Projekte erfolgreich nach Java 9 migrieren

Anton Epple, Eppleton IT Consulting

Das neue Modulsystem in Java 9 bietet endlich eine Lösung für die „Classpath Hell", aber was bedeutet das für bestehende Projekte? Dieser Artikel zeigt, was man tun muss, um ein bestehendes Projekt nach Java 9 zu portieren.

Es hat lange gedauert, bis das Java Platform Module System (JPMS) endlich Realität wurde. Vor zwölf Jahren wurde der erste Java Specification Request dazu formuliert. Nach vielen Diskussionen und einigen gescheiterten Anläufen ist es nun endlich soweit. Wer nun erwartet, dass das neue Modulsystem Frameworks wie OSGi ablöst, liegt allerdings falsch.

Zumindest in der derzeitigen Version bietet das Modulsystem nur minimale Features und kann ein ausgewachsenes Modulsystem wie OSGi nicht ersetzen. Der Fokus des JPMS liegt vielmehr darauf, eine „verlässliche Konfiguration" („reliable configuration") zu gewährleisten und „starke Kapselung" („strong encapsulation") zu ermöglichen. Ob diese Aufgaben wirklich erfüllt werden, wird derzeit noch

```
module de.eppleton.module1{
            // mit requires geben wir bekannt, dass
wir ein bestimmtes Modul benötigen
  requires de.eppleton.module2;
}
```

Listing 1

heftig diskutiert. Dieser Artikel konzentriert sich auf die praktischen Auswirkungen; deshalb zunächst nur ein kurzer Überblick über die wichtigsten neuen Features und darüber, was das in der Praxis für ein „normales" Projekt bedeutet.

Verlässliche Konfiguration

Das Ziel „verlässliche Konfiguration" wurde gewählt, um die bestehenden Probleme mit dem „Classpath" zu beheben. Bisher war der Entwickler dafür verantwortlich, der JVM die richtigen Abhängigkeiten über den Klassenpfad bekannt zu machen. Zur Laufzeit laden die „ClassLoader" dann bei Bedarf die nötigen Klassen aus den Ressourcen, die hier angegeben sind – fehlt eine Klasse, gibt es einen „NoClassDefFoundError".

In Java 9 wird der „Classpath" durch den „Modulepath" ersetzt. Mit dem neuen Modulsystem gibt jedes Modul seine Abhängigkeiten selbst bekannt. Das Modulsystem kann dann bereits beim Start prüfen, ob etwas fehlt. Dadurch werden Fehler früher erkannt und die Konfiguration wird verlässlicher. Das Modulsystem stellt ebenfalls sicher, dass jede Bibliothek nur genau einmal geladen werden kann. Um das zu gewährleisten, enthalten Module eine spezielle „module-info"-Klasse für die nötigen Informationen *(siehe Listing 1)*.

Starke Kapselung

Java bietet dem Entwickler ausreichende Möglichkeiten, um den Zugriff auf Typen und Methoden zu kontrollieren. Diese können als „public", „protected", „private", oder „package private" definiert sein. Auf Ebene der Packages fehlte jedoch bislang solch ein Mechanismus. In Java 9 kann man nun festlegen, wer auf Packages seines Moduls zugreifen darf. Das sind entweder nur das eigene Modul, eine Liste namentlich aufgeführter Module oder alle anderen Module. Die Konfiguration erfolgt wieder über die „module-info"-Klasse.

Jetzt kann man endlich mit Java-Bordmitteln verhindern, dass Entwickler auf Implementierungsdetails zugreifen und versehentlich eine Abhängigkeit darauf setzen, die beim nächsten Release Schwierigkeiten machen könnte. Der Entwickler eines Java-API kann dadurch die Implementierungsdetails ändern, ohne Rücksicht auf eventuelle Abhängigkeiten nehmen zu müssen *(siehe Listing 2)*.

Die modulare Plattform

Die Hauptarbeit bei der Modularisierung von Java steckt in der Aufteilung der Plattform selbst. Das Java-API wurde in Java 9 in kleinere Module unterteilt und der Zugriff auf Interna ist durch die Mechanismen der „starken Kapselung" beschränkt. Das ermöglicht es nun, nur die Teile mit einer Anwendung auszuliefern, die tatsächlich verwendet werden. Dies ist ein großer Vorteil für Desktop-Anwendungen, die eine eigene JVM mitbringen, und für „embedded" Anwendungen, die wenig Speicher zur Verfügung haben. Auch „containerisierte" Anwendungen profitieren von diesen Optimierungen. Gleichzeitig ist die Verkapselung der Interna eine der größten Herausforderungen für bestehende Anwendungen.

```
Module de.eppleton.module2{
            // nur diese Package ist von aussen für
andere Module zugänglich
  exports de.eppleton.module2.api;
            // Es ist auch möglich Exporte mit „to"
auf bestimmte Module zu beschränken
            // Nur diese können dann zugreifen
             exports de.eppleton.module2.impl
                              to de.eppleton.module3;
}
```

Listing 2

Um die Modularisierung möglich zu machen, wurden die Core-APIs massiv umgebaut. Für bestimmte Funktionen, die bislang nur durch den Zugriff auf interne Packages wie „sun.reflect" möglich waren, musste Ersatz geschaffen werden. Diese Änderungen muss man nun in den eigenen Anwendungen nachziehen.

Stufe 1: Test der bestehenden Anwendung auf Java 9

Wer nun sein Projekt auf Java 9 umziehen möchte, sollte zunächst einfach versuchen, die – mit Java 8 kompilierte – Anwendung unter Java 9 zu starten. Java 9 hat zwar den „Modulepath" eingeführt, um den „Classpath" zu ersetzen, es gibt diesen allerdings noch. Beide können sogar miteinander in der gleichen Anwendung zum Einsatz kommen.

Die Klassen des „Classpath" werden dann alle als Teil eines namenlosen Moduls betrachtet („Unnamed Module"). Dieses hat automatisch Abhängigkeiten auf alle exportierten Klassen der echten Module. Das werden wir später auch nutzen, um die Anwendung Schritt für Schritt zu migrieren. Umgekehrt haben die echten Module („named Modules") keinen Zugriff auf die Klassen des „Classpath" beziehungsweise des „Unnamed Module".

Falls man bislang die Regeln befolgt hat und die Anwendung keine internen Packages verwendet, die in Java 9 ersetzt wurden, stehen die Chancen gut, dass die Anwendung läuft. Es kann aber auch sein, dass sie interne APIs verwenden. In den meisten Fällen ist es dann eine Klasse aus einem Package aus dem „sun"-Namensraum, also zum Beispiel „sun.security.action.GetBooleanAction".

Es gibt auch ein paar Überraschungen. Zum Beispiel wurde die Methode „getPeer" aus der Klasse „java.awt.Component" entfernt, weil die Klasse „Peer" zum internen API zählt, obwohl das Package „java.awt.peer" zum Java-Namensraum gehört.

Wenn die Klasse nur zum internen API erklärt wurde, aber dennoch weiter existiert, lässt sich das Problem beheben, indem man mit-

```
swinggreeter-1.0-SNAPSHOT.jar -> java.desktop
   de.eppleton.swinggreeter.SwingGreeterService       ->
java.awt.peer.ComponentPeer                          JDK internal API (java.desktop)

Warning: JDK internal APIs are unsupported and private to JDK implementation that are
subject to be removed or changed incompatibly and could break your application.
Please modify your code to eliminate dependence on any JDK internal APIs.
For the most recent update on JDK internal API replacements, please check:
https://wiki.openjdk.java.net/display/JDK8/Java+Dependency+Analysis+Tool

JDK Internal API                          Suggested Replacement
----------------                          ---------------------
java.awt.peer.ComponentPeer               Should not use. See
https://bugs.openjdk.java.net/browse/JDK-8037739
```

*Listing 3: $ jdeps -jdkinternals libs/**

hilfe des Kommandozeilen-Parameters „-add-exports <Modul-mit-Interna>/<Name-der-internen-Package>=ALL-UNNAMED" die Import-Beschränkung aufhebt. Besser ist es jedoch, das Problem gleich richtig zu erledigen. Dazu kann man wieder das Tool „jdeps" verwenden. *Listing 3* zeigt, wie „jdeps" deutlich auf das Problem hinweist und einen Link mit Hinweisen zur Lösung des Problems liefert. Auch wenn die Anwendung läuft, sollte man das Tool verwenden, da es auch Fälle entdeckt, die beim Test der Anwendung vielleicht nicht abgedeckt sind. Sollte die Anwendung jetzt laufen, ist sie mit Java 9 kompatibel. Die Modularisierung kann nun Schritt für Schritt erfolgen.

Stufe 2: Kompilieren der Anwendung mit Java 9

Die nächste Hürde ist das Kompilieren der Anwendung mit Java 9. Auch wenn der Code Binär-kompatibel ist, ist er nicht automatisch Quelltext-kompatibel. Hier sind allerdings keine schlimmen Überraschungen zu befürchten. Die größte Änderung ist, dass der Unterstrich „_" von Java 9 an als Identifier verboten ist, da er eventuell einmal als Keyword verwendet wird. Man muss also gegebenenfalls einige Felder umbenennen. Auch hier liefert der Compiler Hinweise zur Lösung des Problems *(siehe Listing 4)*.

```
[ERROR] Failed to execute goal org.apache.maven.
plugins:maven-compiler-plugin:3.1:compile (default-
compile) on project hello: Compilation failure
[ERROR] /Users/antonepple/Entwicklung/Eppleton/train-
ings/Java9Migration/projects/legacyproject_porting/
hello/src/main/java/de/eppleton/hello/HelloWorld.
java:[8,9] as of release 9, '_' is a keyword, and may
not be used as an identifier
```

Listing 4

```
#vorher:
$JAVA9_HOME/bin/java -cp
lib1.jar;lib2.jar;lib3.jar;thirdpartylib.jar
de.eppleton.hello.HelloWorld

#nachher:
$JAVA9_HOME/bin/java -cp
lib1.jar;lib2.jar;lib3.jar -p thirdpartylib.jar
de.eppleton.hello.HelloWorld
```

Listing 5

Stufe 3: Third-Party-Bibliotheken werden Automatic Modules

Ist die Quellcode-Kompatibilität hergestellt, können wir mit dem eigentlichen Modularisieren beginnen. Als Erstes sollte man sich darum kümmern, externe Abhängigkeiten zu aktualisieren. Wenn es bereits eine Java-9-kompatible Version der Bibliothek gibt, dann sollte man diese auch verwenden. Wenn nicht, dann benutzt man die externe Bibliothek als „Automatic Module". Dazu wird die Bibliothek einfach vom „Classpath" auf den „Modulepath" verschoben.

Die JVM generiert sich dann automatisch eine „module-info"-Klasse. Da keine Information darüber vorliegt, welche Klassen exportiert werden sollen, werden einfach alle Packages freigegeben. Den Namen des Moduls legt die JVM anhand des Namens der JAR-Datei fest oder anhand eines bestimmten Eintrags in der Manifest-Datei („Automatic-Module-Name").

Wenn wir in einem späteren Schritt unsere eigenen JARs modularisieren, sind Abhängigkeiten auf die so erzeugten Module zu setzen. Zunächst ist das nicht nötig, da unsere Klassen auf dem Klassenpfad Teil des „Unnamed Module" sind und auf alle anderen Module, inklusive der automatischen Module, Zugriff haben.

Stufe 4: Migration von unten

Die empfohlene Vorgehensweise, um die eigenen Module zu portieren, ist die „Bottom-Up-Migration". Dazu muss man zunächst Kandidaten für die Modularisierung identifizieren, um sie aus dem „Unnamed Module" herauszulösen. Auch hier ist wieder das Tool „jdeps" sehr hilfreich. Es hilft bei der Suche nach einem Modul, das keine Abhängigkeiten auf die anderen nicht-modularen JARs hat. Das in *Listing 6* gezeigte JAR hat nur eine Abhängigkeit auf „java.base" und unser „thirdpartylib", aber auf kein anderes JAR des Classpath. Damit wäre es ein guter Kandidat.

Ein JAR mit weiteren Abhängigkeiten könnten wir nicht auf den „Modulepath" legen, denn dort hat es auf die Komponenten des „Classpath" („Unnamed Module") keinen Zugriff mehr. Haben wir ein passendes Modul identifiziert, legen wir eine Datei „module-info.

```
$ jdeps lib2.jar
lib2.jar -> java.base
lib2.jar -> thirdpartylib

   de.eppleton.lib2 -> java.lang         java.base
   de.eppleton.lib2 -> de.acme     thirdpartylib
```

Listing 6

```
$ jdeps lib1.jar
lib1.jar -> java.base
lib1.jar -> lib2.jar

de.eppleton.lib1   -> de.eppleton.lib2.api    lib2.jar
…
```

Listing 7

```
module de.eppleton.lib2{
  requires thirdpartylib;
           exports de.eppleton.lib2.api;
}
```

Listing 8

```
module module3{
            requires de.eppleton.module1;
  provides de.eppleton.module1.api.Service
                with de.eppleton.module3.services.
MyService;
}
```

Listing 9

```
module module2{
            requires de.eppleton.module1;
  uses de.eppleton.module1.api.Service;
}
```

Listing 10

java" im Wurzelverzeichnis des Quellcodes an. Als Modulname wird der Name des Basispackage empfohlen. Wenn unser Modul eine Third-Party-Bibliothek verwendet, müssen wir nun mit „requires" eine Abhängigkeit auf das entsprechende Modul setzen. Zusätzlich müssen wir die Packages exportieren, die von anderen Klassen verwendet werden sollen. Dazu nutzen wir wieder die Ausgabe des „jdeps"-Tools *(siehe Listing 7)*. Daraus ergibt sich dann die „module-info" *(siehe Listing 8)*.

Stufe 5: Schritt für Schritt modularisieren

Sobald alle Abhängigkeiten eines JAR vom „Classpath" auf den „Modulepath" gewandert sind, kann auch dieses JAR zum Modul werden. Die Abhängigkeiten, die wir zuvor mit „jdeps" identifiziert haben, müssen wir nun mit „requires" in der „module-info" angeben. So arbeiten wir uns von unten nach oben durch den Abhängigkeitsgraphen, bis der „Classpath" schließlich leer ist. Unsere Anwendung bleibt während dieses ganzen Prozesses immer lauffähig. So können wir notfalls auch die Zwischenschritte einrichten und Fehler bei der Migration schnell erkennen.

Stufe 6: Service-Abhängigkeiten migrieren

Seit Java 6 gibt es die Möglichkeit, Services zu registrieren und diese lose gekoppelt zu verwenden. Die Services werden bislang mithilfe einer Datei im Verzeichnis „META_INF/services" registriert. Um den Service zu konsumieren, wird die Klasse „ServiceLoader" verwendet. Solange sich die JARs auf dem „Classpath" befinden, funktioniert das auch in Java 9 problemlos. Auch nach der Modularisierung muss am Java-Code nichts geändert werden. Das Modul, das den Service implementiert, muss dies jedoch über die „module-info" mithilfe der Keywords „provides" und „with" bekannt geben *(siehe Listing 9)*. Das Modul, das den Service konsumiert, gibt dies mit dem Keyword „uses" ebenfalls bekannt *(siehe Listing 10)*.

Fazit

Dieser Überblicksartikel ist nicht auf jedes Detail eingegangen. In größeren Projekten wartet sicher noch die eine oder andere interessante Überraschung. Es besteht dennoch kein Grund, wegen der bevorstehenden Migration auf Java 9 in Panik zu verfallen. Es gab zwar viele Diskussionen und heftigen Widerstand gegen die Umsetzung. Sie kommen jedoch vor allem von den Herstellern der Build- oder Testing-Tools, JVMs oder Applikationsservern. Für normale Entwickler haben die Änderungen lange nicht so viele Auswirkungen. „Automatic Modules" und „Unnamed Module" bieten Möglichkeiten, die eigene Anwendung Schritt für Schritt zu portieren und zwischendurch immer wieder zu testen. Die Koexistenz von „Modulepath" und „Classpath" ermöglicht uns in vielen Fällen, Anwendungen ohne Änderungen auf Java 9 laufen zu lassen, sodass man der Migration relativ entspannt entgegensehen kann.

Anton Epple
toni.epple@eppleton.de

Anton „Toni" Epple ist Java-Entwickler der ersten Stunde und hat sich auf Client-Technologien spezialisiert. Er nutzt seine langjährigen Erfahrungen, um mit DukeScript eine moderne, Java-basierte Desktop-Technologie zu entwickeln. Toni ist Java Champion, JavaONE Rockstar, Mitglied im NetBeans Dream Team und Gewinner des Duke's Choice Award. Aktuell bereitet er Entwickler mit seinen Workshops auf den Umzug nach Java 9 vor.