**Must and read to know the maven..please below url…..**

[**https://maven.apache.org/guides/getting-started/maven-in-five-minutes.html**](https://maven.apache.org/guides/getting-started/maven-in-five-minutes.html)

**1.how to make a directory in maven ?**

**Ans : maven is an pro**ject management tool when we run mvn command it looks for pom file, what are the resources described in pom it will execute those things..

We have to provide a mkdir task in pom file.

**2.tell me about setting.xml ?**

**Ans :**

|  |  |
| --- | --- |
|  | settings.xml is a user preferences. It located in main Maven directory (usually $HOME/.m2) and holds our own settings, like listings for non-public repositories, usernames, and other personalized configuration.. |

**3. tell me about pom.xml ?**

**Ans :** pom stands for project object model…. And it is a project preferences..and it is an xml file which holds the information and configuration details about project

pom.xml is the control file for each Maven project or module. It tells Maven which dependencies the project needs, what processing to apply to build it, and how to package it when it's ready. 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.

What information does POM contain?

POM contains the some of the following configuration information −

* project dependencies
* plugins
* goals
* build profiles
* project version
* developers
* mailing list

|  |  |  |  |
| --- | --- | --- | --- |
|  | [difference between scope “import” and “pom” type dependency](https://stackoverflow.com/questions/11778276/difference-between-scope-import-and-pom-type-dependency) ? **Ans :**   |  |  | | --- | --- | |  | You can only import managed dependencies. This means you can only import other POMs into the dependencyManagement section of your project's POM. i.e.  ...  <dependencyManagement>  <dependencies>  <dependency>  <groupId>other.pom.group.id</groupId>  <artifactId>other-pom-artifact-id</artifactId>  <version>SNAPSHOT</version>  <scope>import</scope>  <type>pom</type>  </dependency>  </dependencies>  </dependencyManagement>  ...  What then happens is that all the dependencies defined in the dependencyManagement section of the other-pom-artifact-id are included in your POM's dependencyManagement section. You can then reference these dependencies in the dependency section of your POM (and all of its child POMs) without having to include a version etc.  However if in your POM you simply define a normal dependency to other-pom-artifact-id then all dependencies from the dependency section of the other-pom-artifact-id are included transitively in your project - however the dependencies defined in the dependencyManagement section of the other-pom-artifact-id are not included at all.  So basically the two different mechanisms are used for importing/including the two different types of dependencies (managed dependencies and normal dependencies).  There is a good page on the maven website, which can explain this far better than I can, [Dependency Management in Maven](http://maven.apache.org/guides/introduction/introduction-to-dependency-mechanism.html#Dependency_Management) and it also contains specific information on [importing dependencies](http://maven.apache.org/guides/introduction/introduction-to-dependency-mechanism.html#Importing_Dependencies). |   [**differences between dependencymanagement and dependencies in maven**](https://stackoverflow.com/questions/2619598/differences-between-dependencymanagement-and-dependencies-in-maven) **?**  **Ans :**  [Dependency Management](http://maven.apache.org/guides/introduction/introduction-to-dependency-mechanism.html#Dependency_Management) allows to consolidate and centralize the management of dependency versions without adding dependencies which are inherited by all children. This is especially useful when you have a set of projects (i.e. more than one) that inherits a common parent.  Another extremely important use case of dependencyManagement is the control of versions of artifacts used in transitive dependencies. This is hard to explain without an example. Luckily, this is illustrated in the documentation.  **For more idea :**  In the parent POM, the main difference between the <dependencies> and <dependencyManagement>is this:  Artifacts specified in the <dependencies> section will ALWAYS be included as a dependency of the child module(s).  Artifacts specified in the <dependencyManagement> section, will only be included in the child module if they were also specified in the <dependencies> section of the child module itself. Why is it good you ask? because you specify the version and/or scope in the parent, and you can leave them out when specifying the dependencies in the child POM. This can help you use unified versions for dependencies for child modules, without specifying the version in each child module. |

**What is the scope dependency ?**

**Ans :**

The <scope> element can take 6 values: *compile*, *provided*, *runtime*, *test*, *system* and *import*.

This scope is used to limit the transitivity of a dependency, and also to affect the classpath used for various build tasks.

**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. This scope is only available on the compilation and test classpath, and is not transitive.

**Runtime :**

This scope indicates that the dependency is not required for compilation, but is for execution. It is 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.

**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** (only available in Maven 2.0.9 or later) :

This scope is only used on a dependency of type pom in the section. It indicates that the specified POM should be replaced with the dependencies in that POM's section. Since they are replaced, dependencies with a scope of import do not actually participate in limiting the transitivity of a dependency.

**For more idea :**

Compile means that you need the JAR for compiling and running the app. For a web application, as an example, the JAR will be placed in the WEB-INF/lib directory. Provided means that you need the JAR for compiling, but at run time there is already a JAR provided by the environment so you don't need it packaged with your app. For a web app, this means that the JAR file will not be placed into the WEB-INF/lib directory. For a web app, if the app server already provides the JAR (or its functionality), then use "provided" otherwise use "compile"

**When I add a dependency to the pom.xml file and build with "mvn package", what steps does Maven take with that dependency code?**

|  |  |
| --- | --- |
|  | In general in all pom.xml we found dependency like this -   1. <dependency> 2. <groupId>org.apache.commons</groupId> 3. <artifactId>commons-lang3</artifactId> 4. <version>3.4</version> 5. </dependency>   Here groupId, artifactId and version are 3 keys by which a jar is uniquely identified. These 3 combination works like a coordinate system for uniquely identifying a point in a space using x, y and z coordinates.  Whenever you issue a mvn package command maven tries to add the the jar file indicating by the dependency to you build path. For doing this maven follows these steps -   1. Maven search in your local repository (default is ~/.m2 for linux). If the dependency/jar is found here then it add the jar file to you build path. After that it uses required class file from the jar for compilation. 2. If the dependency is not found in ~/.m2 then it looks for your local private repository (If you already have configured any using setting.xml file) and maven central remote 3. repository respectively. If you don't have any local private repository then it directly goes to the maven central remote repository. 4. Whenever the jar is found in the local/remote repository it is downloaded and saved in ~/.m2.   Going forward, when you again issue a mvn package command then it's never search for the dependency to any repository since it already in your ~/.m2. |

# [How to add local jar files in maven project?](https://stackoverflow.com/questions/4955635/how-to-add-local-jar-files-in-maven-project)

**Ans :**

Install the JAR into your local Maven repository as follows:

mvn install:install-file

-Dfile=<path-to-file>

-DgroupId=<group-id>

-DartifactId=<artifact-id>

-Dversion=<version>

-Dpackaging=<packaging>

-DgeneratePom=true

Where: <path-to-file> the path to the file to load

<group-id> the group that the file should be registered under

<artifact-id> the artifact name for the file

<version> the version of the file

<packaging> the packaging of the file e.g. jar

**2.how to checkout only a single file from git ?**

**Ans :**

git archive --remote=git://git.foo.com/project.git HEAD:path/to/directory filename | tar -x

**##specifying the tar format explicitly:**

git archive --format=tar --remote=origin HEAD -- <file> | tar xf -

**3. how to checkout only a single file from svn?**

**Ans :**

* svn checkout <url\_of\_big\_dir> <target> --depth empty
* cd <target>
* svn up <file\_you\_want>

**4. how to update only a single file in git ?**

**Ans :**

* git fetch
* git fetch will download all the recent changes, but it will not put it in your current checked out code (working area).
* git checkout origin/master -- path/to/file
* git checkout origin/master -- path/to/file will checkout the particular file from the the downloaded changes (origin/master).

**5. how to update only a single file in svn ?**

**Ans :**

* svn up <filename>

**6.what is a snapshot version ?**

**Ans :**

|  |  |
| --- | --- |
|  | A snapshot version in Maven is one that has not been released.  The idea is that **before** a 1.0 release (or any other release) is done, there exists a 1.0-SNAPSHOT. That version is what might become 1.0. It's basically "1.0 under development". This might be close to a real 1.0 release, or pretty far (right after the 0.9 release, for example).  The difference between a "real" version and a snapshot version is that snapshots might get updates. That means that downloading 1.0-SNAPSHOT today might give a different file than downloading it yesterday or tomorrow.  Usually, snapshot dependencies should **only** exist during development and no released version (i.e. no non-snapshot) should have a dependency on a snapshot version. |

**7. what is a real version ?**

**Ans :**

A real version is a stable version.

**8. how to remove/drop snapshot version ?**

**Ans :**

And simple executing: mvn validate -Pdrop-snapshot  
The version of an example pom passed from 0.0.1-SNAPSHOT to 0.0.1

**We**  have to provide finalname task in pom file…

**9.how do u display the files according to size ?**

**Ans :**

* ls –lS
* **is**  the command to display the files and folders according to size
* **if we want t**o reverse display
* **ls –lSr**
* **if we want to exclude** directories
* **ls -lS | grep –v ‘^d’**
* **if we** want to include only directories
* **ls –lS | grep –i ‘^d’**

**10.stat command is also provide file size and some informations ..**

**11.how to create usernames and emails in git from terminal**

**.Setting your email address for *every* repository on your computer**

git config --global user.email "*email@example.com*"

**Setting your email address for a single repository**

git config user.email [*email@example.com*](mailto:email@example.com)

12.**how to revert back to previous version/shavalue in git ?**

git reset <shavalue> --soft

13.**svn question :**

**How to filter/exclude something from the svn dump file ?**

svndumpfilter exclude sandwiches < dumpfile > filtered-dumpfile

14.**linux question :**

**List the files older than 30 days and delete it ?**

Find –mtime +30 –type f -delete

**Maven interview question :**

**15.how can I run** [**only tests phase in Maven without building and**](https://stackoverflow.com/questions/14881730/run-junit-tests-in-maven-without-building-and-copying-files)  **packaging ?**

We also has to call the goals that are bound to the test phase/phases

**mvn surefire:test**

or if you want to run just one test

**mvn -Dtest=NameOfTest surefire:test**

**######mvn help:describe -Dcmd=compile (or any other phase)**

**Or mvn fr.jcgay.maven.plugins:buildplan-maven-plugin:list**

Then it will display the plugins , phases, ids and goals…

|  |
| --- |
| PLUGIN | PHASE | ID | GOAL |
| --------------------------------------------------------------------------------------- |
| maven-resources-plugin | process-resources | default-resources | resources |
| maven-compiler-plugin | compile | default-compile | compile |
| maven-resources-plugin | process-test-resources | default-testResources | testResources |
| maven-compiler-plugin | test-compile | default-testCompile | testCompile |
| maven-surefire-plugin | test | default-test | test |
| maven-jar-plugin | package | default-jar | jar |
| maven-install-plugin | install | default-install | install |
| maven-deploy-plugin | deploy | default-deploy | deploy |

|  |  |
| --- | --- |
|  | **What is manifest.mf file in jar file ?**  The manifest is a special file that contain information about the files packaged in a JAR file |

Mainfest file mainly contains the name of the class that holds the main function, among various classes in the package...

When you create a JAR file, **it automatically receives a default manifest file. There can be only one manifest file in an archive**, and it always has the pathname

META-INF/MANIFEST.MF

When you create a JAR file, the default manifest file simply contains the following:

Manifest-Version: 1.0

Created-By: 1.7.0\_06 (Oracle Corporation)

These lines show that a manifest's entries take the form of "header: value" pairs