

# 10 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 mithilfe 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 einen Einstieg in die Modularisierung und lege damit den Grundstein für das Verständnis der Thematik. Zudem schauen wir uns konkret an, welche Änderungen in Bezug auf die Modularisierung in JDK 9 enthalten sind. In einem separaten Kapitel werden ergänzend fortgeschrittene Themen wie Services und Migrationsszenarien vorgestellt.

## 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_ch6` parallel zum Verzeichnis `src/main/java` liegt. Es ergibt sich folgendes High-Level-Verzeichnislayout:

```
jigsaw_ch10
+-- ch10_2_2_first_module_example
+-- ch10_2_3_packaging_module_example
+-- ch10_2_4_linking_module_example
+-- ch10_2_5_dependencies_module_example
+-- ch10_2_6_include_jdk_modules_example
+-- ch10_3_2_accessibility
+-- ch10_3_3_implied_readability
```

## 10.1 Grundlagen

Für den Begriff Modularisierung hat vermutlich jeder Entwickler leicht unterschiedliche Ideen im Sinn. 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 11.2 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. Letztere 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 adressierend. 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 auch nicht allzu umfangreiche Projekte profitieren von der besseren Nachvollziehbarkeit von Abhängigkeiten. Als Folge findet man weniger Spaghetticode und es reduzieren sich die Wartungsaufwände.

Für die Modularisierungslösung des JDKs gibt es eine weitere Sache zu bedenken: Es lassen sich spezifische minimale Runtime-Images erstellen (vgl. Abschnitt 10.2.4).

### 10.1.1 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, aber nicht auf Sprachebene. Im Speziellen erlauben 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 allerdings folgendes Problem: 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 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 `ClassLoadern`, 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.

**10.1.2 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. Jedoch sind die Abhängigkeiten und benötigten Typen nur indirekt über die `import`-Anweisungen 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.<sup>1</sup> 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

<sup>1</sup>Zwar helfen hier Build-Tools wie Maven oder Gradle, aber wenn in Artefakten gleiche Klassen (mit gleichem voll qualifiziertem Namen) definiert sind, so kann es Konflikte geben.

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 neue Modulkonzept des JDKs und die damit neu 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, da die Suche nach Klassendefinitionen nur dem Weg durch die Modulabhängigkeiten folgen und nicht im Extremfall den gesamten `CLASSPATH` absuchen muss.

## 10.2 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.

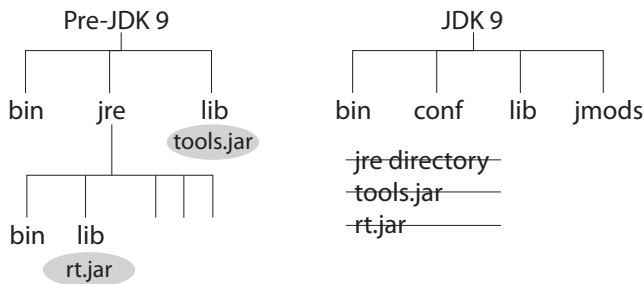
### 10.2.1 Grundlagen zu Project Jigsaw

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 10-1 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 11.3.3.
- Der Mechanismus zum Laden von Klassen wurde geändert. Klassen werden standardmäßig nicht mehr im `CLASSPATH` gesucht, sondern im Module-Path. Im `CLASSPATH` angegebene Klassen ermöglichen 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. Das hatte ich bereits in der Einleitung für verschiedene IDEs angedeutet. Auf einen Kompatibilitätsmodus und mögliche Varianten einer Migration gehe ich später in Abschnitt 11.4 ein.



**Abbildung 10-1** Strukturänderungen im JDK

### Tip: 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:

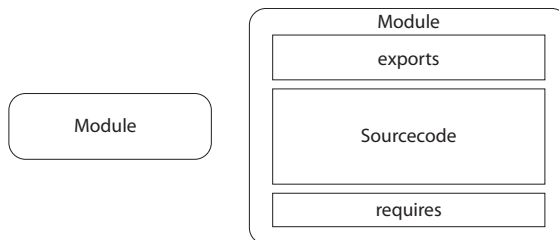
- `module` – Definiert ein Modul.
- `requires` – Beschreibt die Abhängigkeiten von anderen Modulen.
- `exports` – Legt fest, welche eigenen Packages exportiert werden, d. h. für andere Module sichtbar sind.

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 10-2 darstellen. Links ist eine einfache Repräsentation zu sehen. Rechts ist eine ausgeklügeltere Variante abgebildet, die recht gut den Moduldeskriptor widerspiegelt und zudem verdeutlicht, dass der Sourcecode im Inneren des Moduls gekapselt und versteckt ist.



**Abbildung 10-2** Schematische Darstellung eines Moduls

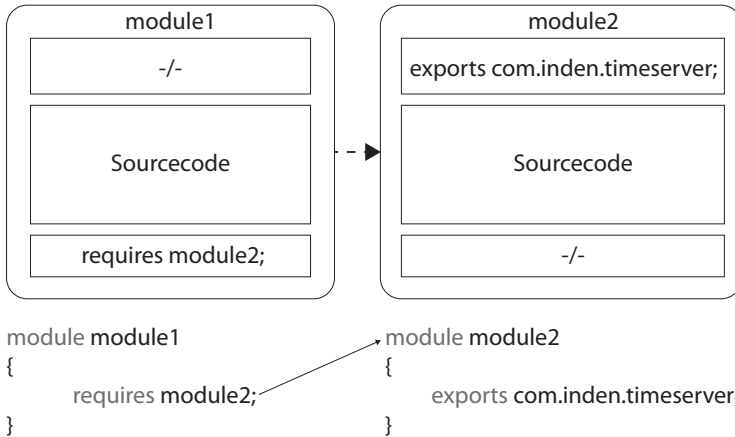
### Tipp: Neue Schlüsselwörter nur im Moduldeskriptor

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 10-3 zeigt.



**Abbildung 10-3** 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.<sup>2</sup>

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 Folgendes: Eine Klasse A aus Modul 2 ist nur dann zugreifbar, wenn Modul 1 das korrespondierende Modul 2 liest und Modul 2 zusätzlich das zugehörige Package exportiert.

<sup>2</sup>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 angepasst wurden.

## 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 Modulkriptor auf der Hauptebene des Moduls.
2. Unterhalb des Modulverzeichnis 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, eher 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 Quellen des JDKs ist es dagegen praktisch alle auf einmal kompilieren zu können. Deshalb wurde diese 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 den weiteren Kapiteln 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 Module-Path weiterhin über `-cp` zur Rückwärtskompatibilität exklusiv nur einen CLASSPATH nutzen kann: Für Applikationen ohne Module verhält sich die JVM genau so, wie man es bislang mit JDK 8 gewohnt war,<sup>3</sup> jedoch profitiert man dann nicht von den Vorteilen der Modularisierung. Deswegen wird in Zukunft für neue Projekte wohl vermehrt der Module-Path zum Einsatz kommen.

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:

```
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
```

<sup>3</sup>Allerdings werden Zugriffe auf JDK-Interna geprüft und gegebenenfalls verhindert, etwa auf die Klassen `BASE64Encoder` oder `Unsafe` aus dem Package `sun.misc`.

```

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 Datenbankverbindungen 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.

### Tip: 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<sup>a</sup> und enthält zum anderen nun Informationen zu dem Modul einer Klasse, wie es Abbildung 10-4 zeigt.

<sup>a</sup>Leider sind dort die Klassen von JavaFX momentan noch nicht indiziert bzw. enthalten – immerhin ist es separat hier erhältlich <http://download.java.net/java/jdk9/jfxdocs/index.html>.

```

Module java.httpclient
Package java.net.http
Class HttpResponse

```



Abbildung 10-4 Angabe des Moduls im JAVADOC