4. Automated build with Maven

Helmut Neukirchen
helmut@hi.is
Together with Andri Valur Guðjohnsen

UNIVERSITY OF ICELAND

FACULTY OF INDUSTRIAL ENGINEERING, MECHANICAL ENGINEERING AND COMPUTER SCIENCE

Chapter objectives

- Understand the foundations and applications of automated builds.
- Learn and apply the build tool Maven.
- Learn to create Maven POM files.

Chapter contents

Section 2 is to a huge extend based on documentation from: https://maven.apache.org

4.1 Build management foundations
Motivation

- IDEs can automatically (even in the background) compile your Java source code.
  - Generated bytecode can then be executed by the Java Virtual Machine (JVM).
- But: there are a lot settings that influence the compilation, e.g.
  - Used Java version and other compiler options:
    - Other Java compiler / JVM versions might not be able to process your code.
  - What if your project setup is more advanced than what your IDE supports?
    - E.g. you depend on external libraries
      (e.g. GUI libraries →HBV201G Graphical User Interface Programming or test libraries →this course):
      - These libraries need to be downloaded manually and used during compilation and execution.
- You can in your IDE
  - Set the Java version to be used in your project,
  - add some external library to your project.
    - But the IDE does not download the external library for you (you need to do this manually).
  - But… →next slide.

Latest in →course "Graphical User Interface Programming" /"Viðmótsforritun"), you will work with external libraries. As you depend on these libraries, they are also called: dependencies

Motivation: Compilation by IDE may not be sufficient in all cases

- But:
  - What if you work in a team?
    - If all using the same IDE: put IDE-specific configuration files under version control: these contain, e.g., info on Java version to be used. ⇒ Everyone in the team will use same settings.
  - What if different IDEs are used?
    - Configuration files are IDE-specific. ⇒ No way to exchange these across IDEs.
  - What if you need to compile from command line?
    - E.g. to do continuous integration (CI), i.e. the server hosting the remote repository compiles at each commit the new changes

      →Course "Software Quality Management"
      - (to check that your changes integrate well with the other's changes).
      - The CI server needs to be able to run the compiler on the command line (without a graphical IDE).
  - What if external libraries are not available on all used computers?
    - Telling the IDE to use an external library is not enough. Need to *automatically* download the library.
- Automated build needed!
  - Can be called from command line, but IDE can call it as well.

Build management

- Build management is (just like version control →Ch. 2) part of software configuration management (SCM) →course HBV101G Introduction to Software Engineering.

- Build management=Automatic creation of the executable software product.
  - Automated build system: Based on rules for compilation and generation of software artifacts in general.

SW artifact=everything that is created during SW development

- Advantages:
  - Saves time during development (in comparison to manual build).
  - Reproducibility by ensuring that right compiler version, compiler options, and right library version are used.
    - Also allows others to build your software,
    - Allows you to build your software still after many years,
    - Automatic build of older versions by retrieving source files from version control.

E.g. when a bug was introduced: Find the underlying defect by finding out in which version the SW still worked and in which version it failed for the first time. Finding this version can be automated using `git bisect`.

- Disadvantage:
  - Takes some time to create the build script and to acquire the knowledge to create it.
    - But pays off the more often the automated build is executed – each build saves time in comparison to manual build.

How to build a Java project (→Ch. 3)

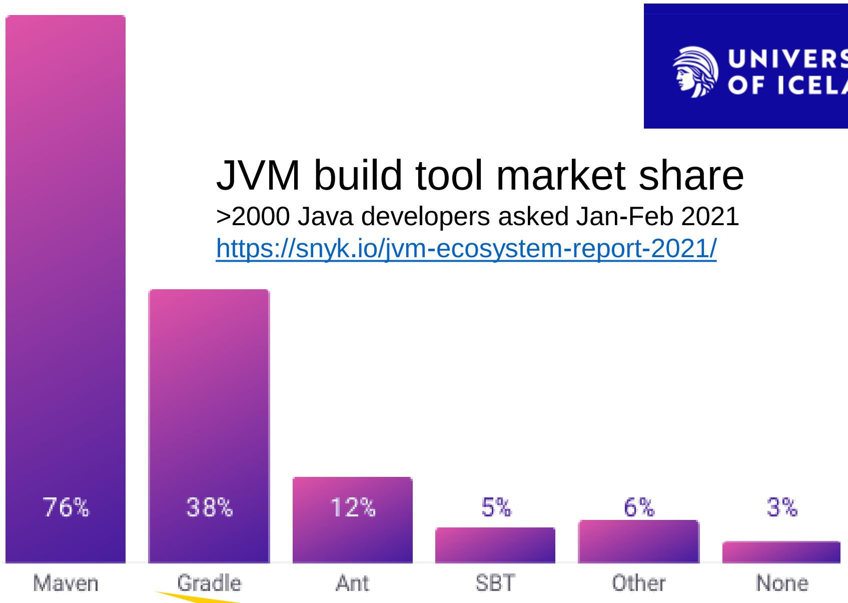
- Your build tool uses the **`javac`** command that is installed on your system.
  - Note: IDEs might have their own built-in Java compiler or can be instructed to use the one installed on your system.
- **`javac`** compiler takes all **`.java`** source files that are passed to it via command-line parameters and creates **`.class`** files containing bytecode as output ("target").
  - Build tools needs to be aware of all **`.java`** source files, so that it can pass them to the compiler.
- Some build tools are generic, i.e. programming language independent (e.g. **`make`**):
  - Need to tell build tool every tiny step, e.g. where is the source code, what compiler to use.
- Some build tools are programming language-specific (e.g. **`Maven`** for Java):
  - They know the compiler to use and assume a project layout so that they find the source code.

4.2 Maven
4.2.1 Introduction & Philosophy

- Maven is a build tool specific to Java/JVM-based.
- Apache open-source project:
  - Download, install, documentation: https://maven.apache.org
    - Other good tutorials:
      - https://www.tutorialspoint.com/maven
      - http://tutorials.jenkov.com/maven/maven-tutorial.html
  - Stable version: 3.x (new version 4.x is in the works.)

Yiddish word meaning “expert”:

JVM build tool market share
>2000 Java developers asked Jan-Feb 2021
https://snyk.io/jvm-ecosystem-report-2021/



Maven uses XML format for the build file description, Gradle uses its own language that you need to learn (but makes Gradle more flexible).

- You can call Maven on the command line.
- In addition, Maven is supported by IDEs.
  - IDEs then adjust their default project layout to the Maven layout.
    - I.e. "in which directory is the source code, in which directory goes the generated bytecode.
  - Start Maven from within IDE.

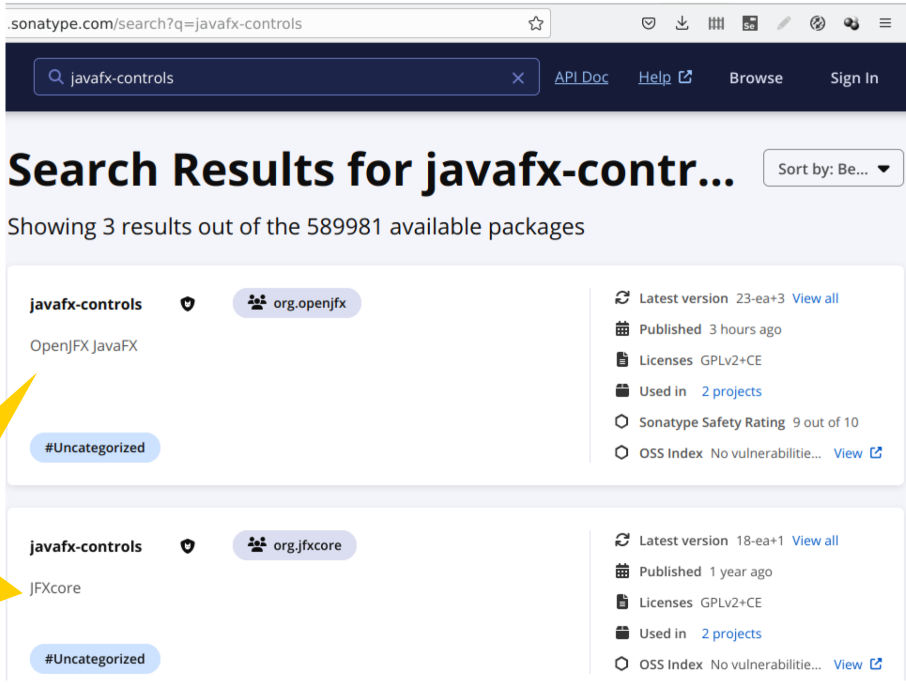
Maven & Dependency handling

- Maven automatically downloads all the external libraries your source code depends on (according to what you specified in the Maven file):
  - No need to download manually or to tell anyone compiling your code "you need to have downloaded the Xyz library" to build my project.
  - To speed-up build and reduce network traffic: Maven keeps a local repository of already downloaded dependencies locally on your computer.
  - By default downloaded from the Maven Central Repository:
    - The world's largest repository of open source software.
      - Browse it via https://central.sonatype.com

You may still find the old name: https://search.maven.org

  - Recursive: if your dependency has its own dependencies, these will be downloaded as well.

Multiple groups offer "**`javafx-controls`**" – to prevent using a potentially malicious version: important to know who is the official provider when you browse Maven Central and select the right variant to use.


.sonatype.com/search?q=javafx-controls

javafx-controls | API Doc | Help | Browse | Sign In

Search Results for javafx-contr... Sort by: Be...

Showing 3 results out of the 589981 available packages

javafx-controls — org.openjfx
OpenJFX JavaFX
#Uncategorized
Latest version 23-ea+3 View all
Published 3 hours ago
Licenses GPLv2+CE
Used in 2 projects
Sonatype Safety Rating 9 out of 10
OSS Index No vulnerabilitie... View

javafx-controls — org.jfxcore
JFXcore
#Uncategorized
Latest version 18-ea+1 View all
Published 1 year ago
Licenses GPLv2+CE
Used in 2 projects
OSS Index No vulnerabilitie... View

Excursion:
Dependencies & Trust

- Components submitted to the Maven Central Repository need to be cryptographically signed by the owner of that component.
  - To assure that they were created by that owner and were not modified.
    - As long as you trust that owner, you can trust that component.
      - Also: need to trust that the security of that owner was not compromised.
        - (That's why GitHub is enforcing two-factor authentication.
        - But: the key to sign components should anyway not be stored on GitHub and can still get stolen.)
  - And: need to trust that security of the server from where you downloaded was not compromised.
    - (Essentially the case with all software that you install.)
- Just like with all open-source software: the hope is that any security issue is sooner or later found because everyone can inspect the source code.
  - Well, severe log4j vulnerability discovered 12/2021 was in the open-source code since 2013…

Excursion: Anecdotes of Dependencies & Trust

Source: https://en.wikipedia.org/wiki/Npm_(software)

- From the Node.js NPM package manager repository used for Javascript – not Maven:
  - In 2018, two malicious packages were successfully added (but found – we obviously do not know about those not found) based on credentials stolen from the package developers.
  - In 2022, the developer of the packages **`colors`** and **`fakers`** was dissatisfied with big money earning companies using his code for free and thus changed his code, making his packages unusable, causing widespread disruptions, including Amazon AWS CDK.
    - GitHub (where his packages were hosted), decided to change the contents of the that developer's GitHub repository in order to restore the old version.

Well, he should have chosen a different license.

To avoid this, Google tries to minimise external dependencies.

Be aware that GitHub might change your repository contents.

- Helmut is not aware of any Maven Central Repository breakages.
  - Repository managed by company Sonatype: seem to do good job.
  - In theory, also here, compromised accounts of maintainers are possible.
  - Known attacks are uploading packages with similar package names (package name squatting) or same package name as popular packages, but with different group name to fool people.
    - E.g. when search for packages via https://central.sonatype.com

Take care to add the correct dependency.

Maven philosophy: Uniformity

- Maven provides a uniform build system:
  - A project object model (POM), an XML file with name **`pom.xml`**: describes your project and how to build it.
    - XML sometimes awkward to write (& read) – but at least it follows the XML syntax, so you do not need to learn a new language – just the Maven-specific XML tags.
      - Maven can create a skeleton XML file for you and your IDE may support you as well.

  - Uniformity: Once you familiarize yourself with how a Maven project is built, you automatically know how all Maven projects build.
    - Saving time when navigating new projects.
    - But Maven's assumes a certain project structure:
      - Structure of your existing project may not match,
      - IDEs may assume a different structure, but support also Maven structure.

Maven philosophy: Convention over Configuration

- Maven uses Convention over configuration:
  - Sensible defaults (based on conventions) are used.
  - Only if one wants to deviate from them, they need to be explicitly configured.

  - Example: Source code of a project by convention located in folder "**`src`**", generated bytecode in folder "**`target`**".

4.2.2. Maven basics
Software Development Life Cycle (SDLC) phases

- Maven supports various phases (e.g. developing, testing, delivering) of the software lifecycle, e.g.:
  - create a new project from a template (“archetype”), including a sample Maven **`pom.xml`** file.
  - **`compile`**: compile the source code of the project (incl. downloading all needed dependency).
  - **`test`**: test the compiled code by running automated tests written using a test framework (→later chapter).
  - **`package`**: package compiled code it in its distributable format, such as a JAR.
- In addition:
  - **`clean`**: cleans up (=delete) artifacts (=files) generated by prior builds.
- Each phase has internally goals registered that are provided by plugins (→next slide).
- See also: http://maven.apache.org/guides/introduction/introduction-to-the-lifecycle.html

Maven Plugins, Goals, and Phases

- Many plugins exist: https://maven.apache.org/plugins/
  - Maven itself can only download and run plugins.

The plugins themselves are downloaded from the Maven Central Repository (and copies kept in local repository) – just like the dependencies. ⇒When you run Maven for the first time, it will download a lot, just to get all the plugins.

No problem to not use the latest Maven version: rather the version of the plugins matter and these are downloaded.

- Example plugins:
  - **`compiler`** plugin knows how to compile Java code.
    - Has a "**`compile`**" goal that is bound to the compile phase (→later slide) to compile the main source files (but not the source code containing automated tests).
    - Possible to call **`compile`** goal from **`compiler`** plugin either explicitly from command line: **`mvn compiler:compile`** or via phase to which it is bound: **`mvn compile`**
      - Note: name of phase happens to be the same as name goal, but this is not always the case.
  - **`archetype`** plugin creates a new Maven-managed project using some template (called "archetype") stored in an archetype catalogue.
    - Has a "**`generate`**" goal to create a new project structure using a particular archetype from a particular archetype catalog.
    - As this archetype goal is not associated with a phase, the only way to call it from command line is: **`mvn archetype:generate`**

We will later use this to let Maven generate a sample project.

Maven Properties

- Properties are parameters to configure Maven and its plugins.
  - Properties have a good default value (→“convention over configuration”).
  - In interactive mode, Maven asks for property values.
    - Hit ENTER key to use default:
      - Default value is displayed in interactive mode within two colons are next to these two colons.
    - or enter new value. Example:

```
Choose a number or apply filter (format: [groupId:]artifactId, case sensitive contains): 1856:
```

    - The value **`1856`** is the default value, but you may enter a new one.
  - In non-interactive mode, exactly these default values are used anyway.
  - Possible to set a property value from command line using ***`-Dproperty=value`*** for which you can set values. E.g.:
    - **`-DinteractiveMode=false`** to set non-interactive mode
      - (default value is **`true`**, i.e. interactive mode is on)

Caveat Windows PowerShell users: PowerShell “eats” the hyphen ⇒ Need surround each hyphen parameter by double quotes, i.e. use: **`mvn "-Dproperty=value"`**

4.2.3 Maven Tutorial
Download & Installation

Install Maven at home while reading this! So that you do not waste time during class with installing.

- Maven itself is implemented in Java:
  - Java needs to be installed. (You use Maven to build Java code, so you have Java installed).
  - Maven runs an all relevant platforms.
  - Either use the latest Maven 3.9.9 or any ≥3.6.3. – they are all compatible with each other.
    - Maven is only loading and running plugins: old versions can still load and run new plugin versions.
- Download as described on https://maven.apache.org/download.cgi
  - I.e.: download “Binary zip archive”
- Install as described on https://maven.apache.org/install.html
  - You have already a JDK installed, so ignore that text.
  - I.e.: unpack zip archive to some directory, e.g. home or system directory.
  - Add Maven’s bin folder to your environment variable **`PATH`**.

Do a web search how set the environment variable PATH on you system. You need to close and restart your terminal to make the changes work.

- Alternative install instructions:
  - https://www.baeldung.com/install-maven-on-windows-linux-mac

Example session (1)

We will do all the steps of the example session as part of the next assignment. ⇒ Sufficient to read at home these slides & watch the videos showing the example sessions.

Start Maven and use a plugin to generate a quickstart project

- For a quick start, we want to let Maven create a new project using the **`archetype`** plugin, but we do not know which archetypes exist…

⇒Just type on command line: **`mvn archetype:generate`**

Try whether your shell can auto-complete the parameters: type **`a`** and **`r`** and press TAB key to get **`archetype:`**, type **`g`** and TAB to get **`generate`**.

  - In particular the very first run takes some time:
    - Maven downloads some standard plugins and the archetype plugin (and all their dependencies) from Maven Central Repository.

The number of available archetypes and the number of the suggested default may differ from Maven version to version.
With the 3.9.6 version, this is most likely: 3352 archetypes, suggested default: 2001
(The videos shows some older version.)

  - As Maven uses interactive mode by default, it will list all available archetypes and suggests a good default archetype:

```
Choose a number or apply filter (format: [groupId:]artifactId, case sensitive contains): 2217:
```

  - The list is probably to long to scroll back, but the suggested default is:

```
2217: remote -> org.apache.maven.archetypes:maven-archetype-quickstart (An archetype which contains
a sample Maven project.)
```

    - I.e. selection 2217 is from remote repository **`org.apache.maven.archetypes`** the plugin **`maven-archetype-quickstart`** .
  - Choosing number 2217 will create a small Java sample project.
  - Just hit enter to use this default.

Example session (2)

Choose the version of the quickstart project and provide project ID and name

- Different versions of this archetype exist, so Maven asks us which one to use – instead of the default (=the latest version), we rather enter **6**):

```
Choose org.apache.maven.archetypes:maven-archetype-quickstart version:
1: 1.0-alpha-1
2: 1.0-alpha-2
3: 1.0-alpha-3
4: 1.0-alpha-4
5: 1.0
6: 1.1
7: 1.3
8: 1.4
Choose a number: 8:
```

Choose 6 to use v1.1. of the quickstart archetype that creates a sample Maven project.

Choice 8 (=v1.4) is the default, but the older versions will give us an even simpler (=easier to understand) project, so in this tutorial, we rather choose 6, i.e. v1.1 of the quickstart archetype.

- The **`archetype:generate`** goal continues to ask questions (and for some, even no default values exist), e.g.
  - groupId: a globally unique identifier of your project group or company (such as org.apache.maven), typically a reverse domain name (just like a Java package name), e.g. enter: **`is.hi.cs.helloworld`**
  - artifactId: the name of your project, used as name of the jar file (if a jar file is generated out of your project), e.g. enter: **`helloworld`**

For everything else (including final confirm), use the default values, i.e. press enter key.

GroupID, ArtifactID (together with Version) are also called “GAV”.

Example session (3)

Based on: https://maven.apache.org/guides/getting-started/maven-in-five-minutes.html

Provide quickstart project version, project ID and name via command line

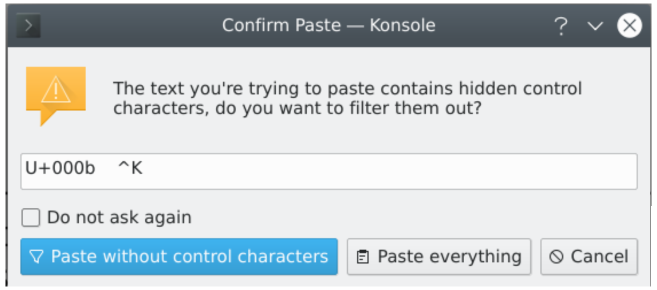
- Alternatively, if we know what we are doing, we could have used non-interactive mode and pass all property values via command line (as single-line command):

  -
    ```
    mvn archetype:generate -DinteractiveMode=false
    -DarchetypeArtifactId=maven-archetype-quickstart
    -DarchetypeVersion=1.1
    -DgroupId=is.hi.cs.helloworld -DartifactId=helloworld
    ```

Reminder: Windows PowerShell users: PowerShell “eats” the hyphen ⇒ Need surround each hyphen parameter by double quotes, i.e. use:

```
mvn archetype:generate
"-DinteractiveMode=false"
"-DarchetypeArtifactId=maven-
archetype-quickstart"
"-DarchetypeVersion=1.1"
"-DgroupId=is.hi.cs.helloworld"
"-DartifactId=helloworld"
```

Reminder from Ch. 3: copy-pasting code snippets from these slides may copy hidden characters that you do not see in your editor. The Linux console windows here is smart enough to notice that – your console window or IDE editor might not be...


Confirm Paste — Konsole

The text you're trying to paste contains hidden control characters, do you want to filter them out?

U+000b ^K

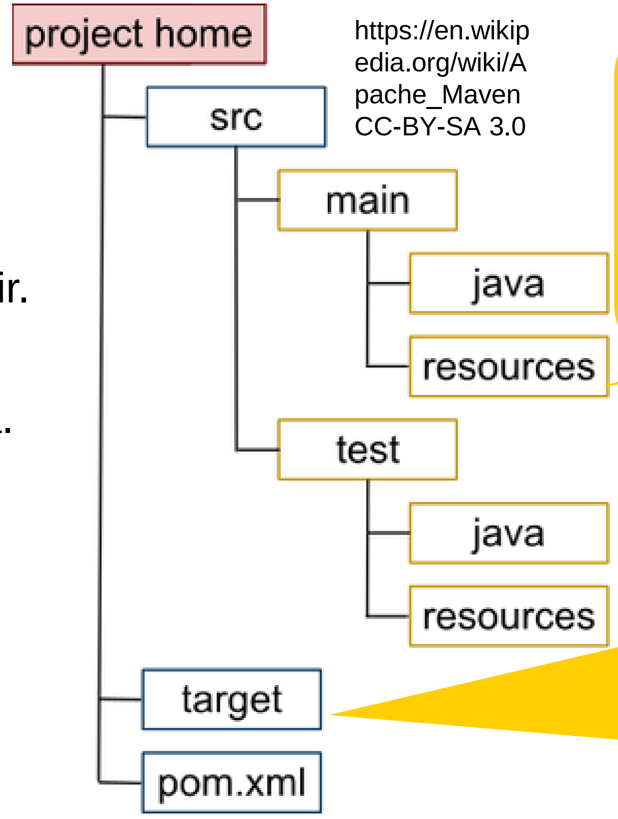
☐ Do not ask again

- If you do not know all the parameters, you can always try the interactive mode;
  - Once you learned the parameters, you can use non-interactive mode.
- If these parameters never change, then rather hardcode the property values in the `pom.xml` file (→next slides).

Result of Example session

- The **`archetype:generate`** goal has created for you a Hello World project:
  - A new **`helloworld`** project home directory with a default project structure and some HelloWorld Java source code incl. some (bogus) JUnit test.
  - **`src`** directory contains sources:
    - **`main`** contains all sources needed to build executable:
      - **`java`** contains all Java source code of implementation (including sub-directories according to package structure).
      - Other needed files (e.g. data) would go in a **`resources`** dir.
    - **`test`** for everything needed to test the implementation:
      - Java sources containing (JUnit) tests & any other test data.
  - **`target`** directory used to store generated bytecode.
  - **`pom.xml`** (Project Object Model) file contains info about project used by Maven to build the project, e.g.:
    - Dependencies, e.g. to JUnit framework in the example.
    - Can be later modified/extended for advanced usage,
    - Assumes the above project directory structure (i.e. convention over configuration)

JUnit will be covered in →Chapter 5. (Ignore for the time being.)



https://en.wikipedia.org/wiki/Apache_Maven CC-BY-SA 3.0

Files in **`resources`** get 1:1 copied to **`target`** directory or generated JAR: use it to store data, e.g. any file that you need to load.

Do not put **`target`** directory under version control: add **`target/`** to **`.gitignore`**. (Maven anyway creates it during build.)

Excursion: eXtensible Markup Language (XML)

To read and write your own `pom.xml` file that is used by Maven, you need to know the XML syntax.

https://www.w3schools.com/xml/default.asp

- XML is format to describe & store data – in fact, HTML (used to describe web pages) is almost XML.
  - HTML has pre-defined tags – XML allows to define your own tags to be understand by your own tools.

Maven is such a tool.

- Textual representation of nested (i.e. tree-like structure) textual data.
  - Extensible: you define your own tags (you or: your tool needs to give them a meaning)
    - Valid format (i.e. which tags can be used) of a XML document can be specified using an XML Schema.
- Constituents of an XML document:

Tag names are **case sensitive** (typically: all lowercase).

  - Elements: Start and end tags surrounding data (content): `<name>Helmut</name>`
    - In absence of content: just `<name/>` as shorthand for `<name></name>`
  - Attributes: optional values provided as part of (start) tag: `<tag attr="value">`
  - Contents: By default: UTF-8 encoded text.
    - Reserved characters (e.g. `<` or `&`) need to be replaced (e.g. by `&lt;` or `&amp;`).
  - Comments:
    - `<!-- a comment -->`
    - **Comment out whole structures**: use a non-existing XML processing instruction (these start with `<?`):

```
<?ignore
  <person>
    …
  </person>
?>
```

Comment out XML structure

Your tool give `<name>` a meaning: is the first name? last name? or both? Is this a birth date? (The tag's name suggests that it is not a date, but your funny tool could expect a date there.)

- Example:

Nested data

```
<persons>
  <person id="123456789">
    <name>Smith</name>
    <place>London</place>
    <year>1934</year>
    <!-- I hate XML -->
  </person>
  <person id="987654321">
    <name>Miller</name>
    <place>Oxford</place>
    <year>1974</year>
    <!-- I love it -->
  </person>
</persons>
```

Tag with an attribute.

Each opening tag needs a matching closing tag.

Comment

Result of Example session: Generated `pom.xml`

Element containing Maven project description

xmlsn=XML namespace to avoid clashes with others who also use tags such as **project**, **modelVersion**, etc. **schemaLocation**=Where the XML schema can be found that describes valid syntax of this XLM document.

```
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>

  <groupId>is.hi.cs.helloworld</groupId>
  <artifactId>helloworld</artifactId>
  <version>1.0-SNAPSHOT</version>
  <packaging>jar</packaging>

  <name>helloworld</name>
  <url>http://maven.apache.org</url>

  <properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
  </properties>

  <dependencies>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>3.8.1</version>
      <scope>test</scope>
    </dependency>
  </dependencies>
</project>
```

Version of the used **pom.xml** syntax – This and the above line are always the same, i.e. ignore.

Values as provided to **archetype:generate**

Version 1.0 assumed as default. **SNAPSHOT** means that this is a development version not intended yet for release.

What to do in the **package** phase. (Creating a JAR is assumed as default).

Human readable project name and URL of project web page. (The default URL **maven.apache.org is** not good: replace URL or delete this tag!)

Character encoding used in source code files (e.g. for Icelandic characters) and therefore to be respected by, e.g., the Java compiler. (Default encoding used by Java compiler is the operating system's default encoding – which varies from OS to OS.) Not specifying the encoding makes the entire build platform-specific (which is bad).

As the quickstart archetype creates JUnit tests, this dependency on the JUnit framework was also added: it is available in the Maven Central Repository and identified via the given groupId, artifactId, and version (GAV).

Dependency only needed during the phase "test", but not in phase "compile".

No need to mention the Java compilation itself: Maven simply knows how to compile Java: in the compile phase, simply all classes and packages in **src/main/java** will be compiled (and in the test phase, in addition **src/main/java**).

Structure of a `pom.xml` file
(and how to comment/comment out in XML)

- Minimal **`pom.xml`** file (of a project that has no dependencies):

```
<project>
 <modelVersion>4.0.0</modelVersion>
 <groupId>com.mycompany.project</groupId>
 <artifactId>my-main-product</artifactId>
 <version>2.1.0</version>
</project>
```

Also known as "GAV".

POM model language version: Maven 2 & 3 use : 4.0.0

Group=organisation/company or a project, typically reverse domain name notation used.

Name of the delivered product, e.g. name of the JAR file that will be delivered.

Version. Add suffix **`-SNAPSHOT`** for not yet finished versions.

- In addition, a POM has further elements (→later slides):
  - Properties: e.g. compiler versions to be used.
  - Dependencies: specify (using GAV) on which libraries your code depends.
    - (Maven downloads them for you – if they are available in a repository.)
  - Build: if default build tool chain is not sufficient, you can add build plugins.
  - Reporting: e.g. to create test coverage report as HTML file (→later chapter)

Maven out-of-the-box compiles all Java files it finds in your project source code directory: Not even needed to add anything for that to the POM – only if you need non-default settings.

- Comments in XML: **`<!-- Some comment -->`**
  - Comment out whole structures: use a non-existing XML processing instruction (these start with **`<?`**):

```
<?ignore
  <project>…</project>
?>
```

The **`pom.xml`** files that you create tend to be hard to read for a human: Make good use of comments!

Example of how to comment out the XML **`<project>`** … **`</project>`** block.

Excursion:
Where are actually all the downloaded dependencies stored?

- Executing **`archetype:generate`** did download a lot of plugins and dependencies.
  - Note: as we did not yet even compile the project, no project dependencies have been downloaded, yet – only the Maven plugins themselves and their dependencies.
- But: where are the downloaded dependencies actually?
  - All dependencies that are downloaded by Maven are typically stored in directory "**`.m2`**" in your home directory: your local repository.
    - The groupId is used to create subdirectories within the **`.m2`** directory.
    - The version number is used to create accordingly subdirectories thus enabling to store different versions (="GAV"-based dir structure).
      - JAR files are used to store the contents of a dependency.
        - Typically, the owner includes only the compiled **`.class`** files.
      - A **`pom.xml`** file describing the dependency (using GAV).
  - During compilation/execution, the compiler/JVM is pointed to the JAR files in **`.m2`**.

```
/home/helmut/.m2/repository/
└── org
    └── apache
        └── maven
            └── archetypes
                ├── maven-archetype-bundles
                │   ├── 1.3
                │   │   ├── maven-archetype-bundles-1.3.pom
                │   │   ├── maven-archetype-bundles-1.3.pom.sha1
                │   │   └── _remote.repositories
                │   └── 1.4
                │       ├── maven-archetype-bundles-1.4.pom
                │       ├── maven-archetype-bundles-1.4.pom.sha1
                │       └── _remote.repositories
                └── maven-archetype-quickstart
                    ├── 1.1
                    │   ├── maven-archetype-quickstart-1.1.jar
                    │   ├── maven-archetype-quickstart-1.1.jar.sha1
                    │   ├── maven-archetype-quickstart-1.1.pom
                    │   ├── maven-archetype-quickstart-1.1.pom.sha1
                    │   └── _remote.repositories
                    ├── 1.3
                    │   ├── maven-archetype-quickstart-1.3.jar
                    │   ├── maven-archetype-quickstart-1.3.jar.sha1
                    │   ├── maven-archetype-quickstart-1.3.pom
                    │   ├── maven-archetype-quickstart-1.3.pom.sha1
                    │   └── _remote.repositories
                    └── 1.4
                        ├── maven-archetype-quickstart-1.4.jar
```

Example session (4): If you use Maven ≤3.8.x

Try compiling the project

Maven searches for the **`pom.xml`** in your working directory, i.e. you need to be in the project directory **`helloworld`** when you run Maven.

- Now that the project has been created, Maven can be used to compile, i.e. in **`helloworld`** try:
- **`mvn compile`** to compile Java classes.
- But, if you have a recent JDK version installed, you will get an error:

```
[INFO] --- maven-compiler-plugin:3.1:compile (default-compile) @ helloworld ---
[ERROR] COMPILATION ERROR :
[INFO] -------------------------------------------------------------
[ERROR] Source option 5 is no longer supported. Use 8 or later.
[ERROR] Target option 5 is no longer supported. Use 8 or later.
[INFO] 2 errors
[INFO] -------------------------------------------------------------
[INFO] ------------------------------------------------------------------------
[INFO] BUILD FAILURE
```

Maven compiler plugin v3.1 is used.

I.e. Maven compiler plugin 3.1 tells the Java compiler that the provided source code conforms to Java 5 and it should generate bytecode for Java 5 JVM.

- Maven ≤3.8.x uses its compiler plugin version 3.1: calls **`javac`** and tells it to assume Java 5 code.
  - Oracle decided for recent Java compilers to stop support for compiling Java source code and for generating Java byte code of Java version 7 and earlier.

⇒Need to configure the Maven project so that the Maven compiler plugin tells the JDK that a newer Java source code and target bytecode version is assumed.

Example session (4): If you use Maven ≥3.9.x

Try compiling the project

Maven searches for the **`pom.xml`** in your working directory, i.e. you need to be in the project directory **`helloworld`** when you run Maven.

- Now that the project has been created, Maven can be used to compile, i.e. in **`helloworld`** try:
- **`mvn compile`** to compile Java classes.
- But, you will get a warning:

Maven compiler plugin v3.11.0 is used.

I.e. Maven compiler plugin 3.11.0 tells the Java compiler that the provided source code conforms to Java 8 and it should generate bytecode for Java 8 JVM.

```
[INFO] --- compiler:3.11.0:compile (default-compile) @ helloworld
[INFO] Changes detected - recompiling the module! :source
[INFO] Compiling 1 source file with javac [debug target 1.8] to target/classes
[WARNING] bootstrap class path not set in conjunction with -source 8
[WARNING] source value 8 is obsolete and will be removed in a future release
[WARNING] target value 8 is obsolete and will be removed in a future release
[WARNING] To suppress warnings about obsolete options, use -Xlint:-options.
[INFO] ------------------------------------------------------------------
[INFO] BUILD SUCCESS
```

- Maven ≥3.9.x uses its compiler-plugin version 3.11.0: calls **`javac`** and tells it to assume Java 1.8 or just: 8.
  - Oracle decided for recent Java compilers to stop in future the support for compiling Java source code and for generating Java byte code of Java version 8 and earlier: while it is still supported now, a warning is given.
  - Another warning is given about the "bootstrap class path": this is the Java runtime library (rt.jar) that is used:
    you should set the version of that to the same version that is assumed for the Java source code and byte code.

⇒Need to configure the Maven project so that the Maven compiler plugin tells the JDK that a newer Java source code, target bytecode, and runtime library version is assumed.

Example session (5)
Setting Java version and successfully compiling the project

BTW: Even if you have, e.g., Java 21 JDK installed: it would still support Java 17 (but not Java 5 or 7). Check using **`javac -version`**

- To address this issue, you need to instruct Maven to tell the Java compiler to assume a higher Java source code, bytecode and runtime version, e.g. Java 17 or 21:
- Change **`pom.xml`** from

BTW: setting the source code encoding is needed when you use, e.g., Icelandic characters in your source code. Take care to have your source code editor as well set to that encoding. Linux and Mac typically use UTF-8 by default, but Windows uses by default some Windows codepage (and as soon as Windows users share source code with Linux or Mac users, the Icelandic character look strange on the other's machine).

```
<properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    <maven.compiler.source>1.7</maven.compiler.source>
    <maven.compiler.target>1.7</maven.compiler.target>
</properties>
```

to

```
<properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    <maven.compiler.source>21</maven.compiler.source>
    <maven.compiler.target>21</maven.compiler.target>
    <maven.compiler.release>21</maven.compiler.release>
</properties>
```

Change these to make the compiler assume Java 21 source code, byte code ("target").

Add this line to make JDK use the Java 21 runtime library.

- Now, try **`mvn compile`** – …now, it should work without error or warning!
  - Maven should now have created a **`target`** directory with the generated bytecode in **`target/classes`**.

Example session (6)
Further phases, e.g. package a JAR file

- In fact, Maven can be used for all the phases of the software development lifecycle,
  - e.g. compile, test, package, etc.
- **`mvn compile`** to compile Java classes.
- **`mvn test`** to run JUnit tests (covered in a →later chapter).
- **`mvn package`** to create a package (default: JAR) file with your bytecode.
  - After creating the JAR, you can now (inside your **`helloworld`** project directory) run your JAR package:
    `java -cp target/helloworld-1.0-SNAPSHOT.jar is.hi.cs.helloworld.App`

Note that this will add only your code to the jar, not any of the external dependencies (i.e. external jars).

Windows: backslash \

- **`mvn install`** copies your project to your local repo (= **`.m2`** folder).
  - So that you can refer in your other projects to your local project.

You would need some extra entries in **`pom.xml`** to add the needed **`Main-Class:`** entry to **`MANIFEST.MF`**

- Notes:
  - Maven assumes ordering of phases, i.e.: **`mvn test`** will also include **`mvn compile`** (but the compiler plugin will only do a (re-)compile if source files have changed).
    - E.g.: Doing **`mvn clean`** to delete generated bytecode, and then **`mvn test`** will include **`mvn compile`**.
    - You may use multiple arguments in a Maven run, e.g: **`mvn clean compile`**
  - Internally, each phase has registered a goal of a plugin to be executed.
    - **`mvn package`** (=phase) will finally call **`mvn jar:jar`** (=plugin:goal).

4.2.4 Advanced Maven
Adding Maven plugins to your POM

- There is a Maven plugin for everything!
  - (In fact, too many plugins to know all of them).
  - In practise, you just do a web search, but some of the most common plugins are listed on: https://maven.apache.org/plugins/index.html
  - Each goal provided by a plugin is internally implemented as a Java class,
    - i.e. a POJO=(Plain Old Java Object), but in the context of Maven, these are called MOJO.
  - No need to install Maven plugins manually, they are automatically retrieved from the Maven repositories.
- Adding non-standard Maven plugins to your **`pom.xml`** follows always this format:

```
<build>
  <plugins>
    <plugin>
      <groupId>groupId of plugin</groupId>
      <artifactId>artefactId of plugin</artifactId>
      <version>version</version>
      <configuration>
        <xml tags for all kinds of plugin-specific configuration that goes beyond convention>
      </configuration>
    </plugin>
  </plugins>
</build>
```

The **`<build></build>`** part follows after the **`<dependencies></dependencies>`** part.

As Maven plugins are dependencies of your automated build, they are referred to like any other dependency in Maven, i.e. using GAV (groupId, artifactId, version).

Adding Maven plugins to your POM:
Registering plugin goals to phases

- Possible to register a plugin goal to be automatically executed as part of a Maven phase (such as **`compile`**, **`test`**, **`package`**):

```
<build>
  <plugins>
    <plugin>
      <groupId>groupId of plugin</groupId>
      <artifactId>artefactId of plugin</artifactId>
      <version>version</version>
      <executions>
        <execution>
          <phase>some Maven phase</phase>
          <goals>
            <goal>one of the goals provided by this plugin</goal>
          </goals>
        </execution>
      </executions>
      <configuration>
        <xml tags for all kinds of plugin-specific configuration that goes beyond convention>
      </configuration>
    </plugin>
  </plugins>
</build>
```

This is just as on the previous slide.

Plugin will be automatically called in this phase.

Place here the name of the goal that your additional plugin offers.

This is just as on the previous slide.

Adding Maven plugins to your POM:
Example: Execute your compiled code

- E.g.: you are to lazy to start your project always by typing:
  `java -cp target/helloworld-1.0-SNAPSHOT.jar is.hi.cs.helloworld.App`
- Some web search reveals that the Exec Maven plugin
  https://www.mojohaus.org/exec-maven-plugin/
  can do the job and that you need to add to your pom.xml:

```
<build>
  <plugins>
    <plugin>
      <groupId>org.codehaus.mojo</groupId>
      <artifactId>exec-maven-plugin</artifactId>
      <!– FIXME: version missing here -->
      <configuration>
        <mainClass>is.hi.cs.helloworld.App</mainClass>
      </configuration>
    </plugin>
  </plugins>
</build>
```

In fact, **the plugin version is missing here: rather have a look at →slide 4-34 for a better example**.

The fully qualified name (e.g. prefixed by Java package name) of the class you want to execute.

If you want to pass in always the same command-line parameters to your programme, use as configuration of that plugin:

```
<configuration>
  <mainClass>is.hi.cs.helloworld.App</mainClass>
  <arguments>
    <argument>argument1</argument>
    <argument>argument2</argument>
  </arguments>
</configuration>
```

Calling **`exec:java`** will never trigger running the **`compile`** goal. But you can use: **`mvn compile exec:java`**

- and that you execute it via its plugin name and provided goal: **`mvn exec:java`**
  - or with extra parameters for your programme: **`mvn exec:java -Dexec.args="argument1 argument2"`**

Adding Maven plugins to your POM:
Example: Execute your compiled code – plugin versions matter!

- When you execute the example from the previous slide, the jar gets executed,
- but earlier, you might see a warning:

```
[WARNING] Some problems were encountered while building the effective model
[WARNING] 'build.plugins.plugin.version' for org.codehaus.mojo:exec-maven-plugin is missing.
[WARNING]
[WARNING] It is highly recommended to fix these problems because they threaten the stability of your build.
[WARNING]
[WARNING] For this reason, future Maven versions might no longer support building such malformed projects.
```

- The reason is that your POM does not specify for the plugin the version to be used.
  - Maven plugins have also a version.

- A future version of that plugin (automatically retrieved from a Maven repository) might behave differently and thus compromise the reproducibility of the build.

We noticed already on →slide 4-26 and 4-27 that the compiler plugin versions 3.1 and 3.11.0 behave differently.

- Therefore, you are supposed to add the plugin version in to your POM (→next slide).

Adding Maven plugins to your POM:
Example: Add version number of plugin to be used

- In the Maven output, you find the used version of each used plugin:

```
[INFO] --- exec-maven-plugin:3.0.0:java (default-cli) @ helloworld ---
```

- Insert that version number into your **`pom.xml`**:

```
<build>
  <plugins>
    <plugin>
      <groupId>org.codehaus.mojo</groupId>
      <artifactId>exec-maven-plugin</artifactId>
      <version>3.1.1</version>
      <configuration>
        <mainClass>is.hi.cs.helloworld.App</mainClass>
      </configuration>
    </plugin>
  </plugins>
</build>
```

The specification of each used plugin shall include a version number, i.e. use GAV everywhere.

`pluginMangement` is inside the **`build`** element (just as the **`plugin`** element is inside the **`build`** element). See also the POM generated by the latest archetype version where the version number for, e.g., the compile **`plugin`** is set. (Plugins such as compile are automatically used, so no need to add them in the **`build`** element, but you can set their version in the **`pluginMangement`**.)

- Alternatively, you can set the version number in the **`pluginManagement`** element.
  - Sets the version number for any references to that plugin in the **`plugin`** element.
    - Applies to any references to that plugin in the current POM and any child POMs.
  - Still, need to add this plugin again in the **`plugin`** element (but can omit version number there).

Versions matter always!

- In fact, the same version problem (i.e. builds depending on the particular Maven plugin versions involved in the build) also applies to the standard Maven plugins (archetype, compiler, etc.).
  - And therefore, the newer versions of the quickstart archetype template add exactly this info also for standard plugins in the `pom.xml`.
    - To prevent overwhelming you, →sl. 4-19 used on older archetype template version to create a simpler `pom.xml`.
      - But in principle, the newer archetype template versions are better.
        - E.g. `mvn site` (to generate HTML pages with documentation) fails with old archetype template version.

    - From now on, always use the latest archetype versions!
    - Also, specify the version of plugins for any plugin that you add on your own.

These will create a `pom.xml` file with more suitable plugin versions.

- By the way: not specifying the version of a normal dependency (i.e. a library such as JUnit) results in an error (in contrast to just a warning for plugins).
  - If versions are specified in a parent POM (→later slide), they are taken from there.
  - `mvn versions:display-dependency-updates` checks for available updates for your dependencies.
  - To let Maven re-write your `pom.xml` to make use of the most recent version of your dependencies:
    `mvn versions:use-latest-versions`

Compiler versions matter: Setting Java language version

- Setting Java language version for versions before Java 9:

```
<properties>
  <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
  <maven.compiler.source>7</maven.compiler.source>
  <maven.compiler.target>7</maven.compiler.target>
</properties>
```

- For Java 9 and later, recommended to use:

```
<properties>
  <maven.compiler.source>7</maven.compiler.source>
  <maven.compiler.target>7</maven.compiler.target>
  <maven.compiler.release>17</maven.compiler.release>
</properties>

<build>
  <plugins>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-compiler-plugin</artifactId>
      <version>3.12.1</version>
    </plugin>
  </plugins>
</build>
```

Does not only set source code version and target bytecode version, but also takes care that the runtime libraries that provide the Java API are used in the right version. If you use `release`, then you do not need to use `source` and `target` properties here.

According to documentation, the above `maven.compiler.release` project property is only understood by compiler plugins version 3.6.0 and later – without specifying a compiler plugin version, a too old compiler plugin might be used. (In fact, without explicitly setting the compiler plugin version, Maven ≤3.8.x uses version 3.1 of the compiler plugin).

As of 2/2024, the most recent compiler plugin version is 3.12.1 – While 3.11.0 would be sufficient, the latest version might have bugs fixed or is faster, so rather enforce latest version. How do you know which plugin versions exist? →Next slide!

Plugin versions

- To find out what are the latest available versions of the plugins that you use (and at which Maven versions, these plugin versions where introduced):
    - **`mvn versions:display-plugin-updates`**

Strictly speaking, to enhance the reproducibility of builds, the used version of Maven itself would need to be specified as well in the POM. In fact, this command complains exactly about it. But we can ignore it and assume that the minimum version is Maven 3.1.0.

If you want to enforce a minimum Maven version, use the **`maven-enforcer-plugin`**. For an example, see: https://maven.apache.org/enforcer/enforcer-rules/requireMavenVersion.html

Parent Super POM & Effective POM Debugging POM/Maven runs

- Maven has a built-in parent “Super POM”, which defines a set of defaults inherited by the POM of each project, e.g. the versions of plugins used.

  - For debugging your POM (e.g. to see what plugin versions are used), it is sometimes useful to see what the “effective POM” is, i.e. your POM and what it inherits from the Super POM:
    - `mvn help:effective-pom`

E.g. even if you do not specify plugin version, the parent POM sets them and you can see what version setting is effective.

- To get more verbose output when running Maven (e.g. for debugging your POM), add command line option **`–X`**, e.g.:
  - `mvn –X compile`

`<pluginManagement>`

- While you can define the plugin version in the **`<plugins> <plugins>`** section, the plugin version is often defined in the **`<pluginManagement> </pluginManagement>`** section.
  - You can then omit specifying the plugin version in the **`<plugins> <plugins>`** section.

- Difference: **`<plugins>`** vs. **`<pluginManagement>`**:
  - **`<pluginManagement>`** sets only the version (and other configurations that you add there) of a plugin, but does not actually add that plugin to your project.
  - Only by listing the plugin in the **`<plugins>`** section, you actually add that plugin to your project.
    - And the version information is then taken from the **`<pluginManagement>`** section (and other configurations that you added there).

- Why all that extra hassle and not just using only the **`<plugins>`** section?
  - You can create skeleton POMs where you set the version number, but leave it to later to list the plugins that are actually used.

    The newer archetype plugins generate exactly such POMs.
  - You can inherit the **`<pluginManagement>`** to sub POMs, so that everywhere the same version of a particular plugin is used.

<pluginManagement> Example

Do not add new plugins only to **pluginManagment** – rather add them to the **plugins** section at the bottom.

```
<build>
   <pluginManagement><!-- lock down plugins versions to avoid using Maven defaults -->
     <plugins>
       <plugin>
         <groupId>org.codehaus.mojo</groupId>
         <artifactId>exec-maven-plugin</artifactId>
         <version>3.1.1</version>
       </plugin>
     </plugins>
   </pluginManagement>

   <plugins>
     <plugin>
       <groupId>org.codehaus.mojo</groupId>
       <artifactId>exec-maven-plugin</artifactId>
       <configuration>
         <mainClass>is.hi.cs.helloworld.App</mainClass>
       </configuration>
     </plugin>
   </plugins>
 </build>
```

The versions in this plugin management section are, e.g., created by the archetype.

You add then this plugin section following the plugin management section.
No need to mention the plugin version here again in this plugin section are, because they have already been defined in the plugin management section.

Have only one `<dependencies>` section, only one `<plugins>` section, only one `<pluginManagement>` section

- Maven does not allow to have in the pom.xml more than one section **`<dependencies>`**, **`<pluginManagement>`**, **`<plugins>`**, etc.
  - Rather have then in the same **`<dependencies>`** section multiple **`<dependency>`** entries, in **`<plugins>`** multiple **`<plugin>`** entries, etc.

```
<dependencies>
    <dependency>
        <groupId>junit</groupId>
        <artifactId>junit</artifactId>
        <version>4.13.2</version>
        <scope>test</scope>
    </dependency>
</dependencies>
</dependencies>
    <dependency>
        <groupId>org.apache.commons</groupId>
        <artifactId>commons-primes</artifactId>
        <version>1.1</version>
    </dependency>
</dependencies>
```

```
<dependencies>
    <dependency>
        <groupId>junit</groupId>
        <artifactId>junit</artifactId>
        <version>4.13.2</version>
        <scope>test</scope>
    </dependency>
    <dependency>
        <groupId>org.apache.commons</groupId>
        <artifactId>commons-primes</artifactId>
        <version>1.1</version>
    </dependency>
</dependencies>
```

Copying JARs of dependencies into the target directory

- Sometimes, you want to have the jar files of all your dependencies.
  - E.g. to deliver them together with your code to some customer or to deploy it on your server.
    - (Or for Assignment 4.)

- While you could find these jars somewhere deep in the **`.m2`** directory where Maven caches all the dependencies ($\rightarrow$slide 4-25), there is a more convenient method:

- **`mvn install dependency:copy-dependencies`**
  - Copies all the jars of your dependencies specified in your POM to: **`target/dependency`**

4.2.5 Maven and IDEs

Have a look at the web pages linked below (& the Panopto video) to get an idea how your IDE supports Maven.

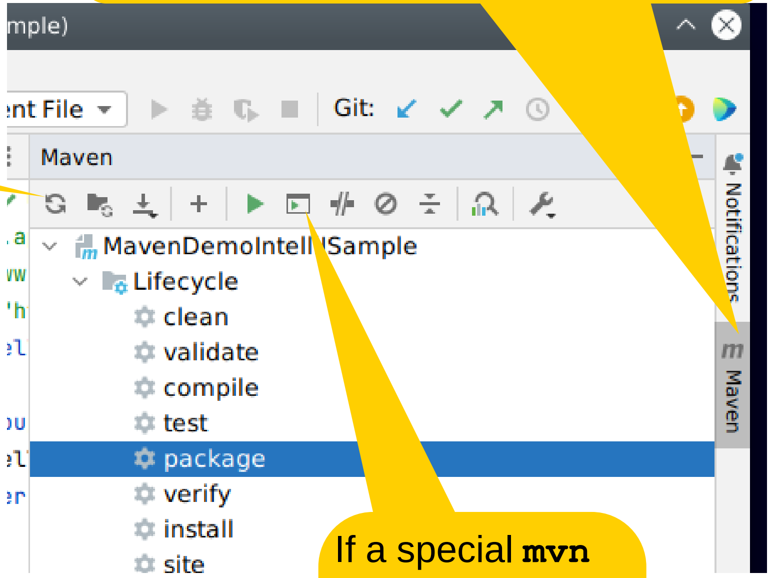
- Eclipse:
  - https://www.vogella.com/tutorials/EclipseMaven/article.html
- IntelliJ:
  - https://www.jetbrains.com/help/idea/maven-support.html
    - Section: "Maven projects"
    - Section: "Maven goals"
    - Section: "Maven dependencies"
  - To create from archetype: File → New → Project… → Maven Archetype

Take care that in the dialog, "Catalog" is not set to "Internal", but to "Maven Central" (to see the latest versions).

- Visual Studio Code
  - https://code.visualstudio.com/docs/java/java-build
    - Section: "Maven"
  - To create from archetype: CTRL-SHIFT P, then type **`create java project`**

You need to click this reload icon after editing your **`pom.xml`** file.

IntelliJ has a Maven tool window hidden here (or hit twice the Ctrl key). Or run Maven in the IntelliJ terminal.



If a special **`mvn`** command is not in the "Lifecycle" list, you can enter vit his icon manually **`mvn`** commands.

4.3 Summary

- Build management: automate your build and make it reproducible.
- Maven:
  - Build tool for Java (and other languages that compile to Java bytecode) projects.
  - “Convention over configuration”
    - As long as your project fits into the convention, the defaults do a good job.
    - But complex behaviour may lead to long, hard to read XML code.
      - Use XML comments.
  - There is a Maven plugin for everything.
  - Automatic dependency download is very convenient.

Common trouble makers

Everything looks correct, still does not work?
Did you copy/paste hidden characters from the slides?
Delete and type on your own.

- Copying examples from the slides → sl. 4-20.
  - May contain hidden characters that your editor does not show, but your tool gets confused by it.
- Using PowerShell → sl. 4-16 & 4-20.
  - PowerShell eats the minus sign/hyphen.
- Using semicolon as path separator with POSIX shell (Bash or zsh) → Chapter 3, sl. 3-22
  - Need to surround the parameter value that contains the semicolon by double quotes.
- Maven-specific:
  - Having Maven not installed or environment variable **`PATH`** not containing the bin directory of Maven. →4-17
  - Adding a plugin (e.g. **`exec-maven-plugin`**) to **`pluginManagement`**, instead to **`plugins`** (in parallel to **`pluginManagement`**. → sl. 4-40
  - **`maven.compiler.release`** property not set to same version as **`source`** and **`target`**. → sl. 4-28
    - If the IDE uses a newer JDK version and compiles in the background, then **`exec-maven-plugin`** will not be able to run the newer bytecode compiled by the IDE.
  - Having more than only one section **`<dependencies>`**, **`<plugins>`**, etc. → sl. 4-41
    - E.g. if you have multiple **`<dependency>`** entries, they all need to be inside the same **`<dependencies>`** section.
  - Tag names/section names are case-sensitive → sl. 4-22
    - E.g. needs to be **`<pluginId>`**, not **`<pluginid>`**.