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
- Demo!

What is *Maven*?

https://maven.apache.org/what-is-maven.html

- **Industry Standard** for building and managing Java-based projects
- **Declarative** (unlike `make`, `ant` which are imperative). Aims to be **DRY** (**D**o not **R**epeat **Y**ourself)
- **Uniform** & **Consistent**
  - Uniform *build lifecycle* customized via *plugins* activated @ well-defined *phases* in the lifecycle
  - Consistent project definition syntax (XML with Schema)
  - Consistent project layout, *e.g.* separate tests from source

  => You get dependency graphs, unit test & code coverage reports, automatic changelogs and more
- **Convention over Configuration** (but customization **is** possible)
- **Verbose** (XML). @see `polyglot-maven` (https://github.com/takari/polyglot-maven)
- Many orgs adopt *Gradle*, which is a Groovy-based DSL for project builds
  - Uses *mostly* the same concepts as Maven and benefits from existing Maven infrastructure
  - Standard for Android apps
- Build tools for other JVM languages often use Maven concepts. *E.g.* `sbt` (Scala), `leiningen` (Clojure)

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'
```

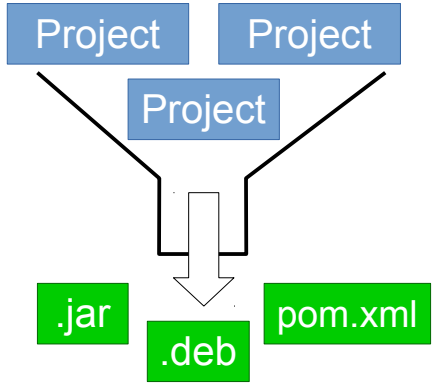
- Via Package Manager

```
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 file, .deb package, ZIP archive with HTML pages etc.
- Artifact is identified by its **Coordinates**:

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

  - `groupId`: Identifies organization and/or top-level project
    Typically is the same as your main package, `reverse.company.dns.your.project`
    E.g. `yandex.cloud.ydb`
  - `artifactId`: Project name. Convention is `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`: Used to pick a different artifact type (*e.g.* `test-jar` to depend on tests)
- `version` is **mostly** SemVer, with a few exceptions `-{alpha,beta,milestone,rc,cr,snapshot,final,ga,sp}`
- Snapshot and Release Versions:
  - xxx-**`SNAPSHOT`**: Development snapshot. Multiple w/same ver allowed, latest by mtime is picked during build
  - xxx: Release. Stable release artifact, immutable

Aside: JAR Files

- .JAR files are just fancy ZIP archives which contain **compiled Java classes**, **resources** and **metainformation** (`META-INF/`)

  @see https://docs.oracle.com/javase/7/docs/technotes/guides/jar/jar.html
- **Compiled classes** and **class resources** are put into directories corresponding to Java packages.
  E.g. class `ru.hse.java.HelloWorld => ru/hse/java/HelloWorld.class`
  - No directories are created for anonymous, inner and static inner classes, because they are synthesized by the compiler
    E.g. class `ru.hse.java.HelloWorld.MyCoolClass => ru/hse/HelloWorld$MyCoolClass.class`
- **Uber JAR/Fat JAR**: JAR with all the classes you depend on. @see https://stackoverflow.com/a/36539885/3438672
- Most important **metainformation** is the Manifest, `META-INF/MANIFEST.MF`:

```
Manifest-Version: 1.0
Main-Class: <Fully Qualified Class Name>
<more key-value pairs...>
```

→

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

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

Artifact Repository

- Artifacts are stored in a **Repository**
  - **Local Repository** (`~/.m2/repository`): Local build artifacts + Cached artifacts from Remote
  - **Remote Repository**
    - Maven Central, the main Maven Repository (@see https://search.maven.org/, better search: https://mvnrepository.com)
    - ~~JCenter: *was once* more complete, performant and secure (offered HTTPS downloads when the Maven Central did not). Default for Android~~
    - Enterprise: *proxy* remote repository that delivers both proprietary artifacts and artifacts from Maven Central, JCenter etc. *E.g.* Sonatype Nexus, JFrog Artifactory
- Large artifact repositories are *the* reason that Maven became successful
  - Single Source of Truth
  - Useful enough to be used by other build tools, *e.g.* Gradle, sbt, Ivy
- Maven build (*e.g.* `mvn clean`) will download all the artifacts necessary to build your project
- Maven will first try the Local Repository
    - *NB:* Artifact resolution errors are cached! Keep calm and `find ~/.m2 -name '*.lastUpdated' -delete`

Dependencies

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

```
<dependencies>
  <dependency>
     <groupId>junit</groupId>
     <artifactId>junit</artifactId>
     <version>4.13.1</version> <!-- WARNING: Do not use Version Ranges -->

     [<!-- provided=JDK/JEE, runtime=execution only, import=BOM -->
     <scope>{compile|test|provided|runtime|import}</scope>]
     [<type>{jar|pom|test-jar|...}</type>]
     [<classifier>...</classifier> <!-- linux-x86_64, javadoc-sources, -->]
     [<optional>true</optional> <!-- For optional functionality -->]
  </dependency>
</dependencies>
```

Transitive Dependencies

- `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!
- **Exclusions:** if you got >1 of the same artifact via transitivity, possibly w/different versions

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

Exclude everything. Then add an explicit dependency, picking a suitable artifact version:

- Max version from `dependency:tree`
- The latest version [with the same `major.`] available

Build Lifecycle

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

```
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 IDE
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 repository/Docker repo/...
```

@see https://maven.apache.org/ref/3.6.3/maven-core/lifecycles.html

- Plugins execute **Goals** (=build actions)
  - At specific lifecycle **Phase(s)**. Default phase-goal bindings + you can define your own
  - By explicit user request

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
```

Build: Plugin Configuration

Run Custom Unit Tests in Parallel

```
<build>
  <plugins>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-surefire-plugin</artifactId>
      <version>${maven-surefire-plugin.version}</version>
      <configuration>
        <includes>
           <include>**/*Tezd.class</include>
        </includes>
        <forkCount>2C</forkCount> <!-- 2 threads/core -->
      </configuration>
    </plugin>
  </plugins>
</build>
```

`mvn -DskipTests ...` → skip tests but compile them. IDEA:

`mvn -Dmaven.test.skip ...` → skip tests altogether.
NOT RECOMMENDED

Run `HelloWorld.main`

```
<plugin> <!-- Run via: mvn exec:exec -->
    <groupId>org.codehaus.mojo</groupId>
    <artifactId>exec-maven-plugin</artifactId>
    <version>${exec-maven-plugin.version}</version>
    <configuration>
        <mainClass>ru.hse.java.HelloWorld</mainClass>
    </configuration>
</plugin>
```

Custom Phase-Goal Binding

```
<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-enforcer-plugin</artifactId>
  <version>${maven-enforcer-plugin.version}</version>
  <executions>
    <execution>
      <id>enforce</id> <!-- to merge same executions -->
      <phase>validate</phase>
      <goals>
        <goal>enforce</goal>
      </goals>
      <configuration><!-- exec config --></configuration>
      </execution>
  </executions>
</plugin>
```

Typical Maven Project – Simple

- 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
```

`clean` goal is **almost never** needed in simple projects. Incremental build **just works**

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
- Commonly Used to:
  - 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
```

  Build your module with updated common deps, *e.g.* after pulling updated common lib sources from VCS

- **[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

- **NEW:** In Maven 3+ you can use `verify` goal (which goes right before install) if you don't need to save artifacts to the local repository

  @see http://andresalmiray.com/maven-verify-or-clean-install/


WHAT DID YOU DO KID?
MVN CLEAN INSTALL
DUDE, WTF?!
imgflip.com

*Enterprise Maven

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

- You can build Parent POM as a module of a Multi-Module Project
  - Multi-module root POM *can also* double as a Parent POM, but this is *weird*
- Most enterprise projects have >1 Multi-Module **Sub**projects
  - …and MIGHT ALSO have >1 Parent POMs
- Rules:
  - Specify in every child POM who their parent POM is
  - Change the parent POMs `<packaging>` to the value `pom`
  - Specify in the parent POM the directories of its modules (children POMs)
- Directory Structure:

```
pom.xml        # Root POM
parent/
    pom.xml    # Parent POM (common configuration)
subprojectN/
    pom.xml    # Sub-Project N POM
    src/{main,test}/{java,resources}/...
```

Root POM

```
<groupId>ru.hse.java</groupId>
<artifactId>root</artifactId>
<version>1.0-SNAPSHOT</version>
<packaging>pom</packaging>
<modules>
  <module>parent</module>
  <module>subprojectN</module>
</modules>
```

Parent POM (boring!)

```
<groupId>ru.hse.java</groupId>
<artifactId>parent</artifactId>
<version>1.0-SNAPSHOT</version>
```

Sub-Project N POM

```
<parent>
  <groupId>ru.hse.java</groupId>
  <artifactId>common</artifactId>
  <version>1.0-SNAPSHOT</version>
  <relativePath>../parent</relativePath>
</parent>
```

Additional Resources

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

    ```
    mvn -Xe <...>
    ```

    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