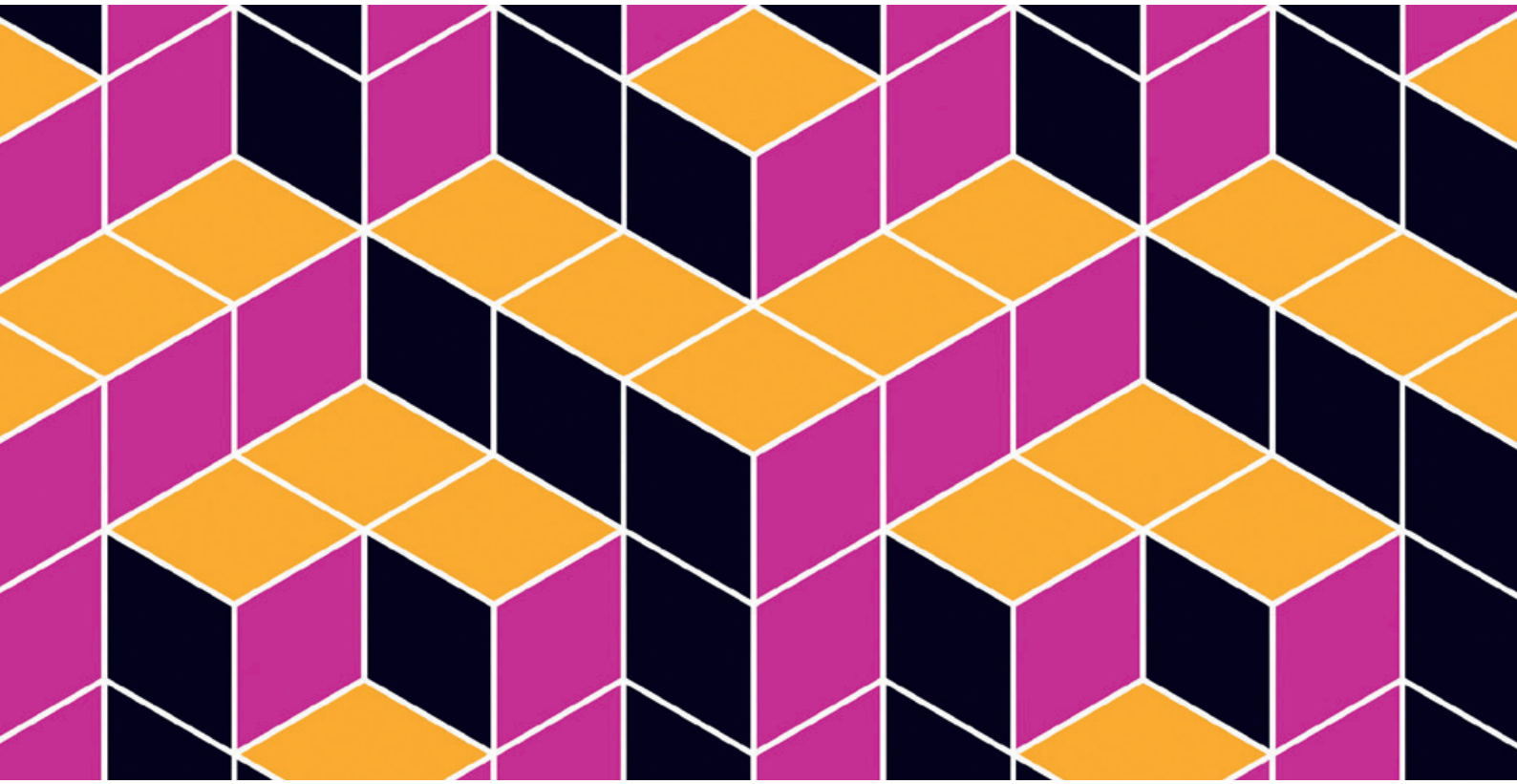

Modulare Multi-Release-JAR-Dateien

Guido Oelmann, Freelancer

Mit Java 9 wurde das Konzept der Multi-Release-JARs, auch als „MRJAR" bekannt, eingeführt. Mit diesem Feature ist es möglich, mehrere Versionen gleicher Klassen für unterschiedliche Java-Laufzeitumgebungen in ein einzelnes JAR zu verpacken. Der Artikel zeigt die Verwendung dieser Möglichkeit unter Berücksichtigung der Java-Versionen 8, 9 und 10 und von Java-Modulen.

Viele Frameworks und Bibliotheken unterstützen unterschiedliche Java-Versionen; beispielsweise unterstützt das Spring-Framework in der Version 4 die Versionen 6, 7 und 8 der Java-Plattform. Die Unterstützung verschiedener Java-Releases führt häufig dazu, dass Sprach-Features der neuesten Java-Version in den einzelnen Frameworks oder Bibliotheken nicht verwendet werden, um eine Abwärtskompatibilität einfacher zu gewährleisten. Die eigentlich notwendigen, bedingten Plattform-Abhängigkeiten im Programmcode auszudrücken oder die Verteilung verschiedener Artefakte für verschiedene Java-Versionen stellt sich dabei als schwierig dar.

Bei der Realisierung von bedingten Plattform-Abhängigkeiten im Programmcode greift man oft auf die Möglichkeiten von Reflection-Zugriffen zurück. Dabei wird dem Service-Provider-Mechanismus folgend eine allgemeine Provider-Schnittstelle definiert, die für alle Java-Versionen Gültigkeit hat und die für die unterschiedlichen Java-Versionen jeweils Implementierungen der Schnittstelle erstellt, die dann als Provider fungieren und zur Laufzeit geladen werden. Eine alternative Möglichkeit ist die Verwendung einer einzigen Klasse mit unterschiedlichen Methoden für unterschiedliche Versionen und der Zugriff auf diese mit Reflection. Dass dies nicht der optimale Lösungsansatz ist, sollte offensichtlich sein.

Die Verteilung verschiedener Artefakte ist eine weitere Möglichkeit, die im Grunde nur die Bereitstellung verschiedener JARs für die einzelnen Java-Versionen bedeutet. Hier würden die verschiedenen Implementierungen der gleichen Klasse gleichzeitig vorgehalten und es obliegt dem Build-Tool, diese in zwei verschiedene Artefakte zu kompilieren, zu testen und zu verpacken. Die Unterstützung für ein solches Vorgehen ist in den Build-Tools der Java-Welt unterschiedlich stark ausgeprägt.

Es zeigt sich also ein echtes Problem insbesondere für Drittanbieter von Bibliotheken und Frameworks, Sprach-Features neuerer Java-Versionen zu nutzen, die zum Teil einen Performance-Gewinn bedeuten würden, bei gleichzeitiger Berücksichtigung der Abwärtskompatibilität. Multi-Release-JAR-Dateien adressieren genau dieses Problem, indem das JAR-Dateiformat in der Form erweitert wurde, dass auch mehrere Java-Release-spezifische Versionen von Klassen und Ressourcen-Dateien gleichzeitig im selben Artefakt ihren Platz finden. Dadurch können insbesondere Drittanbieter von Bibliotheken und Frameworks die mit höheren Java-Versionen eingeführten Sprach-Features einfacher nutzen. Im folgenden Beispiel wird ein Multi-Release-JAR für die Java-Versionen 8, 9 und 10 unter Verwendung von Java-Modulen erstellt.

Multi-Release-JAR-Architektur

Zunächst ein Blick auf den Aufbau eines Multi-Release-JAR. *Listing 1* zeigt den grundsätzlichen Aufbau anhand eines Beispiels. Zu sehen ist der Inhalt eines Multi-Release-JAR, das Klassen für die Java-Versionen 8, 9 und 10 enthält. Bei den Dateien „Main.class" und „Example.class" im Hauptverzeichnis handelt es sich um die für Java 8 kompilierten Klassen. Von der Datei „Example.class" gibt es noch zwei weitere Versionen für Java 9 und 10, die unter „META-INF/versions" zu finden sind. Diese Dateien werden dann von einer Java-9- oder Java-10-Laufzeitumgebung anstatt der Datei im Hauptverzeichnis verwendet.

Im Falle einer Java-8-Laufzeitumgebung wird der Ordner „/versions" einfach ignoriert. Damit die JVM überhaupt weiß, dass es sich nicht um eine gewöhnliche JAR handelt, ist ein Eintrag in der „Manifest"-Datei erforderlich. Hierzu wird der Inhalt der „MANIFEST.MF"-Datei um das Attribut „Multi-Release: true" ergänzt. Bei einer JVM der Version 8 oder niedriger wird dieser Eintrag einfach ignoriert.

Ein Multi-Release-JAR erstellen

Zunächst wird eine geeignete Dateistruktur angelegt *(siehe Listing 2)*. Unterhalb des Projektverzeichnisses „multi-release-jar-example" liegen die drei Ordner „lib8", „lib9" und „lib10" mit den jeweiligen Sourcen für die verschiedenen Java-Versionen. Bei den Java-9- und Java-10-Versionen handelt es sich um Java-Module, weshalb sich oberhalb der Paketstruktur der Modul-Deskriptor „module-info.java" findet. Der Programmcode für die Java-8-Version steht im *Listing 3*.

Das Hauptprogramm „Main" erzeugt eine Instanz der Example-Klasse und führt dort die Methode „showVersion" aus. Die „Main"-Klasse soll für alle Java-Versionen gleich sein, nur die „Example"-Klassen sind unterschiedlich. *Listing 4* zeigt den Modul-Deskriptor und den Code für die Java-9-Version. Der Modul-Deskriptor der Java-10-Version sieht genauso aus und die Klasse „Example" unterscheidet sich nur darin, dass dort als Version „Java 10" ausgegeben wird. Der „ModuleLayer" wird angesprochen, um bei der Ausführung zeigen zu können, ob das JAR-Artefakt auf dem Klassen- oder auf dem Modulpfad liegt.

Bevor das eigentliche JAR erstellt wird, müssen zunächst alle Dateien für die jeweiligen Versionen kompiliert werden. Seit Java 9 gibt es dafür das Compiler-Flag „--release". Zum Kompilieren reicht also der Java-10-Compiler, um auch den Byte-Code für die niedrigeren Versionen zu erzeugen. Im Projektverzeichnis „multi-release-jar-example" sind die Befehle auszuführen, um die einzelnen Sourcen zu

```
multirelease.jar/
├── META-INF/
│   ├── versions
│   │   ├── 9
│   │   │   └── Example.class
│   │   └── 10
│   │       └── Example.class
│   └── MANIFEST.MF
├── Main.class
└── Example.class
```

Listing 1

```
multi-release-jar-example /
├── lib8/
│   ├── src/
│       ├── de/javaakademie/multirelease
│                             ├── Main.java
│                             └── Example.java
├── lib9/
│   ├── src/
│       ├── de/javaakademie/multirelease
│       │                     └── Example.java
│       └ module-info.java
└── lib10/
    └── src/
    └── de/javaakademie/multirelease
        │                     └── Example.java
        └ module-info.java
```

Listing 2

```
package de.javaakademie.multirelease;

import de.javaakademie.multirelease.Example;

public class Main {
    public static void main(String[] args) {
        new Example().showVersion();
    }
}

package de.javaakademie.multirelease;

public class Example {
    public void showVersion() {
        System.out.println("Version: Java 8");
    }
}
```

Listing 3

kompilieren *(siehe Listing 5)*. Danach werden die kompilierten Dateien in ein gemeinsames Multi-Release-JAR verpackt *(siehe Listing 6)*.

Beim Verpacken werden als Erstes die Dateien „Main.class" und „Example.class" als Default-Klassen in die Datei „multi-release-jar-example.jar" gepackt. Danach werden mit den Flags „--release 9" und „--release 10" die weiteren versionsbezogenen Klassen hinzugefügt. Durch die Verwendung des Flag werden die entsprechenden Strukturen unter „META-INF/versions" angelegt. Als Default-Klassen sollten immer die Klassen für die kleinste Java-Version gewählt werden.

Die Anwendung im resultierenden Artefakt kann dann mit „java -jar multi-release-jar-example.jar" auf den verschiedenen JVM-Versionen zur Ausführung gebracht werden. Allerdings würde die JAR-

```
module de.javaakademie.multirelease {
  exports de.javaakademie.multirelease;
}

package de.javaakademie.multirelease;

import java.lang.ModuleLayer;

public class Example {
    public void showVersion() {

        System.out.println("Version: Java 9");

        ModuleLayer ml = Example.class.getModule().getLayer();
        if( ml != null ) {
            System.out.println("Layer.Modules: " + ml.modules());
        } else {
            System.out.println("ModuleLayer ist null");
        }
    }
}
```

Listing 4

```
javac --release 8 -d lib8/classes lib8/src/de/javaakademie/multirelease/*.java

javac --release 9 -d lib9/classes lib9/src/module-info.java lib9/src/de/javaakademie/multirelease/Example.java

javac --release 10 -d lib10/classes lib10/src/module-info.java lib10/src/de/javaakademie/multirelease/Example.java
```

Listing 5

```
jar --create --file multi-release-jar-example.jar --main-class=de.javaakademie.multirelease.Main -C lib8/classes .
--release 9 -C lib9/classes .  --release 10 -C lib10/classes .
```

Listing 6

Datei dann auf dem Klassen- und nicht auf dem Modulpfad liegen. Bei Verwendung einer JVM ab Version 9 lässt sich die Anwendung außerdem mit „java -p . -m de.javaakademie.multirelease" starten, um den Modulpfad zu verwenden. Zum Starten wird lediglich der Modul-Name angegeben und mit den Informationen in „MANIFEST.MF" weiß die JVM zudem, wo die „Main"-Klasse zu finden ist.

Angemerkt sei an dieser Stelle, dass sich das Nachladen von Klassen oder Ressourcen zur Laufzeit etwas anders darstellt als bei gewöhnlichen JARs. Anstatt den Zugriff auf eine Klasse über „jar:file:/multi-release-jar-example.jar!/Example.class" durchzuführen, wird als Ressourcen-Pfad im „UrlClassLoader", wenn beispielsweise die Java-10-Version verwendet werden soll, „jar:file:/multi-release-jar-example.jar!/META-INF/versions/10/Example.class" angegeben. Das komplette Beispiel ist unter *„https://github.com/javaakademie/Modulare-Multi-Release-JAR"* abgelegt.

Fazit

Multi-Release-JARs bieten eine leichte und komfortable Möglichkeit, Klassen auszuliefern, die gleich mehrere Java-Plattform-Versionen unterstützen. Insbesondere Drittanbieter von Frameworks und Bibliotheken können dadurch neue Sprach-Features in ihren Programmcode einbringen, ohne direkt den gesamten Code des Projekts auf eine neue Version migrieren zu müssen. Anzumerken sei allerdings, dass sich in vielen Fällen die Berücksichtigung von Plattform-Abhängigkeiten durch Build-Tools durch das Erzeugen unterschiedlicher JARs eher anbietet. Abzuraten ist meistens von einer programmtechnischen Lösung wie dem Zugriff über Reflections. Der Artikel hat gezeigt, wie Multi-Release-JARs gebaut werden und wie die mit Java 9 eingeführten Java-Module dabei berücksichtigt werden können.

Guido Oelmann
guido.oelmann@javaakademie.de

Guido Oelmann arbeitet als freiberuflicher Software-Architekt, Berater und Trainer. Zu seinen Schwerpunkten gehört neben agilen Entwicklungsmethoden und Software-Architekturen der Einsatz von Java-/Java-EE-Technologien in verteilten Systemen. Er unterstützt Unternehmen durch die Mitarbeit in Entwicklungsprojekten und verfügt über viele Jahre Erfahrung beim Entwurf und bei der Entwicklung großer IT-Systeme in unterschiedlichen Branchen. Darüber hinaus ist er Autor des Buches „Modularisierung mit Java 9", das im dpunkt.verlag erschienen ist.