8 Neues und Änderungen in Java 12

Bei Oracle folgt man mittlerweile der Strategie, Java in kleinen, aber feinen Iterationen um nützliche Funktionalität zu ergänzen und auch bei der Syntax zu modernisieren. Zudem soll ein sechsmonatiger Releasezyklus eingehalten werden. Demnach erfolgte die Veröffentlichung von Java 12 im März 2019. Für uns als Entwickler finden sich folgende interessante Erweiterungen in JDK 12:

- Switch Expressions (Preview) (`https://openjdk.java.net/jeps/325`)
- Microbenchmark Suite (`https://openjdk.java.net/jeps/230`)
- Einige kleinere Neuerungen und Anpassungen in den APIs

Auf die Switch Expressions gehe ich im Folgenden nicht näher ein, da eine aktualisierte und leicht modifizierte Variante wiederum als Preview Einzug in Java 13 gefunden hat und final in Java 14 enthalten ist. Diese Neuerung wird in dem dazu korrespondierenden Kapitel 9 beschrieben.

8.1 Microbenchmark Suite

Mitunter sind einige Teile der eigenen Software nicht so performant, wie man es sich erhofft hat bzw. wie es vom Nutzer erwartet wird. Zur Optimierung der Performance gibt es verschiedene Hilfsmittel. Generell sollte man zunächst gründlich messen und nur mit Bedacht und vor allem nicht direkt optimieren. Das gilt unter anderem deshalb, weil zum einen die JVM bereits diverse Optimierungen eingebaut hat und zum anderen man rein auf Basis von Vermutungen häufig falsch liegt und so nur Programmteile schlechter les- und wartbar macht, jedoch die Gesamtperformance kaum oder überhaupt nicht verbessert.

Als Abhilfe gibt es oftmals verschiedene Möglichkeiten. Manchmal reicht ein anderer Algorithmus und manchmal kann Caching helfen. Immer jedoch empfiehlt es sich, die für die Performance kritischen Stellen zu ermitteln und auch nur dort zu optimieren. Diese Aussage gilt für die Architektur und das Design bis hin zur Low-Level-Implementierung. Details zum Thema Optimierung finden Sie in meinem Buch »Der Weg zum Java-Profi« [4].

Wie schon erwähnt, sollte man die kritischen Stellen keinesfalls nur aufgrund von Vermutungen identifizieren, sondern basierend auf Messungen. Dazu kann man beispielsweise einfache Start-Stopp-Messungen einsetzen, empfehlenswerter sind ausgeklügeltere Verfahren mit mehreren Durchläufen, um temporäre Störeffekte, verursacht durch höhere Systemlast oder von Garbage Collections, zu vermeiden. Dabei hilft die mit Java 12 in das JDK aufgenommene Microbenchmark Suite, die auf dem Toolkit Java Microbenchmark Harness (JMH) basiert.

8.1.1 Eigene Microbenchmarks und Varianten davon

Nachfolgend schauen wir uns die Optimierung auf der Ebene von einzelnen bzw. wenigen Anweisungen an. Dazu nutzt man sogenannte Microbenchmarks. Bedenken Sie bitte aber vorab, dass derartige Optimierungen nur dann sinnvoll sind, wenn auf den höheren Ebenen wie Architektur, Design und Algorithmus keine Verbesserungen mehr erreicht werden können oder die zu optimierenden Anweisungen extrem häufig ausgeführt werden und somit die Gesamtperformance maßgeblich beeinflussen.

Nehmen wir an, wir wollten Aussagen über die Performance für einige Varianten einer Implementierung treffen. Grundsätzlich ist das nicht so einfach, wie man zunächst annehmen könnte, da die Messungen unter möglichst gleichen Bedingungen (CPU-Last, Speicherverbrauch usw.) geschehen sollten, damit man vergleichbare Resultate erhält. Aber auch Optimierungsmechanismen in der JVM wie Loop-Unrolling, Hotspot-Kompilierung usw. führen potenziell dazu, dass sich Messwerte signifikant voneinander unterscheiden.

Einfache Start-Stopp-Messungen

Im einfachsten Fall nutzt man eine Art Stoppuhr, indem man den zu messenden Programmteil mit Aufrufen von `System.currentTimeMillis()` wie folgt umklammert und die Differenz zwischen den Werten ermittelt:

```
// ACHTUNG: keine gute Variante
final long startTimeMs = System.currentTimeMillis();

someCodeToMeasure();

final long stopTimeMs = System.currentTimeMillis();
final long duration = stopTimeMs - startTimeMs;
```

Das scheint vernünftig. Jedoch nutzt `System.currentTimeMillis()` – zumindest unter Windows – ungenaue, zu grobgranulare Zeitgeber des Betriebssystems. Verlässlichere Werte erhält man mithilfe der folgenden Methode:

```
System.nanoTime()
```

Wiederholte Start-Stopp-Messungen

Unabhängig vom Zeitgeber liefern einfache Start-Stopp-Messungen eher unzuverlässige Ergebnisse, da die Messungen durch unterschiedlichste Effekte verfälscht werden können. Deswegen ist es oftmals sinnvoller, eine gewisse Anzahl an Iterationen zu nutzen und dann den Durchschnittswert als Referenzmesswert zu errechnen:

```
// bessere Variante
final long startTimeNs = java.lang.System.nanoTime();

for (int i = 0; i < MAX_ITERATIONS; i++)
{
   someCodeToMeasure();
}

final long stopTimeNs = java.lang.System.nanoTime();

final long avgDurationNs = (stopTimeNs - startTimeNs) / MAX_ITERATIONS;
```

Diese Variante mit mehreren Durchläufen und Durchschnittsbildung ist weniger anfällig für Systemlastschwankungen oder sonstige Störeinflüsse. Zudem ist es recht einfach möglich, auch Minimal- und Maximaldauer oder die Standardabweichung zu ermitteln. Bei diesen erweiterten Anforderungen lagert man diese Funktionalität aber sinnvollerweise in eine oder mehrere separate Klassen aus.

Wiederholte Start-Stopp-Messungen mit Warm-up

Neben kleineren externen Störungen beobachtet man immer mal wieder Einschwing-Effekte: Erst nach einer gewissen Anzahl an Durchläufen zeigt eine Funktionalität ihre optimale Laufzeit: Das kann etwa dadurch verursacht werden, dass Caches zunächst gefüllt werden oder aber der Hotspot-Optimierer verschiedene Codestellen in Maschinensprache übersetzt. Für diese Fälle empfiehlt sich dann ein Warm-up vor der eigentlichen Messung.

Nachfolgend zeige ich exemplarisch ein der eigentlichen Performance-Messung vorangestelltes Warm-up in Form einiger Schleifendurchläufe mit Aufrufen an die später zu messende Funktionalität:

```
// noch bessere Variante der Messung durch Warm-up
for (int i = 0; i < MAX_WARMUP_ITERATIONS; i++)
{
   someCodeToMeasure();
}

// eigentliche Messung nach Warm-up
final long startTimeNs = java.lang.System.nanoTime();

for (int i = 0; i < MAX_ITERATIONS; i++)
{
   someCodeToMeasure();
}
final long stopTimeNs = java.lang.System.nanoTime();

final long avgDurationNs = (stopTimeNs - startTimeNs) / MAX_ITERATIONS;
```

Man erkennt gut, dass die Implementierung der Messung immer unhandlicher wird. Das ist oftmals ein Zeichen dafür, dass man nach einem unterstützenden Framework Ausschau halten sollte. Glücklicherweise gibt es das JMH, das wir nun genauer kennenlernen.

Tipp: Hotspot-Optimierer

Wenn man ein Java-Programm startet, so liegt dieses zunächst in Bytecode vor. Dieser wird von der JVM interpretiert. Obwohl dies in den letzten Jahren immer performanter geworden ist, reicht die Ausführungsgeschwindigkeit interpretierten Bytecodes natürlich nicht ganz an die von Maschinensprache heran. In die JVM ist der Hotspot-Optimierer inklusive eines JIT-Compilers (JIT = Just In Time) integriert. Beide sorgen dafür, dass besonders häufig durchlaufene Teile des Bytecodes erkannt und on the fly in Maschinensprache transferiert werden.

Aufgrund dieser Besonderheit betrachtet man bei naiven Messungen der Performance eventuell nur die interpretierte Ausführung, gegebenenfalls aber auch eine Mischform, wenn bereits ein Teil des Bytecodes in Maschinensprache übersetzt wurde.

Mithilfe des Aufrufparameters `-XX:+PrintCompilation` kann man die Kompilierung protokollieren lassen und `-XX:CompileThreshold` erlaubt es, die Anzahl der benötigten Ausführungen eines Programmstücks bis zu dessen Kompilierung festzulegen. Standardmäßig benötigt es in der seit Java 11 nur noch vorhandenen Servervariante 10.000 Abarbeitungen – die früher verfügbare Client-JVM hat bereits nach 1.500 Abarbeitungen des Bytescodes kompiliert.

8.1.2 Microbenchmarks mit JMH

Seit Java 12 ist das Open-Source-Framework Java Microbenchmark Harness (JMH) ins JDK aufgenommen worden und adressiert die oben angedeuteten sowie weitere Schwierigkeiten bei Performance-Messungen.

Einstieg in JMH

JMH arbeitet mit Annotations und integriert basierend darauf verschiedene Messungen. Hierzu bietet sich die Nutzung eines Maven-Archetypes an. Folgender Aufruf erzeugt ein neues Verzeichnis mit einem vollständig initialisierten JMH-Benchmark:

```
mvn archetype:generate \
    -DinteractiveMode=false \
    -DarchetypeGroupId=org.openjdk.jmh \
    -DarchetypeArtifactId=jmh-java-benchmark-archetype \
    -DgroupId=org.sample \
    -DartifactId=jmh-test \
    -Dversion=1.0-SNAPSHOT
```

Schauen Sie doch kurz in Anhang C, um etwas mit Maven vertraut zu werden.

Dadurch entstehen folgende Verzeichnisse und Dateien:

```
.
`-- jmh-test
    |-- pom.xml
    `-- src
        `-- main
            `-- java
                `-- org
                    `-- sample
                        `-- MyBenchmark.java
```

Als Grundgerüst wird eine Klasse `MyBenchmark` erzeugt, weshalb sich dort auch auch ein englischer Kommentar findet. In der generierten Klasse können wir nun die zu beobachtenden Funktionalitäten aufrufen. Ähnlich zu JUnit lassen sich die gewünschten Einstellungen zu den Messungen per Annotations vorgeben. Die wichtigste ist `@Benchmark`, die eine zu untersuchende Methode kennzeichnet:

```
public class MyBenchmark
{
    @Benchmark
    public void testMethod()
    {
        // This is a demo/sample template for building your JMH benchmarks.
        // Edit as needed. Put your benchmark code here.
    }
}
```

Um dann den Benchmark auszuführen, muss man in das durch den obigen Aufruf erzeugte Verzeichnis wechseln und dort – wie von Maven gewohnt – kompilieren und paketieren:

```
mvn package
```

Dadurch entsteht ein entsprechendes Artefakt `benchmarks.jar` im Verzeichnis `target`. Weil dort insbesondere alle benötigten Klassen und Bibliotheken bereits enthalten sind, kann man das JAR ohne weitere Angaben folgendermaßen starten:

```
java -jar target/benchmarks.jar
```

Da wir bislang noch keine sinnvolle Funktionalität, sondern nur eine leere Methode messen würden, heben wir uns den Start für das nächste Beispiel auf.

Der erste Benchmark in JMH

Wir haben gerade gesehen, wie man eine Performance-Suite mit JMH erzeugt. Allerdings weiß das Framework ja noch nicht, was gemessen werden soll. Deshalb wird auch nur eine Klasse `MyBenchmark` als Grundgerüst erzeugt. Dort können wir die zu beobachtenden Funktionalitäten aufrufen und die gewünschten Messeinstellungen per Annotations vorgeben. Generell empfiehlt es sich aber, einen sprechenderen Namen für

die Benchmark-Klasse zu nutzen, nachfolgend etwa `FirstIntegerBenchmark`, der die Performance der Methoden `toHexString()` und `toBinaryString()` misst.

Insgesamt wird das Schreiben von Microbenchmarks ziemlich einfach: Praktischerweise kann man zwar eine recht feingranulare Steuerung vornehmen, aber dem Gedanken *Convention over Configuration* folgend ist es für erste Messungen oftmals ausreichend, wenn man lediglich die Annotation `@Benchmark` wie folgt nutzt – die Zahlen sind mithilfe des Unterstrichs an den Positionen der Tausenderpunkte etwas besser lesbar:

```
public class FirstIntegerBenchmark
{
    @Benchmark
    public String numberAsHexString()
    {
        final int number = 1_234_567;
        return Integer.toHexString(number);
    }

    @Benchmark
    public String numberAsBinaryString()
    {
        final int number = 123_456_789;
        return Integer.toBinaryString(number);
    }
}
```

Nachdem wir beide Klassen kompiliert und paketiert haben, starten wir das ausführbare JAR. Es ergeben sich in etwa folgende gekürzten Konsolenausgaben:

```
# JMH version: 1.23
# VM version: JDK 14, OpenJDK 64-Bit Server VM, 14+36-1461
# VM invoker: /Library/Java/JavaVirtualMachines/jdk-14.jdk/Contents/Home/bin/
      java
# VM options: <none>
# Warmup: 5 iterations, 10 s each
# Measurement: 5 iterations, 10 s each
# Timeout: 10 min per iteration
# Threads: 1 thread, will synchronize iterations
# Benchmark mode: Throughput, ops/time
# Benchmark: org.sample.FirstIntegerBenchmark.numberAsBinaryString

# Run progress: 0.00% complete, ETA 00:16:40
# Fork: 1 of 5
# Warmup Iteration   1: 56326650.457 ops/s
# Warmup Iteration   2: 59780600.908 ops/s
# Warmup Iteration   3: 64051264.973 ops/s
# Warmup Iteration   4: 64008812.168 ops/s
# Warmup Iteration   5: 64186426.089 ops/s
Iteration   1: 61304141.790 ops/s
Iteration   2: 61953075.846 ops/s
Iteration   3: 63036185.662 ops/s
Iteration   4: 64322247.337 ops/s
Iteration   5: 64872604.082 ops/s
...
```

Nach längerer Ausführung erhält man dann ungefähr folgende Ergebnisse:

```
Result "org.sample.FirstIntegerBenchmark.numberAsBinaryString":
  64224040.592 ±(99.9%) 978073.744 ops/s [Average]
  (min, avg, max) = (61298595.526, 64224040.592, 65767851.332), stdev =
       1305700.466
  CI (99.9%): [63245966.848, 65202114.336] (assumes normal distribution)

Result "org.sample.FirstIntegerBenchmark.numberAsHexString":
  145802842.871 ±(99.9%) 1765144.419 ops/s [Average]
  (min, avg, max) = (140986090.679, 145802842.871, 149109246.667), stdev =
       2356417.299
  CI (99.9%): [144037698.452, 147567987.290] (assumes normal distribution)
```

Möchte man den Benchmark noch ein wenig feintunen, beispielsweise die Anzahl an Iterationen oder das Warm-up und die Anzahl an Messdurchgängen konfigurieren, so könnte man dies folgendermaßen mithilfe der Annotations `@Warmup` sowie `@Measurement` tun:

```
import org.openjdk.jmh.annotations.Measurement;
import org.openjdk.jmh.annotations.Warmup;
import org.openjdk.jmh.annotations.Fork;

@Measurement(iterations = 3, time = 1000, timeUnit = TimeUnit.MILLISECONDS)
@Warmup(iterations = 7, time = 1000, timeUnit = TimeUnit.MICROSECONDS)
@Fork(4)
public class MyBenchmark
{
     ....
}
```

Wendet man diese Werte auf den ersten Benchmark an, dann ändert sich die Ausgabe wie folgt:

```
# Benchmark: org.sample.FirstIntegerBenchmark.numberAsBinaryString

# Run progress: 0.00% complete, ETA 00:33:44
# Fork: 1 of 4
# Warmup Iteration   1: 2612321.666 ops/s
# Warmup Iteration   2: 6316193.097 ops/s
# Warmup Iteration   3: 7323288.665 ops/s
# Warmup Iteration   4: 4430635.983 ops/s
# Warmup Iteration   5: 6554600.103 ops/s
# Warmup Iteration   6: 4660414.309 ops/s
# Warmup Iteration   7: 5267248.079 ops/s
Iteration   1: 32894213.089 ops/s
Iteration   2: 34024432.225 ops/s
Iteration   3: 46131122.469 ops/s
```

Man erkennt sehr schön den zuvor beschriebenen Einfluss der Annotations: Es gibt erwartungsgemäß vier Durchläufe mit jeweils sieben Warm-ups und drei Iterationen.

Der zweite Benchmark in JMH

Um noch ein wenig vertrauter mit JMH zu werden, erstellen wir eine Messung für die arithmetischen Operationen * und / sowie für das Potenzieren von `int`-Werten:

```
import org.openjdk.jmh.annotations.Benchmark;
import org.openjdk.jmh.annotations.BenchmarkMode;
import org.openjdk.jmh.annotations.Mode;
import org.openjdk.jmh.annotations.OutputTimeUnit;
import org.openjdk.jmh.annotations.State;
import org.openjdk.jmh.annotations.Scope;

import java.util.concurrent.TimeUnit;

@State(Scope.Benchmark)
public class SimpleMathBenchmark
{
    int x = 7271;
    int y = 1234;

    @Benchmark
    @BenchmarkMode({Mode.Throughput, Mode.AverageTime})
    @OutputTimeUnit(TimeUnit.NANOSECONDS)
    public long mult()
    {
        return x * y;
    }

    @Benchmark
    @BenchmarkMode(Mode.AverageTime)
    @OutputTimeUnit(TimeUnit.NANOSECONDS)
    public long div()
    {
        return x / y;
    }

    @Benchmark
    @BenchmarkMode(Mode.AverageTime)
    @OutputTimeUnit(TimeUnit.MILLISECONDS)
    public double pow()
    {
        return Math.pow(x, y);
    }
}
```

Im Listing lernen wir einige neue Annotations kennen:

- `@State` – Man kann spezifizieren, wie der Zustand zwischen den Threads des Benchmarks geteilt werden soll. Im Thread-Scope erhält jeder Thread seinen unabhängigen Zustand. Hier nutzen wir den Zustand pro Benchmark: Damit ist dieser für alle Threads gleich.
- `@BenchmarkMode` – Oftmals ist zur Performance-Beurteilung die Anzahl an Operationen pro Zeiteinheit (ops/time = Default) oder die durchschnittliche Ausführungszeit (time/ops) von Interesse. Beides verwenden wir im diesem Beispiel.
- `@OutputTimeUnit` – Mit JMH werden die Laufzeiten normalerweise in Nanosekunden berechnet. Mitunter ist man aber an anderen Zeitskalen interessiert, wie oben an Millisekunden für die Potenzierung.

Nachdem wir beide Klassen kompiliert und paketiert haben, starten wir das Executable JAR. Das produziert Konsolenausgaben etwa wie folgt:

```
# Benchmark mode: Throughput, ops/time
# Benchmark: org.sample.SimpleMathBenchmark.mult

Result "org.sample.SimpleMathBenchmark.mult":
  0.401 ±(99.9%) 0.012 ops/ns [Average]
  (min, avg, max) = (0.348, 0.401, 0.414), stdev = 0.017
  CI (99.9%): [0.388, 0.413] (assumes normal distribution)

# Benchmark mode: Average time, time/op
# Benchmark: org.sample.SimpleMathBenchmark.mult

Result "org.sample.SimpleMathBenchmark.mult":
  2.568 ±(99.9%) 0.108 ns/op [Average]
  (min, avg, max) = (2.414, 2.568, 2.931), stdev = 0.144
  CI (99.9%): [2.460, 2.676] (assumes normal distribution)
```

Zunächst werden zwischen den einzelnen Iterationen die Ergebnisse präsentiert und zum Abschuss erfolgt die Ausgabe einer Zusammenfassung.

```
Benchmark                                    Mode  Cnt         Score
      Error   Units
FirstIntegerBenchmark.numberAsBinaryString  thrpt   12  36490099.791 ±
      7938126.104   ops/s
FirstIntegerBenchmark.numberAsHexString     thrpt   12  78930550.252 ±
      18442012.239  ops/s
SimpleMathBenchmark.mult                    thrpt   25         0.401 ±
      0.012  ops/ns
SimpleMathBenchmark.div                      avgt   25         4.445 ±
      0.115   ns/op
SimpleMathBenchmark.mult                     avgt   25         2.568 ±
      0.108   ns/op
SimpleMathBenchmark.pow                      avgt   25        ≈ 10^5
                   ms/op
```

Mit deren Hilfe kann man verschiedene Aussagen zur Performance ableiten. So sieht man beispielsweise, dass die Multiplikation rund 2,5 ns benötigt, eine Division jedoch rund 4,4 ns. Das Potenzieren hat eine Laufzeit von 0.00005 ms (50 ns) und ist damit um Größenordnungen langsamer als die einfachen Operationen * und /. Das war zu erwarten, lässt sich aber so auch gut nachvollziehen.

8.1.3 Fazit zu JMH

Mitunter ist es wichtig, Aussagen über die Laufzeit spezieller Programmteile zu erhalten, um Optimierungspotenziale zu ermitteln. Neben einem Profiler, der Messungen zur Laufzeit ermöglicht, ist es manchmal auch wünschenswert, die Performance spezieller Anweisungen oder Methoden feingranular zu messen. Implementiert man dies jedoch von Hand, so wird man mit einigen Schwierigkeiten und Tücken konfrontiert. Sinnvoller ist es, hierzu JMH zu nutzen, das das Erstellen korrekter Microbenchmarks deutlich erleichtert. In diesem Unterkapitel konnte ich Ihnen lediglich einen ersten Eindruck von den Möglichkeiten von JMH vermitteln, es gibt noch sehr viel mehr zu entdecken.

11 Modularisierung mit Project Jigsaw

Dieses Kapitel behandelt das Thema Modularisierung und insbesondere die im Project Jigsaw (übersetzt: Stichsäge oder Puzzle) vorangetriebene Modularisierungslösung von Java und des JDKs. Dabei umfasst die Modularisierung zwei Bereiche:

- Die Modularisierung des JDKs an sich
- Die Modularisierung von Anwendungen und Bibliotheken

Neben dem Aufbrechen des monolithischen JDKs in einzelne Module mit der Möglichkeit zur Beschreibung und Kontrolle von Abhängigkeiten sollte vor allem die fehlerträchtige Abhängigkeitsverwaltung basierend auf dem `CLASSPATH` durch eine verlässliche Konfiguration (***Reliable Configuration***) ersetzt werden. Diese verlangt, dass Module ihre Abhängigkeiten untereinander explizit und vollständig beschreiben müssen (abgesehen von Services und spezieller Kommandozeilentricks). Dadurch können Zyklen, nicht eindeutige Modulnamen und fehlende Abhängigkeiten sowohl beim Kompilieren als auch zur Laufzeit geprüft und verhindert werden.

In den folgenden Abschnitten gebe ich eine Einführung in die Modularisierung und lege damit den Grundstein für das Verständnis der Thematik. Zudem schauen wir uns konkret an, welche Änderungen mit JDK 9 in Bezug auf die Modularisierung eingeführt wurden. Ergänzende fortgeschrittene Themen wie Services und Migrationsszenarien stelle ich in einem separaten Kapitel vor.

Verzeichnisaufbau der Beispielapplikationen

Weil einige IDEs noch keinen komfortablen Support für Module bieten und weil wir das Ganze von der Pike auf lernen wollen, werden wir in diesem Kapitel vor allem mit der Konsole arbeiten. Die Beispiele werden in jeweils eigenen Verzeichnissen entwickelt, wobei das Verzeichnis `jigsaw_ch11` parallel zum Verzeichnis `src/main/java` liegt. Es ergibt sich folgendes High-Level-Verzeichnislayout:

```
jigsaw_ch11
+-- ch11_2_2_first_module_example
+-- ch11_2_3_packaging_module_example
+-- ch11_2_4_linking_module_example
+-- ch11_2_5_dependencies_module_example
+-- ch11_2_6_include_jdk_modules_example
+-- ch11_3_2_accessibility
+-- ch11_3_3_implied_readability
```

11.1 Grundlagen

Mit dem Begriff Modularisierung verbindet vermutlich jeder Entwickler leicht unterschiedliche Vorstellungen. Deshalb möchte ich zunächst ein gemeinsames Verständnis für die Modularisierung in Java erreichen und insbesondere kurz auf mögliche Probleme und Schwierigkeiten, aber auch auf Anforderungen eingehen.

Unter ***Modulen*** versteht man in sich abgeschlossene, idealerweise unabhängig deploybare Softwarebausteine (Komponenten), die eine möglichst klar abgegrenzte Funktionalität bereitstellen, ihre Abhängigkeiten zu anderen Modulen beschreiben und nach außen sichtbare Typen explizit festlegen. Beachten Sie bitte, dass Jigsaw-Module in der Regel nicht unabhängig voneinander deployt werden können, wenn es Abhängigkeiten zwischen den Modulen gibt. Optionale Funktionalität lässt sich mithilfe von Services realisieren, wie wir es später in Abschnitt 12.2 sehen werden.

Modularisierung vs. monolithisches Design

Beim objektorientierten Programmieren kennt man verschiedene Designprinzipien wie SOLID oder Separation of Concerns. Vor allem auch hohe Kohäsion sowie lose Kopplung gelten als erstrebenswerte Ziele beim Entwurf. Die letzten beiden führen uns zum Konzept der Module: Mit Modularisierung meint man häufig, dass ein Programm in mehrere Subsysteme oder Module unterteilt ist, die jeweils eine Thematik adressieren. Diese thematische Gruppierung von Klassen und Packages hilft, klare Zuständigkeiten und Abhängigkeiten zu erzielen.

Dem steht das monolithische Design gegenüber, bei dem die einzelnen Softwarekomponenten stark miteinander verknüpft sind. Eine Änderung an einer Stelle zieht mitunter diverse Folgeänderungen an ganz anderen Stellen im Programm nach sich. Erweiterbarkeit und Wartbarkeit leiden. Das kann sich so dramatisch auswirken, dass man von Änderungen möglichst absieht, weil diese durch die Komplexität des Systems nahezu unbeherrschbar werden. Man sollte aber unbedingt vermeiden, in einen solchen Zustand einer »Schockstarre« zu kommen, damit man handlungsfähig bleibt. Ein sauberes Design unter Einhaltung wesentlicher OO-Designprinzipien sowie eine klare Strukturierung und Modularisierung können dabei helfen.

Anforderungen an Module

Bei der Modularisierung sollten folgende Eigenschaften für ein Modul als Softwarekomponente gelten. Jedes Modul ...

- besitzt einen eindeutigen Identifier (z. B. durch Name oder ID und eventuell eine Version),
- bietet Funktionalität über eine wohldefinierte Schnittstelle an,
- versteckt die Implementierungsdetails und veröffentlicht nur das, was explizit festgelegt wird, und
- beschreibt, was es an Abhängigkeiten besitzt.

Vorteile der Modularisierung

Neben der besseren Beherrschbarkeit, den klaren Abhängigkeiten und der reduzierten Komplexität führt die Modularisierung einer Applikation oftmals dazu, dass sich einzelne Module unabhängig von anderen testen und sogar parallel – bei Bedarf durch unterschiedliche Teams – (weiter)entwickeln lassen.

In dieser Hinsicht hilft Modularisierung insbesondere bei mittleren und größeren Projekten, die Wartbarkeit und Verständlichkeit sicherzustellen. Aber selbst kleinere Projekte profitieren von der besseren Nachvollziehbarkeit von Abhängigkeiten. Dadurch findet man weniger Spaghetticode und die Wartungsaufwände reduzieren sich.

Für die Modularisierungslösung des JDKs gibt es eine weitere Sache zu bedenken: Es lassen sich spezifische minimale Runtime-Images erstellen (vgl. Abschnitt 11.2.4).

11.1.1 Bisherige Varianten der Modularisierung

Schauen wir uns zunächst bislang mögliche Umsetzungsvarianten der Modularisierung in Java und ihre Schwachstellen an, bevor ich auf die Neuerungen mit Java 9 eingehe.

Interfaces, Klassen und Packages

Mit objektorientierten Mitteln lassen sich zwar bereits einige der zuvor als vorteilhaft erkannten Eigenschaften einer Modularisierung erreichen. Allerdings verbleiben einige Schwachstellen. Insbesondere die lose Kopplung zwischen Softwarekomponenten lässt sich nicht forcieren, jedoch durch Interfaces erleichtern.

Auch Packages sind nur begrenzt zur Modularisierung hilfreich, weil sie lediglich einen in sich geschlossenen Namensraum bilden, aber sich nicht hierarchisch schachteln lassen. Zwar spricht man häufig von Subpackage – korrekt ist jedoch, dass Package-Verschachtelungen nur zu einer Hierarchie im Dateisystem führen, allerdings nicht auf Sprachebene. Im Speziellen ermöglichen Packages keine feingranulare Kontrolle darüber, welche Typen sie exportieren, also für andere Packages bereitstellen. In einem Package sind `public` definierte Typen global `public` und deshalb auch aus allen anderen Packages beliebig zugreifbar. Folglich bieten Packages keine geeigneten Mittel zur Modularisierung, auch wenn sie unbestritten eine gute Strukturierung erlauben.

JAR-Dateien

JAR-Dateien erleichtern das Deployment und ermöglichen es, eine Menge von Klassen und Packages als eine Einheit zu veröffentlichen. Allerdings sind JAR-Dateien nichts anderes als ZIP-Dateien, in denen Klassen in Packages zusammengefasst sind und in denen gewisse Metainformationen in dem speziellen Verzeichnis `META-INF` bereitgestellt werden können. Abhängigkeiten zu anderen JARs lassen sich jedoch nicht explizit definieren. Welche Auswirkungen hat das?

Nahezu jede Java-Anwendung bindet einige JAR-Dateien über den `CLASSPATH` ein, um die in den JARs definierten Klassen zu nutzen. Hierbei besteht folgendes Pro-

blem: Möglicherweise befindet sich die gleiche Klasse mit abweichendem Stand und Bytecode in unterschiedlichen JARs. Es wird aber standardmäßig nur eine Variante einer Klassendefinition aus dem `CLASSPATH` geladen. Schlimmer noch: Das kann je nach Ausführungspfad durch die Anwendung in unterschiedlicher Reihenfolge geschehen. Man spricht dann von der JAR-Hölle. Die Erfahrung zeigt, dass derartige Probleme recht schnell entstehen. Erschwerend kommt hinzu, dass JARs häufig Typen aus anderen JARs benötigen, dies aber nicht explizit notiert werden kann. Dadurch gestalten sich Abhängigkeiten undurchsichtig und sind schwierig aufzulösen. Tatsächlich befinden sich alle Klassen flach im `CLASSPATH`. Um zumindest eine definierte Reihenfolge beim Laden von Klassen zu erzielen, kann man die Einträge des `CLASSPATH` alphabetisch sortieren. Das ist allerdings eher Symptombekämpfung als eine Lösung.

Zusammenfassend lässt sich festhalten, dass JARs zwar Ansätze zur Modularisierung mitbringen, aber nicht alle dafür benötigten Eigenschaften. Insbesondere fehlt die Möglichkeit, Abhängigkeiten klar definieren und Sichtbarkeiten steuern zu können.

Vergleich zu OSGi (Open Services Gateway initiative)

OSGi ist ein schon seit dem Jahr 2000 existierendes, ausgereiftes und gut funktionierendes Modularisierungssystem mit expliziter Abhängigkeitskontrolle, das in diversen Projekten Verwendung findet. Viele Entwickler haben sich deshalb – vermutlich auch zu recht – gefragt, warum nicht OSGi als Basis für die Modularisierung in Java dient. Tatsächlich sehe ich folgende Punkte, die dagegen sprachen: Es sollte eine Lösung entstehen, die direkt in die JVM und den Compiler integriert ist und die es darüber hinaus erlaubt, das JDK an sich zu modularisieren. Außerdem sollte ein einfacheres, für jeden Java-Entwickler potenziell interessantes Modulsystem entwickelt werden – denn nicht jeder braucht die Dynamik zur Laufzeit von OSGi.

Kommen wir zurück zu OSGi, mit dem beschrieben werden kann, wie Klassen innerhalb einer JVM interagieren. Dazu bietet OSGi folgende Features:

- Verschiedene Varianten von `ClassLoader`n, die eine feingranulare Kontrolle und unterschiedliche Versionen einer Klasse erlauben.
- Eine zentrale Service Registry, um Aufrufer und Implementierer zu entkoppeln – wobei Funktionalität über Interfaces gekapselt und bereitgestellt wird.
- Eine Menge von Standard-Services
- Eine Versionierung, die es ermöglicht, mehrere unterschiedliche Versionen eines Packages gleichzeitig in Betrieb zu haben.
- Ein Lifecycle-Management, wodurch Komponenten sogar erst zur Laufzeit hinzugefügt und ausgetauscht werden können.

Tipp: OSGi verwendet getrennte `ClassLoader`, wieso Jigsaw nicht?

Aus der obigen Aufzählung wird ersichtlich, dass OSGi verschiedene `ClassLoader` nutzt. Bei Jigsaw ist das nicht der Fall. Was ist die Erklärung dafür? Einfach gesagt, benötigt man bei OSGi mehrere `ClassLoader`, um unterschiedliche Versionen einer Komponente gleichzeitig betreiben und gegebenenfalls auch dynamisch nachladen bzw. austauschen zu können. Bei Project Jigsaw liegt der Fokus eher auf der Beschreibung und Kontrolle von Abhängigkeiten und weniger auf der Versionierung und dem Lifecycle-Management.

11.1.2 Warum Modularisierung wünschenswert ist

Bevor wir uns eingehender mit der Modularisierung beschäftigen, möchte ich nochmals kurz die durch Project Jigsaw adressierten Probleme rekapitulieren.

Problem 1: Typauflösung aus dem `CLASSPATH`

Wünschenswert ist es, Abhängigkeiten explizit beschreiben und überprüfen zu lassen. Eine Typauflösung basierend auf dem `CLASSPATH` macht dies fast unmöglich. Das liegt vor allem daran, dass der `CLASSPATH` aus einer Sammlung von Klassendefinitionen (auch denjenigen in JAR-Dateien) besteht. Allerdings sind die Abhängigkeiten und benötigten Typen nur indirekt über die `import`-Anweisungen im Sourcecode beschrieben. Dadurch ist es schwierig, im Vorhinein zu sagen, ob es beim Programmaufruf bzw. während der Ausführung Probleme aufgrund fehlender Abhängigkeiten oder doppelter Definitionen geben wird. Auch die für Typen genutzte Suchstrategie trägt dazu bei: Im gesamten `CLASSPATH` wird nach einer Klassendefinition von einem benötigten Typ gesucht, bis dieser gefunden wird.

Insgesamt lässt sich feststellen, dass man durch die fehlende explizite Beschreibung von Abhängigkeiten beim Applikationsstart nicht mit Sicherheit feststellen kann, ob ein Artefakt fehlt oder ob gleiche Artefakte in verschiedenen JARs vorliegen, die unterschiedliche Versionsstände repräsentieren. Hier macht sich bemerkbar, dass JAR-Dateien keine Komponenten sind, sondern nur Sammelstellen von Klassendefinitionen.

Problem 2: Monolithisches JDK

Das JDK war bis einschließlich JDK 8 ein riesiger Haufen von Klassen und Interfaces mit einer Vielzahl an Querabhängigkeiten. Dadurch wurde es ungemein erschwert, einzelne Bestandteile herauszulösen und separat bereitzustellen.

Durch den Trend zu IoT (Internet of Things) und zu kleinen Geräten, die zwar häufig leistungsfähige Prozessoren, jedoch nicht allzu viel Speicher besitzen, ist es wünschenswert und für die Verbreitung von Java wichtig, dort eine Java-Laufzeitumgebung anbieten zu können. Das wird umso einfacher, je modularer das JDK ist, weil dann spezielle, speicherplatzoptimierte Ausprägungen des JDKs genutzt werden können.

Lösung: Module in Java 9

Die genannten Probleme werden durch das Modulkonzept des JDKs und die damit eingeführten Module adressiert. Ein solches Modul hat einen eindeutigen Namen und besteht aus Packages, Klassen und Interfaces. Zudem besitzt ein Modul klar definierte Abhängigkeiten zu anderen Modulen: Dazu erfolgt eine Auflistung zu nutzender Module und exportierter Packages, die durch andere Module zugreifbar sein sollen. Somit kann explizit gesteuert werden, welche anderen Module ein Modul zum Kompilieren und bei der Ausführung benötigt. Im Gegensatz zum `CLASSPATH` wird eine stärkere Kapselung erzielt. Dabei stellen sowohl der Compiler als auch die JVM sicher, dass nur referenzierte Module und freigegebene Typen zugreifbar sind.

Zudem wird der Lookup einer benötigten Klassendefinition (`.class`-Datei) durch die explizite Beschreibung der Abhängigkeiten performanter. Das liegt daran, dass die Suche nach Klassendefinitionen nur dem Weg durch die Modulabhängigkeiten folgen und nicht im Extremfall den gesamten `CLASSPATH` überprüfen muss.

11.2 Modularisierung im Überblick

In diesem Unterkapitel werden wir uns die mit Project Jigsaw realisierte Modularisierungslösung des JDKs im Detail ansehen. Neben der Beschreibung benötigter Abhängigkeiten wird auch eine stärkere Kapselung als mithilfe von JARs möglich: Das Schlüsselwort `public` besagt in Modulen nun nicht mehr, dass ein Typ für alle Klassen aus beliebigen anderen Modulen zugreifbar ist.

11.2.1 Grundlagen zu Project Jigsaw

Einleitend möchte ich darauf hinweisen, dass das JDK an vielen Stellen für die Modularisierung angepasst wurde. Die folgende Aufzählung nennt die wesentlichen Änderungen:

- Die JDK-Verzeichnisstruktur weicht deutlich ab, wie dies Abbildung 11-1 zeigt. Insbesondere wird die im JDK enthaltene JRE anders als früher nicht mehr innerhalb des JDKs in einem getrennten Verzeichnis installiert.
- Die Dateien `rt.jar` und `tools.jar` gibt es nicht mehr. Stattdessen existieren jetzt verschiedene Module im Unterverzeichnis `jmods`.
- Auf einige interne Packages, z. B. `sun.misc`, kann nicht mehr (ohne Weiteres) zugegriffen werden.

- Reflection beachtet nun auch die von Modulen vorgegebenen Sichtbarkeitsregeln und ist somit eingeschränkter verwendbar als früher. Unter anderem führt das für Tools und Frameworks zu Problemen, kann aber durch Tricks und Kommandozeilenparameter umgangen werden. Details beschreibt Abschnitt 12.3.3.
- Der Mechanismus zum Laden von Klassen wurde geändert. Klassen werden standardmäßig nicht mehr im `CLASSPATH` gesucht, sondern im Module-Path. Dieser besteht aus einem oder mehreren Verzeichnissen, die Module enthalten. Darüber hinaus können Klassen auch aus einem `CLASSPATH` geladen werden. Das ermöglicht einen Kompatibilitätsmodus.
- Die Erweiterung des JDKs mit »endorsed dirs« (vgl. folgenden Praxistipp) wird nicht mehr unterstützt.
- Es gibt einen Linker, mit dem man spezielle Executables erstellen kann.

Aufgrund dieser Änderungen gab und gibt es gewisse Anlaufschwierigkeiten beim Modularisieren von Applikationen. Auf einen Kompatibilitätsmodus und mögliche Varianten einer Migration gehe ich später in Abschnitt 12.4 ein.

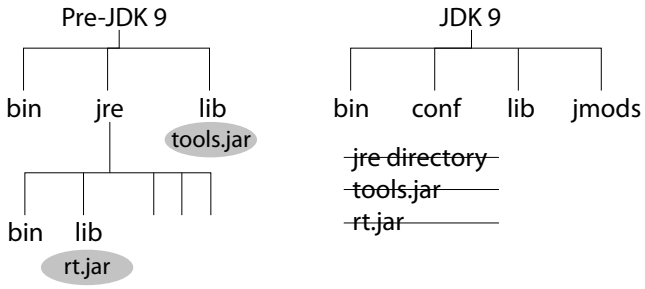

Abbildung 11-1 *Strukturänderungen im JDK*

Tipp: Redefinition von Java-Klassen und »endorsed dirs«

Mit dem Mechanismus der »endorsed dirs« konnte man nachträglich neuere Versionen von Bibliotheken in der JRE bereitstellen, indem diese im Verzeichnis `jre/lib/endorsed` abgelegt wurden. Das war hilfreich, weil etwa im Bereich von XML und JAXP häufiger Aktualisierungen und damit neue Versionen als Java-Releases erschienen. Somit waren zwischenzeitliche Aktualisierungen möglich.

Klassen konnten statt aus der Datei `rt.jar` aus diesem speziellen Verzeichnis geladen werden. Allerdings war dies auf gewisse Klassen beschränkt. Nur so konnte verhindert werden, dass Security-Probleme entstehen, indem Angreifer etwa eine infizierte Variante der Klasse `java.lang.String` bereitstellen.

Spracherweiterungen

Die Sprache selbst wurde unter anderem um folgende Schlüsselwörter erweitert:

- `module` – Definiert ein Modul.
- `requires` – Beschreibt die Abhängigkeiten von anderen Modulen.
- `exports` – Legt fest, welche eigenen Packages exportiert werden, d. h. für andere Module sichtbar sind.

Darüber hinaus gibt es noch einige weitere neue Schlüsselwörter, aber die obigen drei bilden die Basis einer Moduldefinition, auch ***Moduldeskriptor*** genannt. Dieser wird durch eine Datei namens `module-info.java` bereitgestellt:

```
module <ModuleName>
{
    requires <ModuleNameOfRequiredModule>;

    exports <PackageName>;
}
```

Abhängigkeiten von anderen Modulen werden mit dem Schlüsselwort `requires` beschrieben. Zudem kann ein Modul mit dem Schlüsselwort `exports` explizit festlegen, welche Packages nach außen für andere Module zugänglich sein sollen – alle nicht aufgeführten sind von extern nicht zugreifbar. Vereinfachend kann man ein Modul grafisch wie in Abbildung 11-2 darstellen. Links sieht man eine sehr einfache Repräsentation. Rechts ist eine Variante abgebildet, die recht gut den Moduldeskriptor widerspiegelt und zudem verdeutlicht, dass der Sourcecode im Inneren des Moduls gekapselt und versteckt ist.

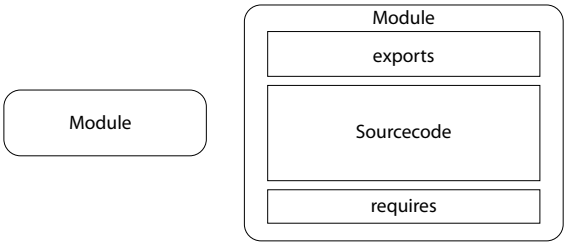

Abbildung 11-2 *Schematische Darstellung eines Moduls*

Tipp: Neue Schlüsselwörter nur im Moduldeskriptor

Die gerade aufgezählten und auch die später genannten neuen Schlüsselwörter werden nur innerhalb von Moduldeskriptoren als solche interpretiert: Das vermeidet Namenskonflikte mit bestehendem Sourcecode, etwa bei folgenden Variablen:

```
String module = "xxx";
int requires = 4711;
```

Auch die Beziehung zwischen Modulen kann man grafisch gestalten, wie dies Abbildung 11-3 zeigt.

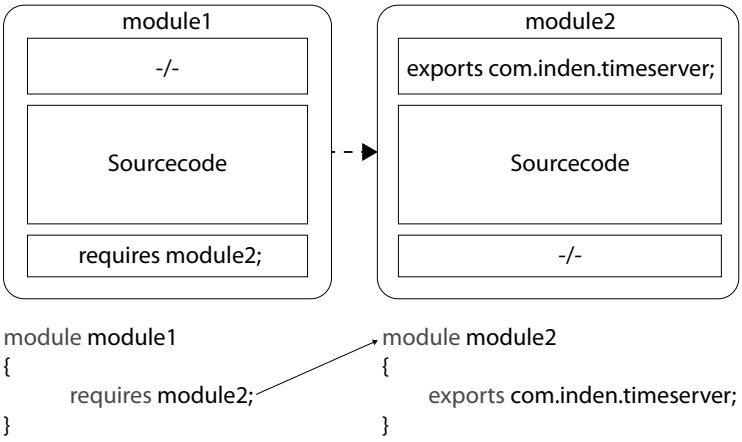

Abbildung 11-3 *Modul 1 greift auf Modul 2 zu.*

Im Zusammenhang mit Abhängigkeitsbeziehungen gibt es die zwei wichtigen Begriffe Readability und Accessibility, die ich nun beleuchte.

Die Begriffe Readability und Accessibility

Dass ein Modul 1 per `requires` auf ein anderes Modul 2 verweist, ist die Voraussetzung dafür, dass es die dort definierten Typen referenzieren kann. Man spricht dann davon, dass Modul 1 das andere Modul 2 liest, oder alternativ, dass Modul 2 für Modul 1 lesbar ist. Per Definition liest jedes Modul sich selbst. Diese ***Readability*** bildet die Grundlage für eine verlässliche Konfiguration (***Reliable Configuration***). Dazu prüft und stellt das mit JDK 9 eingeführte Modulsystem sicher, dass jede Abhängigkeit genau durch ein Modul erfüllt wird und der entstehende Modulgraph azyklisch (zyklusfrei) ist. Zudem wird dafür gesorgt, dass unterschiedliche Module keine Packages gleichen Namens enthalten, sogenannte Split Packages.[1]

Die Readability in Kombination mit den jeweiligen `exports`-Anweisungen sorgen für ***gute Kapselung***, weil wirklich nur die explizit exportierten Packages für andere Module freigegeben sind. Dabei gilt noch die ***Accessibility***. Darunter versteht man, dass eine Klasse A aus Modul 2 nur dann zugreifbar ist, wenn Modul 1 das korrespondierende Modul 2 liest und Modul 2 zusätzlich das zugehörige Package exportiert.

[1] Split Packages hat man früher das eine oder andere Mal zur Redefinition von Klassen genutzt, indem in eigenem Sourcecode einfach ein Package gleichen Namens definiert wurde und dort die benötigten Klassen entsprechend modifiziert wurden: Durch geschickte Positionierung im `CLASSPATH` konnte man so Funktionalität geeignet anpassen.

Benennung von eigenen Modulen

Der Name eines Moduls kann frei vergeben werden, er muss allerdings im Ausführungskontext einer JVM (besser noch global) eindeutig sein. Das lässt sich für Module am einfachsten dadurch erreichen, dass man das Reverse-Domain-Name-Pattern von Packages nutzt. Damit sagt der Name jedoch an verschiedenen Stellen Unterschiedliches aus: Modulname, Package-Name oder Sourcecode-Verzeichnis. Weil dies anfangs verwirrend sein kann, verzichte ich zur Vermeidung von Mehrfachdeutungen für die Beispiele des Buchs auf diese Namenskonvention und nutze einfache Namen für Module.

Duplikate in `exports` oder Packages Wie gerade erwähnt, müssen Modulnamen wie auch Package-Namen eindeutig sein. Dies wird vom Compiler sichergestellt. Bei der Definition von gleichnamigen Packages in unterschiedlichen Modulen kommt es zu Fehlermeldungen wie dieser:

```
DOPPELTE packages in zwei Modulen:
src/jigsaw/timeclient/module-info.java:1:
      error: module timeclient reads package com.server from both
      timeserver and timeserver2
module timeclient
```

Auch ein mehrfacher Export eines Packages (durch Mehrfachangabe eines Packages in verschiedenen Moduldeskriptor-Dateien) wird folgendermaßen angemerkt:

```
DOPPELTE exports:
src/jigsaw/timeserver/module-info.java:7:
      error: duplicate export: com.serverexports com.server to timeclient;
```

Von Oracle empfohlene (aber oftmals ungünstige) Verzeichnisstruktur

Soll eine Applikation in mehrere Module untergliedert werden, so wird von Oracle derzeit empfohlen, ein gemeinsames `src`-Verzeichnis zu nutzen. Pro Modul wird der Sourcecode dann in einem Unterverzeichnis mit dem Namen des Moduls abgelegt. Demgemäß ergibt sich für zwei Module eine Verzeichnisstruktur ähnlich zu folgender:

```
'-- src
    |-- com.inden.module1
    |   |-- com
    |   |   '-- inden
    |   |       '-- module1
    |   |           '-- Application.java
    |   '-- module-info.java
    '-- com.inden.module2
        |-- com
        |   '-- inden
        |       '-- module2
        |           '-- OtherClass.java
        '-- module-info.java
```

Man erkennt folgende wesentliche Punkte:

1. Per Konvention liegt der Moduldeskriptor auf der Hauptebene des Moduls.
2. Unterhalb des Modulverzeichnisses arbeitet man wie gewohnt mit Packages.
3. Etwas irritierend ist das Modulverzeichnis und sein Name. Nutzt man dafür – wie es die Konvention vorsieht – einen ***Reverse Domain Name***, so sind der Modulname und diejenigen der darunterliegenden Packages leicht zu verwechseln.

Beachten Sie unbedingt, dass sich diese Aufteilung für eigene Module in der Regel weniger eignet: Besteht eine Applikation aus mehreren Modulen und sollen diese von verschiedenen Teams realisiert werden, so ist ein gemeinsames Source-Verzeichnis insofern ungünstig, als dadurch erschwert wird, die Module in einzelnen JARs bereitzustellen und möglichst unabhängig voneinander weiterzuentwickeln. Im Idealfall werden die JARs der Module in einem gemeinsamen Repository verwaltet und von dort bei Bedarf heruntergeladen.

Für die Sourcen des JDKs ist es dagegen praktisch, alle auf einmal kompilieren zu können. Deshalb wurde diese Ein-Source-Verzeichnis-Variante vermutlich von Oracle so eingeführt.

In diesem einleitenden Kapitel werden wir der von Oracle vorgeschlagenen Struktur folgen. Einerseits um deren Vorteile für kleinere miteinander verbundene Module zu demonstrieren und andererseits um ein paar spezielle Kommandozeilenparameter kennenzulernen. In einem weiteren Kapitel zur Modularisierung zeige und nutze ich dann eine für die Praxis empfehlenswerte Struktur.

Erweiterungen im Ökosystem

Um eigene modularisierte Applikationen kompilieren und starten zu können, wurden sowohl der Compiler `javac` als auch die JVM, also das Kommando `java`, angepasst. Beiden kann man nun weitere Parameter übergeben:

```
javac --module-path <modulepath> ...

java -p <modulepath> -m <modulename/fully-qualified-class-name>
```

Neu in dieser Aufzählung sind folgende Parameter:

- `--module-path` oder kurz `-p` – Statt eines `CLASSPATH` legt dieser Parameter den Module-Path fest, der einem oder mehreren Verzeichnissen entspricht, die Module enthalten.
- `--module` oder kurz `-m` – Spezifiziert das Hauptmodul, ähnlich zur Main-Klasse einer normalen Java-Applikation. Die Notation beginnt mit dem Namen des Moduls gefolgt von einem Schrägstrich und dem voll qualifizierten Klassennamen. Der Name der Hauptklasse kann im Modul selbst hinterlegt werden, dann ist die Angabe beim Aufruf nicht mehr nötig.

Module-Path und CLASSPATH

Wir haben bislang kennengelernt, dass Module durch Moduldeskriptoren beschrieben werden und Typen in Packages bereitstellen. Damit eine modularisierte Applikation gestartet werden kann, bedarf es eines Mechanismus, um benötigte Typen zu lokalisieren und zu laden. Dazu dient der Module-Path. Dieser kann exklusiv oder ergänzend zum bis einschließlich Java 8 genutzten `CLASSPATH` angegeben werden.

Erwähnenswert ist, dass man statt des oder ergänzend zum Module-Path weiterhin über `-cp` zur Rückwärtskompatibilität auch noch einen `CLASSPATH` nutzen kann: Für Applikationen ohne Module verhält sich die JVM genau so, wie man es bislang mit JDK 8 gewohnt war,[2] jedoch profitiert man dann nicht von den Vorteilen der Modularisierung.

Der Module-Path ist robuster als der `CLASSPATH`: Das liegt vor allem daran, dass beim `CLASSPATH` innerhalb des gesamten Pfades nach Typen gesucht wird, bis diese gefunden werden. Bei einer Suche im `CLASSPATH` ist nur durch die Import-Anweisungen im Sourcecode verzeichnet, welche Typen benötigt werden. Es ist aber nicht extern und explizit definiert, wie die Abhängigkeiten aussehen. Dadurch kann man im Vorhinein beim Applikationsstart nicht mit Sicherheit feststellen, ob ein Artefakt fehlt. Zudem können auch gleiche Klassen in verschiedenen JARs vorliegen, die unterschiedliche Versionsstände repräsentieren.

Mit dem Module-Path werden Abhängigkeiten grobgranularer auf Modulebene beschrieben anstatt auf Basis von Typen. Es wird sowohl zur Kompilierzeit als auch zur Laufzeit möglich, festzustellen, ob eine Abhängigkeit fehlt oder ob es einen Konflikt durch den Export des gleichen Packages aus verschiedenen Modulen gibt.

Modularisierung des JDKs

Im Rahmen von Project Jigsaw wurde auch das JDK in einer groß angelegten Aufräumaktion in diverse Module untergliedert. Mitunter ist es in der Programmierpraxis nützlich, sich die Module des JDKs auf der Kommandozeile auflisten zu lassen. Dazu dient folgendes Kommando:

```
java --list-modules
```

Als Ausgabe erhält man eine lange Liste von Modulen, hier stark gekürzt – bitte beachten Sie, dass die Liste zwar bezogen auf den gesamten Namen alphabetisch sortiert ist, aber durch die Ordnung nach Präfix dominiert wird, was zunächst irritierend sein kann:

[2] Allerdings werden Zugriffe auf JDK-Interna geprüft und gegebenenfalls verhindert, etwa auf die Klassen `BASE64Encoder` oder `Unsafe` aus dem Package `sun.misc`.

```
java.base@11.0.1
java.compiler@11.0.1
java.datatransfer@11.0.1
java.desktop@11.0.1
java.logging@11.0.1
java.naming@11.0.1
java.net.http@11.0.1
java.se@11.0.1
java.sql@11.0.1
java.xml@11.0.1
...
jdk.jconsole@11.0.1
jdk.jdeps@11.0.1
jdk.jdi@11.0.1
jdk.jdwp.agent@11.0.1
jdk.jfr@11.0.1
jdk.jlink@11.0.1
jdk.jshell@11.0.1
jdk.net@11.0.1
jdk.pack@11.0.1
jdk.rmic@11.0.1
jdk.scripting.nashorn@11.0.1
jdk.scripting.nashorn.shell@11.0.1
jdk.xml.dom@11.0.1
jdk.zipfs@11.0.1
```

Die JDK-Module beginnen normalerweise mit dem Präfix `java`, etwa `java.sql` für Datenbankanbindungen per SQL oder `java.logging` für Logging. Darüber hinaus existieren Module mit dem Präfix `jdk`, nämlich diejenigen, die nicht Bestandteil der Java-SE-Plattform sind. Mit Java 11 wurde JavaFX aus dem JDK entfernt, deshalb sind dessen Module hier nicht mehr in der Ausgabe zu finden, wie dies noch für Java 9 und 10 der Fall war.

Tipp: Ermittlung des Moduls zu einer Klasse

Leider gibt es (noch) kein Kommando, mit dem man zu einem gegebenen Klassennamen das zugehörige Modul ermitteln kann. Allerdings kann das neu gestaltete Javadoc des JDKs (`http://download.java.net/java/jdk9/docs/api/`) hilfreich sein: Es bietet zum einen eine Suchfunktionalität und enthält zum anderen nun Informationen zu dem Modul einer Klasse, wie es Abbildung 11-4 zeigt.

Abbildung 11-4 *Angabe des Moduls im* JAVADOC