SCALA 2013: A PRAGMATIC GUIDE TO SCALA ADOPTION IN YOUR JAVA ORGANIZATION

including 15 pages of Q/A with Scala founder Martin Odersky

REBELLABS

INTRODUCTION

Scala is a language created by Martin Odersky and his desire to combine object-oriented and functional programming on the Java Virtual Machine. Odersky's previous work on Pizza, GJ, Java generics, the javac compiler and Funnel eventually led to the creation of Scala and its first release in 2003 at EFPL. Scala is purely object-oriented as every value is an object, and functional as every function is a value. The successful mix of OOP and FP paradigms has made the language relatively popular and a "better Java" for many. It's also a good language for those looking for Haskell-like capabilities on the JVM.

Following the language's rise in popularity, Typesafe was founded in 2011 by the creators of Scala and the Akka middleware, to provide easy to use packaging of both in the form of the Typesafe Stack, as well as better development tools and commercial support.

In the last few years there has been some criticism of Scala; it's mostly been accused of being too complex. In this report, we opine that the perceived complexity of Scala actually arises more from the diversity of the ecosystem rather than the language itself. We then look at a simpler subset of Scala that is perhaps a good place to start for Java teams who are considering adopting it in 2013 or later.

We also include an interview with Martin Odersky, creator of Scala, Chairman and Chief Architect at Typesafe and throughout the report we have comments from Josh Suereth, also from Typesafe, and author of Scala in Depth.

Stay close to the shallow end?

Numerous blog posts during the past year or two have accused Scala of being too complex or highlighting other issues with Scala adoption. Some of the criticism is fair, but some of it could be targeting almost any JVM language out there. Scala seems to get more of the flak because it's more popular and possibly pulling ahead of the others.

And also probably because Scala actually is rather complex, but only if you want it to be.
Think of Scala as a swimming pool - there is a deep end and a shallow end; it seems that many people who haven't learned to swim yet jump straight into the deep end. Inevitably, after splashing about for a while are surprised to find that their legs don't quite reach the floor, or that they sank to the bottom and are startled by not being able to breathe underwater.

JOSHSUERETH:
This is an important point. Scala is not just a "Java++". There's a lot of depth to pull out of the language, and lots you can learn. It doesn't take a lot to start, but it can be intimidating how much is out there. You need to be prepared for it to take some time to learn things in the Scala community. Remember that Scala has been around for many years, and a good portion of its community has been around and experimenting for 5+ years. Don't get flustered if not everything makes sense. Wade into the pool and get comfortable before going deeper.

Why does this happen? Maybe it is hard to distinguish the deep end from the shallow end - it's not quite as simple as having a glance and getting a clear picture of how deep the pool is. Maybe the deep end seems more exciting at first, when you haven't gotten a true taste of it yet.

Why talk about such a dichotomy in the first place? Because the Scala community is very diverse. It can cater both to newbies and expert type-safe library designers. Some Scala users come from an object-oriented programming background (e.g. Java), some come from a functional programming background (e.g. Haskell).

Scala blends together these programming styles, but still lets you code mostly in an “impure” imperative or mostly in a “pure” functional way. And that might be the main problem - the very different coding styles enabled by Scala means that a lot of code written by people from a school outside of one’s own might seem downright illegible.

JOSHSUERETH:
The key here is that team productivity is important. Any language can be introduced in a crippling fashion, regardless of its merits. The same is true with Scala. When working with a team, getting everyone on board and learning is crucial.

Each Scala team needs to find a common style that they agree upon, or the result will be a mess. The Scala Style Guide might help with this.

This piece is mostly written for those coming from the Java (object-oriented / imperative) school.
Some of the suggestions are very much a matter of taste, background and so on, so we’re not expecting everyone to agree with it. But let’s get on with some thoughts, opinions and comments...

WHAT TO AVOID IN THE SCALA ECOSYSTEM

There are some things in the Scala ecosystem that will seem like a foreign language to someone coming from Java. They may be fun to mess with and may actually be useful, but if you are trying to adopt Scala in a team of Java programmers, these things might work against you; such as finding yourself inadvertently in the deep end. We think the following parts of the Scala ecosystem are better avoided when just starting out:

1. **SBT – Simple Build Tool**
2. **Scalaz**
3. **Category theory**
4. **Libraries that overuse “operator overloading”**
5. **Collections Library Source Code**
6. **The Cake Pattern**
7. **The Kingdom of Verbs**
8. **Too Functional or Too Type Safe Code**

1. **Avoid** the SBT

The Simple Build Tool (SBT), is perhaps not so simple after all and if you use it, your build descriptors will be a mishmash of a DSL that uses lots of special symbols, poorly structured meta-data, and code. For example, to turn a project into a web project, it was necessary to add this cryptic plug-in dependency in plugins.sbt:

```
libraryDependencies <+= sbtVersion(v => "com.github.siasia" %% "xsbt-web-plugin" % (v+"-0.2.9"))
```

And then this magic line to build.sbt:

```
seq(webSettings :_*)
```

If this seems like black syntax magic to you, just stick with Maven, Ant, or whatever build tool you used before that has Scala support.

Sometimes you might be more or less forced to use SBT (e.g. if you use Play 2.0). But don't take it too badly — SBT still has some good ideas inside it. There is hope that it will get better and better, and even support a Maven mode. But at the moment, it leaves much to be desired.

JOSHSUERETH:
I know very well that SBT has a high learning curve. The very simple "cut and paste" builds work well, but anything in the intermediate range can take a while to get up to speed. If you're a beginner this can be a cause of frustration and despair. When starting, it's perhaps best to avoid these things.

However, I'd like to state that SBT is by far my favorite build tool. I've had to customize many Maven plugins for past builds, work with quite a few ANT builds, even used Automake, cmake, make and friends. I'm sold 100% on SBT. If we can get this learning curve down to acceptable levels, I believe everyone will see the benefits. I'm even going so far as to write a book on SBT covering Introductory -> Advanced usage.

Whenever you have a complex build environment, SBT really makes repeatable builds simple. We're working furiously to improve that middle ground/learning curve. I think it's a project to pay attention to!

2. **Avoid** Scalaz

If you are thinking about using Scalaz, stop while you still have your sanity! And don't get into arguments with people who imply that you are dumb for not using it; these strangers come from a completely different school of thought that values entirely different things – pure functional programming with very high-level abstractions represented by obscure symbols like <:::, :::>, <+>, ★, ☆, <=<, and so on – there are HUNDREDS of these.
Also, run away if you hear anyone casually utter words like: applicative, bifunctor, cokleisli, endofunctor, kleisli, semigroup, monad, monoid. Ok, some of them are maybe not so scary, but what I'm getting at is that you should also stay away from #3...

3. **Avoid category** theory

Category theory is an area of mathematics full of abstractions that apply to programming in interesting and sometimes useful ways, but it will not immediately help you write better programs that are easy to understand. Eventually, you might find useful things there, but maybe it's better to stay clear of category theory until you are really really sure that you and your team want to go there and will benefit from it (the same goes for Scalaz, above).

4. **Avoid libraries that overuse** "operator overloading"

Scala doesn't have real operator overloading, as most operators are actually methods. But it allows you to use many symbols in method names. This is not bad, because it is useful in places, such as mathematical code, but some libraries take "advantage" of that feature and use easy-to-type symbols (or even not-so-easy-to-type symbols) to mean something entirely different than what those symbols commonly represent.

Again, Scalaz is the main culprit here, but there are many other libraries whose authors have been a bit overly happy-go-lucky with their use of symbols in all the wrong places. SBT errs here as well, as already mentioned. Some examples:

- Periodic table of dispatch operators
- Using parser combinators to extract SQL query results (scroll down to "Using the Parser API")
- Applicative example in Scalaz

5. **Avoid the Collections** Library source code

Don't dig too deep into the collections library implementation. It is a breeze to use the library mostly, for example:

```
val (minors, adults) = people partition (_.age < 18)
```

But the implementation is way more complex than we're used to in Java-land. The internal complexity pays for power, though, and is probably worth it. If you get stuck, the sources might be closer at hand than docs, but try reading some documentation anyway, before peeking into the source.

6. **Avoid the** Cake Pattern

Don't believe people who tell you that Scala doesn't need dependency injection frameworks because of the cake pattern - thankfully, not many people would go quite that far. Not all projects need DI, but where it has worked in Java projects, it can work just as well in Scala projects.

The cake pattern may suit many projects as well, but it makes the code structure unnecessarily more complex, and may force you to use some of the more advanced features of the type system that you could avoid otherwise.

> **JOSH**SUERETH:
> One of the biggest issues with the Cake pattern, is that even when you know the cake pattern, the cost of learning a new code base is high. When a team is spun up on the Cake, things become much easier to digest and work. It's certainly something to investigate for your team, but is probably best avoided when first coming to Scala.

7. **Avoid getting stuck in** the Kingdom of Verbs

If Java is the Kingdom of Nouns, then Scala code can sometimes look like it's coming from the Kingdom of Verbs. You might see code like this:

```
Project.evaluateTask(/*args skipped*/).get.toEither.right.get.map(_.data.
toURI.toURL).toArray
```

To be honest, such stringing of methods can happen in Java as well, but seems less likely there. Not everything in the above snippet is technically a verb, but there is a very long chain of methods with none of the intermediate results being named.

If you look at this code without an IDE, it is nearly impossible to understand what is going on because you know neither the types nor the names of the things. Thank god point-free style doesn't work in Scala, or it could be worse.

Just give names to some of the intermediary results, eh? In other words, your Scala code should be from the Kingdom of Sentences, where both verbs and nouns are in equal standing. It is often possible to even mimic simple natural language sentences without creating a full-blown DSL.

8. Avoid code that is either too functional, or too type safe

This is perhaps our most controversial point, but try not to go too much into the functional side of the language or to achieve maximal type safety. Scala can clearly give you better type safety than Java in many cases, but there is a limit to everything. The type system is complex and if you dive very deep into it, you might just spend a lot of time satisfying the compiler for little gain instead of writing useful code. It's fine to occasionally play with something like shapeless, but it might be too experimental to use in code your team is going to get paid to maintain.

JOSHSUERETH:

One mistake many people make when coming to Scala is preserving types unnecessarily. Since Scala promotes Polymorphism and FP at the same time, it's quite possible that you don't need to preserve a type if you're not going to return it in a function. When people see the possibilities of type-safety, sometimes they take their code too far, or add types which don't add value.

When designing a library and using any feature, you need to ask the question: "Does this addition hold its weight?". A good example of this is the *Like classes in Scala's collections. These are used to preserve specific types when using generic methods. For example, calling the "map" method on a Vector will return a Vector. There's a bit of mechanical type jiggering to make this happen. Not the kind of code you show to someone new to Scala, but the usage of the library remains simple and easy to understand. These are trade-offs and judgement calls that can take time to learn.

The same is with going too functional. Often it makes sense to use flatMap or map over Option values instead of if-else expressions. But writing a long chain of flatMaps without naming any intermediate results is not a good style.

Overusing higher-order functions in hot code may create performance problems. In some cases, you might even have to revert to more Java-like code with mutable variables and while loops — Scala's for loops are compiled to higher-order function calls, but while loops are low-level as in Java or C. This should be improved in Scala 2.10, which adds some new for-loop optimizations.

Still, functional programming is getting more and more popular for a reason. Concurrency and correctness might be easier to achieve with more functional code. So don't stay completely on the imperative side of Scala either, or you might not enjoy as many of the benefits as you could.

Any language has some annoying warts — you just learn to live with them. Decide whether you want to use the more complex features only after becoming somewhat comfortable with the language. Or maybe you already are on good terms with some of them — we're not trying to change your mind about that. If not, though, keep in mind that these things are also part of the Scala community in it's diversity, and ignoring them completely might not work as well when you find that, for example, some of the Scala answers on StackOverflow or on the mailing lists are of the opposite mindset.

THE THINGS YOU MUST LIVE WITH

Ok, we got the things to avoid out of the way — if you discover these as you're learning, thinking that's the "right way" to develop in Scala, they might scare you off, and you would be missing out. In Scala, there is no single "right way" to do things, for better or for worse.

Now we'll look at some more things that are not quite perfect in Scala land, but that you must simply live with, at least for now. For example, the existence of the things to avoid, listed above -- you should be able to coexist with the "deep end" of Scala while staying in the shallow end yourself, and still write good code that is not too foreign even to Java developers.

There are many libraries in Java that are internally divided into so many layers of abstraction that they are extremely hard to debug or understand, but that doesn't take away from the value of Java itself. The same is true for Scala, but as a newer and less popular language, there just isn't a comparable number of libraries that are really solid.

Comparatively, Scala is still in the early days where Java would've had EJB 2 and Struts 1 as the pinnacle of web development. The language is still that young, but perhaps this comparison isn't quite right — Scala probably never had as big of a blunder as EJB 2, and web development with Scala is not necessarily behind or ahead of web development with Java.
But there's still a lot of exploration and mapping of the territory going on. In the Java world, most people would not blame the Java language for EJB 2. They would just stay clear of that single part of it. But with Scala, some folks are quick to blame the language or the community and ecosystem as a whole for any particular misgivings.

And the Java world still hasn't reached a consensus on the eternal question: which web framework to use? And it never will, because no single framework is particularly amazing, for one reason or another. Different variables weigh into that choice each time.

So it is with Scala, that there aren't always perfect solutions and evolution is still happening -- and at a faster pace than in Java. Lets look at some of the unavoidable issues we have to deal with.

1. **Binary Compatibility**
2. **A good IDE is pretty much a requirement for understanding some code**
3. **Compiler speed in conjunction with IDE use patterns can slow you down**
4. **Scala is still evolving and getting new experimental features**

1. **Binary** compatibility

This is perhaps the biggest issue with Scala. Libraries compiled with one version will not necessarily work with the next or previous versions, so they need to be recompiled with each version upgrade. SBT can do this (compile against multiple Scala versions), and that is perhaps one of the main contributors to its relative popularity.

JOSHSUERETH:

This is a big issue, and one that in the Java world has been solved with slowed evolution of the ecosystem. The Scala ecosystem is moving at a high enough velocity that binary compatibility concerns sprout up in many places. We (Typesafe) are doing everything we can to lessen the burden and improve the state of binary compatibility in the ecosystem. However, there's cultures at war here. Those coming from Java that expect things to just work and have few version issues and those in the Scala community that want to rapidly advance their libraries.

Over the past year, we've seen quite a few improvements in this space. I think that trend will continue for some time, especially with our (Typesafe's) attention to the problem.

If some libraries don't recompile, they will hold back users from moving to newer Scala versions, or alternatively force users to use source dependencies or even resort to forking. But things have been getting better even on the binary compatibility front.

2. Need an IDE to understand some code

I personally think you definitely need an IDE for developing Scala code, and maybe even more so for reading code written by others. It is not as easy as in Java to immediately see what a particular line of code does, due to more constructs being available (such as implicit conversions and type inference). An IDE that lets you jump to declarations and shows types when you hover over something with an inferred type is essential in understanding some Scala code.

Thankfully, the Scala IDE for Eclipse is quite good since version 2.0, and the IntelliJ IDEA plug-in might be even better from what we've heard.

“

JOSHSUERETH:
I'm a huge fan of IDEs myself. I'd be lost in C++/Java/Scala/Python/Ruby without them. While I don't think it's true that you absolutely need an IDE, I do think that code discovery with IDEs, especially in strongly typed languages like Scala, is quite amazing. A good IDE can help you learn a code base much quicker than grep and friends. After learning a code base, is when I revert to using text editors, not before. Call me crazy.

”

3. Compiler speed

This is the largest issue that remains with Scala tooling — when you have a project with thousands of lines of code and use Build Automatically in Eclipse, you might end up in situations where changing a single line and saving will result in minutes of waiting. Compiler speed is also something that is being worked on.

4. Scala is still evolving

New features are being proposed for and added to next versions of the language, including value classes, control structure syntax improvements and even compile time meta-programming. These, also known as macros, can seem especially scary. In reality, macros are optionally enabled and not exposed to most users.

Plus, there already are a many features in the language that you shouldn't use without good cause, as mentioned earlier. Library authors would be wise to not jump at the chance of experimenting with every new feature where they could do without. Of course, that doesn't go for everything — some libraries may really benefit from these language evolutions.

There was even a new proposal (SIP-18) for disabling some language features by default, which generated a huge comment thread on the mailing list, but that seemed to be a storm in a teacup and the proposal made it into Scala 2.10. One of the main reasons for the proposal is a longer term direction for Scala that has the potential to unify some of the hairier features of the type system into a simpler set of rules.

CAN WE FIND A BALANCE?

There are some issues everyone in the community must live with when they use Scala. It is not too much, considering the power you are getting from the language.

The Scala community is diverse, perhaps even somewhat divided. Some people with functional programming background seem condescending at times, and probably think that the OO guys don't care enough about writing better code. Many potential Scala programmers with OO backgrounds care about large code bases being maintainable by any able team member or new hire for years to come. And until the functional side has proven to the wider community that programs written in the functional and more abstract way are as maintainable, their voice will not be taken too seriously by the larger group.

JOSHSUERETH:
I think the point here is that a lot of programmers feel strongly about what they believe in. FP has a lot of merits when it comes to reasoning and restructuring code. So much so, that many of us "math" nerds gravitate here, as it feels far more natural and less "chaotic". At the same time, Scala is a blended language. Even when coding in FP, the interpreters/runtimes usually have to muck with the mutable bits. I think Scala offers an interesting solution in that both the "pure" and the "impure" code can live on two sides of a wall, in the same language.

Also, there are some domains which I think lend themselves more naturally towards other models. I personally believe the GUI development, in its current form, is more natural in an OO paradigm, walled off from the "FP" portion of code. That said, I still have high hopes for Functional Reactive Programming.

Actors themselves are inherently not "pure functional". However, I know from experience that the Actor model is very ideally suited, and successful at modelling concurrent processes and backend servers.

What all boils down to, is that in Scala we're exploring many different spaces. You'll always find programmers who feel strongly about their own paradigms and promote them. Healthy discussion and movement is great. I hope we all learn as much as we can from the "functional side", just as we have a lot to learn with the new object-oriented features Scala brings to the table and actor-based designs. Scala is language where you can learn a lot of different things. You'll always find those who promote one way over another. It's a sign of a vibrant, healthy community.

It'll be interesting to see whether the functional/theoretical and OO/pragmatic side can achieve a more symbiotic coexistence and further bridge the functional / OO gap, or whether the groups will diverge even more. In any case, it helps if you know about this division when you start with Scala. Get comfortable with the basic language features, and then decide whether you and/or your team wants to keep avoiding the other side, or start expanding your horizons further.

It's possible that Scala would become an even stronger Java alternative if the community could bring in more pragmatic programmers who would write solid libraries that don't overuse symbols, avoid some of the more complex type system features, maybe even stay more imperative than functional (while still preferring immutability), and don't go overboard with the type safety.

It doesn't mean the more pure/functional/type-theoretical/experimental/research side of Scala should go away — they should all be able to co-exist in some kind of symbiosis. We think this has been happening and will continue.

WHAT MAKES SCALA A GREAT LANGUAGE?

We looked at the challenges of Scala complexity from the perspective of an object-oriented newcomer. Now for some thoughts on what is so great about Scala in the first place, that Java developers should consider using it. A detailed account would take more words than we've space for here, but here's a brief overview with links to relevant reading. This subset of Scala features forms a relatively simple language that is still quite powerful.

More intentional code

-Get rid of most of the boilerplate

DRYer code — repeat yourself less

-No need to write types in every declaration — they are usually inferred

-More abstraction mechanisms means less copy & paste

More concise code

-It's safe to assume 2-3 times less lines of code than equivalent Java.

-Occasionally the difference can be even bigger (or smaller).

Immutability is easier

-Use val (or lazy val) by default instead of var and you might be surprised how seldom you really need mutability

JOSHSUERETH:

Josh Suereth: I think this is a key point pure functional programmers have been screaming at the rest of us. It's often possible to create code which is easy to reason through and retain good performance. Having to track through several different source files to see who modified "var xyz" is nobodies idea of a good time, and we've all been there. Keeping parameters immutable by default is one of Scala's great virtues.

Slightly more functional code

-Make your code more functional without radically changing your current programming style

Functional abstractions over collections

-Multi-line for-loops become one-liners when using lambdas

New ways to compose things

-Traits enable compositions that are impossible in Java

Pattern matching

- A huge improvement over if-else chains or switch statements

Java interoperability

-Your code can easily interact with existing Java code

Good performance

-Code can perform as well as Java, but some abstractions do have their overhead

Scala features that should make you jump for joy

The exact set of favourite language features inevitably varies from person to person, so feel free to let us know if you think there’s something you like that we omitted. But having summed up some of our favourite aspects of Scala, let’s look at the language features that enable these:

1. **Case classes (a.k.a. algebraic data types)**
2. **Pattern matching**
3. **Traits**
4. **Adding methods to existing classes**
5. **Syntax**
6. **No Primitives**
7. **Anonymous and higher-order functions**

1. Case classes (a.k.a. algebraic data types)

It’s amazing how easily you can write small classes for holding data in Scala. Even if you don’t use getters and setters in Java, a public data class is still so much code: it usually needs its own file with imports; equals, hashCode, toString implementations and so on. In Scala, if you need a small data class you can often add it to an existing .scala file and just describe the field names and types:

```
case class Person(firstName: String, lastName: String, age: Int)
```

Case classes like this are immutable by default, like most data classes should be. equals, hashCode and toString are generated by the compiler, based on the combination of the fields.

The compiler even adds a copy method, which uses current values as default arguments. For example, on my birthday, I could execute this code:

```
val newMe = me.copy(age = me.age + 1)
```

Think this is cool? Read more about case classes.

2. **Pattern** matching

Case classes interact quite well with pattern matching, which is like a more generalized switch statement. Given our Person class above, we could do some matching on different persons like this:

```
val aPerson: Person = // find a person
val canDo = aPerson match {
 case Person("Chuck", "Norris", _) =>
   // here we only care about Chuck Norris, and not his age -- Chuck Norris is ageless.
   // Underscore in Scala usually means "wildcard" or "we don't care about it"
   true
 case Person(_, _, age) if (age >= 18) =>
   // matches only adults who are not Chuck Norris
   true
 case _ =>
   // matches anything else. this is like the default case in Java's switch
   // if you leave it out you will get a MatchError for a non-matching object
   false
}
```

Like most control structures in Scala, but unlike those in Java, pattern matching is an expression — it can compute a value, which in our case is a Boolean that is true for Chuck Norris and adults, but false for everyone else. Only one of the cases is executed, there is no fall through and there are no break instructions needed.

Pattern matching can also be a good replacement for the Visitor pattern. Read more about pattern matching.

3. Traits

Traits can be used exactly like interfaces in Java, but they can also contain implementation code. If you have a complex type hierarchy, they make it easy to mix several aspects of the implementation from separate traits into concrete classes.

For example, let's introduce an Ordered trait that defines < and > methods for comparing objects. Implementations must only implement a compareTo method.

```
trait Ordered[T] {
 def compareTo(other: T): Int // abstract method, returns 0, negative or positive
 def >(other: T) = (this compareTo other) > 0
 def <(other: T) = (this compareTo other) < 0
}

// we can mix Ordered into an existing class, implement compareTo and get < and > for free!
val ChuckNorris = new Person("Chuck", "Norris", 0) with Ordered[Person] {
 def compareTo(other: Person) = 1 // Chuck Norris is always greater than another person
}

ChuckNorris > Person("John", "Doe", 22) == true
```

Of course, this small example isn't perfect as it only gives us ChuckNorris > person but not person < ChuckNorris. To get both, we'd change the Person class:

```
case class Person(/*...*/) extends Ordered[Person] {
 def compareTo(other: Person) = /* implement ordering for all persons */
}
```

Read more about traits.

4. **Adding methods** to existing classes

You can, in a way, add methods to existing classes without changing their source code. This is useful for various things, and is achieved by implicitly converting an object of one class to an object of a wrapper class. The methods are defined on the wrapper.

For example, if we wanted to add a method that says whether a person is minor or an adult, we could do this (note that this is Scala 2.10 syntax):

```scala
implicit class PersonWrapper(val p: Person) extends AnyVal {
 def minor = p.age < 18
 def adult = !minor
}

Person(“John”, “Smith”, 43).adult == true
```

It would perhaps be better if “extension methods” were implemented independently of implicits. Implicit conversions hold some dangers and are easy to abuse, so you just have to remember to use them responsibly. Read about some hints here.

5. Syntax

You’ve seen some of the syntax in the examples above. It might seem odd at first, with the types coming after names in declarations, but it is quite good once you get used to that change. It’s a nice concise syntax and def compareTo(other: T): Int appears more readable to me than public int compareTo(T other). As we saw above, one-liner methods are much nicer than Java equivalents, and infix method calls let us drop the dots and parentheses where we want to (ChuckNorris > person is equivalent to ChuckNorris.>(person)).

Generally, the syntax lets you do more with less, compared to Java.

6. **No** Primitives

Scala hides the JVM primitive types from you. Everything is an object, everything can have methods. The compiler will still decide to use primitives where it can, and boxes values where it can't, but all you see are objects. For example, the Scala library adds a the method to Int, such that 1 to 100 produces a Range you can iterate over:

```
for (i <- 1 to 100) {
 // do something a 100 times
}
```

7. **Anonymous and** higher-order functions

These barely need a mention. No modern language should exclude anonymous or higher-order functions, and even Java will get them soon. They allow easier parallelization, creating custom control structures and concise code for dealing with collections, among other things.

For example, lets say we often need to do something with all the adults in a collection of Persons. We could create a higher-order method that allows mimicking control structure syntax:

```
def foreachAdultIn(persons: Iterable[Person])(thunk: Person => Unit) =
 persons filter { p => p.adult } foreach { p => thunk(p) }

// now we can write
foreachAdultIn(listOfPersons) { person =>
 // do something with person
}
```

CONCLUSION LOOK BEFORE YOU JUMP IN!

Generally speaking, the benefits of Scala have the potential to outweigh the disadvantages, if you know which side of the pool you're diving into....you don't want to hit your head on the bottom after the first jump!

The subset of Scala outlined above already makes the language a compelling choice and should be enough to let you write more concise and intentional code than in Java, but getting there is more challenging. The deeper parts of the Scala swimming pool are sometimes fun to explore, but can be scary at first. Once comfortable with the simpler part, you can consider taking advantage of advanced features and going further in the pure functional direction.

For questions or comments, please reach us at labs@zeroturnaround.com

INTERLUDE

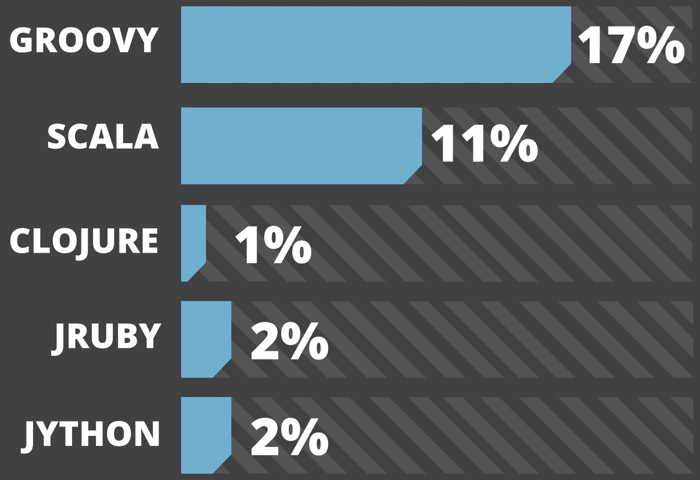
So that was an opinion on adopting Scala in a team of Java programmers, while avoiding the hairier parts. Josh Suereth provided some additional comments based on his own experiences.

Next up, we have an interview with Martin Odersky, founder of Scala. We ask for his opinion on some of the newer JVM languages out there; ask why is Akka important; where are Scala and Typesafe heading; find out how popular Scala actually is and why is it misrepresented in the TIOBE index. For the latter, our own developer productivity report showed that 11% of Java users are using Scala to some extent, though that result has some bias.

Alternative JVM Languages

Groovy has been the dynamic JVM language of choice for years and this fact is reflected in the survey results, although we must admit feeling a bit suspicious at the numbers. Groovy is also trying to appeal to fans of static typing with Groovy++, but Scala seems to have established itself as the statically-typed Java alternative on the JVM - a position which newer languages such as Kotlin and Ceylon will find hard to overtake considering Scala's head start.

Clojure, JRuby and Jython remain as rather niche languages, although Python and Ruby are quite popular outside of the JVM. However, considering the plethora of JVM languages out there, Clojure's quick capture of even 1% of the JVM developers is somewhat impressive.



INTERVIEW WITH MARTIN ODERSKY

Oliver - Hi Martin. First of all thanks for taking this time with us. With me is Erkki Lindpere, author of "Scala: Sink or Swim?" and Anton Arhipov, product lead for JRebel.

Martin - Hi guys, nice to speak with you.

Oliver- I guess my first question would be to ask whether you have you seen our Developer Productivity Report 2012 where we showed that 11% of our respondents were using Scala on at least some of their projects.

Martin - I saw that, yes. That was actually quite encouraging. Seems that we are slowly inching up to become a mainstream technology, which is great.

Oliver - Absolutely! Were you surprised at the penetration of Scala use?

Martin - Well, 11% seems high for the overall developer community. I think the explanation is probably that the more innovative programmers would use both Scala and JRebel and that's why there's a higher percentage than among the general population. Among the population at large, I think we are currently somewhere between 1-2% of the Java market overall and still growing quite rapidly, almost a 100% year-over-year. But 11% for your respondents from the survey is of course great, that's where we want to go.

Oliver - In fact, we did realize that this was a concern and there is some early adopter bias that we mention in the report. We suggest that people who are more interested in responding to an in-depth survey of this kind are a bit more focused on checking out new technologies, being more involved in the communities around Java and the ecosystem. And we mention that there is some bias to be expected. We did see some differences between our findings and TIOBE and RedMonk, and your estimate of 2% of Java is also different. Why is it so difficult to get accurate user base information in the industry for things like this?

Martin - I think the RedMonk study is actually better than most, because at least it correlates two different things - GitHub projects and StackOverflow questions. It is of course very biased towards open source, I would say. I imagine that if you are behind the secrecy wall then you don't dare to write a StackOverflow question - well maybe with your private email account, but not with your company account. But I think it's probably the best we have.

Since this interview, both Scala and Haskell have risen one point on the RedMonk survey: http://redmonk.com/sogrady/2012/09/12/language-rankings-9-12/

It could be that there is just not that much serious work being done coming up with a good survey. For instance, TIOBE is actually very simplistic. We did a study, because Scala was doing pretty good around 2008 - we knew we were climbing - but hasn't progressed at all since then. So something must have gone wrong and, ironically, what went wrong was that I published a book called „Programming in Scala" that led to a lot of search terms (we tracked that) that said "Programming in Scala". Whereas in other languages you typically say e.g. "Java programming", so it's the other way around.

It so happens that TIOBE searches for "language programming" and not "programming in language". If you correct for that bias then there are two languages that do drastically better - one is Scala and the other is Haskell. In Haskell, people also seem to have a bias for saying "programming in Haskell" and not "Haskell programming". All the other languages are more or less the same. That explains at least a large part of why we are doing so bad in TIOBE - it's just an accident of what search terms people use.

Oliver – That's interesting. Hopefully we can start to maybe reverse that trend a little bit because it seems misrepresentative...we'll say "Scala Programming" from now on. Well, I'm glad to get your feedback on that and help us to understand a little bit better. We know that research projects are always difficult in this way.

Let's change pace a little and touch briefly on the Scala series that Erkki wrote and I co-edited, called "Scala: Sink or Swim?". It was a three-part series, which you actually commented on and said there was some good analysis in it. Do you recollect exactly which parts you thought would be most useful for new Scala coders?

Martin – From the start, Scala is a wide tent because it lets you program in what is essentially Java style without semicolons and it lets you do the most advanced parts of functional programming if you so desire. The first thing to know is that being more abstract and more functional is often better but not always. There is no better or worse here. For some problems and themes, a Java-like style is actually very appropriate; for other problem areas, you want to be more abstract. The second observation is that if you start with Scala be prepared that you will make a journey.

It is very easy to pick it up - it only takes a couple weeks to be fluent in Scala. But then it takes much longer to actually do the transition to becoming more abstract - to write more higher-order functions, use functional collections to a larger degree. Ideally learn the limits - being more functional is not always the right thing to do.

For instance, it might be that if you have a very performance-sensitive computation you might rather use a mutable map because it conserves your memory better and it may be faster. The real wisdom in all that is to essentially know when to use functional, because it makes your code better to maintain, and know when to use mutable, because it's maybe more efficient for the problem domain at hand. It's about judging the trade-offs correctly. That's why I'm very much against the people who say that more functional is always better. There is always a trade-off and you have to make the trade-off right. And you have to make it for a team, because in the end having more choice means having more decisions to make.

Erkki – Martin, I wanted to ask was there anything you really disagreed with in the posts?

Martin – No, there was nothing that really got on my nerves. It was all fairly reasonable. I didn't agree 100% with you but it was overall a very reasonable approach and opinion to have.

Oliver – I think it's the best compliment you will get from anyone for a while, Erkki! Martin, I imagine people ask you about the future of Scala a lot and I am not going to do that, but I will ask you what is your favorite answer to give them?

Martin – Hehe. The answer I usually give is that the challenge we all face is essentially how to cope better with parallelism that's on us with multicores and clusters and the challenges of distributed computing, including mobility. Now you have always a bunch of computations that go along together - sequential computations are rapidly vanishing. The main challenge is really finding the right abstraction mechanisms to deal with it and functional programming can play a very good role, though it can't be the whole solution space. You also need to have right concurrency abstractions and so on. That's where we see a future for Scala, because Scala brings to the table a lot of important mechanisms and concepts that let you program concurrent and parallel systems in a way that is understandable for humans, where you don't drown in the accidental complexity given by the wrong model or the wrong concurrency primitives.

Oliver – Wow, I hope you guys understood that :-) On a slightly related note... What do you think about some of the newer JVM languages like Kotlin, Ceylon, Gosu... there seems to be a clear influence on them by Scala. Do you have any thoughts on that?

Martin – I'm always ready to welcome innovation and change on the JVM. The more developments we have, the better. We all want to overcome stasis - the chapter of programming is closed where we thought we all could write programs in the same way we did in the past forever. If we can break that up by many different means, that's good.

Languages like Clojure and JRuby, some of the more mature ones, also have done very important things there. It's interesting to watch the languages, and what they do. Some of the innovative stuff is very interesting. Some of them are still in the early phase so we have to watch how it turns out. I know that many take Scala as the basis, they want to create something just as good but simpler, that seems to be the general consensus.

Essentially when you start out - and when we started out with Scala it was the exactly the same thing - we wanted to do something at least as good as the incumbent but simpler. Often the realities of the environment force you to do solutions that maybe in the end are not quite that simple. That's something that probably everyone will find out sooner or later. I certainly want to be in a friendly competition with the languages to see in the end what's simplest overall....

My intention with Scala is very much to make it simpler in the sense to make it regular and not throw a lot of language features at it. That's a misconception people often have, that Scala is a huge language with a lot of features. Even though that's not generally true. It's actually a fairly small language - it will be much smaller than Java by the time Java 8 is out. The difference is that since it gives you very powerful abstraction mechanisms. You see a lot of features in libraries that get mistaken for language features. That's why we sometimes get a mistaken reputation for being a very large and complex language. Which we are not.

Oliver – I'm glad we could clear that up. Erkki, I think you've got a couple little more direct technical inquiries for Martin. Would you like to ask a few questions now?

Erkki – Yes. Since we got into the complexity thing... Earlier in 2012, there was a really heated discussion in the mailing lists about SIP-18 (Scala Improvement Proposal) - the one that makes some of the language features optional. How's it going with that?

Martin – It's in, it's accepted. It will ship with Scala 2.10. The discussion has pretty much died down because it turned out to not be very onerous to comply with SIP-18.

I think it was a bit of a storm in a teacup, and now it seems to have calmed. I still think it's the right thing to do precisely because by design Scala is a very orthogonal and expressive language, so teams using Scala have a large number of choices available. Some of them need a little bit more guidance with some of the areas that we know are difficult; inevitably, there are hidden problems in some feature interactions, and power can be easily misused. Scala's philosophy is to be as general as possible in its language design. The import mechanism was invented to counter-balance that in the interest of software engineering concerns.

I think it's good to put some of the more easily misused combinations under an import flag. One example – Scala has this very powerful notion of implicit conversions, which I believe is the right thing to do because it lets us cleanly interoperate with Java code and do other things which in the end lead to code that is much simpler for the end user.

But we also found that because of their very power and because they look easy when you write them down, there's a tendency to overuse them. Afterwards If you have too many conversions they might start to be in conflict with each other, and you get surprises. We decided to put implicit conversions under an import flag so you have to demand them explicitly now and be aware of what you are doing. When I applied the flag to the current Scala compiler itself, I indeed found some misuses.

What happened was there was an implicit conversion which we decided was a bad idea. It was first made available uniformly everywhere in the compiler, and we rejected that a while ago and removed it. But then I discovered that in about 10 instances people had reinstantiated that same conversion locally for their code. Because it's convenient to do so. So now you have essentially the same conversion we tried to remove before and you have code duplication. As a manager who's concerned with code quality, that's the kind of thing you want to discover.

JOSHSUERETH:
One thing that SIP-18 is designed to allow is the addition, deprecation and removal of language features within Scala. Scala is a steadily progressing language, and the core team needs the ability to undo decisions that turned out to be poor ideas. Placing a language feature behind a flag is an excellent deprecation path to replace it with something better or more powerful. That's the goal behind the higher kinded types being behind a flag. In the future we hope a better solution is added to the language, and need to leave door open for improvements. In that same vein, the new feature, macros, are behind a language flag. This represents their new status, and that we're unsure if we've ironed out all the nuances of them. It gives us room to adapt them over the next few releases without locking in. Users who wish to try "experimental" language features are welcome and encouraged to do so. As experimental features stabilize, we can remove the flag from their usage, and the general community can welcome a more stable, solidified feature.

Erkki – Another thing that I'm very interested in but haven't tried myself is the Akka framework. What are the advantages it gives to JVM developers?

Martin – I think Akka is the next generation concurrency runtime, where concurrency means you program typically with message passing actors, and it's transparent to whether it runs on the same computer or whether it runs on the Internet. And it has an excellent story on failure.

So essentially when things fail, an Akka system is extremely resilient because every actor is supervised by another actor. We all know that concurrent programming is very hard. Akka is a set of best practices that have been learned for the most part over the last 20 years with Erlang. Erlang pioneered this actor concept and this supervision concept. It's a set of those best practices paired with a really high performance implementation - Akka is very, very fast.

“

JOSHSUERETH:
In addition, Actors let you orient your exception handling around system components rather than lexical scope. Often it's easier to determine how to reboot your search index, or image server than what to do if the image server throws an exception when called.

The Actor model lets you model recovery at a topological/system level. This shift often makes it easier/possible to figure out what to do on failure. While you can do this with standard exception handling, the "surface area" you have to work with is often large and unstructured. Enter concurrency, and the thread that has the exception may not be the thread that caused the exception.

I believe the actor model pays off the most in fault tolerance because of this. Combined with the great performance numbers, and I think you'll see a lot of enterprise shops and projects migrate towards Actors on the JVM.

”

To give you one example, we have a testing cluster of 192 cores and tested Akka on that - essentially just message throughput of actors and very fast context switches. But we found scalability problems.

The thing scaled nicely up to 7 cores, then it was flat afterwards. We looked at it and couldn't believe it. We suspected the hardware and we had people come in to troubleshoot it, but no, it didn't go beyond 7. In the end we found out that it was a limitation in the fork-join framework - in the scheduler.

We pushed the scheduler so hard that it was caving in and wouldn't get faster anymore. Once we talked to [Java Concurrency expert and Typesafe advisor] Doug Lea about it, Doug said: "Oh yeah, I've suspected that for a long time, so now I have a test case. Excellent!" and we put in the changes and since then it scales up nicely. It just shows that essentially to get there you have to optimize a lot of the other things that haven't been discovered before; you never got to this high number of context switches and message passes that we see now with Akka. So it's a very nice platform.

Anton – Hi Martin, Anton here. I have a question for you regarding a project I recently rediscovered from IBM called X10. What amazed me was that it really reminds me of Scala when I look at the code. With regards to Scala's future - distributed programming and concurrency etc, can you compare somehow to X10? It seems that it's a little bit different approach than what you have taken in Akka, because X10 is more a port of MPI to Java. Can you tell a little bit more in comparison to that one?

Martin – There was certainly an influence from Scala to X10. I think an early X10 prototype was actually written in Scala before they had their own bootstrap compiler. A lot of the syntax and some of the ideas came from Scala. One difference I see is that X10 is essentially a language that has a particular approach to high-performance programming - it has a lot of features in the language that are optimized for NUMA - Non-Uniform Access Memory. Starting with async, a type system with regions and things like that. Whereas Scala is a language that relegates much more things to the libraries.

Akka hasn't influenced Scala the language, it's just a library that builds on top of the JVM concurrency abstractions and uses Scala features and parallel collections. In a sense Scala is a smaller language -- the footprint of the language by itself is not optimized for that space whereas X10 is. That's the main difference. We're very much in the same space, but we do it in the libraries where X10 does in part in the language.

Anton - Lately I noticed that they have made good progress on the project. Because the first time I saw it was like in 2004 or 2005.

Martin - It started as a research language, as a DARPA (Defense Advanced Research Projects Agency) project. There were 3 companies doing hardware and languages together for next-generation supercomputing applications. There was Sun with Fortress, IBM with X10 and Cray with Chapel. Those 3 languages share a common heritage, which is this DARPA research project. IBM has now pushed X10 to be a language that is independently funded. I am not sure who funds it within IBM but it is certainly independent of the DARPA project now.

Oliver – I know we only have a few minutes left, but I have one final question for you. You have been the Chairman and the Chief Architect for Typesafe - where James Gosling and Doug Lea are on the board of directors, I understand - since earlier in 2011. We wanted to know if forming Typesafe has changed anything about your work with Scala, and as a follow up question, what do you feel is the most innovative aspect about Typesafe's vision and goals? Does this relate to Scala or not?

Martin – It has changed two things. Before, we simply designed the language and were not really concerned much with what people did with it. We were glad that people did interesting stuff with it but it wasn't really our concern. We published the language and watched what other people did with it.

Now we are very much in a middleware stack -- we have Akka, we have Play! on top of Akka. And we really have a focus on optimizing the whole stack so that essentially Akka runs well and there is no conflict between what Akka does and what Scala libraries are. We also have a good integration between Play and Akka.

The other thing that happened on Typesafe was that I am now much closer than before to customers that do actually very interesting stuff. Unfortunately most of that stuff I can't talk about yet!

But it's extremely interesting to see what they do with it and what challenging problems they have to solve with it and how we can help them doing that. So that was very interesting to actually get out of the ivory tower and really see what people do with your things and what challenges they face.

About the mission: what we want to do is essentially the next generation middleware that works for concurrency and distribution. Think of it as as J2EE for the next decade. Instead of having the current problem of connecting the database to some webpages, you will now have the problems of dealing concurrency, distributed computing, possibly with the inclusion of mobile computing and you need a flexible middleware for that. That's what Typesafe aims to provide, what we do with Play, Akka and on top of the Scala foundation.

We want to do that for both Scala programmers and also Java programmers. So everything has a Java interface as well.

The next thing we're going to come out with is a database connection layer called Slick, which essentially takes a couple of pages from the books of Microsoft LINQ, a relational approach to databases, rather than an object-relational approach that you see in Hibernate. It's a pure relational approach which we believe is actually very suitable to be integrated in a functional language like Scala. It’s a very good fit for that.

Oliver – It does sound pretty slick!

Martin – Thank you.

Oliver – Well, Martin, it’s been a pleasure to chat with you again. Thanks for your time.

Martin - Thank you guys, talk to you soon.

Rebel Labs is the research & content division of ZeroTurnaround

Contact Us

labs@zeroturnaround.com

Estonia
Ülikooli 2, 5th floor
Tartu, Estonia, 51003
Phone: +372 740 4533

USA
545 Boylston St., 4th flr.
Boston, MA, USA, 02116
Phone: 1(857)277-1199