1. [Use of snapshot in artifact](#Snapshot)
2. [what's the difference between package and install?](#diff_package_install)
3. [Maven dependency artifact scopes](#dependency_Scopes)
4. [**Mention the difference between Apache Ant and Maven?**](#diff_apacheAnt_Maven)

Maven is essentially a project management and comprehension tool and as such provides a way to help with managing:

* Builds
* Documentation
* Reporting
* Dependencies
* SCMs
* Releases
* Distribution

#### A Build Lifecycle is Made Up of Phases

#### A Build Phase is Made Up of Plugin Goals

Mvn archtype:goal , here archtype is the plugin and goal is the goal for that plugin.

Pom.xml has build life cycle :

Packaging, plugins…

#### Plugins

**The second way to add goals to phases is to configure plugins in your project**. Plugins are artifacts that provide goals to Maven. Furthermore, a plugin may have one or more goals wherein each goal represents a capability of that plugin. For example, the Compiler plugin has two goals: compile and testCompile. The former compiles the source code of your main code, while the latter compiles the source code of your test code.

pom.xml contains the Project Object Model (POM) for this project.

This is important to remember because Maven is inherently project-centric in that everything revolves around the notion of a 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>

<groupId>com.mycompany.app</groupId>

<artifactId>my-app</artifactId>

<packaging>jar</packaging>

<version>1.0-SNAPSHOT</version>

<name>Maven Quick Start Archetype</name>

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

<dependencies>

<dependency>

<groupId>junit</groupId>

<artifactId>junit</artifactId>

<version>3.8.1</version>

<scope>test</scope>

</dependency>

</dependencies>

</project>

* **project** This is the top-level element in all Maven pom.xml files.
* **modelVersion** This element indicates what version of the object model this POM is using. The version of the model itself changes very infrequently but it is mandatory in order to ensure stability of use if and when the Maven developers deem it necessary to change the model.
* **groupId** This element indicates the unique identifier of the organization or group that created the project. The groupId is one of the key identifiers of a project and is typically based on the fully qualified domain name of your organization. For example org.apache.maven.plugins is the designated groupId for all Maven plug-ins.
* **artifactId** This element indicates the unique base name of the primary artifact being generated by this project. The primary artifact for a project is typically a JAR file. Secondary artifacts like source bundles also use the artifactId as part of their final name. A typical artifact produced by Maven would have the form <artifactId>-<version>.<extension> (for example, myapp-1.0.jar).
* **packaging** This element indicates the package type to be used by this artifact (e.g. JAR, WAR, EAR, etc.). This not only means if the artifact produced is JAR, WAR, or EAR but can also indicate a specific lifecycle to use as part of the build process. (The lifecycle is a topic we will deal with further on in the guide. For now, just keep in mind that the indicated packaging of a project can play a part in customizing the build lifecycle.) The default value for thepackaging element is JAR so you do not have to specify this for most projects.
* **version** This element indicates the version of the artifact generated by the project. Maven goes a long way to help you with version management and you will often see the **SNAPSHOTdesignator in a version, which indicates that a project is in a state of development.** We will discuss the use of snapshots and how they work further on in this guide.
* **name** This element indicates the display name used for the project. This is often used in Maven's generated documentation.
* **url** This element indicates where the project's site can be found. This is often used in Maven's generated documentation.
* **description** This element provides a basic description of your project. This is often used in Maven's generated documentation

### How do I compile my application sources?

mvn compile

### How do I compile my test sources and run my unit tests?

mvn test

If you simply want to compile your test sources (but not execute the tests), you can execute the following:

mvn test-compile

### How do I create a JAR and install it in my local repository?

mvn package

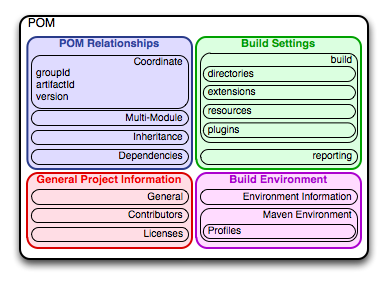
what's the difference between package and install?

Package will compile your code and also package it. For example, if your pom says the project is a jar, it will create a jar for you when you package it and put it somewhere in the target directory (by default).

**Install will compile and package, but it will also put the package in your local repository**. This will make it so other projects can refer to it and grab it from your local repository.

<http://books.sonatype.com/mvnref-book/reference/pom-relationships-sect-pom.html>

we’ve established that the POM describes and declares, it is unlike Ant or Make in that it doesn’t provide explicit instructions, and we’ve noted that POM concepts are not specific to Java. Diving into more specifics, take a look at [Figure 3.1, “The Project Object Model”](http://books.sonatype.com/mvnref-book/reference/pom-relationships-sect-pom.html#fig-pom) for a survey of the contents of a POM.



**Figure 3.1. The Project Object Model**

Use of modelVersion

modelVersion This element indicates what version of the object model this POM is using. The version of the model itself **changes very infrequently** but it **is mandatory in order to ensure stability** of use if and when the Maven developers deem it necessary to change the model.

***groupId*** will identify your project uniquely across all projects

***artifactId*** is the name of the jar without version.

**version** if you distribute it then you can choose any typical version with numbers and dots (1.0, 1.1, 1.0.1, ...). Don't use dates as they are usually associated with SNAPSHOT (nightly) builds. If it's a third party artifact,

# [Maven dependency spring-web vs spring-webmvc](http://stackoverflow.com/questions/13533700/maven-dependency-spring-web-vs-spring-webmvc)

*spring-web* provides core HTTP integration, including some handy servlet filters, Spring HTTP Invoker, infrastructure to integrate with other web frameworks and HTTP technologies (Hessian, Burlap).

*spring-webmvc* is an implementation of Spring MVC. *spring-webvc* [depends on](http://repo1.maven.org/maven2/org/springframework/spring-webmvc/3.1.3.RELEASE/spring-webmvc-3.1.3.RELEASE.pom) on spring-web, thus including it will transitively add *spring-web*. You don't have to add *spring-web* explicitly.

You should depend only on *spring-web* if you don't use Spring MVC but want to take advantage of other web-related technologies that Spring support.

[**what is the difference in maven between dependency and plug-in tags in pom xml**](http://stackoverflow.com/questions/11881663/what-is-the-difference-in-maven-between-dependency-and-plugin-tags-in-pom-xml)

Both plugins and dependencies are Jar files.

But the difference between them is, most of the work in maven is done using plugins; whereas dependency is just a Jar file which will be added to the classpath while executing the tasks.

For example, you use a compiler-plugin to compile the java files. You can't use compiler-plugin as a dependency since that will only add the plugin to the classpath, and will not trigger any compilation. **The Jar files to be added to the classpath while compiling the file, will be specified as a dependency.**

Same goes with your scenario. You have to use spring-plugin to execute some spring executables [ I'm not sure what spring-plugins are used for. I'm just taking a guess here ]. But you need dependencies to execute those executables. And Junit is tagged under dependency since it is used by surefire-plugin for executing unit-tests.

So, we can say, plugin is a Jar file which executes the task, and dependency is a Jar which provides the class files to execute the task.

[**What is Maven artifact?**](http://stackoverflow.com/questions/2487485/what-is-maven-artifact)

An artifact is a file, usually a JAR, that gets deployed to a Maven repository.

A Maven build produces one or more artifacts, such as a compiled JAR and a "sources" JAR.

Each artifact has a group ID (usually a reversed domain name, like com.example.foo), an artifact ID (just a name), and a version string. The three together uniquely identify the artifact.

A project's dependencies are specified as artifacts.

**Maven defines 3 lifecycles in META-INF/plexus/components.xml:**

[**http://maven.apache.org/ref/3.1.1/maven-core/lifecycles.html#default\_Lifecycle**](http://maven.apache.org/ref/3.1.1/maven-core/lifecycles.html#default_Lifecycle)

default Lifecycle

clean Lifecycle

site Lifecycle

**default lifecycle** is defined without any associated plugin. Plugin bindings for this lifecycle are defined separately for every packaging:

Plugins are artifacts that provide goals to Maven. Furthermore, a plugin may have one or more goals wherein each goal represents a capability of that plugin.

Default repository location of maven:

C:\Users\cherukun\.m2\repository\.cache\m2e\1.0.0

Dependency Scopes:

<http://stackoverflow.com/questions/16907682/whats-the-difference-between-these-maven-dependency-scopes-provided-compile-sy>

Ex:

<dependency>

<groupId>com.xx.xx.dorecore</groupId>

<artifactId>client-lib</artifactId>

<version>1.0.04</version>

**<scope>provided</scope>**

</dependency>

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

**There are 6 scopes available:**

* **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.

<http://career.guru99.com/top-20-maven-interview-questions/>

**11)** **Mention the difference between Apache Ant and Maven?**

Apache Ant Maven  
• Ant is a toolbox – Maven is a framework  
• Ant does not have formal conventions like project directory structure – Maven has conventions  
• Ant is procedural; you have to tell to compile, copy and compress – Maven is declarative ( information on what to make & how to build)  
• Ant does not have lifecycle; you have to add sequence of tasks manually – Maven has a lifecycle  
• Ant scripts are not reusable – Maven plugins are reusable

**13) List out what are the build phases in Maven?**

Build phases in Maven are

• Validate  
• Compile  
• Test  
• Package  
• Install  
• Deploy

**14) List out the build, source and test source directory for POM in Maven?**

• Build = Target  
• Source = src/main/java  
• Test = src/main/test

**15) Where do you find the class files when you compile a Maven project?**

You will find the class files ${basedir}/target/classes/.

**16) Explain what would the “jar: jar” goal do?**

jar: jar will not recompile sources; it will imply just create a JAR from the target/classes directory considering that everything else has been done

**17) List out what are the Maven’s order of inheritance?**

The maven’s order of inheritance is

• Parent Pom  
• Project Pom  
• Settings  
• CLI parameters

**18) For POM what are the minimum required elements?**

The minimum required elements for POM are project root, modelVersion, groupID, artifactID and version

**19) Explain how you can produce execution debug output or error messages?**

To produce execution debug output you could call Maven with X parameter or e parameter

The definitions are detailed at [Maven Site](http://maven.apache.org/guides/introduction/introduction-to-the-lifecycle.html), but I have tried to [summarize](https://sites.google.com/site/nosuchmethodexception/buildcycle):

Maven defines 4 "steps" of a Build Process:

1. **Life Cycle**:     3 Life Cycles (aka Build Lifecycles) - Default, Clean, Site. [References](http://maven.apache.org/guides/introduction/introduction-to-the-lifecycle.html#Lifecycle_Reference)
2. **Build Phase**:  Each Life Cycle is made up of Phases - e.g. For default lifecycle: compile, test, package, install, etc
3. **Plugin**:           Artifact that provide one or more goals. Based on packaging(jar,war,etc), a plugin:goal is bind to a phase. [Details](http://maven.apache.org/guides/introduction/introduction-to-the-lifecycle.html#Built-in_Lifecycle_Bindings)
4. **Goals**:           The task (action) that executes. A plugin can have one or more goals. When configuring a plugin in a POM, we specify the goal. Additionally, in case a plugin definition does not have a default build phase, we may specify/bind the phase with the plugin goal.

Phase:Maven phase is a set of action which is associated with 2 or 3 goals

exmaple:- if you run mvn clean

this is the phase will execute the goal mvn clean:clean

Goal:Maven goal bounded with the phase

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 1.0release, 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.