Apache Maven

Péter Jeszenszky
Faculty of Informatics, University of Debrecen
jeszenszky.peter@inf.unideb.hu

Last modified: March 8, 2018

Apache Maven

- Software project management and comprehension tool that provides support for
  - Building projects in a uniform way
  - Publishing project information on the web
  - Release management
  - Distribution of project artifacts
  - Dependency management
  - …

Features (1)

- Convention over configuration.
    - E.g., defines a standard project directory layout.
- Defines project lifecycles and lifecycle phases.
- Declarative.
- Modular and extensible architecture.
    - All of its functionality is provided by plugins.

Features (2)

- Although it is used mainly for Java projects, can be also used for other programming languages, such as:
  - C/C++:
    - nar-maven-plugin http://maven-nar.github.io/
  - Kotlin:
    - kotlin-maven-plugin https://kotlinlang.org/docs/reference/using-maven.html
  - Scala:
    - scala-maven-plugin http://davidb.github.io/scala-maven-plugin/

Development

- Written in Java.
- Free and open source software.
  - Distributed under the Apache License v2.
- The current stable version is 3.5.3 (released on March 8, 2018). https://maven.apache.org/docs/history.html
  - Differences between versions 2.*x* and 3.*x*.

Installation

- Requires a JDK to be installed, not just a JRE!
  - JDK 7 or later is required.
    - Download the latest Oracle JDK from here: http://www.oracle.com/technetwork/java/javase/downloads/
  - The `JAVA_HOME` environment variable must be set properly!
- Download the latest version of Apache Maven from here: http://maven.apache.org/download.html
- Installation consists of unpacking an archive and setting the PATH environment variable.

Installation (Linux) (1)

- If unpacked in the directory `/opt/apache-maven-3.5.3`, set the `PATH` environment variable as shown:
  - `export PATH=/opt/apache-maven-3.5.3/bin:$PATH`
- Hint: add the above command to the `/etc/profile.d/maven.sh` file in order to be automatically executed.

Installation (Linux) (2)

- Alternatively, Apache Maven can be installed via the SDKMAN! tool, executing the following command:

```
sdk install maven
```

- For the installation instructions of SDKMAN! see: http://sdkman.io/install.html

Installation (Windows)

- If unpacked in the directory `C:\Program Files\apache-maven-3.5.3`
  - Add the `C:\Program Files\apache-maven-3.5.3\bin` directory to the `PATH` environment variable.

Verifying Successful Installation

- Execute one of the following two equivalent commands in the command line:

```
mvn --version
mvn -v
```

- If the installation was successful, you should see the following output:

```
Apache Maven 3.5.3 (3383c37e1f9e9b3bc3df5050c29c8aff9f295297;
    2018-02-24T20:49:05+01:00)
Maven home: /opt/apache-maven-3.5.3
Java version: 1.8.0_161, vendor: Oracle Corporation
Java home: /home/jeszy/.sdkman/candidates/java/8u161-oracle/jre
Default locale: hu_HU, platform encoding: UTF-8
OS name: "linux", version: "3.13.0-37-generic", arch: "amd64", family: "unix"
```

IDE Integration

- **Eclipse**: m2eclipse http://eclipse.org/m2e/
  - Update site: http://download.eclipse.org/technology/m2e/releases/
  - The Indigo release of the Eclipse IDE for Java Developers is shipped together with m2eclipse (no installation is required).
- **IntelliJ IDEA**: Provides built-in Apache Maven support. https://www.jetbrains.com/help/idea/maven.html
- **Netbeans**: Provides built-in Apache Maven support since version 6.7. http://wiki.netbeans.org/Maven

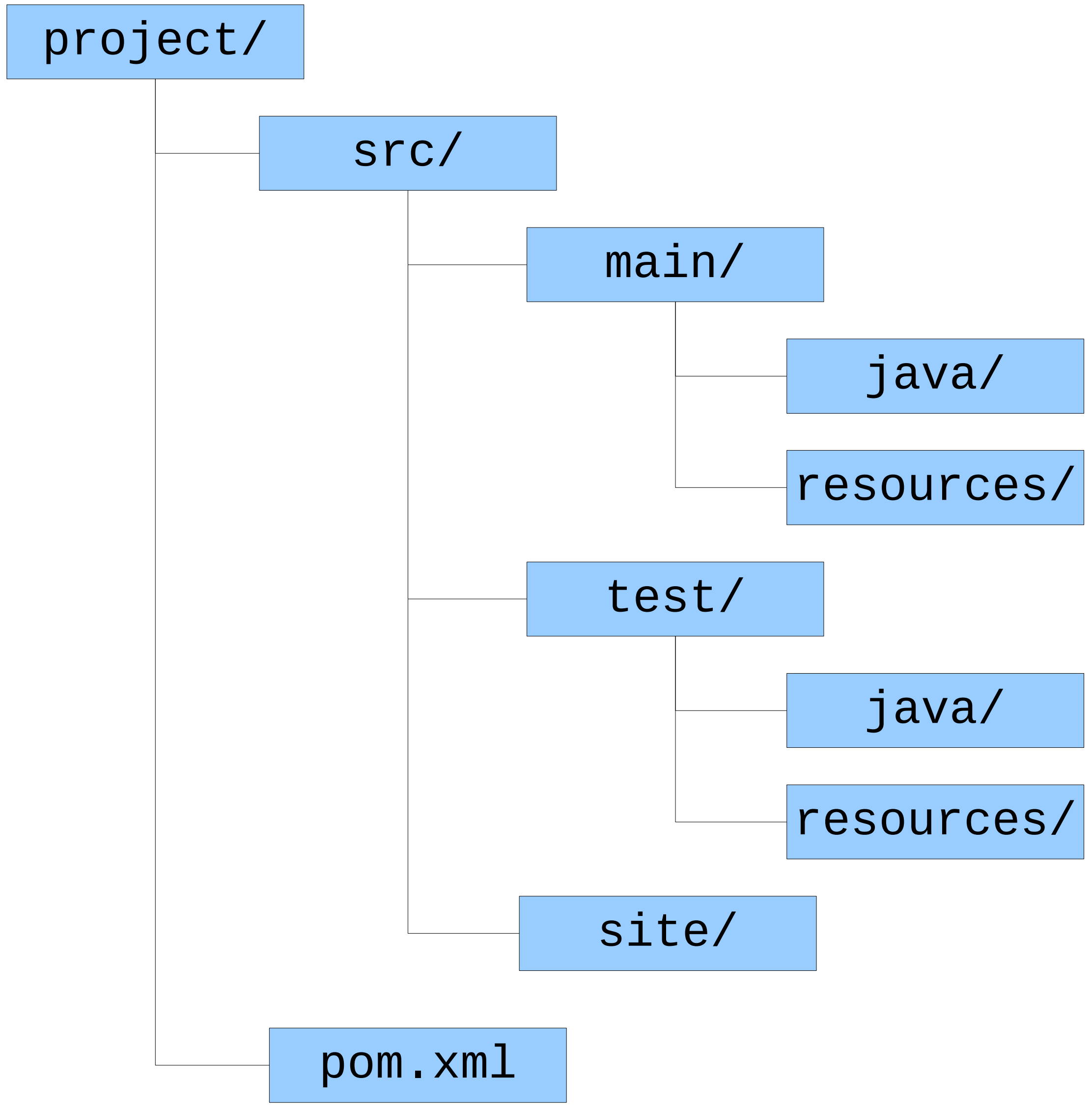
Further Information

- Free electronic books distributed under a Creative Commons License. http://www.sonatype.org/nexus/resources/resources-book-links-and-downloads/
  - *Maven by Example*
  - *Maven: The Complete Reference*
  - *Repository Management with Nexus*
- Mailing lists: http://maven.apache.org/mail-lists.html

Standard Directory Layout (1)

- Standard directory layout for projects:
  - *Introduction to the Standard Directory Layout* http://maven.apache.org/guides/introduction/introduction-to-the-standard-directory-layout.html

Standard Directory Layout (2)



Usage (1)

- Execute the `mvn --help` or `mvn -h` command to display usage information and available command line options.
- Apache Maven accepts lifecycle phases (e.g., `mvn package`) and plugin goals (e.g., `mvn site:run`) as command line arguments.
  - Multiple number of such command line arguments can be supplied.
  - Further options can be passed as system properties in the form `-D`*name*=*value*

Usage (2)

- Plugin goals can be specified in the form *groupId*`:`*artifactId*`:`*version*`:`*goal*
  - This syntax is required when a specific version of the plugin must be used or Apache Maven does not associate a prefix with the plugin.
    - Example: `org.apache.maven.plugins:maven-archetype-plugin:3.0.1:generate`

Maven Help Plugin (1)

- Intended for querying information about plugins, the runtime environment and the project.
  - Further information: http://maven.apache.org/plugins/maven-help-plugin/
- Use the `mvn help:help` command to display usage information.
  - Add the `-Ddetail=true` option for more detailed information.
  - To display help for a specific goal add the `-Dgoal=`*goal* option.
    - Example: `mvn help:help -Dgoal=describe -Ddetail=true`

Maven Help Plugin (2)

- The `describe` goal of the plugin provides information about plugins and lifecycle phases.
  - Examples of use:
    - `mvn help:describe -Dplugin=org.apache.maven.plugins:maven-jar-plugin:3.0.2`
    - `mvn help:describe -Dplugin=jar`
    - `mvn help:describe -Dplugin=jar -Dgoal=sign`
    - `mvn help:describe -Dplugin=jar -Dgoal=sign -Ddetail=true`
    - `mvn help:describe -Dcmd=jar:jar -Ddetail=true`
    - `mvn help:describe -Dcmd=clean`

Maven Help Plugin (3)

- The `effective-pom` goal displays the effective POM.
  - Execution of the `mvn help:effective-pom` command requires a `pom.xml` file to be present in the current directory.
    - If the file is not there, its path can be specified with the `-f` or `--file` option, e.g., `mvn help:effective-pom -f ../project/pom.xml`

Maven Archetype Plugin (1)

- Supports the creation of Maven projects from an existing template called an archetype.
  - Projects can be created interactively with the command `mvn archetype:generate`
    - An archetype must be selected, then the Maven coordinates and the packaging must be provided.
- Further information: http://maven.apache.org/archetype/maven-archetype-plugin/

Maven Archetype Plugin (2)

- Since the 2.1 version of the plugin the list of available archetypes can be narrowed down with a filter pattern.
  - The filter pattern can be specified with the `-Dfilter=`*pattern* option
    - Example of use: `mvn archetype:generate -Dfilter=android`

Maven Archetype Plugin (3)

- A few useful archetypes:
  - `org.apache.maven.archetypes:maven-archetype-plugin`
  - `org.apache.maven.archetypes:maven-archetype-site`
  - `com.zenjava:javafx-basic-archetype`
  - `org.wildfly.archetype:wildfly-javaee7-webapp-archetype`
  - …

settings.xml (1)

- A configuration file that stores project independent settings.
  - The `$M2_HOME/conf/settings.xml` file provides global setting for all users.
  - Users can place an own `settings.xml` file in the `.m2` directory of their home directory to override global settings.
    - On Linux systems the path of this file is `~/.m2/settings.xml`
- XML schema: http://maven.apache.org/xsd/settings-1.0.0.xsd

settings.xml (2)

- The `effective-settings` goal of the Maven Help Plugin displays the current configuration settings.
  - The `mvn help:effective-settings` command outputs a combination of global and user settings that Maven actually uses.
- Hint: use the global `settings.xml` file as a template to create a local one.
  - On Linux systems copy the global `settings.xml` to your home directory with the following command:

```
cp $M2_HOME/conf/settings.xml ~/.m2
```

Fundamental Concepts

- Artifact
- Project Object Model (POM)
- Super POM
- Effective POM
- Maven coordinates
- Plugin, plugin goal
- Remote/local repository
- Lifecycle, lifecycle phase

Artifact

- An artifact is a file that can be considered as the final product of a project.
  - Usually, the product of a project is a single artifact (e.g, a single JAR file in a project with `jar` packaging).
    - Use the `classifier` Maven coordinate when multiple artifacts must be produced.
  - Artifacts are published in repositories that allows them to be used as dependencies for other projects.

Project Object Model (POM)

- An XML document (`pom.xml`) that contains a declarative description of the project.
  - Contains metadata and configuration settings.
- XML schema: http://maven.apache.org/xsd/maven-4.0.0.xsd
- Parent-child relationship can be defined among projects.
  - A child project inherits settings from the POM of its parent that it can override.

Minimal POM

```xml
<project xmlns="http://maven.apache.org/POM/4.0.0">
  <modelVersion>4.0.0</modelVersion>
  <groupId>hu.unideb.inf.maven</groupId>
  <artifactId>maven-hello</artifactId>
  <version>1.0</version>
</project>
```

Super POM

- If a project does not have an explicitly specified parent, it inherits from the super POM.
  - It is the default POM used by Maven that provides defaults for a number of configuration settings.
- For the `3.`*x*`.`*y* versions, it can be found in the `pom-4.0.0.xml` file in the `maven-model-builder-3.`*`x`*`.`*`y`*`.jar` file under the `lib/` directory of the Maven installation.

Effective POM

- The combination of the projects own POM, the POMs of its ancestors and the super POM.
  - Provides the settings actually used for running Maven.
- Can be displayed with the command `mvn help:effective-pom`

Maven Coordinates (1)

- Each artifact is identified with its Maven coordinates that consists of 3 or 4 components.
  - Required components:
    - **`groupId`**: group identifier, that is often a reverse domain name (e.g., `org.apache.maven.plugins`), but not always (e.g., `junit`)
    - **`artifactId`**: the name of the project (e.g., `maven-assembly-plugin`, `junit`)
    - **`version`**: version number (e.g., `1.0`, `1.0-SNAPSHOT`)
  - Optional component:
    - **`classifier`** (see later)

Maven Coordinates (2)

- The `groupId`, `artifactId` and `version` elements of the POM specify the coordinates of the artifacts produced in the project.
  - If a project has an explicitly specified parent then it inherits the Maven coordinates from the parent.
    - In such cases, it is typical to retain the `groupId` and `version` of the the parent and to override the `artifactId.`
- Maven coordinates are often written in the form *groupId*`:`*artifactId*`:`*version* (e.g., `junit:junit:4.12`).

Maven Coordinates (3)

- Can be used to reference artifacts as dependencies, such as

```
<dependency>
  <groupId>junit</groupId>
  <artifactId>junit</artifactId>
  <version>4.12</version>
  <scope>test</scope>
</dependency>
```

Packaging

- The packaging of the project can be specified in the `packaging` element, the allowed values are:
  - `pom`
  - `jar` (default)
  - `maven-plugin`
  - `ejb`
  - `war`
  - `ear`
  - `rar`
  - `par`

Plugins (1)

- Maven uses plugins to perform tasks.
  - Plugins provide goals, each of which has a specific functionality.
- Plugins are artifacts that can be referenced by their Maven coordinates.
  - Example of referencing a plugin in the POM:

```
<plugin>
   <groupId>org.apache.maven.plugins</groupId>
   <artifactId>maven-site-plugin</artifactId>
   <version>3.7</version>
</plugin>
```

- Each plugin has a prefix that allows users to reference plugin goals in the form *prefix*`:`*goal*, e.g., `site:site`.

Plugins (2)

- Naming convention:
  - The `artifactId` of official maven plugins maintained by the Apache Maven team follows the pattern `maven-`*xyz*`-plugin`, where *xyz* is the prefix associated with the plugin.
    - Other plugins must not follow this naming pattern.
  - For plugins from other sources *xyz*`-maven-plugin` is the recommended form, where *xyz* is the prefix associated with the plugin.
- Prefixes are specified by the plugins in their `plugin.xml` file.

Plugins (3)

- By default, a plugin goal can be referenced via a prefix if the plugin belongs to either the `org.apache.maven.plugins` or the `org.codehaus.mojo` group.
  - See the `maven-metadata-central.xml` files in the `$HOME/.m2/repository/org/apache/maven/plugins` and `$HOME/.m2/repository/org/codehaus/mojo` directories.
  - For more information, see: *Introduction to Plugin Prefix Resolution* http://maven.apache.org/guides/introduction/introduction-to-plugin-prefix-mapping.html

Repositories (1)

- Artifacts, including plugins, are published in repositories.
  - Remote repositories can be accessed on the web via HTTP or HTTPS.
    - Central Repository: https://repo.maven.apache.org/maven2
  - A local repository stores artifacts downloaded from remote repositories in the local file system and artifacts installed with the `mvn install` command.
    - Functions as a cache.
    - Resides in the HOME directory of the user (on Linux systems it can be found in the `~/.m2/repository/` directory).
- Remote and local repositories share the same layout.

Repositories (2)

- Repositories map Maven coordinates to a directory structure.
  - Example: `org.apache.maven.plugins` → `/org/apache/maven/plugins/`
    - The remaining compontents of the directory path correspond to the `artifactId` and `version` part of the Maven coordinates (e.g., `junit:junit:4.12` → `/junit/junit/4.12/`).
- Maven 3.*x* can use separate repositories for dependencies and plugins.

Repositories (3)

- Repository managemenet software:
  - Free and open source software:
    - Apache Archiva (Apache License v2) http://archiva.apache.org/
    - Artifactory Open Source (GNU GPL v3) http://www.jfrog.com/open-source/
    - Nexus OSS (Eclipse Public License v1.0) https://www.sonatype.com/download-oss-sonatype
  - Non-free software:
    - Artifactory http://www.jfrog.com/
    - Nexus Repository Pro https://www.sonatype.com/nexus-repository-sonatype

Lifecycles

- A lifecycle consists of a sequence of lifecycle phases.
  - Each phase is identified by a unique name.
  - Plugin goals can be assigned to phases, such an assigment is called a binding.
- Executing a lifecycle phase results in the execution of the plugin goals bound to it.
  - Execution of a phase results in the execution of each phases that precede it in the lifecycle.
- The three standard lifecycles: `clean`, `default`, `site`
  - By default, specific plugin goals are bound to lifecycle phases depending on the packaging of the project.

Lifecycles: the clean lifecycle

- The `clean` lifecycle consists of the following three lifecycle phases:

  (1) `pre-clean`

  (2) `clean`

  (3) `post-clean`

- By default, the `clean:clean` plugin goal is bound to the `clean` phase.
  - Execution of the plugin goal will delete the files generated by Maven in the project's working directory.
- For more information, see: *Lifecycles Reference* http://maven.apache.org/ref/current/maven-core/lifecycles.html

Lifecycles: the site lifecycle

- The `site` lifecycle consists of the following four lifecycle phases:

  ```
  (1) pre-site
  (2) site
  (3) post-site
  (4) site-deploy
  ```

- By default, the `site:site` plugin goal is bound to the `site` phase, and the `site:deploy` plugin goal is bound to the `site-deploy` phase.

- For more information, see: *Lifecycles Reference* http://maven.apache.org/ref/current/maven-core/lifecycles.html

Lifecycles: the default lifecycle (1)

```
(1)  validate
(2)  initialize
(3)  generate-sources
(4)  process-sources
(5)  generate-resources
(6)  process-resources
(7)  compile
(8)  process-classes
(9)  generate-test-sources
(10) process-test-sources
(11) generate-test-resources
(12) process-test-resources
(13) test-compile
(14) process-test-classes
(15) test
(16) prepare-package
(17) package
(18) pre-integration-test
(19) integration-test
(20) post-integration-test
(21) verify
(22) install
(23) deploy
```

Lifecycles: the default lifecycle (2)

- Default bindings for the `ejb`, `jar`, `rar`, and `war` packagings:
  - See: *Plugin Bindings for default Lifecycle Reference* http://maven.apache.org/ref/current/maven-core/default-bindings.html

`process-resources`	`resources:resources`
`compile`	`compiler:compile`
`process-test-resources`	`resources:testResources`
`test-compile`	`compiler:testCompile`
`test`	`surefire:test`
`package`	`ejb:ejb / jar:jar / rar:rar / war:war`
`install`	`install:install`
`deploy`	`deploy:deploy`

Property References (1)

- References of the form `${`*`x`*`}` are substituted in the POM.
  - References of the form `${env.`*`name`*`}` are substituted with the value of the named environment variable.
    - For example, `${env.PATH}` results in the value of `PATH` environment variable.
  - Java system properties can be named in the references.
    - For example, `${java.home}`, `${line.separator}`
  - References of the form `${project.`*`x`*`}` are substituted with the content of the corresponding POM element (only works with simple type elements).
    - For example, `${project.groupId}`, `${project.artifactId}`, `${project.url}`, `${project.build.outputDirectory}`
  - References of the form `${settings.`*`x`*`}` are substituted with the content of the corresponding element in the `settings.xml` file.

Property References (2)

- Properties specified in the `properties` element can be referenced using the syntax.
  - Example:

```
<properties>
  <company.name>unideb</company.name>
</properties>
...
${company.name}
```

POM Reference (1)

- `artifactId`: provides the `artifactId` component of the Maven coordinates

```
<artifactId>commons-lang3</artifactId>
```

- `build`: contains the build settings of the project

```
<build>
    <directory>${project.basedir}/target</directory>
    <outputDirectory>
        ${project.build.directory}/classes
    </outputDirectory>
    <finalName>
        ${project.artifactId}-${project.version}
    </finalName>
    ...
</build>
```

- `ciManagement`: provides information about the continuous integration system used by the project

```
<ciManagement>
    <system>continuum</system>
    <url>http://vmbuild.apache.org/</url>
</ciManagement>
```

POM Reference (2)

- `contributors`: describes the (non-developer) contributors of the project

```
<contributors>
   <contributor>
      <name>Naoki Nose</name>
      <email>ikkoan@mail.goo.ne.jp</email>
      <roles>
         <role>Japanese translator</role>
      </roles>
   </contributor>
   ...
</contributors>
```

POM Reference (3)

- `dependencies`: lists the dependencies of the project

```
<dependencies>
    <dependency>
        <groupId>org.slf4j</groupId>
        <artifactId>slf4j-jdk14</artifactId>
        <version>1.7.25</version>
        <scope>runtime</scope>
    </dependency>
    ...
</dependencies>
```

- `dependencyManagement`: provides default dependency information for projects that inherit from this one

```
<dependencyManagement>
    <dependencies>
        <dependency>
            <groupId>org.slf4j</groupId>
            <artifactId>slf4j-api</artifactId>
            <version>1.7.25</version>
        </dependency>
        ...
    </dependencies>
</dependencyManagement>
```

POM Reference (4)

- `description`: contains the description of the project

```
 <description>JUnit is a regression testing
    framework. It is used by the developer who
    implements unit tests in Java.
</description>
```

- `developers`: provides information about the developers of the projects

```
<developers>
   <developer>
      <id>jeszy</id>
      <name>Péter Jeszenszky</name>
      <email>jeszenszky.peter@inf.unideb.hu</email>
      <url>http://www.inf.unideb.hu/~jeszy/</url>
      <roles>
         <role>developer</role>
      </roles>
   </developer>
   ...
</developers>
```

POM Reference (5)

- `distributionManagement`: provides information about the remote web server and repository to which the Maven-generated site and the project's artifacts are deployed, respectively

```
   <distributionManagement>
      <site>
         <id>morse</id>
<url>scp://jeszy@arato.inf.unideb.hu/home/jeszenszky.peter/p
ublic_html/cli/</url>
      </site>
   </distributionManagement>
```

POM Reference (6)

- `groupId`: provides the `groupId` component of the Maven coordinates

```
<groupId>org.apache.maven.plugins</groupId>
```

- `inceptionYear`: holds the year of the project's inception

```
<inceptionYear>2010</inceptionYear>
```

- `issueManagement`: provides information about the issue management system used by the project

```
<issueManagement>
  <system>Bugzilla</system>
  <url>https://issues.apache.org/bugzilla/</url>
</issueManagement>
```

POM Reference (7)

- `licenses`: describes the licenses of the project

```
    <licenses>
        <license>
            <name>GNU General Public License v3.0</name>
            <url>http://www.gnu.org/copyleft/gpl.html</url>
        </license>
        ...
    </licenses>
```

- `mailingLists`: provides information about the mailing lists of the project

```
    <mailingLists>
        <mailingList>
            <name>User Mailing List</name>
            <subscribe>user-subscribe@tika.apache.org</subscribe>
            <unsubscribe>user-unsubscribe@tika.apache.org</unsubscribe>
            <post>user@tika.apache.org</post>
            <archive>http://mail-archives.apache.org/mod_mbox/tika-
user/</archive>
        </mailingList>
        ...
    </mailingLists>
```

POM Reference (8)

- `modules`: for multimodule projects lists the relative paths of directories containing the modules

```
<modules>
   <module>client</module>
   <module>library</module>
</modules>
```

- `name`: the conversational name of the project

```
<name>JUnit</name>
```

- `organization`: provides information about the organization that runs the project

```
 <organization>
   <name>Faculty of Informatics, University of
      Debrecen</name>
   <url>http://www.inf.unideb.hu/</url>
</organization>
```

POM Reference (9)

- `packaging`: determines the type of artifact produced in the project

```
<packaging>jar</packaging>
```

- `parent`: contains the Maven coordinates of the parent

```
<parent>
    <groupId>org.slf4j</groupId>
    <artifactId>slf4j-parent</artifactId>
    <version>1.7.25</version>
</parent>
```

- `pluginRepositories`: lists the remote repositories to look for plugins

```
<pluginRepositories>
    <pluginRepository>
        <id>central</id>
        <name>Central Repository</name>
        <url>https://repo.maven.apache.org/maven2</url>
        <snapshots>
            <enabled>false</enabled>
        </snapshots>
    </pluginRepository>
    ...
</pluginRepositories>
```

POM Reference (10)

- `prerequisites`: describes the prerequisites for building the project (currently, only the minimum required version of Maven can be specified)

```
<prerequisites>
   <maven>3.0</maven>
</prerequisites>
```

- `profiles`: specifies build profiles (see later)
- `properties`: specifies Maven properties, i.e., key-value pairs

```
<properties>
   <project.build.sourceEncoding>UTF-8
      </project.build.sourceEncoding>
   <project.reporting.outputEncoding>UTF-8
      </project.reporting.outputEncoding>
   ...
</properties>
```

POM Reference (11)

- `reporting`: lists and configures the reporting plugins to use to generate reports to be included on the Maven-generated site

```
   <reporting>
      <plugins>
         <plugin>
      <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-javadoc-
plugin</artifactId>
            <version>3.0.0</version>
         </plugin>
         ...
      </plugins>
   </reporting>
```

POM Reference (12)

- `repositories`: lists the remote repositories to look for dependencies

```
<repositories>
    <repository>
        <id>maven-restlet</id>
        <name>Restlet repository</name>
        <url>http://maven.restlet.com/</url>
    </repository>
    ...
</repositories>
```

POM Reference (13)

- `scm`: provides information for accessing the version control system used by the project

```
<scm>
   <connection>
      scm:git:http://git-wip-us.apache.org/repos/asf/wicket.git
   </connection>
   <developerConnection>
      scm:git:https://git-wip-
us.apache.org/repos/asf/wicket.git
   </developerConnection>
   <url>http://git-wip-us.apache.org/repos/asf/wicket/repo?
p=wicket.git</url>
   <tag>wicket-7.2.0</tag>
</scm>
```

- `url`: contains the URL of the project's homepage

```
<url>http://wicket.apache.org</url>
```

- `version`: contains the version number component of the Maven coordinates

```
<version>3.0</version>
```

Dependency Management

- Maven uses the Maven Artifact Resolver library for dependency management. https://maven.apache.org/resolver/

Specifying Dependencies (1)

```
<dependencies>
   <dependency>
      <groupId>groupId</groupId>
      <artifactId>artifactId</artifactId>
      <version>version</version>
      <classifier>classifier</classifier>
      <type>type</type>
      <optional>false|true</optional>
      <scope>compile|provided|runtime|system|test</scope>
      <systemPath>path</systemPath>
      <exclusions>
         <exclusion>
            <groupId>groupId</groupId>
            <artifactId>artifactId</artifactId>
         </exclusion>
         ...
      </exclusions>
   </dependency>
   ...
</dependencies>
```

Specifying Dependencies (2)

- `groupId`, `artifactId`, `version`, `classifier`: provide the Maven coordinates of the dependency
- `type`: provides the packaging type of the dependency (default: `jar`)
- `optional`: specifies whether the dependency is optional (default: `false`)

Specifying Dependencies (3)

- `scope`: provides the scope of the dependency
  - Determines in which classpaths the dependency is available, and also limits the transitivity of the dependency, valid values are the following:
    - **`compile`**: the dependency will be available in all classpaths, furthermore, it will be inherited by dependent projects (this is the default)
    - **`provided`**: the dependency is provided by the JDK or a container at runtime, it is only available on the compilation and test classpath, and is not transitive
    - **`runtime`**: the dependency is required only for execution (including the execution of tests)
    - **`system`**: the dependency is not looked up in a repository, it is available in the local file system
    - **`test`**: the dependency is required only for the compilation and execution of tests

Specifying Dependencies (4)

- `systemPath`: allowed and required for `system` scope dependencies
  - Contains the absolute path of the dependency, for example:

```
<systemPath>${java.home}/lib/jfxrt.jar</systemPath>
```

- `exclusions`: lists the artifacts not to be inherited from the dependency when calculating transitive dependencies

Specifying Dependencies (5)

- Example for a `system` scope dependency: using JavaFX with JDK 7

```
<dependency>
   <groupId>com.oracle</groupId>
   <artifactId>javafx</artifactId>
   <version>2.2</version>
   <scope>system</scope>
   <systemPath>${java.home}/lib/jfxrt.jar</systemPath>
</dependency>
```

Version Numbers (1)

- Version numbers has the form *p*`.`*q*`.`*r*`-`*s*, where
  - *p* is the major version
  - *q* is the minor version
  - *r* is the incremental version
  - *s* is a build number or a qualifier
- Valid qualifiers: `alpha`/`a`, `beta`/`b`, `milestone`/`m`, `rc`/`cr`, `snapshot`, ‹*empty string*›/`final`/`ga`, `sp`
  - Enumerated in ascending order.

Version Numbers (2)

- Some examples of version numbers: `1.2`, `4.8.2`, `1.6.0-alpha2`, `1.0-beta9`
- An ordering is defined on the set of version numbers.
  - Sorting is done by comparing components from left to right.
    - Components that consist of digits only are sorted numerically.
  - For example:
    - `1.0` < `1.5` < `1.10` < `1.10.1` < `2.0`
    - `1.0-alpha1` < `1.0-beta1` < `1.0-beta2` < `1.0-rc1` < `1.0` < `1.0-sp1`

Version Numbers (3)

- Use the following command to compare version numbers:
  - Linux: `java -jar $M2_HOME/lib/maven-artifact-*.jar`
  - Windows: `java -jar %M2_HOME%\lib\maven-artifact-*.jar`
- The program takes two version numbers as command line arguments.

Version Numbers (4)

- See: `org.apache.maven.artifact.versioning.ComparableVersion`
  - https://maven.apache.org/ref/3.5.3/maven-artifact/apidocs/org/apache/maven/artifact/versioning/ComparableVersion.html
  - https://github.com/apache/maven/blob/master/maven-artifact/src/main/java/org/apache/maven/artifact/versioning/ComparableVersion.java

Dependency Version Requirements (1)

- When specifying dependencies version ranges can be used instead of a single version number in the `version` element.
  - Each of the following forms are valid: `(`*a*`,`*b*`)`, `(`*a*`,`*b*`]`, `[`*a*`,`*b*`)`, `[`*a*`,`*b*`]`
    - Parentheses and brackets denote open, half-closed and closed intervals.
    - Both the lower and the upper bounds can be omitted.
      - In this case, their default values are negative and positive infinity, respectively.
  - A comma separated list of ranges is also accepted (means the union of the ranges).
    - For example: `(,1.0),(1.0,)`

Dependency Version Requirements (2)

- For example, the following states that any version of JUnit with version number *v* is acceptable to which `3.8` ≤ *v* < `4.0` holds:

```
<dependency>
  <groupId>junit</groupId>
  <artifactId>junit</artifactId>
  <version>[3.8,4.0)</version>
  <scope>test</scope>
</dependency>
```

Dependency Version Requirements (3)

- If the `version` element of a dependency contains a single version number, then it is considered as a recommendation only.
  - To force a specific version use the following:

```
<dependency>
  <groupId>junit</groupId>
  <artifactId>junit</artifactId>
  <version>[4.12]</version>
  <scope>test</scope>
</dependency>
```

Transitive Dependencies (1)

- If *A* depends on *B* and *B* depends on *C*, *C* is said to be a **transitive dependency** of *A.*
- Maven manages transitive dependencies automatically.
    - Can resolve conflicts caused by transitive dependencies.

Transitive Dependencies (2)

- The table below shows how transitive dependencies are propagated.
  - If *A* depends on *B* with a scope in the left column, and *B* depends on *C* with a scope in the first row, the table cell in the intersection of the corrresponding row and column contains the scope that *A* depends on *C* with.

	compile	**provided**	**runtime**	**test**
compile	compile	–	runtime	–
provided	provided	–	provided	–
runtime	runtime	–	runtime	–
test	test	–	test	–

Transitive Dependencies (3)

```xml
<project xmlns=
"http://maven.apache.org/POM/4.0.0">
 <modelVersion>4.0.0</modelVersion>
 <groupId>my</groupId>
 <artifactId>project-A</artifactId>
 <packaging>jar</packaging>
 <version>1.0</version>
 <dependencies>
  <dependency>
   <groupId>my</groupId>
   <artifactId>project-B</artifactId>
   <version>1.0</version>
   <scope>compile</scope>
  </dependency>
 </dependencies>
</project>
```

```xml
<project xmlns=
"http://maven.apache.org/POM/4.0.0">
 <modelVersion>4.0.0</modelVersion>
 <groupId>my</groupId>
 <artifactId>project-B</artifactId>
 <packaging>jar</packaging>
 <version>1.0</version>
 <dependencies>
  <dependency>
   <groupId>org.slf4j</groupId>
   <artifactId>slf4j-jdk14</artifactId>
   <version>1.7.25</version>
   <scope>runtime</scope>
  </dependency>
  ...
 </dependencies>
</project>
```

- In the above example, `slf4j-jdk14` is also a `runtime` scope dependency of `project-A` implicitly.

Excluding Transitive Dependencies (1)

- The `exclusions` element is for excluding transitive dependencies.
  - It may be required for resolving dependency conflicts, but may be also useful for excluding redundant dependencies.

Excluding Transitive Dependencies (2)

- Example for excluding an unnecessary dependency:

```
<dependencies>
   <dependency>
      <groupId>org.seleniumhq.selenium</groupId>
      <artifactId>selenium</artifactId>
      <version>2.0b1</version>
      <exclusions>
         <exclusion>
            <groupId>org.testng</groupId>
            <artifactId>testng</artifactId>
         </exclusion>
      </exclusions>
   </dependency>
<dependencies>
```

  - Both JUnit and TestNG were dependencies of the `2.0b1` version of selenium, if JUnit was used, TestNG was not needed and thus had to be excluded. http://seleniumhq.wordpress.com/2011/01/25/2-0b1-and-maven/

Excluding Transitive Dependencies (3)

- Excluding all transitive dependencies of a dependency:

```
<dependency>
  ...
  <exclusions>
    <exclusion>
      <groupId>*</groupId>
      <artifactId>*</artifactId>
    <exclusion>
  </exclusions>
</dependency>
```

Optional Dependencies (1)

- Optional dependencies are nontransitive, i.e., they are not inherited by projects depending upon the project declaring the optional dependency.
  - For example, they are used for alternative dependencies, only one of which is required.

Optional Dependencies (2)

- Example: *JSR 353: Java API for JSON Processing* https://jcp.org/en/jsr/detail?id=353
  - Not part of the Java SE platform, but is included in Java EE 7 (see package `javax.json` and it's subpackages).
    - For compiling Java SE applications the API is provided by the artifact `javax.json:javax.json-api:1.1.2` available from Maven Central.
  - An implementation is required for running applications that use the API, two alternatives available from Maven Central:
    - Oracle's reference implementation (`org.glassfish:javax.json`) https://jsonp.java.net/
    - Genson (`com.owlike:genson`) http://owlike.github.io/genson/

Optional Dependencies (3)

- The project below declares both implementations as optional dependencies:

```
<project xmlns="http://maven.apache.org/POM/4.0.0">
    <modelVersion>4.0.0</modelVersion>
    <groupId>hu.unideb.inf.maven</groupId>
    <artifactId>json-processor</artifactId>
    <packaging>jar</packaging>
    <version>1.0</version>
    <dependencies>
        <dependency>
            <groupId>org.glassfish</groupId>
            <artifactId>javax.json</artifactId>
            <version>1.1.2</version>
            <optional>true</optional>
            <scope>runtime</scope>
        </dependency>
        <dependency>
            <groupId>com.owlike</groupId>
            <artifactId>genson</artifactId>
            <version>1.4</version>
            <optional>true</optional>
            <scope>runtime</scope>
        </dependency>
        ...
```

Optional Dependencies (4)

- (continued from previous page):

```
                ...
                <dependency>
                    <groupId>javax.json</groupId>
                    <artifactId>javax.json-api</artifactId>
                    <version>1.1.2</version>
                    <scope>compile</scope>
                </dependency>
            </dependencies>
```

Optional Dependencies (5)

- Optional dependencies are nontransitive, thus, projects that depend on the previous one must explicitly declare one of the alternative implementations as a dependency!

```
<dependencies>
   <dependency>
      <groupId>hu.unideb.inf.maven</groupId>
      <artifactId>json-processor</artifactId>
      <version>1.0</version>
      <scope>compile</scope>
   </dependency>
   <dependencies>
   <dependency>
      <groupId>com.owlike</groupId>
      <artifactId>genson</artifactId>
      <version>1.4</version>
      <scope>runtime</scope>
   </dependency>
</dependencies>
```

Providing Defaults for Dependencies (1)

- The top level `dependencyManagement` element may contain a single `dependencies` element.
  - Unlike the top level `dependencies` element, artifacts listed in it will not be dependencies of the project automatically!
  - These `dependency` elements provide only default version numbers for the named artifacts, that allows them to be declared as dependencies in the project and in its children without specifying the version number.
    - Scope can be also provided with the version number.

Providing Defaults for Dependencies (2)

- Example:

```
<dependencyManagement>
  <dependencies>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>4.12</version>
      <scope>test</scope>
    </dependency>
    ...
  </dependencies>
</dependencyManagement>
```

Providing Defaults for Dependencies (3)

- Example (continued):
  - In such a case, when declaring the artifact as a dependency in the project or in its children, the version number and the scope can be omitted, both are provided by the `dependencyManagement` element:

```
<dependencies>
  <dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
  </dependency>
  ...
</dependencies>
```

Snapshot Versions

- The suffix `-SNAPSHOT` that may appear at the end of version numbers indicates that the project is under active development.
  - Example: `1.0-SNAPSHOT`
- When the artifact is deployed to a repository the suffix `SNAPSHOT` is expanded with the current timestamp in UTC.
  - For example, on 3 January 2018 at 21:58:34 CET the result of expanding the above version number is the version number `1.0-20180103.205834-`*N*.
  - *N* is an integer that starts at 1 and is increased by 1 with each deployment.

Snapshot and Release Artifacts (1)

- Artifacts with a snapshot version number are called **snapshot artifacts**.
  - Reflect the status of development at a time instant.
  - Should be used only during development.
  - Newer snapshots make them obsolete quickly.
- Other (i.e., non-snapshot) artifacts are called **release artifacts**.
  - Can be considered as stable.
  - Typically used for a longer period of time.

Snapshot and Release Artifacts (2)

- Usually deployed to separate repositories.
- The same repository can be used to publish snapshot and release artifacts.
  - See the `releases` and `snapshots` elements available in the `repositories/repository` and in the `pluginRepositories/pluginRepository` elements.

Uploading Artifacts to a Remote Repository (1)

- Artifacts are uploaded to a remote repository specified in the POM in the `deploy` phase.
- The `repository` and the `snapshotRepository` elements available in the `distributionManagement` element of the POM contain information for accessing remote repositories.
  - The `repository` element describes the repository for release artifacts, and the `snapshotRepository` element describes the repository for snapshot artifacts, respectively.

Uploading Artifacts to a Remote Repository (2)

- The `repository` and the `snapshotRepository` elements:

```
<distributionManagement>
  <repository>
    <id>id</id>
    <name>name</name>
    <url>URI</url>
    <layout>default|legacy</layout>
    <uniqueVersion>true|false</uniqueVersion>
  </repository>
  <snapshotRepository>
    <id>id</id>
    <name>name</name>
    <url>URI</url>
    <layout>default|legacy</layout>
    <uniqueVersion>true|false</uniqueVersion>
  </snapshotRepository>
</distributionManagement>
```

Uploading Artifacts to a Remote Repository (3)

- Elements available in the `repository` and the `snapshotRepository` elements:
  - `id`: a unique identifier for the repository
  - `name`: the human readable name of the repository
  - `url`: the URI for accessing the repository
  - `layout`: the layout of the repository
    - `default`: the layout used by the 2.*x* and 3.*x* versions of Maven (this is the default)
    - `legacy`: the layout used by the 1.*x* versions of Maven
  - `uniqueVersion`: whether to assign snapshots a unique version comprised of the timestamp and build number, or to use the same version each time (default: `true`)

Configuration of Repository Access (1)

- Configuration settings for repositories for dependencies must be provided in the top-level `repositories` element:

```
<repositories>
  <repository>
    <id>id</id>
    <name>name</name>
    <url>URI</url>
    <layout>default|legacy</layout>
```

Configuration of Repository Access (2)

(continued from previous page)

```
      <releases>
         <checksumPolicy>fail|ignore|warn</checksumPolicy>
         <enabled>false|true</enabled>
         <updatePolicy>always|daily|interval:N|
            never</updatePolicy>
      </releases>
      <snapshots>
         <checksumPolicy>fail|ignore|warn</checksumPolicy>
         <enabled>false|true</enabled>
         <updatePolicy>always|daily|interval:N|
            never</updatePolicy>
      </snapshots>
   </repository>
   ...
</repositories>
```

Configuration of Repository Access (3)

- Elements available in the `repository` element:
  - `id`: a unique identifier for the repository
  - `name`: the human readable name of the repository
  - `url`: the URI for accessing the repository
  - `layout`: the layout of the repository
    - `default`: the layout used by the 2.*x* and 3.*x* versions of Maven (this is the default)
    - `legacy`: the layout used by the 1.*x* versions of Maven
  - `releases`: configuration settings for downloading release artifacts
  - `snapshots`: configuration settings for downloading snapshot artifacts

Configuration of Repository Access (4)

- Elements available in the `releases` and the `snapshots` elements:
  - `checksumPolicy`: how to handle artifact checksum errors (repositories maintain an MD5 and/or SHA-1 checksum for each artifact)
    - `fail`
    - `ignore`
    - `warn` (this is the default)
  - `enabled`: indicates whether downloading of artifacts of the respective type (release or snapshot) is enabled (default: `true`)

Configuration of Repository Access (5)

- Elements available in the `releases` and the `snapshots` elements (continued):
  - `updatePolicy`: how often Maven should perform updates from the repository
    - `always`: on each run of Maven
    - `daily`: once a day (this is the default)
    - `interval:`*N* (where *N* is an integer): every *N* minutes
    - `never`

Configuration of Repository Access (6)

- For repositories for plugins configuration settings must be provided in the `pluginRepositories` element:

```
<pluginRepositories>
  <pluginRepository>
    ...
  </pluginRepository>
  ...
</pluginRepositories>
```

  - The content model of the `pluginRepository` element is exactly the same as of the `repository` element.

Depending upon Snapshot Artifacts (1)

- It can be set to use the most recent snapshot available in the remote repository when depending upon a snapshot artifact.
  - The setting is provided in the `snapshots` element available in the `repository` and the `pluginRepository` elements:

```
<repository>
   ...
   <snapshots>
      <enabled>true</enabled>
      <updatePolicy>update policy</updatePolicy>
   </snapshots>
   ...
</repository>
```

- Do not forget that snapshot artifacts are used only during development!

Depending upon Snapshot Artifacts (2)

- If a snapshot artifact that is not available in the local repository is used as a dependency, Maven will download the most recent snapshot from the remote repository.
- If the local repository contains at least one snapshot of the artifact, Maven will check whether the remote repository contains a more recent one.
    - If yes, then the most recent one will be downloaded from the remote repository.
- The `updatePolicy` element controls that how often Maven will check the remote repository for newer snapshots.

Depending upon Snapshot Artifacts (3)

- The meaning of the `updatePolicy` values:
  - **`always`**: Maven will check the remote repository on each run
  - **`daily`**: Maven will check the remote repository on the first run of the day
  - **`interval:`***N* (where *N* is an integer): Maven will check the repository if *N* minutes elapsed since the last check
  - **`never`**: Maven will never check the remote repository

Depending upon Release Artifacts (1)

- For release artifacts the `updatePolicy` element can be used exactly the same as for snapshot artifacts:

```
<repository>
  ...
  <releases>
    <enabled>true</enabled>
    <updatePolicy>update policy</updatePolicy>
  </releases>
  ...
</repository>
```

Depending upon Release Artifacts (2)

- To understand how the `updatePolicy` setting works for release artifacts, the following must be noted:
  - Each release artifact is downloaded from the remote repository only once!
    - A release artifact will not be downloaded again, even if it was overwritten in the remote repository.
- An `updatePolicy` other than `never` will result in the download of a more recent version of the release artifact only in the case of version ranges.

Manual Installation of Artifacts in the Local Repository

- The following command will do the work:

```
mvn install:install-file \
   -Dfile=path \
   -DgroupId=groupId \
   -DartifactId=artifactId \
   -Dversion=version \
   -Dpackaging=packaging \
   -DgeneratePom=true
```

  - For example, use it for JARs that are not provided by any of the available remote repositories.

Inheritance (1)

- A project with `pom` packaging type can be declared as a parent:

```
<project xmlns=
     "http://maven.apache.org/POM/4.0.0">
  <modelVersion>4.0.0</modelVersion>
  <groupId>hu.unideb.inf.maven</groupId>
  <artifactId>parent</artifactId>
  <packaging>pom</packaging>
  <version>1.0</version>
  ...
</project>
```

Inheritance (2)

- The parent is specified in a child's POM as follows:

```
<project xmlns=
  "http://maven.apache.org/POM/4.0.0">
  <modelVersion>4.0.0</modelVersion>
  <parent>
    <groupId>hu.unideb.inf.maven</groupId>
    <artifactId>parent</artifactId>
    <version>1.0</version>
  </parent>
  <artifactId>child</artifactId>
  <packaging>jar</packaging>
  ...
</project>
```

Inheritance (3)

- A child project will inherit elements from the POM of the parent when the effective POM is constructed.
  - Specific elements will be inherited from the parent only if they are not specified explicitly in the child's POM.
    - For example, the `ciManagement`, `contributors`, `developers`, `groupId`, `issueManagement`, `licenses`, `mailingLists`, `organization`, `url` and `version` elements are handled in this way.
  - In the case of other elements, content is combined when they appear both in the parent's and the child's POM.
    - For example, the `plugins`, `scm` and `repositories` elements are handled in this way.

Multi-Module Projects (1)

- A multi-module project, also called an aggregator project, consists of sub-projects called modules.
  - The packaging type of the aggregator project must be `pom`.
    - Modules can have an arbitrary packaging type, they can also be multi-module projects.
  - Modules are listed in the `modules` element.
    - The content of a `module` element is the relative path of the directory that contains the module.
      - An aggregator project typically contains it's modules in subdirectories.

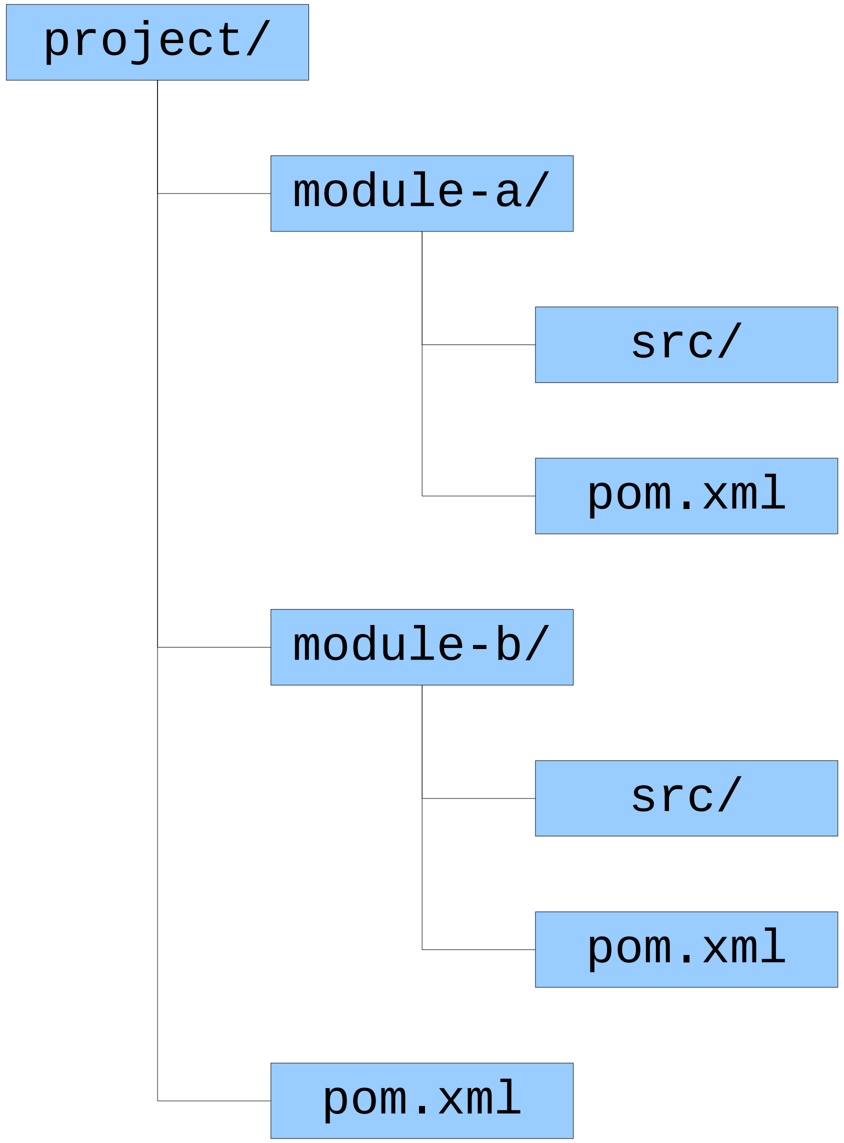
Multi-Module Projects (2)

- When a lifecycle phase or a plugin goal is executed in the directory of the aggregator project then execution will be performed in the directory of each module.
  - Maven will automatically determine the execution order (modules can depend on each other).

Multi-Module Projects (3)

- Parent-child relationship can be defined between an aggregator project and it's modules.
  - This is not mandatory!
  - A multi-module project typically provides defaults in it's POM for the modules.

Structure of Multi-Module Projects



The POM of the aggregator project:

```
<project xmlns=
  "http://maven.apache.org/POM/4.0.0">
  <modelVersion>4.0.0</modelVersion>
  <groupId>my</groupId>
  <artifactId>project</artifactId>
  <packaging>pom</packaging>
  <version>1.0</version>
  <modules>
    <module>module-a</module>
    <module>module-b</module>
  </modules>
  ...
</project>
```

Creating a Multi-Module Project: m2eclipse

1. Create the aggregator project: *File* → *New* → *Project…* → *Maven* → *Maven Project*
   - Click the *Create a simple project (skip archetype selection)* checkbox!
   - Select `pom` from the drop-down list next to *Packaging*.
   - The `src/` directory created automatically is unnecessary in the aggregator project, and thus it can be deleted.
2. To create a module do the following: *File* → *New* → *Project…* → *Maven* → *Maven Module*
   - The parent of the module can be provided/selected in the *Parent Project* field.
   - The module will also be a child of the aggregator project.

Creating a Multi-Module Project: NetBeans

1. Create the aggregator project: *File* → *New Project*
   - Select *POM Project* from the category *Maven*
2. To create a module do the following:
   - In the *Projects* panel right click on *Modules* under the name of the aggregator project and select *Create New Module…*
   - The module will also be a child of the aggregator project.

Profiles (1)

- Profiles are parts of the POM that contain optional settings to be used only on activation.
  - They allow the runtime modification of the POM.
  - They are useful when the project must be built in different environments that require different settings.
    - They provide settings customized for different build environments.

Profiles (2)

- Profiles are specified in the POM as follows:

```
<profiles>
  <profile>
    <id>identifier</id>
    <activation>activation condition(s)</activation>
    profile-specific settings
  </profile>
  <profile>
    <id>identifier</id>
    <activation>activation condition(s)</activation>
    profile-specific settings
  </profile>
  ...
</profiles>
```

Profiles (3)

- The following elements are available in the `profile` element:
  - `build`
  - `dependencies`
  - `dependencyManagement`
  - `distributionManagement`
  - `modules`
  - `pluginRepositories`
  - `properties`
  - `reporting`
  - `repositories`

Profiles (4)

- The Maven Help Plugin provides information about the profiles defined in the POM.
  - The `mvn help:all-profiles` command displays all profiles specified in the POM, and the `mvn help:active-profiles` command displays the list of active profiles.

Profile Activation

- Profiles can be activated by the user explicitly or automatically when specific conditions are met.
  - Automatic activation can be based on:
    - The values of environment variables and system properties
    - The operating system
    - The version number of the JDK
    - The existence or absence of files
  - Profile activation conditions are specified in the `file`, `jdk`, `os` and `property` elements that are available in the `activation` element.
    - If more than one of them is present in the `activation` element, the profile is activated when any of the conditions is met (logical or).

Profile Activation: Explicit

- Profiles can be activated with the `-P` or `--activate-profiles` command line options that require a list of profile identifiers to be specified (multiple identifiers must be separated by `','` characters).
  - A `'!'` or `'-'` character at the beginning of a profile identifier will cause the profile to be deactivated.
    - Warning: in the Bash shell the `'!'` character has special meaning, and thus it must be escaped properly!
- Example (activate the `profile-1` profile and deactivate the `profile-2` profile):

```
mvn help:active-profiles -P profile-1,-profile-2
```

Profile Activation: Profiles Active by Default

- A profile specified as follows is active by default:

```
<profile>
  <id>default</id>
  <activation>
    <activeByDefault>true</activeByDefault>
  </activation>
  ...
</profile>
```

  - Such profiles will be deactivated in the case of explicit profile activation and automatic activation of profiles that are not active by default.
    - Unless they are activated explicitly.

Profile Activation: System Properties

- Activate the profile when the debug system property is set (its value can be anything):

```
<activation>
   <property>
      <name>debug</name>
   </property>
</activation>
```

- Activate the profile when the debug system property is not set:

```
<activation>
   <property>
      <name>!debug</name>
   </property>
</activation>
```

- Activate the profile when the value of the `environment.type` system property is `production`:

```
<activation>
   <property>
      <name>environment.type</name>
      <value>production</value>
   </property>
</activation>
```

Profile Activation: Environment Variables

- Activate the profile when the `DEBUG` environment variable is set (its value can be anything):

```
<activation>
   <property>
      <name>env.DEBUG</name>
   </property>
</activation>
```

- Activate the profile when the value of the ENV environment variable is `test`:

```
<activation>
   <property>
      <name>env.ENV</name>
      <value>test</value>
   </property>
</activation>
```

- Activate the profile when the `DEBUG` environment variable is not set:

```
<activation>
<property>
   <name>!env.DEBUG</name>
</property>
</activation>
```

- Activate the profile when the value of the ENV environment variable is not `test`:

```
<activation>
   <property>
      <name>env.ENV</name>
      <value>!test</value>
   </property>
</activation>
```

Profile Activation: OS-specific (1)

- Use the `os` element to activate a profile based on the OS:

```
<activation>
   <os>
      <arch>...</arch>
      <name>...</name>
      <family>...</family>
      <version>...</version>
   </os>
</activation>
```

  - `arch`: operating system architecture (e.g., `amd64`, `x86`, …)
  - `name`: name of the operating system (e.g., `linux`, `windows xp`, …)
  - `family`: operating system family (`mac`, `unix`, `windows`)
  - `version`: version number of the operating system (exact version number, version ranges are not supported)
- In the `arch`, `name`, `family` and `version` elements the `'!'` sign means negation (e.g., `<family>!windows</family>`).

Profile Activation: OS-specific (2)

- Example:

  -
    ```
    <activation>
      <os>
        <name>linux</name>
        <arch>amd64</arch>
      </os>
    </activation>
    ```

  -
    ```
    <activation>
      <os>
        <name>windows xp</name>
        <version>5.1</version>
      </os>
    </activation>
    ```

Profile Activation: JDK

- Use the `jdk` element to activate a profile based on the version number of the JDK.
  - The element must contain a prefix or a version range. A `'!'` sign as the first character of a prefix means negation.

- Példa:

```
<profile>
  <id>jdk6</id>
  <activation>
    <jdk>1.6</jdk>
  </activation>
  ...
</profile>
```

- Példa:

```
<profile>
  <id>pre-jdk6</id>
  <activation>
    <jdk>[,1.6)</jdk>
  </activation>
  ...
</profile>
```

Profile Activation: Files (1)

- Use the `file` element to activate a profile based on the presence/absence of a given file.

```
<activation>
  <file>
    <exists>...</exists>
    <missing>...</missing>
  </file>
</activation>
```

- A file path must be provided in the `exists` and `missing` elements.

Profile Activation: Files (2)

- Example:

  -
    ```
    <activation>
       <file>
          <exists>${user.home}/.myTool/license.txt</exists>
       </file>
    </activation>
    ```

  -
    ```
    <activation>
       <file>
          <missing>${basedir}/.git</missing>
       </file>
    </activation>
    ```

Using Plugins (1)

```
<build>
  <plugins>
    <plugin>
      <groupId>groupId</groupId>
      <artifactId>artifactId</artifactId>
      <version>version</version>
      <configuration>configuration</configuration>
      <dependencies>dependencies</dependencies>
      <executions>plugin goal executions</executions>
      <extensions>false|true</extensions>
      <inherited>false|true</inherited>
    </plugin>
    ...
  </plugins>
```

Using Plugins (2)

- The elements available in the the `plugin` element:
  - `groupId`, `artifactId`, `version`: provide the Maven coordinates of the plugin
  - `configuration`: provides configuration for the execution of plugin goals
    - The XML schema does not impose any restrictions on the content of the element.
  - `dependencies`: lists the dependencies required by the plugin
    - Dependencies can be specified as discussed earlier.

Using Plugins (3)

- The elements available in the `plugin` element (continued from previous page):
  - `executions`: allows the binding of plugin goals to lifecycle phases, thus providing a means to customize the build process
  - `extensions`: indicates whether or not to load extensions of the plugin (default: `false`)
  - `inherited`: indicates whether plugin configuration will be inherited by child projects (default: `true`)

Using Plugins (4)

- The `executions` element:

```
<plugin>
   <groupId>...</groupId>
   <artifactId>...</artifactId>
   <version>...</version>
   <executions>
      <execution>
         <id>identifier</id>
         <goals>
            <goal>goal1</goal>
            ...
            <goal>goaln</goal>
         </goals>
         <phase>lifecycle phase</phase>
         <inherited>false|true</inherited>
         <configuration>configuratio</configuration>
      </execution>
      ...
   </executions>
   ...
</plugin>
```

Using Plugins (5)

- The `execution` element:
  - **`id`**: a unique identifier for the element
  - **`goals/goal`**: the name of a plugin goal to be executed
  - **`phase`**: the name of the lifecycle phase to bind the goal(s) to
  - **`inherited`**: indicates whether this execution will be inherited by child projects (default: `true`)
  - **`configuration`**: provides configuration for the execution of plugin goals specified in the `goal` elements
    - Refines configuration provided in the `plugin/configuration` element.

Using Plugins (6)

- An example of how to use the `executions` element:

```
<build>
    <plugins>
        <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-antrun-plugin</artifactId>
            <version>1.8</version>
            <executions>
                <execution>
                    <id>confirm-clean</id>
                    <phase>pre-clean</phase>
                    <goals>
                        <goal>run</goal>
                    </goals>
                    <configuration>
                        <target>
                            <echo>Cleaning up...</echo>
                        </target>
                    </configuration>
                </execution>
            </executions>
        </plugin>
    ...
```

Using Plugins (7)

- A plugin goal can have a phase to which it is bound by default, in this case, the `phase` element can be omitted from the `execution` element.
  - If the goal does not have a default phase binding, the goal will not be executed at all in the absence of the `phase` element!

Using Plugins (8)

- For example, the `enforce` goal of the `maven-enforcer-plugin` is bound to the `validate` lifecycle phase by default:

```
<build>
    <plugins>
        <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-enforcer-plugin</artifactId>
            <version>3.0.0-M1</version>
            <executions>
                <execution>
                    <id>enforce-java-version</id>
                    <goals><goal>enforce</goal></goals>
                    <configuration>
                        <rules>
                            <requireJavaVersion>
                                <version>1.9</version>
                            </requireJavaVersion>
                        </rules>
                    </configuration>
                </execution>
            </executions>
        </plugin>
    </plugins>
</build>
```

Generating a Site (1)

- Reporting plugins that generate reports to be automatically included in the site are listed under the `reporting` element:

```
<reporting>
   <outputDirectory>path</outputDirectory>
   <plugins>
      list of reporting plugins (plugin elements)
   </plugins>
   <excludeDefaults>false|true</excludeDefaults>
</reporting>
```

  - `outputDirectory`: the path of the output directory (default: `${project.build.directory}/site`)
  - `excludeDefaults`: exclude reports normally generated by default (default: `false`)

Generating a Site (2)

- Maven 3 also allows the listing of reporting plugins as follows:

```
<build>
   <plugins>
      ...
      <plugin>
         <groupId>org.apache.maven.plugins</groupId>
         <artifactId>maven-site-plugin</artifactId>
         <version>3.7</version>
         <configuration>
            <reportPlugins>
               list of reporting plugins (plugin elements)
            </reportPlugins>
         </configuration>
      </plugin>
      ...
   </plugins>
</build>
```

Generating a Site (3)

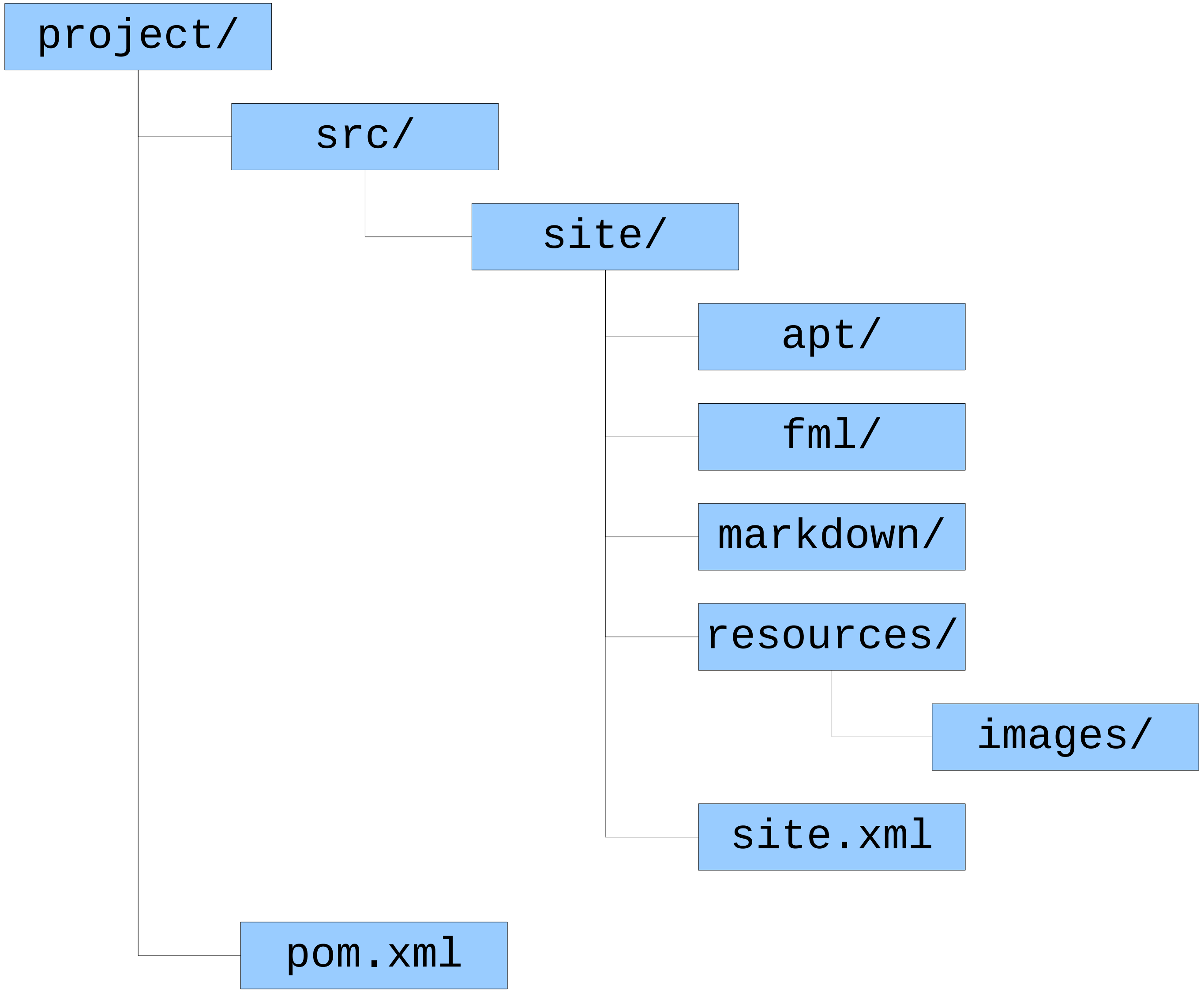
- For multi-module projects the `mvn site site:stage` command must be executed instead of the `mvn site` command in order to generate a site.
  - The site will be created under the `${basedir}/target/staging/` directory!
  - Multi-module site generation requires the following element to be placed in the POM:

```
<distributionManagement>
   <site>
      <id>website</id>
      <url>file:///tmp/fake.com/</url>
   </site>
</distributionManagement>
```

Site Customization (1)

- To customize the site, all the necessary files must be placed in the `${basedir}/src/site/` directory.
  - The structure of the site can be customized in the `site.xml` file (also called site descriptor).
    - The following XML schema describes the site descriptor format: http://maven.apache.org/xsd/decoration-1.7.0.xsd
  - The directories under the `${basedir}/src/site/` directory contain content to be rendered on the site.
    - Content is stored in file formats from which HTML is generated automatically.

Site Customization (2)



Site Customization (3)

- Formats: http://maven.apache.org/doxia/references/
  - APT (Almost Plain Text) http://maven.apache.org/doxia/references/apt-format.html
  - FML (FAQ Markup Language) http://maven.apache.org/doxia/references/fml-format.html
  - Markdown http://daringfireball.net/projects/markdown/