trivadis
Part of Accenture

Java 9 - Die Neuerungen im Überblick

Eine Einführung in das Modulsystem und weitere Neuerungen mit Java 9

AUTOR

Steffen Jacobs
Orientation in Objects GmbH

Veröffentlicht am: 26.6.2018

JAVA 9 - DIE NEUERUNGEN IM ÜBERBLICK

Mit dem Release von Java 9 hat sich in der Java-Welt einiges verändert: Es gibt erstmals ein Modulsystem, welches das JDK in viele kleinere, voneinander abgekapselte Päckchen aufteilt. Um dieses Modulsystem richtig verwenden zu können, ist es höchst relevant, über die Motivation und die Funktionsweise dieses Modulsystems im Java-Kontext Bescheid zu wissen.

Neben dem Modulsystem gab es mit Java 9 auch noch eine größere Menge an Änderungen an den Standardbibliotheken und den zusammen mit dem JDK ausgelieferten Tools. Hierzu gibt es weiter unten eine Übersicht mit weiterführenden Quellen.

Zusätzlich zu all den Änderungen an der Java-Sprache und der Java Plattform gibt es auch organisatorische Neuigkeiten. Für einen Java-Entwickler, insbesondere im Enterprise-Umfeld, ist dabei das neue Java Release Modell höchst relevant, besonders in Bezug auf die Änderungen im Bereich Long-Term-Support. Dieses neue Modell hat weitreichende Implikationen für die nächsten Java-Releases und deren Funktionsumfang, die am Ende des Artikels beschrieben werden. Abschließend folgt noch ein kurzer Ausblick auf Java 10 und Java 11.

) Schulung)

) Beratung)

) Entwicklung)

) Artikel)

Trivadis Germany GmbH

Weinheimer Str. 68
D-68309 Mannheim

Tel. +49 (0) 6 21 - 7 18 39 - 0
Fax +49 (0) 6 21 - 7 18 39 - 50

www.oio.dekontakt@trivadis.com

DAS MODULSYSTEM

Die wahrscheinlich größte und relevanteste Änderung mit Java 9 ist das neue Modulsystem. Dieses Java-Modulsystem (Codename: Jigsaw) wurde initial mit Java 7 angekündigt. Die Einführung hat sich jedoch zunächst bis Java 8 und dann bis Java 9 verzögert. Das Primärziel des Java-Modulsystems ist es, die Java Standard Edition (JSE) und das Java Development Kit insbesondere im Hinblick auf leistungsschwache Endgeräte skalierbarer zu machen. Außerdem soll das Modulsystem dabei helfen, die Erstellung und Wartung von Programmbibliotheken und großen Anwendungen zu verbessern. Gleichzeitig soll die Sicherheit und Performance durch die Abkapselung der JDK-Elemente in Module gesteigert werden. Das Modulsystem soll zuverlässig sein, eine starke Kapselung durchsetzen, die Java Plattform skalierbarer machen und eine bessere Plattformintegrität garantieren. [JGSW, JMOD, JGSWO]

DIE MOTIVATION

Wofür wird das Java-Modulsystem denn nun in einer eigenen Anwendung gebraucht? Diese Frage soll hier anhand eines Beispiels beantwortet werden.

Angenommen, es ginge um die Entwicklung einer browserbasierten Enterprise Application. Die erste Unterteilung, die man hierbei vornehmen könnte, ist das Projekt auf Build-Ebene in Frontend und Backend zu unterteilen. Dazu könnte man in Java etwa zwei separate Packages anlegen: Eins für das Frontend und eins für das Backend. Theoretisch würde man zwischen Frontend und Backend nur über eine wohldefinierte, fixierte Menge an Schnittstellen kommunizieren wollen. Alle anderen internen Klassen, die Implementierungsdetails zu Frontend und Backend enthalten, sollten außerhalb der entsprechenden Packages nicht sichtbar sein.

Das Problem hierbei ist, dass alle Klassen, die `public` sind, im jeweils anderen Package trotzdem sichtbar sind. Um das zu lösen, könnte man komplett auf Unterpackages verzichten und sämtliche Klassen in dem jeweiligen Package auf package-private setzen. Hierbei würde die Übersicht allerdings sehr schnell verloren gehen.

Viel angenehmer ist hier ein Modulsystem, welches sämtliche Zugriffe von einem fremden Package sperren kann. Eine solche Lösung präsentiert das neue Modulsystem in Java 9.

Hierbei wird durch das Modulsystem eine Art Filter oberhalb von `public` einfügt. In jedem Modul können Schnittstellen definiert werden, die von außen sichtbar sind. Außerdem kann festgelegt werden, welche anderen Module zur Ausführung benötigt werden. Somit sind selbst Klassen, die in einem Package `public` sind, nicht zwingend von jedem anderen Package aus sichtbar. [JMOD]

WIE FUNKTIONIERT DAS NEUE MODULSYSTEM

Normalerweise ist jede öffentliche Klasse einer Bibliothek oder eines Anwendungsprojekts von außen für alle sichtbar. Jetzt kann ausgewählt werden, welche Klassen oder Schnittstellen für den Endbenutzer sichtbar und welche Klassen unsichtbar sein sollen.

DER MODULDESKRIPTOR (MODULE-INFO.JAVA)

Dazu muss lediglich ein Moduldeskriptor (`module-info.java` -Datei) in das Projekt eingefügt werden [JGSWQS]. Ein erster, einfacher Moduldeskriptor ist im folgenden Listing zu sehen:

```
module de.oio.firstModule {
}
```

Beispiel 1: Einfache Moduldefinition des Moduls `de.oio.firstModule` in der Datei `module-info.java`

Wie im Listing oben zu erkennen, enthält dieser Deskriptor den Modulnamen und zwei geschweifte Klammern.

Im nächsten Schritt sollen nun die Abhängigkeiten des Moduls `firstModule` über den Moduldeskriptor in der `module-info.java` dargestellt werden. Dazu verwenden wir das Schlüsselwort `requires` gefolgt von dem Namen des Moduls, welches importiert werden soll. In diesem Beispiel werden die beiden Module `someModule` und `otherModule` importiert.

```
module de.oio.firstModule {
   requires de.oio.someModule;
   requires de.oio.otherModule;
}
```

Beispiel 2: Moduldefinition mit Abhängigkeiten

Im letzten Schritt soll nun spezifiziert werden, welche Packages aus unserem `firstModule` nach außen sichtbar gemacht werden sollen. Dazu wird die `exports` -Klausel verwendet.

```
module de.oio.firstModule {
   requires de.oio.someModule;
   requires de.oio.otherModule;
   exports de.oio.firstModule.api;
}
```

Beispiel 3: Moduldefinition mit Abhängigkeiten und Exporten

Unser Modul heißt nun also `de.oio.firstModule`. Es bestehen Abhängigkeiten von `de.oio.someModule` und `de.oio.otherModule`. Nach außen verfügbar gemacht werden alle Klassen im Package `de.oio.firstModule.api`. Sämtliche Klassen, die sich nicht im `de.oio.firstModule.api` -Package befinden, sind von außen weder sichtbar, noch über die Reflection-API abrufbar. [REFL]

Zusätzlich könnte noch spezifiziert werden, wohin die Klassen exportiert werden. So können beispielsweise einige Klassen für alle Module exportiert werden, während andere Klassen nur für konkrete andere Module exportiert werden sollen. Dazu ein Beispiel:

```
module de.oio.firstModule {
   exports de.oio.firstModule.api;
   exports de.oio.firstModule.internalApi to de.oio.secondModule;
}
```

Beispiel 4: Moduldefinition mit Exporten an konkretes anderes Modul

In diesem Beispiel wird etwa das Package `de.oio.firstModule.internalApi` nur an das Module `de.oio.secondModule` exportiert.

TRANSITIVE ABHÄNGIGKEITEN

Nun ein weiteres kurzes Beispiel, dieses Mal mit transitiven Abhängigkeiten. Die Abhängigkeiten zwischen den Modulen `A` bis `D` sind in der folgenden Abbildung visualisiert:

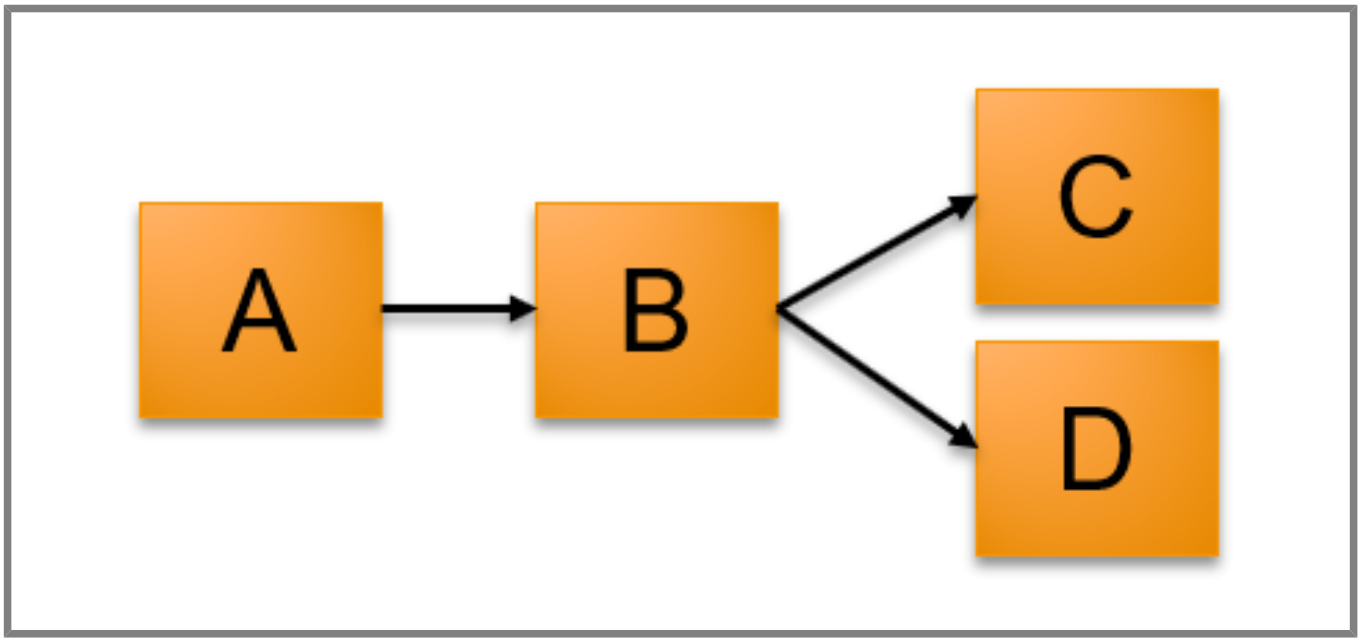

Abbildung 1: Abhängigkeiten der Module `A` bis `D`

In diesem Beispiel hängt Modul `A` von Modul `B` ab und Modul `B` von den Modulen `C` und `D`.

Hierbei ist zu beachten, dass der Abhängigkeitsgraph, der durch die `requirements` beschrieben wird, standardmäßig **nicht transitiv** ist. Daher hängt in unserem Beispiel `A` selbst nicht direkt von `C` oder `D` ab. Somit kann es hier zu einem Compilerfehler kommen. Um dieses Problem zu lösen gibt es zwei mögliche Ansätze:

1. Jede weitere Abhängigkeit wird explizit direkt im Hauptmodul definiert (sprich ein `requires C` und `requires D` im Moduldeskriptor von Modul `A`). Beispiel:

```
module de.oio.A {
   requires de.oio.B;
   requires de.oio.C;
   requires de.oio.D;
}
```

Beispiel 5: Moduldefinition für Ansatz 1 (Direkte Abhängigkeit)

2. Im Moduldeskriptor von Modul A wird statt des Schlüsselworts `requires B` der Satz `requires transitive B` verwendet. Diese "Transitivität" geht allerdings nur eine Ebene tief. Sollte jetzt etwa Modul D von einem weiteren Modul E abhängen, so muss das Schlüsselwort `transitive` eine Ebene tiefer wiederholt werden. Beispiel:

```
module de.oio.A {
   requires transitive de.oio.B;
}
```

Beispiel 6: Moduldefinitionen für Ansatz 2 (Transitive Abhängigkeit)

Nun können auch die exportierten Klassen aus den Modulen C und D im selbst erstellten Modul A verwendet werden.

REFLECTION-ZUGRIFF AUF FREMDE MODULE

Um (beispielsweise via Reflection-API) auch auf die internen Werte und Klassen zugreifen zu können, wird ein weiteres Schlüsselwort benötigt, nämlich `opens`. In diesem Beispiel soll das Modul A per Reflection-API auf Modul B zugreifen. Dafür muss der Moduldeskriptor von Modul B wie folgt angepasst werden:

```
module de.oio.B {
   opens de.oio.B.internal;
   exports de.oio.B.internal;
}
```

Beispiel 7: Angepasste Moduldefinitionen von Modul B für Reflection-Zugriff von Modul A auf Package de.oio.B.internal

Nun kann Modul A per Reflection-API auf die internen Klassen und Werte im Package `de.oio.B.internal` im Modul `B` zugreifen. Im Übrigen ersetzt das Schlüsselwort `opens` das bereits bekannte Schlüsselwort `exports` nicht. Vielmehr ermöglicht das Schlüsselwort `opens` einen Zugriff per Reflection-API, während `exports` einen Zugriff auf die `public` Klassen auf Sprachebene erlaubt.

Das `opens` -Schlüsselwort kann, genau wie `exports` auch, ausschließlich an ein spezifiziertes Modul exportieren:

```
module de.oio.B {
   opens de.oio.B.internal to de.oio.A;
   exports de.oio.B.internal to de.oio.A;
}
```

Beispiel 8: `opens` -Schlüsselwort an konkretes Modul

DER MODULE-PATH

Da ein Java Programm auf der obersten Ebene nun nicht mehr direkt aus Klassen, sondern aus Modulen besteht, die diese Klassen enthalten, wird statt des alten Klassenpfads nun ein Modulpfad benötigt [SMOD]. Dieser neue Modulpfad ist weitaus robuster als der Klassenpfad, da bereits beim Auflösen des Modulgraphen bekannt ist, wenn Module fehlen oder zwei Module den gleichen Namen haben. Derartige Probleme traten bisher erst zur Laufzeit auf und resultierten beispielsweise in einer `ClassNotFoundException`. Da der Modulgraph bereits beim Compile-Vorgang und beim Start jeder Anwendung aufgelöst wird, fällt dieser Fehler viel früher auf und wird nicht erst sichtbar, wenn die betroffene Codestelle ausgeführt wird.

DAS UNNAMED UND DAS AUTOMATIC MODULE

Aus Kompatibilitätsgründen mit alten Java Projekten und JAR-Dateien ist die manuelle Erstellung einer `module-info.java` -Datei theoretisch optional. Sofern keine `module-info.java` -Datei gefunden wurde, kann ein Unnamed Module oder ein Automatic Module automatisch erstellt werden.

In das Unnamed Module werden sämtliche Packages eingefügt, die über den Classpath importiert wurden. Da das Unnamed Module, wie der Name schon sagt, keinen Namen besitzt, können andere Module nicht auf dieses Modul verweisen. Außerdem werden für das Unnamed Module alle anderen Module, die zur Verfügung stehen, importiert.

Das Automatic Module wird erstellt, wenn beispielsweise eine JAR-Datei über den Module-Path eingeladen wurde. Der Modulname wird hier aus dem Namen der JAR-Datei erschlossen. Sofern eine Versionsbezeichnung im Dateinamen vorhanden ist, wird diese vorher entfernt. Wie auch das Unnamed Module, importiert das Automatic Module alle zur Verfügung stehenden anderen Module. Zusätzlich werden hier alle internen Packages exportiert.

Die Classdateien in den jeweiligen Modulen werden dabei nicht analysiert. Daher kann es beim Starten eines der generierten Module zu `ClassNotFoundExceptions` kommen.

Dieses beinhaltet den gesamten alten Classpath. Als Importanforderung (`requires`) werden einfach alle Module aus dem JDK angefordert. Bei den Exporten (`exports`) werden alle Klassen des Moduls nach außen exportiert und damit für alle zugreifbar gemacht.

DAS MODULSYSTEM IN DER PRAXIS

Nach dieser theoretischen Einführung in das Modulsystem nun ein kleines Beispiel mit zwei Modulen. Das erste Modul beinhaltet eine einfache Java Anwendung und das zweite Modul eine minimale Bibliothek mit einer ausgelagerten Funktion. Die Java Anwendung besteht nur aus einer Main-Klasse, während die Bibliothek aus einer API und einer internen Implementierung besteht. Die beiden Module sind hier noch einmal anhand ihrer Dateistruktur visualisiert:

```
Java9Jigsaw/
   de.oio.java9lib
      de
        oio
        java9lib
        export
          SomeAPIInterface.java

        internal
        InternalImpl.java
      module-info.java

   de.oio.java9app
       de
         oio
         java9app
         Main.java
       module-info.java
```

Beispiel 9: Dateistruktur

Die Klasse in `InternalImpl.java`:

```
public class InternalImpl implements SomeAPIInterface {
   public int someValue = 5;
   @Override
   public void doStuff() {
      // do important stuff
   }
}
```

Beispiel 10: Inhalt der Klasse `InternalImpl`

Die `InternalImpl` -Klasse besitzt eine öffentliche Variable `someValue` vom Typ `Integer`, die auf 5 gesetzt ist. Außerdem überschreibt sie die `doStuff()` -Methode des Interfaces `SomeAPIInterface`.

Die Implementierung der `Main` -Klasse sieht wie folgt aus:

```
import (...)
public class Main {
   public static void main(String... args) {
      /* working as expected */
      SomeAPIInterface interf;
      /* not working */
      InternalImpl impl = new InternalImpl();
      /* not working either */
      int value = impl.someValue;
   }
}
```

Beispiel 11: Inhalt der `Main` -Klasse

Nun die Moduldeskriptoren. Zunächst der Moduldeskriptor für das Bibliotheksmodul:

```
module de.oio.java9lib {
   exports de.oio.java9lib.export;
}
```

Beispiel 12: Moduldefinition des Bibliotheksmoduls

Wie aus der `module-info.java` -Datei hervorgeht, ist der Name des Moduls `de.oio.java9lib` (erste Zeile). Es wird nur das Package `de.oio.java9lib.export` exportiert. Im Bild der Dateistruktur oben ist zu sehen, dass sich in diesem Package nur das Interface `SomeAPIInterface` befindet. Die Klasse `InternalImpl` befindet sich in einem anderen Package und wird hier nicht exportiert. Daher wird sie später auch nicht zugreifbar sein.

Der Moduldeskriptor für die Hauptanwendung sieht wie folgt aus:

```
module de.oio.java9app {
   requires de.oio.java9lib;
}
```

Beispiel 13: Moduldefinition des Bibliotheksmoduls

Wie zu erwarten, sieht die Main-Klasse nun zwar das Interface `SomeAPIInterface` und kann korrekt eine Variable von diesem Typ deklarieren. Die nicht exportierte Klasse `InternalImpl` ist dagegen weder sichtbar, noch lässt sich darauf zugreifen. Die Zeilen 9 und 11 geben bereits zur Compile-Zeit einen Fehler.

Hier wird das oben definierte Modul `de.oio.java9lib` in das Modul `de.oio.java9app` (die Hauptanwendung) importiert.

KRITIK

Was noch fehlt, wäre eine standardisierte Versionierung der Module. Diese ist auch nicht mit einem in anderen Quellen beschriebenen Workaround nachrüstbar. Hierbei wurde vorgeschlagen, für unterschiedliche Versionen der gleichen Bibliothek einfach mehrere JARs mit unterschiedlichen Dateinamen (etwa Library-v1.0, Library-v1.1) eingebunden werden. Dies **funktioniert nicht**. Beim Laden von JAR-Dateien wird nämlich zunächst einmal ein gegebenenfalls existierender Versionszusatz entfernt. Außerdem werden Klassen aus dem Classpath ja wie oben beschrieben in das separate Unnamed-Module geladen. Da es nicht möglich ist, mehrmals das gleiche Package aus unterschiedlichen Quellen zu importieren, gibt es spätestens hier einen Fehler oder unerwartetes Verhalten.

Im Kontrast dazu ist es möglich, unterschiedliche Versionen einer Bibliothek oder eines allgemeinen Programms für unterschiedliche Java-Versionen vorzuhalten. Mehr dazu im Abschnitt Multi-Release-JARs.

Außerdem müssen alle Module beim Java-Modulsystem zur Compile-Zeit deklariert werden und können nur beim Start der Anwendung geladen werden. Ein dynamisches Nachladen von Modulen, wie etwa bei OSGi ist nicht möglich. Die Hauptmotivation des Java-Modulsystems ist es hierbei auch gar nicht, OSGi obsolet zu machen. Stattdessen ist und bleibt der Fokus des Modulsystems die Modularisierung des JDK. Es ist auch möglich, das Java-Modulsystem zusammen mit OSGi zu verwenden. Die Kontroverse um Java Module vs. OSGi befindet sich jedoch außerhalb des Rahmens dieses Artikels und soll hier nicht weiter vertieft werden.

OSGi

Das OSGi-Framework der OSGi Alliance [OSGi] beinhaltet eine offene Plattform auf Basis des Komponentenmodells aus dem Automotive-Sektor. In OSGi können Module erstellt werden, sogenannte Bundles, die dann dynamisch bei einer sogenannten Service Registry zur Laufzeit registriert und deregistriert werden. Wie auch im Java-Modulsystem kann in den Modulen definiert werden, welche Packages und Klassen exportiert und welche importiert werden sollen.

MULTI RELEASE JARS MIT DEM MODULSYSTEM

Viele Programmbibliotheken und Frameworks in Java unterstützen mehrere Java-Versionen. Das führt dazu, dass neue Sprachfeatures und neue Funktionen der Plattform-API in diesen Projekten nur zögerlich umgesetzt werden, um die Abwärtskompatibilität mit alten Java-Versionen nicht zu untergraben. Insbesondere im Hinblick auf die größeren Änderungen an der Plattform-API im Zuge von Java 9 ergibt sich das Problem, dass viele Programmbibliotheken entweder das Modulsystem von Java 9 umgesetzt haben und damit voll zu Java 9 konform sind, oder nicht. Als Resultat werden dann mehrere JAR-Artefakte für die unterschiedlichen Java-Versionen veröffentlicht. An dieser Stelle schafft das Multi-Release JAR-File in Java 9 Abhilfe.

Seit Java 9 ist es nun möglich, unterschiedliche Versionen derselben Java-Klasse für unterschiedliche Java-Versionen in der gleichen JAR-Datei vorzuhalten. Bei der Entwicklung einer Programmbibliothek können also für die Java 9-Nutzer der Bibliothek spezielle Funktionen vorgehalten werden und spezielle Sprachfeatures verwendet werden, ohne dass die Java 8-Nutzer davon etwas mitbekommen oder die Klassen auch nur sehen. Beim Ausführen des Programms werden dann ausschließlich die Klassen ausgeführt, welche für die verwendete JVM-Version vorgesehen sind. Diese Funktionalität funktioniert auch für Ressourcendateien. [MRJ, JDK9]

Um ein Multi-Release JAR als solches zu kennzeichnen, muss in der MANIFEST.MF-Datei innerhalb der JAR-Datei zunächst das Attribut `Multi-Release: true` gesetzt werden. Dies ist ein Hinweis an die JVM, die die JAR ausführen soll und wird von JVMs mit einer Java-Version von 8 oder niedriger einfach ignoriert.

MULTI-RELEASE JARS ERSTELLEN

Zunächst erstellen wir drei Ordner für die drei Java-Versionen (8, 9, 10), die unser kleines Testprogramm unterstützen soll. Diese Ordner nennen wir `src8`, `src9` und `src10`. Unser Testprogramm besteht aus den 2 Klassen `Main.java` und `Example.java`:

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

Beispiel 14: Die Main-Klasse

```
public class Example {
   public void doStuff() {
      System.out.prinln("Beispiel für Java 8");
   }
}
```

Beispiel 15: Die Example-Klasse

Die Main-Klasse erzeugt dabei eine Instanz der Example-Klasse und ruft die Methode `doStuff` auf. Diese gibt anschließend den Text "Java 8 Example" auf der Konsole aus. Um die Snippets und insbesondere die später folgenden Konsolenbefehle kurz zu halten, werden diese beiden Klassen im Default-Package belassen. Die Main-Klasse soll für alle Versionen identisch sein, nur die Example-Klasse soll für die Java-Versionen 8, 9 und 10 unterschiedliche Implementierungen erhalten.

Die beiden oben erstellten Java-Dateien legen wir nun unter `src8/` ab. Nun zur Version der Example.java für Java 9:

```
public class Example {
   public void doStuff() {
      System.out.prinln("Beispiel für Java 9");
   }
}
```

Beispiel 16: Die Example-Klasse (Java 9)

Diese sieht sehr ähnlich wie die für Java 8 aus, nur wird hier "Java 9 Example" statt "Java 8 Example" ausgegeben. Diese Datei legen wir unter src9/Example.java ab. Schließlich die Example.java für Java 10:

```
public class Example {
   public void doStuff() {
      System.out.prinln("Beispiel für Java 10");
   }
}
```

Beispiel 17: Die Example-Klasse (Java 10)

Abgelegt wird diese unter `src10/Example.java`. Unsere Dateistruktur sollte nun wie folgt aussehen:

```
./
   src8/
      Main.java
|     Example.java
    src9/
|     Example.java
    src10/
      Example.java
```

Beispiel 18: Dateistruktur

Diese Java-Dateien müssen nun alle für ihre jeweilige Version kompiliert werden. Dafür gibt es seit Java 9 das Compiler-Flag `--release`, welches auch gleich die korrekte Plattform-API verwendet.

Hier die Kommandozeilenbefehle, um alle Dateien mit dem Java 10 Compiler korrekt zu kompilieren:

```
$ javac --release 8 src8/*.java
$ javac --release 9 src9/*.java
$ javac --release 10 src10/*.java
```

Im nächsten Schritt werden diese nun zu einer gemeinsamen JAR-Dateien zusammengefügt. Der zugehörige Befehl ist recht lang und setzt sich aus den folgenden Segmenten zusammen:

- `jar cfe <Dateiname> <Haupteinstiegspunkt>`: Das JAR-Programm aus dem JDK wird mit dem Parametern `cfe` (c: neues Archiv erstellen, f: Dateiname, e: Haupteinstiegspunkt) und den Parametern für Dateiname und Haupteinstiegspunkt gestartet.
- `-C <Ordner>, <Klasse>`: Angabe eines weiteren Ordners mit einer Klasse, welche zum JAR-Archiv hinzugefügt werden soll.
- `-- release 9 C <Ordner>, <Klasse>`: Angabe des Ordners mit der gleichen Klasse für Java-Version 9
- `-- release 10 C <Ordner>, <Klasse>`: Angabe des Ordners mit der gleichen Klasse für Java-Version 10

Der vollständige Befehl für unser Beispiel lautet somit:

```
$ jar cfe multi.jar Main -C src8 Main.class -C src8 Example.class
--release 9 -C src9 Example.class --release 10 -C src10 Example.class
```

Zunächst werden die Dateien `Main.class` und `Example.class` aus dem `src8` -Ordner als Default-Klassen in die Datei `multi.jar` verpackt. Anschließend wird mittels der Flag `--release 9` die Datei `Example.class` aus dem Ordner `src9` für Java-Version 9 vorgemerkt. Analog dazu wird mittels `--release 10` die `Example.class` -Datei aus dem `src10` -Ordner für Java-Version 10 vorgemerkt.

Anschließend kann die Datei multi.jar mit dem Befehl java -jar multi.jar gestartet werden. Hier das Resultat für die Java 10 JVM:

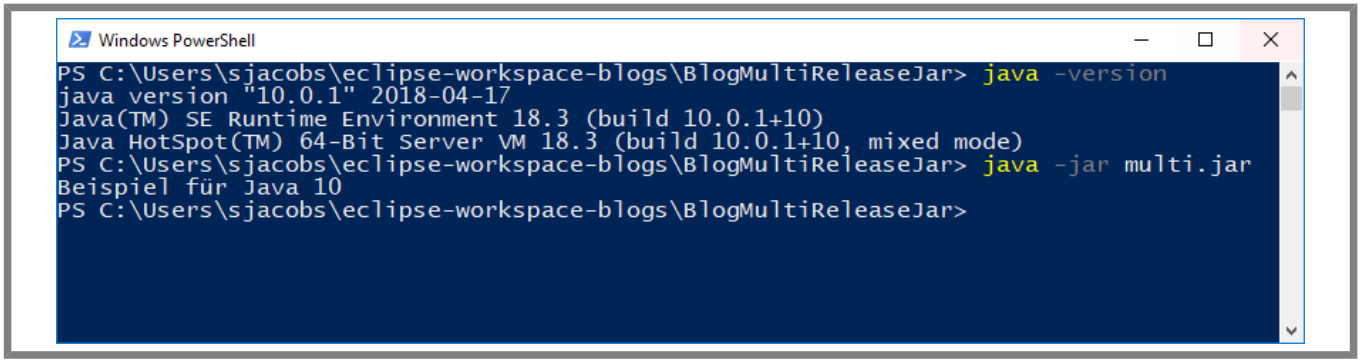

Abbildung 2: Multi Release JAR ausführen unter Java-Version 10

Das gleiche Programm auf einer JVM mit Java 9:

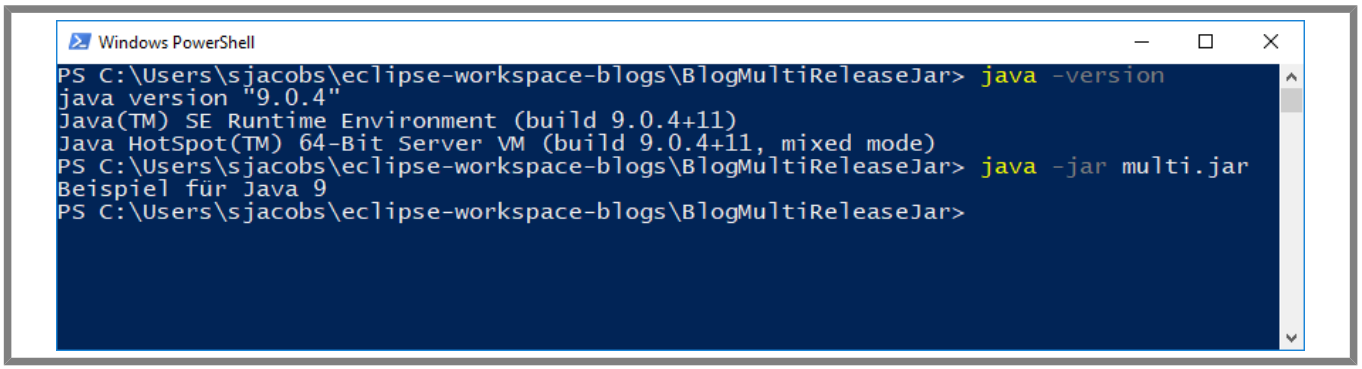

Abbildung 3: Multi Release JAR ausführen unter Java-Version 9

Und mit Java 8:

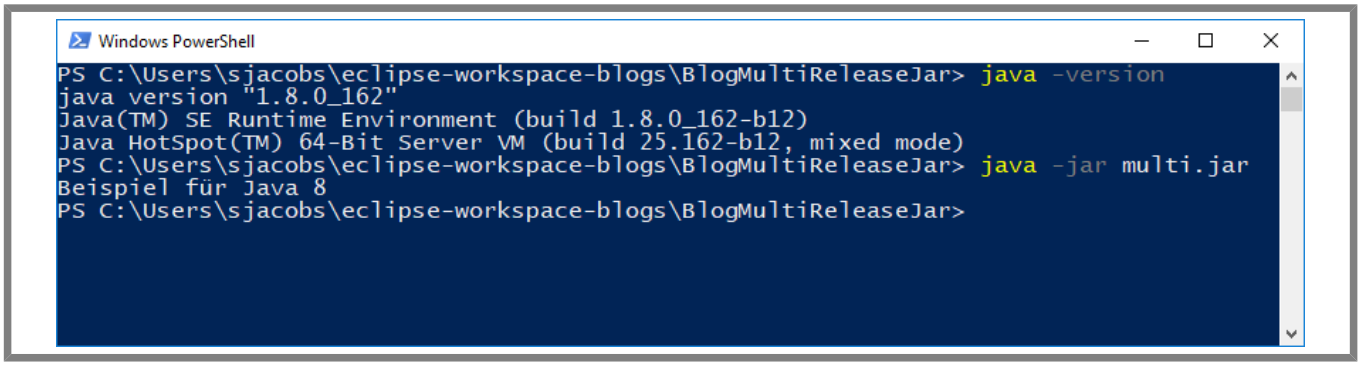

Abbildung 4: Multi Release JAR ausführen unter Java-Version 8

Eine Build-Tool-Unterstützung in Maven und Gradle gibt es für den Multi-JAR-Build momentan nur durch Plugins. Wie diese im Detail funktionieren, geht allerdings bei weitem über den Fokus dieses Artikels hinaus.

ÄNDERUNGEN AN DEN STANDARDBIBLIOTHEKEN

Neben dem Modulsystem gibt es noch zahlreiche weitere kleinere und größere Änderungen. Diese sind im Folgenden tabellarisch aufgelistet. Außerdem ist jeweils ein mehrseitiger Artikel zu jedem der Themen referenziert, welcher sehr viel stärker ins Detail geht.

HTTP/2-Client	Der neue HTTP/2-Client befindet sich seit Java 9 [HTTP] in der Incubation-Phase. Er bietet eine vollständige HTTP/2-Unterstützung [HTTP2] direkt im JDK. Außerdem wurden die APIs für HTTP/1.1 überarbeitet. Die Java-Entwickler schrieben, dass es einfacher gewesen sei, eine komplett neue API zu entwickeln, als die neuen HTTP/2-Features in die bestehende HTTP-API zu integrieren. Final veröffentlicht werden soll dieser HTTP/2-Client mit Java 11. [HTTP, JDOC9, JDK9]
Process-API	Primär ging es den Java-Entwicklern darum, das Management von Prozessen betriebssystemunabhängig einfacher zu gestalten. So lassen sich die Prozesse nun über die ProcessAPI durchlaufen. Außerdem lassen sich einige Kerneigenschaften, wie etwa die PID, der Prozessname, der User, der den Prozess gestartet hat, die Ressourcenauslastung, etc. für jeden Prozess abrufen. [PROC, JDOC9, JDK9]
Flow-API	Mit Java 9 kam zu den bereits bestehenden Concurrency-APIs ein kleines Publish-Subscribe Framework hinzu. Dieses ist in der Klasse `java.util.concurrent.Flow` verpackt, welche zunächst einmal grundlegende Interfaces definiert. Dazu gehören ein Publisher, ein Subscriber, ein Processor und ein Subscription-Objekt. Außerdem gibt es mit der Klasse `SubmissionPublisher` bereits eine Standardimplementierung zum Versand von Objekten. [FLOW, JDOC9, JDK9]
Collection Factory Methoden	Es gibt einige neue Factory Methoden, mit denen sich nun Listen, Sets und Hashmaps in einer Zeile erstellen lassen. Dafür genügt beispielsweise der Aufruf `List.of("a", "b", "c")`. Zurück kommt eine unveränderbare Liste mit den Elementen "a", "b" und "c". [COLL, JDOC9]
Completeable Futures	Die Klasse `CompleteableFuture` hat die neue Methode `completeOnTimeout(...)` bekommen. Diese erhält als Parameter ein Timeout und eine `TimeUnit`. Nach Ablauf des Timeouts wird eine `TimeoutException` geworfen. Außerdem gibt es nun eine `copy()` -Methode, mit der sich defensive Kopien eines Futures anlegen lassen. Der kopierte Future wird dabei completed, sobald der originale Future completed wird. Der originale Future wird dagegen nicht completed, wenn die `complete(...)` -Methode des kopierten Futures aufgerufen wird. [COMP, JDOC9, JDK9]

SafeVarargs-Annotation	Mit der `@SafeVarargs` -Annotation lassen sich Warnungen unterdrücken, wenn der Entwickler sicher ist, dass bei der Ausführung des betreffenden Codes keine Exceptions auftreten. Diese Annotation konnte bisher nur auf `static` oder `final` Methoden verwendet werden. Mit Java 9 dürfen auch private Instanzmethoden mit `@SafeVarargs` annotiert werden. [SAFE, JDOC9, JDK9]
Neue Streams Methoden	Auch bei der Streams-API sind mit Java 9 einige neue Methoden dazugekommen. Dazu gehören zunächst einmal die `takeWhile()` -/ `dropWhile()` -Methoden. Die `dropWhile()` -Methode verwirft so lange Elemente, wie die Bedingung nicht erfüllt ist. Sobald diese einmal erfüllt wurde, gibt sie alle Elemente weiter und prüft die Bedingung nicht mehr. die `takeWhile()` -Methode funktioniert ganz ähnlich. Hierbei werden so lange Elemente zurückgegeben, bis die Bedingung nicht länger erfüllt ist. Außerdem gibt es noch eine neue Überladung für die `iterate(...)` -Methode, mit der das Verhalten einer For-Schleife simuliert werden kann. Schließlich wurde noch die `ofNullable(...)` -Methode hinzugefügt, welche dazu dient, Nullchecks zu vermeiden. Dazu werden die `null` -Elemente herausgefiltert, bevor sie im Stream landen. [STRM, JDOC9, JDK9]

Tabelle 1: Änderungen an den Standardbibliotheken

NEUE TOOLS UND FEATURES

Außerdem wurden noch einige neue Tools und Features zum JDK hinzugefügt. Auch zu jedem dieser Tools und Features gibt es eine weiterführende mehrseitige Erläuterung. Hier eine kurze Liste:

JShell	Java besitzt nun eine eigene Read-Evaluate-Print-Loop-Shell (REPL). Auf dieser lassen sich Java Code-Snippets ausführen. Dazu muss nicht einmal eine Klasse erstellt werden. Die Java-Ausdrücke können einfach eingegeben und ausgewertet werden. Nicht einmal ein Semikolon am Zeilenende ist nötig. [JSHL]
JLinker	Mit dem JLinker lassen sich nun individuelle JRE-Images für eine konkrete Anwendung erstellen. Dank des Modulsystems sind in diesen Images dann nur noch die Module zu finden, die die spezifizierte Anwendung auch benötigt. Das spart Platz und senkt die initiale Startupzeit der JVM beim Programmstart. [JLNK]
Java für alte Plattformen kompilieren	Es gibt ein neues Compilerflag `--release`, welches die bereits bestehenden Flags `-target` und `-source` ersetzen soll. Während bei `-target` und `-source` nur die Version des Java-Compilers angepasst wurde, wird bei der Verwendung von `--release` auch gegen die korrekte Version der Java Klassenbibliotheken kompiliert. [JOLD]
G1 als Standard Garbage Collector	Der G1 Garbage Collector ist seit Java 9 nun der Standard Garbage Collector im JRE und JDK. Er ersetzt den Parallel GC. Der G1 soll insbesondere Heapgrößen von über 6GB besser verarbeiten können und soll besser planbare Pausenzeiten beinhalten. Hierbei geht es darum, die Latenz zu senken, unter der Annahme, dass geringe Pausenzeiten für die Anwendung wichtiger als maximaler Durchsatz sind. Dieser Vorteil wird erkauft durch einen etwas höheren CPU-Overhead. [JGC1]

Tabelle 2: Neue Tools und Features im JDK 9

DAS NEUE RELEASE MODELL

Nachdem die Abstände zwischen den letzten Java Major-Releases vor Java 9 häufig mehrere Jahre überspannten und die Releases dann kurz vor dem Release-Termin noch einmal um Monate oder Jahre verschoben wurden, gibt es nun ein neues Release-Modell. [JUPD]

Statt dem bisherigen featurebasierten Release-Modell (Feature wird releast sobald fertig) wurde nun auf ein Release-Train-Modell gewechselt. Dieses sieht neue Major Releases in einem festen Takt alle sechs Monate vor. Wenn ein Feature bis dahin nicht fertig ist, wird es automatisch in die nächste Release-Version verschoben. [JDSK]

Der Vorteil dieses neuen Modells löst das Problem, dass ein großes Feature (wie beispielsweise das Modulsystem) den Release einer neuen Version und aller anderen für diese Version geplanten Features massiv verzögern kann, nur um dann komplett aus dem Release gestrichen zu werden. Neuerdings können fertige Features einfach mit dem nächsten Major-Release veröffentlicht werden und müssen nicht jahrelang fertig in der Release-Queue warten.

Oracle liegt damit im Trend und setzt auf ein Release-Modell, welches sich bereits im Webbrowsermarkt erfolgreich durchgesetzt hat. Kleine Bugfixes und die Korrektur von Sicherheitsproblemen wird es natürlich auch zwischen den Releases in gewöhnlichen Bugfix-Releases geben.

Da es nun also immer zwei Major-Releases pro Jahr gibt, wird nicht mehr jeder Major-Release auch gleichzeitig ein LTS-Release (Long Term Support Release) sein. Java 8 war der letzte LTS und der nächste LTS ist erst Java 11. Dadurch ist es nur begrenzt sinnvoll, im Unternehmensumfeld die Java-Version von Java 8 auf Java 9 oder 10 zu aktualisieren, da ein Update auf Java 11 mangels Long-Term-Support der Zwischenversionen direkt eingeplant werden sollte.

Der normale Support für eine Java Major-Version dauert jeweils bis zum nächsten Major-Release an. Damit ist der Support für Java 9 mit dem Release von Java 10 im März 2018 bereits eingestellt worden; Der Support für Java 10 endet im September mit dem Release von Java 11. Nach Ablauf des Supportzeitraums gibt es keine öffentlichen Updates mehr für die betroffene Java-Version. Öffentliche Updates für Java 8 wird es laut dem aktuellen Release Plan auf der Website von Oracle noch bis voraussichtlich 2020 geben. Wie lange der LTS von Java-Version 11 läuft, ist bisher nicht bekannt. [OSRM]

AUSBLICK: JAVA 10 UND 11

LOCAL TYPE INFERENCE

Im März 2018 ist bereits Java 10 erschienen. Das Kernfeature von Java 10 ist ein Feature namens "Local Type Inference" [JDOC10]. Diesen Mechanismus gibt es in Java bereits bei den Lambda-Parametern und beim Diamond-Operator für generische Datencontainer.

Neu ist, dass nun mit dem Schlüsselwort `var` auch lokale Variablen definiert werden können. Deren Datentyp ergibt sich aus der Zuweisung eines Wertes. Beim Diamond-Operator wird der Typ des Generics über den Typ der deklarierten Variable ermittelt. In Fall von `var` ist es genau umgekehrt: Der Typ der Variable wird von der rechten Seite der Zuweisung abgeleitet. [RJ10]

```
var str = "Hello World";
var i = 5;
```

Beispiel 19: Deklaration einiger generischer Variablen

Der Datentyp ist zwar beliebig, aber fest. Dies bedeutet, dass er sich nach der initialen Deklaration nicht mehr ändern lässt. Anschließend an das Codebeispiel oben (Deklaration einiger generischer Variablen) könnte jetzt also beispielsweise nicht die Zeile `str = 5l;` folgen. Diese würde in einem Compilerfehler resultieren. Was möglich wäre, ist eine Veränderung unter Beibehaltung des Datentyps, etwa `str = "Goodbye World";`.

Das Ganze funktioniert auch mit Containerdatentypen:

```
var strings = List.of("a", "b", "c");
strings.add("d");
for (var string : strings) {
   System.out.println(string);
}
```

WAS MIT JAVA 11 ENTFERNT WIRD

Dank des mit Java 9 eingeführten Modulsystems ist es nun weitaus einfacher für Oracle, auch größere Teile aus dem JDK und der JRE zu entfernen und separat anzubieten. Hier eine tabellarische Übersicht, welche Module mit Java 11 entfernt werden sollen:

JavaFX	Die 2007 veröffentlichte Oberflächenbibliothek JavaFX ist seit 2011 Open Source und seit 2012 im Oracle JDK. Ab Java 11 wird sie nicht mehr im JDK sein, sondern entkoppelt als separater Download angeboten. Dadurch soll die OpenJFX-Community gestärkt werden. [JFX, OJFX, JFXR]
JAX-WS	Seit Java 9 sind sämtliche Module der Java Standard Edition (JSE), welche Technologien aus der Java Enterprise Edition (JEE) enthalten, bereits mit @Deprecated annotiert. Diese Module werden mit Java 11 nicht mehr Teil des Java SE JDK sein. Dazu gehört auch die Java API for XML-Based Web Services (JAX-WS). Sie sollte die Erstellung von Client-Server Webservices in Java vereinfachen. [JXWS]
JAXB	Die Java Architecture for XML Binding (JAXB) bietet ein Framework, mit dem beispielsweise XML-Daten aus einer Instanz eines XML-Metamodels (XML-Schema) direkt an Java-Klassen gebunden werden können. Auch lassen sich Java Klassen direkt aus dem XML-Schema generieren. [JAXB]
JAF	Das Java Beans Activation Framework (JAF) wird ebenfalls mit Java 11 aus dem Java SE JDK entfernt. Die Hauptaufgabe dieses Frameworks ist es, den Datentyp eines beliebigen Datenobjektes festzustellen und den Zugriff darauf zu kapseln. Anschließend kann das Framework mögliche Operationen, die auf dem Objekt ausgeführt werden könnten, finden und eine passende Bean instanziieren, die diese Operationen ausführen könnte. [JABE]
Common Annotations	Das Modul Common Annotations enthält einige allgemeine Annotationen, wie @PostConstruct, @PreDestroy oder @Resource, welche in anderen (Enterprise-) Frameworks wie etwa Spring verwendet werden und als Orientierungshilfe dienen sollten. [CMAN]
CORBA	Auch die Common Object Request Architecture (CORBA) wird ab Java 11 nicht mehr im JSE JDK enthalten sein. CORBA ist die Spezifikation einer objektorientierten Middleware mit Object Request Broker. [CORB]
JTA	Die Java Transaction API (JTA) war in der Java Standard Edition nie vollständig enthalten. Es wurde nur in Teilen aus der JEE in die JSE übertragen. Diese Teile werden nun wieder entfernt. [JTA] Im Kern geht es bei der JTA darum, mehrere unterschiedliche Systeme (etwa einen Datenbankserver, einen Anwendungsserver, ein Messagingsystem, etc.) innerhalb einer einzelnen Transaktion ansprechen zu können.
Java-EE-bezogene Tools	Alle Tools, die zu den nun entfernten Java-EE Modulen gehören, werden ebenfalls aus dem Java SE JDK genommen. Dazu zählen wsgen und wsimport, die zur Ressourcengeneration auf Basis einer WSDL im Zusammenhang mit JAX-WS verwendet wurden. Ebenso wird es schemagen und xjc nicht mehr geben, mit denen aus einem bestehenden Java-Klassengraphen ein XML-Schema, sowie aus dem XML die Java Klassen generiert werden können. Schließlich werden der indlj (IDL to Java Compiler), sowie die CORBA-bezogenen Tools orbd, servertool und tnamesrv aus dem JDK entfernt.

Tabelle 3: Was mit Java 11 entfernt wird

ZUSAMMENFASSUNG

Das Modulsystem in Java 9 soll die Performance und die Sicherheit und Integrität von Java-Programmen verbessern. Durch die Java Module wurde quasi eine weitere Sichtbarkeitsschicht oberhalb von `public` hinzugefügt, mit der sich neue Projekte nun noch besser organisieren lassen. Einzig die Modulversionierung lässt zu wünschen übrig. Dafür lassen sich jetzt JAR-Dateien erstellen, die gleichzeitig mehrere Java-Versionen unterstützen können. Dadurch wird eine stärkere Codeoptimierung, insbesondere für neue Java-Versionen, ermöglicht.

Auch an den Standardbibliotheken wurde viel geändert und aktualisiert. So ist im JDK nun ein vollständiger HTTP/2-Client enthalten und die Process-API ist nun signifikant leichter zu benutzen. An vielen Stellen wurden kleinere Optimierungen vorgenommen, die das Potenzial haben, den Programmiereralltag deutlich zu vereinfachen.

Die neue JShell soll die Abwendung von Bildungseinrichtungen von Java stoppen. Das Resultat bleibt abzuwarten. Der JLinker ist ein hilfreiches Tool, mit dem sich die Stärken des Modulsystems im Hinblick auf Abwärtsskalierbarkeit und Downsizing vollständig ausspielen lassen.

Abschließend hat die Änderung des Release Modells hin zu regelmäßigen Releases und weniger LTS-Versionen signifikante Implikationen für zukünftige Java-Versionen. So enthält Java 10 außer den Local Type Inferences keine signifikant großen Features und auch die Pre-Release Versionen von Java 11 enthalten momentan kaum mehr als die Entfernung von JavaFX und der Java-Enterprise Module bereit.

REFERENZEN UND WEITERFÜHRENDE MATERIALIEN

- [HTTP] HTTP/2 Client – Java 9
  Jacobs, Steffen , Orientation in Objects GmbH
  2016-08-24
  OIO Blog (https://blog.oio.de/2016/08/24/http2-client-java-9/)
- [PROC] Process API – Java 9
  Jacobs, Steffen , Orientation in Objects GmbH
  2016-09-02
  OIO Blog (https://blog.oio.de/2016/09/02/process-api-java-9/)
- [HTTP2] Hypertext Transfer Protocol Version 2 (HTTP/2)
  Belshe, M. , Bitgo; Peon, R. , Google, Inc; Thomson, M. Ed. , Mozilla
  2015-05-01
  Internet Engineering Task Force (IETF) (https://tools.ietf.org/html/rfc7540)
- [FLOW] Publish-Subscribe mit der Flow-API in Java 9
  Jacobs, Steffen , Orientation in Objects GmbH
  2018-05-04
  OIO Blog (https://blog.oio.de/2018/05/04/publish-subscribe-mit-der-flow-api-in-java-9/)
- [JDOC9] Java® Platform, Standard Edition and Java Development Kit Version 9 API Specification
  Oracle, , Oracle, Inc.
  2017-10-17
  JavaDoc Java 9 (https://docs.oracle.com/javase/9/docs/api/overview-summary.html)
- [COLL] Collections – Factory Methoden
  Jacobs, Steffen , Orientation in Objects GmbH
  2018-05-09
  OIO Blog (https://blog.oio.de/2018/05/09/collections-factory-methoden/)
- [COMP] Completable Futures: Java 9 Update
  Jacobs, Steffen , Orientation in Objects GmbH
  2018-05-07
  OIO Blog (https://blog.oio.de/2018/05/07/completable-futures-java-9-update/)
- [SAFE] SafeVarargs-Annotation in Java
  Jacobs, Steffen , Orientation in Objects GmbH
  2018-05-03
  OIO Blog (https://blog.oio.de/2018/05/03/safevarargs-annotation-in-java/)
- [STRM] Streams – Neue Methoden in Java 9
  Jacobs, Steffen , Orientation in Objects GmbH
  2018-05-15
  OIO Blog (https://blog.oio.de/2018/05/15/streams-neue-methoden-in-java-9/)
- [JSHL] JShell
  Jacobs, Steffen , Orientation in Objects GmbH
  2016-08-25
  OIO Blog (https://blog.oio.de/2016/08/25/jshell/)
- [JLNK] Individuelle JRE-Images mit dem JLinker erstellen
  Jacobs, Steffen , Orientation in Objects GmbH
  2018-05-14
  OIO Blog (https://blog.oio.de/2018/05/14/individuelle-jre-images-mit-dem-jlinker-erstellen/)
- [JOLD] Java für alte Plattformen kompilieren
  Jacobs, Steffen , Orientation in Objects GmbH
  2018-04-25
  OIO Blog (https://blog.oio.de/2018/04/25/java-fur-alte-plattformen-kompilieren/)
- [JGC1] The G1 Garbage Collector
  Jacobs, Steffen , Orientation in Objects GmbH
  2016-08-30
  OIO Blog (https://blog.oio.de/2016/08/30/the-g1-garbage-collector/)
- [JFX] Java FX
  Oracle, , Oracle, Inc.
  2012-08-15
  Oracle Technetwork JavaFX (http://www.oracle.com/technetwork/java/javafx/overview/index.html)
- [OJFX] OpenJDK Java FX
  OpenJDK, , OpenJDK
  2011-11-06
  GitHub Repository OpenJDK Java FX (https://github.com/javafxports/openjdk-jfx)
- [JFXR] The Future of JavaFX and Other Java Client Roadmap Updates
  Donald Smith, , Oracle

2018-03-07
Oracle Blog (https://blogs.oracle.com/java-platform-group/the-future-of-javafx-and-other-java-client-roadmap-updates)

- [JXWS] JSR 224: JavaTM API for XML-Based Web Services (JAX-WS) 2.0
  JCP, , Java
  2006-05-11
  JCP JAX-WS (https://jcp.org/en/jsr/detail?id=224)
- [JAXB] JSR 222: JavaTM Architecture for XML Binding (JAXB) 2.0
  JCP, , Java
  2006-05-11
  JCP JAXB (https://jcp.org/en/jsr/detail?id=222)
- [JABE] JSR 925: JavaBeansTM Activation Framework 1.1
  JCP, , Java
  2006-05-11
  JCP JavaBeans (https://jcp.org/en/jsr/detail?id=925)
- [CMAN] JSR 250: Common Annotations for the JavaTM Platform
  JCP, , Java
  2006-05-11
  JCP Common Annotations (https://jcp.org/en/jsr/detail?id=250)
- [CORB] Common Object Request Broker Architecture: Core Specification
  OMG, , Object Management Group, Inc.
  2012-03-04
  CORBA Spezifikation (www.omg.org/cgi-bin/doc?formal/04-03-12.pdf)
- [JTA] JSR 907: JavaTM Transaction API (JTA)
  JCP, , Java
  2002-11-06
  JCP JTA (https://jcp.org/en/jsr/detail?id=907)
- [JDK9] OpenJDK Java 9 Release Notes
  OpenJDK, , OpenJDK
  2017-09-21
  JDK 9 Release Notes (http://openjdk.java.net/projects/jdk9/)
- [OSRM] Oracle Java SE Support Roadmap
  Oracle, , Oracle, Inc.
  2018-03-05
  Java SE Roadmap (http://www.oracle.com/technetwork/java/javase/eol-135779.html)
- [JUPD] JDK Updates nach Java 9 – Oracle stellt neues Release-Train-Modell in Aussicht
  Maier, Thorsten , Orientation in Objects GmbH
  2017-09-08
  OIO Blog (https://blog.oio.de/2017/09/08/jdk-updates-nach-java-9-oracle-stellt-neues-release-train-modell-in-aussicht/)
- [JDSK] There's not a moment to lose!
  Reinhold, Mark ,
  2017-09-06
  Diskussion zu neuem Release Plan (https://mreinhold.org/blog/forward-faster)
- [JDOC10] Java® Platform, Standard Edition and Java Development Kit Version 10 API Specification
  Oracle, , Oracle, Inc.
  2018-03-20
  JavaDoc Java 10 (https://docs.oracle.com/javase/10/docs/api/overview-summary.html)
- [MRJ] JEP 238: Multi-Release JAR Files
  Sandoz, Paul ,
  2014-06-08
  JEP 238 (http://openjdk.java.net/jeps/238)
- [OSGI] The Dynamic Module System for Java
  OSGi, , OSGi Alliance
  2018-05-22
  Website OSGi Alliance (https://www.osgi.org/)
- [JGSW] Jigsaw – Java 9
  Jacobs, Steffen , Orientation in Objects GmbH
  2016-08-26
  OIO Blog (https://blog.oio.de/2016/08/26/jigsaw-java-9/)
- [MCRS] Microservices - a definition of this new architectural term
  Fowler, Martin , ; Lewis, James , ThoughtWorks
  2014-03-25
  Martin Fowler Blog (https://martinfowler.com/articles/microservices.html)
- [JMOD] JEP 200: The Modular JDK
  Reinhold, Mark ,
  2014-07-22

JEP 200 (https://blog.oio.de/2016/08/26/jigsaw-java-9/)

- [JGSWO] Project Jigsaw
  OpenJDK, ,
  2014-07-22
  Project Jigsaw (http://openjdk.java.net/projects/jigsaw/)
- [JGSWQS] Project Jigsaw: Module System Quick-Start Guide
  OpenJDK, ,
  2014-07-22
  Project Jigsaw Quickstart Guide (http://openjdk.java.net/projects/jigsaw/quick-start)
- [REFL] The Java™ Tutorials - Trail: The Reflection API
  Oracle, , Oracle, Inc.
  2014-07-22
  Reflection API (https://docs.oracle.com/javase/tutorial/reflect/)
- [SMOD] The State of the Module System
  Mark, Reinhold , Oracle, Inc.
  2018-03-08
  The State of the Module System (http://openjdk.java.net/projects/jigsaw/spec/sotms/#the-module-path)