POM stands for **Project Object Model**. It is fundamental unit of work in Maven. It is an XML file that resides in the base directory of the project as pom.xml.

The POM contains information about the project and various configuration detail used by Maven to build the project(s).

Super POM
The Super POM is Maven's default POM. All POMs inherit from a parent or default (despite explicitly defined or not). This base POM is known as the **Super POM**, and contains values inherited by default.

Maven use the **effective POM** (**configuration from super pom plus project configuration**) to execute relevant goal. It helps developers to specify minimum configuration detail in his/her pom.xml. Although configurations can be overridden easily.

An easy way to look at the default configurations of the super POM is by running the following command:

```
mvn help:effective-pom
```

```
c:\> mvn --version
```

Maven - Build Life Cycle

Phase	Handles	Description
prepare-resources	resource copying	Resource copying can be customized in this phase.
validate	Validating the information	Validates if the project is correct and if all necessary information is available.
compile	compilation	Source code compilation is done in this phase.
Test	Testing	Tests the compiled source code suitable for testing framework.
package	packaging	This phase creates the JAR/WAR package as mentioned in the packaging in POM.xml.
install	installation	This phase installs the package in local/remote maven repository.
Deploy	Deploying	Copies the final package to the remote repository.

```
mvn clean dependency:copy-dependencies package
```

Here the clean phase will be executed first, followed by the dependency:copy-dependencies goal, and finally package phase will be executed.

Clean Lifecycle
When we execute mvn post-clean command, Maven invokes the clean lifecycle consisting of the following phases.

- pre-clean
- clean
- post-clean

Maven clean goal (clean:clean) is bound to the clean phase in the clean lifecycle. Its clean:cleangoal deletes the output of a build by deleting the build directory. Thus, when mvn clean command executes, Maven deletes the build directory.

Default (or Build) Lifecycle

This is the primary life cycle of Maven and is used to build the application. It has the following 21 phases.

Sr.No.	**Lifecycle Phase & Description**
1	**validate** Validates whether project is correct and all necessary information is available to complete the build process.
2	**initialize** Initializes build state, for example set properties.
3	**generate-sources** Generate any source code to be included in compilation phase.
4	**process-sources** Process the source code, for example, filter any value.
5	**generate-resources** Generate resources to be included in the package.
6	**process-resources** Copy and process the resources into the destination directory, ready for packaging phase.
7	**compile** Compile the source code of the project.
8	**process-classes** Post-process the generated files from compilation, for example to do bytecode enhancement/optimization on Java classes.
9	**generate-test-sources** Generate any test source code to be included in compilation phase.
10	**process-test-sources** Process the test source code, for example, filter any values.
11	**test-compile** Compile the test source code into the test destination directory.
12	**process-test-classes** Process the generated files from test code file compilation.
13	**test** Run tests using a suitable unit testing framework (Junit is one).
14	**prepare-package** Perform any operations necessary to prepare a package before the actual packaging.

15	**package** Take the compiled code and package it in its distributable format, such as a JAR, WAR, or EAR file.
16	**pre-integration-test** Perform actions required before integration tests are executed. For example, setting up the required environment.
17	**integration-test** Process and deploy the package if necessary into an environment where integration tests can be run.
18	**post-integration-test** Perform actions required after integration tests have been executed. For example, cleaning up the environment.
19	**verify** Run any check-ups to verify the package is valid and meets quality criteria.
20	**install** Install the package into the local repository, which can be used as a dependency in other projects locally.
21	**deploy** Copies the final package to the remote repository for sharing with other developers and projects.

There are few important concepts related to Maven Lifecycles, which are worth to mention −

- When a phase is called via Maven command, for example mvn compile, only phases up to and including that phase will execute.
- Different maven goals will be bound to different phases of Maven lifecycle depending upon the type of packaging (JAR / WAR / EAR).

Maven - Build Profiles
Now open the command console, go to the folder containing pom.xml and execute the following mvn command. Pass the profile name as argument using -P option.

```
C:\MVN\project>mvn test -Ptest
```

Local Repository
Maven local repository is a folder location on your machine. It gets created when you run any maven command for the first time.

Maven local repository keeps your project's all dependencies (library jars, plugin jars etc.). When you run a Maven build, then Maven automatically downloads all the dependency jars into the local repository. It helps to avoid references to dependencies stored on remote machine every time a project is build.

Maven local repository by default get created by Maven in %USER_HOME% directory. To override the default location, mention another path in Maven settings.xml file available at %M2_HOME%\conf directory.

```
<settings xmlns = "http://maven.apache.org/SETTINGS/1.0.0"
  xmlns:xsi = "http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation = "http://maven.apache.org/SETTINGS/1.0.0
  http://maven.apache.org/xsd/settings-1.0.0.xsd">
  <localRepository>C:/MyLocalRepository</localRepository>
</settings>
```

When you run Maven command, Maven will download dependencies to your custom path.

Central Repository

Maven central repository is repository provided by Maven community. It contains a large number of commonly used libraries.

When Maven does not find any dependency in local repository, it starts searching in central repository using following URL − https://repo1.maven.org/maven2/

Key concepts of Central repository are as follows −

- This repository is managed by Maven community.
- It is not required to be configured.
- It requires internet access to be searched.

To browse the content of central maven repository, maven community has provided a URL − https://search.maven.org/#browse. Using this library, a developer can search all the available libraries in central repository.

Remote Repository

Sometimes, Maven does not find a mentioned dependency in central repository as well. It then stops the build process and output error message to console. To prevent such situation, Maven provides concept of Remote Repository, which is developer's own custom repository containing required libraries or other project jars.

For example, using below mentioned POM.xml, Maven will download dependency (not available in central repository) from Remote Repositories mentioned in the same 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.companyname.projectgroup</groupId>
  <artifactId>project</artifactId>
  <version>1.0</version>
  <dependencies>
    <dependency>
      <groupId>com.companyname.common-lib</groupId>
      <artifactId>common-lib</artifactId>
      <version>1.0.0</version>
    </dependency>
  <dependencies>
  <repositories>
    <repository>
      <id>companyname.lib1</id>
      <url>http://download.companyname.org/maven2/lib1</url>
    </repository>
    <repository>
      <id>companyname.lib2</id>
      <url>http://download.companyname.org/maven2/lib2</url>
    </repository>
  </repositories>
</project>
```

Maven Dependency Search Sequence

When we execute Maven build commands, Maven starts looking for dependency libraries in the following sequence −

- **Step 1** − Search dependency in local repository, if not found, move to step 2 else perform the further processing.
- Step 2 − Search dependency in central repository, if not found and remote repository/repositories is/are mentioned then move to step 4. Else it is downloaded to local repository for future reference.
- Step 3 − If a remote repository has not been mentioned, Maven simply stops the processing and throws error (Unable to find dependency).
- Step 4 − Search dependency in remote repository or repositories, if found then it is downloaded to local repository for future reference. Otherwise, Maven stops processing and throws error (Unable to find dependency).

Maven - Plugins

What are Maven Plugins?

Maven is actually a plugin execution framework where every task is actually done by plugins. Maven Plugins are generally used to −

- create jar file
- create war file
- compile code files
- unit testing of code
- create project documentation
- create project reports

A plugin generally provides a set of goals, which can be executed using the following syntax −

```
mvn [plugin-name]:[goal-name]
```

For example, a Java project can be compiled with the maven-compiler-plugin's compile-goal by running the following command.

```
mvn compiler:compile
```

Plugin Types

Maven provided the following two types of Plugins −

Sr.No.	Type & Description
1	**Build plugins** They execute during the build process and should be configured in the <build/> element of pom.xml.
2	**Reporting plugins** They execute during the site generation process and they should be configured in the <reporting/> element of the pom.xml.

Following is the list of few common plugins −

Sr.No.	**Plugin & Description**
1	**clean** Cleans up target after the build. Deletes the target directory.
2	**compiler** Compiles Java source files.
3	**surefire** Runs the JUnit unit tests. Creates test reports.
4	**jar** Builds a JAR file from the current project.
5	**war** Builds a WAR file from the current project.
6	**javadoc** Generates Javadoc for the project.
7	**antrun** Runs a set of ant tasks from any phase mentioned of the build.

Different Archetypes

Sr.No.	**Archetype ArtifactIds & Description**
1	**maven-archetype-archetype** An archetype, which contains a sample archetype.
2	**maven-archetype-j2ee-simple** An archetype, which contains a simplified sample J2EE application.
3	**maven-archetype-mojo** An archetype, which contains a sample a sample Maven plugin.
4	**maven-archetype-plugin** An archetype, which contains a sample Maven plugin.
5	**maven-archetype-plugin-site** An archetype, which contains a sample Maven plugin site.

6	**maven-archetype-portlet** An archetype, which contains a sample JSR-268 Portlet.
7	**maven-archetype-quickstart** An archetype, which contains a sample Maven project.
8	**maven-archetype-simple** An archetype, which contains a simple Maven project.
9	**maven-archetype-site** An archetype, which contains a sample Maven site to demonstrates some of the supported document types like APT, XDoc, and FML and demonstrates how to i18n your site.
10	**maven-archetype-site-simple** An archetype, which contains a sample Maven site.
11	**maven-archetype-webapp** An archetype, which contains a sample Maven Webapp project.

C:\MVN\app-ui>mvn clean package -U
Maven will start building the project after downloading the latest SNAPSHOT of data-service.

Dependency Scope
Transitive Dependencies Discovery can be restricted using various Dependency Scope as mentioned below.

Sr.No.	Scope & Description
1	**compile** This scope indicates that dependency is available in classpath of project. It is default scope.
2	**provided** This scope indicates that dependency is to be provided by JDK or web-Server/Container at runtime.
3	**runtime** This scope indicates that dependency is not required for compilation, but is required during execution.
4	**test** This scope indicates that the dependency is only available for the test compilation and execution phases.
5	**system** This scope indicates that you have to provide the system path.
6	**import** This scope is only used when dependency is of type pom. This scope indicates that the specified POM should be replaced with the dependencies in that POM's <dependencyManagement> section.