LEARNING MADE EASY

Azul Special Edition

OpenJDK Migration

for dummies®
A Wiley Brand

Why enterprises are moving off Oracle Java

Evaluating OpenJDK versus the Oracle JDK

Choosing the right Java partner

Compliments of

azul

Simon Ritter
Java Champion

About Azul

Headquartered in Sunnyvale, California, Azul provides the Java platform for the modern cloud enterprise. Azul is the only company 100 percent focused on Java. Millions of Java developers, hundreds of millions of devices, and the world's most highly regarded businesses trust Azul to power their applications with exceptional capabilities, performance, security, value, and success. Azul customers include 35 percent of the Fortune 100, 50 percent of the *Forbes* Top-Ten World's Most Valuable Brands, all ten of the world's top-ten financial trading companies, and leading brands like Avaya, Bazaarvoice, BMW, Credit Suisse, Deutsche Telekom, LG, Mastercard, Mizuho, Priceline, Salesforce, Software AG, and Workday. Learn more at azul.com and follow us @azulsystems.

OpenJDK Migration

OpenJDK Migration

Azul Special Edition

by Simon Ritter

for dummies®
A Wiley Brand

OpenJDK Migration For Dummies®, Azul Special Edition

Published by
John Wiley & Sons, Inc.
111 River St.
Hoboken, NJ 07030-5774
www.wiley.com

ISBN 978-1-394-20672-8 (pbk); ISBN 978-1-394-20673-5 (ebk)

Publisher's Acknowledgments

Some of the people who helped bring this book to market include the following:

Editor: Elizabeth Kuball

Acquisitions Editor: Traci Martin

Editorial Manager: Rev Mengle

Client Account Manager: Cynthia Tweed

Production Editor: Saikarthick Kumarasamy

Special Help: Faithe Wempen

Table of Contents

Introduction

Modern business runs on Java. It's the programming language of choice for large-scale applications, whether they're running in the cloud or in a private data center. Despite being nearly 30 years old, Java remains one of the most widely used programming languages in the world thanks to its versatility, reliability, and stability. Java is also far more pervasive than meets the eye. The Java platform forms the foundation of derivative languages like Kotlin and Scala, as well as frameworks like Hibernate and Spring and platforms like Cassandra, Hadoop, and Kafka.

Much of Java's appeal stems from its open nature, both in terms of the way it's defined through the Java Community Process (JCP) and in terms of the way it's developed under the open-source OpenJDK project. By paying for commercial support, companies can enjoy all the benefits of open-source software with the stability, reliability, and service they expect from an enterprise vendor. Since Oracle's acquisition of Sun Microsystems in 2010, many companies have subscribed to Oracle Java SE for commercial support.

However, lately, a series of licensing and pricing changes for Oracle Java SE has prompted many companies to move to alternative distributions of OpenJDK. Interest in OpenJDK, and in commercial support for OpenJDK, has intensified over the last four years as Oracle Java SE has become increasingly more expensive.

In this book, you find the details of migrating to an OpenJDK distribution, either from the Oracle JDK or from another OpenJDK distribution. You also see how to evaluate different distributions and support offerings and choose the distribution that's right for your organization.

About This Book

OpenJDK Migration For Dummies is written first and foremost for the application owners, IT operations teams, and engineers who are responsible for undertaking a migration. Migrating from one distribution of OpenJDK to another is straightforward for the

majority of applications, but some factors may require attention. Forewarned is forearmed, and the information in this book will help you complete a successful migration. At Azul, we've learned from hundreds of migrations and believe we can literally write the book on migrating without risks (or at least with very minimal ones). And so we are!

Foolish Assumptions

In writing this — or any book — authors inevitably end up making assumptions about their readers — sometimes mistakenly. Here are some of our beliefs about you:

- You're familiar with open-source software (OSS).
- You care enough about your Java applications to want to run them on the very best Java Development Kit (JDK) available.
- You're cost conscious. You'll pay for quality service, but you expect to be charged a fair price.
- You seek out technology providers who offer true partnerships.

Icons Used in This Book

Following the convention of other *For Dummies* books, we put icons in the margins of the book to make it easier to find the material you're most interested in. Here's a guide to what those little pictures mean:


TIP

We use the Tip icon to flag information that will make your migration easier and may save you time and effort down the road.


TECHNICAL STUFF

We're confident some readers will enjoy the technical nitty-gritty as much as we do, but others will appreciate knowing that there are sections that are okay to skip. Paragraphs marked with the Technical Stuff icon contain details that go beyond the basic understanding that most readers want to gain.

WARNING

We use the Warning icon when we cover old technologies that may trip you up. Make sure you know whether they're included in your Java footprint before you begin your migration.

REMEMBER

We use the Remember icon to highlight the most important takeaways. If you want the ultra-compressed version of this book, browse these icons.

Where to Go from Here

We've done our best to write this book from a neutral perspective. But let's face it, we're a little biased. We pride ourselves on offering the industry's leading OpenJDK distribution with hundreds of millions of downloads and backed by the best support team in the business. If you'd like to learn more about Azul, we have some suggestions.

- Azul Zulu Builds of OpenJDK (Zulu JDK or Zulu for short) are the Oracle JDK alternatives selected by hundreds of thousands of Java developers, powering all manner of applications around the world.
- Azul Platform Core, the largest provider of commercial support for OpenJDK, is trusted by the world's most highly regarded businesses to provide critical Java updates. Find out more at `www.azul.com/products/core`.
- Azul Platform Prime is our hyper-optimized runtime that maximizes performance while driving down infrastructure costs. Find out more at `www.azul.com/products/prime`.
- Azul Vulnerability Detection is a way to ensure the maximum security for your Java Virtual Machine (JVM) and runtime libraries. Find out more at `www.azul.com/products/vulnerability-detection`.

Finally, we couldn't fit all the information we wanted to share on migrating to OpenJDK in an 80-page book, so we built a website (`www.openjdk-migration.com`) where you can find additional information, as well as helpful spreadsheets and sample workplans.

IN THIS CHAPTER

- Understanding the complex licensing and pricing of Oracle Java SE
- Taking stock of Oracle's Java SE licensing subscription
- Replacing the Oracle JDK with OpenJDK
- Understanding TCK testing and what it means for migration

Chapter 1
Replacing Oracle Java SE in the Enterprise

One of Java's most impressive characteristics has been its ability to evolve over nearly 30 years to address the needs of enterprise applications. This result is a combination of the open-source project OpenJDK, the specifications created through the Java Community Process (JCP), and the vibrant and active community of users. Java is the number-one language for overall development today, with more than 60 billion active Java Virtual Machines (JVMs) and 38 billion cloud-based JVMs.

Enterprises have become increasingly reliant on the Java platform in no small part because of its scalability to even the largest internet and transactional workloads. To maximize their investments in Java, they've sought trusted advisors to provide commercial support to augment their own teams.

After Oracle acquired Sun Microsystems in 2010, it continued to deliver Java in a primarily free and flexible way, just as Sun had done. Things became more complicated in 2018, though, when Oracle changed how it licensed its distribution of the Java Development Kit (JDK), which includes the Java Runtime Environment (JRE) and the JVM, and provides a platform for running Java applications.

This chapter covers the challenges created as Oracle repeatedly changed how it priced its support services and licensed its JDK, leading many organizations to consider migrating to OpenJDK.

Untangling Oracle Java SE Licensing and Pricing Complexity

Oracle's changes in the way it licences and prices its JDK in recent years have spurred lots of interest in OpenJDK. This section reviews that chain of events.

In September 2017, Oracle announced significant changes as to how Java would be released moving forward. The new model, as approved by the OpenJDK governing board, provided a new release every six months (in March and in September). Developers had been requesting a more agile approach to developing the core Java platform for some time, and this change led to more features being added to Java — and more rapidly — than ever before.

This change in strategy also resulted in a major change to the way Oracle delivers updates and support. Instead of providing extended maintenance and support for all Java versions, only Long-Term Support (LTS) releases would qualify for that.

More changes came in June 2018, when Oracle announced its new licensing and pricing for JDK, which it bundled with the Java SE subscription.

A Java SE subscription included:

- Certified compatible updates for performance, stability, and security updates
- Security-only updates (curated updates)
- Technical support

Paid support is a familiar component of open-source communities, and the change was accepted — but a move Oracle made three months later wasn't. Oracle announced it would end free public updates for commercial use for Java 8 — which was then the most popular version of Java — in January 2019. Updates would still be

available with a subscription to Java SE; however, if you installed the updates without a subscription, you'd be liable to Oracle for the cost of the subscription.

Then, in April 2019, Oracle unveiled a new Java license: the Oracle Technology License Agreement for Oracle Java SE.

The cost for using Oracle Java under these new terms increased. Organizations like Cornell University began to take steps to minimize or eliminate their dependency on Oracle Java. In Cornell's case, it published a blog post asking users of its network to uninstall Oracle Java from their computers and to install OpenJDK if they needed Java.

Oracle's popularity with Java developers plummeted. According to a survey of 2,000 developers by Snyk (`https://snyk.io`), only 34 percent reported using Oracle JDK in the second half of 2019 — down from 70 percent the year before.

On September 14, 2021, Oracle announced a new No-Fee Terms and Conditions (NFTC) license that partially rolled back the changes announced in 2019. A new LTS release would now be free under NFTC until one year after the next LTS release. This gave users a chance to transition their applications to the next release. The NFTC license was available for developing, testing, prototyping, and demonstrating applications and personal use of the Oracle JDK or for use related to internal business operations. Oracle also shortened the time between LTS release from three years to two years.

The move was generally viewed as positive, but it had little effect on OpenJDK's growing momentum.

Taking Stock of Oracle's Java SE Universal Subscription

In this section, we cover Oracle's fourth major change in four years — and the one that prompted the loudest outcry.

On January 23, 2023, Oracle quietly replaced the online link to the Oracle Java SE Subscription Global Price List with a link to a new Oracle Java SE Universal Subscription Global Price List.

The old pricing was based on a processor-count metric for servers and a named-user-plus metric for desktops. It was far from perfect. With the processor-count metric, the license cost was determined by multiplying a processor's total number of cores by a core processor licensing factor specified by Oracle. The named-user metric applied to all individuals authorized to use a program regardless of whether they were actively using it. These definitions left the door open for uncomfortable conversations with Oracle reps who were famous for setting up meetings "to talk about your Java usage." Unpleasant in themselves, these meetings could be a prelude to painful Java audits.

The new pricing seemed straightforward. Instead of counting processors and authorized users, organizations needing an Oracle Java SE subscription would count their employees. But how? The fine print contained a complex definition and caused many existing customers to do a double take.

Employee-count metrics are not uncommon in the technology world. Companies like Dropbox, Okta, Salesforce, and Workday all charge per employee per month (PEPM). Oracle itself charges on a PEPM basis for products like Oracle Cloud HCM. But typically, PEPM pricing is tied to usage.

The new pricing for Oracle Java SE was disconnected from actual Java usage and could result in a massive price increase. The more employees, the larger the price increase. Some examples:

- The Java subscription costs for a small 20-person company with one quad-core server and 20 Java desktops would triple.
- A 100-person company with two quad-core servers and 100 Java desktops would see a 328 percent price increase.

Replacing the Oracle JDK with OpenJDK

One question we've been asked a lot over the years is how difficult a migration will be. Behind this question, there is typically an assumption that the migration will be a heavy lift.

In fact, migrating to certified builds of OpenJDK can be very straightforward and simple for the vast majority of enterprises. If you're migrating server applications, you're not likely to

encounter any challenges. There are just a few potential issues, typically involving older, deprecated desktop technologies (see Chapter 2).

Migrating from Oracle JDK to an OpenJDK distribution is so straightforward because the Oracle JDK and OpenJDK distributions that companies like Amazon, Azul, BellSoft, and Red Hat provide are all built using the OpenJDK source code and undergo extensive compatibility testing. (Chapter 2 covers some additional Oracle JDK components that are not from the OpenJDK project.)

REMEMBER

At its simplest, migration consists of installing the new OpenJDK distribution and configuring your application to use it instead of the previously installed JDK. Ninety-nine percent of the time, your application will work perfectly.

This book covers the 1 percent of the time when something *doesn't* work as expected, and explains how to avoid that or diagnose and fix it. Later chapters cover these corner cases exhaustively. But before getting into that, you may want to understand how OpenJDK providers can be so sure that their distribution is a drop-in replacement for the Oracle JDK.

Understanding TCK Testing and What It Means for Migration

The Java Technology Compatibility Kit (TCK) was created to ensure compatibility between different implementations of the Java specification. It's essential for Java's portability — for delivering the "write once, run anywhere" promise. The TCK provides a high level of confidence that an application that runs on one TCK-tested distribution will run the same way on another distribution that has also passed the TCK test suite.

REMEMBER

Originally developed and licensed by Sun Microsystems, today the TCK suites are Oracle's intellectual property and require an Oracle license.

To claim compatibility with Java SE, individual binaries within each OpenJDK distribution must pass the TCK suite associated with that version of Java. TCK suites consist of more than 100,000 tests (about 126,000 tests for Java 8 and about 139,000 tests for Java 11).

Examples of TCK tests include language and application programming interface (API) tests, which verify that an implementation of Java meets the required specification of the syntax, semantics, and functionality of the language and its associated libraries. In addition, TCK tests for the JVM ensure compatibility across different operating systems and hardware architectures. TCK tests cover every aspect of the JDK, extending to features like printing, sound, and graphical interfaces.

REMEMBER

Passing the TCK suite ensures that one distribution of a particular binary — say, JDK 17.0.7+7 — can be swapped for another distribution of that same binary that also passed the TCK. In this way, Oracle JDK can be exchanged for Azul Zulu Builds of OpenJDK, for example.

Passing the TCK also carries an additional benefit critical for commercial use: Conforming implementations inherit the right to use all the intellectual property in the specification defining that Java version.

Thanks to TCK testing, enterprises have a choice of OpenJDK distributions to choose from. These distributions differ from each other in ways that we explain in Chapter 4. For now, the important thing to understand is that you have options, and your chances of finding a distribution that meets the exact needs of your organization are high.

IN THIS CHAPTER

- Setting your goals for migration
- Understanding the three-phase migration process
- Identifying the risks of older technologies

Chapter 2
Preparing for Your Migration

What are the most important things to do before starting your migration? In this chapter, we walk you through the recommended sequence of steps and the potential issues you should know about before you roll up your sleeves, fire up some scripts, and take your Java inventory.

Identifying Migration Goals

As in any other endeavor, the success or failure of a migration will be measured by how well you met your goals. What are you hoping to achieve?

There are several reasons why an organization may choose to migrate to an alternative OpenJDK distribution from Oracle. In addition to lowering the total cost of ownership (TCO) of Java apps, frameworks, platforms, and tools, common goals include receiving support for older versions of Java (such as Java 6 or 7) that have reached their end of life, as well as minimizing the impact of a migration on application users and equal or better application performance.

Many organizations poised to migrate wonder how long the process will take. There isn't a one-size-fits-all answer to that question, though. The time needed for a migration depends on at least half a dozen variables that are specific to your organization. One of those variables is your migration goals; some goals take longer to achieve than others. For example, if your goal is to completely transition off Oracle Java as quickly as possible, you'll have a different migration plan than an organization that is primarily seeking support for legacy applications running older versions of Java like Java 6 and 7 and would prefer a phased approach over a longer period of time.

Introducing the Three-Phase Migration Process

After you've identified your goals, the methodology we suggest for migrating from the Oracle Java Development Kit (JDK) can be broken down into three stages:

1. Discovery. Identify which versions of Java are being used by which applications and on which machines within your organization, including cloud instances. You'll use this inventory to create a migration plan.
2. Execution. For each machine that requires a Java runtime, install the same version (or versions) of the OpenJDK distribution you choose.
3. Validation. Test your applications to verify that everything works as expected.

The rest of this chapter covers the discovery stage, where a little bit of additional attention can have the biggest impact on shortening your migration time frame and eliminating issues. Chapter 3 walks you through the execution and validation stages.

Making an Inventory of JDKs Currently in Use

The first stage is often the most time-consuming in a JDK migration because of the variety of JDK versions in use. Typically, when a new application is deployed, it uses the latest version of the JDK

at that time and continues to use it even as newer versions of Java are released. This is quite logical, because it has been tested with the deployed version.


REMEMBER

Assuming the application works without issue, there is no requirement to update its JDK version. An update is required only when security patches and bug fixes are no longer available for the version in use.

Building your inventory


TIP

To create a complete JDK usage inventory, you must examine each machine in your estate that runs any Java Virtual Machine (JVM)–based applications. This can be straightforward if you use IT asset management (ITAM) tools to monitor software usage. Many enterprises deploy these tools to ensure that they're complying with licensing terms and conditions. These tools can quickly produce a report showing which machines have which versions of Java installed.

Even with a report like this, however, producing a definitive inventory of JDKs in use may not be straightforward. For example, many enterprises use standard builds for server and desktop deployments. These may include a Java runtime even where no Java applications will be used.

Many users won't be using ITAM software, so they'll need to perform a manual inventory of machines. Tools are available to assist with this. These tools can scan the filesystem of a machine, looking for a Java executable and recording the version string when it's run. They can also scan the process table to determine whether Java applications are running and which JDK they use. Take care in analyzing these results — applications may be used only when required. If an application isn't running at the time the process table is being scanned, it won't be included.

Deciding which JDKs to include

The process of choosing which JDKs to include is similar to one you might follow when dealing with a collection of cords and power adapters for mobile devices. Over time, you build up a collection of different adapters, each applicable to one or more devices. When clearing out your collection, you must determine if each is still required. The same goes for JDKs; you must figure out which ones still have a purpose and which ones don't match up with anything you still run.

TIP

Typically, when a JDK is installed using an automated installer like a Microsoft Software Installer (MSI) file on Windows or a Red Hat Package Manager (RPM) file on Linux, it becomes the default Java runtime. However, some applications may be installed using a bundled JDK that is used only for that application. In such situations, the JDK may be included in the application's support contract. In many (perhaps most) cases, maintaining that support is desirable and you won't want to migrate that application. When compiling the inventory, you'll want to be aware of such applications and, when in doubt, confirm the terms of ongoing JDK support with the application provider.

With the continued trend of moving server applications to the cloud, it's also necessary to include JDKs used in these environments. Most cloud providers supply and support Java runtimes as part of their platform as a service (PaaS) offering. However, there may be situations where a different JDK is used (for example, to deliver better performance).

It's becoming increasingly common to use microservice architectures to deploy applications into the cloud. Doing so separates monolithic applications into multiple, loosely coupled services packaged in containers. A *container* is a fully functional, portable, virtualized computing environment surrounding a service and isolating it from other services. This is a very flexible deployment method, because it eliminates the need to ensure that the correct versions of libraries, frameworks, and runtimes are available where the service is deployed. All dependencies, including the JDK, are bundled into the container. Once again, when creating a JDK inventory, all containers that include a Java runtime should be added.

Deciding what information to collect about each JDK

For each entry in the inventory, there should be the following fields at a minimum.

- **Type:** A physical desktop, physical server, cloud instance, and/or container.
- **Access details:** The credentials needed to access a physical machine or cloud instance via a network connection with sufficient privileges to permit the JDK's installation. For

containers, these details will relate to how the container image is generated. This will probably be via continuous integration/continuous delivery (CI/CD) tooling and should include information enabling you to configure a different JDK for inclusion in the image.

- **Operating system (OS):** Which OS is in use, plus additional details about it, such as the edition, version, build number, and whether it's 32 bit or 64 bit. The OS will typically be Windows, macOS, or Linux. For Linux, you should also note the distribution (distro) because it may make a difference in the installation format.
- **Automated or manual install:** Whether the installation will take place using an installer. If the JDK doesn't use an installer, this field should also note the location in which the JDK should be manually unpacked.
- **JDK version:** This field should also include the installed update level, such as JDK 8u202.

At the end of this process, you should have a complete list of all places where the Oracle JDK is located.


TIP

For classification purposes, you may want additional fields. Visit `www.openjdk-migration.com/inventory-worksheet` for a complete worksheet.


TIP

If you have a large Java deployment, you'll be looking for additional resources to assist with the inventory. You can find sample worksheets and a survey to use with application owners at `www.openjdk-migration.com/application-owner-survey`.

Recognizing the Risks of Older Technologies

Next, you should start thinking about the potential problems that may crop up based on what you found when making your inventory. This section describes some potential issues that you may encounter when migrating applications from the Oracle JDK to OpenJDK distributions. Altogether there are about a dozen possible edge cases to be aware of. It's unlikely any of these will affect your applications. Almost all relate only to desktop applications

that use specific deployment technologies. If your organization is among those still using desktop apps, you'll want to be aware of these.

Very old versions of Java

Java started life in the mid-'90s and has continued to be developed ever since. Its popularity took off quickly, resulting in people using early versions of the platform to deploy enterprise-wide and mission-critical applications. Most JVM-based applications are replaced or upgraded to a newer version of Java before support for the JDK version expires. However, some applications continue to use very old versions of the Java platform, and that can be a problem.

Through Azul's Platform Core support, you can continue to receive all scheduled and out-of-bounds updates to JDK 6 and 7, even though Oracle and other distributions have discontinued support. (OpenJDK started with Java SE 7, but a project for Java SE 6 was subsequently created. You can read more about the early days of Java and OpenJDK in Appendix A.)

Versions of Java prior to JDK 6 were never released as open source, so there is no way to provide distributions with backported security patches or bug fixes. The last public update to JDK 5 was in November 2009 (which was when Sun Microsystems was still developing Java).

WARNING

Because of Java's excellent backward compatibility, it's quite probable that something that runs on JDK 5 (or even earlier versions) will run without issue on JDK 6. However, if you're still running applications from JDK 1.0 or 1.1, these may not work on JDK 6.

Oracle JDK-specific features

Prior to JDK 11, the Oracle (and Sun Microsystems) JDK included features that were not included in the core OpenJDK project. The following sections cover these features.

JavaFX

JavaFX is a cross-platform graphical user interface (GUI) tool kit for Java. Although it was never included in the Java SE specification or the main OpenJDK project, it was released as a separate open-source project in 2011, called OpenJFX.

TIP

If you're using JavaFX in your applications, you'll want to use a distribution like Azul that provides OpenJDK builds, including JavaFX libraries. These use the OpenJFX source code and build scripts to provide identical functionality to the JavaFX provided in older (JDK 8, 9, and 10) builds from Oracle.

Migrating JavaFX-based applications from the Oracle JDK to an alternative distribution is very straightforward, provided you use a build with JavaFX libraries. There is no requirement to change any application code or recompile the application.

Applets

Applets are small applications designed to add interactive content to a web page. Introduced in the mid-'90s, applets ran in web browsers and reached their heyday in the late '90s — until browsers withdrew support for them due to security concerns. Vulnerabilities became apparent both in applets (backdoor, cross-site scripting, and cross-site request forgery) and in the Java Runtime Environment (JRE) and browser (with attackers being able to track users, perform malicious exploits, and escape from sandboxes).

Applets should be considered a dead technology for several reasons:

- The Java plug-in, which is necessary to run applets in a browser, was deprecated in Oracle JDK 9 and removed from Oracle JDK 11.
- Even if you're using Oracle JDK 8 through a Java SE subscription, Oracle ended support for the Java plug-in in March 2019.
- Oracle has left the components required to run applets in Oracle JDK 8, but only for the Windows platform. These components were removed from Linux and macOS in July 2020 (JDK 8u261).
- Due to security and stability implications, no current mainstream browser supports the Netscape Plugin Application Programming Interface (NPAPI), which is necessary to use the browser plug-in. Therefore, it simply isn't possible to run an applet within most modern browsers, including Google Chrome and Mozilla Firefox. Support for the last legacy browser that still supported the NPAPI (and,

thus, the Java plug-in), Microsoft Internet Explorer 11, ended in June 2022. Since the Java plug-in remains closed source, only two options remain for continuing to run applets on a supported JDK:

- Convert the applet from being browser-based to running with a Java Web Start environment.
- Continue using your existing Java plug-in but modify registry entries to direct it to a different Java runtime.

TIP

For a migration worksheet on applets, see `www.openjdk-migration.com/applets`.

Java Web Start

In the early days of Java, internet speeds were very slow. Many people used dial-up connections that would run at a maximum speed of 57 kilobits per second (Kbps). Adding applets to web pages increased the amount of data that needed to be downloaded quite substantially, resulting in a much slower browsing experience.

Part of this problem was that every time a user visited a web page that included an applet, the code for the applet needed to be downloaded, regardless of whether it had changed since the last visit.

The alternative to applets was a full-blown application. Although this meant that the code for the application didn't need to be downloaded each time it was used, there was no simple way to determine if there was an update. If a user were told an update was available, they would need to manually download and install the application.

Java Web Start aimed to correct such problems by providing the best of both applets and applications through a web-centric application model.

TECHNICAL STUFF

Java Web Start provides a Java Network Launch Protocol (JNLP) client. JSR-56 specifies the JNLP and, at its core, is a JNLP file, which is an Extensible Markup Language (XML) document. The JNLP file describes an application in terms of the resources it requires, such as Java Archive (JAR) files, icons, Java runtime, and so on. All resources are specified as a Uniform Resource Locator (URL), enabling the application to be accessed remotely like an applet (but without using a web browser). Java Web Start caches

the resource files on the user's machine. Each time the application is started, it checks the locally held resources against the remote ones to see if there has been any update. If not, it uses the local files, so no network activity is required. This also enables the application to be used when there is no network connection.

IcedTea-Web

The Java Web Start included in the Oracle JDK remains closed source, so it isn't included in standard OpenJDK distributions. For users who need to continue using the Java Web Start functionality, there is an alternative, open-source project called IcedTea-Web.

IcedTea-Web processes JNLP files, as specified by JSR-56, and offers the most commonly used features of Java Web Start. However, it isn't a drop-in replacement, so an application's configuration may sometimes require changes to make it work in the same way as the Oracle Java Web Start.


TIP

When switching to IcedTea-Web, enable logging so that more detail about any issues will be provided. You can do this through the `itw-settings` utility included in IcedTea-Web. Alternatively, you can manually edit the `deployment.properties` file.

These are the most common issues with switching from Java Web Start to IcedTea-Web:

- **Custom security certificates:** By default, IcedTea-Web assumes that a trusted certificate has signed all JAR files. If the JAR files are self-signed, you must add the certificates to the certificate store. You can configure that using the `itw-settings` utility.
- **Network proxy settings:** This can be set manually or use the same settings as browsers using `itw-settings`.
- **Suppressing security checks on manifests:** You may need to do this if a signed JAR file's manifest is missing attributes such as permissions.
- **Unsigned JAR files:** For applications that use unsigned JAR files, you must modify the `deployment.properties` file for each user. You should change the `deployment.security.level` value to `ALLOW_UNSIGNED` and add the setting `deployment.itw.ignorecertissues` with a value of `True`. In addition, you must include the `–nosecurity` command-line flag when calling `javaw.exe`.

TIP

For a migration worksheet on Java Web Start, go to `www.openjdk-migration.com/web-start`.

Some Less-Common Considerations

This section lists the furthest corners of edge cases for OpenJDK migration and offers some possible solutions.

Font rendering

Prior to JDK 8, the Oracle JDK used a closed-source font rendering engine called Ductus. OpenJDK provided an open-source equivalent called Pisces. From JDK 11, both the Oracle JDK and OpenJDK use the same engine called Marlin. In most situations, this makes no difference. However, for a font that doesn't include a bold version, the rendering engine will extrapolate the plain font to bold. On JDK 8, this may appear different when using the Oracle JDK and an OpenJDK distribution.

The only solution to this is to use a font that includes a bold version.

Lucida fonts

Oracle JDK 8 and earlier included a set of default Lucida fonts. These fonts are used if no suitable alternative can be found on the target deployment system. Lucida fonts are commercial fonts and are not included with a default OpenJDK distribution.

Azul can provide a Commercial Compatibility Kit (CCK), which includes these fonts, if an application requires them.

NTLM authentication

In Oracle JDK 8 update 201, Oracle introduced a new security authentication parameter for Windows NT LAN Manager, `jdk.http.ntlm.transparentAuth`. By default, this parameter is set to `disable`, which differs from earlier updates where the value was effectively `allHosts`. This change may affect applications, including those deployed with Java Web Start/IcedTea-Web.

You can resolve the issue by modifying the `jre/lib/net.properties` file in the JDK installation. The value can be set to `allHosts`, but the recommended setting is `trustedHosts`.

TIP

Stack Overflow provides more detail on configuring trusted hosts for NTLM: `https://stackoverflow.com/questions/56840215/ntlm-no-longer-working-with-java-webstart-following-a-java-upgrade`.

Custom security configurations

The JDK uses a security sandbox model to limit an application's access to resources such as files, network connections, and so on. Some applications may require you to modify these settings in order to use custom security certificates and to change resource restrictions.

When migrating from the Oracle JDK, you'll need to replicate any of these changes for the new JDK installation.

Java Access Bridge

This is a technology that exposes the Java Accessibility API through a Windows-native library, enabling assistive technologies on Windows systems to work with Swing and Abstract Window Toolkit (AWT)–based applications.

When migrating to a build of OpenJDK, you should use the `jabswitch` command to configure the Java Access Bridge.

Java Control Panel

You can use the Java Control Panel to configure parts of the Java environment, such as security certificates and Java Web Start. OpenJDK doesn't include it, so it won't be part of a standard distribution. You can manage certificates manually using the JDK `keytool` utility, and IcedTea-Web has its own configuration utility, described earlier.

SNMP JMX Gateway

SNMP JMX Gateway is not included in OpenJDK, so it won't be part of a standard distribution. In the unlikely event that an app requires it, you can use the open-source library SNMP4J as a replacement. Consult the documentation for this library to learn how to install and configure it as an alternative.

Version string incompatibility

Some applications use the version string that the Java runtime provides to determine whether a supported JDK is in use. In the event that an application does rely on a particular version string, you should contact the application vendor.

Windows registry keys

The Oracle JDK installation for Windows adds registry keys that an application can use to locate the Java runtime. There are three settings, all located in `HKEY_LOCAL_MACHINE/SOFTWARE/JavaSoft/Java Development Kit`:

- **JavaHome:** The full path of the installed JDK.
- **MicroVersion:** Since JDK 5, this is always zero.
- **RuntimeLib:** The full path of the Java runtime dynamic link library (DLL).

Azul Zulu Builds of OpenJDK add similar registry settings to ensure compatibility. Other distributions may not.

Handling Third-Party Applications

It's very common for users to buy applications instead of developing them. This can be very cost-effective to not have to develop and maintain bespoke software in-house.

Many such applications that use Java will specify a required version of the JDK and even possibly a minimum version update level. (This is no different from the way other applications specify a minimum version of Windows or Linux.)

The user must source the JDK and make it available to the application. Application providers often state in their documentation that they'll support the application only if you're using the correct JDK. This is sensible for the application developer, because it can help eliminate issues caused by using out-of-date or inappropriate Java runtimes. Because the Oracle JDK has been so ubiquitous in the past, many application providers have stated that only the

Oracle JDK will qualify for support. With the recent changes to the Oracle JDK licensing and pricing, however, users are increasingly demanding that they provide support when running on alternative JDKs.


TIP

Many application vendors will now provide support so long as the app is running on a Technology Compatibility Kit (TCK)–certified build of OpenJDK. Because they can trust such distributions to be functionally identical to the Oracle JDK, they don't need to be worried about testing multiple distributions with their applications.

Sometimes an application will state that certain OpenJDK distributions are valid for support but not the one you want to use. In this case, you should contact the application vendor to have them add the distribution you want to use to their list. As long as it's TCK tested, the vender should have no objection.


REMEMBER

In any large organization, where multiple Java applications are in use, applications will have different owners. When planning a successful migration, it's essential to include all concerned parties — that is, all owners of the applications that must use the new Java runtime.

IN THIS CHAPTER

- Surveying the list of available formats
- Updating the JDK
- Making sure your applications perform as expected

Chapter 3
Migrating Your Applications

With a survey of your Java runtime usage complete, it's time to move on to switching all your Java Development Kits (JDKs). This chapter explains the process of choosing a format, selecting an update version (probably the latest one available), and testing your applications post-installation to make sure everything is working as expected.

Reviewing Available Formats

This chapter's examples use Azul Zulu Builds of OpenJDK (also referred to as the Zulu JDK) as the new JDK. You can use any JDK distribution you prefer, but other distributions may differ in some ways, such as installation formats or supported platforms. (Chapter 4 explains how to select the right distribution for your organization.)

You can download Azul Zulu Builds of OpenJDK in all the formats available for the Oracle JDK. The following sections review the formats available and provide a few key facts about each one.

Windows

Windows JDKs typically come in two installation formats:

- For a manual install, archive files are available in ZIP format. Copy the file to the appropriate location and extract the file.
- Microsoft Software Installer (MSI) files are available for automated installation, including updating registry entries and environment variables. You can copy MSI files to individual machines, or you can install remotely using `msiexec` and `psexec`.

Linux

Linux JDKs are available in six installation formats:

- Archive files are available either in ZIP or compressed TAR files. As with Windows, you can copy these to the appropriate location and extract the file.
- Debian Format Package (DEB) files are available for automated installation using the `dpkg` command on Debian-based distributions.
- Red Hat Package Manager (RPM) files can be used for automated installation using the `rpm` command.
- Two Azul package repositories are available for Linux: one for use with `apt` (for Debian-based distributions) and one for use with `yum` (for Red Hat–based distributions). You can find instructions for configuring and using these on the Azul website.

macOS

macOS JDKs can be installed using three formats.

- Archive files are available either as ZIP or compressed TAR files. As with Windows and Linux, copy these to the appropriate location and uncompress them.
- Apple Disk Image (DMG) files can be used for automated installation, either graphically or via `hdiutil attach` from the command line.

Docker

Multiple image files are available from Docker Hub to install a JDK.

Zulu Docker files are available from Docker Hub for Alpine, CentOS, Debian, and Ubuntu Linux distributions. You can use these files as the base for building containerized applications and services.

Performing the Update

OpenJDK distributions do not support patch-in-place for updates; applying an update to the JDK requires installing a whole new JDK. This means the migration installation process can be treated exactly like deploying a new update, except that it installs the Zulu JDK update instead of the Oracle JDK update. Figure 3-1 provides an example of the process.

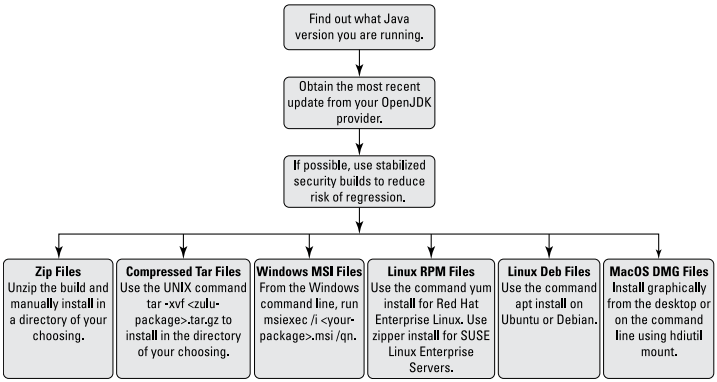


FIGURE 3-1: Performing an update on a server running Java applications.

REMEMBER

Unless there is a specific bug causing application stability issues, users won't see any pressing need to move to a new version of Java, even if they are using quite an old version. However, from an administrative standpoint, it's always a good idea to install the latest update when migrating to a new OpenJDK distribution, to maintain the highest level of security for your applications.

In the vast majority of cases, the update process is straightforward, and you don't encounter any issues. However, sometimes an update includes changes that can change an application's behavior.

Let's look at a particular example using the very commonly used Apache Tomcat servlet engine. Suppose we have Tomcat 8 running on Oracle JDK 8u202. This is not the most recent release of Tomcat, but it works perfectly for our application, so it hasn't been upgraded. We have a servlet running that takes data from the client, makes a query in a MySQL database, and returns a result.

For our migration, we install Zulu 8 update 372 and restart the server. However, when we try to use the application, we get an error message telling us there was a communication link failure. This has nothing to do with switching from Oracle to Zulu. Instead, it results from a change made in JDK 8 update 291 from October 2021. In this update, the default settings for Transport Layer Security (TLS) were changed to disable TLSv1.0 and TLSv1.1 by default. Therefore, to get the application working as before, we must modify the `jre/lib/security/java.security` file and remove the TLS references from the `jdk.tls.disabledAlgorithms` setting.


TIP

When something like that occurs, it's important to verify that the problem is caused by using a different *update* of the JDK rather than a different *distribution.* Having a commercial support provider who can provide older updates can be extremely beneficial in this case. Simply install the same update level of JDK, and then retest the application. In our example case, the Azul support team would be able to provide details of the configuration changes required to resolve the issue.

After installing the new JDK, you'll need to make any required changes to switch applications to use it. For example, it may be necessary to change the `JAVA_HOME` environment variable and the launchers used for individual applications, such as the Tomcat servlet engine. The safest option is to change this environment variable on *all* migrated machines.

Testing Your Applications

After installation, you should test all applications that have been switched to the new JDK to ensure correct functionality. You probably won't observe any different behavior unless changes have occurred between updates.

The testing process will vary greatly depending on each application. Internally developed applications may have extensive regression tests that can fully exercise all parts of the application to verify correct behavior. Third-party applications (either open-source or commercial) may have a set of standard tests that you can run. If not, an experienced user should run the application and try as many functional aspects as possible.

After all tests have passed to each user's satisfaction, the migration is complete. You'll now be in a strong position to maintain your Java estate to the highest levels of security and stability, often more so than before. In addition, you'll have a clear picture of where the Java runtime is in use and more experience in upgrading machines to the latest Java update.

IN THIS CHAPTER

- Differentiating between OpenJDK distributions
- Answering commonly asked questions
- Comparing OpenJDK distributions

Chapter 4
Evaluating OpenJDK Distribution Providers

OpenJDK is an open-source project, and anyone is free to download the source code for a specific version of the Java Development Kit (JDK). Using the provided build scripts, anyone can compile all parts of a JDK for mainstream platforms, such as Windows running on a 64-bit Intel processor. These executables and libraries can be packaged and provided as an OpenJDK distribution, and there are lots of different distributions available. This means you have your choice of providers — from free and unsupported to commercial providers that offer all the support of Oracle Java SE and more.

It also means that it's worthwhile educating yourself on how providers of OpenJDK differ from each other. The OpenJDK distribution you choose can affect the ease of your migration — and have an even greater impact on your Java estate's stability and security.

Differentiating Between OpenJDK Distributions

Because all OpenJDK distributions use the same source code as a starting point, which distribution you choose will depend on a variety of other considerations.

Here are some of the most important factors to consider:

- **Which versions are supported?** As of March 2021, there were 20 versions of Java, including three releases known as Long-Term Support (LTS) releases. It's easy to find support for the most current LTS release, but the farther back you go, the fewer providers there are to support that version. For example, Azul is one of only two providers that support JDK 6 and 7.
- **Which platforms are supported?** Most users run applications on mainstream operating systems like Linux and Windows and use common processors like those from Intel and AMD. If your environment includes less-common platforms like Automatic Resource Management (ARM)–based processors or perhaps the Solaris operating system, will the distribution provide builds for these?

A MULTITUDE OF BINARIES

Commercial support providers typically offer updates for just a fraction of the JDKs that are used in production.

There are more than 2,000 possible binaries if you only consider LTS versions for the most popular Linux, Windows and macOS operating systems; for 32-bit and 64-bit x86 and ARM architectures; and for just the JDKs and Java Runtime Environments (JREs) delivered in the most popular file types. If you expand this criteria, there are potentially thousands more.

Organizations like Azul also provide binaries for additional versions of Java (Azul currently supports 6 Java versions, 14 operating systems, 7 architectures, and 4 packages).

- **How long will a version be supported?** Certain versions are classified as LTS, but how long is *long?* Different distributions may offer various maintenance and support lengths. Some distributions may also commit to supporting a version after they stop providing the scheduled updates. In this passive support phase, users can still report issues and, if necessary, the build provider may provide a special update containing a fix.
- **Are all builds Technology Compatibility Kit (TCK) tested?** Each version of Java has its own suite of TCK tests. TCK testing is vital to ensure that the binary distribution being delivered implements the specification exactly, providing full compatibility.
- **How quickly are updates available?** The scheduled JDK updates are developed through the OpenJDK project and embargoed by the OpenJDK Vulnerability Group until a preplanned date and time. What is a distribution's track record for providing updates within hours of the embargo lifting? Has it seen long delays in the past? Is there a time-based service-level agreement (SLA) specifying when an update is guaranteed to be available?

  Update speed is important because, after the embargo is lifted, details of security vulnerabilities are made public, and those who want to do so can start developing exploits. If updates only become available days or even weeks later, the possibility of systems being compromised rises exponentially.
- **Are stabilized updates available?** Oracle Java SE provides two formats for each update: the Critical Patch Update (CPU; a stabilized security update) and Patch Set Update (PSU; the full update). To maintain the maximum level of JDK security, both are essential. Only Oracle and Azul provide both CPU and PSU updates.

REMEMBER

Azul sets the industry standard for delivering security updates to OpenJDK on a strict SLA. Since Oracle Java SE was launched, Azul has delivered stabilized security updates within an hour of the lifting of the embargo.

Answering Common Questions

In addition to the questions in the preceding section, organizations that are planning a migration frequently share common concerns. Here are several broader questions that often come up

about the impact of switching from the Oracle JDK to another OpenJDK distribution.

Will I lose functionality if I switch?

From JDK 11 onwards, Oracle JDK has used only the source code included in the relevant OpenJDK repository. Prior to that, Oracle also included several features that were not open source.

These features are almost all associated with desktop applications and deployment technologies. The two most obvious are the Java plug-in (required to run applets in a browser) and Java Web Start. Chapter 2 provides some specifics on both of those.

The Oracle JDK features that don't apply just to desktops include Mission Control and the Java Advanced Management Console. Oracle open-sourced Mission Control, a low-overhead, interactive monitoring and management tool for Java workloads, and you can obtain builds from Azul and other vendors. The Java Advanced Management Console, a sysadmin tool, is available only from Oracle.

What about Oracle applications?


REMEMBER

If you're using Oracle Java applications, you should check carefully to determine whether the JDK is included in the support contract. If it isn't included, consider which distribution is the best choice in this situation.

What's my risk of regression when using alternatives to Oracle?

Provided the OpenJDK distribution you choose is built from OpenJDK source code and is TCK tested, there is essentially zero chance of a functional regression created by swapping one distribution for another. Applications will behave no differently when running on an Oracle JDK alternative.

Only in a very extreme case would you notice a performance regression where an application may be slower or faster than before. This would only be because a different compiler or compiler flag was used during the build process.

Will I need to move to the latest Java JDK version?

All distributions of OpenJDK provide extended maintenance for the LTS versions of Java in the form of updates. How long these updates will continue to be delivered depends on the distribution.

If you're using an older version of the Java JDK that is still being maintained, your application will continue to receive the maximum level of security and stability from its runtime. There is no need to update to the latest Java version.


REMEMBER

For some users, the availability of newer features and potentially improved performance will be worth investing time and resources to migrate and test applications. It's important to understand, though, that moving to the latest Java version isn't essential.

Will I need to recompile my application?

From the very beginning, Java used the slogan "Write once, run anywhere." The cornerstone of this claim was that all Java runtimes would provide identical functionality. The same bytecodes would execute in the same way regardless of the operating system and chip architecture.

This concept applies equally when moving to a different distribution of OpenJDK. As long as both the current and new distributions are of the same Java version and have passed all the TCK tests, you won't need to recompile any application code.


REMEMBER

Sometimes changes made in an OpenJDK update can impact application stability. In this case, it may be necessary to change configuration parameters, but it still won't be necessary to recompile the application.

Will I need to rewrite or modify my application code?


REMEMBER

There is no need to recompile code when switching OpenJDK distributions. By extension, therefore, it isn't necessary to modify or rewrite any application code as long as the Java version of both distributions is the same.

RECOMPILING CAVEATS

If you do recompile existing code with a newer JDK, here are a few things to watch out for:

- **If the code is very old, you may find that certain variable names are no longer valid.** Java SE 1.4 introduced assertions, and *assert* became a reserved word. Similarly, in Java SE 5, the introduction of enumerations resulted in *enum* becoming a reserved word, so not valid as a variable name.
- **In the unlikely event that you've used a single underscore as a variable name, you'll need to change that.** As of JDK 9 this is no longer allowed.
- **Prior to JDK 9, many application programming interface (API) elements (classes, interfaces, methods, and so on) had been deprecated for removal, but none had ever been removed.** In JDK 11, many legacy APIs covering functionality like CORBA, JAX-B, and JAX-WS were removed. Other JDK versions have removed a few little-used APIs. You can download these libraries separately and include them via changes to your build scripts.

Even if the Java version is *not* the same, the code is still probably okay. The Java platform maintains excellent backward compatibility across versions. As a result, code written for one version of Java will almost always compile and run on a newer version.

Comparing OpenJDK Distributions

Figure 4-1 shows some of the most popular OpenJDK distributions and how they compare from the perspective of the things we discuss in this chapter.

A full circle means that the feature/capabilities are covered 100 percent in that distribution. An empty circle means that the feature/capabilities are not covered at all. A half- or quarter-circle indicates partial coverage. For example, Azul, Red Hat, and Eclipse Temurin offer IcedTea-Web as an alternative to Java Web Start, so they get 50 percent credit for the applets and Java Web Start feature. Oracle gets 100 percent credit, even though it has

discontinued these features, because older builds of Oracle JDK with these features work just fine. Note that although current feature and LTS releases provided by all five distributions are TCK tested, there are no longer TCK tests available for Java 6 and 7, because these have reached "end of life" from an Oracle support perspective.

Features and Characteristics	Azul	Oracle	Corretto	Red Hat	Eclipse Temurin
Based on OpenJDK	●	●	●	●	●
100% Open Source, Freely Available (no field of use restrictions)	●	○	●	●	●
TCK Tested (guaranteed Java SE compliance)	●	●	●	●	●
Patent Grant (inherited patent rights to use the JDK)	●	●	●	●	●
Free Quarterly Updates (single builds combining security & enhancements)	●	◑	●	●	●
Performance Parity with Oracle Java SE	●	●	●	●	●
Multi-platform (Windows, Linux, macOS, Solaris)	●	●	◕	◑	●
Native Alpine Linux (musl libc)	●	○	●	○	●
Multiple Installers & Packages (.tar, deb, MSI, DMG, JDK/JRE)	●	◑	◑	◑	●
Java Flight Recorder and Mission Control (for Java 8)	●	●	●	●	●
OpenFX (JavaFX)	●	●	◐	○	○
LTS (Java 8, 11, 17 …) and non-LTS versions	●	●	●	◑	●
Java 7 Extended Support	●	○	○	○	○
Java 6 Extended Support	●	○	○	○	○
32-bit Support	●	●	○	●	◔
Java Web Start and Applets	◔	●	○	○	○

FIGURE 4-1: An OpenJDK Distribution Comparison Matrix.

TIP

For a full discussion of the features and capabilities compared in this matrix and a comprehensive listing of OpenJDK distributions, see `www.openjdk-migration.com/openjdk-distributions`.

IN THIS CHAPTER

- Securing Java apps by applying quarterly updates
- Backporting security fixes
- Understanding stabilized security builds
- Maintaining bundled technologies
- Reporting bugs and receiving fixes
- Protecting your code from GPL contamination
- Getting help during your migration
- Previewing new technologies

Chapter 5
Exploring the Benefits of Commercial Support

One key distinction between the various OpenJDK providers is whether they provide commercial support — and if they do, the nature and extent of that support.

The OpenJDK project doesn't provide any formal support. Users can report any bugs through the Oracle Java Bug Database. Still, there are no guarantees or service-level agreements (SLAs) that specify when (or even if) a bug will be addressed and potentially fixed. Even if someone else has reported a bug and it has been fixed, it may be necessary to wait up to three months for the next scheduled update to include the fix. This won't be acceptable if the bug means that mission-critical applications are unable to deliver critical functionality.

Bug reporting — and timely bug fixing — are benefits of commercial support for OpenJDK. But the services of the leading

enterprise providers of OpenJDK support extend far beyond typical help-desk support.

In this chapter, you see how providers like Azul enable you to scale your Java team by assuring the stability and security of your Java Development Kit (JDK), maintaining bundled technologies, protecting your intellectual property (IP), assisting with migration planning, and delivering innovation directly to your doorstep.

Applying Quarterly Updates

The OpenJDK project releases updates four times a year, on the third Tuesday of January, April, July, and October. Changes are applied directly to the current version of Java at that time as well as to the most current Long-Term Support (LTS) version. This means that anyone can download the OpenJDK source code and build their own updated JDK, and it will always be current with security patches, bug fixes, and enhancements.

The problem is that building your own JDK isn't feasible for most organizations. It's a little like assembling a working car from parts, assuming all the parts are digital but require the same level of know-how about mechanical engineering, electrical engineering, physics, materials science, safety regulations, and testing. For a good portion of the 2010s, organizations that ran Java apps relied on Oracle for their JDKs — unless they needed a very high-performing JDK, in which case they turned to a provider like Azul.

This started to change with the release of OpenJDK 9 in September 2017, when Oracle introduced the concept of a long-term release on a fixed schedule.

Until OpenJDK 9, the gap between Java releases varied. JDK 6 shipped in 2006, followed by JDK 7 in 2011 and JDK 8 in 2014.

Feature releases would receive two updates before the next feature release. Oracle LTS releases would receive updates for five years with an additional three years of passive support (no scheduled updates). The feature releases were targeted at developers so they can leverage new features in production as soon as possible. The LTS releases were for enterprises that preferred stability so that they could run multiple large applications on a single shared Java release.

The new release cadence and the new distinction between feature releases and LTS releases paved the way for Oracle to announce a paid support offering in the form of an Oracle Java SE subscription the following year.

Several license and pricing changes later, Oracle settled into the cadence of free updates noted earlier. The current version of Java (the feature release) and the long-term release were updated on a quarterly basis. Organizations who were using older versions of Java could pay Oracle for support, but many also began turning to alternative providers who supported more versions on more platforms and who moved more quickly to new architectures, such as Apple silicon ARM-based Macs and Amazon Web Services (AWS) Graviton2 and Graviton3.


TIP

Getting the support you need to remain on older or less common versions of Java, or on the newest processor architectures, is one reason enterprises turn to commercial support.

Protecting Older JVMs

Following the introduction of LTS releases and the Java SE subscription, organizations quickly learned that getting the highest levels of reliability from their Java runtimes required more than free updates. They needed timely access to updates on a continuous basis — and they needed those updates backported to older versions. This was important because a newly discovered vulnerability that got fixed in a current feature release probably also existed in many older Java versions, too.


WARNING

Vulnerabilities affecting older Java releases continue to be reported. In 2022, vulnerabilities were reported as far back as Java 6, as well as in Java 7 and 8. However, Oracle stopped premier support for Java 7 in July 2019 and ended extended support in July 2022. In March 2022, Oracle ended premier support for Java 8, once the most popular release, and still in use in 33 percent of applications in production (according to New Relic's *2023 State of the Java Ecosystem,* published in April 2023).

Java's built-in security measures like strong data typing make it naturally more robust than some other languages, so it can be tempting to take the security of your JDK for granted. But

you'd be running an unnecessary risk to do so. Since Oracle stopped providing free updates in 2019, there have been 139 security vulnerabilities fixed in OpenJDK 8, including 21 classified as critical or high severity by the Common Vulnerability Scoring System (CVSS), published in the National Vulnerability Database (`https://nvd.nist.gov`). Figures 5-1 through 5-3 show the frequency of new common vulnerabilities and exposures (CVEs) in JDK 6, 7, and 8 by severity score and update. The size of the circles corresponds to the number of vulnerabilities with that severity score in that release.

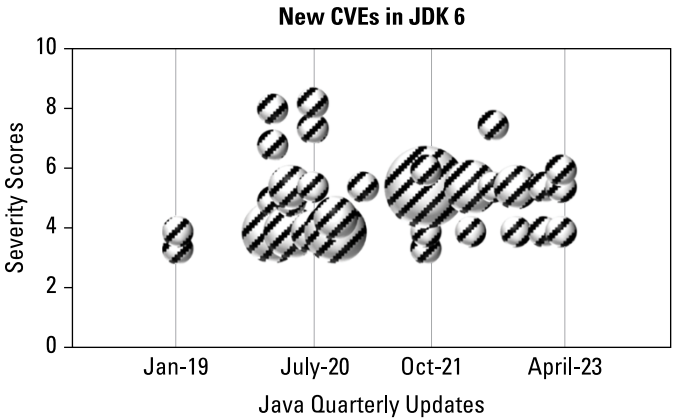


FIGURE 5-1: New CVEs in JDK 6 from 2019 to 2023.

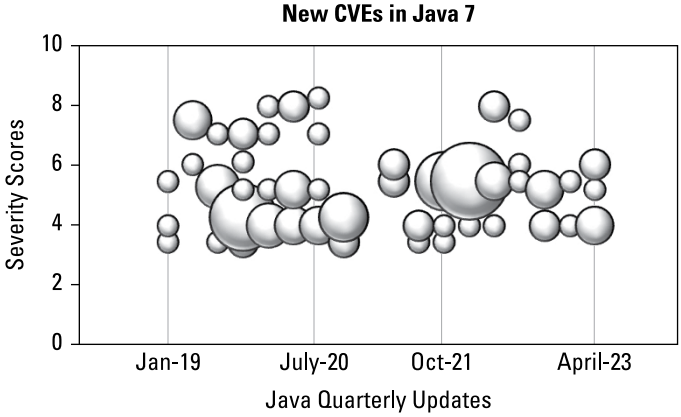


FIGURE 5-2: New CVEs in JDK 7 from 2019 to 2023.

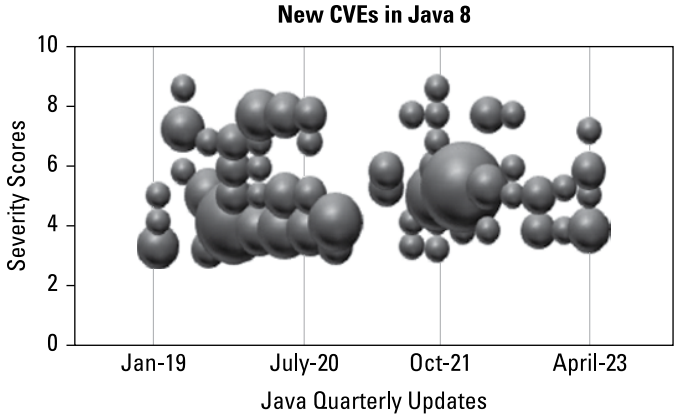


FIGURE 5-3: New CVEs in JDK 8 from 2019 to 2023.

TIP

Receiving backported security fixes is another reason enterprises turn to commercial support.

Reducing Risk with Stabilized Security Builds

Receiving backported security fixes from a commercial provider is important, and how those fixes are delivered is equally important.

Quarterly updates typically contain several hundred changes. The majority are bug fixes and minor enhancements, with security patches typically numbering less than 20 per release. In the last ten years, the largest number of security patches in an update was 37. This means that the risk of a security patch impacting the stability of an application is much lower than the risk from a non-security-related change.

For this reason, Oracle and Azul offer two versions of each update:

- **Critical Patch Update (CPU):** This update contains only the changes related to security patches.
- **Patch Set Update (PSU):** This update includes all the changes, security, bug fixes, performance enhancements, and everything else.

WHY CPUs MATTER

An example of how CPUs can protect an organization from a regression is a bug fix that was delivered in update 252 of JDK 8 in July 2020. The fix had unintended consequences: It prevented certain popular applications, like Hadoop Cluster and Solr, from running.

The security-only update, technically update 251, was delivered on the PSU from the previous quarter. It didn't contain the breaking change. Therefore, the security-only update could safely be deployed into production without compromising application stability. This situation clearly illustrates how organizations benefit when both format updates are available.

Since the end of free updates to Oracle JDK 8, there has been at least a 25 percent chance that a PSU would experience a regression. Not only can regressions affect application uptime, but rolling back an update can expose an organization to known vulnerabilities and create compliance risk.

TIP

Securing your JDK without impacting application stability can be an important benefit of commercial support.

Updating Bundled Technologies

Just before the annual JavaOne conference in 2007, the management at Sun Microsystems was looking around for something to include as a keynote-worthy announcement. On very short notice, the decision was made to announce a new graphical application development platform that used the Java runtime. This project, dubbed F3 (for "Form follows function") was created primarily by a single employee, Chris Oliver, and quickly rebranded as JavaFX.

Although JavaFX ran on the Java Virtual Machine (JVM), it was developed using a non-Java language that looked a lot like JavaScript (but was subtly different). This proved too hard to sell to Java developers, so two years later (again at JavaOne), JavaFX 2 was announced. This was a complete rewrite of the platform as a set of Java class libraries. Over time, Oracle added more functionality to JavaFX, including features like 3D rendering.

Although JavaFX was not included in the OpenJDK main project, Oracle bundled it with the Oracle JDK for Java 8, 9, and 10. Companies that built applications on these distributions need more than commercial support for older Java — they need formal support for JavaFX.

If a bug affects an application that relies on JavaFX, this can be reported and fixed in the same way as the JDK. (See Chapter 2 for a more detailed explanation about migration concerns for JavaFX.)

REMEMBER

Extending the life of bundled technologies is another reason to choose commercial support.

Encountering New Bugs

Many millions of Java users work with hundreds of thousands of applications, including some of the most demanding on the internet (such as credit card fraud detection for Mastercard and movie streaming on Netflix). Over nearly three decades of use, thousands of bugs have been reported and fixed in Java. This process continues as Java continues to evolve.

Although the JDK is updated regularly, there may still be situations where an application encounters a bug in it that has not been reported or fixed. What a user can do at this point will depend on which distribution they're using.

Free distributions do not offer any formal support channel. In this case, your options are limited to reporting the problem details through the Oracle Java Bug Database. And even after submitting a bug, there is no guarantee it will be resolved in the next update (or ever).

For commercially supported distributions of OpenJDK, a formal support channel will be available to receive reports of issues 24/7/365 via a web form or email. (Some support organizations may also include phone access.) Each case will be assigned an identifier, often referred to as a *support ticket.* Many support contracts will include an SLA specifying how quickly the support team will respond to a ticket. This may be as little as an hour, so customers will know that for mission-critical applications, problems will get attention as quickly as possible.

The best support teams will provide deep root cause analysis. They'll go beyond the immediate symptoms of a problem and investigate the underlying causes of problems or issues. This approach can save time and resources by eliminating the need for multiple fixes. When the solution to the problem is upstreamed to the reference implementation, everyone who uses that version of Java benefits.

Of course, the customers of the support team that developed the fix won't have to wait until the fix is included in an OpenJDK update. They can receive the fix directly from their support provider. Likewise, the customer can expect to receive major out-of-cycle bug fixes when they're critically necessary.

REMEMBER

Getting expert help on a firm SLA, out-of-cycle bug fixes, and more is a common reason to get commercial support.

Understanding and Addressing GPL Contamination

Open source has become a fundamental part of how the software development community operates. Freely reusing source code from other developers, rather than having to continually reinvent the wheel and write code that has been written many times before, has had a profound impact on software reliability and developer productivity.

When using open-source software, it's essential to understand the terms of the original author's license. However, doing so can become more complex and more fraught with potential problems the more aggregation of code that occurs.

The primary license used for OpenJDK distributions or runtimes is the GNU Public License version 2 with Classpath Exception (GPLv2+CPE). However, some of the code contributed to the project uses other licenses, such as Apache and Berkeley Software Distribution (BSD). For example, parts of the Extensible Markup Language (XML) processing libraries use the Apache license.

Not all open-source licenses are the same; some are more permissive than others. GPLv2 imposes quite strict requirements on how code can be reused. The most significant of these requirements

is the copyleft nature of the license. Copyleft grants the right to freely distribute and modify the source code but mandates that the rights granted by the GPL must be preserved in any derivative works using the code. This is often referred to as a *viral license*, because it forces anyone using code licensed under the GPL to license their code in the same way.

If OpenJDK distributions used only the GPLv2, it would force developers to make their code available when they don't necessarily want to.

Consider the following scenario: Let's say you decide to ship your application bundled with an OpenJDK runtime. Doing this removes ambiguity and can increase reliability by making your code easier to test. You know precisely which runtime will be used and can test against that. However, by bundling your application with OpenJDK runtimes, you've created a derivative work, and the GPL now applies to your application. Anyone can now request a copy of the source code your team spent so much time and effort developing! OpenJDK would lose its appeal to many users if this were the case.

To remedy this situation, the OpenJDK license includes the Classpath Exception (CPE). Any application code on either the class path or module path is not affected by the copyleft nature of the GPLv2.

Some people who look at this assume that all GPLv2 licensed code in OpenJDK includes the CPE, but this is not the case. For example, if you search the HotSpot (JVM) source code directory of more than 3,000 files, only 5 include the CPE. Conversely, looking at the `java.base` library source, just over 50 of the 3,000-plus files don't have the CPE. This is because the CPE is only required when application code touches the JDK code directly.

For commercial OpenJDK distributions, it's essential that all files that need the CPE have the CPE. If even one file that requires the CPE doesn't contain it, and your application or one of your included libraries happens to make use of an application programming interface (API) in that file, the licensing implication for your applications can be as bad as having no CPE at all for the whole JDK.

Some commercial distributions of OpenJDK perform careful scanning of the source code used to build them (both static and dynamically created) and will guarantee no exposure to the copyleft nature of the GPLv2. As a commercial feature, distributions may go even further and provide indemnification against claims through GPL contamination.

REMEMBER

Building on open source without the risk of copyleft contamination of your code is a valuable benefit of commercial support.

Planning with Expert Guidance

You may be surprised that in a book about migrating to OpenJDK, migration assistance shows up toward the *end* of a list of the benefits of commercial support for OpenJDK. For most providers, a migration support ticket is no different from any other support ticket. It will be handled according to the support tier you signed up for.

At Azul, we've found that what's most helpful to our customers is assistance in planning their migration. Migrating itself is usually straightforward, but a good plan is still essential, particularly for companies that have been using Java applications for years on the desktop and on servers and, as a result, run on a variety of JDK versions.

TIP

The Azul Migration Workshop helps customers capture the scope of their migration, as well as the resources they need and the resources they have available. This enables them to build a realistic timeline and ensures they don't miss any of the critical steps we've discussed here, such as taking a thorough inventory of their JDKs using IT asset management (ITAM) tools in combination with an application owner questionnaire. The workshop is customized so that each customer ends up with a comprehensive migration plan that is ready to execute. (Learn more at `www.openjdk-migration.com/azul-migration-workshop`.)

REMEMBER

Assistance with migration planning is another example of the value added by some commercial support providers.

Leveraging Support for a Competitive Advantage

You could argue that assuring your JDK is patched against known vulnerabilities is the first job of an OpenJDK support provider, followed closely by offering stabilized builds to lower the risk of regression. But providers differ in how they deliver against these goals. In addition, some providers offer additional benefits — such as support for bundled technologies, SLA-driven responses to bug reports, and IP indemnification. But what can really set a provider of OpenJDK apart is its role in driving Java innovation and providing its customers with previews of new technologies that can offer a competitive advantage.

Major enhancements to OpenJDK around security and start-up/warm-up time can have a long incubation time as community projects before they're included in a feature release or an LTS release. One example of this is the OpenJDK project Coordinated Restore at Checkpoint (CRaC).

The CRaC Project defines public Java APIs that allow for the coordination of resources during checkpoint and restore operations. With CRaC, you can set a checkpoint at any point where an application can be safely paused. You can then use this checkpoint to reduce start-up and warm-up times by orders of magnitude. After about three years of community incubation, the first and only commercially supported CRaC-configured build was released by Azul in the form of a Java 17 Linux/x64 update (JDK version 17.0.7+7). As part of that update, developers using this build can deploy applications using CRaC into production with confidence that updates will be provided by Azul.

REMEMBER

Deploying new technologies to production with full commercial support is an important advantage of working with a commercial support provider.

IN THIS CHAPTER

- Evaluating a potential partner's track record
- Getting references from other customers
- Selecting an appropriate service level

Chapter 6
Choosing the Right Java Partner

Once an organization interested in migrating to OpenJDK fully understands the three-step migration process and the straightforward path to migration success, another concern can rise to the fore.

Enterprises want to know if they can trust their business-critical Java applications to the care of a new provider. Java's rapid evolution has made it the language of choice for enterprise applications. It has also created a need for extensive knowledge and experience with Java that can be hard to find.

Creating a Java application is comparatively easy, but deep root cause analysis of issues affecting the JDK requires sophisticated knowledge of a highly complex, ever-changing system made up of seven million lines of code (and growing). Architectural changes, like the rise of microservices and serverless functions and containers, have created new performance challenges, too, and the move to the cloud has led to escalating costs — some of which can be reduced by faster code. Organizations that run on Java or maintain a significant Java footprint need more than a JDK help

desk; they need a true Java partner. And, let's face it, looking for a partner for any endeavor can be like going off on a quest for the holy grail. Luckily, there are signposts that point to a provider's level of proficiency. We point these out in this chapter.

Evaluating a Track Record

Java is nearly 30 years old. Providers with real expertise have a track record of supporting the Java platform — and participating in the processes and standards bodies that move Java forward — extending back to the first complete release of OpenJDK's source code in 2009. (You can read more about the history of OpenJDK in Appendix A.)

A key organization is the Java Community Process (JCP). Originally created by Sun Microsystems in 1998, the JCP develops and maintains Java technology standards. It was designed as an open, inclusive, and collaborative process consisting of working groups, expert groups, and an executive committee. The JCP's executive committee is responsible for approving Java Specification Requests (JSRs) and reconciling discrepancies between specifications and test suites.

Organizations and individuals who serve on the JCP's executive committee are actively involved in determining Java's future. There are also a handful of individuals and organizations whose continuous involvement with Java over the years has created a knowledge base that is rich and deep. Over time, they've ensured that Java remained relevant even as software went through a step change and client–server architectures were supplanted by cloud-based architectures utilizing microservices. Gil Tene, Azul's cofounder and chief technology officer (CTO), was first elected to the JCP in 2011. He is one of the longest-serving members and has been recognized as Member of the Year.

Other activities of note include participating in the OpenJDK Vulnerability Group and actively contributing security fixes. Active involvement in online communities like Friends of OpenJDK (Foojay) and the Adoptium Working Group can serve as further proof points.

Considering Customer References

A potential partner's list of existing customers is a good place to start to find out if that organization is a fit for your own. It will help you determine the level of expertise they have in your particular industry and market. In addition, understanding the size and scope of existing support relationships will offer insight into whether a potential partner can handle the complexity of your own organizational needs.

Customer satisfaction is another component to pay attention to. It's a leading indicator of a partner's current level of service, and it can be a helpful macro data point when combined with testimonials and direct references.

Deciding on a Service Level

The services offered by commercial support providers for OpenJDK can vary quite a bit. Differences start with the breadth of Java versions and the associated platforms and underlying architectures that are supported, as well as the length of support. It's also important to understand how a provider defines *support* and whether those services are covered by a service-level agreement (SLA).

REMEMBER

Here are some differences in the delivery of OpenJDK support that you'll want to consider:

- Basic help-desk support versus deep root-cause analysis, dedicated account managers, monthly strategy calls, best practice consulting, and quarterly security briefings.
- Quarterly updates delivered as a Patch Set Update (PSU) versus curated stabilized builds (which is what Oracle and Azul refer to as Critical Patch Updates [CPUs]).

- Updates that rely on "best effort" versus updates delivered on a strict SLA.
- OpenJDK builds with known issues around GNU Public License (GPL) contamination versus builds that are indemnified.

These differences add up to fundamentally different views of the relationship between the OpenJDK provider and customers who run their business on OpenJDK. A true partnership will focus on achieving the most business value from Java, starting with your migration and continuing through the life cycle of your Java Virtual Machines (JVMs).

IN THIS CHAPTER

- Asking the right questions
- Understanding how the answers will affect your service

Chapter 7
Ten Questions for Your Next Request for Information

There are a lot of Java partners out there who want your business. How can you drill down in the stack to find the best one for your needs? It helps to know what questions to ask potential vendors in your requests for information (RFIs). The answers you receive will affect the quality of service you receive. Here are ten questions that can help you make the right decision:

- **Are your binaries Technology Compatibility Kit (TCK) tested?** TCK tests are the suite of tests that ensure that distributions of each version of Java are compatible with each other. This testing drastically cuts down on incompatibility issues.
- **What versions of Java do you support, and for how long?** Even if you have a good idea of which versions of Java are running in your organization, working with a vendor who supports more versions will mean you're prepared if you're surprised by the results from your application inventory.

- **What operating systems and architectures do you support?** In addition to the operating systems and chip sets currently in use in your organization, you'll want a provider who supports operating systems and architectures that you may want to move to in the future.
- **Do you provide quarterly security updates on stabilized builds with a service-level agreement (SLA)?** Organizations can minimize the risk of an expensive regression by updating to stabilized builds each quarter. These builds are binaries released the previous quarter as Patch Set Updates (PSUs). They've been used in production worldwide for three months. Also known as Critical Patch Updates (CPUs), stabilized builds provide security-only fixes — patches for new vulnerabilities reported and fixed during the most recent quarter. These ensure that your Java applications are secure and compliant with internal policies and, depending on your industry, external regulations.
- **Do you backport fixes to security issues in later releases to all supported versions on an SLA?** It's common for large organizations to have departments running on older versions of Java. To remain compliant with internal policies and external regulations, they need a vendor who backports patches for newly reported vulnerabilities. For example, a patch in Long-Term Support (LTS) 17, the current LTS release, may need to be backported to Java Development Kit (JDK) 8, an earlier LTS release that's reaching the end of its life.
- **What is your track record for releasing binaries immediately following the OpenJDK Vulnerability Group lifting the embargo on quarterly updates?** Even leading providers of OpenJDK can find themselves having to delay the release of new quarterly binaries by several days or even sometimes more than a week, often leaving deployments vulnerable to severe vulnerability exploits during the delay window. If the timely release of binaries matters to you, look for a provider with a track record for releasing binaries in under an hour.
- **Will you provide out-of-cycle updates for critical common vulnerabilities and exposures (CVEs)?** Vulnerabilities with the highest scores in the Common Vulnerability Scoring System (that is, those described as "critical") must be patched right away and may require an out-of-cycle fix. Otherwise, your organization could be exposed for weeks or months.

Providers committed to providing critical out-of-cycle fixes will keep your Java applications secure and ensure you stay in compliance.

- **Do you support optional components such as JavaFX?** Over the years, more than 78 vulnerabilities have been reported that affect JavaFX. If you have applications that use components like JavaFX that are not included in OpenJDK, you need a JDK provider who will support those components with JavaFX-configured builds for your JDKs and Java Runtime Environments (JREs).
- **Do you provide indemnification in case of patent litigation?** Patent indemnification protects software users against patent infringement claims. It means that your JDK provider will cover legal costs or damages if a third-party claims you're infringing on their patents through your use of their JDK.
- **Do you provide indemnification against GNU Public License (GPL) contamination?** Similar to patent indemnification, indemnification can protect you in the case of GPL contamination that can result from copyleft licensing. With copyleft licensing, anyone who modifies or distributes the software must make their modification or distribution available under the same license. To prevent this, users of OpenJDK must rely on their provider to protect them by using the Classpath Exception (CPE). Providers who provide indemnification are offering to stand behind their distributions and protect their users from copyleft claims.


TIP

You can find a full-fledged sample request for proposal (RFP) in the online toolkit at `www.openjdk-migration.com/openjdk-support-sample-rfp`.

Appendix A
A Brief History of Java

The origins of the Java platform can be traced back to the early '90s, when internet users could be counted in the thousands and the first web browser had not yet been publicly released.

Sun Microsystems, a company famous for UNIX workstations, allowed a small engineering team to go off and figure out what would be the next wave of computing. They designed a device that was similar in concept to today's Apple iPad called the *7 (Star Seven). Unfortunately, this was when processor speeds, memory costs, and display technologies were not up to the task, so the device never saw the light of day. Part of the design, however, required a new programming language and runtime platform that would enable applications to be easily moved across a network and run without worrying about what hardware or operating system was in use. The language and interpreter, created by James Gosling, was initially called Oak (after the tree outside his office window).

After unsuccessful attempts at promoting Oak for use in set-top boxes and interactive TV, the project was on the verge of being canceled. Then, in February 1995, John Gage, the director of the Science Office at Sun, gave a TED Talk that culminated in showing Oak being used within a web browser. The demonstration showed a picture of a molecule that could be dragged, rotated, and

manipulated using the mouse. This was revolutionary! Suddenly web browsing, which had been like an electronic book, became interactive. The possibilities were limitless.

After rebranding due to trademark issues and negotiations with the fledgling browser company Netscape, Java was officially made public on May 23, 1995, with the first official release in January 1996. Two years later, in 1998, Sun released the first open specification.

Developers and users wanted Sun to go further with ceding control and make Java open source. But Sun continued to resist calls to release the source code to their Java Development Kit (JDK). This led some developers to work on an alternative implementation of Java that would be available under an open-source license. In 2005, the Apache Foundation (a nonprofit corporation that hosts multiple open-source projects) started Project Harmony. This used the code from the GNU Classpath project, a project to develop a complete set of standard Java class libraries, and added an implementation of the Java Virtual Machine (JVM).

Sun could see that eventually there would be an open-source version of Java, so it would make more sense for it to be theirs, over which they could also maintain control.

At the annual JavaOne conference in 2006, Sun announced they were releasing the source code of their implementation of the JDK through a project called OpenJDK. As a result of the due diligence required to ensure Sun had the necessary rights to publish source code, it was not until April 2009 that a complete JDK could be built from the OpenJDK source (this was OpenJDK 7 build 39).

The initial version of Java provided through the OpenJDK was JDK 7. A build of JDK 6 was made possible by creating a backward branch of OpenJDK 7, essentially removing what had been added to develop JDK 7.

The license chosen for the OpenJDK was the GNU Public License (GPL) version 2. The Classpath Exception was added to avoid application code having also to use the GPL.

Oracle, best known for its database products, agreed in April 2009 to acquire Sun Microsystems. The deal closed in February 2010, and Java had a new home.

Appendix B
Optimizing the JVM for Lower Latency, Higher Throughput, and Faster Warm-up

The Java Virtual Machine (JVM) is very powerful, providing an ideal managed runtime environment for robust, scalable, and secure enterprise applications. However, although the JVM is a great choice for reliably running some of the world's most demanding workloads, the standard OpenJDK JVM is not always a perfect choice.

During the last three decades, research on improving the JVM has focused on four areas: reducing latency, increasing throughput, achieving faster start-up/warm-up, and optimizing cloud workloads. There has been steady progress from release to release, and we've seen notable innovations over the years, summarized in this appendix.

Minimizing Latency with Garbage Collection

One of the top benefits of Java is automatic memory management. Memory management is a task that all programming languages must handle, and it's also one of the most fundamental and error-prone aspects of software development.

Here's how it commonly worked before Java. As a computer program ran, it allocated memory to store data assigned to particular variables. Then, at a certain point, that data was no longer used by the program. In languages like C++, programmers wrote (and continue to write) code that explicitly released this memory to ensure that there was enough memory available. In doing so, they prevented memory leaks — and potentially disastrous crashes — as well as dangling pointers and other memory-related bugs.

Concurrent Mark Sweep garbage collection


REMEMBER

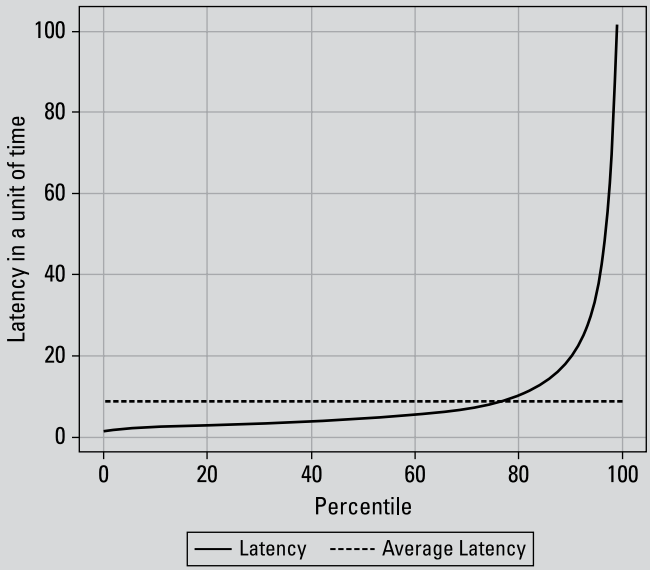
With Java, memory is released automatically through a process known as *garbage collection.* Java's approach to memory management increases developer productivity and enhances the robustness and reliability of Java applications.

But there are trade-offs. Garbage collection algorithms that support high throughput can increase latency. Conversely, an algorithm that supports lower latency can affect throughput. Algorithms that are highly optimized for throughput or latency can increase an application's memory footprint.

The first garbage collection algorithms required applications to pause as they did their work. This was necessary to ensure the safety of data, which could otherwise be left in an inconsistent state. Stop-the-world pauses could be very short and not noticeable to users, but some were long enough to introduce latency outliers and significant enough that applications failed to meet their requirements.

UNDERSTANDING LONG-TAIL LATENCY

A challenge operations teams sometimes face in understanding latency is that the effect of averaging latency over time can be quite small. This is because extreme latency effects are "long-tail" effects. They occur less frequently than average latency effects but can still be devastating to a business. In the following figure, the *y*-axis represents an arbitrary unit of time correlating to an application's latency. The *x*-axis is the frequency of the latency effect for that unit of time by percentile. The dotted line is average latency for the entire application run. The solid line shows that latency is below average for more than 70 percent of the time that an application is running, but then latency spikes sharply in the upper 90th percentile.



To address this, the Concurrent Mark Sweep (CMS) garbage collector was introduced in 2001 as part of HotSpot JVM (JDK 1.4). CMS sought to minimize pause times by performing garbage collection concurrently with an application's execution. By using multiple garbage collection threads, CMS aimed to reduce stop-the-world pauses.

G1 Collector

JDK 9 replaced CMS with the G1 (Garbage-First) collector. G1 had several advantages, including its ability to:

- Deliver more predictable pause times by dividing the *heap* (the region of memory used for dynamic memory allocation) into smaller regions.
- Improve overall throughput by dynamically adjusting the duration of garbage collection cycles.
- Promote more efficient heap utilization by compacting regions concurrently.
- Support scalability with multithreading and parallelism.
- Reduce administrative overhead.

Shenandoah and Z Garbage Collector

Despite these improvements, however, latency remained a problem. Red Hat's Shenandoah and Oracle's Z Garbage Collector (ZGC) are alternatives to G1 that use additional concurrent and pauseless technologies to reduce pause times, even for applications with very large heaps. In addition to concurrent compaction, multithreading, and parallelism, they provide flexibility in heap sizing and use techniques such as load barriers, reference processing, and adaptive heuristics.

C4

Azul Platform Prime's C4 Collector (for Continuously Concurrent Compacting Collector) is another advanced garbage collector. There are similarities between C4 and Shenandoah and ZGC, but C4 is designed to provide low-latency operations with high throughput while managing large heap sizes. Its emphasis on achieving pauseless operations leads to consistent low-latency behavior with no noticeable pauses even as applications scale to very large heap sizes — like multiple terabytes per JVM.

C4 uses a read barrier to support concurrent compaction, concurrent remapping, and concurrent incremental update tracing. With C4, enterprises can run two to five times more transactions through their infrastructure without pauses, jitters, or timeouts. From an operational perspective, engineers and developers spend less time tuning applications.

Increasing Throughput

Garbage collection algorithms can influence throughput. But other factors that affect throughput in Java applications have been the focus of sustained attention.

The problem of interpreting

In Java, source code is compiled into bytecode, which the JVM interprets at runtime. This system offers multiple advantages:

- It enables Java applications to be platform independent.
- It enables dynamic class loading at runtime.
- It provides higher degrees of security because the JVM is executing bytecode in a controlled environment.

However, interpreting bytecode — and then compiling it — is resource-intensive and slow. Interpreting at runtime can create a bottleneck that reduces throughput, particularly in compute-intensive tasks or tight loops.

The JIT compiler

REMEMBER

One way Java compensates for the slowness of the interpreter is with just in time (JIT) compilation, which produces faster code. The JIT compiler identifies frequently executed bytecode routines and compiles them into native machine code on the fly.

In OpenJDK, the JIT compiler is based on the HotSpot JVM (first released with JDK 1.3). Although the HotSpot compiler has been continuously improved and updated, companies like Azul, IBM, and Oracle have introduced even more highly optimized compilers.

Techniques like speculative optimization leverage profile information gathered at runtime to guide the JIT compiler. Statistical data identifies hot spots, optimizes frequently executed code paths, and eliminates unnecessary or infrequently used code branches.

Azul's Falcon JIT Compiler

Azul's Falcon JIT Compiler, a feature of Azul Platform Prime, uses advanced compilation techniques and optimizations to generate

highly optimized native machine code, as well as specific optimizations that target applications running on Azul's Platform Prime.

Among the features that set Falcon apart are the compiler's memory optimization techniques. For example, Native Memory Tracking (NMT) tools provide extended information on how the JVM and the compiler use native memory.

As of Stable Release 23.02, an ergonomics heuristic based on the number of Falcon compiler threaders control when Falcon resets internal caches. Armed with information on native memory usage, engineers can also explicitly adjust cash reset behavior with the `FalconContextReset` flag. The more often caches are reset, the less memory is consumed.

Reducing Application Warm-up Times

Using advanced compilers and garbage collection algorithms can lead to a dramatic performance boost. Recent studies have found that enterprises that invest in Java optimization can expect to see two times the throughput on the same infrastructure and eliminate stop-the-world pauses.

Additional optimizations can also solve Java's well-known start-up and warm-up issues. These issues are one of the biggest challenges of using Java: Getting Java programs to run fast — by optimizing for high throughput and low latency — can take some time.

First, Java applications need to start up. This involves activities like class loading, initializing static variables, and setting up the runtime environment. After start-up, an application is ready to run, but it isn't optimized for peak performance. That happens during warm-up.

During warm-up, a program runs with typical or representative workloads while the JIT compiler gathers profiling data. By collecting and analyzing the profiling information, the JIT compiler can dynamically recompile and optimize the code to improve execution speed and reduce interpretation overhead, as well as apply various other optimizations.

Warm-up can take seconds to minutes or even longer. Factors that influence these times include the size and complexity of an application and a workload, the performance of the underlying hardware, and the efficiency of the JVM and its optimization strategies.

There are a few approaches you can take to shorten start-up and warm-up times.

Ahead of time compilation

Ahead of time (AOT) compilation involves converting Java instructions into native machine code ahead of execution so the application can be run as a stand-alone binary. Oracle's GraalVM native image solution utilizes AOT.

AOT significantly reduces start-up time and eliminates warm-up time. It can also help reduce an application's memory footprint. It's useful for applications that start up frequently or have strict performance and memory requirements.

But AOT has several significant drawbacks. AOT-compiled applications can't be optimized at runtime and will not (generally) benefit from the dynamism of the JDK. Perhaps most important, AOT-compiled applications can also be difficult to debug. Because AOT compilation is done in the development environment, which can differ significantly from the production environment, the ability to easily debug issues can be a nonstarter.

ReadyNow!

An alternative to AOT is to persist the profiling information gathered by the compiler so it doesn't need to start from scratch with each subsequent run. With ReadyNow!, a feature of Azul Platform Prime, running an application generates a profile log that is consumed and updated with subsequent runs. Warm-up improves each application run until optimal performance is reached.

This approach ensures consistent, peak performance and is great for applications that must meet specific levels of service such as financial trading systems. It's far superior to other workarounds for slow start-up and warm-up like using synthetic data or overprovisioning. Among additional benefits: Developers can get more control over Java compilation and can better manage runtime

de-optimizations, which happen when compiled code doesn't behave as expected and the JVM reverts to reinterpreting and recompiling the code.

ReadyNow! is a great choice for applications that are not required to start instantly and that have sufficient central processing unit (CPU) resources available to handle aggressive optimizations.

ReadyNow! performs two key functions:

- It gives operations teams the ability to save and reuse accumulated optimization profiles across runs.
- It provides a robust set of application programming interfaces (APIs) and compiler directives that give developers more control over the timing and impact of JVM de-optimization.

CRaC

Another way to achieve superfast start-up and warm-up using standard builds of OpenJDK is by using Coordinated Restore at Checkpoint (CRaC), which we discuss in Chapter 5. CRaC is a proposed feature of OpenJDK that allows a running application to pause, take a snapshot of its state, and restart later, including on a different machine or in another time zone.

CRaC is currently available in Azul Zulu Builds of OpenJDK 17 with CRaC support for Linux x86 and ARM architectures (64 bit). Amazon is also using the CRaC APIs in Amazon Web Services (AWS) Lambda SnapStart for Java functions. Frameworks like Micronaut, Quarkus, and Spring Boot also support CRaC or were planning to implement support at the time this book was published.

The main drawback to CRaC is its complexity. Compared to ReadyNow!, CRaC requires a higher level of Java expertise. You must make code changes in libraries, frameworks, and applications to coordinate resource management during checkpoint and restore events. ReadyNow! does not require such code changes.

Optimizing Cloud Workloads

In the last few years, the location where applications are typically deployed has shifted from local to cloud-based hosts. Instead of provisioning physical machines in a data center, users are

switching to using public clouds. The utility-based pricing model *can* be much more cost-efficient — but not always.

Overprovisioning and resource waste

To gain the most benefit from the resource elasticity of a public cloud environment, monolithic applications are divided up into multiple container-based microservices. When a particular service becomes heavily loaded and introduces a performance bottleneck, more service instances can be started to balance the load and eliminate the bottleneck. As the load reduces, instances can be stopped — providing a dynamic architecture exceptionally well suited to minimizing cloud infrastructure costs.

But the scenario in which cloud resources are balanced against load can be challenging to achieve. Applications typically consume resources in spikes, and spikes can be notoriously difficult to predict.

As a result, companies routinely overprovision to ensure they're prepared for unexpected increases in demand. A recent analysis by CAST AI found that, on average, 37 percent of the CPU resources provisioned for cloud-native applications are never used (see CAST AI, "The State of Kubernetes Report: Overprovisioning in Real-Life Containerized Applications," 2011).

And, when surveyed by Forrester Consulting, two out of three IT professionals attributed cloud waste to idle or unused resources and 59 percent to overprovisioning (see "HashiCorp 2022 State of Cloud Strategy Survey," 2022).

Azul's Cloud Native Compiler

Azul's Cloud Native Compiler (CNC) is a centralized service that directly addresses certain aspects of overprovisioning. When bytecodes need to be compiled, they're sent to the CNC along with the necessary profiling data to allow optimization.

This compiling method offers significant benefits for microservices architectures. When an instance of a microservice starts and warms up, the CNC compiles bytecodes for heavily used code and then caches the compiled code. When another instance of the same service is started, the CNC returns the code from its cache without the need for compilation. This reduces the time required for all subsequent invocations.

By moving the compiler's work to the CNC, the JVM no longer must share its resources between the work of the application and compilation. This results in better performance during warm-up and eliminates the need to provision additional resources for a container that will be used only during warm-up.

You can do things like set higher CPU limits for horizontal scaling, and by using the JIT optimizations delivered by CNC, you can get more traffic through each JVM and provision fewer instances overall to meet your demand. You end up with benefits identical to those from highly optimized JVMs in more monolithic environments.

In a nutshell, faster Java code equals lower cloud costs. Instead of limiting utilization of your cloud instances, you can raise your threshold and still meet your performance service-level agreements (SLAs) using the techniques in this appendix.

Appendix C
Runtime Security

We see an increasing number of cyberattacks, whether intended as malicious fun, for financial gain or even being politically motivated. It has never been more critical to ensure that all systems are maintained with the maximum level of security available.

One of the original design goals of the Java platform was to be able to move code securely around a network for execution. As a result, many aspects of the way the Java Virtual Machine (JVM) works, like bytecode verification during class loading, are designed with this in mind.

One common technique for improving security is to scan application source code for vulnerabilities (technically referred to as common vulnerabilities and exposures [CVEs]). Another is to create a software bill of materials (SBOM) that lists all libraries used when building an application, along with the version details.

Some of the most dangerous vulnerabilities are referred to as *zero-day*. When a zero-day vulnerability is discovered, it can immediately be exploited and expose systems to remote code execution. Hackers can easily take control of machines and steal confidential data. In this case, the vulnerability is made public before a fix for it exists.

One example of this vulnerability that affected the Java environment was the Log4Shell exploit of the Log4j library. Log4j is an extremely popular library for generating and recording logging messages. When the vulnerability was reported, almost all Java

users had to identify whether and where they were using Log4j and then update the library with the necessary security patch.

Vulnerabilities in Java's third-party libraries can be particularly challenging because the infrastructure to address them is far less mature than the organizations and processes that ensure the security of OpenJDK. Table C-1 provides examples of high-risk vulnerabilities affecting Java libraries that were recently discovered.

TABLE C-1 Common Vulnerabilities and Exposures in the Java Ecosystem

CVE	Score	Java-Related Component	Vulnerability Type
CVE-2023-28462	9.8	Payara Server	Java Naming and Directory Interface (JNDI) rebind
CVE-2023-32697	9.8	SQLite JDBC	Remote code execution
CVE-2023-26049	5.3	Jetty	Data exfiltration via cookies
CVE-2023-24815	5.3	Vert.x	Classpath file exfiltration
CVE-2023-26919	7.2	Nashorn	Nashorn sandbox escape
CVE-2022-41966	7.5	XStream	Denial of Service (DoS)
CVE-2022-33915	7.0	AWS Log4j Hotpatch	Privilege escalation

Even with an SBOM for applications, locating all instances of an application containing new vulnerabilities in third-party libraries can be difficult and time-consuming if sophisticated software asset management (SAM) software is not in use.

To simplify this, some commercial distributions of OpenJDK can offer ways to quickly identify machines using specific versions of libraries so the update process can be completed as promptly as possible.

Detecting Vulnerabilities in Production

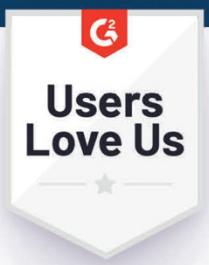
Azul Vulnerability Detection is one such system. Azul Vulnerability Detection uses information already available within the JVM as part of the class loader architecture to track all loaded libraries. This can even extend to providing details about which parts of a library (classes and methods) are in use versus ones that are part of the application but not in an execution path that has so far been followed. This information can be reported to a centralized cloud service. Because the information being used is already in the JVM, this is implemented without the use of a separately installed management agent, thus eliminating any performance impact of this monitoring.

When zero-day vulnerabilities like Log4Shell occur, users can request a real-time picture of exactly which applications running on which machines are using affected versions of Log4j. The system administration team can quickly and efficiently work through the generated list to address the vulnerability and ensure the maximum level of security continues to be provided.

azul

Azul Platform Core (Zulu)

Supporting More Versions, Platforms, and Configurations than You Can Imagine



4.9 out of 5 Stars
G2 Peer Review Rating

Discover why Azul Platform Core is rated 4.9/5 stars for ease of use and support. It's designed for the enterprise with certified builds, cost efficiencies, and security assurance you need to run today's business-critical Java-based applications & infrastructure.

Get 100% open source, fully tested and certified, Java SE standards-compliant builds of OpenJDK.

Check out G2 peer reviews at azul.com/g2-reviews

Migrate from Oracle Java with ease

For the fourth time in four years, Oracle has changed how it prices and/or licenses Java. Many organizations that rely on Java apps are switching from the Oracle JDK to OpenJDK and taking advantage of significant cost savings in the process. But if you have a variety of specialized Java apps, created at different times and running on different JDK versions, you may have some anxiety about making the switch.

This handy book offers a concise, fun-to-read overview of OpenJDK migration, explaining how OpenJDK came to be, what it offers modern organizations, and what steps to take when you're ready to move your apps. You'll also get tips on how to select the right Java partner to assist with your migration and provide ongoing support, security updates, application tuning, cost reductions, and expertise.

Inside...

- Learn about the three-phase strategy for migration success
- Simplify your migration process and avoid pitfalls with older releases
- Find out how to evaluate commercial support providers and reduce your costs
- Get access to free migration tools and templates

azul

Simon Ritter is Azul's Deputy CTO, a Java Champion, and a member of the Java SE JSR Expert Group.

ISBN: 978-1-394-20672-8
Not For Resale

for dummies®
A Wiley Brand

WILEY END USER LICENSE AGREEMENT

Go to www.wiley.com/go/eula to access Wiley’s ebook EULA.