Exceptions”, and in general Scala devs avoid throwing Exceptions

- Return a response wrapped in `Either` or `Try` when failure is expected and should be handled by callers.

# Null-checks

Defensive programming that you often see in Java projects are seen as a code smell in Scala.

- Use `Option` instead of null-checks and let the compiler decide whether a value can be null.

# Mutability

- Prefer ``val`` instead of ``var``
- Use the immutable collections library.

The Scala Coursera course is a great place to learn about how to structure your code in a way that makes programming without mutability possible.



# Dependency Injection

Dependency Injection frameworks are often discouraged.

- Just use constructor injection. Potentially use implicits to decrease the boilerplate, but not necessary.
- Dick Wall felt *constructor-injection* wasn't a marketable enough name, so he started calling this the *parfait-pattern*.
- If you really feel the need to use a DI framework, I've mostly seen people using *macwire* or *guice*.

# Ex Nihilo Nihil Fit

Scala originators are nihilists, and have many different terms with subtle differences that sound similar to *null*.

- None
- Nothing
- Null
- null
- Nil
- Unit

## What Scala is best at (in my experience)

Backend Long Running Web Services.

Batch ETL processing with Spark.

Not mucking with your global dev environment  
(Maven and gradle users will take this for granted,  
but Python users will respect this.

# Things Scala is not-so-great at

- Code reuse between a web front-end and back-end. ScalaJS exists, but the cross-compilation experience is rough compared to the experience with Node.
- It has all the long-start-up woes as Java, which hampers using it for short running command line tools.
- Exploratory Data Analysis is somewhat of a pain, but is rapidly improving. It currently is not nearly as viable as using Python with Jupyter Notebooks or R.
- Game programming in Scala is possible through libgdx, but is a worse experience than Java since most of the tutorials and documentation are written in Java.

# My personal opinion on Scala teams

If you have a team that wants a more concise & expressive version of Java, Scala is a valid option. It serves all of the exact same use cases as Java well.

If you have a team that wants pure FP like Haskell, but with the library ecosystem of Java, Scala is a valid option, albeit more verbose.

If you have a team that writes a mix of these two styles in Scala within the same codebase, consider finding a new team.

# What a Scala team needs to succeed

- Willing to invest in training for new members.
- Mandate adherence to a style guide. Use tooling to enforce it.
- Limit technological diversity (IOW: Pick a JSON framework and under no circumstances do you add a dependency on another one)
- Macros and compiler plugins are neat, but ban them anyway at least until Scala 3 gets here.



**Why you should learn Scala, or some other programming language this year whether you'll use it at your day job or not**

**Disclaimer: All of these points equally apply to Clojure.**

**Come back in May for Alex Miller's Clojure talk if I have failed to convince you that Scala is pretty darn cool.**



Immutability is kinda awesome, and Scala makes it easy.

The expressivity (high business logic to boilerplate ratio) enables programming styles that are useful in Java, but no one uses them because of the verbosity.

It will make you a better Java programmer.

# Alright, you've convinced me. Now where do I go from here?

[Scala Exercises](#) - Simple web based interface to learn the basics of Scala and various important libraries in the Scala ecosystem.

[Udemy - Scala Applied I](#) - A good cheap introductory Scala course.

[Coursera - Functional Programming Principles in Scala](#) - Taught by Martin Odersky himself. Some report that this is tough if you're brand new to both Scala and FP, so consider a few days of familiarizing yourself with basic syntax before jumping in the deep end.

[Underscore.io](#) - Great PWYW Scala books. I found their book on Slick to be more useful than the official documentation.

But the rest of the very few who could understand the book were less enthusiastic!

THEY ARE SAYING THAT, DESPITE OUR HUNDREDS OF PAGES OF SYMBOLIC CALCULATIONS, WE'VE **NOT** MADE THE FOUNDATIONS ANY LESS SHAKY.

ACH! THEY ARE SUCH **BLOODY FOOLS!**



THE GIST OF IT IS THAT THE **PREMISES** OF THE THEORY OF TYPES DON'T GO DOWN WELL...

...JUST AS I'D FEARED!

BUT DON'T THEY UNDERSTAND THE **SIGNIFICANCE** OF TYPES?



THEY ARE OUR **SAFEGUARD** AGAINST PARADOX, THEY ARE ESSENTIAL TO LOGIC ITSELF! TYPES MUST BE SALVAGED...

...AT ALL COSTS!



# Thank You!

Questions?