Guide   
for

Embedding Java

with your

Application

**Fabian Depry**

October 31, 2018

Contents

[Introduction 3](#_Toc528754223)

[Creating a Runtime Image 4](#_Toc528754224)

[Step 1 – Setup a working folder 4](#_Toc528754225)

[Step 2 – Analyze your module dependencies 4](#_Toc528754226)

[Step 3 – More dependencies analysis (optional) 4](#_Toc528754227)

[Step 4 – Create the Runtime Image 6](#_Toc528754228)

[Step 5 – Test the Runtime Image 6](#_Toc528754229)

[Step 6 – Package the new Runtime Image with your application 7](#_Toc528754230)

[Conclusion 7](#_Toc528754231)

# Introduction

The Module system got introduced with Java 9 and changed a lot of things in the Java ecosystem. If you are not familiar at all with modules, I strongly recommend you read online about that system. You do not need to be an export in modules to complete this guide, but you will need some basic understanding.

Another really big change that we are dealing with is that we won’t be able to use the Oracle Java 11 JVM without paying a subscription (which IMS is not going to do). That means when Java 8 is out-of-live by the end of December 2018, all our projects need to be switched to Java 11 OpenJDK (<http://jdk.java.net/>).

IMS has put together guidelines about how Java developers need to maintain the JDKs on their machine. It is important that you follow those guidelines.

Here is a summary that we provided to the SEER\*DMS developers.

*As Java developers, we need to be able to maintain multiple versions of Java on our machine and have full controls on our Java environment.*

*That is not possible using the Java version available in the Software Center.*

*For that reason, IMS is allowing us to copy Java installations into a specific folder on our computers: C:\jdk.*

*When you need to (re)install a specific version, checkout the following network folder: \\omni\st\seerdms\development\jdk. Just copy the zip file into your local C:\jdk\ folder and unzip it there (7zip will do a better job than the native Windows tool!).*

*We will keep that network folder up-to-date with all the Java versions we are allowed to use. If you need a version that is no in there, talk to your project manager. Do not download Java installations yourself online.*

*Don't forget to update or set a JAVA\_HOME environment variable (you have to set it under your own variables, not the System ones) that points to the folder you unzipped. If not done yet, also add a %JAVA\_HOME%/bin at the beginning of the Path variable.*

*Unless a software explicitly requires a Java version referenced via the Windows Registry, I would recommend not installing Java from the Software Center at all.*

If you do not have access to the “seerdms” network folder, you may download the JDK from the link I provided in this document. It is my understand that IMS might setup shared network folders with those OpenJDK installations so they all come from the same place and we don’t have to go online, but as far as I know, that work has not been completed yet.

The rest of this document assumes that you have Java 11 installed under “C:\jdk\jdk-11.0.1”.

# Creating a Runtime Image

## Step 1 – Setup a working folder

Start by assembling your application so you have access to all the JAR files (application + all dependencies) in a single folder (with Gradle, you can run the “assemble” task and it will put all the JAR under “build/libs” I believe.

Create a working folder you are going to use to execute the different tools (I called mine "seerabs-java11" since I tested this with my “seerabs” application) and put all the libraries in a subfolder (I kept the same "libs" name).

## Step 2 – Analyze your module dependencies

Execute the “jdeps” tool to analyze your dependencies. Run the following command; that last referenced JAR file should your main application JAR file:

C:\dev\seerabs-java11>C:\jdk\jdk-11.0.1\bin\jdeps --list-deps -cp libs\\*.jar libs\seerabs-application-2.13-beta.jar

When I did that first, it complained that one of the JAR as a “multi-version release”; if that’s the case, just remove the offending JAR for now. For me, I had to remove the “jaxb-api-x.x.jar” file.

You might see some warnings first but you should see the list of modules your applications uses at the end.

You will probably see a "jdk.unsupported" which is not great, it means your application (or more likely one of its dependency) is referencing a module internal to the JDK... For now those still work but at some point they will stop working...

The list of modules that are printed by the command will be used to create the Runtime Image.

## Step 3 – More dependencies analysis (optional)

This step is optional and not necessary to create the Runtime Image, but if you are curious about the warning and the unsupported JDK references, you can do the following to investigate them and see what library is the offender. Personally, I went through some of those “bad” libraries and found way to not include some of them in my project).

To drill down on the dependencies, you can use the “jdeps” tool again but with different arguments (it's very verbose, so write to a text file):

C:\dev\seerabs-java11>C:\jdk\jdk-11.0.1\bin\jdeps -cp libs\\*.jar libs\seerabs-application-2.13-beta.jar > jdeps-output.txt

For each dependency (and for the main application JAR), this will tell you what modules the dependency uses, and then it will tell you what packages of that dependency uses what packages of the target module; that allows you to really pin-point the issues.

For example, the first thing you can do in the output text file is look for " -> not found"; those indicate references to modules that “are not found”, meaning they are internal. Here is the result for me:

Line 138: caffeine-2.6.2.jar -> not found

Line 176: cglib-nodep-2.2.2.jar -> not found

Line 862: commons-logging-1.2.jar -> not found

Line 919: core-renderer-R8.jar -> not found

Line 1266: derby-10.10.2.0.jar -> not found

Line 2666: groovy-2.5.3-indy.jar -> not found

Line 4051: groovy-xml-2.5.3.jar -> not found

Line 4605: jaxb-core-2.2.11.jar -> not found

Line 4748: jaxb-impl-2.2.11.jar -> not found

Line 5115: joda-time-2.9.3.jar -> not found

Line 5342: log4j-1.2.17.jar -> not found

Line 6668: lucene-sandbox-4.7.2.jar -> not found

Line 6693: mongo-java-driver-3.4.0.jar -> not found

Line 7319: naaccr-xml-4.12.jar -> not found

Line 7624: seerabs-application-2.13-beta.jar -> not found

Line 8650: xercesImpl-2.8.0.jar -> not found

Line 9158: xstream-1.4.10.jar -> not found

All those libraries use internal JDK modules and will break in a future release. For libraries, the only thing you can do is to wait for a fix or switch to another library. For our own libraries or your application JAR, you can drill down and see what the problem is.

For example, there is one in that list for "naaccr-xml-4.12" which is an IMS library I maintain on GitHub. When I went down to that section and looked at the packages under "naaccr-xml-4.12", I saw a single entry for the missing package:

com.imsweb.naaccrxml -> javax.xml.bind not found

This is saying that the NAACCR XML library has a dependency to the internl "javax.xml.bind" package. It turned out I already looked into this one and I was able to remove the dependency. So updating to "naaccr-xml-4.13" resolved this bad dependency.

Like I said, you can only fix your own dependencies/application; for the third-party libraries, you just have to wait until they fix it (or switch no another library if that's possible). None of those internal access are a problem for now, they will become a problem in a future Java release.

PS - if you are using "seerutils" and/or "seerutils-gui", you should update to the latest version available. That's probably true for any "imsweb" library you use.

Another tool to help with understanding the dependencies of your application is to use Gradle itself (if you project is a Gradle project). This is useful because Gradle will correctly show the tree of dependencies, helping you understand the transitive nature of those dependencies. Instead, jdep flattens all of them.

To run Gradle, you have to go back in the folder of you project and execute the “dependencies” task.

Here is an example of how I did this:

C:\dev\projects\seerabs>gradlew dependencies > dep-output.txt

You might see some stuff still appearing on the command line, you can ignore them. Once this is done, copy the "dep-output.txt" file in your working folder (seerabs-java11 in my case). Open that file and look for "compile - Dependencies for source set 'main'"; that section will show you the dependencies in a tree. Very useful to know what JAR causes some issues in jdep since sometimes it’s a dependency of a dependency and you have no way to know that with just jdep.

## Step 4 – Create the Runtime Image

To create a minimized version of the JDK, you will need to add all the modules referenced in the jdeps output on the command line, using the “jlink” command. It's tedious and I am sure there will be tools to do that for us soon, but as a proof of concept, it can be done manually. Here is my command line:

C:\jdk\jdk-11.0.1\bin\jlink --module-path C:\jdk\jdk-11.0.1\jmods --add-modules java.compiler,java.datatransfer,java.desktop,java.logging,java.management,java.naming,java.prefs,java.security.jgss,java.security.sasl,java.sql,java.transaction.xa,java.xml,jdk.unsupported,jdk.xml.dom --output java-min --strip-debug --compress 2 --no-header-files --no-man-pages

This created a "java-min" folder that is about 40MB (compared to the initial jdk-11.0.1 which is 285MB!).

SEER\*Abs is distributed via a zipped custom installer written in Java as well; that installer went from 56MB to 84MB; it's not a tiny increase, but most people would find that acceptable (I certainly do).

You might want to experiment with the last arguments, I am not 100% sure of their effect (for example, I still need to make sure that full stracktrace are displayed in the logs of my application even if the strip-debug flag is set). Those options are the ones I found online to create a very small runtime image, but that doesn’t mean they are appropriate for you.

## Step 5 – Test the Runtime Image

Once you runtime image is created, do a "java -version" against it to make sure everything looks OK:

C:\dev\seerabs-java11>java\bin\java -version

openjdk version "11.0.1" 2018-10-16

The second test is to run your application with that runtime (if you had removed the jaxb-api JAR, make sure to re-add it to the libs folder). Here is how I started mine on the command line:

C:\dev\seerabs-java11>java\bin\java -Xmx1g -jar libs\seerabs-application-2.13-beta.jar

WARNING: An illegal reflective access operation has occurred

WARNING: Illegal reflective access by com.thoughtworks.xstream.core.util.Fields (file:/C:/dev/seerabs-java11/libs/xstream-1.4.10.jar) to field java.util.TreeMap.comparator

WARNING: Please consider reporting this to the maintainers of com.thoughtworks.xstream.core.util.Fields

WARNING: Use --illegal-access=warn to enable warnings of further illegal reflective access operations

WARNING: All illegal access operations will be denied in a future release

Most likely you will see warnings about accessing private fields; nothing we can do about that until the maintainers of the 3rd party libraries fix their code.

For me, the application started correct (it's not a huge app, but it's pretty decent and so the fact that it started was a very good sign). It's possible that yours won't start; you will have to then go back to the modules, see if you forgot any, and keep investigating.

Even if the application starts, it needs to be 100% tested on this new runtime; every single part of it.

## Step 6 – Package the new Runtime Image with your application

There is a lot of flexibility for this step and so what I am describing here might not be exactly what you need. Still, the idea is always the same: when releasing your application, you need to include this runtime environment, and you need to start your application against it. With this setup, your client doesn’t need a java environment on their local machine at all!

To start, I strongly suggest you include this new runtime image to your code repository (probably Git). It’s not that big and that will allow you to see changes over time, which is a nice bonus.

For example, in my SEER\*Abs project, I added a new “jre” folder that and under it a new “jre-11.0.1” with the new runtime. That will allow me in the future to add a “jre12” and test it while still being on the 11.0.1. At the end, when I build my installer, I just point to the runtime image I want to include.

If you are using an installer like InnoSetup, all you have to do is include the runtime image you want. Same if you distribute a Zip file without a proper installer.

For launching your application, what you need to do really depends on your setup. All my application use Launch4l but there are many other similar tools available. Launch4j has always supported an embedded JVM and so the transition to this method was trivial: I just had to tell Launch4j that when the executable file is double-clicked (seerabs.exe), it needs to look for a “jre” folder that will contain the Java environment. There is also an option to control whether Launch4j needs to look in other places if that environment is not there; that can be useful if some of your client don’t want to use the embedded environment (but I think that’s dangerous to allow that).

# Conclusion

I hope this document helped you!

You can find many information about “jdeps” and “jlink” online.

Here are a few links you mind find interesting:

A video about the new module system:

<https://www.youtube.com/watch?v=nKJbDYRsO0s&feature=youtu.be>

Pretty good migration document for JDK 8 to 10 (but most apply to 11 as well):

<https://docs.oracle.com/javase/10/migrate/toc.htm#JSMIG-GUID-7744EF96-5899-4FB2-B34E-86D49B2E89B6>

Minimizing the embedded JDK:

<https://www.azul.com/the-incredible-shrinking-java-platform/>

Another article about embedding the JDK:

<https://medium.com/azulsystems/using-jlink-to-build-java-runtimes-for-non-modular-applications-9568c5e70ef4>