Programmieren in Java

Vorlesung 12: Metawissen
Java Bibliotheken, Maven

Robert Jakob

Albert-Ludwigs-Universität Freiburg, Germany

SS 2013

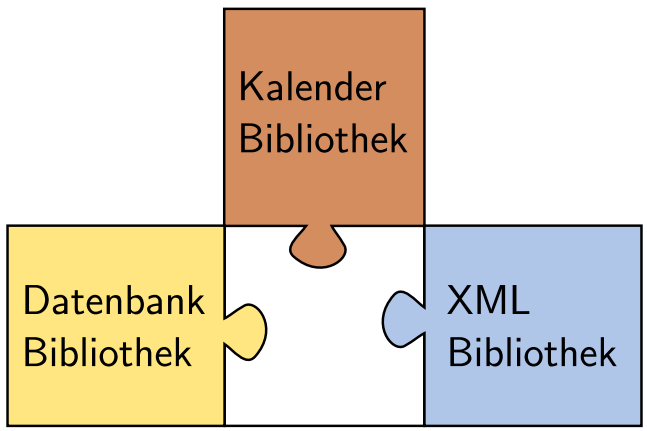
Inhalt

Java Bibliotheken

Maven

Bibliotheken

- Sammlung von Paketen und deren Klassen
- Lösungen für thematisch zusammengehörendes Problem
- Selbst nicht ausführbar



Beispiel

Bibliothek:

```
package simpleMath;

public class Arith {
   public static int add(int x, int y) { ... }
   public static int sub(int x, int y) { ... }
}
```

Programm:

```
package calculator;

// Uses package from library
import simpleMath.Arith;

public class Calculator {
public static void main(String[] args) { ... }
```

Bibliotheken in Java

- Java Archive: JAR
- Zip-Dateien mit Metainformationen

Inhalt eines JAR:

- Ordner(packages) mit Klassen (`.class`) und/oder Quellen (`.java`)
- Ordner `META_INF` mit Metainformationen
- Resourcen (Bilder, . . .)

Ausschnitt einer Jar-Datei:

```
| – META–INF
        \ MANIFEST.MF
\ – simpleMath
        \ Arith.class
```

Das Jar-Tool

Kommandozeilentool `jar`

- Anzeigen der Inhalte eines Jar:
  ```
  $ jar tf simpleMath.jar
  ```
- Entpacken eines Jar:
  ```
  $ jar xf simpleMath.jar
  ```
- Erzeugen eines Jar
  ```
  $ jar cf simpleMath.jar simpleMath
  ```

Erzeugen in Eclipse: Projektexport > Java > Jar file

Aufruf eines Java Programms

Einfacher Aufruf eines Java-Program (Eclipse > Run):

```
$ java calculator.Calculator
```

- Main-Methode von `calculator.Calculator` soll aufgerufen werden
- Classfile wird erwartet unter

```
\ – calculator
        \ Calculator.class
```

Aufruf mit Bibliotheken

Mit Verwendung von Jars:

- Angabe wo Klassen bzw. Jars zu suchen sind: Classpath
- Angaben: *CLASSPATH* oder Argument zum java Aufruf

```
$ java -cp arith.jar:. calculator.Calculator
```

Classpath:

- Pakete und Klassen in `arith.jar`
- Pakete und Klassen im Verzeichnis `.`

Aufruf innerhalb von Eclipse:
Show View, Debug, Debug, Properties auf Prozess

Einbinden einer Bibliothek in ein Eclipseprojekt

- Copy and Paste (JAR liegt im Repository)
- Add External Jar (Absoluter Pfad im .classpath)
- User Library (Namensreferenz im Workspaces)

Metadaten

Meta-Daten in `MANIFEST.MF` (Ausschnitt):

- Ausführbare Klasse/Ausführbares Jar
- Versionsnummer
- Versiegelte Klassen
- Sicherheit

Ausführbares JAR

`META-INF/MANIFEST.MF`:

```
Manifest-Version: 1.0

Main-Class: calculator.Calculator
```

Datei muss mit Newline enden!

Ausführbares JAR

`META-INF/MANIFEST.MF`:

```
Manifest-Version: 1.0

Main-Class: calculator.Calculator
```

Datei muss mit Newline enden!

- Setzen der ausführbaren Klassen beim Erzeugen:
  `$ jar cfe calculator.jar calculator.Calculator calculator`
- Ausführen:
  `$ java -jar calculator.jar`
  Angabe von Classpath nicht möglich!

Eclipse: Export Jar File, Angabe der Main-Class.

Versionsinformationen

```
Name: calculator
Specification-Title: My Calculator
Specification-Version: 1.2
Specification-Vendor: My Company, Inc
Implementation-Title: My Calculator
Implementation-Version: build57
Implementation-Vendor: Example Tech, Inc.
```

Sealing Packages in JARs

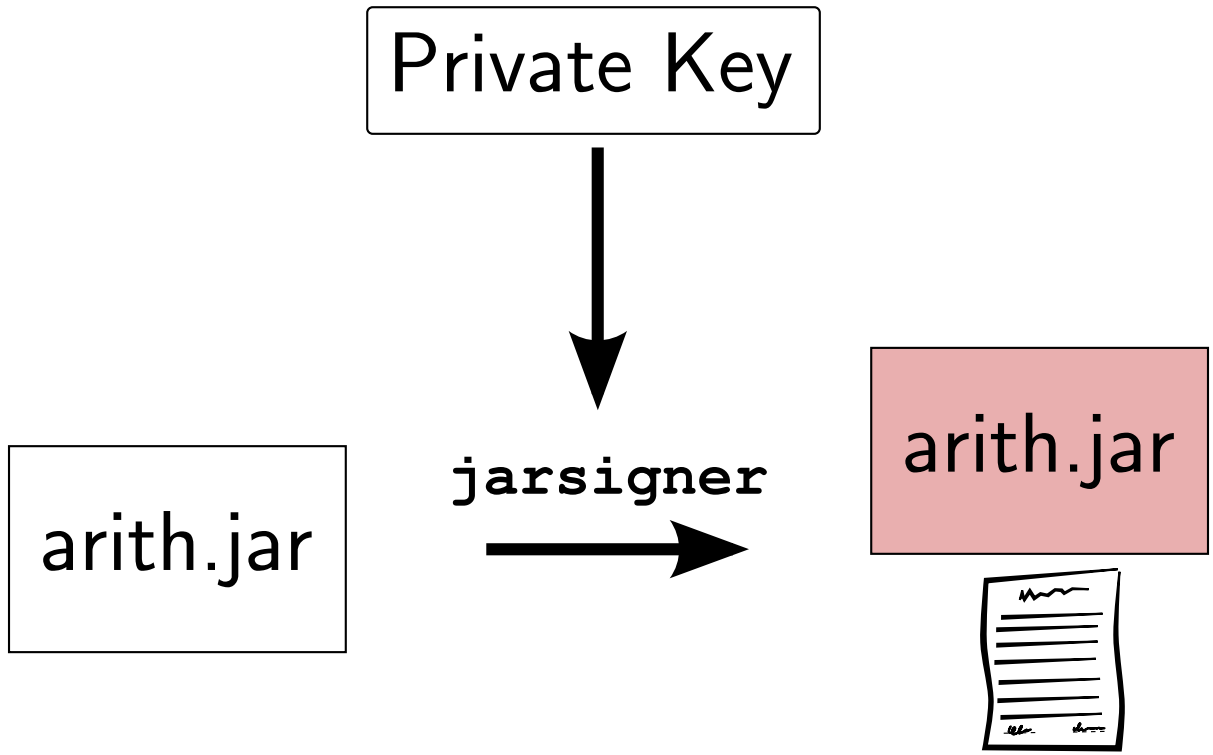
- Idee: Sicherstellung, dass alle Inhalte eines Paketes aus einem Jar kommen.
- Vermeidung von Konflikten beim verwenden mehrere Jars.

```
Name: simpleMath/
Sealed: True
```

Hinweis: Package muss mit / enden!

Signieren von JARs

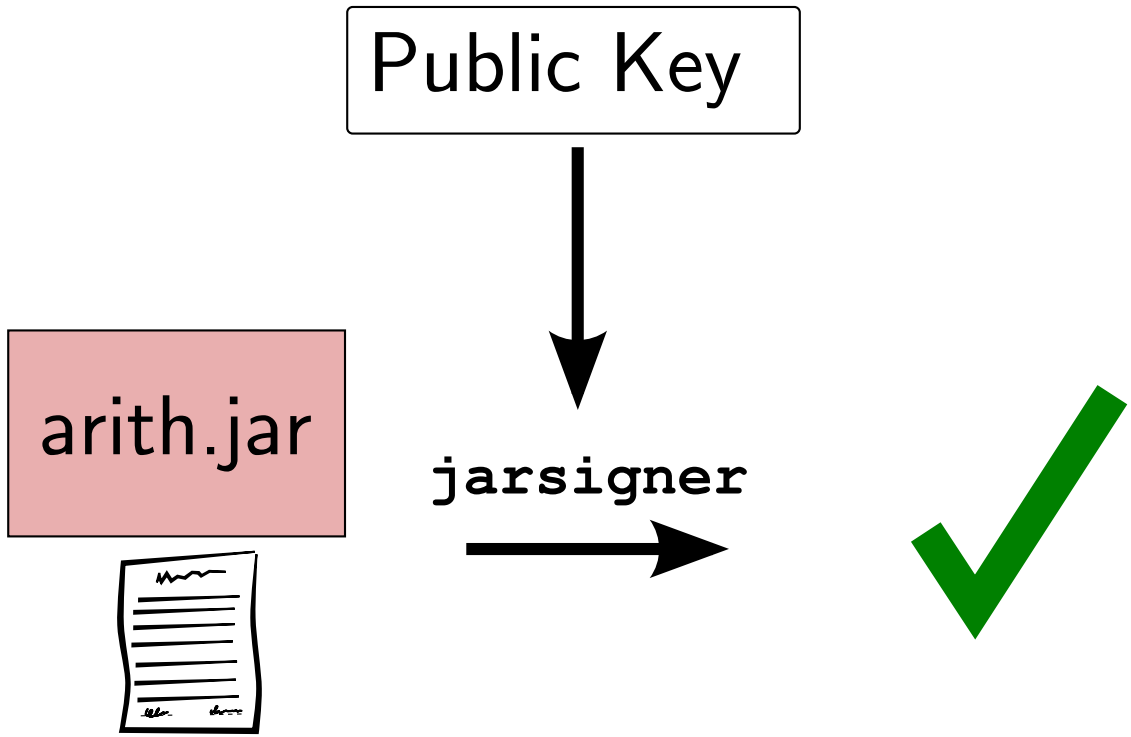
Basiert auf Public-Private-Key Kryptographie[1]



[1] Details siehe Vorlesung Internetsecurity.

Verifizieren

Nutzer der Bibliothek:



Probleme mit Jars

- Sichtbarkeit, Schnittstellen, erforderliche Pakete, dynamisches Laden/Entfernen $\rightarrow$ OSGi-Bundles
- Versions- und Namenskonflikte
  - `NoSuchMethodException`
  - `ClassNotFoundException`
- Abhängigkeitsverwaltung $\rightarrow$ Maven

Java Bibliotheken

Maven

Maven - Was ist das?

Anwendungsbereiche:

- Kompilierung
  Quelltext $\rightarrow$ Verteilung
- Projektmanagement
  Quelltext $\rightarrow$ Websites, Berichte, ...

Wofür ist das gut?
Vereinfacht *Verwaltung* von Java-Projekten mit Abhängigkeiten

Convention over Configuration

- Standardeinstellungen sind ausreichend für “normalen” Gebrauch
- In Maven:
  - Verzeichnisse für Quelltext, Tests
  - Verzeichnisse für Ausgabe
  - Namensgebung der Ausgabedateien
  - Vorgefertigter Lebenszyklus (life-cycle)
- Erweiterung durch Plugins
  - Compiler (`maven-compiler-plugin`)
  - Unit Tests (`maven-surefire-plugin`)
  - Code Coverage (`maven-emma-plugin`)
  - Viele mehr

Projektmodel

Project Object Model (POM)
Kennzeichen:

- Abhängigkeiten zu anderen Projekten (Dependency Managment)
- Orte anderer Projekte (Repositories)
- Wiederverwendbarkeit der Build Logik
- Portierbarkeit und Integration (Eclipse, Netbeans,...)
- Suchen nach Projekten und deren Metainformationen (Nexus[2])

[2] https://repository.apache.org/

Projektmodel als XML

- `pom.xml` als Projektbeschreibungsdatei
- Informationen die nicht den Standardeinstellungen entsprechen

```
<project>
    <modelVersion>4.0.0</modelVersion>
    <groupId>de.uni_freiburg.informatik.proglang</groupId>
    <artifactId>myproject</artifactId>
    <version>1.0-SNAPSHOT</version>
</project>
```

Project Object Model

- Super POM[3]
  Basis für alle POM

[3] http://books.sonatype.com/mvnref-book/reference/pom-relationships-sect-pom.html#ex-super-pom

Project Object Model

- Super POM[3]
  Basis für alle POM
- Einfachstes POM

```
<project>
    <modelVersion>4.0.0</modelVersion>
    <groupId>de.uni_freiburg.informatik.proglang</groupId>
    <artifactId>myproject</artifactId>
    <version>1.0-SNAPSHOT</version>
</project>
```

[3] http://books.sonatype.com/mvnref-book/reference/pom-relationships-sect-pom.html#ex-super-pom

Project Object Model

- Super POM[3]
  Basis für alle POM
- Einfachstes POM

```
<project>
    <modelVersion>4.0.0</modelVersion>
    <groupId>de.uni_freiburg.informatik.proglang</groupId>
    <artifactId>myproject</artifactId>
    <version>1.0-SNAPSHOT</version>
</project>
```

- Effektives POM
  `mvn help:effective-pom`

[3] `http://books.sonatype.com/mvnref-book/reference/pom-relationships-sect-pom.html#ex-super-pom`

Standardvorgaben

Quelltext:

- `src/main/java` enthält Java Quelltext
- `src/main/resources` enthält Bilder, etc.

Standardvorgaben

Quelltext:

- `src/main/java` enthält Java Quelltext
- `src/main/resources` enthält Bilder, etc.

Tests:

- `src/test/java` enthält Unit Tests
- `src/test/resources` enthält Unit Tests

Standardvorgaben

Quelltext:

- `src/main/java` enthält Java Quelltext
- `src/main/resources` enthält Bilder, etc.

Tests:

- `src/test/java` enthält Unit Tests
- `src/test/resources` enthält Unit Tests

Ausgabe:

- `target` Ausgabeverzeichnis
- `target/classes` Klassenausgabe
- `target/test-classes` Testklassenausgabe

Java Version

- Standard Java Version ist 1.3
- Setzen der Version in `pom.xml`

```
<properties>
    <maven.compiler.source>1.7</maven.compiler.source>
    <maven.compiler.target>1.7</maven.compiler.target>
</properties>
```

Skelett erzeugen

- Einfaches “Hello World”-Skelett
- Interaktive Abfrage nach weiteren Feldern des POM

```
mvn archetype:generate \
  -DarchetypeGroupId=org.apache.maven.archetypes \
  -DarchetypeArtifactId=maven-archetype-quickstart
```

Lebenszyklus

- `clean`
  Aufräumen von generierten Objekten[4]
- `default`
  Erzeugen von Objekten
- `site`
  Erzeugen von Berichten, Webseiten, ...

[4] Sourcecode, Resourcen, ...

Lebenszyklus `clean`

- `clean:pre-clean`
  Phase, welche von Plugins genutzt werden kann
- `clean:clean`
  Eigentlicher Löschvorgang von `${basedir}/target`
- `clean:post-clean`
  Phase, welche von Plugins genutzt werden kann

```
$ mvn clean
```

Lebenszyklus `default`

- Manchmal auch “build” genannt
- Phasen (Auswahl):
  - validate
  - generate-sources
  - generate-resources
  - compile
  - generate-test-sources
  - test-compile
  - test
  - package
  - integration-test
  - verify
  - install
  - deploy
- Plugins werden in den entsprechenden Phasen aufgerufen (z.B. maven-compiler-plugin)
- Aufruf z.B. `$ mvn test`

Bindung der Phasen an Ziele

Phase	*Plugin-Ziel*
process-resources	resources:resources
compile	compiler:compile
process-test-resources	resources:testResources
test-compile	compiler:testCompile
test	surefire:test
package	jar:jar
install	install:install
deploy	deploy:deploy

Lebenszyklus `site`

Erzeugen eines Reports bzw. einer Webseite

```
$ mvn site
```

- Projektzusammenfassung
- Ansprechpartner
- Abhängigkeitsinformationen
- Javadoc
- Checkstyle

Dependencies

Abhängigkeiten zu anderen Bibliotheken und Projekten

- Dependencies in Maven:
  `groupId:artifactId:jar:version` (`junit:junit:jar:4.11`)
- Scope: `compile`, `provided`, `runtime`, `test`
- Zentrales Repository (`http://search.maven.org`)

Dependencies

Abhängigkeiten zu anderen Bibliotheken und Projekten

- Dependencies in Maven:
  `groupId:artifactId:jar:version` (`junit:junit:jar:4.11`)
- Scope: `compile`, `provided`, `runtime`, `test`
- Zentrales Repository (`http://search.maven.org`)

Dependency in POM:

```
<dependencies>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>4.11</version>
      <scope>test</scope>
    </dependency>
</dependencies>
```

Dependencies (2)

- Bibliotheken können von weiteren Bibliotheken abhängen
- `junit` hängt von `org.hamcrest:hamcrest-core:jar:1.3` ab
- Diese Abhängigkeiten werden automatisch erfüllt
- Konflikte möglich:
  - Bibliothek A in Version 1.0 erforderlich
  - Bibliothek B in Version 2.0 erforderlich
- Lösungsmöglichkeit
  - Angabe von Versionsbereichen: `(3.8.1,4.11]`
  - gezieltes Entfernen von Abhängigkeiten

Weitere Infos hier[5].

[5] `http://books.sonatype.com/mvnref-book/reference/pom-relationships-sect-project-dependencies.html`

Referenzen

Weitere Infos

- Maven Homepage
  `http://maven.apache.org`
- Maven: The Definitive Guide
  `http://books.sonatype.com/mvnref-book/reference/`