Java Enterprise Edition

ANUDIP FOUNDATION

Maven Fundamentals

Objective:	Materials Required:
• Making the build process easy • Providing a uniform build system • Providing quality project information • Encouraging better development practices	1. Eclipse/IntelliJ/STC 2. Maven
Theory:180mins	**Practical:60mins**
Total Duration: 240 mins	

I . Introduction

Maven, a Yiddish word meaning *accumulator of knowledge*, began as an attempt to simplify the build processes in the Jakarta Turbine project. There were several projects, each with their own Ant build files, that were all slightly different. JARs were checked into CVS. We wanted a standard way to build the projects, a clear definition of what the project consisted of, an easy way to publish project information, and a way to share JARs across several projects.

The result is a tool that can now be used for building and managing any Java-based project. We hope that we have created something that will make the day-to-day work of Java developers easier and generally help with the comprehension of any Java-based project.

Maven's Objectives

Maven's primary goal is to allow a developer to comprehend the complete state of a development effort in the shortest period of time. In order to attain this goal, Maven deals with several areas of concern:

- Making the build process easy
- Providing a uniform build system
- Providing quality project information
- Encouraging better development practices

Making the build process easy

While using Maven doesn't eliminate the need to know about the underlying mechanisms, Maven does shield developers from many details.

Providing a uniform build system

Maven builds a project using its project object model (POM) and a set of plugins. Once you familiarize yourself with one Maven project, you know how all Maven projects build. This saves time when navigating many projects.

Providing quality project information

Maven provides useful project information that is in part taken from your POM and in part generated from your project's sources. For example, Maven can provide:

- Change log created directly from source control
- Cross referenced sources
- Mailing lists managed by the project
- Dependencies used by the project
- Unit test reports including coverage

Third party code analysis products also provide Maven plugins that add their reports to the standard information given by Maven.

Providing guidelines for best practices development

Maven aims to gather current principles for best practices development and make it easy to guide a project in that direction.

For example, specification, execution, and reporting of unit tests are part of the normal build cycle using Maven. Current unit testing best practices were used as guidelines:

- Keeping test source code in a separate, but parallel source tree
- Using test case naming conventions to locate and execute tests
- Having test cases setup their environment instead of customizing the build for test preparation

Maven also assists in project workflow such as release and issue management.

Maven also suggests some guidelines on how to layout your project's directory structure. Once you learn the layout, you can easily navigate other projects that use Maven.

While takes an opinionated approach to project layout, some projects may not fit with this structure for historical reasons. While Maven is designed to be flexible to the needs of different projects, it cannot cater to every situation without compromising its objectives.

If your project has an unusual build structure that cannot be reorganized, you may have to forgo some features or the use of Maven altogether.

What is Maven Not?

You might have heard some of the following things about Maven:

- Maven is a site and documentation tool
- Maven extends Ant to let you download dependencies
- Maven is a set of reusable Ant scriptlets

While Maven does these things, as you can read above in the “What is Maven?” section, these are not the only features Maven has, and its objectives are quite different.

II. FOLDER STRUCTURE OF MAVEN

Having a common directory layout allows users familiar with one Maven project to immediately feel at home in another Maven project. The advantages are analogous to adopting a site-wide look-and-feel.

The next section documents the directory layout expected by Maven and the directory layout created by Maven. Try to conform to this structure as much as possible. However, if you can't, these settings can be overridden via the project descriptor.

src/main/java	Application/Library sources
src/main/resources	Application/Library resources
src/main/filters	Resource filter files
src/main/webapp	Web application sources
src/test/java	Test sources
src/test/resources	Test resources
src/test/filters	Test resource filter files
src/it	Integration Tests (primarily for plugins)
src/assembly	Assembly descriptors
src/site	Site
LICENSE.txt	Project's license
NOTICE.txt	Notices and attributions required by libraries that the project depends on
README.txt	Project's readme

At the top level, files descriptive of the project: a `pom.xml` file. In addition, there are textual documents meant for the user to be able to read immediately on receiving the source: `README.txt`, `LICENSE.txt`, etc.

There are just two subdirectories of this structure: `src` and `target`. The only other directories that would be expected here are metadata like `CVS`, `.git` or `.svn`, and any subprojects in a multiproject build (each of which would be laid out as above).

The `target` directory is used to house all output of the build.

The `src` directory contains all of the source material for building the project, its site and so on. It contains a subdirectory for each type: `main` for the main build artifact, `test` for the unit test code and resources, `site` and so on.

Within artifact producing source directories (ie. `main` and `test`), there is one directory for the language `java` (under which the normal package hierarchy exists), and one for `resources` (the structure which is copied to the target classpath given the default resource definition).

If there are other contributing sources to the artifact build, they would be under other subdirectories. For example `src/main/antlr` would contain Antlr grammar definition files.

III. The POM.XML

What is a POM?

A Project Object Model or POM is the fundamental unit of work in Maven. It is an XML file that contains information about the project and configuration details used by Maven to build the project. It contains default values for most projects. Examples for this is the build directory, which is `target`; the source directory, which is `src/main/java`; the test source directory, which is `src/test/java`; and so on. When executing a task or goal, Maven looks for the POM in the current directory. It reads the POM, gets the needed configuration information, then executes the goal.

Some of the configuration that can be specified in the POM are the project dependencies, the plugins or goals that can be executed, the build profiles, and so on. Other information such as the project version, description, developers, mailing lists and such can also be specified.

[top]

Super POM

The Super POM is Maven's default POM. All POMs extend the Super POM unless explicitly set, meaning the configuration specified in the Super POM is inherited by the POMs you created for your projects.

You can see the Super POM for Maven 3.6.3 in Maven Core reference documentation.

Minimal POM

The minimum requirement for a POM are the following:

- `project` root
- `modelVersion` - should be set to 4.0.0

- `groupId` - the id of the project's group.
- `artifactId` - the id of the artifact (project)
- `version` - the version of the artifact under the specified group

Here's an example:

```
<project>
  <modelVersion>4.0.0</modelVersion>

  <groupId>com.mycompany.app</groupId>
  <artifactId>my-app</artifactId>
  <version>1</version>
</project>
```

A POM requires that its groupId, artifactId, and version be configured. These three values form the project's fully qualified artifact name. This is in the form of <groupId>:<artifactId>:<version>. As for the example above, its fully qualified artifact name is "com.mycompany.app:my-app:1".

Also, as mentioned in the first section, if the configuration details are not specified, Maven will use their defaults. One of these default values is the packaging type. Every Maven project has a packaging type. If it is not specified in the POM, then the default value "jar" would be used.

Furthermore, you can see that in the minimal POM the *repositories* were not specified. If you build your project using the minimal POM, it would inherit the *repositories* configuration in the Super POM. Therefore when Maven sees the dependencies in the minimal POM, it would know that these dependencies will be downloaded from `https://repo.maven.apache.org/maven2` which was specified in the Super POM.

Project Inheritance

Elements in the POM that are merged are the following:

- dependencies
- developers and contributors
- plugin lists (including reports)
- plugin executions with matching ids
- plugin configuration
- resources

The Super POM is one example of project inheritance, however you can also introduce your own parent POMs by specifying the parent element in the POM, as demonstrated in the following examples.

Example 1

The Scenario

As an example, let us reuse our previous artifact, com.mycompany.app:my-app:1. And let us introduce another artifact, com.mycompany.app:my-module:1.

```
<project>
  <modelVersion>4.0.0</modelVersion>

  <groupId>com.mycompany.app</groupId>
  <artifactId>my-module</artifactId>
  <version>1</version>
</project>
```

And let us specify their directory structure as the following:

```
.
 |-- my-module
 |   `-- pom.xml
 `-- pom.xml
```

Note: `my-module/pom.xml` is the POM of com.mycompany.app:my-module:1 while `pom.xml` is the POM of com.mycompany.app:my-app:1

The Solution

Now, if we were to turn com.mycompany.app:my-app:1 into a parent artifact of com.mycompany.app:my-module:1,we will have to modify com.mycompany.app:my-module:1's POM to the following configuration:

com.mycompany.app:my-module:1's POM

```
<project>
  <modelVersion>4.0.0</modelVersion>

  <parent>
    <groupId>com.mycompany.app</groupId>
    <artifactId>my-app</artifactId>
    <version>1</version>
  </parent>

  <groupId>com.mycompany.app</groupId>
  <artifactId>my-module</artifactId>
  <version>1</version>
</project>
```

Notice that we now have an added section, the parent section. This section allows us to specify which artifact is the parent of our POM. And we do so by specifying the fully qualified artifact name of the parent POM. With this setup, our module can now inherit some of the properties of our parent POM.

Alternatively, if we want the groupId and / or the version of your modules to be the same as their parents, you can remove the groupId and / or the version identity of your module in its POM.

```
<project>
  <modelVersion>4.0.0</modelVersion>

  <parent>
    <groupId>com.mycompany.app</groupId>
    <artifactId>my-app</artifactId>
    <version>1</version>
  </parent>

  <artifactId>my-module</artifactId>
</project>
```

This allows the module to inherit the groupId and / or the version of its parent POM.

Example 2

The Scenario

However, that would work if the parent project was already installed in our local repository or was in that specific directory structure (parent `pom.xml` is one directory higher than that of the module's `pom.xml`).

But what if the parent is not yet installed and if the directory structure is as in the following example?

```
.
 |-- my-module
 |   `-- pom.xml
 `-- parent
     `-- pom.xml
```

The Solution

To address this directory structure (or any other directory structure), we would have to add the `<relativePath>` element to our parent section.

```
<project>
  <modelVersion>4.0.0</modelVersion>

  <parent>
    <groupId>com.mycompany.app</groupId>
    <artifactId>my-app</artifactId>
    <version>1</version>
    <relativePath>../parent/pom.xml</relativePath>
  </parent>

  <artifactId>my-module</artifactId>
</project>
```

As the name suggests, it's the relative path from the module's `pom.xml` to the parent's `pom.xml`.

Project Aggregation

Project Aggregation is similar to Project Inheritance. But instead of specifying the parent POM from the module, it specifies the modules from the parent POM. By doing so, the parent project now knows its modules, and if a Maven command is invoked against the parent project, that Maven command will then be executed to the parent's modules as well. To do Project Aggregation, you must do the following:

- Change the parent POMs packaging to the value "pom".
- Specify in the parent POM the directories of its modules (children POMs).

Example 3

The Scenario

Given the previous original artifact POMs and directory structure:

com.mycompany.app:my-app:1's POM

```
<project>
  <modelVersion>4.0.0</modelVersion>

  <groupId>com.mycompany.app</groupId>
  <artifactId>my-app</artifactId>
  <version>1</version>
</project>
```

com.mycompany.app:my-module:1's POM

```
<project>
  <modelVersion>4.0.0</modelVersion>

```

```
    <groupId>com.mycompany.app</groupId>
    <artifactId>my-module</artifactId>
    <version>1</version>
  </project>
```

directory structure

```
.
 |-- my-module
 |   `-- pom.xml
 `-- pom.xml
```

The Solution

If we are to aggregate my-module into my-app, we would only have to modify my-app.

```
  <project>
    <modelVersion>4.0.0</modelVersion>

    <groupId>com.mycompany.app</groupId>
    <artifactId>my-app</artifactId>
    <version>1</version>
    <packaging>pom</packaging>

    <modules>
      <module>my-module</module>
    </modules>
  </project>
```

In the revised com.mycompany.app:my-app:1, the packaging section and the modules sections were added. For the packaging, its value was set to "pom", and for the modules section, we have the element `<module>my-module</module>`. The value of `<module>` is the relative path from the com.mycompany.app:my-app:1 to com.mycompany.app:my-module:1's POM (*by practice, we use the module's artifactId as the module directory's name*).

Now, whenever a Maven command processes com.mycompany.app:my-app:1, that same Maven command would be ran against com.mycompany.app:my-module:1 as well. Furthermore, some commands (goals specifically) handle project aggregation differently.

Example 4

The Scenario

But what if we change the directory structure to the following:

```
.
 |-- my-module
 |   `-- pom.xml
 `-- parent
     `-- pom.xml
```

How would the parent POM specify its modules?

The Solution

The answer? - the same way as Example 3, by specifying the path to the module.

```
<project>
  <modelVersion>4.0.0</modelVersion>

  <groupId>com.mycompany.app</groupId>
  <artifactId>my-app</artifactId>
  <version>1</version>
  <packaging>pom</packaging>

  <modules>
    <module>../my-module</module>
  </modules>
</project>
```

Project Inheritance vs Project Aggregation

If you have several Maven projects, and they all have similar configurations, you can refactor your projects by pulling out those similar configurations and making a parent project. Thus, all you have to do is to let your Maven projects inherit that parent project, and those configurations would then be applied to all of them.

And if you have a group of projects that are built or processed together, you can create a parent project and have that parent project declare those projects as its modules. By doing so, you'd only have to build the parent and the rest will follow.

But of course, you can have both Project Inheritance and Project Aggregation. Meaning, you can have your modules specify a parent project, and at the same time, have that parent project specify those Maven projects as its modules. You'd just have to apply all three rules:

- Specify in every child POM who their parent POM is.
- Change the parent POMs packaging to the value "pom" .
- Specify in the parent POM the directories of its modules (children POMs)

Example 5

The Scenario

Given the previous original artifact POMs again,

com.mycompany.app:my-app:1's POM

```
<project>
  <modelVersion>4.0.0</modelVersion>

  <groupId>com.mycompany.app</groupId>
  <artifactId>my-app</artifactId>
  <version>1</version>
</project>
```

com.mycompany.app:my-module:1's POM

```
<project>
  <modelVersion>4.0.0</modelVersion>

  <groupId>com.mycompany.app</groupId>
  <artifactId>my-module</artifactId>
  <version>1</version>
</project>
```

and this **directory structure**

```
.
 |-- my-module
 |   `-- pom.xml
 `-- parent
     `-- pom.xml
```

The Solution

To do both project inheritance and aggregation, you only have to apply all three rules.

com.mycompany.app:my-app:1's POM

```
<project>
  <modelVersion>4.0.0</modelVersion>

  <groupId>com.mycompany.app</groupId>
  <artifactId>my-app</artifactId>
  <version>1</version>
  <packaging>pom</packaging>

  <modules>
    <module>../my-module</module>
  </modules>
</project>
```

com.mycompany.app:my-module:1's POM

```
<project>
  <modelVersion>4.0.0</modelVersion>

  <parent>
    <groupId>com.mycompany.app</groupId>
    <artifactId>my-app</artifactId>
    <version>1</version>
    <relativePath>../parent/pom.xml</relativePath>
  </parent>

  <artifactId>my-module</artifactId>
</project>
```

NOTE: Profile inheritance the same inheritance strategy as used for the POM itself.

Project Interpolation and Variables

One of the practices that Maven encourages is *don't repeat yourself*. However, there are circumstances where you will need to use the same value in several different locations. To assist in ensuring the value is only specified once, Maven allows you to use both your own and pre-defined variables in the POM.

For example, to access the `project.version` variable, you would reference it like so:

```
  <version>${project.version}</version>
```

One factor to note is that these variables are processed *after* inheritance as outlined above. This means that if a parent project uses a variable, then its definition in the child, not the parent, will be the one eventually used.

Available Variables

Project Model Variables

Any field of the model that is a single value element can be referenced as a variable. For example, `${project.groupId}`, `${project.version}`, `${project.build.sourceDirectory}` and so on. Refer to the POM reference to see a full list of properties.

These variables are all referenced by the prefix "`project.`". You may also see references with `pom.` as the prefix, or the prefix omitted entirely - these forms are now deprecated and should not be used.

Special Variables

project.basedir	The directory that the current project resides in.
project.baseUri	The directory that the current project resides in, represented as an URI. *Since Maven 2.1.0*
maven.build.timestamp	The timestamp that denotes the start of the build (UTC). *Since Maven 2.1.0-M1*

The format of the build timestamp can be customized by declaring the property `maven.build.timestamp.format` as shown in the example below:

```
<project>
  ...
  <properties>
    <maven.build.timestamp.format>yyyy-MM-dd'T'HH:mm:ss'Z'</maven.build.timestamp.format>
  </properties>
  ...
</project>
```

The format pattern has to comply with the rules given in the API documentation for SimpleDateFormat. If the property is not present, the format defaults to the value already given in the example.

Properties

You are also able to reference any properties defined in the project as a variable. Consider the following example:

```
<project>
  ...
  <properties>
    <mavenVersion>3.0</mavenVersion>
  </properties>

  <dependencies>
    <dependency>
```

```
      <groupId>org.apache.maven</groupId>
      <artifactId>maven-artifact</artifactId>
      <version>${mavenVersion}</version>
    </dependency>
    <dependency>
      <groupId>org.apache.maven</groupId>
      <artifactId>maven-core</artifactId>
      <version>${mavenVersion}</version>
    </dependency>
  </dependencies>
  ...
</project>
```

IV. Dependencies

Dependency management is a core feature of Maven. Managing dependencies for a single project is easy. Managing dependencies for multi-module projects and applications that consist of hundreds of modules is possible. Maven helps a great deal in defining, creating, and maintaining reproducible builds with well-defined classpaths and library versions.

- Transitive Dependencies
  - Excluded/Optional Dependencies
- Dependency Scope
- Dependency Management
  - Importing Dependencies
  - Bill of Materials (BOM) POMs
- System Dependencies

Transitive Dependencies

Maven avoids the need to discover and specify the libraries that your own dependencies require by including transitive dependencies automatically.

This feature is facilitated by reading the project files of your dependencies from the remote repositories specified. In general, all dependencies of those projects are used in your project, as are any that the project inherits from its parents, or from its dependencies, and so on.

There is no limit to the number of levels that dependencies can be gathered from. A problem arises only if a cyclic dependency is discovered.

With transitive dependencies, the graph of included libraries can quickly grow quite large. For this reason, there are additional features that limit which dependencies are included:

- *Dependency mediation* - this determines what version of an artifact will be chosen when multiple versions are encountered as dependencies. Maven picks the "nearest definition". That is, it uses the version of the closest dependency to your project in the tree of dependencies. You can always guarantee a version by declaring it explicitly in your project's POM. Note that if two dependency versions are at the same depth in the dependency tree, the first declaration wins.
  - "nearest definition" means that the version used will be the closest one to your project in the tree of dependencies. Consider this tree of dependencies:

```
A
├── B
|   └── C
|       └── D 2.0
└── E
    └── D 1.0
```

In text, dependencies for A, B, and C are defined as A -> B -> C -> D 2.0 and A -> E -> D 1.0, then D 1.0 will be used when building A because the path from A to D through E is shorter. You could explicitly add a dependency to D 2.0 in A to force the use of D 2.0, as shown here:

```
A
├── B
|   └── C
|       └── D 2.0
├── E
|   └── D 1.0
|
└── D 2.0
```

- *Dependency management* - this allows project authors to directly specify the versions of artifacts to be used when they are encountered in transitive dependencies or in dependencies where no version has been specified. In the example in the preceding section a dependency was directly added to A even though it is not directly used by A. Instead, A can include D as a dependency in its dependencyManagement section and directly control which version of D is used when, or if, it is ever referenced.
- *Dependency scope* - this allows you to only include dependencies appropriate for the current stage of the build. This is described in more detail below.
- *Excluded dependencies* - If project X depends on project Y, and project Y depends on project Z, the owner of project X can explicitly exclude project Z as a dependency, using the "exclusion" element.

- *Optional dependencies* - If project Y depends on project Z, the owner of project Y can mark project Z as an optional dependency, using the "optional" element. When project X depends on project Y, X will depend only on Y and not on Y's optional dependency Z. The owner of project X may then explicitly add a dependency on Z, at her option. (It may be helpful to think of optional dependencies as "excluded by default.")

Although transitive dependencies can implicitly include desired dependencies, it is a good practice to explicitly specify the dependencies your source code uses directly. This best practice proves its value especially when the dependencies of your project change their dependencies.

For example, assume that your project A specifies a dependency on another project B, and project B specifies a dependency on project C. If you are directly using components in project C, and you don't specify project C in your project A, it may cause build failure when project B suddenly updates/removes its dependency on project C.

Another reason to directly specify dependencies is that it provides better documentation for your project: one can learn more information by just reading the POM file in your project, or by executing **mvn dependency:tree**.

Maven also provides dependency:analyze plugin goal for analyzing the dependencies: it helps making this best practice more achievable.

Dependency Scope

Dependency scope is used to limit the transitivity of a dependency and to determine when a dependency is included in a classpath.

There are 6 scopes:

- **compile**
  This is the default scope, used if none is specified. Compile dependencies are available in all classpaths of a project. Furthermore, those dependencies are propagated to dependent projects.
- **provided**
  This is much like `compile`, but indicates you expect the JDK or a container to provide the dependency at runtime. For example, when building a web application for the Java Enterprise Edition, you would set the dependency on the Servlet API and related Java EE APIs to scope `provided` because the web container provides those classes. A dependency with this scope is added to the classpath used for compilation and test, but not the runtime classpath. It is not transitive.
- **runtime**
  This scope indicates that the dependency is not required for compilation, but is for execution. Maven includes a dependency with this scope in the runtime and test classpaths, but not the compile classpath.
- **test**
  This scope indicates that the dependency is not required for normal use of the application, and is only available for the test compilation and execution phases. This scope is not transitive. Typically this scope is used for test libraries such as JUnit and Mockito. It is also used for non-test libraries such as

Apache Commons IO if those libraries are used in unit tests (src/test/java) but not in the model code (src/main/java).

- **system**
  This scope is similar to `provided` except that you have to provide the JAR which contains it explicitly. The artifact is always available and is not looked up in a repository.
- **import**
  This scope is only supported on a dependency of type `pom` in the `<dependencyManagement>` section. It indicates the dependency is to be replaced with the effective list of dependencies in the specified POM's `<dependencyManagement>` section. Since they are replaced, dependencies with a scope of `import` do not actually participate in limiting the transitivity of a dependency.

Each of the scopes (except for `import`) affects transitive dependencies in different ways, as is demonstrated in the table below. If a dependency is set to the scope in the left column, a transitive dependency of that dependency with the scope across the top row results in a dependency in the main project with the scope listed at the intersection. If no scope is listed, it means the dependency is omitted.

	compile	provided	runtime	test
compile	compile(*)	-	runtime	-
provided	provided	-	provided	-
runtime	runtime	-	runtime	-
test	test	-	test	-

(*) Note: it is intended that this should be runtime scope instead, so that all compile dependencies must be explicitly listed. However, if a library you depend on extends a class from another library, both must be available at compile time. For this reason, compile time dependencies remain as compile scope even when they are transitive.

Dependency Management

The dependency management section is a mechanism for centralizing dependency information. When you have a set of projects that inherit from a common parent, it's possible to put all information about the dependency in the common POM and have simpler references to the artifacts in the child POMs. The mechanism is best illustrated through some examples. Given these two POMs which extend the same parent:

Project A:

```
<project>
  ...
  <dependencies>
    <dependency>
      <groupId>group-a</groupId>
      <artifactId>artifact-a</artifactId>
      <version>1.0</version>
      <exclusions>
        <exclusion>
          <groupId>group-c</groupId>
          <artifactId>excluded-artifact</artifactId>
        </exclusion>
      </exclusions>
    </dependency>
    <dependency>
      <groupId>group-a</groupId>
      <artifactId>artifact-b</artifactId>
      <version>1.0</version>
      <type>bar</type>
      <scope>runtime</scope>
    </dependency>
  </dependencies>
</project>
```

Project B:

```
<project>
  ...
  <dependencies>
    <dependency>
      <groupId>group-c</groupId>
      <artifactId>artifact-b</artifactId>
      <version>1.0</version>
      <type>war</type>
      <scope>runtime</scope>
    </dependency>
    <dependency>
      <groupId>group-a</groupId>
      <artifactId>artifact-b</artifactId>
      <version>1.0</version>
      <type>bar</type>
      <scope>runtime</scope>
    </dependency>
  </dependencies>
</project>
```

These two example POMs share a common dependency and each has one non-trivial dependency. This information can be put in the parent POM like this:

```
<project>
  ...
  <dependencyManagement>
    <dependencies>
      <dependency>
        <groupId>group-a</groupId>
        <artifactId>artifact-a</artifactId>
        <version>1.0</version>

        <exclusions>
          <exclusion>
            <groupId>group-c</groupId>
            <artifactId>excluded-artifact</artifactId>
          </exclusion>
        </exclusions>

      </dependency>

      <dependency>
        <groupId>group-c</groupId>
        <artifactId>artifact-b</artifactId>
        <version>1.0</version>
        <type>war</type>
        <scope>runtime</scope>
      </dependency>

      <dependency>
        <groupId>group-a</groupId>
        <artifactId>artifact-b</artifactId>
        <version>1.0</version>
        <type>bar</type>
        <scope>runtime</scope>
      </dependency>
    </dependencies>
  </dependencyManagement>
</project>
```

Then the two child POMs become much simpler:

```
<project>
  ...
  <dependencies>
    <dependency>
      <groupId>group-a</groupId>
      <artifactId>artifact-a</artifactId>
    </dependency>

    <dependency>
      <groupId>group-a</groupId>
      <artifactId>artifact-b</artifactId>
      <!-- This is not a jar dependency, so we must specify type. -->
      <type>bar</type>
    </dependency>
  </dependencies>
</project>

<project>
  ...
  <dependencies>
    <dependency>
      <groupId>group-c</groupId>
      <artifactId>artifact-b</artifactId>
      <!-- This is not a jar dependency, so we must specify type. -->
      <type>war</type>
    </dependency>

    <dependency>
      <groupId>group-a</groupId>
      <artifactId>artifact-b</artifactId>
      <!-- This is not a jar dependency, so we must specify type. -->
      <type>bar</type>
    </dependency>
  </dependencies>
</project>
```

NOTE: In two of these dependency references, we had to specify the <type/> element. This is because the minimal set of information for matching a dependency reference against a dependencyManagement section is actually **{groupId, artifactId, type, classifier}**. In many cases, these dependencies will refer to jar artifacts with no classifier. This allows us to shorthand the identity set to **{groupId, artifactId}**, since the default for the type field is `jar`, and the default classifier is null.

A second, and very important use of the dependency management section is to control the versions of artifacts used in transitive dependencies. As an example consider these projects:

Project A:

```
<project>
 <modelVersion>4.0.0</modelVersion>
 <groupId>maven</groupId>
 <artifactId>A</artifactId>
 <packaging>pom</packaging>
 <name>A</name>
 <version>1.0</version>
 <dependencyManagement>
   <dependencies>
     <dependency>
       <groupId>test</groupId>
       <artifactId>a</artifactId>
       <version>1.2</version>
     </dependency>
     <dependency>
       <groupId>test</groupId>
       <artifactId>b</artifactId>
       <version>1.0</version>
       <scope>compile</scope>
     </dependency>
     <dependency>
       <groupId>test</groupId>
       <artifactId>c</artifactId>
       <version>1.0</version>
       <scope>compile</scope>
     </dependency>
     <dependency>
       <groupId>test</groupId>
       <artifactId>d</artifactId>
```

```
        <version>1.2</version>
      </dependency>
    </dependencies>
  </dependencyManagement>
</project>
```

Project B:

```
<project>
  <parent>
    <artifactId>A</artifactId>
    <groupId>maven</groupId>
    <version>1.0</version>
  </parent>
  <modelVersion>4.0.0</modelVersion>
  <groupId>maven</groupId>
  <artifactId>B</artifactId>
  <packaging>pom</packaging>
  <name>B</name>
  <version>1.0</version>

  <dependencyManagement>
    <dependencies>
      <dependency>
        <groupId>test</groupId>
        <artifactId>d</artifactId>
        <version>1.0</version>
      </dependency>
    </dependencies>
  </dependencyManagement>

  <dependencies>
    <dependency>
      <groupId>test</groupId>
      <artifactId>a</artifactId>
      <version>1.0</version>
      <scope>runtime</scope>
    </dependency>
    <dependency>
      <groupId>test</groupId>
      <artifactId>c</artifactId>
      <scope>runtime</scope>
    </dependency>
  </dependencies>
</project>
```

When maven is run on project B, version 1.0 of artifacts a, b, c, and d will be used regardless of the version specified in their POM.

- a and c both are declared as dependencies of the project so version 1.0 is used due to dependency mediation. Both also have runtime scope since it is directly specified.
- b is defined in B's parent's dependency management section and since dependency management takes precedence over dependency mediation for transitive dependencies, version 1.0 will be selected should it be referenced in a or c's POM. b will also have compile scope.
- Finally, since d is specified in B's dependency management section, should d be a dependency (or transitive dependency) of a or c, version 1.0 will be chosen - again because dependency management takes precedence over dependency mediation and also because the current POM's declaration takes precedence over its parent's declaration.

The reference information about the dependency management tags is available from the project descriptor reference.

Importing Dependencies

The examples in the previous section describe how to specify managed dependencies through inheritance. However, in larger projects it may be impossible to accomplish this since a project can only inherit from a single parent. To accommodate this, projects can import managed dependencies from other projects. This is accomplished by declaring a POM artifact as a dependency with a scope of "import".

Project B:

```
<project>
  <modelVersion>4.0.0</modelVersion>
  <groupId>maven</groupId>
  <artifactId>B</artifactId>
  <packaging>pom</packaging>
  <name>B</name>
  <version>1.0</version>

  <dependencyManagement>
    <dependencies>
      <dependency>
        <groupId>maven</groupId>
        <artifactId>A</artifactId>
        <version>1.0</version>
        <type>pom</type>
        <scope>import</scope>
      </dependency>
      <dependency>
        <groupId>test</groupId>
        <artifactId>d</artifactId>
        <version>1.0</version>
      </dependency>
    </dependencies>
  </dependencyManagement>

  <dependencies>
    <dependency>
      <groupId>test</groupId>
      <artifactId>a</artifactId>
      <version>1.0</version>
      <scope>runtime</scope>
    </dependency>
    <dependency>
      <groupId>test</groupId>
      <artifactId>c</artifactId>
      <scope>runtime</scope>
    </dependency>
  </dependencies>
</project>
```

Assuming A is the POM defined in the preceding example, the end result would be the same. All of A's managed dependencies would be incorporated into B except for d since it is defined in this POM.

Project X:

```
<project>
 <modelVersion>4.0.0</modelVersion>
 <groupId>maven</groupId>
 <artifactId>X</artifactId>
 <packaging>pom</packaging>
 <name>X</name>
 <version>1.0</version>

 <dependencyManagement>
   <dependencies>
     <dependency>
```

```
      <groupId>test</groupId>
      <artifactId>a</artifactId>
      <version>1.1</version>
    </dependency>
    <dependency>
      <groupId>test</groupId>
      <artifactId>b</artifactId>
      <version>1.0</version>
      <scope>compile</scope>
    </dependency>
  </dependencies>
</dependencyManagement>
</project>
```

Project Y:

```
<project>
 <modelVersion>4.0.0</modelVersion>
 <groupId>maven</groupId>
 <artifactId>Y</artifactId>
 <packaging>pom</packaging>
 <name>Y</name>
 <version>1.0</version>

 <dependencyManagement>
   <dependencies>
     <dependency>
       <groupId>test</groupId>
       <artifactId>a</artifactId>
       <version>1.2</version>
     </dependency>
     <dependency>
       <groupId>test</groupId>
       <artifactId>c</artifactId>
       <version>1.0</version>
       <scope>compile</scope>
     </dependency>
   </dependencies>
 </dependencyManagement>
</project>
```

Project Z:

```
<project>
  <modelVersion>4.0.0</modelVersion>
  <groupId>maven</groupId>
  <artifactId>Z</artifactId>
  <packaging>pom</packaging>
  <name>Z</name>
  <version>1.0</version>

  <dependencyManagement>
    <dependencies>
      <dependency>
        <groupId>maven</groupId>
        <artifactId>X</artifactId>
        <version>1.0</version>
        <type>pom</type>
        <scope>import</scope>
      </dependency>
      <dependency>
        <groupId>maven</groupId>
        <artifactId>Y</artifactId>
        <version>1.0</version>
        <type>pom</type>
        <scope>import</scope>
      </dependency>
    </dependencies>
  </dependencyManagement>
</project>
```

In the example above Z imports the managed dependencies from both X and Y. However, both X and Y contain dependency a. Here, version 1.1 of a would be used since X is declared first and a is not declared in Z's dependencyManagement.

This process is recursive. For example, if X imports another POM, Q, when Z is processed it will simply appear that all of Q's managed dependencies are defined in X.

Bill of Materials (BOM) POMs

Imports are most effective when used for defining a "library" of related artifacts that are generally part of a multiproject build. It is fairly common for one project to use one or more artifacts from these libraries. However, it has sometimes been difficult to keep the versions in the project using the artifacts in synch with the versions distributed in the library. The pattern below illustrates how a "bill of materials" (BOM) can be created for use by other projects.

The root of the project is the BOM POM. It defines the versions of all the artifacts that will be created in the library. Other projects that wish to use the library should import this POM into the dependencyManagement section of their POM.

```xml
<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>com.test</groupId>
  <artifactId>bom</artifactId>
  <version>1.0.0</version>
  <packaging>pom</packaging>
  <properties>
    <project1Version>1.0.0</project1Version>
    <project2Version>1.0.0</project2Version>
  </properties>

  <dependencyManagement>
    <dependencies>
      <dependency>
        <groupId>com.test</groupId>
        <artifactId>project1</artifactId>
        <version>${project1Version}</version>
      </dependency>
      <dependency>
        <groupId>com.test</groupId>
        <artifactId>project2</artifactId>
        <version>${project2Version}</version>
      </dependency>
    </dependencies>
  </dependencyManagement>

  <modules>
    <module>parent</module>
  </modules>
</project>
```

The parent subproject has the BOM POM as its parent. It is a normal multiproject pom.

```xml
<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>
  <parent>
    <groupId>com.test</groupId>
    <version>1.0.0</version>
    <artifactId>bom</artifactId>
  </parent>

  <groupId>com.test</groupId>
  <artifactId>parent</artifactId>
  <version>1.0.0</version>
  <packaging>pom</packaging>

  <dependencyManagement>
    <dependencies>
      <dependency>
        <groupId>log4j</groupId>
        <artifactId>log4j</artifactId>
        <version>1.2.12</version>
      </dependency>
      <dependency>
        <groupId>commons-logging</groupId>
        <artifactId>commons-logging</artifactId>
        <version>1.1.1</version>
      </dependency>
    </dependencies>
  </dependencyManagement>
  <modules>
    <module>project1</module>
    <module>project2</module>
  </modules>
</project>
```

Next are the actual project POMs.

```
<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>
  <parent>
    <groupId>com.test</groupId>
    <version>1.0.0</version>
    <artifactId>parent</artifactId>
  </parent>
  <groupId>com.test</groupId>
  <artifactId>project1</artifactId>
  <version>${project1Version}</version>
  <packaging>jar</packaging>

  <dependencies>
    <dependency>
      <groupId>log4j</groupId>
      <artifactId>log4j</artifactId>
    </dependency>
  </dependencies>
</project>

<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>
  <parent>
    <groupId>com.test</groupId>
    <version>1.0.0</version>
    <artifactId>parent</artifactId>
  </parent>
  <groupId>com.test</groupId>
  <artifactId>project2</artifactId>
  <version>${project2Version}</version>
  <packaging>jar</packaging>

  <dependencies>
    <dependency>
      <groupId>commons-logging</groupId>
      <artifactId>commons-logging</artifactId>
    </dependency>
  </dependencies>
</project>
```

The project that follows shows how the library can now be used in another project without having to specify the dependent project's versions.

```
<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>com.test</groupId>
  <artifactId>use</artifactId>
  <version>1.0.0</version>
  <packaging>jar</packaging>

  <dependencyManagement>
    <dependencies>
      <dependency>
        <groupId>com.test</groupId>
        <artifactId>bom</artifactId>
        <version>1.0.0</version>
        <type>pom</type>
        <scope>import</scope>
      </dependency>
    </dependencies>
  </dependencyManagement>
  <dependencies>
    <dependency>
      <groupId>com.test</groupId>
      <artifactId>project1</artifactId>
    </dependency>
    <dependency>
      <groupId>com.test</groupId>
      <artifactId>project2</artifactId>
    </dependency>
  </dependencies>
</project>
```

Finally, when creating projects that import dependencies, beware of the following:

- Do not attempt to import a POM that is defined in a submodule of the current POM. Attempting to do that will result in the build failing since it won't be able to locate the POM.
- Never declare the POM importing a POM as the parent (or grandparent, etc) of the target POM. There is no way to resolve the circularity and an exception will be thrown.

- When referring to artifacts whose POMs have transitive dependencies, the project needs to specify versions of those artifacts as managed dependencies. Not doing so results in a build failure since the artifact may not have a version specified. (This should be considered a best practice in any case as it keeps the versions of artifacts from changing from one build to the next).

System Dependencies

Important note: This is deprecated.

Dependencies with the scope *system* are always available and are not looked up in repository. They are usually used to tell Maven about dependencies which are provided by the JDK or the VM. Thus, system dependencies are especially useful for resolving dependencies on artifacts which are now provided by the JDK, but were available as separate downloads earlier. Typical examples are the JDBC standard extensions or the Java Authentication and Authorization Service (JAAS).

A simple example would be:

```
<project>
  ...
  <dependencies>
    <dependency>
      <groupId>javax.sql</groupId>
      <artifactId>jdbc-stdext</artifactId>
      <version>2.0</version>
      <scope>system</scope>
      <systemPath>${java.home}/lib/rt.jar</systemPath>
    </dependency>
  </dependencies>
  ...
</project>
```

If your artifact is provided by the JDK's `tools.jar`, the system path would be defined as follows:

```
<project>
  ...
  <dependencies>
    <dependency>
      <groupId>sun.jdk</groupId>
      <artifactId>tools</artifactId>
      <version>1.5.0</version>
      <scope>system</scope>
      <systemPath>${java.home}/../lib/tools.jar</systemPath>
```

```
   </dependency>
  </dependencies>
  ...
</project>
```

V. Maven Goals

The Maven build follows a specific life cycle to deploy and distribute the target project.

There are three built-in life cycles:

- default: the main life cycle as it's responsible for project deployment
- clean: to clean the project and remove all files generated by the previous build
- site: to create the project's site documentation

Each life cycle consists of a sequence of phases. The *default* build life cycle consists of 23 phases as it's the main build lifecycle.

On the other hand, *clean* life cycle consists of 3 phases, while the *site* lifecycle is made up of 4 phases.

Maven Phase

A Maven phase represents a stage in the Maven build lifecycle. Each phase is responsible for a specific task.

Here are some of the most important phases in the *default* build lifecycle:

- *validate:* check if all information necessary for the build is available
- *compile:* compile the source code
- *test-compile:* compile the test source code
- *test:* run unit tests
- *package:* package compiled source code into the distributable format (jar, war, …)
- *integration-test:* process and deploy the package if needed to run integration tests
- *install:* install the package to a local repository
- *deploy:* copy the package to the remote repository

For the full list of each lifecycle's phases, check out the Maven Reference.

Phases are executed in a specific order. This means that if we run a specific phase using the command:

```
mvn <PHASE>
```

This won't only execute the specified phase but all the preceding phases as well.

For example, if we run the *deploy* phase – which is the last phase in the *default* build lifecycle – that will execute all phases before the *deploy* phase as well, which is the entire *default* lifecycle:

```
mvn deploy
```

Maven Goal

Each phase is a sequence of goals, and each goal is responsible for a specific task.

When we run a phase – all goals bound to this phase are executed in order.

Here are some of the phases and default goals bound to them:

- *compiler:compile* – the *compile* goal from the *compiler* plugin is bound to the *compile* phase
- *compiler:testCompile* is bound to the *test-compile* phase
- *surefire:test* is bound to *test* phase
- *install:install* is bound to *install* phase
- *jar:jar* and *war:war* is bound to *package* phase

We can list all goals bound to a specific phase and their plugins using the command:

```
mvn help:describe -Dcmd=PHASENAME
```

For example, to list all goals bound to the *compile* phase, we can run:

```
mvn help:describe -Dcmd=compile
```

And get the sample output:

```
compile' is a phase corresponding to this plugin:
org.apache.maven.plugins:maven-compiler-plugin:3.1:compile
```

Which, as mentioned above, means the *compile* goal from *compiler* plugin is bound to the *compile* phase.

Maven Plugin

A Maven plugin is a group of goals. However, these goals aren't necessarily all bound to the same phase.

For example, here's a simple configuration of the Maven Failsafe plugin which is responsible for running integration tests:

```
<build>
   <plugins>
      <plugin>
         <artifactId>maven-failsafe-plugin</artifactId>
         <version>${maven.failsafe.version}</version>
         <executions>
            <execution>
               <goals>
                  <goal>integration-test</goal>
                  <goal>verify</goal>
               </goals>
            </execution>
         </executions>
      </plugin>
   </plugins>
</build>
```

As we can see, the Failsafe plugin has two main goals configured here:

- *integration-test*: run integration tests
- *verify*: verify all integration tests passed

We can use the following command to **list all goals in a specific plugin**:

```
mvn <PLUGIN>:help
```

For example, to list all goals in the Failsafe plugin:

```
mvn failsafe:help
```

And the output of this will be:

```
This plugin has 3 goals:

failsafe:help
  Display help information on maven-failsafe-plugin.
  Call mvn failsafe:help -Ddetail=true -Dgoal=<goal-name> to display parameter
  details.

failsafe:integration-test
  Run integration tests using Surefire.

failsafe:verify
  Verify integration tests ran using Surefire.
```

To run a specific goal, without executing its entire phase (and the preceding phases) we can use the command:

```
mvn <PLUGIN>:<GOAL>
```

For example, to run *integration-test* goal from Failsafe plugin, we need to run:

```
mvn failsafe:integration-test
```

Building a Maven Project

To build a Maven project, we need to execute one of the life cycles by running one of their phases:

```
mvn deploy
```

This will execute the entire *default* lifecycle. Alternatively, we can stop at the *install* phase:

```
mvn install
```

But usually we'll use the command:

```
mvn clean install
```

To clean the project first – by running the *clean* lifecycle – before the new build.

We can also run only a specific goal of the plugin:

```
mvn compiler:compile
```

Note that if we tried to build a Maven project without specifying a phase or a goal, that will cause the error:

```
[ERROR] No goals have been specified for this build. You must specify a valid lifecycle phase or a goal
```

VI.SCOPE:.

There are 6 scopes:

- **compile**
  This is the default scope, used if none is specified. Compile dependencies are available in all classpaths of a project. Furthermore, those dependencies are propagated to dependent projects.
- **provided**
  This is much like `compile`, but indicates you expect the JDK or a container to provide the dependency at runtime. For example, when building a web application for the Java Enterprise Edition, you would set the dependency on the Servlet API and related Java EE APIs to scope `provided` because the web container provides those classes. A dependency with this scope is added to the classpath used for compilation and test, but not the runtime classpath. It is not transitive.
- **runtime**
  This scope indicates that the dependency is not required for compilation, but is for execution. Maven includes a dependency with this scope in the runtime and test classpaths, but not the compile classpath.
- **test**
  This scope indicates that the dependency is not required for normal use of the application, and is only available for the test compilation and execution phases. This scope is not transitive. Typically this scope is used for test libraries such as JUnit and Mockito. It is also used for non-test libraries such as Apache Commons IO if those libraries are used in unit tests (src/test/java) but not in the model code (src/main/java).
- **system**
  This scope is similar to `provided` except that you have to provide the JAR which contains it explicitly. The artifact is always available and is not looked up in a repository.
- **import**
  This scope is only supported on a dependency of type `pom` in the `<dependencyManagement>` section. It indicates the dependency is to be replaced with the

effective list of dependencies in the specified POM's `<dependencyManagement>` section. Since they are replaced, dependencies with a scope of `import` do not actually participate in limiting the transitivity of a dependency.

VII. The Compiler Plugin:

Apache Maven Compiler Plugin

The Compiler Plugin is used to compile the sources of your project. Since 3.0, the default compiler is javax.tools.JavaCompiler (if you are using java 1.6) and is used to compile Java sources. If you want to force the plugin using javac, you must configure the plugin option forceJavacCompilerUse.

Also note that at present the default source setting is 1.6 and the default target setting is 1.6, independently of the JDK you run Maven with. You are highly encouraged to change these defaults by setting source and target as described in Setting the -source and -target of the Java Compiler.

Other compilers than javac can be used and work has already started on AspectJ, .NET, and C#.

NOTE: *To know more about the JDK javac, please see: http://download.oracle.com/javase/6/docs/technotes/tools/windows/javac.html.*

Goals Overview

The Compiler Plugin has two goals. Both are already bound to their proper phases within the Maven Lifecycle and are therefore, automatically executed during their respective phases.

- compiler:compile is bound to the compile phase and is used to compile the main source files.
- compiler:testCompile is bound to the test-compile phase and is used to compile the test source files.

Usage

General instructions on how to use the Compiler Plugin can be found on the usage page. Some more specific use cases are described in the examples given below.

In case you still have questions regarding the plugin's usage, please have a look at the FAQ and feel free to contact the user mailing list. The posts to the mailing list are archived and could already contain the answer to your question as part of an older thread. Hence, it is also worth browsing/searching the mail archive.

If you feel like the plugin is missing a feature or has a defect, you can fill a feature request or bug report in our issue tracker. When creating a new issue, please provide a comprehensive description of your concern. Especially for fixing bugs it is crucial that the developers can reproduce your problem. For this reason, entire debug logs, POMs or most preferably little demo projects attached to the issue are very much appreciated. Of course, patches are welcome, too. Contributors can check out the project from our source repository and will find supplementary information in the guide to helping with Maven.

Compiling Sources Using A Different JDK

Using Maven Toolchains

The preferable way to use a different JDK is to use the toolchains mechanism. During the build of a project, Maven, without toolchains, will use the JDK to perform various steps, like compiling the Java sources, generate the Javadoc, run unit tests or sign JARs. Each of those plugins need a tool of the JDK to operate: javac, javadoc, jarsigner, etc. A toolchains is a way to specify the path to the JDK to use for all of those plugins in a centralized manner, independant from the one running Maven itself.

To set this up, refer to the Guide to Using Toolchains, which makes use of the Maven Toolchains Plugin.

With the maven-toolchains-plugin you configure 1 default JDK toolchain for all related maven-plugins. Since maven-compiler-plugin 3.6.0 when using with Maven 3.3.1+ it is also possible to give the plugin its own toolchain, which can be useful in case of different JDK calls per execution block (e.g. the test sources require a different compiler compared to the main sources).

Configuring the Compiler Plugin

Outside of a toolchains, it is still possible to tell the Compiler Plugin the specific JDK to use during compilation. Note that such configuration will be specific to this plugin, and will not affect others.

The compilerVersion parameter can be used to specify the version of the compiler that the plugin will use. However, you also need to set fork to true for this to work. For example:

```
<project>
  [...]
  <build>
    [...]
    <plugins>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-compiler-plugin</artifactId>
        <version>3.8.1</version>
        <configuration>
          <verbose>true</verbose>
          <fork>true</fork>
          <executable><!-- path-to-javac --></executable>
          <compilerVersion>1.3</compilerVersion>
        </configuration>
      </plugin>
    </plugins>
    [...]
  </build>
  [...]
</project>
```

To avoid hard-coding a filesystem path for the executable, you can use a property. For example:

```
            <executable>${JAVA_1_4_HOME}/bin/javac</executable>
```

Each developer then defines this property in settings.xml, or sets an environment variable, so that the build remains portable.

```
<settings>
  [...]
  <profiles>
    [...]
    <profile>
      <id>compiler</id>
        <properties>
          <JAVA_1_4_HOME>C:\Program Files\Java\j2sdk1.4.2_09</JAVA_1_4_HOME>
        </properties>
    </profile>
  </profiles>
  [...]
  <activeProfiles>
    <activeProfile>compiler</activeProfile>
  </activeProfiles>
</settings>
```

VIII.Apache Maven Source Plugin

The Source Plugin creates a jar archive of the source files of the current project. The jar file is, by default, created in the project's target directory.

Important Hint

Starting with version 3.0.0 of the plugin all properties which could be used via command line have been named based on the following schema **maven.source.***. Further details can be found in the goal documentations.

Goals Overview

The Source Plugin has five goals:

- source:aggregate aggregrates sources for all modules in an aggregator project.
- source:jar is used to bundle the main sources of the project into a jar archive.

- source:test-jar on the other hand, is used to bundle the test sources of the project into a jar archive.
- source:jar-no-fork is similar to **jar** but does not fork the build lifecycle.
- source:test-jar-no-fork is similar to **test-jar** but does not fork the build lifecycle.

Usage

General instructions on how to use the Source Plugin can be found on the usage page. Some more specific use cases are described in the examples given below.

In case you still have questions regarding the plugin's usage, please have a look at the FAQ and feel free to contact the user mailing list. The posts to the mailing list are archived and could already contain the answer to your question as part of an older thread. Hence, it is also worth browsing/searching the mail archive.

If you feel like the plugin is missing a feature or has a defect, you can fill a feature request or bug report in our issue tracker. When creating a new issue, please provide a comprehensive description of your concern. Especially for fixing bugs it is crucial that the developers can reproduce your problem. For this reason, entire debug logs, POMs or most preferably little demo projects attached to the issue are very much appreciated. Of course, patches are welcome, too. Contributors can check out the project from our source repository and will find supplementary information in the guide to helping with Maven

Configuring Source Plugin

To customize the plugin, you can change its configuration parameters in your POM, as shown below:

```xml
<project>
  ...
  <build>
    <plugins>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-source-plugin</artifactId>
        <version>3.2.0</version>
        <configuration>
          <outputDirectory>/absolute/path/to/the/output/directory</outputDirectory>
          <finalName>filename-of-generated-jar-file</finalName>
          <attach>false</attach>
        </configuration>
      </plugin>
    </plugins>
  </build>
  ...
</project>
```

The generated jar file will be named by the value of the finalName plus "-sources" if it is the main sources. Otherwise, it would be finalName plus "-test-sources" if it is the test sources. It will be generated in the specified outputDirectory. The attach parameter specifies whether the java sources will be attached to the artifact list of the project.

Apache Maven JAR Plugin

This plugin provides the capability to build jars. If you like to sign jars please use the Maven Jarsigner Plugin.

Goals Overview

- jar:jar create a jar file for your project classes inclusive resources.
- jar:test-jar create a jar file for your project test classes .

Major Version Upgrade to version 3.0.0

Please note that the following parameter has been completely removed from the plugin configuration:

- useDefaultManifestFile

If you need to define your own **MANIFEST.MF** file you can simply achieve that via Maven Archiver configuration like in the following example:

```
<project>
  ...
  <build>
    <plugins>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-jar-plugin</artifactId>
        <version>3.2.0</version>
        <configuration>
          <archive>
            <manifestFile>${project.build.outputDirectory}/META-INF/MANIFEST.MF</manifestFile>
          </archive>
        </configuration>
        ...
      </plugin>
    </plugins>
  </build>
  ...
```

```
</project>
```

Usage

General instructions on how to use the JAR Plugin can be found on the usage page. Some more specific use cases are described in the examples given below.

In case you still have questions regarding the plugin's usage, please have a look at the FAQ and feel free to contact the user mailing list. The posts to the mailing list are archived and could already contain the answer to your question as part of an older thread. Hence, it is also worth browsing/searching the mail archive.

If you feel like the plugin is missing a feature or has a defect, you can fill a feature request or bug report in our issue tracker. When creating a new issue, please provide a comprehensive description of your concern. Especially for fixing bugs it is crucial that the developers can reproduce your problem. For this reason, entire debug logs, POMs or most preferably little demo projects attached to the issue are very much appreciated. Of course, patches are welcome, too. Contributors can check out the project from our source repository and will find supplementary information in the guide to helping with Maven

Manifest customization

The Default Manifest

The default contents of the manifest is described in the documentation for Maven Archiver.

Starting with version 2.1, the maven-jar-plugin uses Maven Archiver 3.5.0. This means that it no longer creates the Specification and Implementation details in the manifest by default. If you want them you have to say so explicitly in your plugin configuration. This is also is described in the documentation for Maven Archiver.

Customization the Manifest

The default manifest can be altered with the archive configuration element. Below you will find some of the configuration options that are available. For more info see the Maven Archiver reference.

```
<project>
  ...
  <build>
    <plugins>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-jar-plugin</artifactId>
        <version>3.2.0</version>
        <configuration>
          <archive>
```

```
            <index>true</index>
            <manifest>
              <addClasspath>true</addClasspath>
            </manifest>
            <manifestEntries>
              <mode>development</mode>
              <url>${project.url}</url>
              <key>value</key>
            </manifestEntries>
          </archive>
        </configuration>
        ...
      </plugin>
    </plugins>
  </build>
  ...
</project>
```

Available Plugins

Maven is - at its heart - a plugin execution framework; all work is done by plugins. Looking for a specific goal to execute? This page lists the core plugins and others. There are the build and the reporting plugins:

- **Build plugins** will be executed during the build and they should be configured in the `<build/>` element from the POM.
- **Reporting plugins** will be executed during the site generation and they should be configured in the `<reporting/>` element from the POM. Because the result of a Reporting plugin is part of the generated site, Reporting plugins should be both internationalized and localized. You can read more about the localization of our plugins and how you can help.

Supported By The Maven Project

To see the most up-to-date list browse the Maven repository, specifically the `org/apache/maven/plugins` subfolder. *(Plugins are organized according to a directory structure that resembles the standard Java package naming convention)*

Plugin	Type *	Version	Release Date	Description	Source Repository	Issue Tracking
Core plugins				**Plugins corresponding to default core phases (ie. clean, compile). They may have multiple goals as well.**		
clean	B	3.1.0	2018-04-14	Clean up after the build.	Git / GitHub	Jira MCLEAN
compiler	B	3.8.1	2019-04-28	Compiles Java sources.	Git / GitHub	Jira MCOMPILER
deploy	B	3.0.0-M1	2018-09-23	Deploy the built artifact to the remote repository.	Git / GitHub	Jira MDEPLOY
failsafe	B	3.0.0-M5	2020-06-17	Run the JUnit integration tests in an isolated classloader.	Git / GitHub	Jira SUREFIRE
install	B	3.0.0-M1	2018-09-23	Install the built artifact into the local repository.	Git / GitHub	Jira MINSTALL

resources	B	3.2.0	2020-08-11	Copy the resources to the output directory for including in the JAR.	Git / GitHub	Jira MRESOURCES
site	B	3.9.1	2020-06-24	Generate a site for the current project.	Git / GitHub	Jira MSITE
surefire	B	3.0.0-M5	2020-06-17	Run the JUnit unit tests in an isolated classloader.	Git / GitHub	Jira SUREFIRE
verifier	B	1.1	2015-04-14	Useful for integration tests - verifies the existence of certain conditions.	Git / GitHub	Jira MVERIFIER
Packaging types/tools				**These plugins relate to packaging respective artifact types.**		
ear	B	3.2.0	2021-01-03	Generate an EAR from the current project.	Git / GitHub	Jira MEAR
ejb	B	3.1.0	2020-06-12	Build an EJB (and optional client) from the current project.	Git / GitHub	Jira MEJB
jar	B	3.2.0	2019-11-03	Build a JAR from the current project.	Git / GitHub	Jira MJAR
rar	B	2.4	2014-09-08	Build a RAR from the current project.	Git / GitHub	Jira MRAR

war	B	3.3.1	2020-07-13	Build a WAR from the current project.	Git / GitHub	Jira MWAR
app-client/acr	B	3.1.0	2018-06-19	Build a JavaEE application client from the current project.	Git / GitHub	Jira MACR
shade	B	3.2.4	2020-05-31	Build an Uber-JAR from the current project, including dependencies.	Git / GitHub	Jira MSHADE
source	B	3.2.1	2019-12-21	Build a source-JAR from the current project.	Git / GitHub	Jira MSOURCES
jlink	B	3.1.0	2020-12-28	Build Java Run Time Image.	Git / GitHub	Jira MJLINK
jmod	B	3.0.0-alpha-1	2017-09-17	Build Java JMod files.	Git / GitHub	Jira MJMOD
Reporting plugins				**Plugins which generate reports, are configured as reports in the POM and run under the site generation lifecycle.**		
changelog	R	2.3	2014-06-24	Generate a list of recent changes from your SCM.	Git / GitHub	Jira MCHANGELOG
changes	B+R	2.12.1	2016-11-01	Generate a report from an issue tracker or a change document.	Git / GitHub	Jira MCHANGES

checkstyle	B+R	3.1.2	2021-01-30	Generate a Checkstyle report.	Git / GitHub	Jira MCHECKSTYLE
doap	B	1.2	2015-03-17	Generate a Description of a Project (DOAP) file from a POM.	Git / GitHub	Jira MDOAP
docck	B	1.1	2015-04-03	Documentation checker plugin.	Git / GitHub	Jira MDOCCK
javadoc	B+R	3.2.0	2020-03-16	Generate Javadoc for the project.	Git / GitHub	Jira MJAVADOC
jdeps	B	3.1.2	2019-06-12	Run JDK's JDeps tool on the project.	Git / GitHub	Jira MJDEPS
jxr	R	3.0.0	2018-09-25	Generate a source cross reference.	Git / GitHub	Jira JXR
linkcheck	R	1.2	2014-10-08	Generate a Linkcheck report of your project's documentation.	Git / GitHub	Jira MLINKCHECK
pmd	B+R	3.14.0	2020-10-24	Generate a PMD report.	Git / GitHub	Jira MPMD
project-info-reports	R	3.1.1	2020-08-30	Generate standard project reports.	Git / GitHub	Jira MPIR

surefire-report	R	3.0.0-M5	2020-06-17	Generate a report based on the results of unit tests.	Git / GitHub	Jira SUREFIRE
Tools				**These are miscellaneous tools available through Maven by default.**		
antrun	B	3.0.0	2020-04-15	Run a set of ant tasks from a phase of the build.	Git / GitHub	Jira MANTRUN
artifact	B	3.0.0	2021-02-20	Manage artifacts tasks like buildinfo.	Git / GitHub	Jira MARTIFACT
archetype	B	3.2.0	2020-07-21	Generate a skeleton project structure from an archetype.	Git / GitHub	Jira ARCHETYPE
assembly	B	3.3.0	2020-04-30	Build an assembly (distribution) of sources and/or binaries.	Git / GitHub	Jira MASSEMBLY
dependency	B+R	3.1.2	2020-03-07	Dependency manipulation (copy, unpack) and analysis.	Git / GitHub	Jira MDEP
enforcer	B	3.0.0-M3	2019-11-23	Environmental constraint checking (Maven Version, JDK etc), User Custom Rule Execution.	Git / GitHub	Jira MENFORCER
gpg	B	1.6	2015-01-19	Create signatures for the artifacts and poms.	Git / GitHub	Jira MGPG

help	B	3.2.0	2019-04-16	Get information about the working environment for the project.	Git / GitHub	Jira MPH
invoker	B+R	3.2.2	2021-02-20	Run a set of Maven projects and verify the output.	Git / GitHub	Jira MINVOKER
jarsigner	B	3.0.0	2018-11-06	Signs or verifies project artifacts.	Git / GitHub	Jira MJARSIGNER
jdeprscan	B	3.0.0-alpha-1	2017-11-15	Run JDK's JDeprScan tool on the project.	Git / GitHub	Jira MJDEPRSCAN
patch	B	1.2	2015-03-09	Use the gnu patch tool to apply patch files to source code.	Git / GitHub	Jira MPATCH
pdf	B	1.4	2017-12-28	Generate a PDF version of your project's documentation.	Git / GitHub	Jira MPDF
plugin	B+R	3.6.0	2018-11-01	Create a Maven plugin descriptor for any mojos found in the source tree, to include in the JAR.	Git / GitHub	Jira MPLUGIN
release	B	3.0.0-M4	2021-04-16	Release the current project - updating the POM and tagging in the SCM.	Git / GitHub	Jira MRELEASE

remote-resources	B	1.7.0	2020-01-21	Copy remote resources to the output directory for inclusion in the artifact.	Git / GitHub	Jira MRRESOURCES
scm	B	1.11.2	2019-03-21	Execute SCM commands for the current project.	Git / GitHub	Jira SCM
scm-publish	B	3.1.0	2020-12-26	Publish your Maven website to a scm location.	Git / GitHub	Jira MSCMPUB
scripting	B	3.0.0	2021-03-01	The Maven Scripting Plugin wraps the Scripting API according to JSR223.	Git / GitHub	Jira MSCRIPTING
stage	B	1.0	2015-03-03	Assists with release staging and promotion.	Git / GitHub	Jira MSTAGE
toolchains	B	3.0.0	2019-06-16	Allows to share configuration across plugins.	Git / GitHub	Jira MTOOLCHAINS
wrapper	B	3.0.2	2021-04-08	Download and unpack the maven wrapper distribution (works only with Maven 4)	Git / GitHub	Jira MWRAPPER

* **B**uild or **R**eporting plugin

There are also some sandbox plugins into our source repository.

Previous archived versions of plugins reference documentations are located here.

Retired

Plugin	Type*	Version	Retired Date	Description
ant	B	2.4	2019-06-02	Generate an Ant build file for the project.
eclipse	B	2.10	2015-10-07	Generate an Eclipse project files for the current project.
idea	B	2.2.1	2013-07-26	Create/update an IDEA workspace for the current project (individual modules are created as IDEA modules)
one	B	1.3	2013-07-30	A plugin for interacting with legacy Maven 1.x repositories and builds.
reactor	B	1.1	2014-03-24	Build a subset of interdependent projects in a reactor (Maven 2 only).
repository	B	2.4	2019-04-30	Plugin to help with repository-based tasks.

Outside The Maven Land

At MojoHaus (formerly known as codehaus.org)

There are also many plug-ins available at the MojoHaus project at GitHub.

Here are a few common ones:

Plugin	Description
animal-sniffer	Build signatures of APIs (JDK for example) and checks your classes against them.
build-helper	Attach extra artifacts and source folders to build.
castor	Generate sources from an XSD using Castor.
clirr	Compare binaries or sources for compatibility using Clirr
javacc	Generate sources from a JavaCC grammar.
jdepend	Generate a report on code metrics using JDepend.
nar-maven-plugin	Compiles C, C++, Fortran for different architectures.
native	Compiles C and C++ code with native compilers.
sql	Executes SQL scripts from files or inline.
taglist	Generate a list of tasks based on tags in your code.
versions	Manage versions of your project, its modules, dependencies and plugins.

MCQ

1. Which of the following is not type of Maven Repository?

- **A.** Local
- **B.** Remote
- **C.** Maven Central
- **D.** Maven Local

Answer: Option D

2. Which of the following command removes the target directory with all the build data before starting the build process?

- **A.** mvn clean
- **B.** mvn build
- **C.** mvn compile
- **D.** mvn site

Answer: Option A

3. ____________ is an ExpectationErrorTranslator that doesn't do any translation.

- **A.** AssertionErrorTranslator
- **B.** CamelCaseNamingScheme
- **C.** CurrentStateMatcher
- **D.** IdentityExpectationErrorTranslator

Answer: Option D

4. Which of the following is not a maven goal?

- **A.** clean
- **B.** package
- **C.** install
- **D.** debug

Answer: Option D

5. Which of the following is not a dependency scope in Maven?

- **A.** Compile
- **B.** Test
- **C.** System
- **D.** Export

Answer: Option D

MAVEN with TDD

1. The ___________ package contains plugins that make it easier to use jMock with legacy code.
a)org.jmock.api
b)org.jmock.lib.action
c)org.jmock.lib.script
d) org.jmock.lib.legacy

Answer:d
Explanation: org.jmock.lib.legacy contains several plugins that make it suitable to use jMock with legacy code.

2. ______________ class enables to imposterise abstract and concrete classes without calling the constructors of the mocked class.
a)ClassImposteriser
b)Imposteriser
c)ImposterisingClass
d) Imposter

Answer: a
Explanation: The ClassImposteriser implements Imposteriser interface.

3. _________ method reports if the Imposteriser is able to imposterise a given type.
a) canImposterise(Class<?> type)
b) Imposterise(Class<?> type)
c) imposterise(Invokable mockObject, Class<T> mockedType, Class<?>… ancilliaryTypes)
d) imposter()

4. ____________ creates an imposter for a given type that forwards Invocations to an Invokable object.
a) canImposterise(Class<?> type)
b) Imposterise(Class<?> type)
c) imposterise(Invokable mockObject, Class<T> mockedType, Class<?>… ancilliaryTypes)
d) imposter()

Answer: c
Explanation: imposterise(Invokable mockObject, Class<T> mockedType, Class<?>… ancilliaryTypes) returns a new imposter. The imposter must implement the mockedType and all the ancialliaryTypes.

5. The mockObject parameter of imposterise() is the class representing the static type of the imposter.
a) True
b) False

Answer: b
Explanation: The mockType parameter of imposterise() is the class representing the static type of the imposter.

6. The ancilliaryTypes parameter of the imposterise() function must all be interfaces.
a) True
b) False

Answer:a
Explanation: The types must all be interfaces because Java only allows single inheritance of classes.

7. The ___________ package contains plugins that make it easier to write custom actions by scripting their behaviour with BeanShell.
a)org.jmock.api
b)org.jmock.lib.action
c)org.jmock.lib.script
d) org.jmock.lib.legacy

Answer: c
Explanation: org.jmock.lib.script is the package containing plugins to write custom actions by scripting.

8. ____________ class is an Action that executes a BeanShell script.
a)ScriptedAction
b)Scripted
c)Action
d) ScriptedActionClass

Answer:a
Explanation: ScriptedAction class makes it easy to implement custom actions.

9. _______________ method performs an action in response to an invocation.
a)describeTo(Descriptiondescription)
b)invoke(Invocationinvocation)
c)perform(Stringscript)
d) where(String name, Object value)

Answer:b
Explanation: invoke(Invocation invocation) returns the result of the invocation, if not throwing an exception.

10. The invoke method throws which exception?
a)ArrayIndexOutOfBounds
b)StringIndexOutOfBounds
c)Throwable
d) NullPointer

Answer:c
Explanation: Any checked exception thrown must be in the throws list of the invoked method.

Maven Projects

https://maven.apache.org/archetype/index.html

https://maven.apache.org/resolver/index.html

https://maven.apache.org/doxia/index.html

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

https://maven.apache.org/plugins/index.html

https://maven.apache.org/plugin-testing/index.html

https://codefresh.io/howtos/using-docker-maven-maven-docker/