payara®

No Nonsense Guide to JVM Implementations: OpenJDK, OpenJ9, GraalVM

The Payara® Platform - Production-Ready, Cloud Native and Aggressively Compatible.

User Guide

Contents

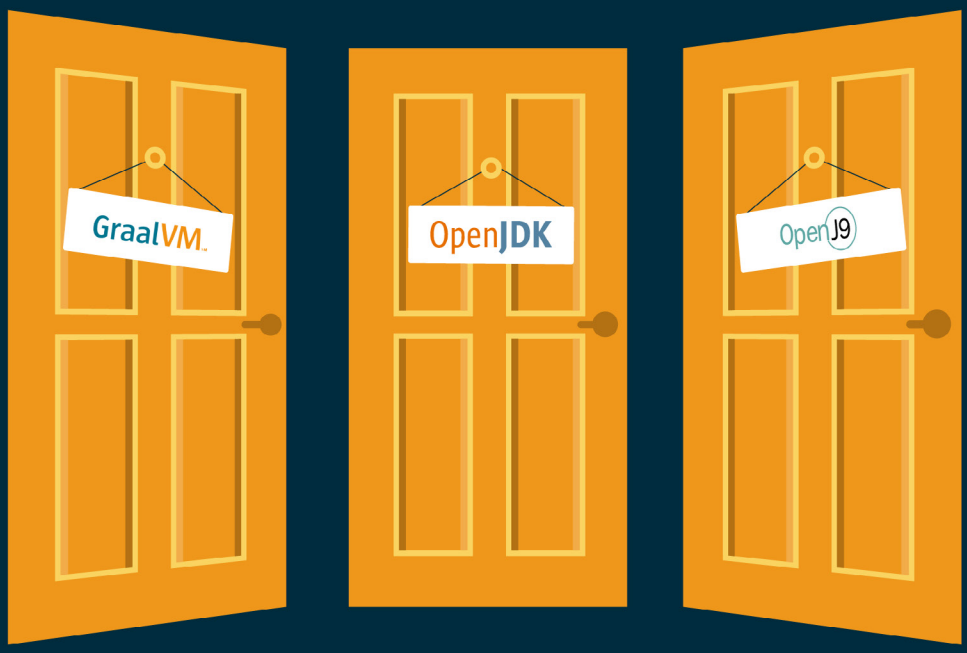
Introduction

Ten years ago, the OpenJDK JVM codebase was open-sourced. Now you have a choice of more than ten different JVM implementations.

But which should you choose? And what about JVM implementations that do not have their origin as the OpenJDK codebase?

In this guide, we examine the history, similarities, and differences between OpenJDK, OpenJ9 and GraalVM. This includes looking at those many different JVM implementations that come from OpenJDK.

You should study the strengths of each of these three groups of implementations to make the best choice for your environment.



OpenJDK History

The ability to choose between so many JVM implementations can be traced back to 2011, which was the start of the process to make the codebase of the JVM open source. This was when Oracle launched Java SE 7, as a full reference implementation, OpenJDK 7.

Legal processes took some time, but by the OpenJDK 10 release in March 2018, the Oracle OpenJDK codebase was simply the upstream version, with many JVM releases based on it.

The OpenJDK Community leads the open source development of Java within Oracle, whilst you can still get commercial Java licenses, enterprise features and suppport from many places.

The Java Community Process (JCP) defines the future direction of the platform's evolution. The JCP allows interested parties to develop standard technical specifications for the Java technology. Anyone can become a JCP Member by filling out a form available at the JCP website. JCP membership for organizations and commercial entities requires annual fees – but is free for individuals. Change is brought about through JSRs, Java Specification Requests (JSRs): formal documents that describe proposed specifications and are voted on by all JCP members, to decide if they are or are not implemented.

Besides the actual codebase from which the implementations are created, there is also a large test suite. The Technology Compatibility Kit (TCK) is a suite of tests that at least nominally checks a particular alleged implementation of a JSR or the entire JVM for compliance. It allows any interested party to verify if the JVM implementation they have created is correct. It also ensures that if you download a JVM implementation, it will work as expected, and that there are no major differences between the different implementations.

It is one of the three parts that each JSR provides; the document that describes the functionality; a reference implementation that proves that the proposed functionality in the JSR can be implemented; and the TCK that tests if it works as described.

OpenJDK

As mentioned, the OpenJDK release base was the first open source version of the JVM.

Today, there are many implementations that are based on the OpenJDK code base includingAzul Zulu, Eclipse Adoptium, IBM Semeru, Oracle OpenJDK, RedHat OpenJDK, Amazon Coretto, Alibaba Dragonwell, Microsoft OpenJDK and Jetbrains Runtime.

What unites all the above JVM distributions is that they are built based on the OpenJDK codebase. As a result, they share the same functionality and of course, have all the required functionality that you expect for your Java runtime.

The reason that most vendors have their own version of the Java runtime is that they have made some additions. The above list of JVMs are all free to use, even in production, but most vendors also have commercial versions of the JVM. If some of their customers have specific requests, or the vendor wants to offer some additional functionality on top of that available within the OpenJDK code base, they can add it to the codebase.

As they know how to build a runtime out of the source code for their paid customers, they also produce a free version. Even in the free version of their product, there are additions on top of the codebase of OpenJDK.

Let us briefly go over some of the JVM implementations and their specific features.

Eclipse Adoptium / AdoptOpenJDK

The AdoptOpenJDK releases were just the binary builds of the OpenJDK codebase. They were not TCK tested so there was always the possibility that some functionality would not work as expected. The alternative at that time was the Oracle build of the OpenJDK, but as it was released under the Oracle Binary Code License Agreement, use could lead to discussions around if you were able to use it for free or not.

AdoptOpenJDK is now tested against the TCK. Eclipse Foundation took on stewardship of AdoptOpenJDK. It, along with vendors in the Adoptium Working Group, provides infrastructure where the OpenJDK source code is built and the TCK tests are performed. These builds are released under the name Eclipse Adoptium and are a free to use and a tested JVM implementation.

Oracle have also adjusted their licenses on the Oracle build of the OpenJDK to make it less ambitious. You can now use that build for free, but note that the Oracle JDK is the commercial version and you need some agreement in place before you should use it.

Azul Zulu OpenJDK Builds and Platform Core

For JDK 8, Azul added a few tools and functionalities that were not available at that time. Azul Zulu provides the integration of Flight Recorder and the tool Mission Control. It also made an implementation of TLS 1.3 available on JDK 8 before it was added officially into the OpenJDK codebase.

Azul is also working hard on performance and startup. Their supported JVM, Azul Platform Prime (previously Zing), can handle large heap sizes very efficiently. Regarding the startup of a classic JVM, they have solutions like ReadyNow! and the recently announced Cloud Native Compiler that improves the startup time greatly, without the need to rewrite your application to allow native compilation.

They also have the broadest support for operating systems and JDK versions.It bundles the OpenJFX binaries, the open source version of the JavaFX client-side platform.

Liberica

The Liberica implementation focuses on some client-side aspects of the Java Platform. It also bundles OpenJFX binaries. It maintains the OpenWebStart framework since that is no longer part of the JDK offering.

Dragonwell

Dragonwell is the name of the JVM build created by Alibaba, one of the large Cloud providers in the world. They have added some functionality to improve the performance, stability, and functionality for their Cloud platform. One of the remarkable features is 'WISP', an implementation of the Virtual Threads proposal that is also the goal of Project Loom within the OpenJDK. When activated, the JVM switches to an implementation that uses Virtual Threads managed by a Thread Manager in Dragonwell,and does not only rely on the Threading management of the Operating System. This way you can achieve some great performance improvements without the need to adapt your application.

Another surprising element is that they don't run the TCK tests on their product. Although their implementation is used on 100,000+ installations without any issues, this means you can never be sure of functionality.

Amazon Coretto

Another Cloud provider that produces its own JDK implication is Amazon. Amazon Coretto has a few performance and security fixes on top of the standard OpenJDK version, mainly for JDK 8.

You can use it within their cloud environment but also outside it, as it is completely free to use.

Microsoft OpenJDK

Microsoft has a build available of the OpenJDK that can be used on their Cloud Platform Azure, but also outside of it - for running Minecraft, for example! They also support the Eclipse Adoptium initiative. This is a remarkable change since in the past Oracle and Microsoft had a dispute about licenses that ended up in court.

Other JVM implementations

As you can see in the previous section, there are many JVM implementations available that build upon the OpenJDK codebase. But there are also other implementations that do not have their roots in the OpenJDK codebase.

OpenJ9

The OpenJ9 JVM implementation started out of the K8 Virtual Machine for the Smalltalk programming language in the late 1990s. The VM code was adapted so it could also read the Java byte code and execute it, largely keeping the memory management functionality of Smalltalk. It was developed by IBM and later on open sourced and made available through the Eclipse Foundation.

OpenJ9 has some impressive statistics for performance, memory usage, and startup times. Many tests provide evidence that the startup time is up to 50% faster. The ramp-up time to achieve maximum throughput is much smaller and also memory usage is 10% or more lower compared to OpenJDK-based solutions.

The OpenJ9 implementation also supported creating the Startup Archive for the Class Data Sharing option from a running instance. Before the Dynamic Archive functionality was introduced into the OpenJDK, the creation of the Archive file was a two-step process. First, you needed to startup your process to get a good idea of the Java classes that were used. In the second step, the Archive file was created based on this classes list and the classpath.

OpenJ9 implemented a much easier solution that allowed the creation of the Archive when the JVM was shut down, the same principle as Dynamic Archive. This allowed a much easier and faster creation of the Archive, to improve the start-up time of the JVM process next time.

Graal VM

The term GraalVM may cause confusion, as it can indicate several things. The earliest experimentations, which later on become the Graal JIT compiler, started in around 2005. The idea was to create an entire JVM within Java and no longer be dependent on C++. It proved a huge undertaking and at first, only the C1/C2 compilers (that turn the Java Byte code into native code as part of the optimizations carried out by the JVM) were written in Java.

This GraalVM Compiler was also added as part of the OpenJDK between JDK 10 and JDK 16. It was removed from the OpenJDK because it was not used much by end-users and imposed a too large maintenance overhead. But it is still part of the GraalVM JVM distribution as it plays a crucial role in the native compilation I will touch on in a moment.

The term GraalVM is nowadays used for the entire JDK distribution product that can run multiple languages on the JVM and can perform native compilation. The basic idea is that the JVM is capable of running applications in many computer language, including non-JVM-based languages, and integrating them. The JVM-based languages like Groovy, Scala, and Kotlin are compiled to JVM byte code so they can be run easily by the JVM. But GraalVM has an integration with an LLVM Runtime to integrate with C++ code amongst others. And through the Truffle project, it is possible to turn any code written in any language into a structure that can be understood and executed by the JVM. This makes it possible to execute a Python, R, and Ruby program on the JVM and share the same address space as your Java program, making it possible to have a seamless integration between all those languages. Also, JavaScript and Node.js are supported since the GraalVM contains a runtime for them.

With the GraalVM it becomes possible to write and execute a polyglot application. You can reuse some libraries in non-Java languages and combine them with other code snippets written in other languages. Then, you can run it all on the JVM.

But of course, you can also use it to run your Java-only programs, just as with the OpenJDK-based implementations or OpenJ9. And GraalVM contains another nice piece of engineering, the native compilation. Since the Graal JIT compiler is written in Java and creates native Operating System code, it can be used to compile Java applications to native code also. This is done with the Graal JIT compiler, so that the performance of the JVM is comparable with other JDK implementations that have the C1/C2 compilers written in C++ and natively compiled. But it can also be used to compile your application to native code. There is no JVM needed anymore at runtime as your Java application is now a native application for your OS. This means that it starts faster (no more the overhead of the JVM startup) and lowers memory consumption since the JVM itself doesn't need any memory.

There are however a few drawbacks, such as the slow compilation to native code and the lower peak performance of your application. Our CEO Steve Millidge wrote about them here. Since the ahead-of-time compilation cannot use the information around how your code is used within the application at runtime, it can make fewer optimisations. This means that after a short while, the JIT-compiled code in Java performs better than the natively compiled code since it can make more optimisations.

About Payara Platform

Payara Platform Enterprise is stable, supported software for enterprise designed for mission-critical production systems and containerized Jakarta EE (Java EE) and MicroProfile applications.

Payara Server

Payara Server is a cloud-native middleware application platform that supports reliable and secure deployments of Jakarta EE applications on-premise, in the cloud or in hybrid environments. A stable platform with monthly releases, bug fixes, and a 10-year software lifecycle, Payara Server is aggressively compatible with the ecosystem, cloud vendors, Docker, and Kubernetes. An official Jakarta EE compatible implementation, Payara Server is developed in collaboration with an industry-leading DevOps team and the global Payara community to ensure Payara Server is the best option for the production of Jakarta EE applications today and in the future. Find out more here.

Payara Micro

Payara Micro is the lightweight middleware platform of choice for containerized Jakarta EE application deployments. Less than 80Mb, Payara Micro requires no installation, configuration, or code rewrites - so you can build and deploy a working app within minutes. Compatible with Eclipse MicroProfile, Payara Micro is the microservices-ready version of Payara Server. It comes with a Java API to embed and launch from your own Java applications and offers the ability to run war files from the command line without any application server installation. Automatic and elastic clustering makes Payara Micro the platform of choice for running Jakarta EE applications in a modern virtualized infrastructure. Find out more here.

Payara Enterprise

If you are looking to run your deployment in production, we offer Payara Platform Enterprise Edition as stable, supported software for mission critical systems in production.

With Payara Enterprise Edition, you are able to:

- Get support directly from engineers
- Enjoy a 10-year software lifecycle
- Monthly releases, bug fixes and patches
- Security alerts and fixes
- Stability
- OpenJDK support

You will benefit from Payara's partnership with Azul, providing you with access to use Azul Platform Core for Distribution (previously Zulu Embedded) with Payara Server.

You will receive all these benefits and more with a Payara Enterprise subscription – find full list here. With Payara Enterprise, you will join the list of global companies who trust our solution to deliver their mission critical environments, including some from the Fortune 500 list. For example, Rakuten Card moved to an 100% cloud-native architecture using Payara Enterprise, and BMW successfully migrated from GlassFish to Payara Server in 2016 – and never looked back!

Payara and JVM Implementations

You can use any supported JDK with Payara Server.

Payara Platform Community currently runs with the JDK 8, 11 and 17 iterations of the following:

- Azul Zulu OpenJDK
- Oracle OpenJDK: 8, 11, 17
- Amazon Corretto
- Adoptium
- Adopt OpenJDK with Eclipse OpenJ9

This extends to any JDK based on OpenJDK 8u162+ or 11.0.5+ or 17.0.2

Payara Platform Enterprise will run with JDK 8 and 11 iterations of the above, JDK 17 support will follow.

If you do decide to move to Payara Enterprise for mission-critical projects, you can benefit from our partnership with Azul. With Payara Enterprise, support for Azul Platform Core OpenJDK, the Enterprise build of Zulu, is included.

Conclusion

There are many implementations based on the OpenJDK codebase, which goes back to the first releases of Java by Sun Microsystems. Since the code was open sourced 10 years ago, many companies have created their own build and some have additions for performance or client-side applications.

OpenJ9 also has very good runtime specifications for performance and memory usage.

The GraalVM has large parts of the OpenJDK in it, the JIT Compiler and the possible integration of other languages at runtime, including non-JVM-based languages like Python, Ruby, and C++. The Graal JIT compiler also makes it possible to have the ahead-of-time compilation functionality.

As you can see, there are pros and cons to all JVM implementations. Your decision will be guided by what other technologies you want to integrate with - for example, if you are using Azure, you may want the Microsoft OpenJDK, or Dragonwell with Alibaba Cloud. A need for lower memory usage may lead you to OpenJ9 - or maybe you were using it before Dynamic Archive was in the OpenJDK and see no reason to change. If polyglot programming is important to you, perhaps GraalVM is your solution.

There is no single right answer. Make sure to have a good look around to choose the optimal JVM installation for your use case.

Further reading:

- OpenJDK Support Payara Server Information Page
- VIDEO: 7 Reasons to Switch to OpenJDK 17 as a Jakarta EE Developer
- FREE EBOOK: Dismiss the Java Myths
- Comprehensive Getting Started Guide – Payara Server

sales@payara.fish **+44 207 754 0481** **www.payara.fish**