It’s time to move your applications to Java 17. Here’s why. And here’s how.

What you need to know about code migration from the previous Long-Term-Support versions of the platform: Java 11 and Java 8

by Johan Janssen

August 27, 2021

Java 17, the next Long-Term-Support (LTS) version of the Java language and runtime platform, will be officially released on September 14. Unfortunately, many applications still run on old versions of Java, such as the previous LTS versions: Java 11 and Java 8. This article explains why you should upgrade your application and helps you with the actual upgrade to Java 17.

But first, here’s the question many of you may be asking: “Why upgrade?”

Why would anyone even care to upgrade to the latest Java version? It’s reasonable to wonder, especially if your applications run perfectly well on Java 8, Java 11, Java 14, or whatever version you are using. Upgrading to Java 17 requires effort, especially if the goal is to truly leverage the new language features and functionality within the JVM.

Yes, it might require some effort to upgrade depending on the environment and the application. Developers and other team members need to update their local environment. Then the build environments and runtime environments, such as those for production, require an upgrade as well.

Fortunately, many projects and organizations use Docker, which helps a lot in this effort. In my own organization, teams define their own continuous integration/continuous deployment (CI/CD) pipelines, and they run everything in Docker images. Teams can upgrade to the latest Java version by simply specifying that version in their Docker image—and this doesn’t impact other teams who might be running on older Java versions, because those teams can use older Docker images.

The same goes for test and production environments running on Kubernetes. Whenever a team wants to upgrade to a newer Java release, they can change the Docker images themselves and then
deploy everything. (Of course, if you still have shared build environments, or other teams who manage your environments, the process might be a bit more challenging.)

Applications might require some changes as well. I’ve noticed that teams find it challenging to estimate that amount of work, resulting in estimates of weeks to months for upgrading one application from Java 8 to Java 11. Those high estimates often result in the company postponing the upgrade because of other priorities.

I managed to upgrade one application, which was estimated to take several weeks, in only a matter of days, mainly due to waiting for builds to complete. That was partly due to years of upgrade experience, but it’s also a matter of just getting started and trying to fix issues along the way. It’s a nice job for a Friday afternoon; seeing how far you get and what challenges are left makes it easier to estimate the remaining work.

However, even after years of experience, I cannot estimate how long an upgrade will take without having in-depth information about the project. A lot depends on how many dependencies your application has. Often, upgrading your dependencies to the latest version resolves many of the issues that would occur during a Java upgrade.

**LTS releases**

This article keeps referring to Java 8, Java 11, and Java 17 as LTS releases. What does that mean? Here’s a quote from the Oracle Java SE support roadmap:

For product releases after Java SE 8, Oracle will designate a release, every three years, as a Long-Term-Support (LTS) release. Java SE 11 is an LTS release. For the purposes of Oracle Premier Support, non-LTS releases are considered a cumulative set of implementation enhancements of the most recent LTS release. Once a new feature release is made available, any previous non-LTS release will be considered superseded. For example, Java SE 9 was a non-LTS release and immediately superseded by Java SE 10 (also non-LTS), Java SE 10 in turn is immediately superseded by Java SE 11. Java SE 11 however is an LTS release, and therefore Oracle Customers will receive Oracle Premier Support and periodic update releases, even though Java SE 12 was released.

**What needs to change during a Java upgrade?**

Your application contains code you and your team wrote, and it probably contains dependencies also. If something is removed from the JDK, that might break the code, the dependencies, or both. It often helps to make sure those dependencies are up to date to resolve these issues. Sometimes you might have to wait until a framework releases a new version that is compatible with the latest Java version before you begin the upgrade process. This means that you have a good knowledge of the dependencies as part of the preupgrade evaluation process.

Most functionality isn’t removed all at once from the JDK. First, functionality is marked for deprecation. For instance, **Java Architecture for XML Binding (JAXB)** was marked for deprecation in Java 9 before being removed in Java 11. If you continuously update, then you see the
deprecations and you can resolve any use of those features before the functionality is removed. However, if you are jumping straight from Java 8 to Java 17, this feature removal will hit you all at once.

To view the API changes and, for instance, see which methods are removed or added to the String API in a specific Java version, look at The Java Version Almanac, by Marc Hoffmann and Cay Horstmann, or Foojay’s Java Version Almanac.

Multirelease JAR functionality

What if your application is used by customers who still use an old JDK and an upgrade at their site is out of your control? Multirelease JAR functionality, introduced in Java 9 with JEP 238, might be useful because it allows you to package code for multiple Java versions (including versions older than Java 9) inside one JAR file.

As an example, create an Application class (Listing 1) and a Student class (Listing 2) and place them in the folder src/main/java/com/example. The Student class is a class that runs on Java 8.

Listing 1. The Application class

```java
public class Application {
    public static void main(String[] args) {
        Student student = new Student("James ");
        System.out.println("Implementation " + student.implementation());
        System.out.println("Student name James contains ", student.isBlankName());
    }
}
```

Listing 2. The Student class written for Java 8

```java
public class Student {
    final private String firstName;

    public Student(String firstName) {
        this.firstName = firstName;
    }

    boolean isBlankName() {
        return firstName == null || firstName.trim().isEmpty();
    }

    static String implementation() { return "class";
}
```

As an example, create a Student record (Listing 3) that uses not only records (introduced in Java 14) but also the String.isBlank() method (introduced in Java 11), and place it in the folder src/main/java17/com/example.

Listing 3. A Student record using newer Java features

```java
public record Student(String firstName) {
    boolean isBlankName() {
        return firstName == null || firstName.trim().isEmpty();
    }
```

Next to that, create a Student record (Listing 3) that uses not only records (introduced in Java 14) but also the String.isBlank() method (introduced in Java 11), and place it in the folder src/main/java17/com/example.
Some configuration is required depending on the build tool you use. A Maven example can be found in my GitHub repository. The example is built on Java 17 and creates the JAR file. When the JAR file is executed on JDK 17 or newer, the `Student` record is used. When the JAR file is executed on older versions, the `Student` class is used.

This feature is quite useful, for instance, if new APIs offer better performance, because you can use make use of those APIs for customers who have a recent Java version. The same JAR file can be used for customers with an older JDK, without the performance improvements.

Please be aware that all the implementations, in this case, the `Student`, should have the same public API to prevent runtime issues. Unfortunately build tools don't verify the public APIs, but some IDEs do. Plus, with JDK 17 you can use the `jar --validate` command to validate the JAR file.

Something to be aware of is the preview functionality present in some versions of the JDK. Some bigger features are first released as previews and might result in a final feature in one of the next JDKs. Those preview features are present in both LTS and non-LTS versions of Java. The features are enabled with the `enable-preview` flag and are turned off by default. If you use those preview features in production code, be aware that they might change between JDK versions, which could result in the need for some debugging or refactoring.

### More about Java deprecations and feature removals

Before upgrading the JDK, make sure your IDE, build tools, and dependencies are up to date. The Maven Versions Plugin and Gradle Versions Plugin show which dependencies you have and list the latest available version.

Be aware that these tools show only the new version for the artifacts you use—but sometimes the artifact names change, forks are made, or the code moves. For instance, JAXB was first available via `javax.xml.bind:jaxb-api` but changed to `jakarta.xml.bind:jakarta.xml.bind-api` after its transition to the Eclipse Foundation. To find such changes, you can use Jonathan Lermitage's Old GroupIds Alerter plugin for Maven or his plugin for Gradle.

**JavaFX.** Starting with Java 11, the platform no longer contains JavaFX as part of the specification, and most JDK builds have removed it. You can use the separate JavaFX build from Gluon or add the OpenJFX dependencies to your project.

**Fonts.** Once upon a time, the JDK contained a few fonts, but as of Java 11 they were removed. If you use, for instance, Apache POI (a Java API for Microsoft Office–compatible documents), you will need fonts. The operating system needs to supply the fonts, since they are no longer present in the JDK. However, on operating systems such as
Alpine Linux, the fonts must be installed manually using the `apt install fontconfig` command. Depending on which fonts you use, extra packages might be required.

**Java Mission Control.** This is a very useful tool for monitoring and profiling your application. I highly recommend looking into it. Java Mission Control was once included in the JDK, but now it's available as a separate download under the new name: JDK Mission Control.

**Java EE.** The biggest change in JDK 11 was the removal of Java EE modules. Java EE modules such as JAXB, mentioned earlier, are used by many applications. You should add the relevant dependencies now that these modules are no longer present in the JDK. **Table 1** lists the various modules and their dependencies. Please note that both JAXB and JAX-WS require two dependencies: one for the API and one for the implementation. Another change is the naming convention now that Java EE is maintained by the Eclipse Foundation under the name **Jakarta EE.** Your package imports need to reflect this change, so for instance `jakarta.xml.bind.*` should be used instead of `javax.xml.bind.*`.

**Table 1.** Java EE modules and their current replacements

<table>
<thead>
<tr>
<th>Module</th>
<th>Replacement artifactid</th>
</tr>
</thead>
<tbody>
<tr>
<td>java.activation</td>
<td>jakarta.activation</td>
</tr>
<tr>
<td>java.xml.ws.annotation</td>
<td>jakarta.annotation-api</td>
</tr>
<tr>
<td>java.transaction</td>
<td>jakarta.transaction-api</td>
</tr>
<tr>
<td>java.xml.bind</td>
<td>jakarta.xml.bind-api</td>
</tr>
<tr>
<td></td>
<td>jakxb-impl</td>
</tr>
<tr>
<td>java.xml.ws</td>
<td>jakarta.xml.ws-api</td>
</tr>
<tr>
<td></td>
<td>jaxws-rt</td>
</tr>
</tbody>
</table>

**CORBA.** There is no official replacement for Java's CORBA module, which was removed in Java 11. However, **Oracle GlassFish Server** includes an implementation of CORBA.

**Nashorn.** Java 15 removed the Nashorn JavaScript engine. You can use the `nashorn-core` dependency if you still want to use the engine.

**Experimental compilers.** Java 17 removes support for GraalVM's experimental ahead-of-time (AOT) and just-in-time (JIT) compiler, as explained in the documentation for JEP 410.

**Look out for unsupported major files**

You might see the error

`Unsupported class file major version 61`. I've seen it with the JaCoCo code coverage library and various other Maven plugins. The `major version 61` part of the message refers to Java 17. So in this case, it means that the version of the framework or tool you're using doesn't support Java 17. Therefore, you should upgrade the framework or tool to a new version. (If you see a message that contains `major version 60`, it relates to Java 16.)
Be aware that some tools such as Kotlin and Gradle don’t support Java 17 yet, at least as of the time I’m writing this (mid-August 2021). Sometimes it’s possible to work around that, for instance, by specifying Java 16 as the JVM target for Kotlin. However, I expect that Java 17 support will be added soon.

Encapsulated JDK internal APIs

Java 16 and Java 17 encapsulate JDK internal APIs, which impacts various frameworks such as Lombok. You might see errors such as module jdk.compiler does not export com.sun.tools.javac.processing to unnamed module , which means your application no longer has access to that part of the JDK.

In general, I recommend upgrading all dependencies that use those internals and making sure your own code no longer uses them.

If that’s not possible, there is a workaround to still enable your application to access the internals. For instance, if you need access to the comp module, use the following:

```
--add-opens=jdk.compiler/com.sun.tools.javac.comp=
```

However, use this workaround only as a last resort and preferably only temporarily, because you are circumventing important protections added by the Java team.

Learn more about this issue for Java 16 in JEP 396 and for Java 17 in JEP 403.

Java upgrades resource

I recommend that you look at the JavaUpgrades GitHub repository I created, which contains examples, common errors, and solutions that can help you during the upgrade process.

Conclusion

Upgrading your dependencies and adding dependencies for removed JDK features solves many of the Java upgrade challenges. I recommend a structured approach to upgrading step by step: First, make sure the code compiles, then run your tests, and then run the application.

The migration process is a lot better if you can tell yourself, your team, and your company that you can compile and test everything on JDK 17 instead of having to tell them it’s nearly done or, even worse, it’s only 80% complete.

My personal experience is that the upgrade from JDK 11 to JDK 17 is a lot easier than the upgrade from JDK 8 to JDK 11. However, in both scenarios, it was a matter of hours to days for nontrivial applications, and that was mainly due to waiting for builds to complete.

I hope this article eases your upgrade process.
Johan Janssen

Johan Janssen (@johanjanssen42) is a JavaOne Rock Star and a software architect at Sanoma Learning. He has been working for various fast-paced companies where rapidly delivering high-quality software is very important. He regularly writes articles and presents about various subjects, but he mainly focuses on Java.

Dig deeper

- A peek into Java 17: Continuing the drive to encapsulate the Java runtime internals
- Pedal to the metal: High-performance Java with GraalVM Native Image
- The hidden gems in Java 16 and Java 17, from Stream.mapMulti() to HexFormat