

JAVA & JVM CONQUER THE WORLD

TOP 10 REASONS WHY JAVA ROCKS

True story!

TABLE OF CONTENTS

INTRODUCTION	1
<i>to establishing JVM superiority over other virtual machines</i>	
High Performance JVM	4
Core API	6
Compiler	8
Bytecode	11
Memory Model	15
Open Source	17
Intelligent IDEs	20
Profiling Tools	22
Backward Compatibility	24
Maturity and Innovation	26
CONCLUSION <i>and goodbye comic</i>	28

*Reload code changes
instantly*



THIS REPORT IS SPONSORED BY:

JRebel

INTRODUCTION TO ESTABLISHING JVM SUPERIORITY OVER OTHER VIRTUAL MACHINES

“ *Most people talk about Java the Language, and this may sound odd coming from me, but I could hardly care less. At the core of the Java ecosystem is the JVM.* ”

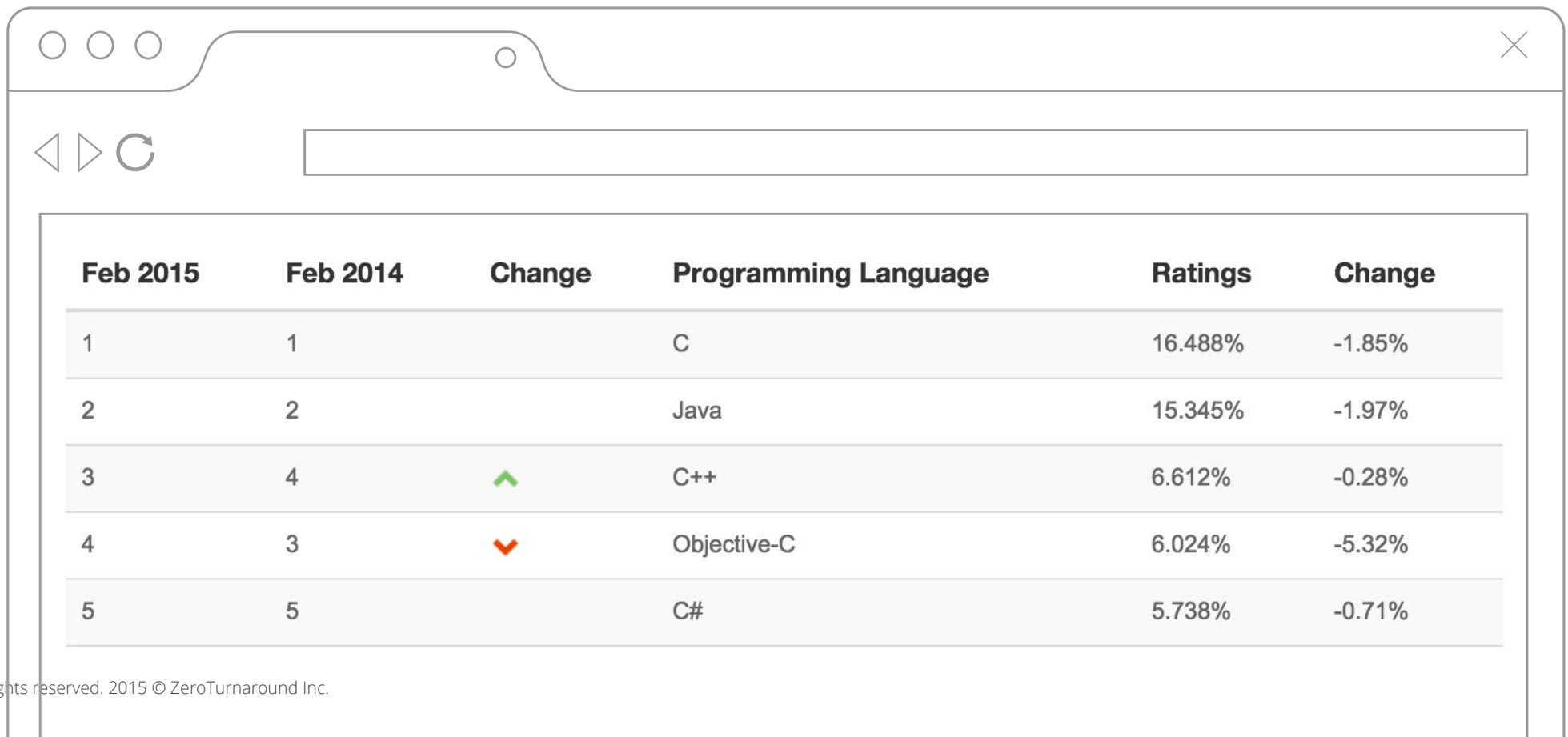
JAMES GOSLING

Creator of the Java Programming Language
(2011, *TheServerSide*)

Twenty years of Java

For many programmers, Java holds a special place in their heart. It may not be the most hip language today, but it has stood the test of time as the enterprise programming platform of choice for two decades. There are many things that Java developers take for granted. This special report is dedicated to paying tribute to the top 10 reasons why Java is still considered the best programming language after 20 years.

TIOBE Programming
Community Index
↓



A screenshot of a web browser window displaying the TIOBE Programming Community Index. The browser has a single tab and a search bar. The table below shows the top 5 programming languages by rating in February 2015 and February 2014, along with their percentage change.

	Feb 2015	Feb 2014	Change	Programming Language	Ratings	Change
1	1	1		C	16.488%	-1.85%
2	2	2		Java	15.345%	-1.97%
3	4	4	▲	C++	6.612%	-0.28%
4	3	3	▼	Objective-C	6.024%	-5.32%
5	5	5		C#	5.738%	-0.71%

2015 is another glorious year for Java as it casually sits at the top of many programming language charts. Take a look at the TIOBE Index, where Java is situated comfortably at the second position — despite all the “Java is dead” voices. Redmonk programming languages rating, IEEE Spectrum Top 10 Programming languages list and various other ratings put Java on the top of their charts. We have an enormous and vibrant community, an established open source process with many initiatives to evolve the language and the JVM platform even further; lots of things are in favor of picking Java as the platform of choice for your current and next projects.

This year also marks the 20th anniversary of Java — a wonderful fact makes some of us feel older than we actually are. Indeed, the first public beta release of our favorite, statically typed JVM language and the virtual machine itself was in 1995. A lot has changed since then. Luckily applets are now an almost forgotten technology, the JVM has improved considerably, it is now open and enjoys multiple implementations of the specification.

This report will explore the Top 10 reasons why we at Rebellabs believe that Java has become and will continue to be the leading platform for software projects. The Top 10 biggest things that we appreciate in Java and which we will cover in this report include:

High Performance JVM

Core API

Compiler

Bytecode

Memory Model

Open Source

Intelligent IDEs

Profiling Tools

Backward Compatibility

Maturity with Innovation

While you may or may not agree on our choice of the Top 10 items, we hope that this report increases your appreciation for Java. We would like to look beyond the surface and the buzzwords and analyze what actually makes Java the platform of choice for so many. We will also investigate which design trade-offs resulted in this popularity and what features could we appreciate more instead of taking them for granted.

HIGH-PERFORMANCE JVM



Most people will tell you that the differentiator for Java is its cross-platform nature. Java offers a write-once-run-anywhere approach which is provided by the virtual machine. What makes the JVM unique is its extensive breadth and depth of capabilities compared to any other application VM out there.

Why does this matter to us?

When writing software for different physical machines, you have to make sure that you properly test and compile for all target architectures. Up to the point where it becomes annoyingly tedious. Developers have to be versed in different versions of the major operating systems and ensure that these different development environments can actually co-exist.

Many languages without such a powerful JVM tend to off-load critical path functionality to native extensions (Ruby and Python, we're looking at you), which further increases the pain in to the struggle for portability. The JVM is so fast that you get the best of both worlds — as long as you don't require real-time operations, which is the case for most applications out there.

What does the future hold for the JVM?

With [Internet of Things](#) being all the rage, you will want your application to run on most devices and new devices will continue to appear at an even faster pace. So, a high performance virtual machine is becoming more relevant than ever.

With Java 8, [Profiles](#) allows you to select exactly which features you need to keep the footprint down, while continuing to use the same knowledge and tools that are available for desktop or enterprise development. The [Java Embedded Suite](#) even provides an optimized application server, service layer and database, making it extremely easy to create a RESTful web application that communicates with other devices.

Regardless of what your interests or needs may be, you can rest assured that the JVM will be able to perform at the level required for your applications – from high-throughput web apps to setting up a Raspberry Pi with Java SE Embedded for powering that beer chauffer or a kitty-box cleaning robot.



Try [Java Embedded SDK](#), if you haven't already. Different profiles can make JVM startup faster and reduce the memory footprint. If you did, take a step further and check out the embedded SDK.



[Apache Spark](#) is a data processing engine for both in-memory and disk using operations. Spark can even run on existing Hadoop installations.



[Glassfish Application Server](#) is the reference implementation of Java EE specification. It can easily run on your home desktop, high performance machines or on a tiny, throwaway [Raspberry Pi board](#). If you have never tried coding for a Raspberry Pi, you should. It is fascinating to see how easily JVM handles scarce resources of a weak device.

CORE API



Interestingly, the most awesome aspect of the Core API is that it has always been there, that it still is there and that it's still backward compatible. It's complete enough to allow you to write most applications without requiring anything else, should you wish to do so. However, the most common complaint you hear about Java's Core API is that it can be verbose. This is precisely why people use additional libraries or [JVM languages](#) that provide concise language-level access to many of the Core API features.

Most of these actually rely on the Java Core API underneath, wrapping around it and allowing library and language authors to focus on the neat features they want to provide. Java's decision to not only create a language and a virtual machine, but to also standardize the core libraries from day one, really paid off. Of course, other languages later followed with a similar approach, but few remained as stable as Java.

Stability allows you to move forward

Python is another language that has a nice core API with an extensive set of standard library features. Sadly though, it has historically been far from stable. Both the internal language type system and the standard libraries went through drastically incompatible changes over the years. This effectively ties existing projects to older versions, as porting a large application to a newer version can be a major undertaking.

Stand on the shoulders of ... everyone

When library and framework authors can communicate using the same programming language and the same API primitives, every project becomes a useful instrument in the toolbox that incrementally increases the overall possibilities of the platform. This is definitely one of the reasons why Java has such a **vibrant** open source community.

Here are some ways to enjoy the power of Java core API

Guava is a powerful collection of libraries and utilities for collections, caching, primitives support, concurrency libraries, common annotations, string processing, and I/O. Created and managed by Google, Guava is mature, powerful and very educational to poke around in the source code.

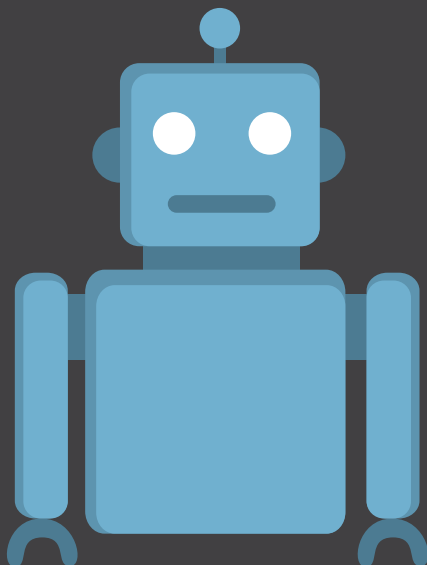


Any number of JVM languages: **Scala**, **Clojure**, **Groovy**, **Kotlin**, **Ceylon**, **Fantom**. Play around with these, see how they build on top of existing API and expand it.

Hibernate - a well-known ORM library. For better or worse, it makes use of the default JDBC driver API and provides you with lots of automagical features that you either love or hate.

*Hibernate attracts strong opinions,
but at its heart is a good ORM*

COMPILER



You'll need to use the Java compiler — whether writing a simple hello world example or a major enterprise app — to turn your source code into bytecode and then into the executable.

The JVM's JITting like a champ

You might think that the JIT (Just in Time) is just a performance improvement, made to leave interpreters in the dust and rival native performance speeds. But it goes much further than that. When compiling straight to native (like with C++) you have to deal with static optimization that has to be decided upon at compilation time.

It is quite a common practice to just step through the optimization levels of the produced binaries with manual testing, hoping to find one that works. Without a deep understanding of the compiler internals, it's often impossible to predict how the profilers will change the execution of your code, as well as which optimization levels will work or which won't.

Focus on your code, not on the compiler architecture

Since the Java compiler only has to transform source code into bytecode, it is very simple to operate the `javac` command itself. Usually, all you have to be concerned with is providing the correct classpath information, deciding on the VM version compatibility, and telling it where you want the class files to be placed. You need a total of three different compiler options: `-classpath`, `-target` and `-d`.

No static or dynamic linking, just run-time linking

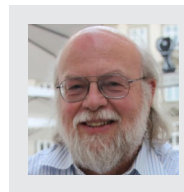
When you produce native binaries, you have the option to decide how you're going to link your own libraries or modules. Static linking packages everything into a single executable, but prevents you from independently updating the libraries. It can also generate very large binaries. This has the advantage of easier distribution and avoids the problems associated with dynamic linking. In practice though, it's impractical for larger products to ship a single, statically-linked executable. Sooner or later you will have to deal with dynamic linking — even if it's just to split up your build or to deal with 3rd party libraries.

Java defers linking to run-time. Everything is dynamic and since the symbols are exported through bytecode while being isolated with packages, you rarely have to be concerned with the linking phase. If needed, you can still dynamically load classes or methods, but that's rare. The bytecode symbol representation is stable and as the versions of classes evolve, your older code will continue to run without problems (unless the library author intentionally changes the API).



One of the Java features that people hardly think about is the way linking works. Linking between modules, so if you look at the Java class file format, there's no fixed offsets to things, it's always symbolic references and strong rules that decide how dynamic linking works across version changes.

It makes it really-really easy for libraries to change versions without the code that depends on them having to change. And that's been just huge in building this API ecosystem. So if you look in the C world, just the smallest change in a C struct and everything that uses it has to recompile.



JAMES GOSLING

VirtualJUG session (January 2015)



How do you take advantage of these compiling capabilities using tools?

JITWatch is a log analyzer and visualizer for the HotSpot JIT compiler. You run JVM with a couple of runtime options and obtain a JIT log that says which methods were compiled, when and into what. JITWatch shows all that information in a useful UI and makes investigating it much easier.



Javap, the Java Class Disassembler that is shipped with the JDK itself. It takes .class files and lists useful information about Java symbols.

Maven, the build tool. Whenever we talk about compiling the project or downloading JARs, we have Maven in mind. Looking for declarative way of turning your Java code into .class files, packaging and resolving dependencies? Use Maven.



Maven

BYTECODE

You know that bytecode allows the Java compiler to express instructions in a format that is directly understood by the Java Virtual Machine.

Some users compare bytecode to assembly, but we have found that it's nowhere near as complicated. Thankfully you get all the good bits of the [JVM](#) and its verifier, drastically reducing the potentially harmful instructions that you can write.

Even if you don't use third party bytecode manipulation tools, for example [ASM](#), the Java platform has actually everything included — allowing you to inspect the bytecode of any class. Let's take a quick look, for instance at this very complex piece of Java code:

A stylized illustration of a code editor window. It has a title bar with three window control buttons (minimize, maximize, close) and a tab. Below the title bar is a toolbar with icons for undo, redo, and search. The main area of the window contains Java code for a HelloWorld class. The code is as follows:

```
public class HelloWorld {  
    public HelloWorld() {  
    }  
  
    public static void main(String[] args) {  
        System.out.println("Hello World!");  
    }  
}
```

It will look like this in bytecode after using the JDK's javap command:

```
javap -p -c HelloWorld
```



```
public class HelloWorld {
    public HelloWorld();
        Code:
            0: aload_0
            1: invokespecial #1    // Method java/lang/Object.<init>():()V
            4: return

    public static void main(java.lang.String[]);
        Code:
            0: getstatic      #2    // Field java/lang/System.out:Ljava/io/PrintStream;
            3: ldc           #3    // String Hello World!
            5: invokevirtual #4    // Method java/io/PrintStream.println:(Ljava/lang/String;)V
            8: return
}
```

Looks quite familiar, doesn't it?

Bytecode manipulation changed the Java platform

Since it's so close to Java, you might wonder why you'd want to get interested in bytecode manipulation and generation. Well, you've probably been using it all over the place. Since JVM can modify bytecode and use new bytecode while it is running, this generated a whole universe of languages and tools that far surpassed the initial intent of the Java language.

Some examples include:

- **FindBugs** inspects bytecode for static code analysis.
- Languages like **Groovy**, **Scala**, **Clojure** generate bytecode from different source code.
- ORM tools, like **Hibernate**, can instrument your setters and getters to automatically perform the required database operations.
- Dependency injection frameworks like **Spring** use it to seamlessly weave your application life cycle together.
- Language extensions like **AspectJ** can augment the capabilities of Java by modifying the classes generated by the Java compiler.
- ZeroTurnaround's **JRebel** instruments your classes to instantly reload changes — without having you to go through the lengthy compilation and redeploy cycles.

What libraries can get you closer to the bytecode level?

ASM is the Java bytecode manipulation library. Its focus on performance made it the choice for bytecode generation for almost any tool that does bytecode transformation, from **Spring** to **JRebel**, to **Clojure**, to **Play Framework**.



ByteBuddy is a code generation library. It is written on top of ASM and it provides a type-safe DSL for bytecode manipulation.



javassist is another library that aims at making bytecode manipulation friendlier than specifying code tree visitors. Javassist can take Java code in a String and compile it on the fly.

JD is a Java code decompiler. When you have access to the compiled class files, which you do have if you use the library, you can run it through JD and happily obtain readable Java source. This can then be modified and repackaged at your discretion. It is mostly used to understand what libraries do without directly accessing the source code.

JRebel

RELOAD CODE CHANGES
INSTANTLY

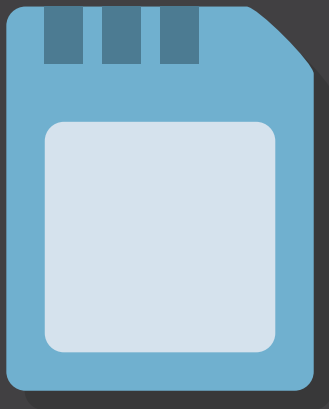
Get a free
t-shirt!



TRY JREBEL AND GET AN AWESOME T-SHIRT

TRY IT NOW!

JAVA MEMORY MODEL



Parallelism is the current way toward increased computing power. Concurrency has been entrenched in applications for many years, even if this was just in the form of an event dispatch thread. To allow for a coherent memory view, processors include memory models where barriers can be used to ensure that writes are visible to other threads or processes. The Java Virtual Machine goes one step further by providing its own additional hardware-independent memory model. As opposed to other languages, where concurrency primitives are handled by external libraries (such as Perl, Python and JavaScript), Java provides this at its core and it can be used by everything that needs coordination and data consistency. The Java Memory Model has allowed the language, the compiler and the core API to be designed together — to provide a stable foundation for concurrent operations and a shared state.

In 2011, [C++11](#) and C11 added their first take on a memory model, inspired by the work done in Java beforehand.

Building blocks for other parallelism solutions

Getting concurrency right is difficult. Different problems can nowadays benefit from different solutions. Instead of using threads, an [Actor-based](#) model might be more appropriate. Maybe even [Software Transactional Memory](#), or maybe [Fork/Join](#), or maybe [different lock](#) implementations.

The ability to pick whatever solution is most appropriate for you without having to make sure that your compiler, hardware or libraries are compatible is very convenient. Since the implementations of these different solutions all rely on the Java Memory Model, all the basics are covered. It is much easier to grab the technology you need for a particular concurrency problem.

How to get closer to understanding concurrency and parallelism?

[jctstress](#) (Java Concurrency Stress) is an experimental harness and a suite of tests to aid the research in the correctness of concurrency support in the JVM, class libraries, and hardware. This allows you to verify that you understand how those volatiles work or just try your hand at creating a test case for the fuzzy concurrency tests.

[Apache Pig](#) is a platform for parallel computations used for example by Apache Hadoop. Pig consists of a high level language and a runtime to evaluate programs specified in it. While we do not think that you should turn to Pig whenever you find yourself in need of a parallel computation, we do believe that playing around and looking through [source code](#) is a useful exercise.



OPEN SOURCE



There's a JAR for that

We immediately hear you say that open source is not unique to Java, and you're right, it isn't! What is unique though is that the Java platform ranges all the way from mobile to enterprise and that many of the world's critical systems rely on it. Linux is possibly the only other open source technology that has achieved similar ubiquitousness, with Java being the only software development platform with that status.

Thanks to open source popularity and platform independence, we've always been able to [find a viable](#) library for general purpose functionality with often even the burden of having to choose between different fundamental approaches to a particular problem (web frameworks or template engines anyone?). You only need a glance at the [language-based categorization](#) of projects in the Apache Foundation to see how far-reaching this is.

Professional Open Source

Everything about Java is **open**: from the language to the standards, to the core libraries, including the virtual machine and the development tools. Since the open source mindset is so pervasive in the Java world — present at its core and tested at enterprise-grade level — it has become logical for commercial services to be built around thriving projects. The prevalence of professional open source companies removes the risk of using an open source solution that relies on individuals. As a solution provider, you get the safety net of continuity, combined with top-to-bottom consistency and the freedom of open source.

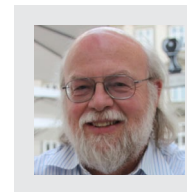
This is a very powerful combination to beat for a development platform, one that is mostly only seen for specific products or operational tools!

*Easily mistaken for
Ryan Gosling :)*

“

Q: What helped Java succeed?

A: The combination of strength and diversity, I mean you find people doing just the coolest, strangest, weirdest things in corners. You find a mindboggling array of libraries that we have in the Java world and the projects behind them.



JAMES GOSLING

VirtualJUG session (January 2015)

”

Community spirit

Obviously, open source goes hand-in-hand with community and Java is here again as vibrant as it gets. More than a hundred [conferences](#) spread over the world and close to 400 registered [Java User Groups](#) allow any developer to learn from his peers. Obviously this is just the tip of the iceberg with many online communities, for example [Virtual JUG](#), that are bursting at their seams.

This community spirit has become standard for us Java developers, but we expect that it's not as evident as one might think.

How do you find a JAR you need?

[Java Megajar](#) is a single JAR file that contains all the best JARs you might end up using anyway. It's everything you need when writing Java in one easy download. Just look at its [pom.xml](#) and be amazed.

[Bintray](#) is a software distribution center. You can find almost any open source Java project there -- pretty much like Maven Central -- just user friendlier, secure and more powerful. All those libraries are available to you and your application and are just waiting there to be useful.





INTELLIGENT IDES

It took a while for IDEs to become as good as they are today. We still remember when [Omnicores CodeGuide](#) was released some time around 1998. It was the most amazing thing we had ever seen. When IBM released Eclipse for free as open source, the whole IDE landscape changed. Commercial IDE vendors had a very difficult time to stay afloat and JetBrains seems to be the only one that's still there today.

A Puff of Magic

Fast forward to 2014. We now have three fully integrated, great environments: NetBeans, Eclipse and IntelliJ IDEA. Projects have become a lot more heterogeneous and understanding a single language really doesn't cut it anymore.

Thanks to low level architectures, built for efficient abstract syntax tree and code dependency handling, these complex environments are handled rather well in all available IDEs. Standard project management tools like Maven are fully integrated and are replacing proprietary formats, allowing developers to easily collaborate on the same project with different tools.

Today, you get free, open source access to all core IDE features, while plugins enhance the feature set to the next level. It's dazzling that such advanced creation tools are community-driven and freely available to everyone that wants to build software.

Whether you fancy the amazing HTML5, dynamic language and Maven support of NetBeans, the versatility and scope of the Eclipse platform; smart code completion, intentions, database viewer or the dark theme in IntelliJ... Each one is just as capable as the next and having the choice is really a problem of luxury.

A Splurge of Knowledge

At ZeroTurnaround, we're very interested in how tools make you more productive. We have already published a [detailed report about Eclipse](#) in this light. We've heard rumors about colleagues diving deep into IntelliJ IDEA with the aim of publishing their findings. If you're a fan of NetBeans and would like to share its uniqueness with your fellow developers, do let us know. RebelLabs is always looking for new writers and contributors.

Which IDEs should you check out?

MPS by JetBrains is a DSL development environment. Try designing your own DSL and you will see how an IDE can support it by providing much needed help to the user of the language.

Then of course there are multiple industry-wide used IDEs. We won't describe these in detail in an attempt to avoid holy wars. We know that you're using one of these anyway. Just check out one of the others as well.

- Eclipse
- IntelliJ IDEA
- NetBeans

Developers want to try
IDEA the most



PROFILING TOOLS



Measure before you jump

The worst thing you can do when trying to fix a performance problem is to make assumptions. You're guaranteed to either make things worse or to waste precious time while your application is sliding into the pits of doom.

The question is: how do you measure? Well, with the Java platform this is easy! You already have capable profiling tools in the JDK itself. If you haven't tried it yet, make sure to give [VisualVM](#) a go. If you need to go further than that, there's a plethora of free and affordable tools available that take the measurement and analysis to the next level.

Measure meaningful data

The Java Virtual Machine has been created with profiling and monitoring in mind. This is especially important since you have to do absolutely nothing to your code to get useful data out of profilers. What's even more amazing is that you can just as easily profile remote applications as well as local ones. You can even profile running production code without having to take down the server and without causing any measurable impact on performance!



Measure more effectively

Performance tuning is one of the most overlooked skill for developers. Our friend [Kirk Pepperdine](#) has dedicated much of his career to specialize in this area. During his workshops he has found that testers have an easier time finding the root cause than programmers, because instead of reaching straight for the code, they measure, observe and analyze the data.

So pull your head out of the code and embrace the monitoring and profiling capabilities of the JVM. It often beats any other solution to quickly pinpoint the cause of a problem. If you want to learn more, you can find all Java performance tuning information on javaperformancetuning.com.

What to look for in the sea of profilers?

JProfiler is the leading JVM profiler out there. Its intuitive UI and maturity make it a solid profiler tool, which can definitely be your first choice of the profiler if you need one.

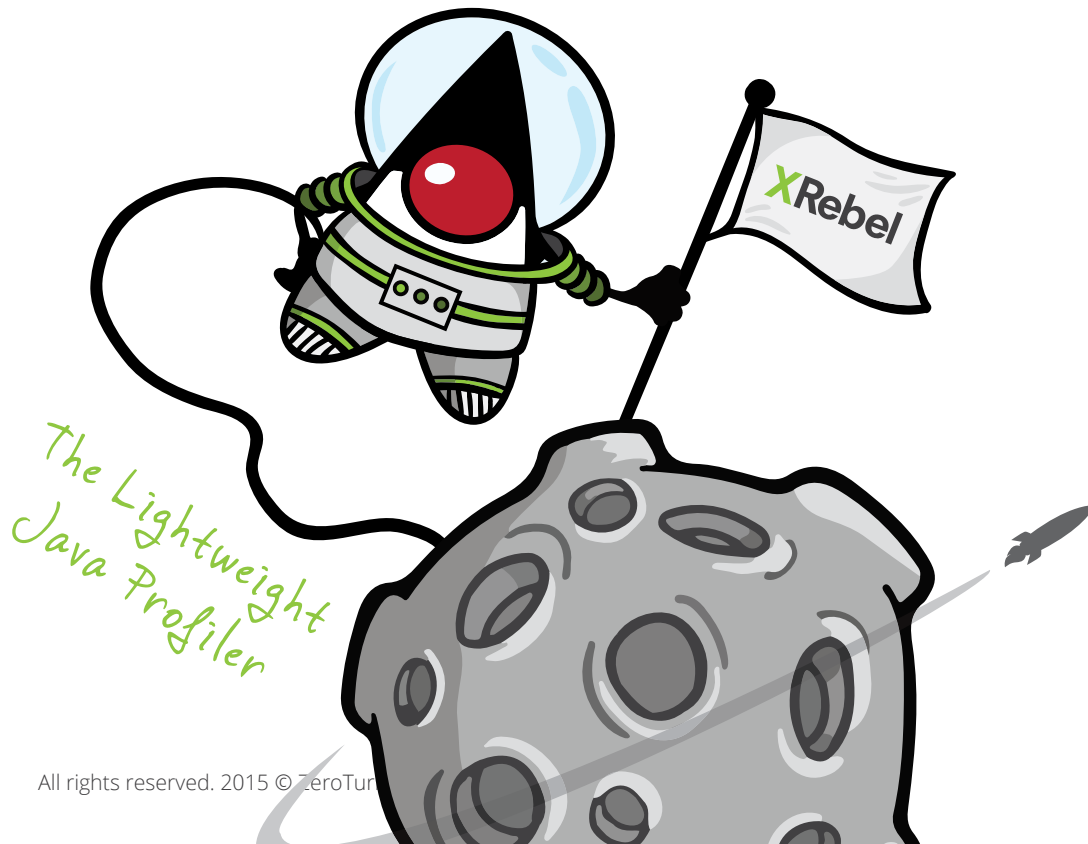
XRebel is a lightweight Java application profiler. It shows you production grade performance issues during application development. Created by ZeroTurnaround, i.e. us.

JMC or Java Mission Control is an advanced set of tools that enables efficient and detailed analysis of extensive data collected during runtime profiling.

Honest profiler is a Java profiler that tries to avoid common profiling issues, like stopping threads at safe point or influencing the gathered data by its own performance.

Plumbr

Plumbr is a Java performance monitoring solution that can also point out root causes of issues.





BACKWARD COMPATIBILITY

Still running Java applications that were compiled more than 10 years ago? These might not even be running on the same infrastructure. Have you progressively migrated them across VPS hosts and through different dedicated servers, each time running the latest versions of the JVM? All it takes is copying the old JAR and WAR files and starting the application up again. And there you go, running in all their glory as if nothing happened.

The Java language has remained very stable, even after introducing new constructs like try-with-resource, generics, auto-boxing, enhanced for loops, string switch statements, typesafe enums, annotations, static imports, varargs etc.

Thanks to the backward compatibility of the language as well as the JVM, the Java community has been able to stand on each other's shoulders for more than 20 years! Undoubtedly this has been a major contributor to why Java libraries and frameworks cover such a large spectrum of industries and interests.

While it's very reassuring that newer versions of the JVM are capable of running your older binaries, it may be even more exciting to know that your old applications will automatically benefit from improvements that are made to newer versions of the JVM.

As real machines get more powerful, so does the Java Virtual Machine. You stand to reap the benefits of both — without having to worry about educating yourself on low level semantics.

It's super convenient to know that you will not be tied down to these environments when developing in Java. Again, this goes further than merely being able to run the code anywhere, it actually liberates you and makes it possible to seamlessly work together with other developers with different preferences.

It's a given that in the future, the whole computer landscape will be radically different. We think it's pretty damn awesome that not only will the software someone wrote decades ago continue to work without problems, but also what we will write in the coming decades will similarly face no barriers. Everything will continue to coexist peacefully!

How to check your own code's backward compatibility

Java API Compliance Checker is a tool for checking backward binary and source level compatibility of a Java library API. The tool checks class declarations of old and new versions and analyzes changes that may break compatibility: removed methods, removed class fields or added abstract methods.

JAF or JavaBeans Activation Framework. An ancient beast, the latest version of which was released ages ago. It still works on modern JRE.



Eclipse, the famous IDE. The latest version Luna requires JDK 7, but will probably also work on JVMs released 10 years from now.



MATURITY INNOVATION WITH

Maturity never sounds glamorous or exciting. Why would it, when it instinctively makes you think of outdated, slow and wrinkled? However, in our book it means that we can rely on it without surprises. It means that we know exactly what we can expect, and most importantly, it means that we can use it to build a product as a team without the risk of stepping on each others' feet.

The availability of **intelligent IDEs**, a backward **compatible** platform, a massive **core API**, a vibrant **open source** community, and a language that was designed for readability and encapsulation makes Java perfect for teamwork.

It allows code to survive individuals and to be passed on to new programmers or colleagues with fewer surprises. It allows individuals to contribute without making invisible far-reaching changes. It allows people to build complex solutions that require a long time to be delivered and will probably be supported even longer.

The last thing that you would want when building a house is for the ground to be unstable or for the foundations to collapse. Similarly for software, it's important to build with proven technology if you're creating a product that is intended to last. Java has proven this maturity as it is used for the largest enterprise and mission critical projects on the planet.

Constantly Innovating

Even with this maturity, Java does not rest on its laurels. For 20 years now, it has been improving and adapting, all the while keeping stability and backward compatibility in mind. This obviously means that it improves slowly and very carefully. The point however is that it has done so relentlessly.



It doesn't matter how slowly you go -
so long as you do not stop.

– CONFUCIUS



*Are these the droids you're
looking for?*



Where to look for innovation?

Java 9 modules. Project Jigsaw is getting traction. The first step only includes JDK reorganisation, but we can always hope that the next steps won't take as much time as the first one.

Money API. Given that Data & Time API in Java 8 had a warm welcome from developers, this trend continues. There are several API updates planned for JDK 9, but money and currency is probably the most prominent.

No JSON in JDK 1.9. This is a sign of maturity and common sense. Initially including JSON library into the platform was agreed, but now it will be kept out of 1.9 for greater good.

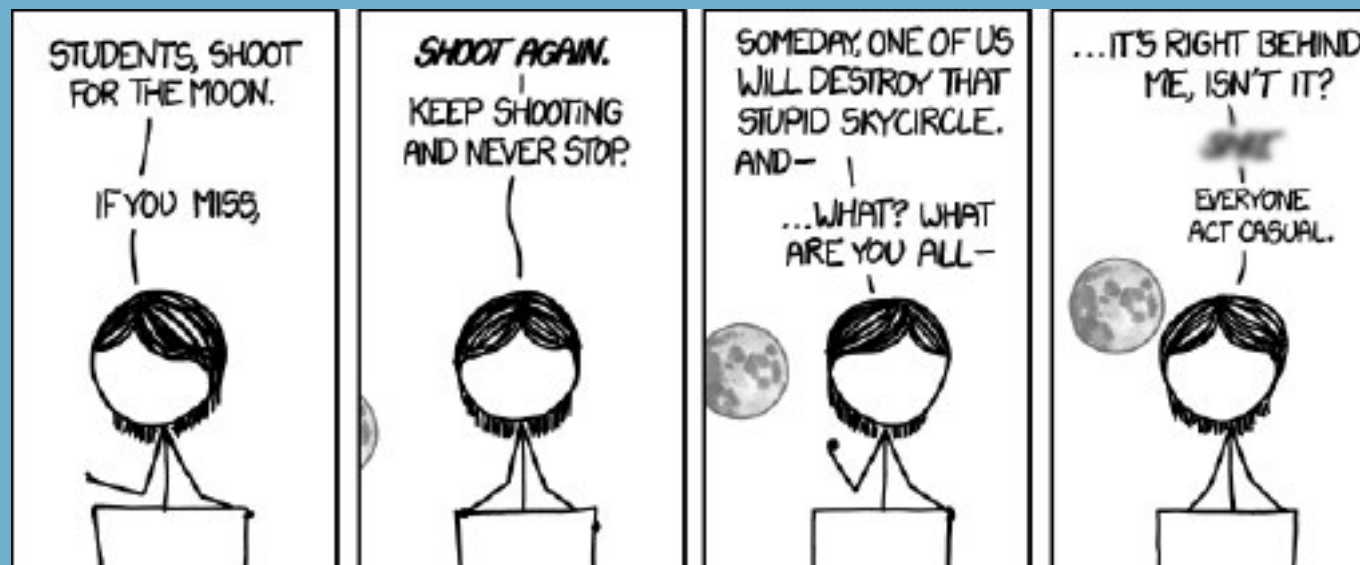
The effort to provide **HTML5 javadoc** shows that the platform is not afraid to push things forward when this doesn't break millions of existing applications. With Java 9 we will be able to comment and illustrate the code using modern tools.

CONCLUSION AND GOODBYE COMIC

One of the major aspects that make Java and the JVM so great and powerful is that there is no one thing by itself that makes it great. Yes, each of the individual aspects mentioned in this report really add to the effectiveness of the technology, but Java and the JVM do not rely on one of them. It is the mix of IDE support, backward compatibility, CoreAPI, memory model, maturity etc. that in combination make the JVM rock. And it speaks volumes as to why Java as a technology will be celebrating its 20 years in 2015.

Raises glass Congratulations Java, here's to another 0b10100!

Here's an XKCD comic that illustrates the importance of setting high goals and actually reaching them:





↙ Contact us

Twitter: @RebelLabs

Web: <http://zeroturnaround.com/rebellabs>

Email: labs@zeroturnaround.com

Estonia

Ülikooli 2, 4th floor
Tartu, Estonia, 51003
Phone: +372 653 6099

USA

399 Boylston Street,
Suite 300, Boston,
MA, USA, 02116
Phone: 1 (857) 277-1199

Czech Republic

Jankovcova 1037/49
Building C, 5th floor,
170 00 Prague 7, Czech Republic
Phone: +420 227 020 130

Written by:

Geert Bevin (@gbevin), Simon Maple (@sjmaple),
Oleg Shelajev (@shelajev), Mart Redi (@martredi)

Designed by: Ladislava Bohacova (@ladislava)