Building Java Projects with *Maven*

- What is Maven?
- Installing Maven
- Projects, Artifacts and Dependencies
- Build Lifecycle. Phases, Plugins and Goals
- Parent POMs and Multi-Module Projects

What is Maven?

- Open-source build tool developed by the Apache Group
- **Industry Standard** for building and managing Java-based projects
- Helps to **build, publish, deploy** several projects at once
- Written in Java. Can be used to build projects written in Java, C#, Scala, Ruby, etc.
- Based on **P**roject **O**bject **M**odel (POM) – written in **XML**
- Makes developer’s life easier as Maven is taking care of:
  - builds, documentation, dependencies, reports, releases, etc.
- Alternatives: Gradle
  - Standard for Android builds

Why Maven?

Most Java projects consists of libraries:

ex.: for Spring MVC you need ~10-12 libraries

1. Download them
2. Add those jar/war files to your project

Case 1:

Imagine you are going to upgrade Java version.
Then you need to download all the Spring dependencies again :(

Case 2:

Your project has 3 Java classes + tests + libraries.
To load your project on another computer you need to copy/paste all the classes and the libraries :(

Here Maven comes to the rescue!

Maven - Features

- A huge repository of user libraries
- Set up projects easily – helps to avoid as mush configuration as possible via project templates
- Backwards compatibility with previous versions (*transitive dependencies*)
- Isolation between plugins and dependencies
- Dependency management, automatic updates
- Consistent usage across all projects
- Automaitc parent versioning
- You can write own plugins as Maven is extensible

Installing Maven

- Bundled with *IntelliJ IDEA* (https://www.jetbrains.com/ru-ru/idea/download/)
  - Amend your `~/.bash_aliases` or `~/.zshrc`:

```
alias mvn='/bin/sh /opt/idea/plugins/maven/lib/maven3/bin/mvn'
```

- Via Package Manager

```
sudo apt-get install maven3   # Ubuntu
brew install maven            # Mac OS
```

- From Official Site: https://maven.apache.org/download.cgi

Projects and Artifacts

https://maven.apache.org/pom.html

- **Project** is *the* central entity in Maven. Maven builds projects
  - Defined by Project Object Model (POM), most commonly expressed through XML (pom.xml)
- Project build produces an **Artifact**, *e.g.* a JAR, Debian package, ZIP archive with HTML pages etc.
- Artifact is identified by its **Coordinates**:

  `groupId:artifactId:version[:packaging[:classifier]]`

- `groupId:` Organization and/or top-level project
  Convention: main package name, *`e.g.`* `yandex.market.content`
- `artifactId:` [Sub]project.
  Convention: kebab-case, *e.g.* ydb-sdk-java
- `classifier:` Used to pick platform-dependent artifacts, or source-JAR/javadoc-JAR instead of the lib itself
- `packaging:` Artifact type (*e.g.* test-jar to depend on tests)
- `version:`
  - xxx-**SNAPSHOT**: Development snapshot. Multiple w/same ver allowed, latest by mtime is picked during build
  - xxx: Release. Stable release artifact, immutable

Project Object Model – pom.xml

```xml
<project>
    <modelVersion>4.0.0</modelVersion>
    <groupId>com.yandex.market</groupId> <!-- unique company or group name where project was created -->
    <artifactId>market</artifactId>      <!-- unique project name -->
    <packaging>jar</packaging>           <!-- packaging method -->
    <version>1.0-SNAPSHOT</version>      <!-- project version -->
    <name>com.yandex.market</name>
    <url>http://maven.apache.org</url>
    <dependencies>
        <!-- https://mvnrepository.com/artifact/junit/junit -->
        <dependency>
            <groupId>junit</groupId>
            <artifactId>junit</artifactId>
            <version>4.13.2</version>
            <scope>test</scope>
        </dependency>
    </dependencies>
    <build>
        <plugins>
            <plugin>
            //...
            </plugin>
        </plugins>
    </build>
</project>
```

A combination of `groupId:artifactId:version` defines the unique identificator

JARs

https://docs.oracle.com/en/java/javase/17/docs/specs/jar/jar.html

- **JAR** (**J**ava **AR**chive) is just a ZIP archive with **compiled Java classes**, **resources** and **metainformation**
- **Compiled classes** and **class resources** are put into directories corresponding to Java packages.
  - Top-Level class `ru.hse.java.HelloWorld => ru/hse/java/HelloWorld.class`
  - Anonymous, inner and static inner: `ru.hse.java.HelloWorld.Insider => ru/hse/HelloWorld$Insider.class`
- Most important **metainformation** is the Manifest, META-INF/MANIFEST.MF:

```
Manifest-Version: 1.0
Main-Class: <Fully Qualified Class Name>
                     <...>
```

```
java -jar my-project-1.0.jar
```

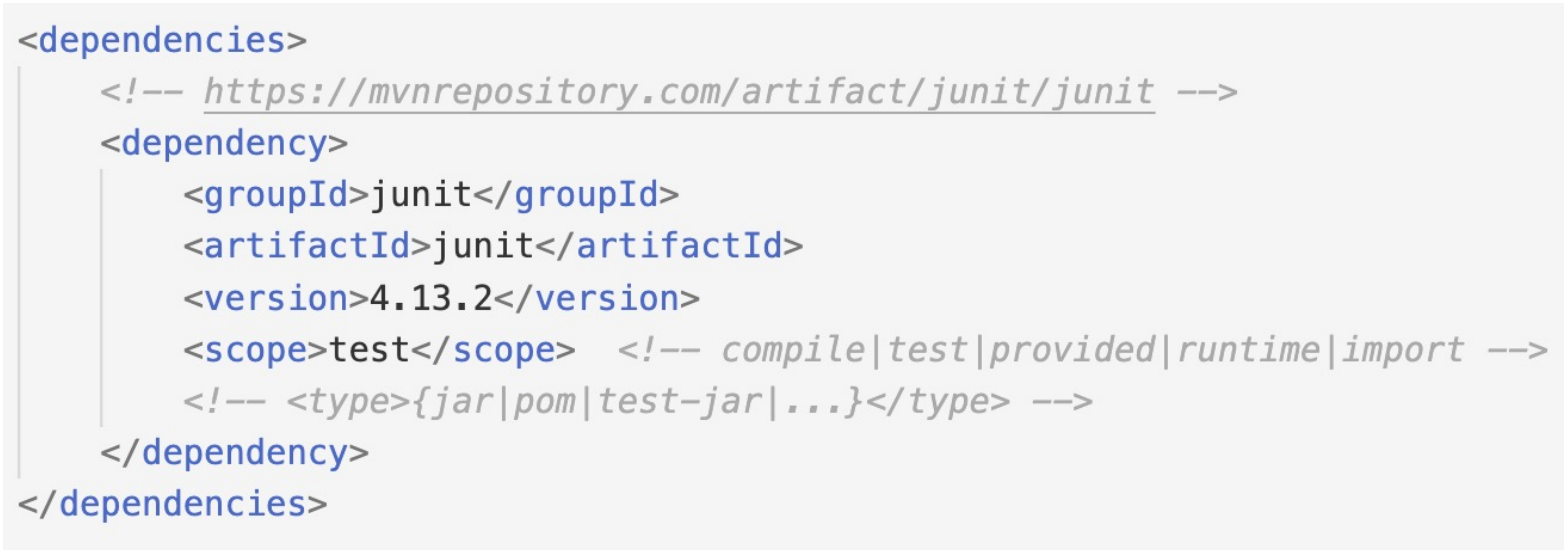
- META-INF/ directory MAY also contain:
  - Digital signature files (`*.RSA, *.DSA, SIG-*`)
  - **Service Provider** definitions (`META-INF/services/<fully-qualified name of Service Class Impl>`)
    @see future seminar on DI

Artifact Repository

- Artifacts are stored in and retrieved from a **Repository**
- **Remote** (public or private), *e.g.* Maven Central
- **Local Repository** (~/.m2/repository): Locally built + Cached from Remote
- Artifact repositories are *the* reason that Maven became hugely successful
- Single Source of Truth for dependency resolution
- Useful enough to be used by other build tools, *e.g.* Gradle, sbt, leiningen, Ivy, …
- Maven build (`e.g. mvn clean`) downloads artifacts necessary for the build
- …including plugins. Plugins **are** artifacts, too!
- Maven tries your **Local** Repository first!
- *NB:* Artifact resolution errors are cached for 1h, this helps:
  ```
  find ~/.m2/repository/your/artifact -name '*.lastUpdated' -
  delete
  ```

Dependencies

http://maven.apache.org/guides/introduction/introduction-to-dependency-mechanism.html



```xml
<dependencies>
    <!-- https://mvnrepository.com/artifact/junit/junit -->
    <dependency>
        <groupId>junit</groupId>
        <artifactId>junit</artifactId>
        <version>4.13.2</version>
        <scope>test</scope>  <!-- compile|test|provided|runtime|import -->
        <!-- <type>{jar|pom|test-jar|...}</type> -->
    </dependency>
</dependencies>
```

- More detailed rationale for using exact versioning: https://jlbp.dev/JLBP-14
- More about scopes: here

Transitive Dependencies

`A -> B -> C -> D 1.5 and A -> E -> D 1.2`

- `compile`-scoped Dependencies are **Transitive:** you implicitly depend on **dependencies of your dependencies**
- Other scopes are NOT transitive
- **Bill of Materials (BOM) Artifacts:** Common dependencies and plugins
  - <packaging>pom</packaging>
  - Everything from BOM is included in your POM when you add a <dependency> on it (with <scope>import</scope>)
- **Dependency Tree**: `mvn dependency:tree`
- **No** cyclic dependencies!
- If you different versions of the same artifact via transitivity, you must explicitly **exclude** it:

```xml
<exclusions>
    <exclusion>
        <groupId>org.slf4j</groupId>
        <artifactId>slf4j-api</artifactId> <!-- Maven already knows the version -->
    </exclusion>
</exclusions>
```

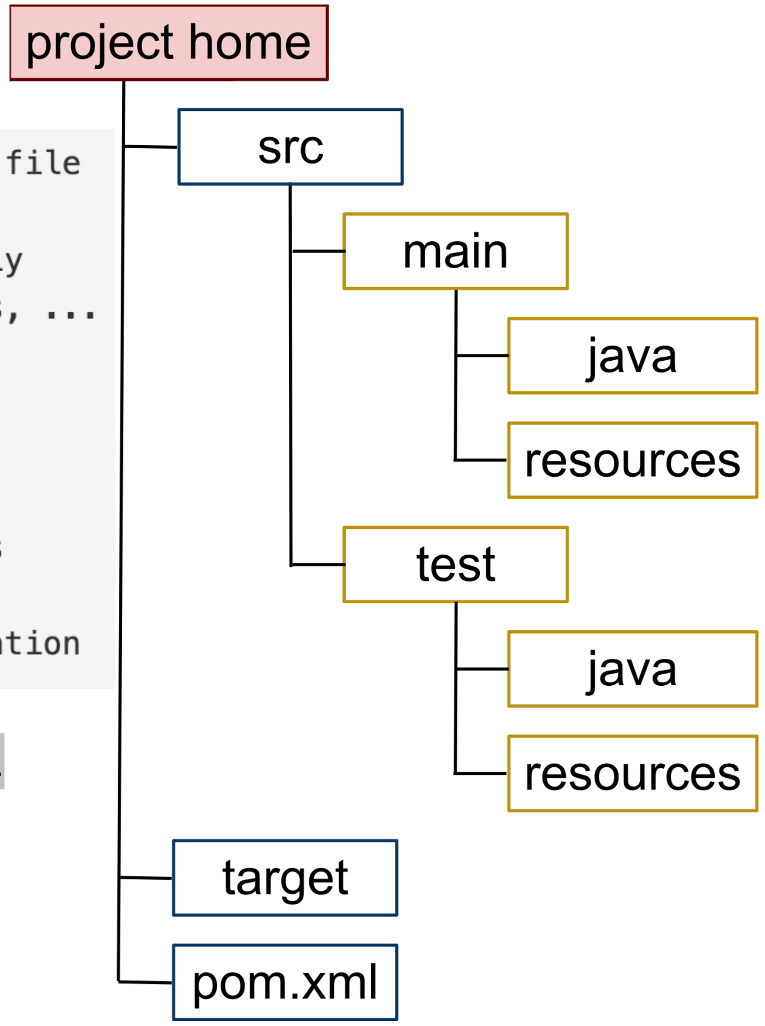
- Then, add an explicit dependency, picking a suitable artifact version:
- Pick max version from dependency:tree
- If you are feeling lucky live on the bleeding edge, use the latest version [with the same major.] available

* More about transitive dependencies: here

Typical Maven Project

- Directory Structure (Convention):

```
pom.xml                            # "POM" (Project Object Model) specification as XML file
src/main/java/                     # Java source code
src/main/{groovy,kotlin,proto,...}/ # Groovy, Kotlin, Protobuf, ... sources, respectively
src/main/resources/                # JAR resources, e.g. message bundles (i18n), images, ...
src/test/{java,resources}/         # Test sources and resources
target/                            # Artifact and corresponding files
your-artifact-0.0.0-SNAPSHOT.jar   # Artifact
classes/                           # Compiled classfiles
generated-source/                  # Generated source code, e.g. Protobuf class sources
generated-classes/                 # Classfiles built from generated code
surefire-reports/                  # Unit Test reports, used by e.g. Continuous Integration
```



- To build the project and install the artifact to local repository, run: `mvn install`

Build Lifecycle

- Maven is a **generic tool** and **delegates** most of the work to Plugins
- Plugins are **Artifacts**! They can be released independently of Maven, consumed from your enterprise Artifact Repository etc.
- Build has a linear Lifecycle composed of multiple Phases. Some Default Lifecycle phases are:

```
validate →                                     # Validate project, e.g. dependency versions
{generate,process}-{sources,resources} →       # Generate source code and resources
compile →                                      # Compile source code
{generate,process}-test-{sources,resources} →  # Generate test code and resources
test-compile →                                 # Compile test code
test →                                         # Run tests. Skip: -DskipTests/⚡ in IntelliJ IDEA
package →                                      # Create the artifact, e.g. JAR
verify →                                       # Verify the artifact, e.g. run integration tests
install →                                      # Add artifact to local repository
deploy                                         # Deploy artifact to remote repo/Docker repo/...
```

- Plugins execute Goals (=build actions) @ specific Phase(s) or by explicit user request (e.g., `mvn exec:exec`)
- There are default phase-goal bindings + you can define your own

* More about lifecycle: here

Build Lifecycle: Goals

Lifecycle

```
mvn install
```

```
maven-compiler-plugin:compile (compile) →
maven-compiler-plugin:testCompile(test-compile) →
maven-surefire-plugin:test (test) →
maven-jar-plugin:jar (package) →
maven-install-plugin:install (install)
```

```
mvn clean
```

```
mvn clean install
```

clean Lifecycle

default Lifecycle

Plugin

```
mvn dependency:tree
```

`maven-dependency-plugin`

Goal = `tree`

```
mvn exec:exec
```

```
mvn clean:clean
```

Parent POM

https://maven.apache.org/guides/introduction/introduction-to-the-pom.html#project-inheritance

- Projects can inherit configuration from other projects (**Parent POMs**).
- Parent POM specifies common build patterns for multiple projects
- Common Usages:
  - Unify dependency versions (`<dependencyManagement>`)
  - Unify plugin versions & configuration (`<pluginManagement>`)
  - Define properties (=project attributes) used throughout all your projects (`<properties>`). Interpolation syntax: ${property}
  - Specify Artifact Repository configuration (`<repositories>, <pluginRepositories>`). **Discouraged**, use `settings.xml` in project root instead

Parent POM

```
<project>
  <groupId>ru.hse.java</groupId>
  <artifactId>common</artifactId>
  <version>0.0.1</version>
  <packaging>pom</packaging>

  <!-- ... -->

</project>
```

Child POM

```
<project>
  <parent>
    <groupId>ru.hse.java</groupId>
    <artifactId>common</artifactId>
    <version>0.0.1</version>
    [<relativePath>../pom.xml</relativePath>]
  </parent>
  <!-- ... -->
</project>
```

Multi-Module Projects

https://maven.apache.org/guides/introduction/introduction-to-the-pom.html#project-aggregation

- Root project explicitly lists subprojects in `<modules>`
- Subprojects can depend on each other
- Directory Structure:

```
pom.xml          # Root POM
subproject1/
  pom.xml        # Sub-Project 1 POM
  src/{main,test}/{java,resources}/...
subproject2/
  pom.xml        # Sub-Project 2 POM
  src/{main,test}/{java,resources}/...
common/
  pom.xml        # Common Libs POM
  src/{main,test}/{java,resources}/...
...
```

Root POM

```
<project>
    <modelVersion>4.0.0</modelVersion>
    <groupId>ru.hse.java</groupId>
    <artifactId>root</artifactId>
    <version>1.0-SNAPSHOT</version>
    <packaging>pom</packaging>
    <modules>
        <module>subproject1</module>
        <module>subproject2</module>
        <module>common</module>
    </modules>
</project>
```

Sub-Project 1 POM

```
<project>
    <modelVersion>4.0.0</modelVersion>
    <groupId>ru.hse.java</groupId>
    <artifactId>subproject1</artifactId>
    <version>1.0-SNAPSHOT</version>
    <!-- depends on common... -->
</project>
```

Building a Multi-Module Project

- Build both the root project and all of its subprojects (topologically sorting dependencies):

```
mvn [clean] install
```

- **[Typical]** Build `subproject1` and everything it depends on (e.g., some common libs):

  ```
  mvn -am -pl :subproject1 [clean] install
  ```

  - **`-pl`** – build specified module
  - **`-am`** – build dependent project list, if the list is specified (“**al**so **ma**ke”)

- **[MORE RARE]** Build common and everything that depends on IT (`subproject{1,2}):`

  ```
  mvn -amd -pl :common [clean] install
  ```

  Rebuild a common dependency (an utility library etc.) and check that everything that uses it still works

  `-amd` – build projects that depend on projects (“**al**so **m**ake **d**ependents”)

* More is here

Additional Resources

- Troubleshooting:
  - Tail of Maven output shows which project failed to build
    - Scroll up to the last lines of failed build (there will be A LOT), and you will see the error message
    - Google the error!
  - If the error you see is too generic, enable debug mode:

    `mvn -Xe <…>`

    and look for `ERROR` and `WARN` in the logs, these might give you an insight (or at least a search query…)
- Recommended Reading: *Maven by Example* (a bit dated but covers all the basics)

  https://books.sonatype.com/mvnex-book/reference/index.html
- Q&A @ Stackoverflow

  https://stackoverflow.com/questions/tagged/maven