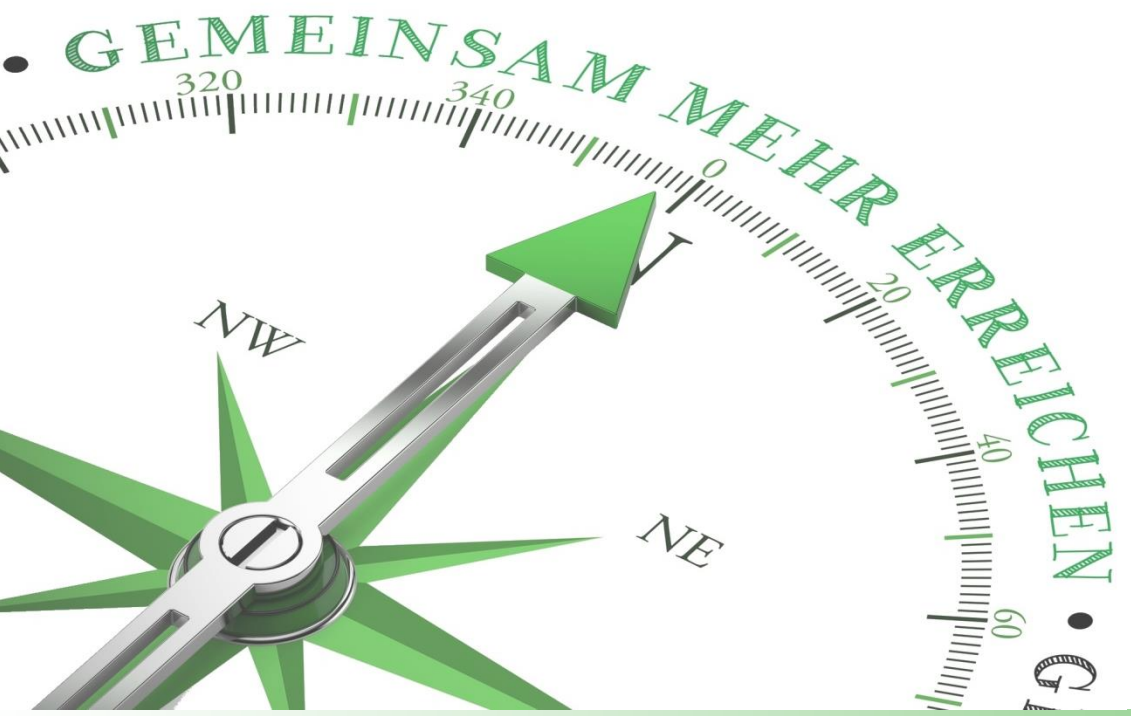
avision
REVIVING YOUR SOFTWARE

Have fun with Java 9



Feature - Übersicht

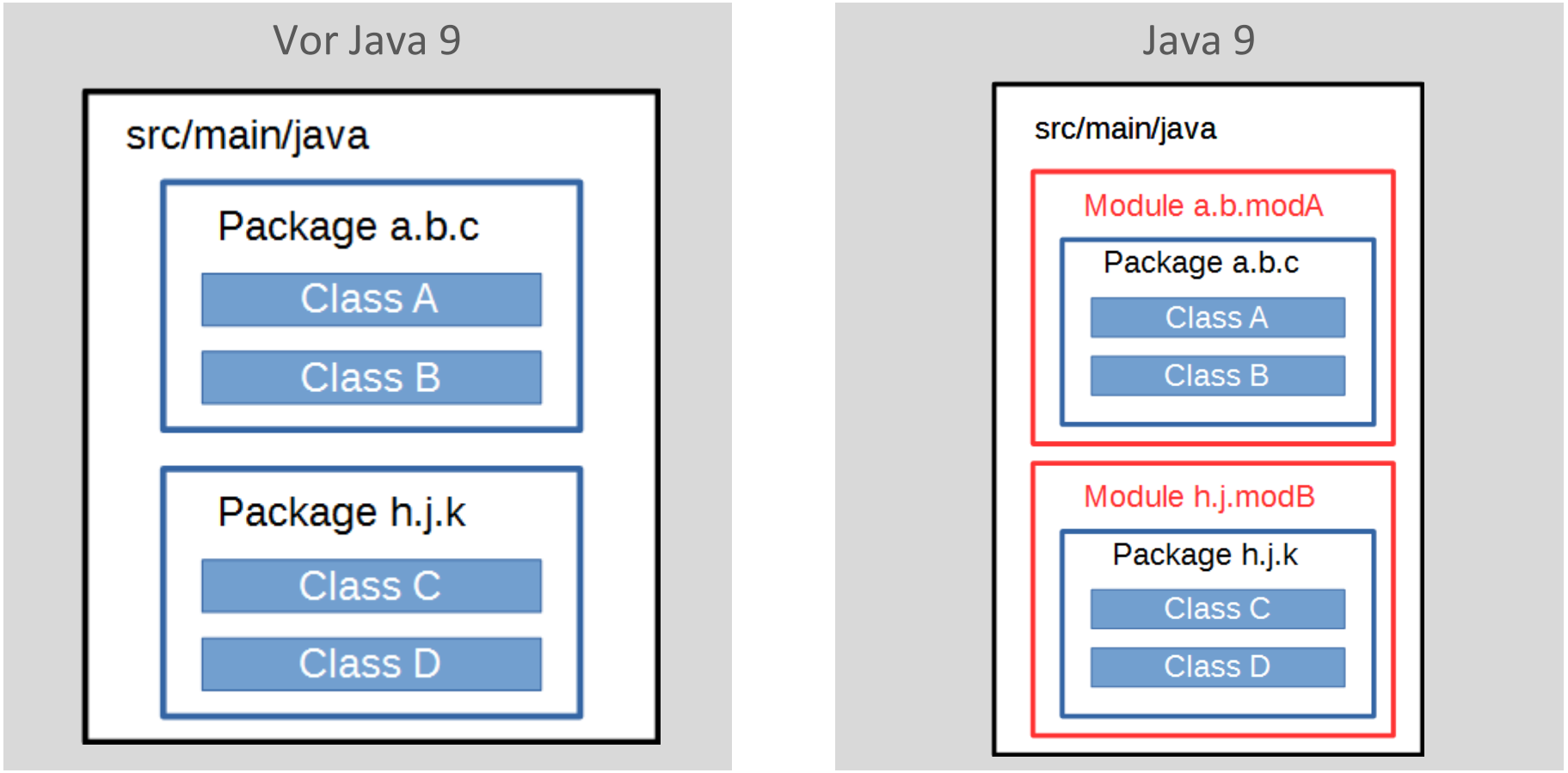
- Einführung eines neuen Modulsystems (Projekt „Jigsaw“)
- Erstellung von benutzerspezifischen JRE‘s
- JShell Command Line Tool
- Neuer HTTP Client
- Verbesserter Zugriff auf Systemprozesse
- Interfaces für Reactive Streams (Publish-Suscribe)
- Vereinheitlichtes JVM-Logging
- HTML5 Javadoc
- Sprachverbesserungen

Modularisierung - Ziele

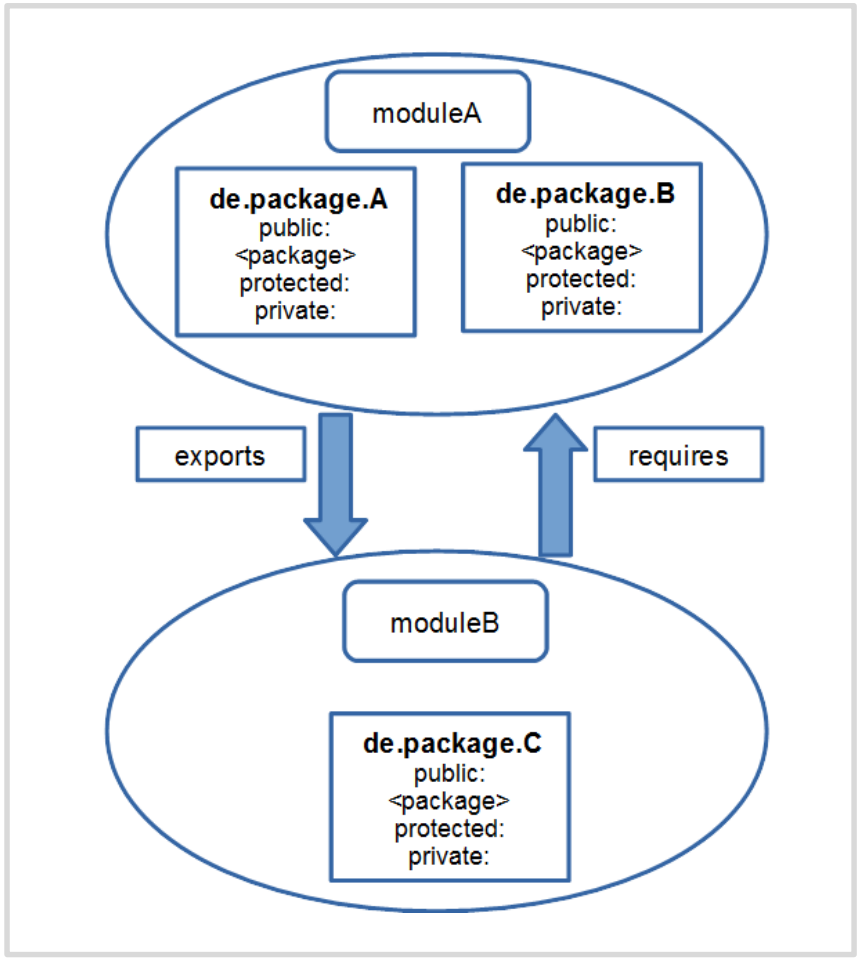
- **Reliable Configuration**: Ablösung des `CLASSPATH`
  - Problem: das Konzept ist fehleranfällig, wenn Klassen mehrfach enthalten sind → Reihenfolge?
  - Neu: explizite Definition von Abhängigkeiten
- **Strong encapsulation**: Komponenten können öffentliche API definieren und Implementierung verbergen
- **Scalable Java SE Platform**: Modularisierung der Java-Plattform (Anwendungsbeispiel: IoT Anwendungen)

Modul-Konzept

Modul := Menge von Java-Packages & Resourcen



Sichtbarkeits-Prinzip (neu)



Zugriffscheck

Bsp: `moduleA` liest `moduleB`

I. Read-Beziehung in `moduleB` `[=true]`

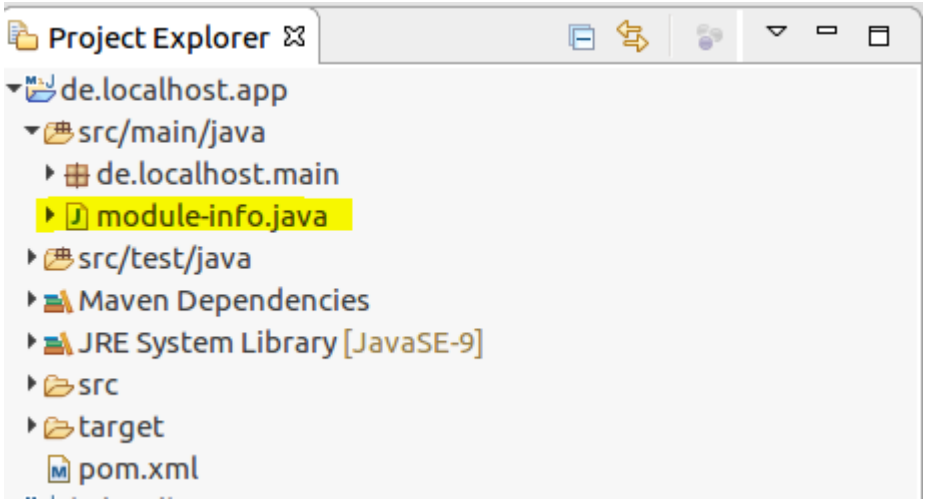
II. Export-Beziehung in `moduleA` `[=true]`

III. Sichtbarkeitsprüfung (`public`, …) `[=true]`

Konsequenz: alte Sichtbarkeitsregeln reichen nicht mehr aus!

Modul-Kennzeichen

- Wird in ein „Modular JAR“ kompiliert → `MODULEPATH`
  - Eindeutigkeit: 1 Modulares JAR := 1 Module
  - Split von Packages auf mehrere Module ist nicht möglich
- Modul-Name muss eindeutig sein (nicht erlaubt: „-“)
- Steuerung durch Config-File: module-info.java



Module-info.java

avision
REVIVING YOUR SOFTWARE

```
module de.localhost.application {

    /** Grundlegende Beziehungen **/
    requires de.localhost.utils;          /* benoetigt dieses Module */
    exports de.localhost.main;            /* exportiert dieses Package */

    /** Spezielle Anweisungen**/
    /* uses de.localhost.provider;           verwendet diesen Service Provider  */
    /* provides de.localhost.provider;       exportiert diesen Service Provider */

    /* opens de.localhost.test;              macht test-Package zur Laufzeit
                                             verfuegbar (deep reflection) */

}
```

Module-info.java - Eigenschaften

avision
REVIVING YOUR SOFTWARE

- Keine Wildcard-Unterstützung (daher nicht zulässig (z.B.): `de.localhost.*;`)
- Keine Hierarchie-Unterstützung (Unterpakete müssen explizit angegeben werden)
  - Beispiel:

```
requires de.localhost.foo;
requires de.localhost.foo.A;
```

- Sonderfälle
  - Export kann eingeschränkt werden:

```
export de.localhost.foo to de.locahost.bar;
```

  - Lösung für implizite Abhängigkeiten:

```
requires transitive de.localhost.xxx;
```

- Module-Informationen können auch zur Compile/Laufzeit festgelegt werden
  - Export (Kommandozeile): `javac/java –add-exports de.localhost.bar`
  - Import (Kommandozeile): `javac/java –add-modules de.localhost.bar`

Arten von Modulen

avision
REVIVING YOUR SOFTWARE

Explicit module

Vollwertiges Modul (Standard)

- mit modules-info.java
- liegt im `MODULEPATH`
- Spezialfall: „open module“

Automatic module

Modul mit automatisch generierter modules-info.java

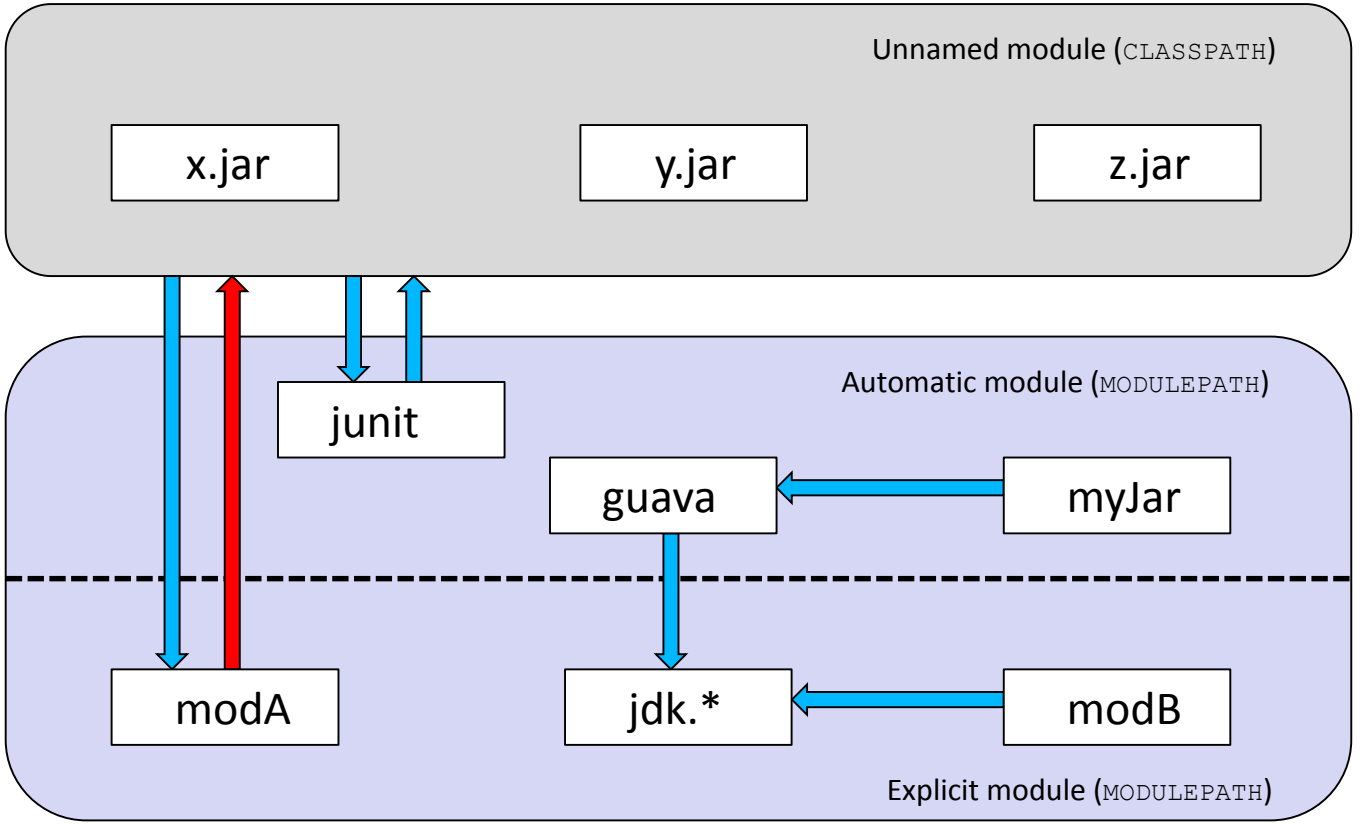
- Vorgehen: jar-File in `MODULEPATH` legen
- Versionsnummern werden entfernt:
  `junit-4.12.jar` → `junit`
- Reads auf alle explicit/unnamed modules
- Export aller Packages

Unnamed module

Sammel-Modul für alle jar-Files im `CLASSPATH`

- Reads auf alle Module des `MODULEPATH`
- Export aller Packages
  (sichtbar nur für automatic module)

Lesezugriff von Modulen



Kommandozeilen Tools

avision
REVIVING YOUR SOFTWARE

Kompilieren

```
javac --module-source-path <Liste von Java-Dateien>
      --module-path <Liste von Verzeichnissen>
      -d <Output-VZ>
      <Pfad modules-info.java>
      <Pfad zu anderen Java-Dateien>
```

Aufruf

```
java --module-path <Liste von VZ mit Modulen>
     --module <Name des Moduls>/<Name der Hauptklasse>
```

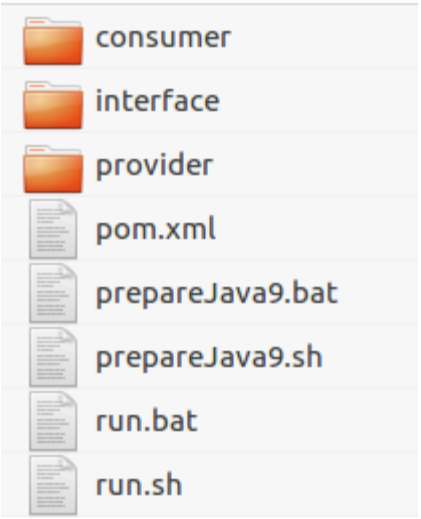
Zusätzlich

- `jdeps`: zeigt Dependencies von jar-File an
- `jdeps` && `dot` (graphviz): Visualisierung der Abhängigkeiten
- `javap`: liest Infos aus module-info.class
- `jdeprscan`: sucht nach depricated APIs

Java-Bausystem (Modulariserung)

avision
REVIVING YOUR SOFTWARE

Root-Folder Struktur



- Reminder: JDK ist nicht für das Bausystem verantwortlich!
- Konsequenz (Modularisierung)
  - hohe Anzahl an Kombinationsmöglichkeiten
  - Betriebssystemabhängige Bauskripte
- Projektübergreifende Standards sind **zwingend** notwendig!
  - Ordnerstruktur festlegen (Build-Skripte, Module-Directories)
  - Anforderungen definieren (Windows/Linux)
  - Modul-Einbindung festlegen (z.B. per Kommando-Zeile,…)

Maven (Basis-Config)

- Voraussetzung: Maven 3.5.0, Maven Compiler Plugin: 3.7.0
- Möglichkeiten der pom.xml-Konfiguration
  - Standard JDK ist Version 9:

```
<artifactId>maven-compiler-plugin</artifactId> <version>3.7.0</version>
<configuration> <source>9</source><target>9</target> </configuration>
```

  - Explizite Angabe der JDK:

```
<toolchain>
    <type>jdk</type>
    <provides>
      <version>1.9</version>
      <vendor>oracle</vendor>
    </provides>
    <configuration>
      <jdkHome>/path/to/your/jdk-9</jdkHome>
    </configuration>
</toolchain>
```

Maven (Useful pom.xml-Configs)

avision
REVIVING YOUR SOFTWARE

- Hauptprojekt als Maven Modul

```
<modules>
  <module>application</module>
  <module>utils</module>
 </modules>
```

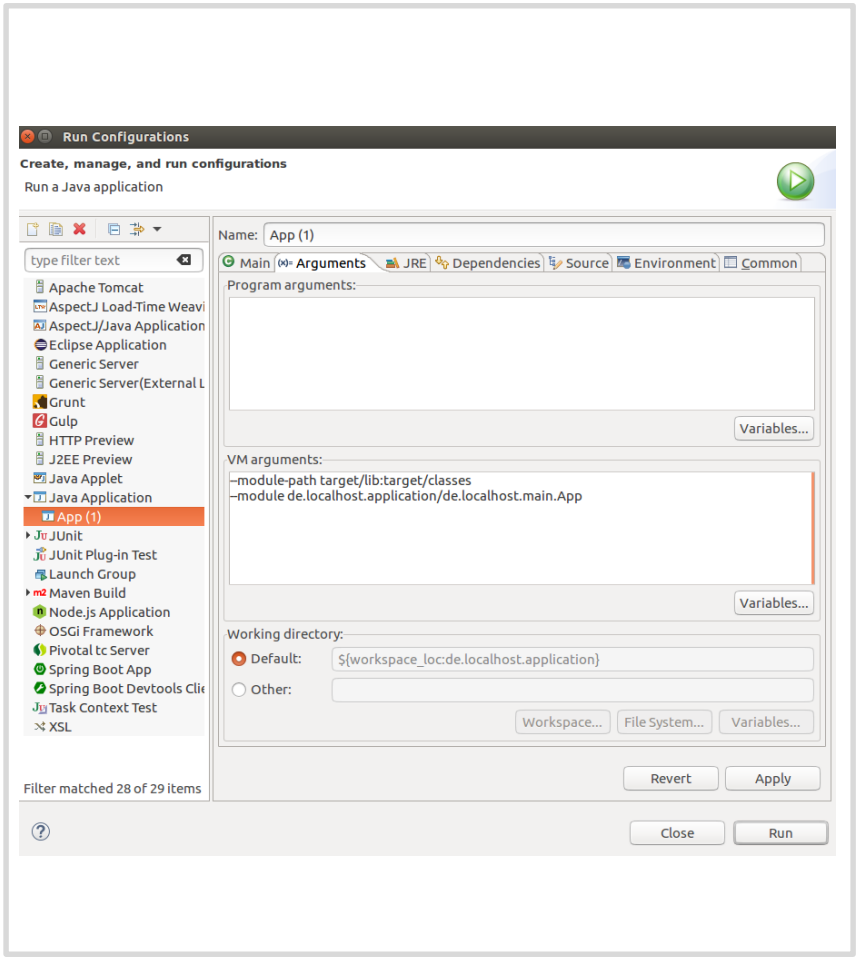
- Modul-Dependencies von Maven-Modulen spezifizieren
- Kopieren der Module in lib-Verzeichnis

```
<artifactId>maven-dependency-plugin</artifactId>
 <executions> <execution>
       <id>copy-dependencies</id>
       <phase>package</phase>
       <goals>
            <goal>copy-dependencies</goal>
       </goals>
       <configuration>
            <outputDirectory>${project.build.directory}/lib</outputDirectory>
       </configuration>
 </execution></executions
```

- Maven Build: `mvn clean package`

IDE

- IntelliJ IDEA: bessere Java 9 Unterstützung
- Eclipse erfordert derzeit viele manuelle Nachjustierungen
  - Oracle-JDK scheint besser zu funktionieren als OpenJDK
  - Wichtige Menüs:
    - Preferences: *Installed JRE*
    - Properties: *Java Build Path* und *Java Compiler*
  - Run Configuration



Migration - Fehlerquellen

- Zugriffsprobleme auf java.se.ee-Pakete:

  `xml.ws`, `xml.bind`, `transaction`, `corba`, `activation`, `annotations.common`

  Lösung: Manuelles Hinzufügen zur Kompilierung/Laufzeit: `--add-modules`

- Kein Zugriff mehr auf interne JDK APIs
  - Beispiel: `sun.*` - Packages
  - Betrifft auch implizite Abhängigkeiten (gibt dazu Beispiele bei Spring Boot)

- Kein Zugriff mehr auf rt.jar, tools.jar, dt.jar
  - Führt z.B. zu geänderten Rückgabewerten bei ClassLoader::getSystemResource()

- Underscore ('_') wird reserved keyword (illegal als identifier)

- Java-Versions-String wird planar: „9“ (und nicht „1.9“)

  Neue Schema:`$MAJOR.$MINOR.$SECURITY.$PATCH`

- Nur Spring Boot 2 unterstützt Java 9

Migration - Modularisierung I

avision
REVIVING YOUR SOFTWARE

Strategie

1. Alt-Anwendung mit Java 9 ausführen: kann zu Fehlermeldungen kommen

2. Weitere Vorgehensweise festlegen
   a. „Top-Down“:
      I. Alle jar-Dateien in `MODULEPATH` -> Generierung von automatischen Modulen
      II. Schrittweise Anpassung der module-info.java
      III. Kompilieren der Gesamtanwendung

   b. „Bottom-Up“
      I. Ein jar-File auswählen & Abhängigkeiten mit `jdeps` bestimmen
         `(jdeps --jdkinternals …)`
         -> Günstig: jar-Files, die keine Abhängigkeiten zu `CLASSPATH` haben
      II. Modul-Deskriptor generieren: `jdeps –generate-module-info …`
      III. Überprüfung der generierten module-info.java
      IV. Kompilieren der Gesamtanwendung (`--add-modules`)

Migration - Modularisierung II

- `java` Kill-Switch: `--permit-illegal-access` (only Java 9)
- Wenn Package über mehrere jar-Files verteilt → kann nur ein Teil modularisiert werden
- Third-Party jar-Files: eigenständige Modul-Migration nicht sinnvoll
- (Lösung: automatic module)
- Unit-Test müssen auch migriert werden: komplex!
  - Sichtbarkeit des zu testenden Codes spielt eine Rolle: Modul-Intern/Exportiert
  - Unterschiedliche Szenarien: z.B. Testcode im (produktiven) Modul, separates Test-Modul
  - Es existieren Mixed-Szenarien: Modul + JUnit-Tests im `CLASSPATH`

- Es gibt die Möglichkeit Multi-Release jar-Files zu generieren

  (mit Versionen der Klassen für JDK8, JDK9, …)

Sprachverbesserungen (Beispiele)

avision
REVIVING YOUR SOFTWARE

- „One-Liner“ für Immutable Collections

```
Map<String, String> immutableMap = Map.of("key1", "Value1", "key2", "Value2");
```

- Systemprozess-API Verbesserungen

  z.B. Betriebssystem-unabhängig:

```
System.out.println("Your pid is " + ProcessHandle.current().pid());
```

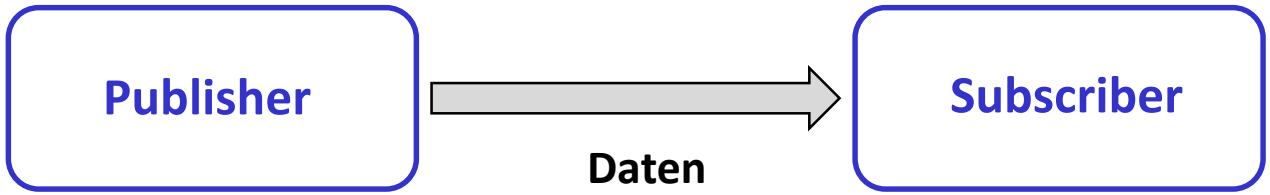
- Streams: `takeWhile()/dropWhile()`
  - `myProgLangs.stream().takeWhile(s -> !s.contains("Java")).forEach(System.out::println);`
  - `myProgLangs.stream(). dropWhile(s -> !s.contains("Java")).forEach(System.out::println);`
- `Try-With` mit Inline-Resources: `try (new BufferedWriter())`
- `stream`-Method für Optionals

```
// In Java 8:

Stream.of("a", "b", "c")
.map(UserAccount::lookup)
.filter(Optional::isPresent)
.map(Optional::get)
.collect(toList());
```

```
// In Java 9:

Stream.of("a", "b", "c")
.map(UserAccount::lookup)
.flatMap(Optional::stream)
.collect(toList());
```

Reaktive Programmierung

Prinzip: Event-basierte Verarbeitung eines asynchronen Datenstroms



- Beispiele
  - `z:=x+y` → `z` ändert sich automatisch, wenn sich `x` oder `y` ändern
  - Excel: ändern von Zeilenwert → Summenwert-Feld ändert sich automatisch
- Grundlage ist das „Reaktive Manifest“:
  - event-driven (nachrichtenorientiert)
  - responsive (antwortbereit)
  - resilient (fehlertolerant)
  - elastic (skalierbar)
- Spring 5: WebFlux framework

java.util.concurrent.Flow

avision
REVIVING YOUR SOFTWARE

```
@FunctionalInterface
public static interface Flow.Publisher<T> {
    public void    subscribe(Flow.Subscriber<? super T> subscriber);
}

public static interface Flow.Subscriber<T> {
    public void    onSubscribe(Flow.Subscription subscription);
    public void    onNext(T item) ;
    public void    onError(Throwable throwable) ;
    public void    onComplete() ;
}

public static interface Flow.Subscription {
    public void    request(long n); //request n items for consumption (load control)
    public void    cancel() ; //end subscription
}

public static interface Flow.Processor<T,R>  extends Flow.Subscriber<T>, Flow.Publisher<R> {
}
```

Fazit

avision
REVIVING YOUR SOFTWARE

Java 9: große strukturelle Veränderungen in der JDK

- Bei Umstieg können Probleme auftreten
- Modularisierung
  - Bestands-Software: Umstieg entspricht echter Migration
    - Nicht jedes Projekt kann umgestellt werden
    - Refactoring oftmals zwingend notwendig
    - Meistens mit Zwischen-Stufen verbunden: Hybride aus modularisiertem und altem Code
    - Unit-Tests: Anpassung erforderlich (analog zum Code)
    - Projekt-Übergreifende Standardisierung des Bau-Systems ist zwingend notwendig
- JRE-Spezifikation wird wichtig: „Standard“ oder „Customized“

Referenzen

- https://www.informatik-aktuell.de/entwicklung/programmiersprachen/java-9-das-neue-modulsystem-jigsaw-tutorial.html
- https://entwicklertag.de/frankfurt/2017/ein-geduldsspiel-das-1000-module-puzzle-f%C3%BCr-entwickler-das-neue-modulsystem-jigsaw-java-9
- https://www.journaldev.com/13121/java-9-features-with-examples
- https://jaxenter.de/java-9-migration-59148
- https://community.oracle.com/docs/DOC-1006738