THE ADVENTUROUS DEVELOPER'S GUIDE TO JVM LANGUAGES

JAVA 8, SCALA, GROOVY, FANTOM, CLOJURE, CEYLON, KOTLIN, XTEND

Shameless taunt:
Are you brave enough to dive in?

REBELLABS

TABLE OF CONTENTS

INTRODUCTION & HISTORY 1-3

JAVA 8 5-10

SCALA 12-14

GROOVY 16-19

FANTOM 21-25

CLOJURE 27-30

CEYLON 32-35

KOTLIN 37-39

XTEND 41-44

OVERALL SUMMARY 46-47

PART I
KEEPING YOUR CODE CLEAN WITH UNIT TESTING

Unit tests let developers internally control the functionality and compatibility of their applications when they made changes to features, code or the environment.

Why do we need so many JVM languages?

It’s 2013 and you have over 50 JVM languages to choose from for your next project. But even if you can name more than a dozen or so, are you going to pick a new one for your next project?

Nowadays it’s easier than ever to create a new language with support from tooling like Xtext and ANTLR. Many new JVM languages have emerged as a result of limitations and disadvantages, either for creative individual coders or the masses, perceived in existing JVM languages, historically Java.

New JVM language developers feel as though their work is a result of existing languages either providing too limited functionality for their needs or too much functionality, leading to overweight or complex languages. Software development spans across such a wide range of appliances, so the usefulness of a language is frequently determined based on how relevant it is to a specific task, or how generic it can be across a larger area. All of which leads to development libraries and frameworks.

With so many languages already around, will the language graveyard become overpopulated? A danger exists that with so much choice in the market, many languages will just not last because they don't get enough attention or contribution in the face of less and less community resources.

The industry, however, survives based on innovating and creating, which very often comes from a project starting from scratch, dropping existing constraints and designs, and going with a blank slate.

There is a fine line that we will naturally dance around here: between helping to build existing languages and frameworks--helping to create the community backing it needs to survive and perhaps become the next Java--and the creation of new ideas, structures and paradigms which ultimately will make their way into existing languages.

This RebelLabs report takes a look at Java 8, Scala, Kotlin, Ceylon, Xtend, Groovy, Clojure and Fantom. But wait--with so many JVM languages to choose from, how did we settle on these eight only?

The team over here at RebelLabs has been discussing for 6 months how to do a report like this, and which languages to choose. Basically, we wanted to give something for everyone: Java is an incredibly well-known, widespread language, but Java 8 has new things that we wanted to explore. Groovy, Scala and Clojure have found their niches in the market and are becoming more popular, and languages like Ceylon, Kotlin, Xtend and Fantom are relatively new on our radars, gaining credibility and in need of some investigation.

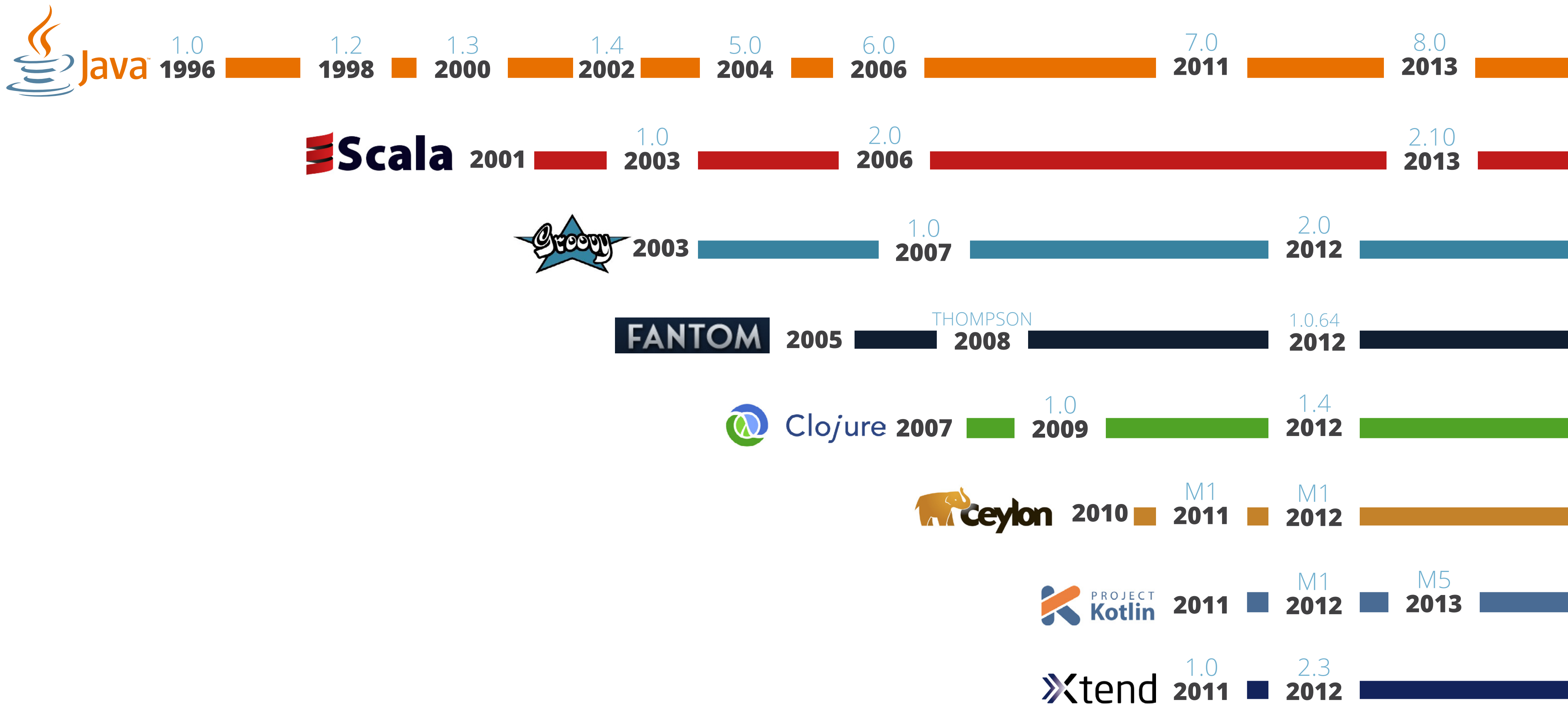
Our goal is to give an understanding of where each language came from, how they evolved and where they may be going. So in this report you will see us express our first impressions of the languages, including the features which rocked and the features which... well, didn’t rock!

You’ll be able to see our source code of a basic HTTP Server example in all implementations, with links to GitHub so you can play along with us.

A Little History

In the beginning, there was only Java, and it used to be the only programming language for the JVM. But the industry realized the need, and potential, for more options on the JVM quite early on. The first alternatives were targeting the scripting domain: Jython, a Python implementation for the JVM, as well as Rhino and JavaScript engines for the JVM, appeared in 1997, followed by BeanShell in 2000 and JRuby in 2001.

Scripting facilities were in high demand at that time due to the need for dynamic customization of applications. Nowadays, application servers like Oracle WebLogic and IBM WebSphere utilize Jython scripts for automation and Rhino was bundled with Java 6 to make JavaScript a first-class citizen on the JVM.



However, scripting facilities weren't the one and only factor for the growth of alternative programming languages on the JVM. Due to Java backwards-compatibility principles, alternative programming languages started to appear, offering some innovative features that weren't provided by Java or its standard libraries. Scala and Groovy were the first successful projects to offer something beyond Java.

An interesting observation can be made: most of the recent programming languages utilize static typing. Developers who use Scala, Ceylon, Xtend, Kotlin and Java itself rely on the compiler that verifies target types at compile time. Fantom tries to find a golden middle between static and dynamic typing, and Groovy, although initially a dynamic language, now added compile time static type checking in its 2.0 release in 2012. Clojure, the flavor of Lisp, is still the only reasonably-popular JVM programming language that sticks to dynamic typing, which some developers working on large teams in enterprise organizations consider to be a disadvantage.

The trend for new programming languages running on the JVM is changing from dynamic scripting languages for application customization towards statically-typed general purpose languages for application development.

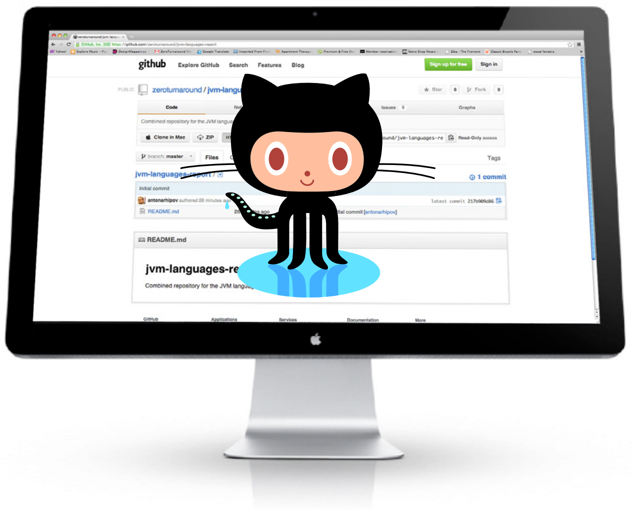
Java is still the most used programming language for JVM and with its upcoming Java 8 release it tries to keep up with the modern trends in syntax esthetics as well as with the requirements for multi-core programming.

Check out our merged repo on Github (all langs inside!)

Visit the Github Repo for Code Examples

Let's get geeky and under the hood of some JVM languages. In this report we look at Java (namely, things in Java 8), Scala, Groovy, Fantom, Clojure, Ceylon, Kotlin and Xtend--mostly from a fresh n00b perspective, and give our first impressions around what rocked, what sucked and what was just, meh.

Every language we look at in this section has it's own HTTPServer example which is currently in github. You can check out our code for all JVM languages covered in this report here:
https://github.com/zeroturnaround/jvm-languages-report



JAVA 8

> *"What I really care about is the Java Virtual Machine as a concept, because that is the thing that ties it all together; it's the thing that makes Java the language possible; it's the thing that makes things work on all kinds of different platforms; and it makes all kinds of languages able to coexist."*

JAMES GOSLING,
creator of the Java programming language (2011, TheServerSide)

Getting started with Java 8

Java Standard Edition version 8 is promising.
Let's look at the general evolution policy for the Java platform:

1. Don't break binary compatibility
2. Avoid introducing source incompatibilities
3. Manage behavioral compatibility changes

In general, the goal is to keep the existing binaries linking and running and keep the existing sources compiling. The backwards compatibility policy has influenced the set of features in Java the language as well as how these features are implemented. For instance, with current Java features it is not possible to evolve API as changing the interfaces would break source compatibility of the existing libraries that depend on JDK interfaces. This resulted in a change that affects both the language and VM.

With Jigsaw, the modularity theme, being dropped from Java 8, Project Lambda became the most important topic of the upcoming release. The name is a bit misleading though. While lambda expressions are definitely one big part of it, it is not the most important feature on its own, but rather a tool for Java in its multi-core efforts.

In this multi-core era, **parallel libraries** are in demand and this puts a pressure on the collections API in Java. In its turn, **lambda expressions** are required to make this API friendlier and convenient to use. **Defender methods** are the tool for API evolution and this is how the existing collections' libraries will make a step towards multi-core support.

So you want to use lambdas, huh?

If you are familiar with other languages that include lambda expressions, like Groovy or Ruby, you will be surprised first, that it is not as simple in Java. In Java, lambda expression is the representation of "SAM type", which is an interface with a single abstract method (yes, interfaces can include non-abstract methods now - the defender methods).

So for instance the well known `Runnable` interface is perfectly suitable for serving as a SAM type:

```
Runnable r = () -> System.out.println("hello lambda!");
```

Or the same could be applied to the Comparable interface:

```
Comparator<Integer> cmp = (x, y) -> (x < y) ? -1 : ((x > y) ? 1 : 0);
```

The same can be written as follows:

```
Comparator<Integer> cmp = (x, y) -> {
  return (x < y) ? -1 : ((x > y) ? 1 : 0);
};
```

So it seems like the one-liner lambda expressions have implicit `return` for the statement.

What if you want to write a method that can accept a lambda expression as a parameter? Well, you have to declare the parameter as a functional interface, and then you can pass the lambda in:

```
interface Action {
  void run(String param);
}
```

```
public void execute(Action action){
  action.run();
}
```

Once we have a method that takes a *functional interface* as a parameter we can invoke it as follows:

```
execute((String s) -> System.out.println(s));
```

Effectively, the same expression can be replaced with a method reference since it is just a single method call with the same parameter:

```
execute(System.out::println);
```

However, if there's any transformations going on with the argument, we can't use method references and have to type the full lambda expression out:

```
execute((String s) -> System.out.println("*" + s + "*"));
```

The syntax here is rather nice and we now have quite an elegant solution for lambdas in Java language despite Java doesn't have functional types *per se*.

Functional interfaces in JDK8

As we learned, the runtime representation of a lambda is a functional interface (or a "SAM type"), an interface that defines only one abstract method. And although JDK already includes a number of interfaces, like `Runnable` and `Comparable`, that match the criteria, it is clearly not enough for the API evolution. It just wouldn't be as logical if we started to use `Runnables` all around the code.

There's a new package in JDK8, `java.util.function`, that includes a number of functional interfaces that are intended to be used in the new API. We won't list all of them here - just make yourself a favor and study the package yourself :)

It seems that the library is evolving quite actively as some interfaces come and go. For instance, it used to provide `java.util.function.Block` class which isn't present in the latest build that we have at the time of writing this report:

```
anton$ java -version
openjdk version "1.8.0-ea"
OpenJDK Runtime Environment (build 1.8.0-ea-b75)
OpenJDK 64-Bit Server VM (build 25.0-b15, mixed mode)
```

As we discovered, it is now replaced with Consumer interface and is used for all the new methods in the collections library. For instance, the `Collection` interface defines the `forEach` method as follows:

```
public default void forEach(Consumer<? super T> consumer) {
  for (T t : this) {
    consumer.accept(t);
  }
}
```

The interesting bit of the `Consumer` interface is that it actually defines one abstract method – `accept(T t)`, and a defender method - `Consumer<T> chain(Consumer<? extend T> consumer)`. It means that it is possible to chain the calls using this interface. We're not quite sure how it will be used since we couldn't find `chain(..)` method usages in the JDK library yet.

Also, notice that all the interfaces are marked with `@FunctionalInterface` (http://download.java.net/jdk8/docs/api/java/lang/FunctionalInterface.html) runtime annotation. But aside its runtime presence the annotation is used by javac to verify if the interface is really a functional interface and there's no more than one abstract method in it.

So if you try to compile code like this:

```
@FunctionalInterface
interface Action {
   void run(String param);
   void stop(String param);
}
```

The compiler will tell you:

```
java: Unexpected @FunctionalInterface annotation
  Action is not a functional interface
  multiple non-overriding abstract methods found in interface Action
```

While the following will compile just fine:

```
@FunctionalInterface
interface Action {
   void run(String param);
   default void stop(String param){}
}
```

Defender methods

A new concept that appeared in Java 8 is the default methods in interfaces. It means, that the interfaces can now not only declare the method signature but also hold the default implementation. The need for such approach came from the requirement of evolving the interfaces in JDK API.

The most obvious use case for defender methods is the Collections API in Java. If you used Groovy you probably have written code like this:

```
[1, 2, 3, 4, 5, 6].each { println it }
```

Currently, we iterate in Java with a for-each loop as follows:

```
for(Object item: list) {
   System.out.println(item);
}
```

This loop is possible because the `java.util.Collection` interface extends the the java.util.Iterable interface that defines a single method `Iterator<T> iterator()`. For Java to make use of a Groovy style iteration we need a new method in either `Collection` or `Iterable` interfaces. However, adding it breaks the source level compatibility for the existing collection libraries. So in Java 8 the `java.util.Iterable` adds the `forEach` method and provides the default implementation.

```
public interface Iterable<T> {

  Iterator<T> iterator();

  public default void forEach(Consumer<? super T> consumer) {
    for (T t : this) {
      consumer.accept(t);
    }
  }
}
```

Adding the new default method doesn't break the source level compatibility, as implementors of the interface don't need to provide their own implementation of the method, so the existing sources will continue to compile when switching from Java 7 to Java 8. Thus, in Java 8 we will be able to write iteration code as follows:

```
list.forEach(System.out::println);
```

The `forEach` method takes a functional interface as a parameter, therefore we can pass in a lambda as a parameter or a method reference as in the example above.

This approach to iteration is essential to multi-core support since with this style you basically say that there's a task that needs to be fulfilled and don't care about the iteration details - the underlying library carries out the iteration for you. The new lambda expressions alone would not make sense to Java developers since without the the proper API in the collections library it wouldn't be possible to use them to their full extent.

We asked the creators & project leads from different JVM languages what they thought about the new features in Java 8

SVEN EFFTINGE - XTEND

Yes, they are definitely needed and a good step in the right direction. Xtend will use Java 8 as an optional compilation target to improve the generated code.

Java 8 lambdas are semantically very similar to the ones from Xtend. The new stream API for instance works nicely with Xtend without any further ado. In fact it works even better from Xtend than from Java 8. I've written a short blog post on that [http://blog.efftinge.de/2012/12/java-8-vs-xtend.html] :-). Still I prefer the Guava APIs since they are more convenient and readable than the Java 8 stream API.

The defender methods are a good addition as well, although I don't like the syntax with the 'default' keyword. They struggled with the different default visibility of interface and class methods. I think they are trying to make a significant syntactic distinction so people don't confuse classes with interfaces. In Xtend the default visibility of methods is the same for classes and interfaces that isn't a problem here.

BRIAN FRANK - FANTOM

Excited might be a stretch :-) Virtually every other modern language has had the basic mechanisms for functional programming for years now. But there are armies of programmers out there just using Java and still unfamiliar with these concepts, so I think it will be good to indoctrinate more developers to the functional style. We don't see functional programming as a panacea, but it is a very useful tool in the toolbox.

GAVIN KING - CEYLON

To the extent that Java 8 rekindles interest in the Java platform, and brings developers back to Java, that will be a wonderful thing for Ceylon and other languages for the JVM.

With Java 8, some very common programming tasks are made a whole lot more convenient. On the other hand, I'm extremely disappointed that the Java SE team has still not managed to deliver built-in modularity after many years of trying. While they definitely deserve praise for the good work that's gone into Java 8, I think it's only fair to be equally critical of the failure to deliver on _such_ a critical front. Lambdas are a useful and convenient syntax sugar. But modularity is key to everything Java is about, and is a key cause of its failure in certain areas.

JOCHEN THEODOROU - GROOVY

Not excited, no. These are things Groovy has been able to do for years. Defender methods look to me like a step only half done, plus I don't like the mixup with interfaces. Lambdas are much more interesting to me, but once again I see Groovy Closures as the more powerful concept. Lamdas can surely make Java a better language, but they will complicate Java concept wise as well. Without Scala and Groovy I am pretty sure those features would have never come into existence. They are IMHO a reaction to pressure from alternatives. And then they had to make a complicated balancing act between staying a beginners language and being sexy to advanced users. And they got stuck somewhere in the middle, resulting in lambdas and defender methods.

GUILLAUME LAFORGE - GROOVY

I'm not as negative as Jochen here, although his views aren't that far from the reality in how other languages influenced Java to evolve towards that direction.

Even if Java 8 lambdas, the "stream"-ing reworked collections, or the defender methods aren't exactly how we would have liked them to be, I think all those things combined should bring some renewal to how developers evolve their APIs, and it should hopefully make for nicer designs and more streamlined usage of new and old frameworks.
So overall, I think it's a good thing for Java.

ANDREY BRESLAV - KOTLIN

Java getting better means millions of developers getting happier. Kotlin can make some of them even more happy, but it's another question :)

People who only need "Java with closures" will get it with Java 8 and be happy. But there are other people, who need more than just anonymous functions (that are very important indeed, but the world does not end there).

SCALA

“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.”

MARTIN ODERSKY,
creator of Scala
(from Scala 2013: A Pragmatic Guide to Scala Adoption in Your Java Organization)

Getting started with Scala

Scala is rather well-established compared to most of the languages covered in this report. It saw its first release already in 2003, but started appearing on many radars beginning with 2006, when version 2.0 was released at EPFL. Since then it has been gaining popularity and may even be approaching a place among first-tier languages, depending on which language ranking you believe.

Much of what makes it interesting was already present in 2006 – it's a statically typed mix of OOP and FP (Functional Programming) with type inference; has no primitives, but compiles to fast code. It has pattern matching, an advanced type system with a bottom type, algebraic data types, structural types and even dependent types. It also makes possible to express category theory abstractions such as monads, but you can make up your own mind about whether you care about that or not. Even if you use some of the monads in the standard libraries, you can do that even without knowing what a monad is.

There is no such thing as a typical Scala developer – some are Java developers who wanted a more expressive language, and some are functional programmers who saw a good language to use on the JVM. This means that Scala programs can be written in wildly different styles – pure functional, impure imperative, or a mix of the two. And even orthogonal to that choice, you can either go the way of abstracting as much as possible, taking advantage of the advanced type system (see Scalaz & Shapeless libraries), or abstracting just a bit more than you would in Java code.

Scala has gone through some major changes since 2006. One of the biggest changes coming with Scala 2.8 was an overhauled collections API, which is perhaps the most powerful that any language has, but also has more complexity in it's implementation details than most collection libraries. Version 2.9 added parallel collections and 2.10 brought a bunch of features, some of them experimental: hygienic macros, new reflection library, String interpolation, greatly improved pattern matching code generator, and more.

Main differences compared to Java

While implementing the HTTPServer sample in Scala, we can immediately notice it does away with the `static` keyword, and instead, what would be `static` in Java will be in a companion / singleton object. A companion `object` is one that has the same name as a class. So in effect, we have split our HttpServer into an object and a class – the object has the static parts and the class has the dynamic parts. These are two separate namespaces in Scala, but we can import the static namespace in the dynamic part for convenience:

```
import HttpServer._ // import statics from companion object
```

Scala allows import statements anywhere and you can import members into the current scope from anything visible in that scope that has members; so the structure of Scala code is actually quite a bit more regular than Java, which only allows you to place imports at the beginning of a file and import only classes or static members of a class.

A couple of things that immediately stand out in comparison to Java is that methods are defined in the form `def name(argument: Type)` and similarly variables are defined

```
val name: Type = initializer // final "variable"
```

or

```
var name: Type = initializer // mutable variable
```

You should prefer not to use mutable variables, so use `val` by default -- actually our sample code does not have any vars at all because they are needed much less often than you'd think if you've mostly written Java code. Often you can leave out the type declaration from definitions and have it inferred, but method arguments must always have explicit types.

Calling Java code from Scala is usually easy as can be seen from the `main()` method, which creates some Java objects such as `ServerSocket` and a thread pool via the `Executors` class.

Case classes & pattern matching

An interesting feature in Scala are case classes which are like normal classes, but have compiler-generated implementations of equals, hashcode, toString, pattern matching support and so on. This lets us create small types for holding data in only a few lines. For example, we chose to use a case class to hold the HTTP status line information:

```
case class Status(code: Int, text: String)
```

In the `run()` method we can see a pattern match expression. It is similar to switch in Java, but much more powerful. However, here we don't go into it's real power, we just use an `| (or)` pattern and the fact that you can bind a matched result to a name by prefixing it with `name @`.

The pattern we use is to match the HTTP method name read from the HTTP connection input stream. We match either "`GET`" or "`HEAD`" in the first pattern, and anything else in the second:

```
case method @ ("GET" | "HEAD") =>
 ...
case method =>
```

```
respondWithHtml(
  Status(501, "Not Implemented"),
  title = "501 Not Implemented",
  body = <H2>501 Not Implemented: { method } method</H2>
)
...
```

In the second case we take advantage of named (and default) arguments when calling `respondWithHtml` -- they allow us to name the arguments at the method call site, to avoid having to remember the order of similarly-typed arguments, or just to make the code more clear. Here we chose to not name `status` because its meaning already obvious from the `Status(...)` constructor call.

Fun with Strings

Another interesting feature -- added in Scala 2.10 -- is String interpolation. You write your normal String constants prefixed with s and that allows you to embed Scala code in the String, escaped with `${}` or just `$` for simple names. Combined with multi-line Strings, we can easily construct the HTTP header we are going to send without any String concatenation:

```
val header = s"""
  |HTTP/1.1 ${status.code} ${status.text}
  |Server: Scala HTTP Server 1.0
  |Date: ${new Date()}
  |Content-type: ${contentType}
  |Content-length: ${content.length}
""".trim.stripMargin + LineSep + LineSep
```

(Note: we can optionally leave out parentheses on zero-argument method invocations)

Scala allows us to implement our own String interpolation as well and give it a custom prefix, but the default one is sufficient here.

The `trim` method is the normal `Java String.trim()`, which removes whitespace (including line breaks) from the beginning and end of the String. The `stripMargin` method removes everything up to the | character from the beginning of each line in a `String`, allowing us to use normal indentation for the multi-line `Strings`. The method is added to the String type via implicit conversion from `String` to `WrappedString` and similarly we could add our own margin stripping logic, for example one that would `trim` each line, so you could do without the extra | characters.

Built-in XML, love it or hate it

In the `respondWithHtml` method we see another interesting, but perhaps not so loved, Scala feature: built-in XML expressions. The method takes a sequence of XML nodes (`scala.xml.NodeSeq`) representing the children of a XHTML `<body>` element as an argument, and wraps that in another XML expression that adds the `HTML/HEAD/BODY` elements around the actual title and body, then converts it to bytes. When we call this method, we can again use XML expressions to provide the element(s) for the body.

```
def respondWithHtml(status: Status, title: String, body: xml.
NodeSeq) =
 ...
 <HTML>
   <HEAD><TITLE>{ title }</TITLE></HEAD>
   <BODY>
     { body }
   </BODY>
 </HTML>

 ...
```

Avoiding null-pointer errors

In `toFile` and `sendFile`, we use Scala's preferred way of dealing with optional values, the `Option` type (note that Scala has `null` as well).

`toFile` returns `Some(file)` or None if it doesn't find a file to serve, and then `sendFile` does a pattern match covering both of those cases. If we left one of the cases out, the compiler would warn us about it.

```
def toFile(file: File, isRetry: Boolean = false): Option[File] =
 if (file.isDirectory && !isRetry)
   toFile(new File(file, DefaultFile), true)
 else if (file.isFile)
   Some(file)
 else

   None
```

Everything is an expression

We also make use of the fact that almost all constructs in Scala are expressions, so the `if-else-if-else` control structure actually results in a value. Thus the method is made of a single expression and we can leave out the braces `{}`.

In the `sendFile` method we see more of this, where we use local blocks `{...}` that result in a value – the last expression in a block is it's return value, so we hide all our temporary variables in a block and assign the result of the block to a less-temporary variable.

```
val contentType = {
 val fileExt = file.getName.split('.').lastOption getOrElse ""
 getContentType(fileExt)
}
```

Even though the full depth of Scala wasn't demonstrated here, we saw that it can be quite expressive, and hopefully this example gives some idea of what the language is like. The key point is to embrace immutability and try to model code as composable expressions. Thus the `toFile` method seemed better extracted out of `sendFile` compared to our reference sample.

GROOVY

“*Groovy has sweet spots beyond what Java will ever offer, for example in its ability to easily be embedded and compiled on the fly in host applications to provide customizable business rules, and in how it's capable of offering a very elegant, concise and readable syntax for authoring Domain-Specific Languages.*”

GUILLAUME LAFORGE,
Project Lead for Groovy

Getting started with Groovy

Groovy is not as adventurous as some of the languages we're covering in this report, but definitely is a JVM language you should be interested in. It's become a mature choice that developers trust when Java verbosity hurts and dynamic typing isn't an issue.

In any case, I'm not one to argue when given an opportunity to play around with programming languages.

Java gets supercharged

Java developers can just dive in and be productive in Groovy. Syntax matches Java where possible and this seems to be the trend in the future, as 2.0 release added Java 7 Project Coin enhancements. In addition Groovy smooths Java annoyances encountered on a daily basis. Safe navigation `(?.)` and Elvis `(?:)` operators are great examples.

```
// streetName will be null if user or
// user.address is null - no NPE thrown
def streetName = user?.address?.street

// traditional ternary operator usage
def displayName = user.name ? user.name : "Anonymous"

// more compact Elvis operator - does same as above
def displayName = user.name ?: "Anonymous"
```

"Groovy is a multiparadigm language for the JVM. Using a syntax close to Java, it's an easy to learn language allowing you to write code from scripts to full applications, including powerful DSLs. Groovy is probably the only language on the JVM that makes all runtime metaprogramming, compile-time metaprogramming, dynamic typing and static typing easy to handle."

CÉDRIC CHAMPEAU
senior software engineer with Groovy

Closures

We expected the grooviness to stop with syntactic enhancements, but then we spotted “Closures” in the documentation. Why it is called as such is beyond us, as functions are first-class citizens in Groovy--function values, higher-order functions and lambda expressions are all supported.

```
square = { it * it }  // ‘it’ refers to value passed to the function
[ 1, 2, 3, 4 ].collect(square)  // [2, 4, 9, 16]
```

Excellent use of closures in the standard library makes it a pleasure to use and demonstrates their power. Here’s a good example with syntactic sugar for closures as last method parameter:

```
def list = ['a','b','c','d']
def newList = []

list.collect( newList ) {
  it.toUpperCase()
}
println newList  // [A, B, C, D]
```

Collections

Almost every application relies on collections. Unfortunately collections largely represent the pain of Java. And if you doubt me, please try to have some fun with JSON manipulation. Groovy packs native syntax for collection definitions and makes heavy use of closures for powerful manipulation.

```
def names = ["Ted", "Fred", "Jed", "Ned"]
println names  //[Ted, Fred, Jed, Ned]

def shortNames = names.findAll { it.size() <= 3 }
println shortNames.size()  // 3
shortNames.each { println it }  // Ted
                                // Jed
                                // Ned
```

Static typing

People often get excited about dynamic languages, since for less code you appear to get more functionality. It is often less understood, that surplus gets taken back in maintenance. So we can see more and more dynamic languages getting static typing, and vice versa.

Static typing was added to Groovy 2.0.
Why static typing and how does static typing improve Groovy?

"Static typing makes the transition path from Java to Groovy even smoother. A lot of people came (and still come) to Groovy because of it's lighter syntax and all the removed boilerplate, but, for example, don't want (or need) to use dynamic features. For them it's often difficult to understand that Groovy doesn't throw errors at compile time where they used to have them, because they don't really understand that Groovy is a dynamic language. For them, we have @ TypeChecked now. The second reason is performance, because Groovy still supports older JDKs (1.5, currently) and that invokedynamic support is not available to them, so for performance critical sections of code, you can have statically compiled code. Also note that static compilation is interesting for framework developers that want to be immune from monkey patching (the fact of being able to change the behaviour of a method at runtime)."

CÉDRIC CHAMPEAU
senior software engineer with Groovy

Groovy is no exception, static checks can be enabled by annotation @TypeChecked in the relevant code.

```
import groovy.transform.TypeChecked

void someMethod() {}

@TypeChecked
void test() {
    // compilation error:
    // cannot find matching method sommeeMethod()
    sommeeMethod()

    def name = "Marion"

    // compilation error:
    // the variable naaammme is undeclared
    println naaammme
}
```

What is your favourite application / framework / library written in Groovy and why are you passionate about it? Can name more than one.

"*Well that's easy, my passion is Griffon, a desktop application development platform for the JVM. Groovy is used as the primary language in the framework, however other JVM languages may be used. For web development I cannot see myself writing webapps without Grails, it's simply that good not to mention refreshing and fun. Gradle is another precious addition to my toolbox; I use it whenever I have the chance. Finally, Spock shows the power of the Groovy compiler and AST expression handling by enabling a simple yet very powerful testing DSL.*"

ANDRES ALMIRAY
Groovy Committer

FANTOM

"Fantom is an elegant, next generation language with a strong focus on concurrency and portability. Immutability is deeply baked into Fantom's type system and concurrency is enforced using the actor model. Fantom was designed from the ground up for portability and has production quality implementations for both the Java VM and JavaScript/HTML5. Fantom takes a pragmatic approach to style with a focus on static typing, but easily allows dynamic programming. It is an object oriented language, but includes first class functions and uses closures for many of the standard APIs."

BRIAN FRANK
creator of Fantom

Getting started with Fantom

Fantom is a bit different from most of the languages we are looking at in this report, as it targets multiple platforms. Compilation for JVM, .Net and JavaScript are currently supported, and given the infrastructure they've put into place, it should be possible to target other platforms as well.

But despite the fact that portability and platform maturity are important issues for Fantom's authors (Brian and Andy Frank), it's not what defines the language for them. They claim that Fantom is a practical language, for getting things done.

The first step to do that is to setup an environment and the tooling. Fortunately, Fantom made this very easy for us. Xored makes an Eclipse-based IDE called F4, which includes everything we needed to get Fantom up and running.

Pods/ Scripts

Fantom can execute files as scripts, you just need to put a class with main method into a file and there's an executable fan to run it.

```
class HelloWorldishScript
{
  static Void main() { echo("Woah! Is it that easy?") }
}
```

However that's not the main way of structuring Fantom programs. For larger projects and production systems precompiled modules, called pods, are created using Fantom's build toolkit.

The build is orchestrated by a build script, which is essentially just another piece of Fantom code. Here is the build script for Fantom's implementation of the HTTP Server sample:

```
using build
class Build : build::BuildPod
{
  new make()
  {
    podName = "FantomHttpProject"
    summary = ""
    srcDirs = [`./`, `fan/`]
    depends = ["build 1.0", "sys 1.0", "util 1.0", "concurrent 
1.0"]
  }
}
```

There are a couple of things to notice straight away, dependency specification for example allows us to build larger systems more easily and in a less jar-hellish manner. Additionally, a pod does not only define deployment namespace, but also type namespace, unifying and simplifying these two. Now you can see that our server depends on pods: sys, util and concurrent.

How did the idea appeared to support multiple backends in Fantom (.NET, JavaScript)?

In a previous life, we built a product in Java, but had many issues selling the solution into .NET shops. So we designed Fantom to target both ecosystems. But as we began development of our current product, we started steering Fantom in a new direction with the goal of using one language and codebase for our backend running on the JVM and our front end running in HTML5 browsers. This has been a very successful strategy now for several years.

BRIAN FRANK
creator of Fantom

Standard Library & Elegance

Fantom poses not as just a language for the JVM platform (or any other platform in fact), but more as a platform itself relying on the JVM. A platform offers API and Fantom makes sure that API is beautiful and elegant. On the most basic level it offers several literals like:

```
Duration d := 5s
Uri uri := `http://google.com`
Map map := [1:"one", 2:"two"]
```

And it's a little thing having a duration literal, but when you want to set a timeout for something it just feels so right that someone has thought about that for you.

The IO APIs are covered by several base classes like `Buf`, `File`, `In/OutStreams` which are very pleasant to use. Network communication is also supplied, JSON support, DOM manipulation and graphic library. Essential things are there for you. `Util` pod contains some useful things as well. Instead of having a class with `main` method, file server extends `AbstractMain` class and gets parameters passing, logging setup for free. Another API which is very pleasant to use is Fantom's concurrency framework, but we'll talk about that in a few minutes.

Interop

Any language that is built on top of the JVM platform offers some interoperability with plain Java code. It is essential to make use of the enormous ecosystem Java has. And it is said that creating a language better than Java is easy. Creating a language better than Java that offers decent interoperability with Java is harder. Part of that is dealing with the collections (which is kinda old and plain, and sometimes a pain to use).

Fantom offers an Interop class with a `toFan` and `toJava` methods to convert types back and forth.

```
// socket is java.net.socket
InStream in := Interop.toFan(socket.getInputStream)
OutStream out := Interop.toFan(socket.getOutputStream)
```

Here you can see that we have a plain Java Socket and it naturally provides us with Java Input and `OutputStreams`. Using `Interop` we convert them to Fantom counterparts and use later just them.

Static AND Dynamic typing?

Another major topic in any language review is if the language supports static / dynamic typing. Fantom here hits the middle ground and we liked it a lot. Fields and method signatures feature strong static typing. But for local variables, types are inferred. This leads to a mix that is really intuitive, method contracts are spelled out, but you don't actually need to type everything.

And naturally there are two method invoke operations in Fantom. The dot `(.)` invocations go through a compiler check and are strongly typed, and arrow `(->)` invocations do not. This enables duck-typing and everything you want from a dynamically typed language.

Immutability & Concurrency

Fantom offers an actors framework to handle concurrency. Message passing and chaining asynchronous calls are easily incorporated into code. To create an actor (which would be backed by some `ActorPool` and subsequently by a thread pool), you need to extend an `Actor` class and override (curiously enough you must explicitly type the `override` keyword) a receive method.

Please note that to avoid sharing state between threads, Fantom will insist you only pass immutable messages to your actors. Immutability is built into the language by design, so you can define your classes and all their fields as const. The compiler verifies that the message for the actor is in fact immutable and will throw an exception otherwise.

A cool and somewhat difficult to find tip is that if you really need to pass a mutable object you can wrap it into `Unsafe` (no, not that `Unsafe`, Fantom's `Unsafe`).

```
while(true) {
  socket := serverSocket.accept
  a := ServerActor(actorPool)
  //wrap a mutable socket to sign that we know what are we doing
  a.send(Unsafe(socket))
}
```

Later you can obtain the original object back.

```
override Obj? receive(Obj? msg) {
  // Unsafe is just a wrapper, get the socket
  log.info("Accepted a socket: $DateTime.now")
  Socket socket := ((Unsafe) msg).val
  …
}
```

Apparently, an official approach to this problem is different as Andy Frank has kindly pointed out.

> *You should consider Unsafe an absolute last resort - since it can undermine the entire concurrency model in Fantom. If you need to pass mutable state b/w Actors - you would use serialization - which is a built-in feature: http://fantom.org/doc/docLang/Actors.html#messages*
>
> **ANDY FRANK,**
> **creator of Fantom**

It means, that a proper solution here would be something like:

```
while(true) {
  socket := serverSocket.accept
  a := ServerActor(actorPool, socket)
  a.send("handleRequest")
}
```

This way we can store the socket for the following IO operations on that, and we don't need to pass anything immutable as a message. By the way, this code looks better too!

Functions & Closures

Fantom is an object-oriented language and, as with many modern languages, functions are first-class citizens in Fantom also. The following example shows an actor creation, where we specify the receive function implicitly and then send a message several times.

```
pool := ActorPool()
a := Actor(pool) |msg|
{
  count := 1 + (Int)Actor.locals.get("count", 0)
  Actor.locals["count"] = count
  return count
}

100.times { a.send("ignored") }
echo("Count is now " + a.send("ignored").get)
```

Fantom's syntax is friendly enough and doesn't get in the way. This is probably the right place to notice that variable declarations have := syntax, which would be a disaster for me (we don't have the best memory for small syntax details). However, the IDE supports this well and notifies you every time you make this mistake.

Little things

The whole time we were exploring Fantom, we were pleasantly surprised by the small things that help make the language better. For example, null supporting types: one can declare a method to accept null as an argument or not.

```
Str   // never stores null
Str?  // might store null
```

This way, the code is not polluted with null checking and interoperability with Java is easier.

There are other features that are worth mentioning. Multi-line strings with variable interpolation:

```
header :=
         "HTTP/1.1 $returnCode $status
          Server: Fantom HTTP Server 1.0
          Date: ${DateTime.now}
          Content-type: ${contentType}
          Content-length: ${content.size}
          ".toBuf
```

Parameters can have default values, mixins and declarative programming support, and operator overloading. Everything is in place. One thing we didn't find and felt the lack of were tuples. However we only needed that for multiple returns, so using a list was enough.

CLOJURE

“*I set out to create a language to only deal with the problems I was contending with in Java and C# for the kinds of applications that I write, which are broadcast automation and scheduling and elections systems and things like that, where there is a lot of concurrency. I found just object oriented programming and their approaches to concurrency of those languages is just not good enough for those kinds of problems - they are too hard. I'm a big fan of Lisp and the other functional languages and what I want to do is solve those problems, make a practical language, not to have to program in Java anymore.*”

RICH HICKEY
creator of of Clojure,
in a 2009 InfoQ interview

Getting started with Clojure

Clojure first appeared in 2007 and is relatively new compared to more established languages. It was created by Rich Hickey as a Lisp dialect targeted for the JVM. Version 1.0 appeared in 2009 and the name is a pun on C (C#), L (Lisp) and J (Java). The current version of Clojure is 1.4. Clojure is open-source (released under the Eclipse Public License v 1.0 – EPL)

After starting reading the Clojure documentation, we decided to setup our development environment using Leiningen which is a Maven like tool for Clojure. Leiningen (or Lein for short) can perform most tasks for Clojure that you would expect from Maven such as:

- Create a project structure (think: Maven archetypes)
- Handle dependencies.
- Compile Clojure code to JVM classes
- Run tests
- Publish built artifacts to a central repository

If you have used Maven, then you will feel right at home with Lein. In fact, Lein even supports Maven repositories. A major difference between the two however is that the project file in Lein is written in Clojure itself while Maven uses XML (pom.xml). We have to admit that Lein was a very welcome addition and although it is possible to develop in Clojure without it, it really makes several things much easier.

To get the project skeleton, please do the following:

```
$ lein new clojure-http-server
```

IDE support

After getting the basic project structure in place, it's time to start editing code. If you are a long-time Eclipse user, the first thing you might do is search for an Eclipse plugin that handles Clojure. This would offer syntax highlighting and code completion in a familiar workspace. Luckily there is Eclipse plugin for Clojure called CounterClockWise. Just install the plugin and created a new Clojure project in Eclipse.

While this was OK, we didn't manage to import into Eclipse the existing Clojure project that we had created from the command line with Lein as described into the previous section. We expected the CounterClockWise plugin to function like the Maven Eclipse plugin where there is two-way interaction between the pom.xml and the Eclipse GUI.

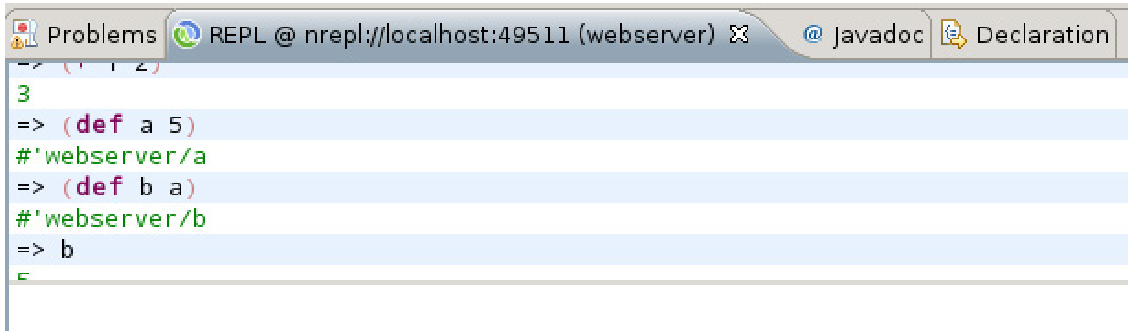
Just for fun, we also looked at Clooj, which is a lightweight IDE for Clojure developed in Clojure itself. While it is very easy to download and run it, we found it very lacking compared to Eclipse.

In the end we developed Clojure the hard way using just Lein from the command line and the trusty GVIM as a text editor, mainly in order to see how Lein works in detail.

The Read Eval Print Loop (REPL)

Like most Functional languages, Clojure offers a command line shell where you can directly execute Clojure statements. This shell is very handy for development since it allows you not only to test small code snippets but also to run only parts of the program during development.

This is nothing new for developers who have already used languages like Python or Perl. But for Java developers its brings a refreshing and more interactive way to coding.



```
3
=> (def a 5)
#'webserver/a
=> (def b a)
#'webserver/b
=> b
```

Functional programming – a different way of thinking

Clojure is a functional language very similar to Lisp or Scheme. The functional paradigm is very different for those who are accustomed to the Java way of OOP and working with side effects all the time.

Functional programming promotes:

- Little or no side effects
- Functions that always return the same result if called with the same argument (and not methods that depend on the object state)
- No global variables
- Functions as first-order objects
- Lazy evaluation of expressions

These features are not particular to Clojure, but rather to functional programming in general:

```
(defn send-html-response
 "Html response"
 [client-socket status title body]
 (let [html (str "<HTML><HEAD><TITLE>"
   title "</TITLE></HEAD><BODY>" body "</BODY></HTML>")]
  (send-http-response client-socket status "text/html"
  (.getBytes html "UTF-8"))
 ))
```

Java Interoperability

Clojure offers excellent interoperability with Java libraries. In fact, for some basic classes Clojure does not even provide its own abstractions, instead you are expected to use the Java classes directly. In this HTTP server example, we take classes such as Readers and Writers from Java:

```
(ns clojure-http-server.core
(:require [clojure.string])
(:import (java.net ServerSocket SocketException)
(java.util Date)
(java.io PrintWriter BufferedReader InputStreamReader
BufferedOutputStream)))
```

Creating Java objects and calling them is very straightforward. There are actually two forms (as explained in this excellent Clojure introduction):

```
(def calendar (new GregorianCalendar 2008 Calendar/APRIL 16)) ;
April 16, 2008

(def calendar (GregorianCalendar. 2008 Calendar/APRIL 16))
```

Calling methods:

```
(. calendar add Calendar/MONTH 2)

(. calendar get Calendar/MONTH) ; -> 5

(.add calendar Calendar/MONTH 2)

(.get calendar Calendar/MONTH) ; -> 7
```

Here is an actual sample:

```
(defn get-reader
"Create a Java reader from the input stream of the client socket"
[client-socket]
(new BufferedReader (new InputStreamReader (.getInputStream client-
socket))))
```

However, for some structures we decided to use the Clojure way. The original Java code uses `StringTokenizer` which goes against the pure functional principle of immutable objects and no side effects. Calling the `nextToken`() method not only has side effects (since it modifies the `Tokenizer` object) but also returns a different result when called with the same (non-existing argument).

For this reason we used the Split function of Clojure which is more “functional”:

```
(defn process-request
"Parse the HTTP request and decide what to do"
[client-socket]
(let [reader (get-reader client-socket) first-line
 (.readLine reader) tokens (clojure.string/split first-line #"\s+")]
(let [http-method (clojure.string/upper-case
 (get tokens 0 "unknown"))]
(if (or (= http-method "GET") (= http-method "HEAD"))
(let [file-requested-name (get tokens 1 "not-existing")
[...]
```

Concurrency

Clojure was designed with concurrency in mind from the beginning and not as an afterthought. It is very easy to write multithreaded applications in Clojure since all functions implement by default the `Runnable` and `Callable` interfaces from Java allowing any method to run into a different thread on its own.

Clojure also provides other constructs specifically for concurrency, such as atoms and agents, but we didn't use them in the HTTP server example, preferring instead the familiar Java Threads.

```
(defn new-worker
"Spawn a new thread"
[client-socket]
(.start (new Thread (fn [] (respond-to-client client-socket)))))
```

Order of methods matters

One thing that we noticed is that the order of methods inside the source file is critical. Functions must be defined before they are first used. Alternatively, you can use the declare special form to use a function before its actual definition. This reminded us of the C/C++ way of doing things, with header files and function declarations.

CEYLON

“Ceylon is inspired by Java. We've tried to make a more powerful language, always keeping in mind that we didn't want to make it _worse_ than Java - that is, we didn't want to break the things Java does well, or lose the qualities that make Java such a good language for writing large, stable programs in a team environment.”

GAVIN KING
creator of Ceylon

Getting started with Ceylon

As a Java developer you would expect to be able to adopt Ceylon very quickly. Basically, to start coding it is enough to walk through the Tour of Ceylon on the project homepage - you will get most of the information about the language from there.

The very first thing we will notice about Ceylon is that its "infrastructural" part is very well prepared. By this we mean modules. Ceylon runtime is based on JBoss Modules and Ceylon uses modules for everything. Even your brand new project is actually a Ceylon module, which is declared via module.ceylon file in the project directory:

```
module com.zt '1.0.0' {
  import ceylon.interop.java '0.4.1';
  import java.base '7';
}
```

This means that the module is called `com.zt` and its version is `1.0.0`. One thing to notice about the module declaration format is that the module name is not in quotes and the version is in quotes, e.g. `'1.0.0'`. That is a bit strange since the version is a number and package isn't. Why not skip the quotes for the version? Probably because version might contain non-numeric characters, like `1.0.0-SNAPSHOT`. But then it would make sense to skip the quotes entirely.

Ceylon makes a great deal of modularity and the runtime is built on top of JBoss Modules. Why is that important?

First, because without modularity you can't fix mistakes without breaking things.

Second, because without modularity you can't be cross-platform. Java's monolithic SDK means I can't define what it means to run a Java program in a JavaScript virtual machine. Ceylon's "tiny" language module works just as well in a web browser as on the server.

Third, because without modularity you can't build tools that work with module artifacts and module repositories instead of individual files sitting on your hard drive.

Fourth, because without modularity at the language level, you grow monstrous overengineered technologies like Maven and OSGi. And you wind up with bloated monolithic platforms like Java SE and Java EE (prior to EE 6).

GAVIN KING
creator of Ceylon

Next, the main file of the project is `run.ceylon` and we can just implement `run()` method to launch our app:

```
void run(){
   print("Hello from Ceylon!");
}
```

Methods (or functions) in Ceylon can be standalone and do not belong to any class. Same applies to attributes. Actually, it was quite interesting to find out what it compiles to. Every standalone attribute (or field) and method are compiled into a dedicated class. That means that the following code is compiled into 2 classes, `port_.class` and `run_.class` - which will also include the public static void main method.

```
import java.net { ServerSocket, Socket }
import java.lang { Thread }

shared Integer port = 8080;

void run(){
  ServerSocket server = ServerSocket(port);
  print("Listening for connections on port " port "...");

  while(true){
    Socket socket = server.accept();
    print("New client connection accepted!");
    HTTPServer httpServer = HTTPServer(socket);

    Thread threadRunner = Thread(httpServer);
    threadRunner.start();
  }
}
```

In the process we learned that Ceylon tooling compiles and packages everything into Ceylon ARchive (.car) and cleans up after itself, so you won't find class files after compilation - we had to unpack the archive to get access to the class files.

One nice part of the tooling was that when a new dependency is added into `module.ceylon` file the dependency was automatically downloaded from Ceylon Herd (http://modules.ceylon-lang.org) and resolved by the IDE. That's another win from the infrastructure of the language.

Java Interoperability

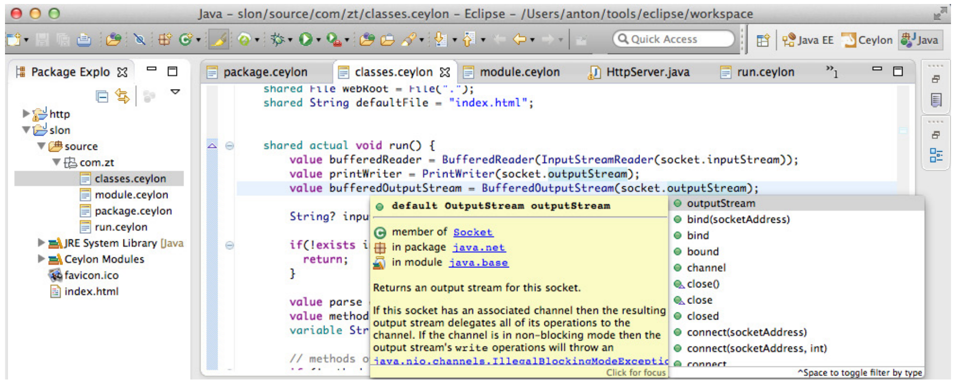
Java interoperability was a tricky part for me. First of all, you have to add dependency into module.ceylon to be able to use Java classes:

```
import java.base '7';
```

When you're using Java classes in Ceylon, intuitively you would expect all the same methods to be available on the Java class instance as if you would normally use it from Java program. In Ceylon it is not like that. Here's an example:

```
value printWriter = PrintWriter(socket.outputStream);
```

You would expect to be able to use the `getOutputStream()` method on `socket`, but Ceylon interprets it as a getter for a field and replaces its semantics with a property access (perhaps?). Even if `getOutputStream()` isn't really a getter, if you check its source.


Java – slon/source/com/zt/classes.ceylon – Eclipse – /Users/anton/tools/eclipse/workspace

```
    shared File webRoot = File(".");
    shared String defaultFile = "index.html";


    shared actual void run() {
        value bufferedReader = BufferedReader(InputStreamReader(socket.inputStream));
        value printWriter = PrintWriter(socket.outputStream);
        value bufferedOutputStream = BufferedOutputStream(socket.outputStream);

        String? inpu

        if(!exists i
          return;
        }

        value parse
        value method
        variable Str

        // methods o
```

default OutputStream outputStream

member of Socket
in package java.net
in module java.base

Returns an output stream for this socket.

If this socket has an associated channel then the resulting output stream delegates all of its operations to the channel. If the channel is in non-blocking mode then the output stream's write operations will throw an java.nio.channels.IllegalBlockingModeExceptic

The second challenge we faced was about type conversion. There's a special mapping between Ceylon types and Java types.

So `fileData` must be of a **byte array** type, and according to the type mapping rules it should be declared as `Array<Integer>` in Ceylon:

```
Array<Integer> fileData = arrayOfSize {
  size = fileLength; element = 0;
};
```

Oh my.. no luck:

```
Exception in thread "Thread-0" java.lang.ClassCastException:
  [J cannot be cast to [B
  at com.zt.HTTPServer.run(classes.ceylon:59)
  at java.lang.Thread.run(Thread.java:722)
```

Apparently, for type conversion to work correctly we have to import special Java interoperability module (in `module.ceylon`):

```
 import ceylon.interop.java '0.4.1'
```

Then it is possible to use a special helper method provided by the interoperability module. Here's the code that required the type conversion:

```
Array<Integer> fileData = createByteArray(fileLength);
value fileIn = FileInputStream(file);
fileIn.read(fileData);
```

Actually it's a truly painful corner case. Our usual strategy for handling Java's primitive types is unsound for container types, which is usually perfectly fine, since Java generics don't abstract over Java primitive types: you can't have a List<int>. But then Java has arrays, which are this special case container type which _do_ abstract over primitive types, and so we need a special case of a special case to deal with them. This will be fixed in the future versions of the compiler.

GAVIN KING

Basically, after all this issues the HTTP server implementation was running but then one more surprise appeared at runtime: null checks.

Now the application started but after submitting a GET request, it failed with the following stack trace.

```
Exception in thread "Thread-1" java.lang.NullPointerException
  at com.redhat.ceylon.compiler.java.Util.checkNull(Util.java:478)
  at com.zt.HTTPServer.run(classes.ceylon:19)
```

The surprising part was that at classes.ceylon:19 there was the following line for code:

```
String input = bufferedReader.readLine();
```

This is where some knowledge of how to read decompiled code using javap helped. The bytecode revealed that Ceylon compiler inserts null checks after the returns from methods of Java classes. So to workaround we had to use the '?' suffix for the declaration to fix that:

```
String? input = bufferedReader.readLine();
```

But then there's a problem of how this value is used. For instance, if it is used as a parameter for `StringTokenizer`, then instantiating the class would fail as well. So here's just a lame fix to make it work:

```
if(!exists input){
  return;
}
```

There definitely should be a better way on how to handle this situation despite it solved the immediate problem.

If you have a case where you _really_ don't want to do something with the null case, you can just write "assert (exists input);" and that will narrow input to a non-null type.

GAVIN KING
creator of Ceylon

KOTLIN

“ *We think of Kotlin as a modern language for industry: it is focused on flexible abstractions for code reuse and readability, static type safety for early error detection and explicit capturing of intent for maintainability and clarity. One of the most important use cases for Kotlin is a big Java codebase whose developers want a better language: you can mix Java and Kotlin freely and migration can be gradual and doesn't have to alter entire codebase.* ”

ANDREY BRESLAV
creator of Kotlin

As a creation of JetBrains, support for other IDEs was the first 'challenge' we was faced with when investigating Kotlin. As most of us are Eclipse users, switching to the IntelliJ IDEA environment is always difficult, but it's a necessary step if you want to code in a rich Kotlin environment. It's fairly easy to install the Kotlin plugin, and create Kotlin artifacts, but just a shame the support isn't also being pushed to other IDEs. Maybe we can subtly hint this to the JetBrains team? ;)

Elegant coding

Coding in Kotlin really does produce some very elegant code. It removes the need for null checks, uses primary constructors, smart casts, range expressions... the list goes on. Let's take a look at an example.

From our Java background we did like the combination of the `is` and `as` casting with the `when` structure. In Java terms, consider them as `instance of`, cast - `(A) obj` and switch respectively. The `is` usage will also infer a cast if you go ahead and use the object straight away. E.g: `if (stream is Reader) stream.close()`. In this example, the `close()` method is being called on the Reader interface. This would be the same as saying `if (stream is Reader) (stream as Reader).close()` but the extra code is not needed. This in combination with `when` allows you to switch over a variable, but not just using its value as you can get a richer involvement. Consider the following:

```
when (stream) {
  is Reader -> stream.close()
  is Writer -> stream.close()
  is InputStream -> stream.close()
  is OutputStream -> stream.close()
  is Socket -> stream.close()
  else -> System.err.println("Unable to close object: " + stream)
}
```

This is really clean, elegant code if you consider how you might want to implement this in Java. C# interestingly has a very similar usage of `is` and `as`, and also implements nullable types.

Method calls make use of parameter naming as well as defaults. It can be such a headache when a method takes in 5 booleans as you have to call the method very carefully so you pass `true` and `false` in the right order. Parameter naming gets round the confusion by qualifying the parameter name with the parameter itself on the method invocation. Nice. Again, this has been done in the past by other languages and scripting frameworks, but it's a welcome addition to any language, particularly with parameter defaults, which allow the method invocation to omit certain parameters if the user is happy to accept default values. Lets take a look at an example:

```
private fun print(out : PrintWriter,
                  pre : String,
                  contentType: String = "text/html",
                  contentLength : Long = -1.toLong(),
                  title : String, body : () -> Unit)
```

Here we have a method with several `String` parameters as well as a function as input. It is invoked using the following call. Notice that `contentType` and `contentLength` are both omitted from the invocation meaning the defaults in the declaration are used.

```
    print(out = out,
          pre = "HTTP/1.0 404 Not Found",
          title = "File Not Found",
          body = {out.println("<H2>404 File Not Found: " +
                  file.getPath() + "</H2>")})
```

So what does this mean? Well, there will be much less method overloading! Seriously though, it's amazing to think Java gone over 20 years without additions like this? Sometimes it does feel like you're coding in the dark a little. Come on Java, catch up!

Kotlin will help you write safe code, unless you don't want to!

First off, you can say goodbye to NPEs! Kotlin uses 'nullable types' and 'non-nullable types' to differentiate between vars which could be null, and those which will never be null. Consider the following code:

```
var a : String = "a"
a = null // compilation error
```

To allow nulls, the var must be declared as nullable, in this case, written `String?`:

```
var b : String? = "b"
b = null // valid null assignment
```

Now, if you call a method on variable 'a', it's guaranteed not to cause an NPE, so you can safely say

```
val l = a.length()
```

But if you want to call the same method on b, that would not be safe, and the compiler reports an error:

```
val l = b.length() // error: variable 'b' can be null
```

By knowing which vars can be null, the Kotlin compiler mandates that when you dereference a nullable type, you do so using one of the following methods:

Safe calls in Kotlin are very similar to those in Groovy, including the notation. By dereferencing a nullable type using `'.?'` like in the example below, tells the compiler to call length on object b, unless it is null, in which case do nothing.

```
b?.length()
```

Those of you who are thinking 'What? Get rid of null? Where's the fun in that?", there is the `!!` operator which allows for the potential of a NPE to be thrown, if you so wish.

You can also use the `?` notation with `as` to avoid exceptions being thrown if the cast is not possible. This is called a safe cast.

Functions

Functions can be created inside (member functions) or outside of a class. A function can contain other functions (local functions) and you can use functions to extend existing classes such as the following:

```
fun Int.abs() : Int = if (this >= 0) this else -this
```

This example extends the `Int` class to return the absolute value it contains.

Functions are very powerful and are well used on the JVM in many languages (Still not in Java, until Lambdas appear in Java 8). Kotlin also allows the use of higher order functions which means you can pass a function as an argument to a method call (function literal).

Documentation/Help

We found that the documentation for Kotlin was all in the same place.
All Google searches led back to the community site. This is a shame, as it would be great to go somewhere else with other examples and resources with which to play around with, but it's still quite young. It would be nice to see more code in Github and the like, for others to follow.

XTEND

“*Xtend is a statically typed programming language which is translated to readable Java source code. It supports existing Java Idioms and is designed to work even better with existing Java APIs than Java itself. Xtend is a flexible and powerful tool to build useful abstractions and comes with advanced Eclipse IDE integration. It’s an open source project at Eclipse.*”

SVEN EFFTINGE
creator of Xtend

Xtend self-proclaims itself as 'Modernized Java'. The language has placed itself as 'faster than Groovy, simpler than Scala and incorporates all the benefits of Java'.

Xtend is built on top of the Xtext language development tool, which gives it great integration with the Eclipse IDE. Well it's not hard to see why, as both Xtend and Xtext are eclipse projects! Other IDE fanboys will not get tight integration with their IDE of choice, in a similar way to Kotlin preferring IntelliJ IDEA.

On your marks, Get set, Code!

We went straight to Eclipse to get some code down and noticed that it's actually quite familiar ground (as a Java developer). The Xtend language syntax has remained fairly true to Java, which could be considered a good or a bad thing, depending on your impressions of the Java syntax :) Yes it is simpler as the the usual Java baggage has been removed (as with most new langs) leaving the code more readable and only using structure and key-words where needed, leaving us with a readable, cleaner, implementation. Lets take a look as some code:

```
package com.zt

class Oppa {

       def static void main(String[] args)
       {
              println('Gangnam, Style')
       }
}
```

Note we don't like using "Hello, World" examples as you rarely get much out of them, so here is our "Gangham Style" example :) So far we see very little has changed from Java, but one thing we did notice in the IDE is an xtend-gen source folder which has interpreted the Xtend code into Java code:

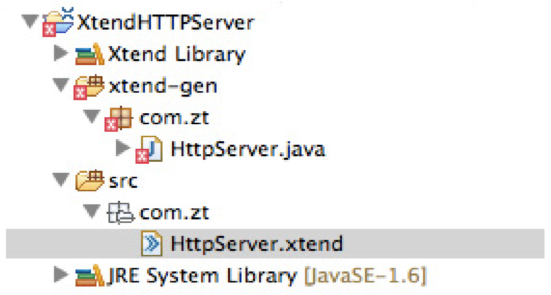
```
package com.zt;

import org.eclipse.xtext.xbase.lib.InputOutput;

@SuppressWarnings("all")
public class Oppa {
  public static void main(final String[] args) {
    InputOutput.<String>println("Gangnam, Style");
  }
}
```

Interestingly it pulls in it's own println implementation via the InputOutput Xtend class.

Overall in the Xtend HTTP server example, we wrote 137 lines of code, which was translated into 309 lines of Java. The Java created isn't particu-larly readable and it wouldn't be particularly nice to edit it, so I'm not really sure why it's shown.



Amusingly, at one stage we got into a state where the Xtend code compiled yet the Java code did not :)

Why did you make the decision to translate your Xtend code into Java code, rather than directly to bytecode?

"*Fast and optimizing Java compilers we have. Xtend code usually runs as fast as the equivalent but much more verbose Java code.*

By far the biggest advantage is transparency. Having your Xtend code translated to Java source lets you understand what the Xtend compiler actually does in detail. Java is well known and trusted, translating to it gives people the needed confidence. This is especially helpful when you are new to the language and want to learn how a certain snippet of code is translated and how the compiler works under the covers. You can always just have a look at the Java code. The Eclipse based Xtend IDE even provides a so called "Generated Code View" that shows the Xtend code and generated Java code side-by-side. You can see through selections what parts where generated from what Xtend code on a very fine-grained basis.

Also during debugging you can switch between debugging through Xtend or Java source code. Xtend is generated to what I call 'statementisized' Java code. Long, chained expressions which are typical for a functional programming style get translated to multiple statements with synthetic variables. So you can step over the individual expressions and inspect the intermediate results."

SVEN EFFTINGE
creator of Xtend

What can you do with Xtend that you can't do with Java?

This is where it gets interesting, as Xtend provides many of the features which Java lacks. For instance, it provides extension methods, import extensions, lambdas, data set objects, implicit typing and more. This is great! But you can list pretty much all of these off in many of the other languages described in this report. What makes this language any different? Well, in our opinion, it's not trying to be different, it's trying to bridge a gap between where Java is now and where it should be. It's Java on steroids.

Code snippets

Enough talk already! Need more pictures and code! Lets dive into some of the neat parts of the language, starting with extension methods.

Here we're creating a new method called `sendHtmlResponse` which takes 4 parameters, however there is a neat syntactical trick here which allows the first parameter to also be the object we're trying to extend.

```
def sendHtmlResponse(Socket socket, String status, String title,
String body){
    val html =
'''<HTML>
        <HEAD><TITLE>«title»</TITLE></HEAD>
        <BODY>
            «body»
        </BODY>
</HTML>'''

    socket.sendHttpResponse(status, "text/html", html.toString.
getBytes("UTF-8"))
}
```

This means we could actually call `sendHtmlResponse` on the `Socket` object.

```
socket.sendHtmlResponse(
    "HTTP/1.0 404 Not Found",
    "File Not Found",
    "<H2>404 File Not Found: " + file.getPath() + "</H2>")
```

Of course in our xtend-gen directory, we're actually generating Java code which looks like this:

this.sendHtmlResponse(socket, "HTTP/1.0 404 Not Found", "File Not Found", _plus_3);

While this spoils the illusion, it is really nice to be able to extend classes and this is in fact where Xtend gets it's name from.

In our first code snippet we used the following construct:

```
'''<HTML>
        <HEAD><TITLE>«title»</TITLE></HEAD>
        <BODY>
            «body»
        </BODY>
</HTML>'''
```

This is called a template expression and is used in many other languages. It allows us to easily create HTML without tripping over `out.println()` statements, allowing us to better see what kind of doc we're creating. You'll notice we can escape the statement to drop in variables, and you're also able to use conditionals in the templates.

We did miss named parameters and defaults from other languages, which we still think just should be a part of all languages anyway! It does make use of implicit typing though which was nice. The language is still statically typed but the type of a variable is determined at compile time by the value of the expression it will be assigned. For example:

```
val WEB_ROOT = "."
val DEFAULT_FILE = "index.html"
var Socket socket
```

In this example the first two variables are clearly `Strings`, whereas the third cannot be determined at this point, so we need to declare its type is `Socket`. This makes the code a lot clearer compared to the following in Java:

```
private final String WEB_ROOT = ".";
private final String DEFAULT_FILE = "index.html";
private final Socket socket;
```

Some of the more syntactical differences took more than a second or two to work out, like 'Where have our class methods gone!') Oh, you have to use a different notation to get that - `File::separator`. I'm not really sure why this has been imposed but it's not a big issue so didn't bother us too much.

Interestingly there were issues whereby the IDE would complain about a million things when there was only one problem. This is quite frustrating as you have to fix things in the order the Xtend compiler wants you to, rather than the order you want to. Java tooling support still has the experience here and has a rare win on this topic.

Java Interoperability

While many languages can sometimes struggle to achieve Java interoperability, Xtend really doesn't share their problems as they share a near identical type system proving an easy mapping to and from the languages. You're able to directly call from Xtend to Java and vice-versa without issues. Enough said.

OVERALL SUMMARY

TL;DR (TOO LONG, DIDN'T READ)

This report has been awesome--educational, challenging and entertaining in a truly geeky way! It's been fun for us to learn these new languages and try to get to grips with what is different between each of the languages. They're all clearly quite different to Java simply in terms of the functionality they provide. Clearly the newer JVM languages all find closures important, and there is an interesting split on the billion dollar question around how to deal with nulls. Here's what we think you should take away from each of the languages:

Java

Java is still alive and kicking! We can definitely say that lambdas and the accompanied features (defender methods, collections library improvements) will have a great impact on Java very soon. The syntax is quite nice and once developers realize that these features provide value to their productivity, we will see a lot of code that leverages these benefits in the future.

Scala

We saw that Scala can be quite expressive, and allows us to leave out a lot of boilerplate syntax compared to Java. It's real advantages probably don't come out our HTTP Server example, but perhaps this gave you some idea of what the language is like. The key point, is to embrace immutability and try to model code as expressions that can be composed.

Groovy

As Java developers, Groovy is easy to get into, and a great choice for web apps, DSLs, templates and script engines. It approaches annoyances common in Java, packs a modern set of features and has a great standard library. Groovy will definitely continue to be widely adopted and a great choice for a dynamic language on JVM. We only fear the loss of focus, as the feature set continues to grow.

FANTOM

Fantom is a mature and stable programming language with a very fluent and elegant API. By no means do we claim to be experts, so please show us a better Fantom if you can: our Github repo is open to comments and pull requests. So if you judge a horse by its teeth, then so would you judge a programming language by its syntax. Fantom has a very Java-like syntax, with no semicolons and with a number of its own keywords thrown in. Otherwise, its a nice syntax to use coming from a Java land for sure.

Clojure

Clojure offers a great platform for those looking for a functional language running on the JVM. Developers with experience in Lisp/Scheme will feel right at home. However, Java developers will not only face a new syntax but also a new paradigm.

Although it would be possible to abuse Clojure and write code in a Java-like way, this is not optimal. The big strength of Clojure can be found in the concurrency constructs which could be handy in the future as more programmers are interested in parallelization of their code.

Ceylon

The syntax and the keywords of Ceylon are quite different from the ones that most developers are used to in Java. However, it did not disturb us that much. In most cases it was enough to associate the keyword from Ceylon with the one from Java, for instance, ‘satisfies’ in Ceylon vs ‘implements’ in Java.

Overall, it was quite an interesting experience. The impression we've got from Ceylon is that it is focused on the style of OOP and infrastructure (module system) rather than on conciseness of the expressions, not trying to squeeze a million of meanings into a one-liner. It actually means that Java verbosity is not something that Ceylon creators would be worried about, but focus rather on the ecosystem and language design at large.

PROJECT Kotlin

Overall Kotlin is a very enjoyable language to code with. Nothing about Kotlin is particularly groundbreaking, but what makes it great is that it cherry picks some of the best parts of other languages.

With features including functions, parameter names, default values and extending classes, the language is a pleasure to use. It’s also a pretty language! The keyword and ratio of fluff:useful code is much better in Kotlin using constructs such as ‘when’, ‘is’ and ‘as’.

With milestone releases every 2-3 months, Kotlin may soon be a language which many Scala beginners may turn to, although we don't think happy Scala users will be too interested in it.

Xtend

Xtend provides concise neatness with the functionality that Java has only promised in the future but currently lacks. The question is, would we recommend others to use it over and above Java or other JVM languages?

Well, for Java users it's all going to depend on how long these benefits last for, as once some of the nice feature additions in Xtend make it into the Java language, just as lambdas and method extensions, We’re not sure there's enough to justify the move. We imagine developers using other JVM languages such as Scala see Xtend more as Java n+1 or Java with a plugin rather than a language which breaks any boundaries, so we don't see why Scala users might break away to use Xtend.

BONUS 1-UP:
WE ASKED THE EXPERTS SOME MORE QUESTIONS...

QUESTION:

HOW DID YOU COME UP WITH THE IDEA OF A NEW PROGRAMMING LANGUAGE FOR THE JVM?

SVEN EFFTINGE - XTEND

The Java ecosystem has many good properties, like the many mature open-source frameworks or the extremely advanced tools. But the language itself is old, rigid and too limiting.

Xtend addresses the problems of the Java language holistically. It relies on and reuses the solid foundation of the Java ecosystem but gives you a dense and more flexible syntax. With Xtend you can finally write concise code and define powerful abstractions without switching to a completely different ecosystem.

BRIAN FRANK - FANTOM

We were looking for a language which had the right mix of features to use for our startup, especially portability and modern support for concurrency. Finding all the existing solutions lacking, we built Fantom as a platform from which we were able to build our commercial software technology.

GAVIN KING - CEYLON

Ceylon grew out of our experiences working within the JCP to improve Java, and specifically Java EE. I've been involved with several successful efforts to enhance the Java EE programming model, including JPA, EJB 3, and CDI. Eventually, we ran into diminishing returns here, and I felt that it was going to be very difficult to continue to significantly improve the platform without tackling problems that were rooted in Java SE, some in the Java language itself, and many more in the Java SDK. Unfortunately, whereas Java EE was able to reinvent itself, Java SE, for better or worse, has had such a strong commitment to stability that backward compatibility concerns have precluded fixing mistakes. So we felt it would be better to start with a somewhat cleaner slate.

From a slightly technical point of view, major motivating factors were the failure of Java on the client, and the consequent need for a language that works on both client and server, the failure of the Java ecosystem to provide good UI frameworks, and the language-level limitations that make it difficult to develop generic frameworks and libraries in Java.

ANDREY BRESLAV - KOTLIN

We have a huge Java codebase developed over a decade, IntelliJ IDEA, and we wanted to switch to a better language without abandoning the code we already have.

After examining existing alternatives, we found that none of them fulfills our requirements (that are rather moderate: smooth integration with the existing infrastructure, efficient tooling possible, good compiler and runtime performance), and as we believe that we are not unique in this sense, we decided to leverage our expertise in programming languages (seven IDEs on the market today, and each one is almost a compiler), and create a language that people need.

QUESTION:

WHAT WAS THE PROGRAMMING LANGUAGE THAT INSPIRED YOUR LANGUAGE THE MOST?

SVEN EFFTINGE - XTEND
Definitely Java, as it is the language we know best and have a lot of experience with. Beyond that Xtend borrows features from many different languages, including Python, Smalltalk, C#, Scala and Lisp.

BRIAN FRANK - FANTOM
Fantom is inspired by a whole mix of languages including Java, C#, Ruby, Python, Clojure, Eiffel, Self, and Lisp.

GAVIN KING - CEYLON
Ceylon is inspired by Java. We've tried to make a more powerful language, always keeping in mind that we didn't want to make it _worse_ than Java - that is, we didn't want to break the things Java does well, or lose the qualities that make Java such a good language for writing large, stable programs in a team environment.

On the other hand, we've also paid lots of attention to other languages we like, which I guess for me would include Smalltalk and ML, and yeah, I've even done my time wading through some of the academic literature.

JOCHEN THEODOROU - GROOVY
I think there is not really a most inspirational thing. Groovy is a community driven language with users providing and adapting ideas from other languages. And we have all kinds of people. Many say that Groovy is a bit like the marriage of Ruby and Java. When I did come to Groovy I didn't know Ruby, to me it looked more like Smalltalk. And 2 years earlier I would have maybe thought of JavaScript and before that LISP. I think there is no clear "most".

ANDREY BRESLAV - KOTLIN
It's hard to name only one. Java, Scala, C#, Groovy were the biggest influencers. We also learned from many others, including Objective-C, Gosu, Spec#, Python, Eiffel and so on.

QUESTION:

WHAT ARE THE FEATURES YOU WOULD LIKE TO SEE IN JAVA.NEXT?

SVEN EFFTINGE - XTEND

That would of course exactly be the features you see in Xtend plus some you will see in the near future. :-)

BRIAN FRANK - FANTOM

I see the two cornerstones of Fantom as being the most important for any next generation language: portability and concurrency. To me, it seems extremely non-productive for most software projects to have one codebase in JavaScript for the front end and an entirely different codebase for the backend (often in Java, C#, Ruby on Rails, PHP, etc). Concurrency in most mainstream languages is completely broken - where memory between threads is shared and programmers have to explicitly remember to lock data structures for concurrency. Languages like Go and Clojure are really interesting in this respect.

GAVIN KING - CEYLON

Modularity, clearly. Local type inference would be nice, and I think it would fit reasonably into the language. This silly diamond syntax thingy should have been passed over in favor of doing it right.

ANDRES ALMIRAY - GROOVY

Collection literals would be a great boon, a nice compliment to lambdas. It's very likely will see them in JDK9 however the roadmap has yet to be set on stone. However in my mind there's one particular feature that rises to the top: AST manipulation. Groovy has this feature (we call it AST transformations) and I must confess is the best thing since Groovy was released (which happens to be the best thing since sliced bread, no really, it is ;-).

ANDREY BRESLAV - KOTLIN

Declaration-site variance and some way of storing collections of primitives without so much performance overhead.

QUESTION:

HOW HARD IS IT TO GET JAVA INTEROP WORKING FOR YOUR LANGUAGE? WHAT ARE THE MAIN CHALLENGES?

SVEN EFFTINGE - XTEND

Many JVM languages take Java interoperability just as the plain possibility to call out to Java functions somehow. With Xtend, we take Java interoperability seriously since we want to leverage existing Java APIs. Xtend doesn’t bring its own big standard lib, but relies on the JDK and Google Guava. Both are well written and mature Java libraries and we wouldn’t dare to rewrite those since they are already very good and solid. There are so many more interesting things that we can work on :-)

The basis for 100% interoperability is the exact same type system. Xtend doesn’t go overboard and introduce all kinds of new types. Instead it’s 100% the same as in Java. Things like generics or auto boxing which 10 million Java devs are used to, work exactly as in Java. Also method resolution and method overloading are exactly like in Java.

But Xtend goes further than that. We have carefully analyzed and identified commonly used Java idioms and made sure they are well supported by Xtend. This starts with the obvious support for JavaBeans properties and doesn’t end with how you pass event handlers. Also with extension methods you can add new methods to existing types easily.

Therefore Xtend is often even more interoperable with existing Java APIs than Java itself.

BRIAN FRANK - FANTOM

Fantom already has really good interop. But like all JVM languages which provide features not found in Java (like first class functions), care must be taken for Fantom classes to present a good API to Java-land. But I think the main use case for interop is for Fantom programs to consume Java libraries - and this is a great strength of Fantom as a JVM lang.

GAVIN KING - CEYLON

It was (and continues to be) extremely difficult. Ceylon has a quite different type system to Java, and mapping between the two type systems is difficult and painful. We deliberately chose _not_ to compromise the design of our type system to take into account the features of the JVM, since Ceylon is not just for the JVM. Instead we decided to wear the pain of writing code to transform between the two type systems.

GUILLAUME LAFORGE - GROOVY

There's a big step between mere interoperability and full integration.

Groovy made the choice for the latter.

It means we made some decisions to stay as close as Java as possible, so ruling out some more opinionated approaches, but on the other hand, the benefits are clear in that we've got the best seamless experience when mixing both Java and Groovy together, using each language for what it's best at.

With this approach, we're not reinventing the wheel, believing we can be smarter than all the other kids, and instead we adapt to the (JVM+Java) environment all developers are used to, and avoid the costs associated with non-standard approaches.

ANDREY BRESLAV - KOTLIN

For the user it's very easy: just works out of the box. For us, it's a lot of work. The challenges include using Java classes in a smart way while they lack the necessary type information: nullability, declaration-site variance, mutability of collections and such. Currently we are working on a tool that performs additional analyses on Java binaries and thus infers more precise type information. Other challenges concern well-known design mistakes made mostly in Java's early days, like covariant arrays and such.

QUESTION:

IF YOU HAD AN OPPORTUNITY TO DESIGN YOUR LANGUAGE FROM SCRATCH AGAIN, WHAT WOULD YOU DO DIFFERENTLY?

SVEN EFFTINGE - XTEND

Not too much. It’s a relatively young project and we are very careful with the features and APIs we add.

BRIAN FRANK - FANTOM

There probably isn't too much I would change actually. Fantom has grown over the years from feedback and usage by lots of very smart people. And I don't think we've ever felt hamstrung to make a major improvement because of an old design decision. If I had to pick one thing I would change, it would be to make Fantom an expression-oriented language instead of a statement-oriented language. I would probably also use a more Java/C# like function syntax instead of the Ruby inspired syntax.

GAVIN KING - CEYLON

For the M5 release, we just redesigned some significant things, so you can check out the release notes! There's so far nothing that I feel is wrong but it's too late to fix. Ask me again in a couple of years.

JOCHEN THEODOROU - GROOVY

Many things in Groovy are like they are because one design goal of the language is to have a high level of integration with Java. If you keep that goal, then there is not really much I would do different that I cannot do in future Groovy versions. If I would not care about Java integration at all, then I would do a lot of things very different. But then there would probably be no users for the language ;)

ANDREY BRESLAV - KOTLIN

I would think faster and make no mistakes I've made and had to correct later :)

QUESTION:

SOME JVM LANGUAGES SEEM TO HAVE A BIT DIFFERENT APPROACH TO THE “BILLION DOLLAR MISTAKE” (I.E. NULL VALUES). WHAT IS THE RATIONALE FOR YOUR APPROACH TO THIS?

SVEN EFFTINGE - XTEND

Today Xtend supports the safe operator and the elvis operator, to navigate nullable types easily. Also extension methods and the switch expression help to handle null pointers.

Than there is the Optional type in Guava and soon a similar one in Java 8.

That said, we plan to work on nullable and non nullable types, but it won't compromise Java interoperability.

BRIAN FRANK - FANTOM

Nullability is deeply baked into the Fantom type system. A type can't contain null unless marked with the "?" character. So we have tackled this problem head-on. All of our APIs are clearly typed as accepting or returning null. So its a big feature in our static type system. But like all type system features, I see this as just one tool in the toolbox.

GAVIN KING - CEYLON

A design principle that we hold very dear is to not build primitive types, special cases, or ad hoc exceptions into the type system. Ceylon has a very powerful type system that is easily capable of representing and abstracting over things like "a string or null" (an optional type) or "a sequence of a string, followed by an integer, followed by another string" (a tuple). So we represent these kinds of things within the language itself, then provide a little syntax sugar for convenience. So we have a class named Null written in Ceylon, and to represent an optional string, we use a union type String|Null, which we then let you write as String?. Or to represent tuples, we have a recursive generic class named Tuple and some syntax sugar to make it easy to use.

Speaking more directly about null, Ceylon's approach is quite different to other languages, because our null is neither a primitive value assignable to the bottom type (like in Java), nor do we need a wrapper class to represent optional values (as in ML, for example). The reason Ceylon is so different here is that, rather uniquely, Ceylon supports union and intersection types as a really basic feature of the language. Once you start to get the feel for Ceylon, you'll realize what a really special thing this is.

JOCHEN THEODOROU - GROOVY

If Hoare had decided different in the 1965s I would probably think different of this, but we are all kids of our time. For Groovy null is just another object with methods and properties. You can even add them using runtime metaprogramming. Sure, we have the null-safe navigation, but other than that, we are more to keen on embracing null, then trying to banish it like a maybe demon in Haskell. Adding it to the type system is currently no option for me, not even for the static compiler.

ANDREY BRESLAV - KOTLIN

Kotlin uses a notion of a "nullable type". References in Kotlin do not admit nulls by default, you have to specify nullability explicitly in the type. You are not allowed to dereference a nullable pointer without an explicit check. This effectively guarantees NPE-free code, unless you do really bad things, e.g. use Java to throw bad data at Kotlin, and even then we blow early and give reasonable error messages.

QUESTION:

IS THERE A PARTICULAR ASPECT OF YOUR LANGUAGE YOU WOULD LIKE TO EMPHASIZE?

SVEN EFFTINGE - XTEND

I'm very excited about a new feature called Active Annotations. It is basically a much better annotation processing. With that you can can participate in the translation process from Xtend to Java code. For instance the @Property annotation which generates JavaBean getters and setters for an annotated field is an active annotation. This is not new and is basically Lisp macros brought to the Java landscape. The speciality is that it is very easy to build and deploy your own active annotations and that it works nicely with the type system and therefore Eclipse. The IDE is 100% aware of any changes you do through active annotations and they become instantaneously available as you hit save. Also you will be able to do additional compiler checks, add default imports and even provide quick fixes. Of course it is a huge advantage to be able to look at the generated Java code as it let's you easily understand what your processors do.

I think this has big potential and I'm already having a lot of fun with it.

We are releasing a new version of Xtend on March, 20. It will contain a provisional version of the Active Annotation API with which you can already do very cool things.

BRIAN FRANK - FANTOM

By far the two most import aspects of Fantom are portability and concurrency. Being able to leverage one codebase for both HTML5 front end and a fast, robust JVM backend is a huge productivity booster. Software is complex enough already, having two separate languages, tools, and codebases for frontend vs backend only adds a new level of complexity to software projects. We leverage the type system to prevent shared mutable memory between threads and use the actor model for concurrency. This has proven to be a godsend in creating robust software - I can't remember the last time I debugged a race condition or deadlock (always the worse bugs to debug!).

GAVIN KING - CEYLON

OK, so the language is quite feature-rich, and ticks the boxes, and has just the right amount of cutesy syntax sugar without going overboard with crazy weirdo combinations of ASCII symbols. But that's not what makes it powerful. What makes it powerful is that these features are layered over and grow out of a very clean, simple, consistent core designed according to _principles_.

ANDREY BRESLAV - KOTLIN

Too many of them. :) But I'll tell about one here: tackling declarative data with builders.

Builders were something only possible in dynamic languages before we Kotlin supported them in a completely type-safe way. For example, you can write type-safe HTML and CSS in Kotlin and have true reuse of declarative structures (as opposed to rather cumbersome and sometimes limited template languages and LESS/SASS).
The Kara web framework is doing a rather good job there. It's under development, but what they have so far is very impressive. Just google for "Kotlin Builders", you'll like it.

QUESTION:

PROGRAMMING LANGUAGE DESIGN IS ABOUT COMPROMISES. IS THERE ANY FEATURE THAT YOU HAD TO DROP BECAUSE IT DIDN'T FIT YOUR CONSTRAINTS OR BECAUSE OF JVM LIMITATIONS?

SVEN EFFTINGE - XTEND

There are thousands of features one could add but it has never been a compromise to leave them out. If you add a feature you can never remove it again.

It's important to grow a language carefully and slowly - maybe not as slowly as Java :-)

BRIAN FRANK - FANTOM

The JVM has been a great platform for running Fantom. But the JVM was definitely designed for the Java language. There are lots of irritating design decisions in the JVM such as the lack of function pointers or a true invokenonvirtual opcode. But with the latest work on method handles and invokedynamic, I think the JVM is really maturing as a platform for functional and dynamic languages too.

GAVIN KING - CEYLON

We'll see whether we need to drop reified generics. Stef has a prototype implementation that we're experimenting with now. The question is what will it cost in terms of performance.

We had to entirely drop -introductions-, because they didn't fit the principles of our type system. And that was something I _really_ wanted, right from my earliest ideas for a new language.

ANDREY BRESLAV - KOTLIN

Most notably — reified generics. It turns out that we'd have to completely spoil Java interop to implement them, and considering the relatively little benefit they bring, it's not worth it.

QUESTION:

IT LOOKS LIKE MOST OF THE RECENT JVM LANGUAGES TAKE ADVANTAGE OF STATIC TYPES. HAVE STATICALLY TYPED LANGUAGES WON THE RACE?

SVEN EFFTINGE - XTEND

Advanced IDEs like Eclipse, Netbeans and IntelliJ need static type information to work well. Most of the time static types don't get into your way and with the kind of type inference we have in Xtend you rarely have to write them, where they don't contribute to the readability of the code.

That said, sometimes you don't want to use static types, which is why we have a reflection API in Java. I think a dynamic type like they have in C# is a good idea. Also the method-missing feature most dynamic languages have comes in handy at times.

BRIAN FRANK - FANTOM

I think the proponents of static languages promote type systems as a sort of religious issue. So many programming issues get cast as black or white, when the reality is grey. We have a more pragmatic approach to type systems. They are one tool in the toolbox. They are extremely helpful in documenting APIs and catching bugs for "code in the small".   But when you start to work with ad-hoc data or get into gluing different subsystems together, a static type system is not necessarily the most productive solution - so I feel its really important to have that dynamic escape hatch when its the better solution.

GAVIN KING - CEYLON

Well, they are certainly where the action is. They're certainly the most _interesting_ kind of programming language!

The way I look at it is: it's very easy to design a very expressive language, in the sense of giving the programmer a lot of freedom to write whatever they want in any way they like. Unfortunately, code written in a language like that is impossible for a tool to understand, and even very difficult for other programmers to understand. So if you want to help the programmer create maintainable code, you need a language with _rules_. These are rules that the language designer writes down that the compiler and other programmers use to reason about the code. And these rules get in the way! So there is this struggle to make the language more expressive without compromising the system of rules. And its just fascinating.

CÉDRIC CHAMPEAU - GROOVY

Absolutely not. I think static languages also borrow from dynamic languages (otherwise invokedynamic wouldn't have been necessary) and the opposite is true too. What I lilke about Groovy is that you have the choice, without needing to change of language. Some parts of your code may need more type safety or more performance and you *can* choose to be static here, but in general, it's not necessary and you can benefit from multiple dispatch and duck typing.

Also it's not true that the new languages are all static. The last one I know of, Golo, is dynamic ;) http://golo-lang.org/

ANDREY BRESLAV - KOTLIN

I don't think there can be a winner there. There was a lot of confusion, because old statically typed languages had no/little type inference, so they became too verbose and ugly. Modern statically typed languages look almost as nice as their dynamic rivals, but provide better tooling support and runtime performance.
On the other hand, there's a lot of "dynamic magic" some people like so much, and quite a lot of it is not possible in a statically typed language. That's why many statically typed language introduce "dynamic types" or something similar...
I think eventually everyone finds their niche: dynamic languages for small, short-lived, quickly written projects, static ones for bigger, longer-lived, more effort-demanding ones.

QUESTION:
HOW DO YOU SEE THE JVM LANGUAGE LANDSCAPE IN 5 YEARS?

SVEN EFFTINGE - XTEND
I have no idea whether there will be more languages but I think regarding the platforms there will be much more uses of Dalvik. Android is going to be used in the embedded industry much more and even gaming platforms start to use it. There’s definitely a bright future for Java.

I believe that Java will remain the dominant language on Dalvik and the JVM. And I hope it improves, I especially look forward to project Jigsaw. Xtend will remain staying three steps ahead of Java, since there is much less legacy we have to carry with us.

BRIAN FRANK - FANTOM
I don't think much will change over the next 5 years. I think Java will continue to be the main language used for the JVM. But it is exciting to see so many emerging new options for the JVM.

GAVIN KING - CEYLON
I think the JVM platform will remain very popular, and that it will be increasingly common to see people writing programs for the JVM in languages other than Java. It's very hard to guess _which_ languages will be most the popular. Certainly Java will be one of them.

ANDRES ALMIRAY - GROOVY
Each day that passes by Kotlin and Ceylon get closer and closer to reaching the 1.0 status. They have the potential to grab a sizeable piece of the JVM space. Considering both are statically oriented, sporting similar features but very different goals I wouldn't rule out a third one entering the fray in the next 5 years. I wouldn't count out another dynamic language making the scene either.

ANDREY BRESLAV - KOTLIN
Java's market share will be smaller, i.e. in total, other languages will attract more people than today, but it's hard to say who will get what.

QUESTION:

IN YOUR OPINION, HOW IMPORTANT IS THE TOOLSET FOR YOUR PROGRAMMING LANGUAGE?

SVEN EFFTINGE - XTEND

A slick editing experience is super important. Things like content assist, navigation and refactorings are crucial. Also auto formatting, quick fixes and automatic management of import sections is something I don't want to live without anymore. But it has to be fast and slick. Also as with languages I don't like tools which are overly polluted with bells and whistles.

But people have different workflows and preferences, so Xtend can of course be used from the command line with e.g. Maven or Ant. Especially the maven support has been improved for the upcoming release. We are also going to work on supporting other editors and IDEs.

BRIAN FRANK - FANTOM

Strong tooling and IDEs are critical for any programming language. I think early adopters tend to be programmers who like Emacs or Vi, where the tools aren't as important. But tools are absolutely required for adoption by more mainstream programmers.

GAVIN KING - CEYLON

Utterly central. Statically typed languages are languages for tools. The extra productivity you gain from using a statically-typed language with an IDE, compared to using a dynamically-typed language is _huge_ as soon as you're working with a codebase of any size. I expect that almost nobody will write Ceylon code without the use of an IDE. Though, if you want to try, there is a vim mode available...

I personally invest a lot of my development hours in Ceylon IDE.

JOCHEN THEODOROU - GROOVY

Java set a new standard when it comes to IDEs. I remember me programming in nothing but a text editor with only syntax highlighting even in the late 90s and still producing thousands of lines of code. For many this is no option anymore, thus it is important to have good tools like IDEs, but also like debuggers and others. It is a main point for adoption.

ANDREY BRESLAV - KOTLIN

In general it depends on what your project is. If you have many developers working on a long-lived project, the tooling is very important, since you simply have a lot of code to maintain, i.e. browse through and refactor. And Kotlin is targeted at this kind of projects.

QUESTION:

CAN YOU SUGGEST ANY GOOD READING FOR THE WANNABE LANGUAGE DESIGNERS?

SVEN EFFTINGE - XTEND

Yes, source code :-) Seriously, reading a lot of code in different languages definitely helps. I also like to read books and specifications about programming languages.

My team and I have been working on Xtext, a language development framework, for a couple of years now. Since than we have implemented and seen many different languages, which has taught us a pretty good understanding of what sound language design is. I think doing is at least as important as reading.

BRIAN FRANK - FANTOM

I think a great way to get involved with language design is to start reading code for compilers. I remember years ago, that I particularly liked the compiler for a .NET language called Boo. It inspired a lot of the design for Fantom's compiler.

GAVIN KING - CEYLON

My forthcoming book: "Design and implement your own JVM programming language: from language spec to IDE in just a few thousand commits!... Bazinga!!!"

CÉDRIC CHAMPEAU - GROOVY

For JVM language developers, I would tell them to read writings from Charles Nutter, as well as watching talks from the JVM language summit.

ANDREY BRESLAV - KOTLIN

Steven Muchnick's “Advanced Compiler Design and Implementation”, Benjamin C. Pierce's “Types and Programming Languages”, and of course, Dostoyevsky's "Crime and Punishment", it's a very good book and no language designer should miss it.

RebelLabs is the research & content division of ZeroTurnaround

Contact Us

Twitter: @RebelLabs
Web: http://zeroturnaround.com/rebellabs
Email: 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

Czech Republic
Osadní 35 - Building B
Prague, Czech Republic 170 00
Phone: +372 740 4533

This report is brought to you by:
Simon Maple, Anton Arhipov, Erkki Lindpere, Kostis Kapelonis, Oleg Šelajev, Mirko Adari, Ryan St. James and Oliver White