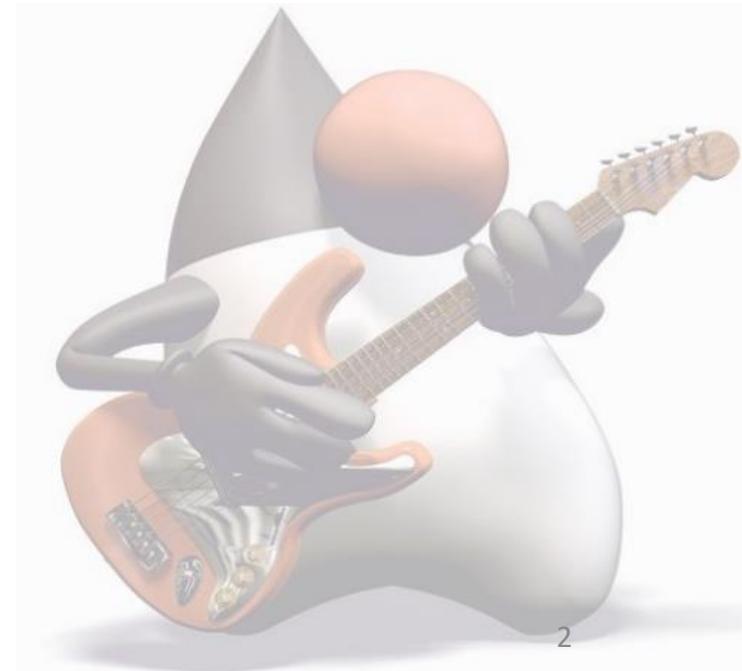


Java Modules



Übersicht

- Modularisierung (Java 9)



Modularisierung

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

Anforderungen an Module

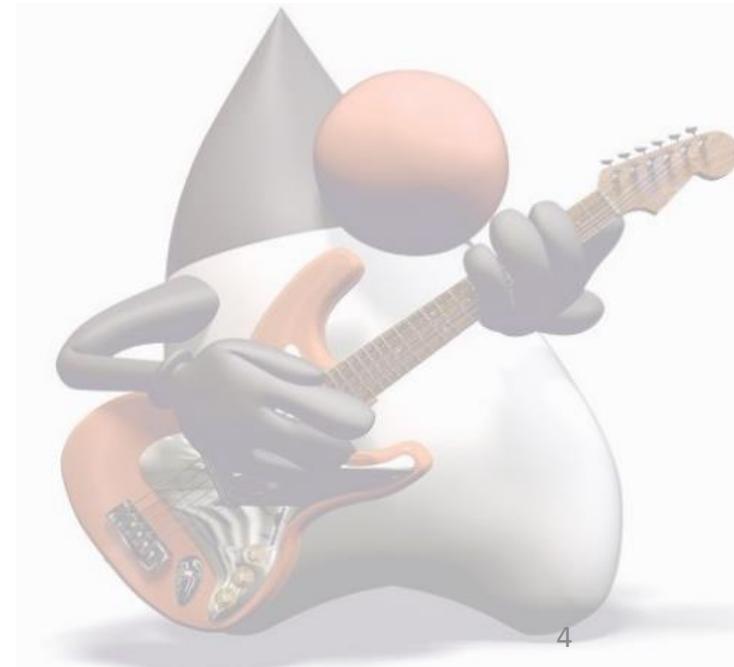
Bei der Modularisierung sollten folgende Eigenschaften für ein Modul als Softwarekomponente gelten. Jedes Modul ...

- 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.

Modularisierung

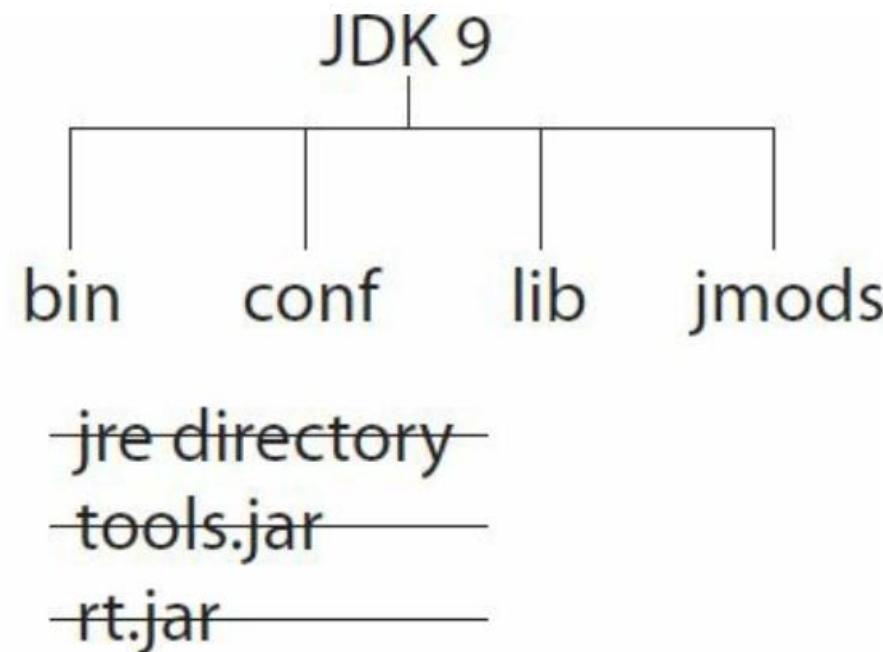
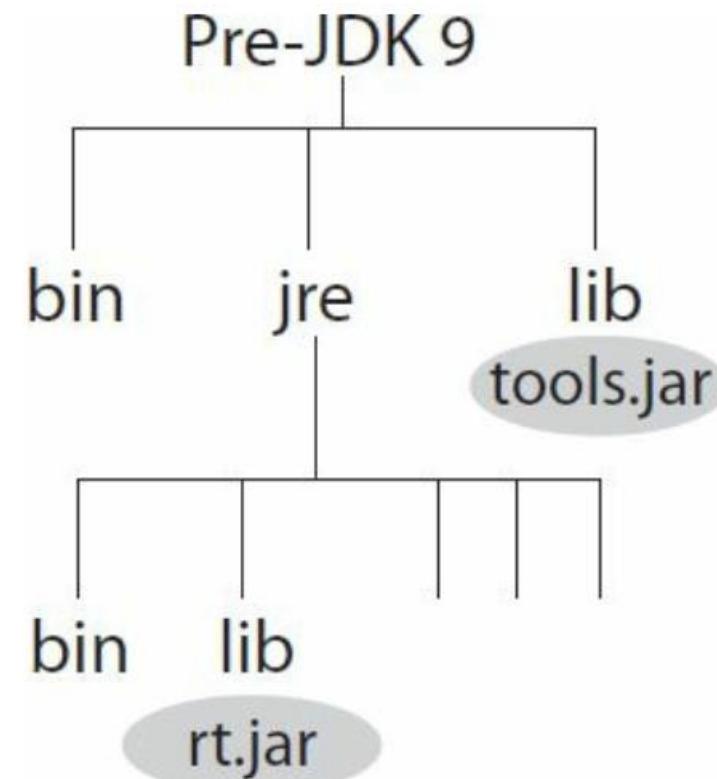
Interfaces, Klassen und Packages

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.



Modularisierung

Die Struktur des JDK hat sich wie folgt geändert:



Modularisierung

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.



Modularisierung

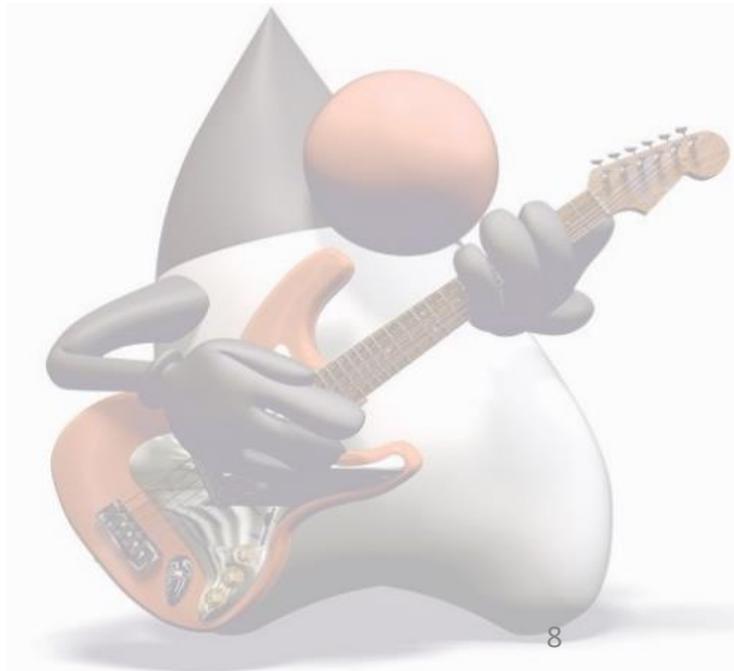
Darüber hinaus gibt es noch einige weitere neue Schlüsselwörter, aber die obigen drei bilden die Basis einer Moduldefinition, auch **Moduldeskriptor** genannt. Dieser wird durch eine Datei namens module-info.java bereitgestellt:

```
module <ModuleName>
{
    requires <ModuleNameOfRequiredModule>;
    exports <PackageName>;
}
```

Modularisierung

Beispieldatei module-info.java

```
module Modules {  
    requires java.base;  
    requires java.logging;  
    exports at.javatraining.mypackage;  
}
```



Modularisierung

Tipp: Neue Schlüsselwörter nur im Moduldeskriptor

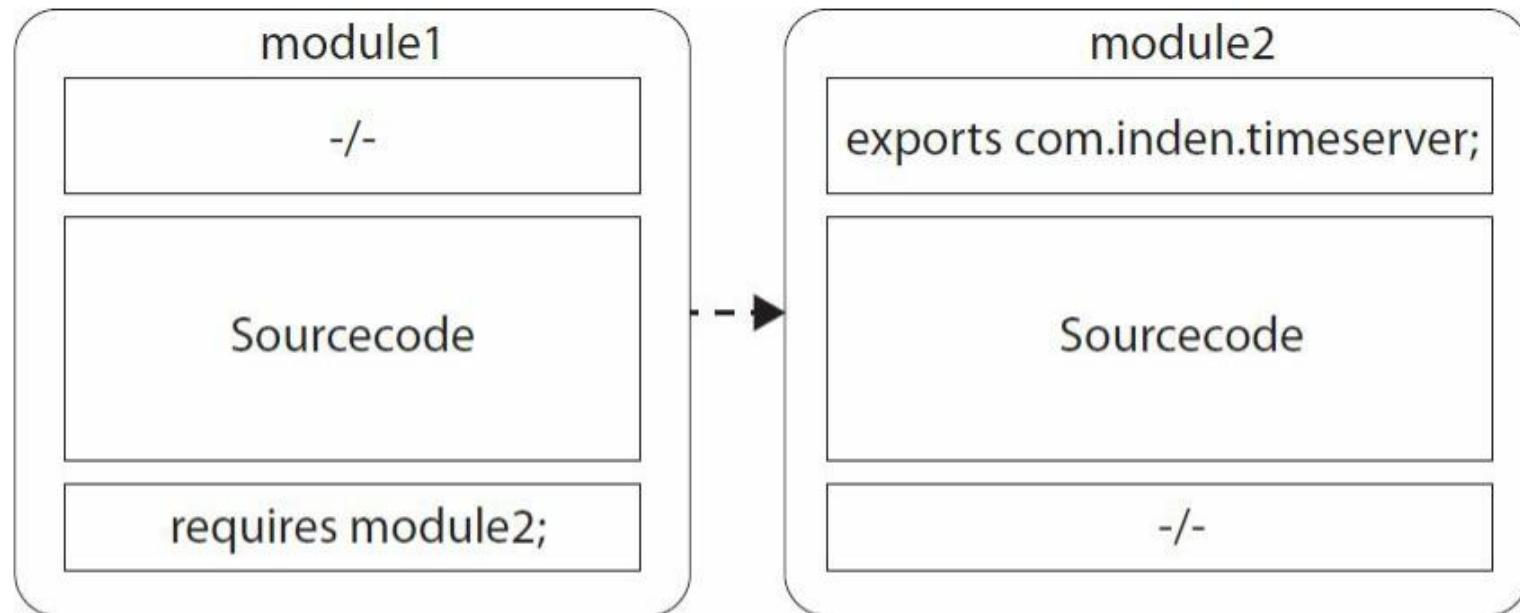
Die gerade aufgezählten und auch die später genannten neuen Schlüsselwörter werden nur innerhalb von Moduldeskriptoren als solche interpretiert: Das vermeidet Namenskonflikte mit bestehendem Sourcecode, etwa bei folgenden Variablen:

```
String module = "xxx";
```

```
int requires = 4711;
```

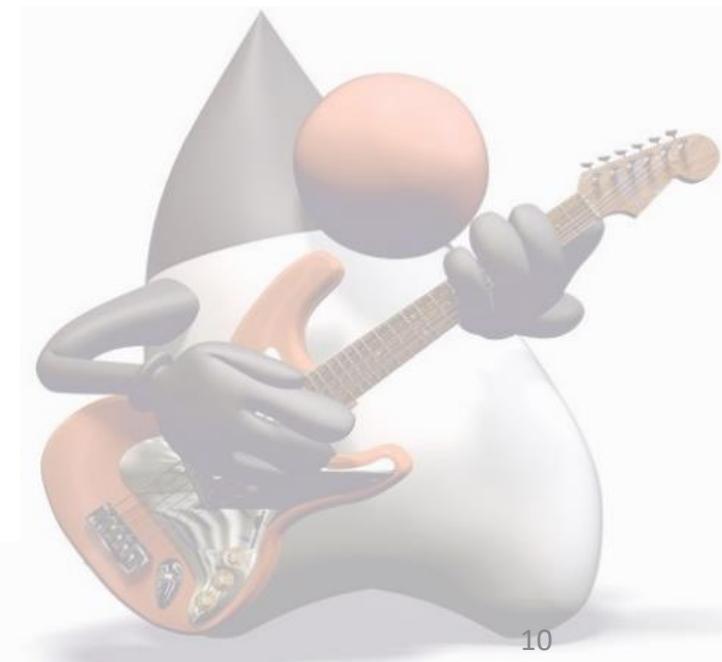


Modularisierung



```
module module1
{
    requires module2;
}

module module2
{
    exports com.inden.timeserver;
}
```



Modularisierung

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
```

Modularisierung

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>
```

- **--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.

Modularisierung

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:

```
C:\Users\michael>java --list-modules
java.base@13
java.compiler@13
java.datatransfer@13
java.desktop@13
java.instrument@13
java.logging@13
...
...
```



Modularisierung

Tipp: Ermittlung des Moduls zu einer Klasse

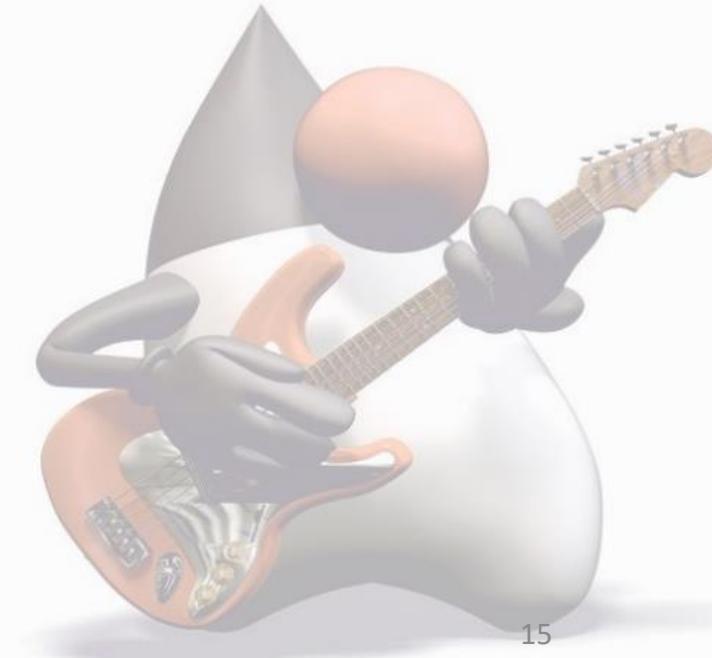
Leider gibt es (noch) kein Kommando, mit dem man zu einem gegebenen Klassennamen das zugehörige Modul ermitteln kann. Allerdings kann das neu gestaltete Javadoc des JDKs (<http://download.java.net/java/jdk9/docs/api/>) hilfreich sein: Es bietet zum einen eine Suchfunktionalität^a und enthält zum anderen nun Informationen zu dem Modul einer Klasse, wie es Abbildung 10-4 zeigt.

^aLeider 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>.



Modularisierung – ein Beispiel

Wir wollen zwei Module erzeugen: Das eine nennen wir `jigsawapp` und das andere heißt `services`. Im Modul `jigsawapp` soll sich die Applikation befinden, die für ihre Arbeit auf eine Klasse `MessageService` zugreift. Diese wird im Modul `services` implementiert. Damit ergibt sich folgende Abhängigkeitsbeziehung:



Modularisierung – ein Beispiel

ch10_2_2_first_module_example

```
'-- step<N>
  '-- src
    |-- jigsawapp
      |  |-- com
      |  |  '-- inden
      |  |  '-- javaprofi
      |  |      '-- MessageExample.java
      |  '-- module-info.java
  '-- services
    |-- com
      |  '-- services
      |  '-- MessageService.java
    '-- module-info.java
```



Modularisierung – ein Beispiel

The screenshot shows an IDE interface with two projects:

- jigsawapp** (C:\Users\michael\IdeaProjects\Modularization\jigsawapp):
 - src
 - com
 - inden
 - javaprofi
 - MessageExample.java
 - module-info.java
 - jigsawapp.iml
- services** (C:\Users\michael\IdeaProjects\Modularization\services):
 - src
 - com
 - services
 - MessageService.java
 - module-info.java
 - services.iml

Two code editors are visible:

- jigsawapp\...\module-info.java**:

```
1 module jigsawapp {  
2     requires services;  
3 }
```
- MessageExample.java**:

```
1 import com.services.MessageService;  
2  
3 public class MessageExample {  
4     public static void main(final String[] args) {  
5         System.out.println("Generated msg: " +  
6             MessageService.generateMessage());  
7     }  
8 }  
9  
10 }
```
- services\...\module-info.java**:

```
1 module services {  
2     exports com.services;  
3 }
```
- MessageService.java**:

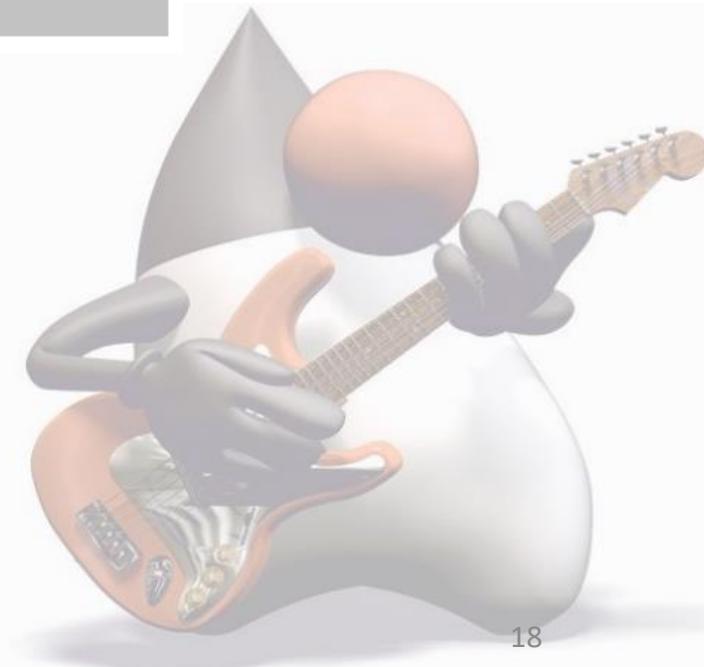
```
1 package com.services;  
2  
3 public class MessageService {  
4     @Override  
5     public static String generateMessage() {  
6         return "Message from module services!";  
7     }  
8 }
```

Modularisierung – ein Beispiel

Hinweis: Wissenswertes rund um **module-info.java**

Würden wir nur die Datei `module-info.java` für das Modul `services` wie gezeigt anpassen, so würde zwar die Klasse `MessageService` freigegeben, könnte aber immer noch nicht aus dem Modul `jigsawapp` zugegriffen werden, weil dort keine Abhängigkeit auf das Modul `services` definiert wäre. Denken Sie an die in [Abschnitt 10.2.1](#) vorgestellten Begriffe Readability und Accessibility.

Interessanterweise spezifiziert man durch das Schlüsselwort `requires Module` und beim Export gibt man freizugebende Packages mit `exports` an. Damit ist `import` das Pendant zu `exports`. Dagegen besagt `requires`, durch welches Modul oder welche Module die Imports erfüllt werden sollen.



Modularisierung – ein Beispiel

```
javac -d build/jigsawapp -p build \
      src/jigsawapp/*.java \
      src/jigsawapp/com/inden/javaprofi/*.java
```

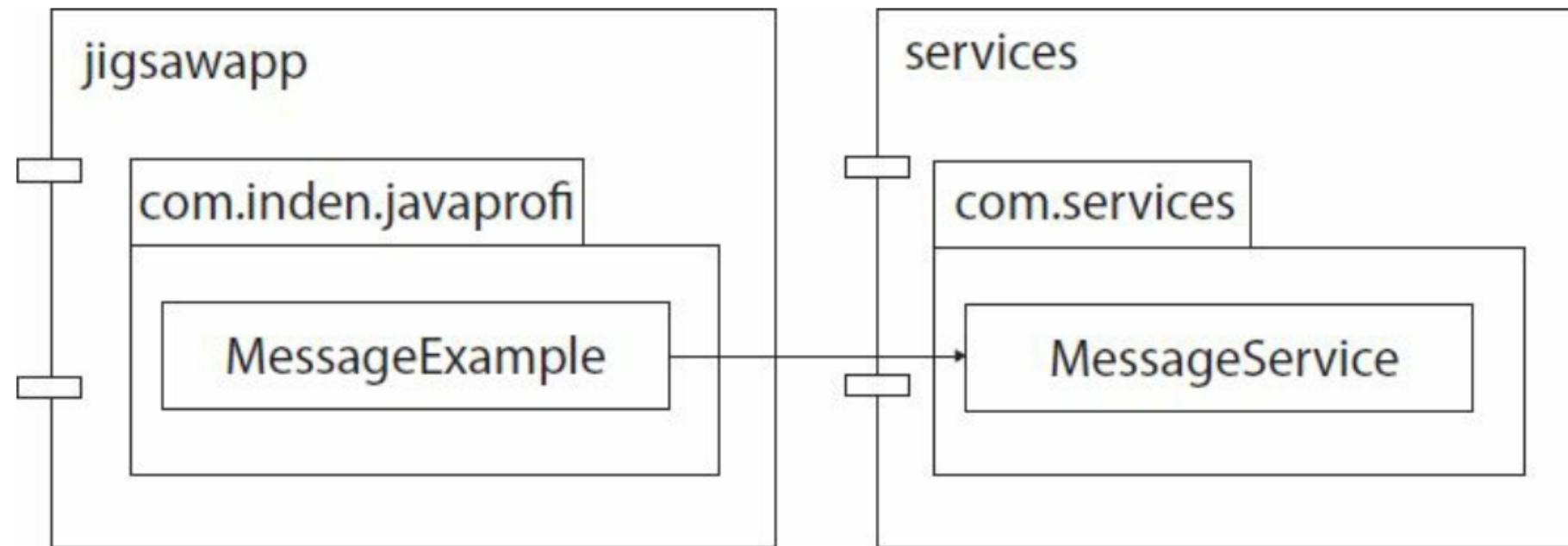
Multi-Module Build Statt die Modulabhängigkeiten beim Kompilieren explizit anzugeben, ist es beim Arbeiten mit der Kommandozeile oftmals bequemer, die Sourcen aller Module in einem Rutsch kompilieren zu können. Diese Art zu kompilieren wird **Multi-Module Build** genannt, erfordert allerdings zwingend das Standardverzeichnislayout – bei komplexeren modularisierten Systemen und abweichendem Verzeichnislayout, wie wir es in [Abschnitt 13.2](#) kennenlernen werden, kann man den Multi-Module Build nicht mehr nutzen.

Man kann das Compiler-Flag `--module-source-path` folgendermaßen einsetzen, sofern das von Oracle propagierte Standardverzeichnislayout eingehalten wird:

```
javac -d build --module-source-path src $(find src -name '*.java')
```

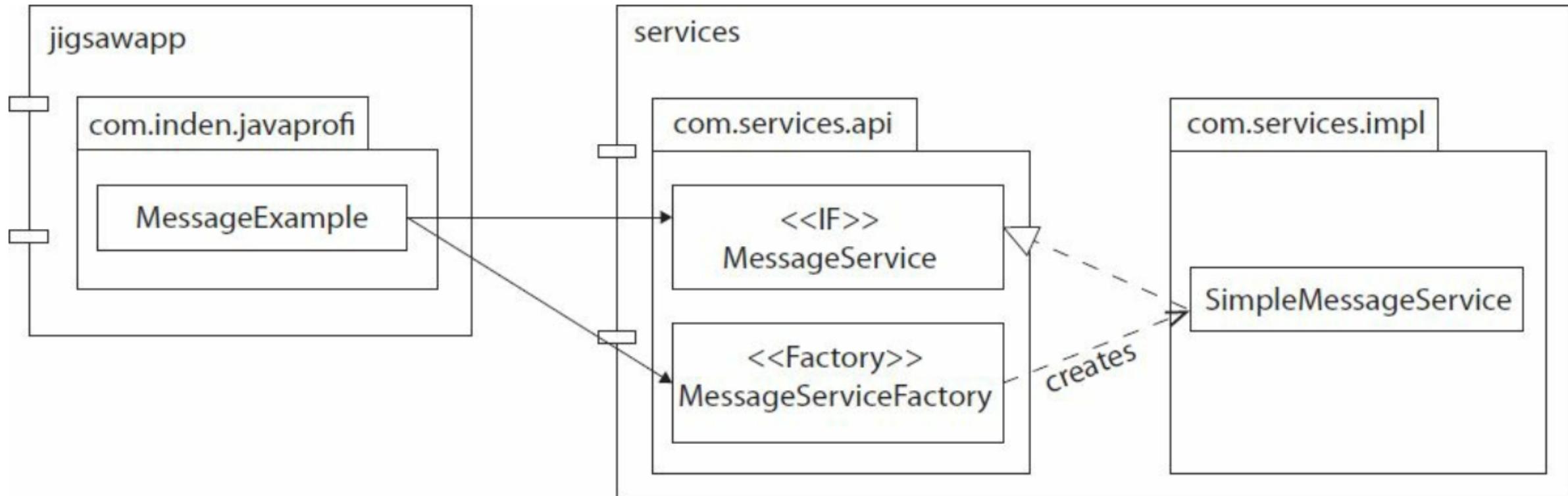
Modularisierung – ein Beispiel

Wir haben in den letzten Abschnitten gelernt, wie wir eine Applikation mithilfe von Modulen in kleinere Bestandteile untergliedern können. Das war ein erster Schritt in Richtung Modularisierung. Momentan verweist die nutzende Applikation direkt auf eine Klasse aus einem anderen Modul. Das stellt eine stärkere Implementierungsabhängigkeit dar: Die Module sind eng miteinander gekoppelt und bilden logisch eigentlich ein Modul. Diese enge Verknüpfung ist in [Abbildung 10-6](#) nur angedeutet.



Modularisierung – ein Beispiel

Idealerweise verweist eine nutzende Applikation lediglich auf Interfaces und erhält die Realisierungen über Factory-Methoden, Dependency Injection oder mithilfe von Services, wie dies in [Abschnitt 11.2](#) thematisiert wird. Nachfolgend schauen wir uns eine Entkopplung der beiden Module durch das Einfügen eines Interface und einer Factory-Methode an. Das Zieldesign ist in [Abbildung 10-7](#) gezeigt.



Modularisierung – ein Beispiel

The screenshot shows the IntelliJ IDEA interface with two projects open:

- jigsawapp** (C:\Users\michael\IdeaProjects\jigsawapp):
 - src folder:
 - com package:
 - inden folder:
 - javaprofi folder:
 - MessageExample.java
 - module-info.java
 - jigsawapp.iml
 - services** (C:\Users\michael\IdeaProjects\services):
 - src folder:
 - com package:
 - services folder:
 - api folder:
 - MessageService.java
 - MessageServiceFactory.java
 - impl folder:
 - SimpleMessageService.java
 - module-info.java

Two code editors are visible:

 - jigsawapp\...\module-info.java**:

```
1 module jigsawapp {  
2     requires services;  
3 }  
4  
5  
6 public class MessageExample {  
7     public static void main(final String[] args) {  
8         System.out.println("Generated msg: " +  
9             MessageServiceFactory.  
10            .createMessageService()  
11            .generateMessage());  
12    }  
13 }
```
 - services\...\module-info.java**:

```
1 module services {  
2     exports com.services.api;  
3 }  
4  
5  
6 package com.services.api;  
7  
8 import com.services.impl.SimpleMessageService;  
9  
10 public class MessageServiceFactory {  
11     public static MessageService createMessageService() {  
12         return new SimpleMessageService();  
13     }  
14 }
```



Modularisierung – ein Beispiel

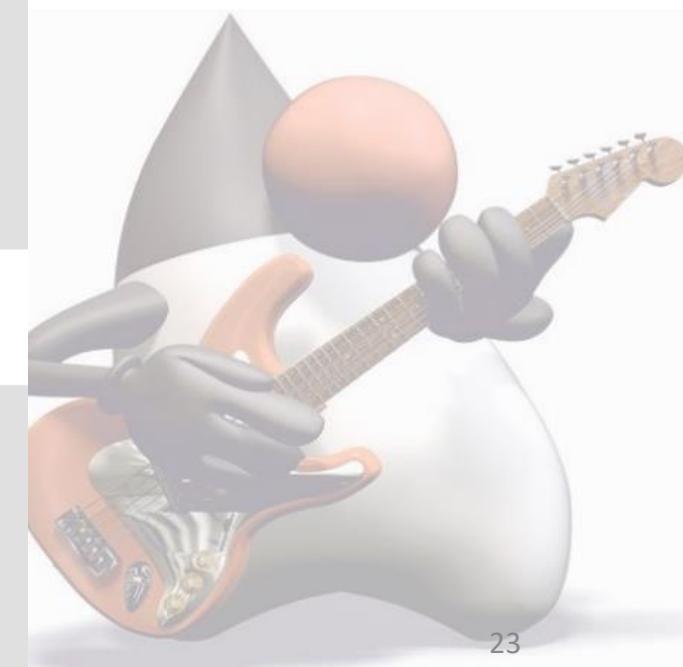
```
javac -d build --module-source-path src $(find src -name '*.java')
```

Nach wie vor lässt sich das Programm mit

```
java -p build -m jigsawapp/com.inden.javaprofi.MessageExample
```

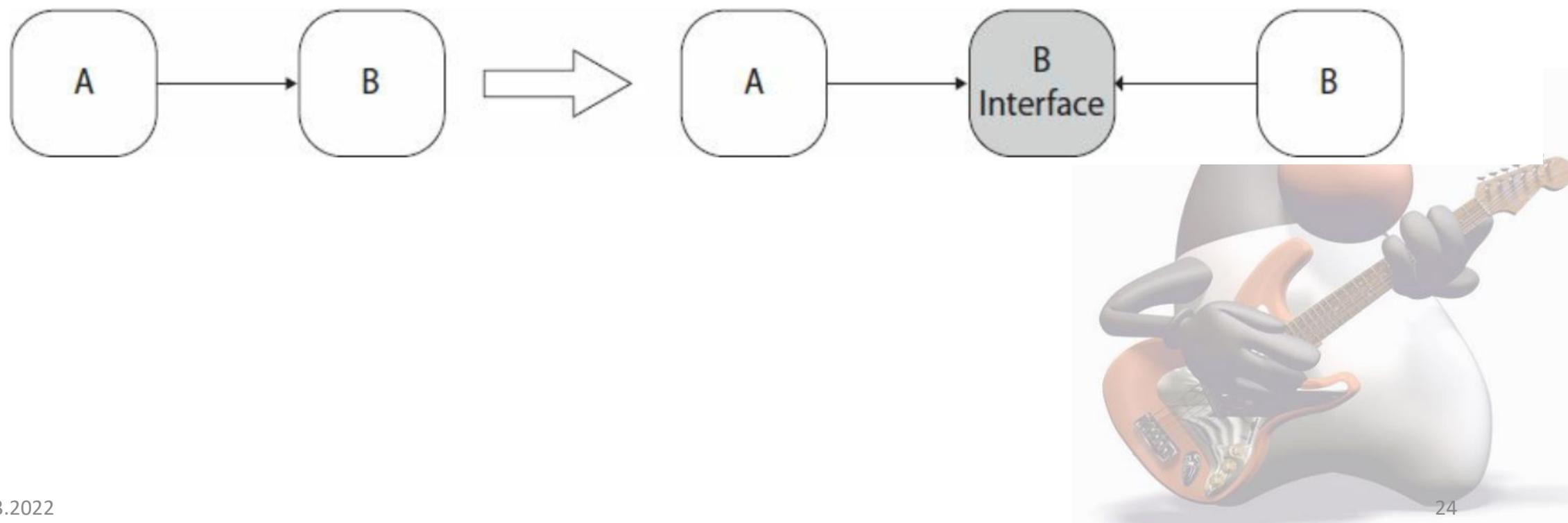
starten und wir erhalten die bereits bekannte Konsolenausgabe:

```
Generated msg: Message from module services!
```



Modularisierung – ein Beispiel

Fazit Wir sind funktional immer noch auf demselben Stand wie zuvor, jedoch besitzt unser nutzendes Modul keine (starke) Implementierungsabhängigkeit auf wichtige konkrete Klassen mehr – es verbleibt lediglich die Abhängigkeit auf die Factory-Klasse und das Interface. Eine Variante, wie man die Kopplung weiter reduziert, sind Services, die in [Abschnitt 11.2](#) vorgestellt werden. Eine andere wäre es, ein Modul mit Interface-Definitionen einzuführen, auf das dann beide Module verweisen. Dies folgt dem Dependency Inversion Principle und ist exemplarisch in [Abbildung 10-8](#) angedeutet.



Modularisierung – ein Beispiel

```
mkdir lib
```

```
jar --create --file lib/jigsawapp.jar -C build/jigsawapp .
```

```
jar --create --file lib/services.jar --module-version 1.0 -C build/services .
```

- --create – Archiv erzeugen⁶
- --file – Der Name der zu erzeugenden Archivdatei
- --module-version – Modulversionsnummer (optional und rein informativ)
- -C – Die Dateien aus dem angegebenen Verzeichnis verwenden

```
ch10_2_3_packaging_module_example
```

```
‘-- lib
```

```
    |-- jigsawapp.jar
```

```
    '-- services.jar
```



Modularisierung – ein Beispiel

Wie schon erwähnt, wollen wir ein JAR erzeugen, das ein direktes Starten erlaubt. Dazu nutzen wir folgendes Kommando mit dem Parameter --main-class:

```
jar --create --file lib/jigsawapp.jar \
--main-class com.inden.javaprofi.MessageExample -C build/jigsawapp .
```

Nun können wir den Klassennamen beim Starten des Programms weglassen und lediglich den Modulnamen angeben:

```
java -p lib -m jigsawapp
```

Modularisierung – ein Beispiel

Nun wollen wir mithilfe von jlink ein Executable erstellen, das sämtliche Applikationsklassen sowie die zur Ausführung benötigten Bestandteile aus dem JDK bündelt.

Für unser Beispiel schreiben wir dann zur Angabe der ausführbaren Klasse mit dem Parameter --launcher im Verzeichnis ch10_2_4_linking_module_example Folgendes:

```
jlink --module-path $JAVA_HOME/jmods:lib --add-modules jigsawapp \
      --launcher jigsawapp=jigsawapp/com.inden.javaprofi.MessageExample \
      --output exec_example
```

Zum Starten des Programms gibt man Folgendes ein:

```
./exec_example/bin/jigsawapp
```



Modularisierung – ein Beispiel

Hinweis: Ausführen von Kommando aus der Java-Installation

Das JDK bringt einige Kommandozeilenprogramme mit, wie das eben erwähnte jlink. Während die beiden bisher vornehmlich genutzten Kommandos javac und java automatisch nach einer Installation eines JDKs verfügbar sind, sollte man für die anderen noch sicherstellen, dass die beiden Umgebungsvariablen JAVA_HOME und PATH korrekt gesetzt sind.

```
export JAVA_HOME=$(/usr/libexec/java_home)
```

```
export PATH=$PATH:$JAVA_HOME/bin
```

Die Webseite <https://www.baeldung.com/java-home-on-windows-7-8-10-mac-os-x-linux> bietet detaillierte Hinweise zur Einstellung der relevanten Pfade für viele Betriebssysteme.

Modularisierung – jdeps

Oftmals möchte man einen genaueren Blick auf die Abhängigkeiten werfen. Dabei kann das Tool `jdeps` unterstützen. Nachfolgend wollen wir damit die beiden zuvor erzeugten modularen JARs untersuchen:

```
jdeps lib/*.jar
```

```
jigsawapp
```

```
[file:///Users/michaeli/Desktop/PureJava9/quelltext/jigsaw_ch10/
```

```
ch10_2_5_dependencies_module_example/lib/jigsawapp.jar]
```

```
requires mandated java.base (@9-ea)
```

```
requires services
```

```
jigsawapp -> java.base
```

```
jigsawapp -> services
```

```
com.inden.javaprofi -> com.services.api services
```

```
com.inden.javaprofi -> java.io java.base
```

```
com.inden.javaprofi -> java.lang java.base
```

```
com.inden.javaprofi -> java.lang.invoke java.base
```

```
services
```

```
[file:///Users/michaeli/Desktop/PureJava9/quelltext/jigsaw_ch6/
```

```
ch10_2_5_dependencies_module_example/lib/services.jar]
```

```
requires mandated java.base (@9-ea)
```

```
services -> java.base
```

```
com.services.api -> com.services.impl services
```

```
com.services.api -> java.lang java.base
```

Modularisierung – jdeps

```
jdeps -s lib/*.jar
```

Dieses Kommando produziert folgende übersichtliche Ausgabe:

```
jigsawapp -> java.base
```

```
jigsawapp -> services
```

```
services -> java.base
```

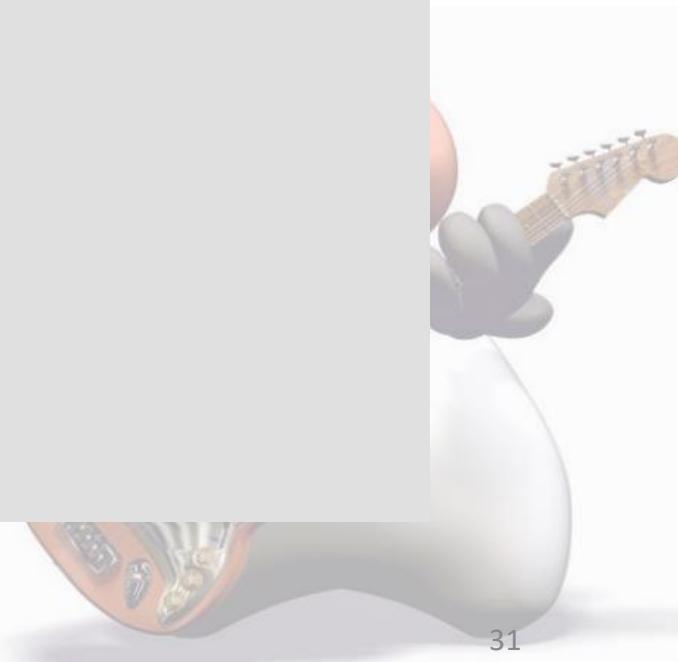


Modularisierung – jdeps

```
jdeps -dotoutput graphs lib/*.jar
```

Damit werden verschiedene Dateien im Verzeichnis **graphs** erzeugt:

```
ch10_2_5_dependencies_module_example  
‘-- graphs  
    |-- jigsawapp.dot  
  
    |-- services.dot  
  
    ‘-- summary.dot
```



Modularisierung – Arten von Modules

- **Named Platform Modules** – Bekanntermaßen wurde im Rahmen von Project Jigsaw auch das JDK in Module unterteilt. Dort findet man rund 80 Module, die Sie sich mit dem Kommando `java --list-modules` auflisten lassen können. Diese speziellen Module des JDKs haben keinen Zugriff auf den Module-Path.
- **Named Application Modules** – Auch Named Application Modules haben wir schon in verschiedenen Beispielen selbst erstellt. Darunter werden Module verstanden, die Anwendungen oder Bibliotheken bündeln. Im Speziellen sind dies modulare JARs, also solche, die in ihrem JAR einen Moduldeskriptor in Form der Datei `module-info.class` enthalten. Diese können auf alle über den Module-Path erreichbaren Module zugreifen, nicht jedoch auf Klassen aus dem CLASSPATH.
- **Open Modules** – Sie entsprechen der vorherigen Modulart, allerdings geben Open Modules alle Packages für Reflection nach außen frei (vgl. [Abschnitt 11.3.1](#)).
- **Automatic Modules** – Für die Migration von Anwendungen ist es von Vorteil, dass man auch gewöhnliche JAR-Dateien, also ohne `module-info.class`, im Module-Path angeben kann. Diese JAR-Dateien werden zu Automatic Modules. Diese exportieren alle ihre Packages, außerdem sind sämtliche Module aus dem Module-Path sowie JARs aus dem CLASSPATH verwendbar. Zudem können Named Application Modules auf Automatic Modules per `requires` zugreifen.



Modularisierung – Arten von Modules

- **Unnamed Modules** – Ergänzend zum Module-Path kann man sowohl beim Kompilieren als auch beim Programmstart einen CLASSPATH angeben. Alle dort vorhandenen Typen werden zu dem sogenannten Unnamed Module zusammengefasst. Enthält der CLASSPATH modulare JARs, so werden diese wie normale JARs ohne Modularisierung behandelt: Sie exportieren alle enthaltenen öffentlichen Typen. Somit können alle öffentlichen Typen aus einem Unnamed Module direkt aufeinander zugreifen, also ohne Sichtbarkeitsschutz. Zudem gilt Folgendes: Automatic Modules können auf das Unnamed Module zugreifen, aber Named Application Modules können dies nicht. Dieser Sachverhalt ist für die schrittweise Migration wichtig. So kann man ein benötigtes JAR als Automatic Module einbinden und dessen Abhängigkeiten aus dem CLASSPATH erfüllen.



Modularisierung – Arten von Modules

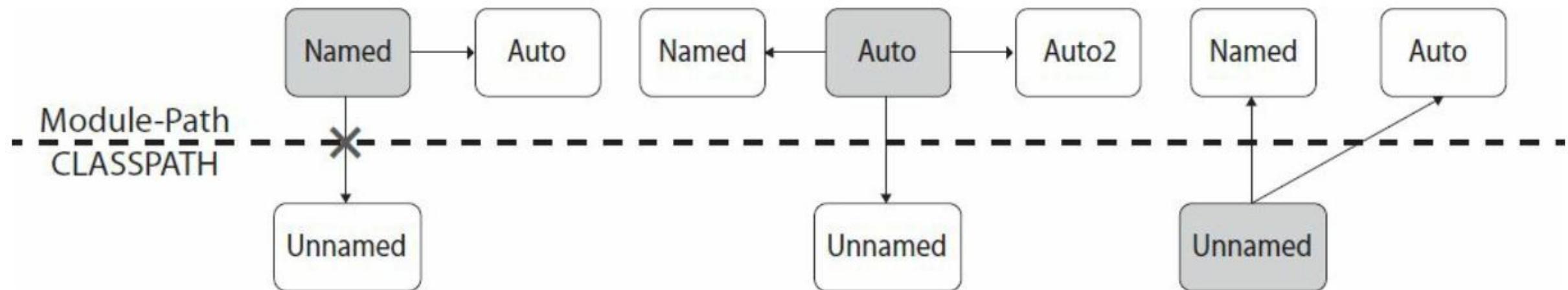
Nachfolgende Tabelle zeigt, wie sich die Art von Modulen auf deren Exporte sowie die Readability auswirkt.

Tabelle 10-1 Modultyp, Exporte und Readability

Modultyp	Ursprung	Exporte	Readability
Platform	JDK	durch exports	-/-
Application	JAR mit Moduldeskriptor	durch exports	Platform, Application, Automatic
Automatic	JAR ohne Moduldeskriptor	alle Packages	Platform, Application, Automatic, Unnamed
Unnamed	alle JARs im CLASSPATH	alle Packages	Platform, Application, Automatic

Modularisierung – Arten von Modules

Um das auch grafisch zu veranschaulichen, hilft [Abbildung 10-13](#). Hier wird deutlich, welche Zugriffe aus welchen Arten von Modulen erlaubt sind. Dabei ist das jeweils relevante Modul leicht dunkler eingefärbt. Im Beispiel ist mit Named ein Named Application Module gemeint und nicht eines aus dem JDK.



Modularisierung – Arten von Modules

Abhangigkeit der Modularart vom Ablageort

Typ des JARs	Im Module-Path	Im CLASSPATH
Modulares JAR	Named Module	Unnamed Module
Normales JAR	Automatic Module	Unnamed Module

Besonderheit: Explizite und implizite Named Modules

Bei der Bezeichnung von Named Modules zeigt sich eine Uneindeutigkeit. Die gerade als Named Modules vorgestellten Module sind explizit als solche definiert. Zur besseren Abgrenzung nennt man Automatic Modules auch ***Implicit Named Modules*** und die gewohnlichen Named Modules nennt man ***Explicit Named Modules***.

Modularisierung – Sichtbarkeit

Sichtbarkeit vor Java 9

Rekapitulieren wir zunächst den Status quo in Java 8: Bekanntermaßen besaßen Typen damals eine der folgenden vier Sichtbarkeiten:

- **private** – Nur in der eigenen Klasse sichtbar
- ***default / package private*** (kein Schlüsselwort) – Nur im eigenen Package sichtbar
- **protected** – Wie *package private*, aber auch in abgeleiteten Klassen sichtbar
- **public** – Aus allen Packages zugreifbar

Weil **public** bedeutet, dass ein Typ im **CLASSPATH** für alle anderen Typen zugänglich ist, kann man hierfür nicht wirklich von einer Sichtbarkeitssteuerung sprechen.



Modularisierung – Sichtbarkeit

Sichtbarkeit ab Java 9

Seit Java 9 lässt sich die Sichtbarkeit von Typen genauer als in Java 8 spezifizieren. Relevant ist dies in der Regel nur für die als `public` definierten Typen:

- **Global** — `public` für alle Module – Wenn das Package einer `public` definierten Klasse mit `exports` zum Zugriff freigegeben wurde, ist diese von allen anderen Modulen zugreifbar, die auf dieses Modul per `requires` verweisen.
- **Eingeschränkt** auch ***Qualified Export*** genannt – `public` für angegebene Module – Mithilfe von `exports to` kann eine Liste von Modulen spezifiziert werden, die Zugriff erhalten sollen, sofern sie auf dieses Modul per `requires` verweisen.
- **Modul intern** – `public` im Modul selbst – Alle Klassen aus Packages, die nicht in `exports` aufgeführt sind, sind nur innerhalb des eigenen Moduls zugreifbar.

Den ersten und dritten Fall haben wir schon in verschiedenen Beispielen kennengelernt. Der zweite Fall ist ein Spezialfall des ersten: Allgemein geht es darum, dass man das Exportieren von Klassen auf eine vordefinierte Menge an Modulen beschränken kann. Ein Bild sagt mehr als tausend Worte. Somit sollte sich Ihr Verständnis nach einem Blick auf die folgende [Abbildung 10-14](#) verbessern.

Modularisierung – Sichtbarkeit

```
module specialexport
{
    exports com.inden.utils to myapp;
    exports com.inden;
}
```

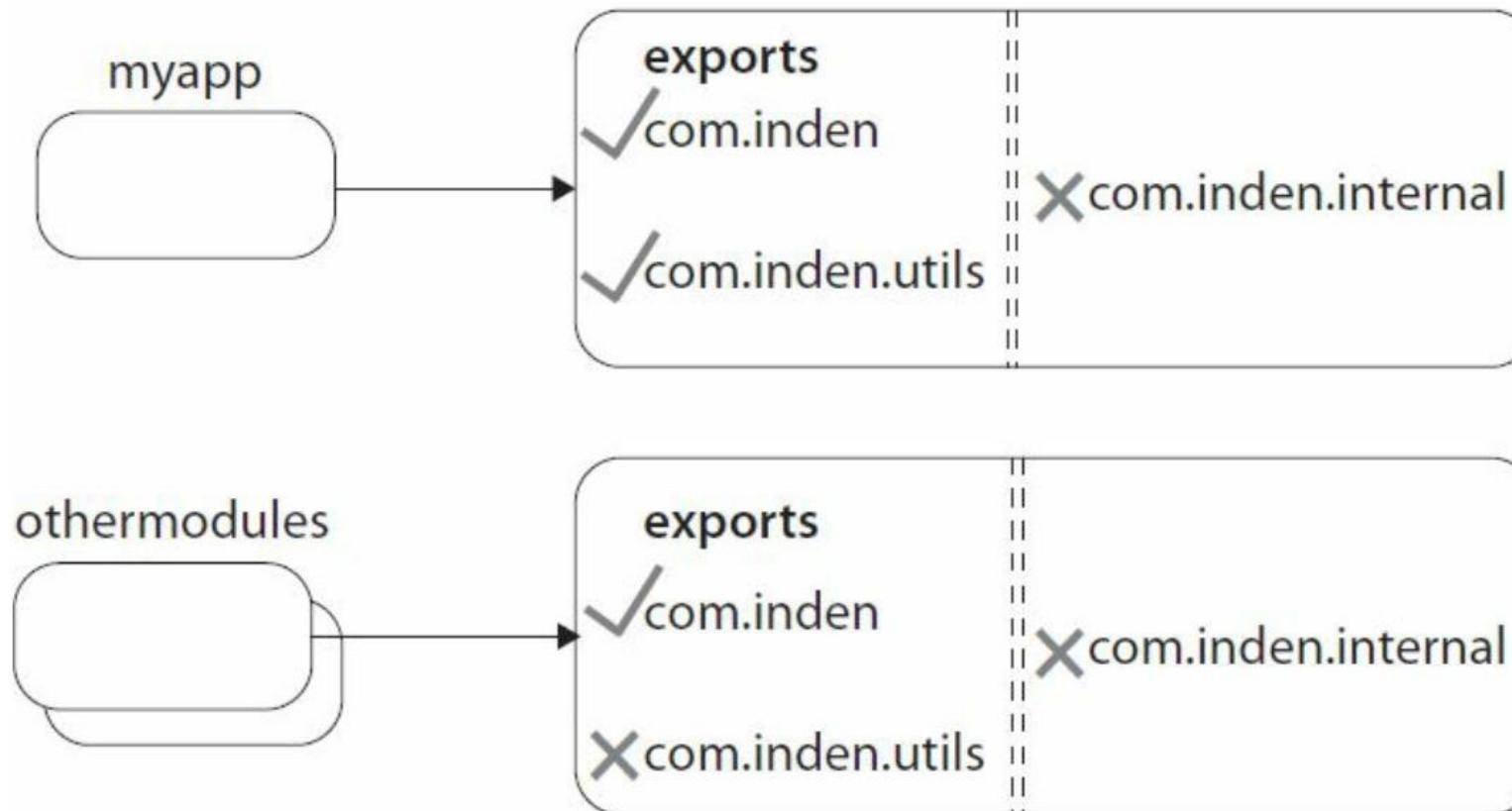


Abbildung 10-14 Auswirkungen der drei Sichtbarkeiten bei public



Modularisierung – Qualified Exports

Gerade zuvor haben wir nochmals erkannt, dass Zugriffe auf in den Moduldeskriptoren nicht exportierte Packages verhindert werden. Nachfolgend schauen wir auf die explizite Freigabe von Packages an spezielle Module, wodurch interne Funktionalität auf eine gewisse Nutzergruppe begrenzt werden kann. Einen solchen eingeschränkten Export (Qualified Export) für eine definierte Menge an Modulen notiert man bekanntermaßen wie folgt:

```
exports <package_to_export> to <modulename>;
```



Modularisierung – Qualified Exports

```
module timeserver
{
    requires java.logging;
    exports com.server;
    exports com.server.internal to privatetimeclient, timeclient;
}
```



Modularisierung – Qualified Exports

```
module timeserver
```

```
{
```

```
    requires java.logging;
```

```
    exports com.server;
```

```
    exports com.server.internal to privatetimeclient, timeclient;
```

```
}
```



Modularisierung – Transitive Abhängigkeiten

Die in Moduldeskriptoren beschriebenen Abhängigkeiten werden nicht automatisch an nutzende Module propagiert. Immer dann, wenn mehrere Module kombiniert werden, kann dies umständlich werden. Betrachten wir dazu drei Module: Modul 1 verweist auf Modul 2 und verwendet bei der Implementierung Typen aus Modul 2, etwa in einem Interface Type_Module1 die beiden Typen TypeA_Module2 und TypeB_Module2:

```
interface Type_Module1  
{  
    TypeA_Module2 doSomething(TypeB_Module2 someVar);  
}
```



Modularisierung – Transitive Abhängigkeiten

Würde ein Modul 3 nun auf Modul 1 verweisen, so müsste dieses zusätzlich auch immer noch auf Modul 2 verweisen, um die Abhängigkeit zu erfüllen. Mit der sogenannten ***Implied Readability*** kann man eine transitive Abhängigkeit wie folgt beschreiben:

```
module Module1
{
    requires transitive Module2;
}
```

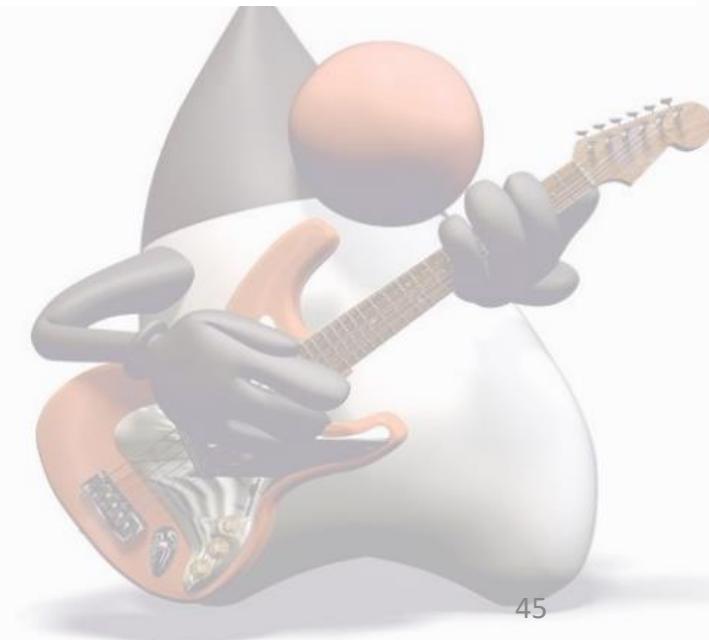


Modularisierung – Transitive Abhängigkeiten

Aggregator Module und transitive Abhängigkeiten

Oftmals ist es praktisch, verschiedene Module zu größeren Einheiten zu bündeln. Das ist mithilfe von requires transitive möglich. Damit lassen sich sogenannte **Aggregator Modules** erstellen, die keinen eigenen Inhalt besitzen, sondern vor allem dazu dienen, Abhängigkeiten zu anderen Modulen zentral unter einem spezifischen Namen zu bündeln.

Das ist insbesondere praktisch, weil so weniger direkte Abhängigkeiten notiert werden müssen und die Aggregator Modules sinnvolle Namen zur Strukturierung tragen können.



Modularisierung – Reflection

- **Open Modules** – Module können durch das Schlüsselwort `open` vor `module` im Moduldeskriptor explizit und vollständig für Reflection geöffnet werden.
- **Das Schlüsselwort `opens`** – Im Moduldeskriptor können nach `opens` aufgezählte Packages explizit für den externen Zugriff per Reflection geöffnet werden. Dazu gibt es auch die Variante spezifischer Öffnung für bestimmte Module.
- **Kommandozeilenparameter `--add-opens`** – Mithilfe des Kommandozeilenparameters `--add-opens` können bei der Programmausführung einzelne Packages explizit für Reflection geöffnet werden.



Modularisierung – Reflection

Open Modules Mitunter sollen ganze Module, genauer alle Packages eines Moduls, zur Laufzeit per Reflection zugreifbar sein. Dies wird durch das Schlüsselwort `open` vor der Moduldefinition wie folgt möglich:

```
open module reflectionuser
```

```
{
```

```
    requires reflectionutils;
```

```
}
```



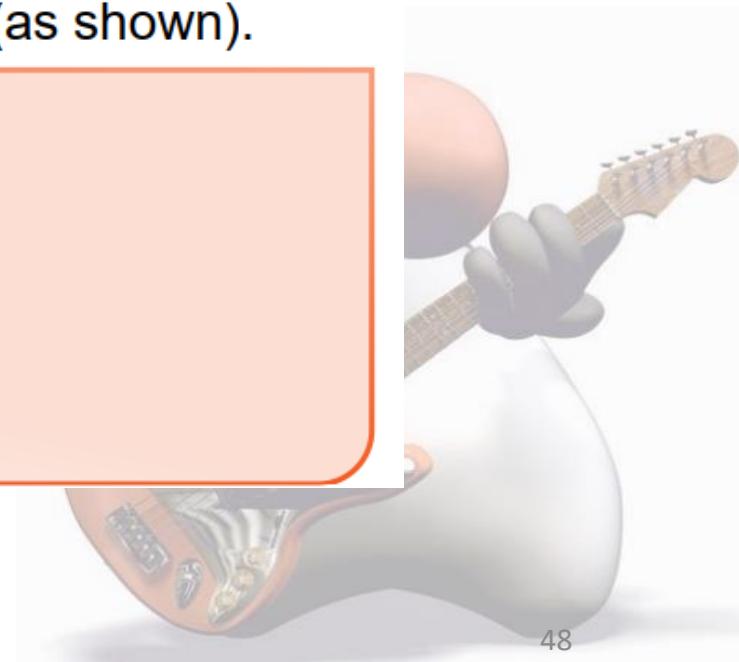
Modularisierung – Reflection

- Open the entire module.

```
open module soccer {  
    requires java.logging;  
    exports soccer;  
    exports util;  
}
```

- Open just the package needed to all modules or to a specific module (as shown).

```
module soccer {  
    requires java.logging;  
    exports soccer;  
    exports util;  
    opens soccer to jackson.databind;  
}
```



Modularisierung – Services

- **Definition eines Service Interface**

Ein Service Interface kann man in Form einer abstrakten Klasse oder idealerweise eines Interface definieren.

Dieses Service Interface wird sinnvollerweise in einem eigenen Modul bereitgestellt.



Modularisierung – Services

```
public interface PlaylistService {  
    public List<String> getTitles();  
    public List<String> searchByArtist(final String artist);  
}
```

```
module playlistservice  
{  
    exports com.javaprofi.spi;  
}
```



```
public class RockPlaylistService implements PlaylistService {  
    private final Map<String, List<String>> songMap = Map.of(  
        "Bryan Adams", List.of("Summer of '69"),  
        "Bon Jovi", List.of("Livin' On A Prayer"),  
        "Metallica", List.of("Nothing Else Matters"),  
        "Nickelback", List.of("How You Remind Me"),  
        "Toto", List.of("Africa", "Hold The Line"));  
  
    @Override  
    public List<String> getTitles() {  
        final Stream<List<String>> titlesStream = songMap.values().stream();  
        final List<String> allTitles = titlesStream.reduce(new ArrayList<>(),  
            (list1, list2) -> {list1.addAll(list2); return list1});  
        return allTitles;  
    }  
  
    @Override  
    public List<String> searchByArtist(final String artist) {  
        return songMap.getOrDefault(artist,  
            List.of("No song found"));  
    }  
}  
  
module playlistserviceprovider  
{  
    requires playlistservice;  
    provides com.javaprofi.spi.PlaylistService  
        with com.javaprofi.services.RockPlaylistService;  
}
```



Modularisierung – Services

```
public class ServiceConsumerExample {  
    public static void main(final String[] args) throws Exception {  
        final Optional<PlaylistService> optService =  
            lookup(PlaylistService.class);  
        optService.ifPresentOrElse(service ->  
            {  
                System.out.println("Service: " + service.getClass());  
                final List<String> allTitles = service.getTitles();  
                System.out.println("All titles: " + allTitles);  
            },  
            () -> System.err.println("No service provider found!"));  
    }  
}
```

```
private static <T> Optional<T> lookup(final Class<T> clazz) {  
    final Iterator<T> iterator = ServiceLoader.load(clazz).iterator();  
    if (iterator.hasNext()) {  
        return Optional.of(iterator.next());  
    }  
    return Optional.empty();  
}
```

```
module com.serviceconsumer  
{  
    requires playlistservice;  
    uses com.javaprofi.spi.PlaylistService;  
}
```



Modularisierung – Services

Fazit

Wir haben in drei Schritten die Definition eines Service Interface, eines Service Provider und eines Service Consumer in jeweils eigenständigen Modulen nachvollzogen. Die so realisierte Applikation bietet eine lose Kopplung. Zudem ist die umgesetzte Funktionalität identisch zu der zuvor mithilfe der Klasse ServiceLoader realisierten Variante.



Java Modules

