

# Übungen zur Modularisierung

## Aufgabe 1

Gegeben sei folgende einfache Applikation, in der die Hauptfunktionalität in der `main()`-Methode und die eigentliche Berechnung in einer Utility-Methode momentan noch monolithisch innerhalb einer Klasse realisiert sind.

Diese soll nun schrittweise mit dem Modularisierungsansatz aus JDK 9 anhand der folgenden Unteraufgaben in eine modularisierte Applikation umgewandelt werden.

```
package com.timeexample;

import java.time.LocalDateTime;

public class CurrentTimeExample
{
    public static void main(final String[] args)
    {
        System.out.println("Now: " + getCurrentTime());
    }

    public static LocalDateTime getCurrentTime()
    {
        return LocalDateTime.now();
    }
}
```

**Aufgabe 1a:** Spalten Sie Funktionalität der obigen Klasse in die zwei Klassen `TimeInfoApplication` und `TimeUtils`. Nutzen Sie als sinnvolle Strukturierung die Packages `app` und `services`.

```
└─ src
   └─ com
      └─ timeexample
         ├── app
         │   └─ TimeInfoApplication.java
         └── services
             └─ TimeUtils.java
```

### Kompilieren

```
javac src/com/timeexample/app/TimeInfoApplication.java \
      src/com/timeexample/services/TimeUtils.java -d build
```

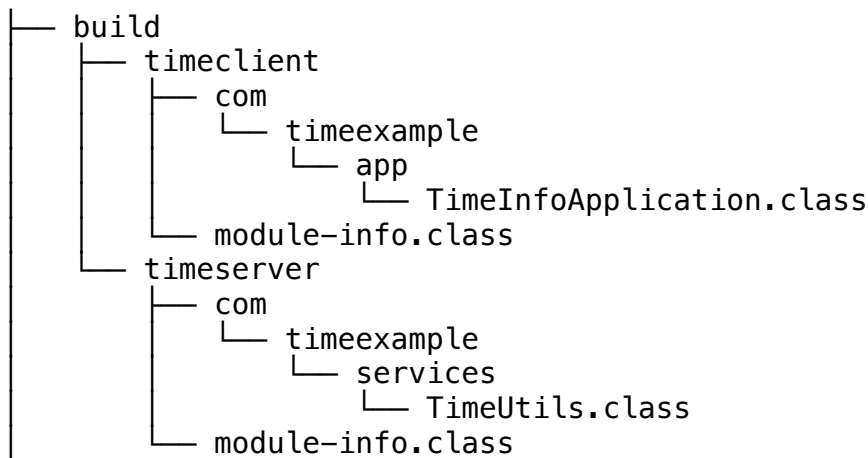
### Starten

```
java -cp build com.timeexample.app.TimeInfoApplication
```

### Ausgabe

```
Now: 2016-11-26T12:29:41.251956
```

**Aufgabe 1b:** Überführen Sie die Verzeichnisse in zwei Module `timeclient` und `timeserver`. Nutzen Sie dazu jeweils eine `module-info.java`-Datei zur Moduldefinition, in der die Abhängigkeiten korrekt spezifiziert sind. Kompilieren Sie die modularisierte Applikation in das Ausgabeverzeichnis `build`. Das Verzeichnis sieht danach in etwa wie folgt aus:



### Tipp

Starten Sie mit dem unabhängigen Kompilieren jedes der beiden Module:

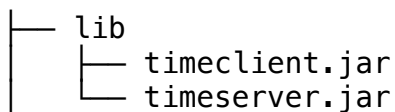
```
javac -d build/timeserver \
    src/timeserver/module-info.java \
    src/timeserver/com/timeexample/services/TimeUtils.java
```

Beim Kompilieren könnte es zu Fehlern kommen:

- Denken Sie daran, den Module-Path anzugeben.
- Prüfen Sie die Moduldeskriptoren auf Abhängigkeiten und Freigaben.

### Aufgabe 1c:

Überführen Sie die Applikation in zwei modulare JARs und legen dazu ein Verzeichnis `lib` an.



Das Programm sollte sich nun wie folgt starten lassen:

```
java -p lib -m timeclient/com.timeexample.app.TimeInfoApplication
```

Was muss man tun, um den Start auch folgendermaßen zu ermöglichen?

```
java -p lib -m timeclient
```

### Tipp

Nutzen Sie das `jar`-Tool:

```
jar --create --file ...
```

## Aufgabe 2

Basierend auf den Ergebnissen der Modularisierung aus Aufgabe 1 soll nun ein Abhängigkeitsgraph erzeugt und visualisiert werden.

### Aufgabe 2a:

Bei der Ermittlung der Abhängigkeiten hilft das Tool `jdeps`. Damit sollten Sie in etwa folgende Ausgaben erzeugen:

```
[file:///Users/michaeli/Desktop/jdk9workshop/exercises/lib/
timeclient.jar]
  requires mandated java.base
  requires timeserver
timeclient -> java.base
timeclient -> timeserver
  com.timeexample.app      -> com.timeexample.services timeserver
  com.timeexample.app      -> java.io                      java.base
  com.timeexample.app      -> java.lang                    java.base
  com.timeexample.app      -> java.lang.invoke            java.base
  com.timeexample.app      -> java.time                      java.base
timeserver
[file:///Users/michaeli/Desktop/jdk9workshop/exercises/lib/
timeserver.jar]
  requires mandated java.base
timeserver -> java.base
  com.timeexample.services -> java.lang                      java.base
  com.timeexample.services -> java.time                      java.base
```

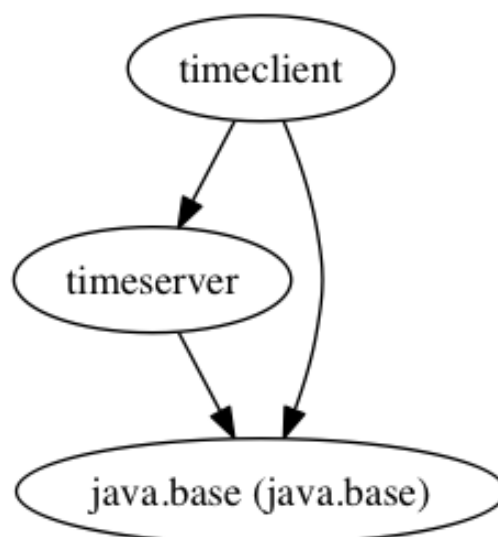
### Aufgabe 2b:

Zur grafischen Aufbereitung dieser doch etwas unübersichtlichen Informationen kann man `jdeps` in Kombination mit dem Tool `graphviz` (<http://www.graphviz.org/>) nutzen.

#### Tipp

Bei der Umwandlung eines DOT-Graphen in ein PNG ist folgendes Kommando nützlich — hier am Beispiel der Datei namens `summary.png`:

```
dot -Tpng graphs/summary.dot > summary.png
```



## Aufgabe 3

Erweitern Sie die Klasse `TimeUtils`, sodass Methodenaufrufe protokolliert werden. Nutzen Sie dazu einen `Logger` aus `java.util.logging`. Was fällt beim Kompilieren auf? Wie lässt sich der gemeldete Fehler korrigieren?

Der Start mit `java -p lib -m timeclient` sollte in etwa folgende Ausgaben produzieren:

```
Nov. 26, 2016 2:09:57 NACHM. com.timeexample.services.TimeUtils
getCurrentTime
INFORMATION: getCurrentTime() called
Now: 2016-11-26T14:09:57.281634
```

### Tipp 1

Das Logging kann wie folgt implementiert werden:

```
Logger.getLogger().log(Level.INFO, "getCurrentTime() called");
```

Dazu werden folgende Imports benötigt:

```
import java.util.logging.Level;
import java.util.logging.Logger;
```

### Tipp 2

Mit `java --list-modules` lassen sich alle Module des JDKs auflisten. Wählen Sie das passende für das Logging.

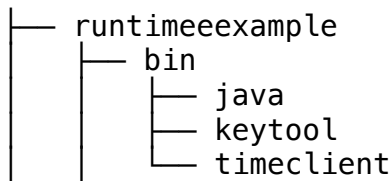
### Tipp 3

Das Kompilieren wird einfacher, wenn man den Multi Module Build nutzt:

```
MAC: javac -d build --module-source-path src $(find src -name '*.java')
PC:  javac -d build --module-source-path src $(dir src -r -i '*.java')
```

## Aufgabe 4

Das zuvor modularisierte Programmsystem soll nun als eigenständiges lauffähiges Executable in einem Verzeichnis `runtimeexample` bereitgestellt werden.



Starten Sie dann das Programm mit `runtimeexample/bin/timeclient`, um die Funktionsweise anhand folgender Ausgaben zu prüfen.

```
Nov 26, 2016 2:27:16 PM com.timeexample.services.TimeUtils  
getCurrentTime  
INFORMATION: getCurrentTime() called  
Now: 2016-11-26T14:27:16.375422
```

### Tipp

Denken Sie daran, dass dazu das JAR als Executable-JAR mit einer Main-Class versehen werden muss und nutzen Sie das Kommando `jlink` sowie die Option `--add-modules timeclient`. Geben Sie auch das JDK im Module-Path an.