7.2 Modularisierung und Reflection

Wir haben mittlerweile ein gutes Verständnis für die Modularisierung aufgebaut. Nun wollen wir uns anschauen, wie die neuen Sprachkonstrukte mit Reflection zugänglich sind und verarbeitet werden können. Wir konzentrieren uns hier auf die Typen `java.lang.Module` und `java.lang.module.ModuleDescriptor`. Darüber hinaus lernen wir aber auch ein paar weitere Klassen und Interfaces kennen, die wir dann geeignet kombinieren, um einen Lookup des zu einer Klasse gehörigen Moduls zu realisieren. Zum Abschluss implementieren wir eine JavaFX-Applikation, die eine Liste von Imports in eine Liste von `requires`-Angaben konvertieren kann.

7.2.1 Verarbeitung von Modulen mit Reflection

Im Anschluss schauen wir uns einige wichtige Klassen und Interfaces an, um mit Reflection auf Module und deren Moduldeskriptoren zugreifen zu können.

Die Typen `Module` und `ModuleDescriptor`

Die Klasse `Module` repräsentiert die mit Project Jigsaw in das JDK neu eingeführten Module. Diese besitzen vor allem einen Namen und einen Moduldeskriptor, der durch die Klasse `ModuleDescriptor` zugreifbar ist.

Anhand eines Beispiels wollen wir diverse Methoden von `Module` und `ModuleDescriptor` im Einsatz erleben und damit die JDK-Module `java.logging` und `java.rmi` ein wenig untersuchen. Zum Ermitteln der Module nutzen wir die Klasse `java.lang.ModuleLayer` und deren statische Methode `boot()`. Anschließend gewährt uns ein Aufruf von `findModule(String)` Zugriff auf Module:

```
public static void main(final String[] args) throws ClassNotFoundException
{
    final Optional<Module> optRmiModule =
                           ModuleLayer.boot().findModule("java.rmi");
    final Optional<Module> optLogModule =
                           ModuleLayer.boot().findModule("java.logging");

    optLogModule.ifPresent(logModule ->
    {
        optRmiModule.ifPresent(rmiModule -> printModuleInfo(logModule,
                                                            rmiModule));
        printModuleDescriptorInfo(logModule.getDescriptor());
    });
}

private static void printModuleInfo(final Module logModule,
                                    final Module rmiModule)
{
    System.out.println("Module: " + logModule);
    System.out.println("Packages: " + logModule.getPackages());
    System.out.println("log canRead RMI: " + logModule.canRead(rmiModule));
    System.out.println("RMI canRead log: " + rmiModule.canRead(logModule));
}
```

Listing 7.2 *Ausführbar als '***MODULEINSPECTOR***'*

Die Ausgabe der Informationen zum Moduldeskriptor schreiben wir lesbar wie folgt:

```
private static void printModuleDescriptorInfo(final ModuleDescriptor descriptor)
{
    System.out.println("\nDescriptor: " + descriptor);
    System.out.println("\nrequires:  " + descriptor.requires());
    System.out.println("provides:  " + descriptor.provides());
    System.out.println("exports:   " + descriptor.exports());
    System.out.println("automatic: " + descriptor.isAutomatic());
    System.out.println("packages:  " + descriptor.packages());
}
```

Starten wir das Programm MODULEINSPECTOR, so erhalten wir folgende Ausgaben:

```
Module: module java.logging
Packages: [sun.net.www.protocol.http.logging, sun.util.logging.resources, java.
      util.logging, sun.util.logging.internal]
log canRead RMI: false
RMI canRead log: true

Descriptor: module { name: java.logging@9-ea, [mandated java.base], exports: [
      java.util.logging], provides: [jdk.internal.logger.DefaultLoggerFinder with
       [sun.util.logging.internal.LoggingProviderImpl]] }

requires:  [mandated java.base]
provides:  [jdk.internal.logger.DefaultLoggerFinder with [sun.util.logging.
      internal.LoggingProviderImpl]]
exports:   [java.util.logging]
automatic: false
packages:  [sun.net.www.protocol.http.logging, sun.util.logging.resources, java.
      util.logging, sun.util.logging.internal]
```

Die Ausgaben zeigen zunächst die Packages des Moduls `java.logging` und, ob das Modul `java.rmi` das Modul `java.logging` lesen kann und umgekehrt. Schließlich geben wir Informationen zum `ModuleDescriptor` des Moduls `java.logging` aus.

Die Typen `ModuleFinder` und `ModuleReference`

Nachfolgend wollen wir uns das Interface `java.lang.module.ModuleFinder` und die Klasse `java.lang.module.ModuleReference` anschauen. Mit einem `ModuleFinder` kann man Module aus Verzeichnissen oder diejenigen des Systems, also des JDKs, ermitteln. Eine `ModuleReference` bietet Zugriff auf Name und Verzeichnis sowie den Moduldeskriptor eines Moduls.

Eine `ModuleFinder`-Instanz erhält man durch eine der beiden folgenden statischen Methoden `of()` und `ofSystem()`:

```
final ModuleFinder finder =  ModuleFinder.of(dir1, dir2, dir3);
final ModuleFinder systemFinder = ModuleFinder.ofSystem();
```

Die Klasse `ModuleFinder` bietet die Methode `findAll()`. Deren Aufruf liefert ein `Set<ModuleReference>`. Basierend auf dieser Menge von Instanzen ist der Moduldeskriptor durch die Methode `descriptor()` zugreifbar.

Zur Demonstration ermitteln wir alle Systemmodule, sortieren diese anhand ihrer Moduldeskriptoren und geben die ersten sieben davon aus:

```
public static void main(final String[] args) throws ClassNotFoundException
{
    final ModuleFinder systemFinder = ModuleFinder.ofSystem();
    final Set<ModuleReference> modules = systemFinder.findAll();
    modules.stream()
           .sorted(Comparator.comparing(ModuleReference::descriptor))
           .limit(7)
           .forEach(System.out::println);
}
```

Listing 7.3 *Ausführbar als '*MODULEFINDEREXAMPLE*'*

Startet man das Programm MODULEFINDEREXAMPLE, so kommt es zu folgender Ausgabe, die sieben `ModuleReference`s auflistet:

```
[module java.activation, location=jrt:/java.activation]
[module java.base, location=jrt:/java.base]
[module java.compiler, location=jrt:/java.compiler]
[module java.corba, location=jrt:/java.corba]
[module java.datatransfer, location=jrt:/java.datatransfer]
[module java.desktop, location=jrt:/java.desktop]
[module java.instrument, location=jrt:/java.instrument]
```

7.2.2 Tool zur Ermittlung von Modulen zu Klassen

In diesem Abschnitt kombinieren wir das bisher Gelernte, um einen Lookup eines Moduls zu einer Klasse zu realisieren. Zum Abschluss implementieren wir mit diesem Wissen eine JavaFX-Applikation, die eine Konvertierung einer Liste von Import-Anweisungen (von JDK-Typen) in eine Liste von `requires`-Angaben vornimmt. Das ist insbesondere für die Migration bestehender Anwendungen hilfreich.

Ermittlung des Moduls einer Klasse

In Abschnitt 3.2.9 führte der Aufruf von `getModule()` zur Abfrage des Moduls zu der Klasse `ScrollPane` wie folgt zu einer Exception:

```
final Class<?> clazz = Class.forName("javafx.scene.control.ScrollPane");
System.out.println(clazz.getModule());
```

Dieses Problem existiert auch für andere Klassen des JDKs. Wir wollen nun eine allgemeingültige Möglichkeit einsetzen, um das Modul zu einer Klasse zu ermitteln.

Zuvor haben wir schon kennengelernt, dass man ein Modul mithilfe seines Namens folgendermaßen suchen kann:

```
final Optional<Module> optModule = ModuleLayer.boot().findModule(name);
```

Was uns jetzt noch fehlt, ist das programmatische Auffinden aller JDK-Module. Als Ausgangspunkt dient wieder die Klasse `ModuleLayer` und deren `boot()`-Methode. Eine Instanz der Klasse `java.lang.module.Configuration` erhält man durch Aufruf der Methode `configuration()`, die uns durch Aufruf von `modules()` ein `Set<ResolvedModule>` bereitstellt:

```
final Configuration config = ModuleLayer.boot().configuration();
final Set<ResolvedModule> modules = config.modules();
```

Damit haben wir mit `java.lang.module.ResolvedModule` neben `Module` und `ModuleReference` die dritte Repräsentation eines Moduls im Kontext von Modularisierung und Reflection kennengelernt. Ein `ResolvedModule` liefert über `name()` seinen Namen und damit schließt sich der Kreis: Mit dem Namen können wir per `findModule(String)` suchen.

Prototyp eines Module-Lookup-Tools Mit diesem Wissen implementieren wir ein Analysetool, das zu einem gegebenen voll qualifizierten Namen eines Typs das zugehörige Modul des JDKs ermittelt. Diese Aufgabe erledigt die Kombination der selbst geschriebenen Methoden `findModuleByName(String)` und `tryFindClass(ResolvedModule, String)`. Die Methode `findModuleByName(String)` prüft für alle JDK-Module durch Aufruf von `tryFindClass(ResolvedModule, String)`, ob der als Name übergebene Typ im jeweiligen Modul gefunden wird. Das Ganze implementieren wir folgendermaßen:

```
private static Optional<ResolvedModule> findModuleByName(final String className)
{
    final Configuration config = ModuleLayer.boot().configuration();

    final Set<ResolvedModule> modules = config.modules();
    return modules.stream()
                  .filter(module -> tryFindClass(module, className))
                  .findFirst();
}

public static boolean tryFindClass(final ResolvedModule resModule,
                                   final String name)
{
   final Optional<Module> optModule =
                          ModuleLayer.boot().findModule(resModule.name());

   if (optModule.isPresent())
   {
       final Class<?> clazz = Class.forName(optModule.get(), name);
       return clazz != null;
   }
   return false;
}
```

Um die Methoden in Aktion zu erleben, definieren wir eine Liste von voll qualifizierten Klassennamen, für die wir jeweils das korrespondierende Modul ermitteln:

```
public static void main(final String[] args) throws ClassNotFoundException
{
    // JDK 9: List.of() => Collection-Factory-Methode
    final List<String> classNames = List.of("java.util.logging.Logger",
                                            "java.time.LocalDate",
                                            "javafx.application.Application",
                                            "javafx.scene.control.ScrollPane");

    final Set<String> jdkModules = new TreeSet<>();

    for (final String className : classNames)
    {
        final Optional<ResolvedModule> optModule = findModuleByName(className);
        optModule.ifPresent(mod -> jdkModules.add(mod.name()));
    }

    System.out.println("Classes: " + classNames);
    System.out.println("Modules: " + jdkModules);
}
```

Listing 7.4 *Ausführbar als* **'MODULEFORCLASSFINDER'**

Startet man das Programm MODULEFORCLASSFINDER, so werden zu den Klassen die benötigten Module ermittelt und ausgegeben (hier etwas netter ausgerichtet):

```
Classes: [java.util.logging.Logger, java.time.LocalDate,
          javafx.application.Application, javafx.scene.control.ScrollPane]
Modules: [java.base, java.logging, javafx.controls, javafx.graphics]
```

7.2.3 Konvertierungstool für `import` zu `requires`

Wir haben nun alle benötigten Informationen zusammen, um ein ausgefeilteres Konvertierungstool mit GUI zu erstellen. Ziel dabei ist es, die Imports eines beliebigen Java-Programms in ein Textfeld kopieren zu können. Das Tool soll daraus die benötigten voll qualifizierten Klassennamen extrahieren und die zugehörigen Module ermitteln.[3] Selbstverständlich möchte man bei den benötigten Modulen potenzielle Duplikate aussondern. Bei den Berechnungen setzen wir diverse Stream-Funktionalitäten ein und profitieren von den vielfältigen Erweiterungen aus Java 8.[4] Die ermittelten Modulnamen werden gleich als geordnete Menge an Modulnamen so aufbereitet, dass sie per Copy-Paste in den gewünschten Moduldeskriptor übertragen werden können. Die Implementierung des Tools übernimmt viele Bausteine aus dem zuvor entwickelten Programm.

[3] Als kleine Einschränkung sei noch darauf hingewiesen, dass das Tool keine Wildcard-Imports berücksichtigt und nur Typen des JDKs zu Modulen zuordnen kann.

[4] Einen ersten Überblick bietet Anhang A.

```
public class ImportToModuleDescriptorTool extends Application
{
    @Override
    public void start(final Stage primaryStage)
    {
        final Label importsLabel = new Label("Imports:");
        final TextArea importStatementsArea = new TextArea();
        final Button convertButton = new Button("Convert");
        final Label requiresLabel = new Label("ModuleDescriptor:");
        final TextArea requiresStatementsArea = new TextArea();

        GridPane.setHgrow(importStatementsArea, Priority.ALWAYS);
        GridPane.setHgrow(requiresStatementsArea, Priority.ALWAYS);
        GridPane.setVgrow(importStatementsArea, Priority.SOMETIMES);
        GridPane.setVgrow(requiresStatementsArea, Priority.SOMETIMES);

        final GridPane gp = new GridPane();
        gp.setPadding(new Insets(7));
        gp.setHgap(7);
        gp.setVgap(7);
        gp.add(importsLabel, 0, 0);
        gp.add(importStatementsArea, 0, 1);
        gp.add(convertButton, 0, 2);
        gp.add(requiresLabel, 0, 3);
        gp.add(requiresStatementsArea, 0, 4);

        primaryStage.setTitle("ImportToModuleDescriptorTool");
        primaryStage.setScene(new Scene(gp, 525, 400));
        primaryStage.show();

        convertButton.setOnAction(event ->
        {
            final String importStatementText = importStatementsArea.getText();
            final String requiresStatements =
                        convertToRequires(importStatementText);

            requiresStatementsArea.setText(requiresStatements);
        });
    }

    private String convertToRequires(final String importStatementText)
    {
        final List<String> importStatements =
                           List.of(importStatementText.split("\n"));
        final List<String> classNames =
                           extractClassNamesFromImports(importStatements);

        final Set<String> jdkModules = new TreeSet<>();

        for (final String className : classNames)
        {
            final Optional<ResolvedModule> optModule =
                                           findModuleByName(className);
            optModule.ifPresent(mod -> jdkModules.add(mod.name()));
        }

        return createRequiresStatements(jdkModules);
    }
    ...
```

Listing 7.5 *Ausführbar als '***IMPORTTOMODULEDESCRIPTORTOOL***'*

Die kleinteilige Arbeit zur Extraktion der Klassennamen aus den Import-Statements haben wir in die folgende Hilfsmethode ausgelagert:

```
    ...

    private static List<String> extractClassNamesFromImports(final List<String>
          importStatements)
    {
       final Function<String, String> removeImportAndSemicolon =
                                      str -> str.substring(7).replace(" ", "").
                                                              replace(";", "");

       final List<String> classNames = importStatements.stream()
                                       .map(str -> str.trim())
                                       .filter(str -> !str.isEmpty())
                                       .filter(str -> str.startsWith("import "))
                                       .map(removeImportAndSemicolon)
                                       .collect(Collectors.toList());
       return classNames;
    }

    private String createRequiresStatements(final Set<String> jdkModules)
    {
        return jdkModules.stream()
                         .map(name -> "requires " + name + ";")
                         .collect(Collectors.joining("\n"));
    }

    // ...
}
```

Abbildung 7-4 zeigt das Programm IMPORTTOMODULEDESCRIPTORTOOL in Aktion, nachdem bereits eine Konvertierung erfolgt ist.

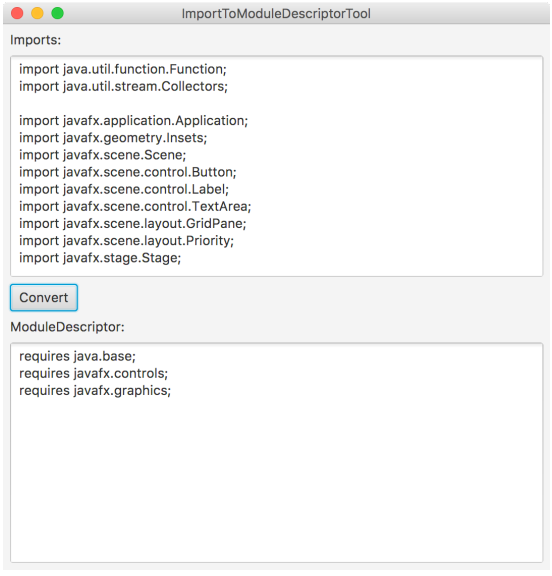

Abbildung 7-4 *Darstellung des Programms* IMPORTTOMODULEDESCRIPTORTOOL

Schlussbemerkung

Dieses Tool kann Ihnen einige Arbeit abnehmen, wenn Sie eine bestehende Applikation in eine modularisierte Applikation umwandeln wollen und dazu die jeweiligen Abhängigkeiten von den Klassen des JDKs in den jeweiligen Moduldeskriptoren aufführen müssen.

Hinweis: Vorgehen beim Entwurf

Grundsätzlich ist es beim Programmieren immer eine gute Idee, komplexere Funktionalität in kleinere zu lösende Probleme aufzubrechen und diese durch einzelne kurze Methoden zu realisieren. Diese folgen oftmals eher dem SINGLE RESPONSIBILITY PRINCIPLE und lassen sich meistens auch gut durch Unit Tests prüfen,[a] da kaum externe Abhängigkeiten existieren. Zudem hilft es, den Überblick zu behalten und sich im Sourcecode leichter zurechtzufinden.

[a] Einen Einstieg in das professionelle Programmieren und insbesondere auch das Schreiben von Unit Tests bietet mein Buch »Der Weg zum Java-Profi« [5].

7.2.4 Besonderheiten bei Reflection

Normalerweise sind alle Typen von nicht exportierten Packages außerhalb ihres Moduls nicht sichtbar und nicht zugreifbar. Das gilt, wie wir schon in Abschnitt 6.3.2 gesehen haben, auch für Reflection, und zwar sowohl für eigene Module als auch für diejenigen des JDKs. Rekapitulieren wir kurz anhand eines Zugriffs auf eine JDK-interne Funktionalität in Form der Methode `getURLStreamHandler()` aus der Klasse `URL`:

```
public static void main(final String[] args) throws Exception
{
    final Method method = URL.class.getDeclaredMethod("getURLStreamHandler",
                                                       String.class);
    method.setAccessible(true);

    System.out.println(method.invoke(null, "http"));
}
```

Listing 7.6 *Ausführbar als* '**REFLECTIONEXAMPLE**'

Die in die JVM integrierte Zugriffskontrolle produziert bei folgende Warnmeldungen:

```
WARNING: An illegal reflective access operation has occurred
WARNING: Illegal reflective access by ch7_2_4.ReflectionExample (file:/Users/
      michaeli/Desktop/PureJava9/quelltext/build/libs/Jdk9-DieNeuerungen.jar) to
      method java.net.URL.getURLStreamHandler(java.lang.String)
WARNING: Please consider reporting this to the maintainers of ch7_2_4.
      ReflectionExample
WARNING: Use --illegal-access=warn to enable warnings of further illegal
      reflective access operations
WARNING: All illegal access operations will be denied in a future release
sun.net.www.protocol.http.Handler@79b4d0f
```

An der obigen Meldung ist zu erkennen, dass kein Zugriff auf das Package `java.net` aus dem Modul `java.base` besteht und es in zukünftigen Java-Versionen auch nicht mehr unterstützt wird – in früheren Vorab-Version von Java 9 war der Zugriff restriktiver geregelt und der obige Aufruf führte zu einer Exception.

Das wurde jedoch zugunsten der besseren Rückwärtskompatibilität abgeschwächt, denn diese Restriktivität bezüglich Sichtbarkeit ist zwar für die Entwicklung von Business-Applikationen sehr gewünscht, um die Kapselung sicherzustellen, doch diese ist für einige Tools und Bibliotheken zu einschränkend. Schauen wir uns dies schrittweise an einem Beispiel an, das mit einer nicht modulariiserten Applikation und dem Einsatz von Reflection beginnt.

Beispiel

Ich möchte Ihnen zunächst drei Klassen zeigen, wobei die eine eine Utility-Klasse ist, die über Reflection den Zugriff selbst auf private Attribute beliebiger Klassen erlaubt. Als zweites sehen wir eine extrem einfache Klasse und zuletzt den nutzenden Sourcecode:

```
public class ReflectionAttributeAccessor
{
    public Object getAttributeValue(final Object object,
                                     final String attributeName)
    {
        try
        {
            final Class<?> clazz = object.getClass();
            final Field field = clazz.getDeclaredField(attributeName);

            field.setAccessible(true);

            return field.get(object);
        }
        catch (ReflectiveOperationException ex)
        {
            throw new IllegalStateException("problems while using reflection");
        }
    }
}
```

Die Beispielklasse ist bewusst nicht clever realisiert, deshalb benötigt es ja die zuvor vorgestellte Utility-Klasse mit dem Reflection-basierten Zugriff auf das Attribut:

```
public class Example
{
    private final String name;

    public Example(final String name)
    {
        this.name = name;
    }
}
```

Schauen wir auf die einsetzende Klasse:

```
public class ReflectionUtilsUsageExample
{
    public static void main(final String[] args)
    {
        final ReflectionAttributeAccessor raa =
                                            new ReflectionAttributeAccessor();
        final Object value =
                    raa.getAttributeValue(new Example("Micha"), "name");

        System.out.println("Value = " + value);
    }
}
```

Startet man das obige Programm, so wird der Wert des Attributs `name` per Reflection ausgelesen und man erhält folgende Ausgabe:

```
Value = Micha
```

Wir sehen nochmals, dass Reflection mit Java 9 ohne modulare Applikation problemlos funktioniert. Fragen wir uns nun: Was passiert aber, wenn wir diese Klassen in unterschiedliche Module überführen?

Besonderheiten im Kontext der Modularisierung

Wir wählen für die Utility-Klasse das Modul `reflectionutils` und für die Applikation und die Domain-Klasse das Modul `relfectionuser`. Damit ergibt sich folgende Verzeichnisstruktur:

```
ch7_2_4_reflection
'-- src
    |-- reflectionuser
    |   |-- com
    |   |   |-- domain
    |   |   |   '-- Example.java
    |   |   '-- reflectionuser
    |   |       '-- ReflectionUtilsUsageExample.java
    |   '-- module-info.java
    '-- reflectionutils
        |-- com
        |   '-- reflectionutils
        |       '-- ReflectionAttributeAccessor.java
        '-- module-info.java
```

Kompilieren wir einfach einmal:

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

Das ist erfolgreich. Doch beim Applikationsstart mit

```
java -p build -m reflectionuser/com.reflectionuser.ReflectionUtilsUsageExample
```

erhalten wir statt einer Ausgabe von »Micha« wie in Falle des nicht modularisierten Programms hier eine `java.lang.reflect.InaccessibleObjectException` wie folgt:

```
Exception in thread "main" java.lang.reflect.InaccessibleObjectException: Unable
      to make field private java.lang.String com.domain.Example.name accessible:
      module reflectionuser does not "opens com.domain" to module
     reflectionutils
    at java.base/java.lang.reflect.AccessibleObject.checkCanSetAccessible(
        AccessibleObject.java:337)
    at java.base/java.lang.reflect.AccessibleObject.checkCanSetAccessible(
        AccessibleObject.java:281)
    at java.base/java.lang.reflect.Field.checkCanSetAccessible(Field.java:176)
    at java.base/java.lang.reflect.Field.setAccessible(Field.java:170)
    at reflectionutils/com.reflectionutils.ReflectionAttributeAccessor.
        getAttributeValue(ReflectionAttributeAccessor.java:25)
    at reflectionuser/com.reflectionuser.ReflectionUtilsUsageExample.main(
        ReflectionUtilsUsageExample.java:18)
```

Die in die JVM integrierte Zugriffskontrolle verhindert bei der Modularisierung, dass man per Reflection auf Elemente (Typen, Methoden, Attribute) zugreifen kann, die nicht `public` sind. Dies sichert den Zugriffsschutz, der ja einer der Vorzüge der Modularisierung ist. Nun ist es aber doch für einige Anwendungsfälle immer einmal wieder wünschenswert, Reflection zuzulassen. Das gilt für einige Tools und Bibliotheken, etwa für Spring (für Dependency Injection) oder für JPA-Provider wie Hibernate (zur Persistierung).

Es gibt verschiedene, in der folgenden Aufzählung genannte Konzepte, um Reflection in gewissem Rahmen auch bei der Modularisierung zu ermöglichen. Eine Idee was notwendig ist, liefert die Fehlermeldung »`opens com.reflectionuser`«. Wir müssen nämlich unser Modul für Reflection »öffnen«) – tatsächlich unterscheidet man nun zwischen Shallow Reflection und Deep Reflection (vgl. nachfolgenden Praxishinweis). Dazu gibt es folgende Varianten:

- **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 Ausführung einzelne Packages explizit für Reflection geöffnet werden.

Hinweis: Shallow Reflection und Deep Reflection

Bezüglich Zugriffsschutz und Reflection gab es einiges an Klärungsbedarf und unterschiedlichste Lösungsansätze. Schließlich wurde der starken Kapselung viel Wert beigemessen, was aber gerade auch für Build-Tools, Frameworks usw. durchaus das eine oder andere Problem verursacht hat. Reflection wird nun in die zwei Typen Shallow und Deep Reflection untergliedert:

- Reflection oder **Shallow Reflection** – Shallow Reflection erlaubt Zugriffe per Reflection, wenn dies auch direkt statisch möglich ist.
- **Deep Reflection** – Deep Reflection verkörpert Zugriffe, wie sie früher nur mit `setAccessible(true)` auf Typen stattfinden konnten, etwa weil die Sichtbarkeit geringer als `public` ist. Ohne Deep Reflection führt ein Aufruf von `setAccessible(true)` – wie eingangs im Beispiel gesehen – zu einer `java.lang.reflect.InaccessibleObjectException`.

Um mögliche Probleme zu adressieren, wurde mit dem Öffnen von Packages und Modulen eine Variante eingeführt, die sowohl den Export zur Kompilierzeit ermöglicht als auch zur Laufzeit weiter gehende Zugriffe mit Reflection. Interessanterweise erlaubt das Öffnen keine Zugriffe zur Kompilierzeit, sondern wirklich nur zur Laufzeit.

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;
}
```

Durch diesen Zusatz im Moduldeskriptor werden die Sichtbarkeitskontrolle und der Zugriffsschutz zur Laufzeit ausgehebelt und für Reflection-basierte Frameworks bereitgestellt. Somit ist es zur Laufzeit möglich, sich Zugriff selbst auf private Elemente zu verschaffen. Zur Kompilierzeit sind allerdings weiterhin nur die explizit per `exports` aufgelisteten Packages nach außen zugreifbar.

Nach dieser Korrektur im Moduldeskriptor müssen wir mit

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

neu kompilieren und können dann die Applikation mit

```
java -p build -m reflectionuser/com.reflectionuser.ReflectionUtilsUsageExample
```

starten, wodurch dann der Attributwert ie folgt ausgegeben wird:

```
Value = Micha
```

Das Schlüsselwort `opens` Die gerade gezeigte explizite Freigabe eines gesamten Moduls erlaubt mitunter viel zu wenig Kontrolle. Des Öfteren möchte man wohl eher einzelne Packages eines Moduls später zur Laufzeit per Reflection zugreifbar machen. Dazu dient das Schlüsselwort `opens`. Dadurch wird der Zugriff mit Reflection und `setAccessible(true)` selbst auf private Bestandteile möglich.

```
module reflectionuser
{
    requires reflectionutils;

    opens com.domain; // Zugriff nur zur Laufzeit
}
```

Wie beim Qualified Export gibt es auch beim Öffnen eines Modules bzw. dessen Bestandteil die Möglichkeit, spezielle Module zu spezifizieren, die Zugriff erhalten sollen:

```
module reflectionuser
{
    requires reflectionutils;

    opens com.domain to reflectionutils; // Zugriff nur zur Laufzeit
}
```

Kommandozeilenparameter `--add-opens` Beim Programmstart ergänzen wir den nachfolgend fett markierten Kommandozeilenparameter:

```
java -p build --add-opens reflectionuser/com.domain=reflectionutils \
     -m reflectionuser/com.reflectionuser.ReflectionUtilsUsageExample
```

Dadurch wird der Zugriff für das Module (`reflectionutils`) ermöglicht.

Einfluss von `opens` und `exports`

Folgende Tabelle 7-1 fasst die bisher genannten Zugriffsmöglichkeiten zusammen und verdeutlicht, dass mithilfe von Shallow Reflection Zugriffe wie beim Kompilieren erlaubt sind und Deep Reflection sogar Zugriffe zulässt, die beim Kompilieren nicht möglich sind.

Tabelle 7-1 *Einfluss von* `opens` *und* `exports`

Schlüsselwörter	Kompilieren	Reflection	
		Shallow	Deep
`exports`	erlaubt	erlaubt	-/-
`opens`	-/-	erlaubt	erlaubt
`exports` und `opens`	erlaubt	erlaubt	erlaubt