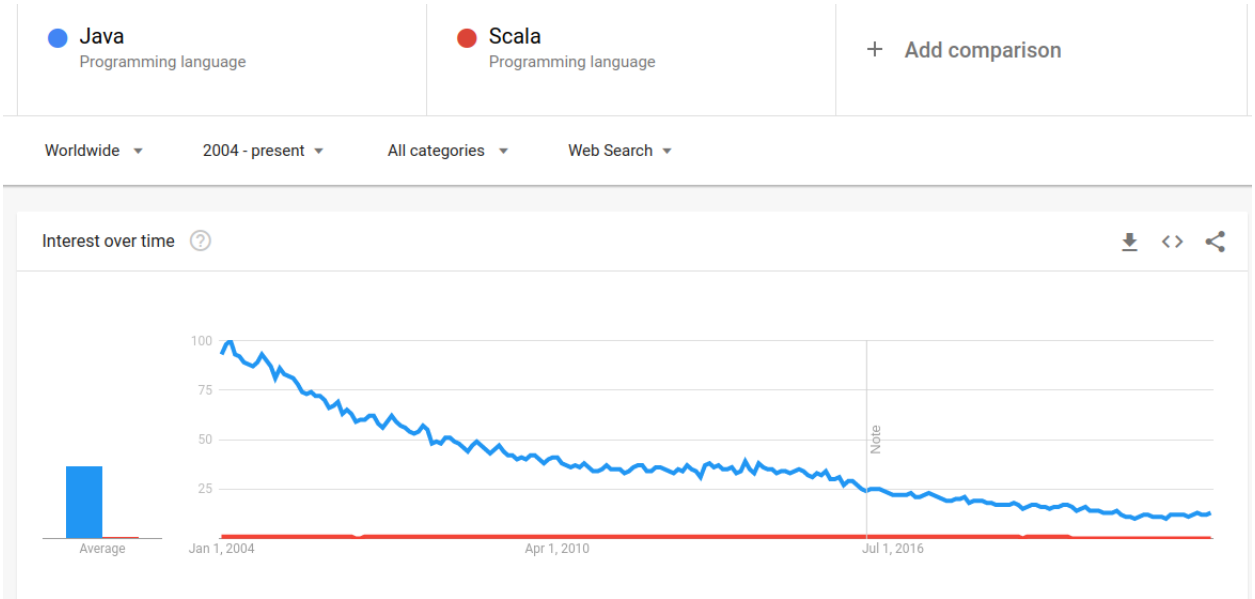
Scala vs Java

by Glenn Engstrand

Which programming language is better suited for enterprise computing, Java or Scala? Can Scala rise above its niche status to overtake Java in terms of adoption one day or will Java continue to maintain its hegemony in this space for the foreseeable future? Should Java developers consider moving to Scala? What would such a move look like?

Both Scala and Java are programming languages that target a runtime environment known as the JVM or Java Virtual Machine. The JVM is a popular choice of server side custom business microservices for the purposes of enterprise computing. Scala code can easily call Java code. Java code can call Scala code unless that Scala code requires any of the exclusively Scala language features to call it. In my experience, it is usually pretty easy to include Java friendly wrappers to just about any Scala library if they don't have that already.

As of the time of this writing, Scala has been around for eighteen years and Java is twenty seven years old. In terms of popularity, Scala peaked about four years ago at position twenty according to TIOBE. It is now at position thirty four. By comparison, Java is currently at position three. Two years ago, it was considered as the most widely used programming language. As of last year, IEEE still ranks Scala at position twenty one and Java at position two.



In this blog, I am sharing some lessons learned in a Scala shop that hires Java developers then trains them up on Scala. The latest version of Scala is 3 which has been out for about a year from the time of this writing; however, the engineers in this company have over a decade's worth of Scala 2 code so the lessons here are about moving from Java 8 to Scala 2. The differences between versions 2 and 3 of Scala would have an impact on these lessons and most probably presents the case that justifies the slight amount of incompatibilities between them.

A lot of these lessons are centered around two programming language themes, modularity and readability. We will show how idiomatic Scala and Java deal with modularity and readability differently in order to handle the design time scalability requirements (a.k.a. code complexity) typical in enterprise level business application development.

Scala can be coded to look more like Java and Java can be coded to look more like Scala. Here the term “idiomatic” denotes how it is generally done in the respective engineering community.

There are profound differences between methods, functions, and lambdas in Scala but for the purposes of this blog, they are treated more or less the same.

First Impressions

There are three programming language features that Java developers first appreciate about Scala when they start learning it. All three of these features are valued because they result in less code to maintain. The three features are case classes, match case, and type inference.

Case classes save you from having to type constructors, getters and possibly setters in your value objects otherwise known as POJOs or Plain Old Java Objects. There are a lot of POJOs in a typical business application. You may think that Project Lombok puts Java on par with Scala in terms of ease of POJO specification but it does not.

The match case expression allows the developer to easily employ pattern matching against those case class generated POJOs. This can significantly reduce the coding necessary to implement complex business rules.

The final feature is type inference which means that the coder usually doesn't have to specify the type of a variable when it is initialized in its declaration.

```
case class Widget(foo: String, bar: String, baz: String)
```

Here is an example of a POJO coded in Scala.


```java
import lombok.*;

@Getter @RequiredArgsConstructor
public class Widget {
    private final @NonNull String foo;
    private final @NonNull String bar;
    private final @NonNull String baz;
}
```

Here is the same POJO coded in Java with Lombok.

These three language features have such a broadly recognized benefit that they have all been somewhat included in more recent versions of Java. First introduced in JDK 14, records serve the same purpose as case classes with some caveats. Switch expressions were added in JDK 12 with pattern matching added in JDK 17 bringing Java sort of on par with match case. The var keyword was added in JDK 10 and provides limited type inference to Java developers. It has been suggested that these new Java language enhancements (and functional programming from Java 8) have been the most predominant cause of Scala's slow decline.

Advanced Features

There are also nine programming language features or patterns in Scala that Java developers, who are new to Scala, may not appreciate immediately but may come to recognize their value eventually. The first three features covered here are the most popular: higher kinded types, monads, and type

classes. By popular I mean that you are most likely to encounter their use when studying real-world idiomatic Scala code. They were inspired by an older programming language called Haskell.

Higher kinded types permit Scala developers to more easily specify nested generics possibly with variance oriented type checking wildcards. You see this feature used a lot in the Cats ecosystem of libraries.	Monads are used to safely wrap the effects of method calls in a consistent manner such that you can always get what you need out of any free monad via its map and flatmap methods. The for comprehension language construct in Scala allows the code to work with monads in a very simple way that ends up calling map and flatmap under the covers. The Zio library is a good example of how to use monads effectively in Scala.

This is not the last time in this blog that you will hear about the for comprehension in Scala. Be advised that this is different from the for loop in either Java or Scala. Looping can occur in a for comprehension but only if any of the monads are collections. Otherwise, the loop is either zero or one iteration.

Type classes give you another form of polymorphism that doesn't directly depend on inheritance. Polymorphism is great at reducing complexity but too much inheritance comes at the cost of increased rigidity. Here is how type classes work. You start with a generic trait that has possibly multiple different implementations. Then you define an implicit variable of that

type initialized with the appropriate implementation. In Scala, there is a language feature known as currying where it looks like you can provide a second set of parameters in parenthesis to a method call. If the curried parameters are implicit, then you don't have to specify them when calling that method in the scope where the matching types were implicitly defined. You can think of this as a way to separate data structures from algorithms or as a lightweight form of dependency injection where you can have circular references along with an immutable state. The Akka streams library uses type classes a lot.

```
Optional[Widget] rv = Optional.empty();
Optional[Foo] foo = getFoo();
if (foo.isPresent()) {
   Optional[Bar] bar = getBar(foo.get());
   if (bar.isPresent()) {
      Optional[Baz] baz = getBaz(bar.get());
      if (baz.isPresent()) {
         rv = Optional.of(new Widget(foo.get(), bar.get(), baz.get()));
      }
   }
}
```

Getting a widget in Java using defensive programming.

The next four patterns are less popular but still somewhat likely for you to run into. These are extension methods, the magnet pattern, DSLs, and shapeless.

```
val rv = for {
   foo ← getFoo()
   bar ← getBar(foo)
   baz ← getBaz(bar)
} yield Widget(foo, bar, baz)
```

Getting the same widget using defensive programming in Scala.

With Scala, you can add new methods to types whose source code you do not control directly. This is known as extension methods or the pimp my library pattern. This is also useful to avoid adding methods to a class that would go against the single responsibility principle. You define a new type that wraps the old type and provides the new method. Then you define an implicit converter that converts the old type to the new type. After that, you can call the new method on objects of the old type. The JSON processing library Circe uses this technique.

Have you ever wanted to overload a method in Java with a different return type? Scala gives you that ability in what is known as the magnet pattern. First you define a sealed trait that serves as the magnet and includes the return type and the apply constructor which returns an object of that type. You define one method that accepts a parameter of the magnet type and returns the magnet return type as applied. The overloaded methods are all defined in implicit converters (first introduced in extension methods above) that convert each different tuple of input parameters to the magnet type with the proper return type and lambda. The bson-scala project from the mongo-java-driver repo employs this pattern.

Scala has a lot of language features that make it flexible enough as a good choice for embedding code written in an internal DSL or Domain Specific Language. First, you design a rich semantic model which includes the nouns and verbs that are specific to the domain. Next, you design a fluent interface for this model using the expression builder pattern. Finally, you implement one or more runners each of which navigates through the expression of the model and does whatever it is that the domain requires of it. Often criticized, the SBT or Scala Build Tool itself is an arguably notorious example of a DSL.

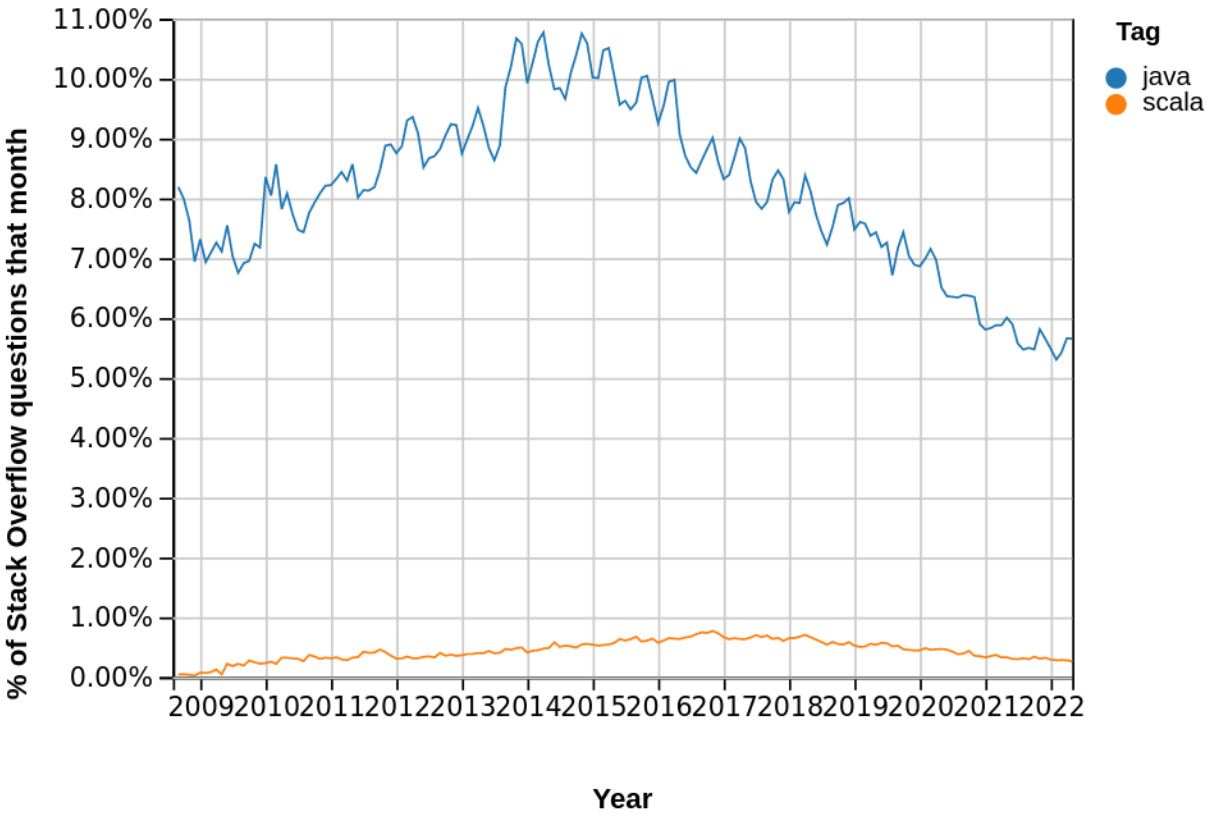
If you are into generic programming where you want to code methods that take parameters of any type but share some structural commonality, then you will want to learn about the shapeless library. It is easy to convert case class objects to or from Hlists which can be passed into methods that specify the structural commonality easily in the parameter list of the method declaration. ETL programs (Extract, Transform, Load) that don't rely on Spark are good candidates for shapeless.

These last two Scala patterns used to enjoy more popularity but have since then fallen out of favor with the community. I speak of tagless final and the cake pattern.

Earlier, I covered the trend of easily composing functions using monads. In that scenario, the monad itself is hard coded. Typically, you use the Option, Try, Either, or an IO monad.

What if you wanted to be able to change the actual type of the monad at runtime? The usual example is of using a Future when running as a reactive service but using an Either when running the unit tests where all blocking IO calls are mocked. That is what tagless final is all about. The monad itself is generic with a higher kinded type. You implement interpreters and bridges in order to transform that generic monad into something more specific at runtime.

Neither Java nor Scala permit multiple inheritance as it was originally envisioned when Object Oriented Programming was first introduced. This limitation avoids what is known as the dreaded diamond dependency problem. You could avoid some aggravating code duplication if you could have multiple inheritance. The Ruby programming language first introduced a more limited form of multiple inheritance that could reduce duplicate code without incurring the dreaded diamond dependency problem. This was known as the mixin. Scala followed suit with what is known as the cake pattern. With the cake pattern, you combine traits, self type annotation, and object creation using the "with" keyword in order to mix in multiple traits when creating new objects at runtime.



The Bad

There are some typical negative reactions from most junior Java developers when they are first exposed to idiomatic Scala code (i.e. code which uses Scala’s more advanced features as documented in the previous section).

Java developers learn to use the import statement in order to let the compiler know where to find each type that it is checking on. In that way, import statements are used to resolve compile errors resulting from ambiguity. They are used that way in Scala too but they are also used to bring implicits into scope. In that scenario, imports don't resolve compile errors, they change how the code behaves. This is unsettling to what Java developers are used to once they finally realize just what is going on.

Those implicits are great because you don't always have to type everything if the compiler can figure out what to use instead. The downside to that is when the compiler finds something wrong with it. You can end up looking at compile errors for code that is not actually explicitly found in your source. The compiler identifies the line of code where it realizes something is wrong but that most probably isn't what you need to fix. The change needs to be made somewhere else. This can be quite challenging for folks who don't take to coding easily.

Scala is all about building up layers of abstractions for writing programs at a higher level of abstraction but abstractions can easily become leaky, especially if the developers are neither disciplined nor focused on preventing that. Scala code with leaky abstractions can become misleading. Another downside to a higher level of abstraction is that developers can lose sight of precisely what is going on in the lower layers. This can lead to unexpected behavior or poor performance. This can happen in Java too. Perhaps it is all about perception but Scala seems to me to be a little more notorious about this since the community doubles down on it so much.

Take Slick for example. Slick is a modern relational database query and access library for Scala. You can use the query builder part of Slick instead of coding your own SQL queries. That's fine but then you don't really get the opportunity to tune the generated SQL query that came out of the query builder. SQL tuning is an important part to improving performance. You can also explicitly code the SQL in Slick or Doobie which is another library with a similar purpose. You can have the same problem on the Java side when using an ORM such as Hibernate or Spring Data.

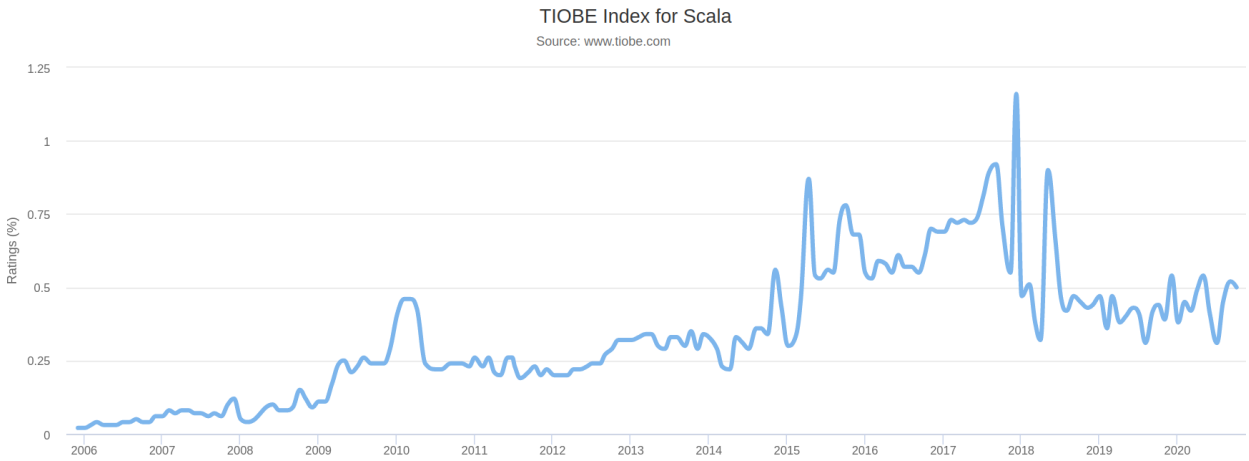
In Scala, there is this notion of lifting that is rare in Java. You can lift partial functions to functions, methods to functions, pure functions to effectful functions, and monads to monad transformers. This sounds a lot harder to learn than it actually is but typically Java devs encounter lifting at just about the same time that they start to believe that they have figured out what Scala is all about so it can be a little bit discouraging in the moment.

The Ugly

One thing that Java has that Scala does not is primitive data types. In Scala, everything is an object including integers and longs. In addition to that, implicit conversions, monads, lifting, and DSLs all create a lot more objects. This can put significantly more pressure on the Garbage Collector (responsible for reclaiming memory that is no longer in use) which can further aggravate periods of degraded performance for services that are struggling under high load. Modern GCs can handle this but may require some tuning which can involve a lot of guesswork to it. This can also obfuscate the diagnosis of real bugs such as memory leaks.

In Scala, method names can include and be completely composed of special symbols. By this I speak of, you know, punctuation characters. There are also special symbols as keywords in some of the type checking parts of Scala. This can make Scala code read as more dense than Java code. If you see addAll in Java, then you most probably are more likely to be able to follow along more easily than if you see ::: (three colons) in Scala. Also, the underscore character is overloaded a lot. It can be used for pattern matching and wildcards, ignoring things, and conversions.

You can enjoy the benefits of improving the readability of your code when wisely using just about all of the features in Scala covered in this article.



Scala libraries usually give you lots of extra methods so that you don't have to write much code for commonly occurring tasks. Sometimes those methods can be misleading. Take, for example, the fromTry method in the Future object which creates an already completed Future with the specified result or exception. You might find yourself needing to make a blocking database or external service call that can end up throwing an IOException but you need to call it from code that is running in a reactive framework. A more junior developer sees that method which takes a Try monad and returns a Future and says to themself "that is what is needed" without considering the threading model implications.

The Good

As you can see, there are some perceived downsides when transitioning from Java to Scala. What are the upsides? Why would you want to transition from Java to Scala in the first place? In order to understand that, we need to cover how Java and Scala handle modularity and readability differently. Let's introduce those two terms in more depth.

Modularity is a measure of how well that the code is compartmentalized. The lowest level of code compartmentalization is the method (or function or lambda). Instead of duplicating the same block of code over and over again, that block is placed inside the body of a method which gets called from everywhere that block of code is needed to be run. Methods are further grouped into classes which, in turn, are grouped into packages which are grouped into libraries which are loaded by services, etc. It is easier to find what you are looking for in a well organized code base. When code calls a method, there is an understanding that the method should perform what is expected of it. With each invocation, either that method succeeded in meeting that expectation or it didn't.

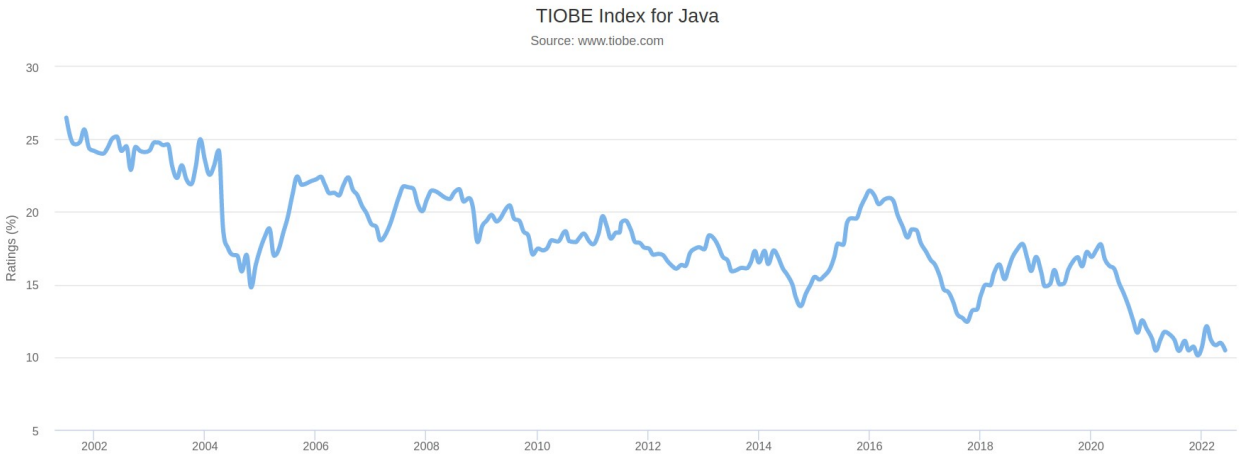
The readability of code lies in how easy it is for developers to read the code and understand what it is doing. The easier it is to read and understand an application, the less likely new bugs will get introduced with feature driven changes, the less costly it is to maintain the application over time. A program is written in code which serves as the implementation of a solution to a problem. One school of thought believes that the key to easily readable code is to express the solution in the language of the problem domain itself. DDD or Domain Driven Design subscribes to this view.

One thing that Java has that Scala does not is primitive data types.

The idiomatic way that Java code deals with success or failure in a method call is to return the declared value on success and to throw an exception (which could be declared or not) on failure. This entails wrapping the code that calls these methods in try blocks then including catch blocks and possibly a finally block to deal with the outcomes. In theory, you could use the try, catch, and finally blocks at the top of the stack only but in practice they end up everywhere. All of those try, catch, and finally keywords can detract from the readability of the code itself because it interjects languaging that is outside of the problem domain. Another downside is that you start to notice a code smell similar to having lots of goto statements. It can make the code hard to follow as you have to consider what is happening farther up the stack.

Although try, catch, and finally are also available in Scala, the idiomatic way that Scala code deals with success or failure of a method call is to wrap what the function should return on success in a monad. The lambda passed in to the map or flatmap methods will get called only on success. Other methods are available if you do need to inspect failure. With the for comprehension, you don't even need to explicitly call map or flatmap. It just looks like you are calling these methods and naively storing the success results in variables.

It can be argued that the for and yield keywords can detract from the readability in Scala as much as try, catch, and finally does in Java. In my experience, the Scala code is less distracting than the Java code since you get the benefits of defensive coding automatically without any extra boilerplate code. In a big system, that can add up to a lot of boilerplate code that you no longer need to maintain in Scala. Also, there is no smell of goto with the everything-is-a-monad approach to handling method invocation outcomes.



You can enjoy the benefits of improving the readability of your code when wisely using just about all of the features in Scala covered in this article. Type classes reduce the amount of plumbing code needed between controllers, services, DAOs, and drivers. Extension methods reduce the amount of code needed for creating wrapper objects for such purposes as the decorator pattern. The magnet pattern lets you use method names that are a better fit for the problem domain. Monads and DSLs are all about improving readability by making the code look more like the problem domain. Shapeless lets you have fewer methods that can work on a greater variety of data.

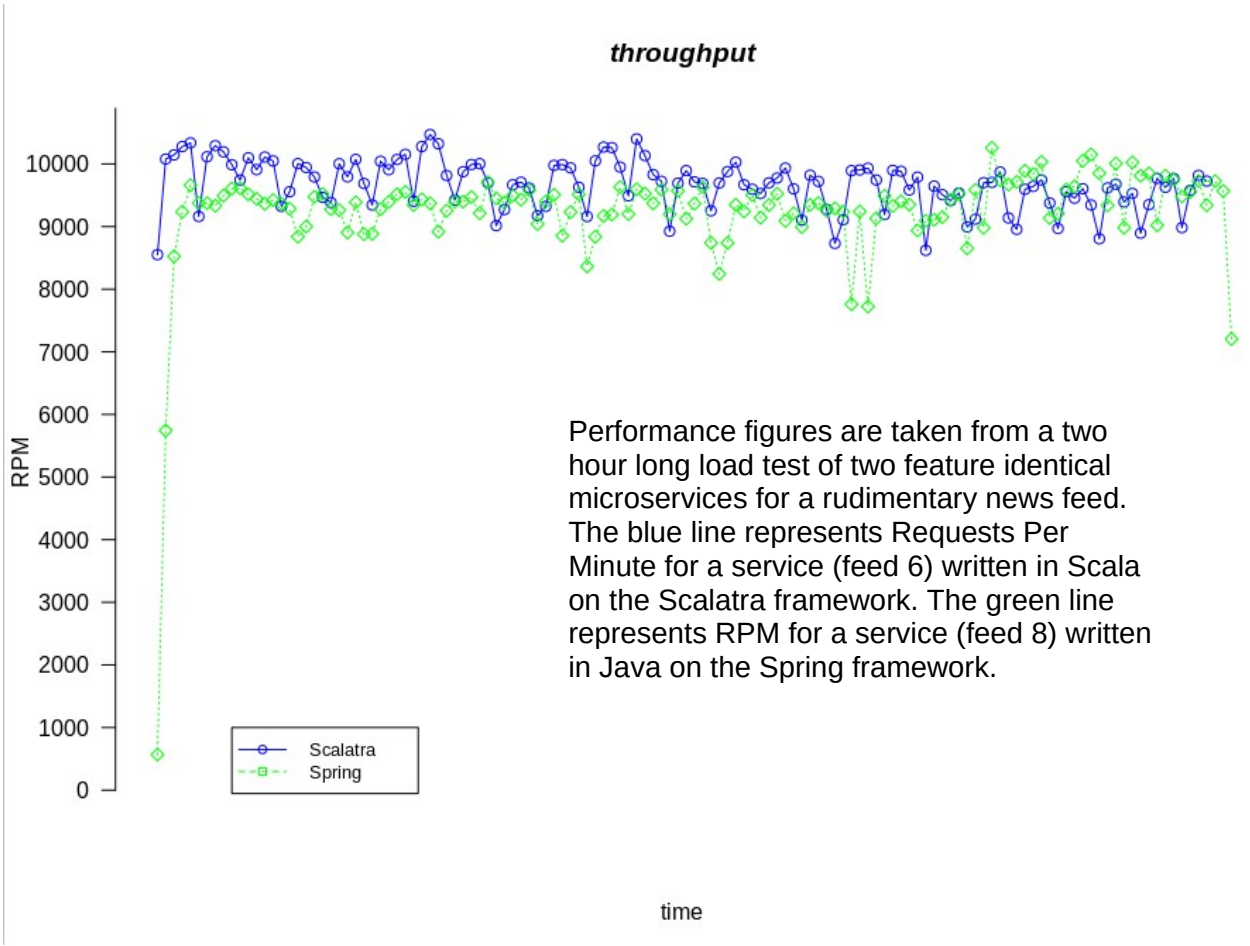
The Elephant in the Room

If Scala is so great, then why is it still a niche programming language on a slow decline after all these years while Java continues to prevail so highly? That is a difficult and subjective question but this is what I believe.

Learning Scala will be most attractive to your more senior or principal level engineers who have grown tired of Java's limitations and wish to learn new ways to program at scale. It will also be least attractive to your junior developers who are still struggling with the finer points of working on a large, multi-team, microservice architecture code base. These junior developers are busy wrapping their collective heads around a complex business model that has evolved over time and don’t wish to get overwhelmed with learning more advanced computer science constructs.

Engineering groups are typically composed of many junior level developers with just a small number of senior or principal level engineers sprinkled in to assume technical leadership responsibilities. This is mostly for economic reasons as pay rate usually aligns with experience level. There are other reasons too such as most of the coding work needed in a typical business application would be boring to the more gifted developers. It makes sense to standardize on a language that is more attractive to the larger number of junior programmers that you will need to build your bench with.

So there you have it. Scala will never overtake Java. Will Java wipe out Scala entirely? I doubt that. It also makes sense to retain your senior developers as it is harder, takes longer, and is more costly to replace them. Perhaps Scala can help prevent them from burning out and exploring other opportunities. Your results may vary.



Performance figures are taken from a two hour long load test of two feature identical microservices for a rudimentary news feed. The blue line represents Requests Per Minute for a service (feed 6) written in Scala on the Scalatra framework. The green line represents RPM for a service (feed 8) written in Java on the Spring framework.

Considering its merits around mitigating code complexity, why not sprinkle in a few Scala services especially in the areas that incur the most complexity? Those services will most likely require more senior attention anyway. Does your organization already have code written in different programming languages? Full stack development includes Swift (iOS), Kotlin (Android), Javascript or Typescript (cross platform, web, and node.js), Python and Scala (data science). That’s right. You may already have Scala in your technology stack if you are running any Spark jobs.

Getting Started

If you have Java developers who are Scala curious, then here is some advice on how to proceed.

Don't try to boil the ocean with some overly ambitious project to rewrite all Java microservices to Scala. Start with writing or rewriting a fairly simple microservice in Scala as a PoC or Proof of Concept project. In that way, you limit the scope of the exercise and set the expectation that a debriefing will occur at or near the end so that any lessons learned can be formally declared and reviewed.

Assuming you have a sufficiently healthy engineering culture that can accommodate this, form a Community of Practice or CoP around Scala development. In that way, the Java developers can share what they have learned about Scala and what they have experienced (good, bad, or indifferent) when coding in Scala.

Latency in Milliseconds for Feed Microservices 6 and 8

Framework	Mean	Median	95th Percentile
Scalatra	9	9	14
Spring	9	8	19

If possible, at least one of your developers should already have some prior positive experience in advanced Scala development. That person or persons should serve primarily as a mentor to the others. Sometimes, it can take a little encouragement to see the differences between Scala and Java in a positive light.

Framework	Avg Per File LoC	Cyclomatic Complexity
Scalatra	40	1981
Spring	56	273

Depending on the scope of the PoC and the experience levels of your existing developers, be prepared to spend some money on more formal training. There are lots of available resources at a wide variety of pricing options ranging from Udemy to Coursera to Scalac.

Even after living through all of its downsides, I still consider Scala to have a lot of computer science merit and encourage Java developers to learn Scala at some point in their professional careers. What better way to learn Scala than to join a group where you will be paid to do so (between twelve and twenty six percent higher on average than Java) in the construction of real world software? I also advocate for learning many programming languages because I believe that improves your programming skills overall.

