MAVEN

1) what are jar files ??

1. You write your program in **Java source code** (e.g., HelloWorld.java).
2. When you compile it using javac HelloWorld.java, Java converts it into **bytecode** (HelloWorld.class).
3. This .class file is what the **JVM (Java Virtual Machine)** understands and executes.
4. Example:
   You typed English → compiler translated it into a language JVM can understand.
5. **Manifest file (telling JVM entry point)**
6. If your project has multiple .class files, JVM won't know **which class has the main method** (starting point of execution).
7. The **Manifest file** says: Main-Class -> HelloWorld
8. This tells JVM: *"Hey, start running the program from HelloWorld class."*
9. Without it, JVM might get confused.
10. Packaged into a .jar file
11. .jar is just like a **.zip file** but for Java programs.
12. Instead of giving someone **many scattered .class files**, you compress them into **one JAR file**.
13. This makes it neat, portable, and reusable.
14. Instead of sending someone 5–10 .class files, you just give one JAR file. They can run it directly.
15. You built a big Java project with 20 classes (A.class, B.class, … Z.class).
16. If you want to share it with a friend or deploy it to a server, you'd have to **send all 20 .class files** separately.
17. That's messy and confusing.
18. Instead: Put all .class files into **one .jar** (say, MyApp.jar).
19. Now you just give them MyApp.jar.
20. They can run it easily using:
21. java -jar MyApp.jar
22. Example in Real Life:
    When you download apps like **Minecraft** (Java Edition), you actually download a .jar file.
    Double-click → runs the whole program.

---

2) What is Project Management in Maven?

1. In Maven, **project management** means:
2. Defining **how your project is built**
3. What **dependencies** (libraries) it needs
4. What **plugins** (extra tools) it will use
5. And the **lifecycle** (steps from compile → test → package → deploy).
6. All of this is written in **one file**: pom.xml
   (POM = Project Object Model).

---

3) What is Maven?

- **Maven** is a build automation tool primarily used for Java projects. It addresses two main aspects of building software: dependency management and project build lifecycle management.
- It simplifies the build process by managing dependencies, compiling source code, packaging it into a deliverable (such as a JAR file), and deploying it to a repository.
- Maven is based on the concept of a Project Object Model (POM), which is a central piece of information that manages a project's build, reporting, and documentation.

a) History and Evolution of Maven

- Maven was developed by the Apache Software Foundation and released in 2004.
- It was created to address the problems with Apache Ant, a popular build tool at the time. Maven was designed to simplify the build process, provide a uniform build system, and offer project information that could be shared among developers.(Ant = manual, verbose, no standardization Maven = automatic, simple, standardized, dependency-aware)
- Over the years, Maven has become one of the most widely used build tools in the Java ecosystem.

b) Understanding the .m2 Directory and settings.xml

How .m2 works with your MYPROJECT

I. In your project pom.xml, you already have a **JUnit dependency**.

II. <dependency>

III. <groupId>junit</groupId>

IV. <artifactId>junit</artifactId>

V. <version>4.13.2</version>

VI. <scope>test</scope>

VII. </dependency>

VIII. When you run:

IX. mvn test

Maven checks:

- "Do I already have JUnit in .m2/repository/?"
- If **yes** → use it.
- If **no** → download JUnit from the internet (Maven Central) and save it inside:
- .m2/repository/junit/junit/4.13.2/

2. Now, even if you open a **new project**, Maven won't re-download JUnit → it will reuse the one saved in .m2.

So .m2 is basically Maven's **library storage bag** for all projects.

c) What is settings.xml

- settings.xml is a **configuration file** that Maven looks for inside your .m2 folder.
- It **does not list dependencies**.
- It only tells Maven **how and where to get those dependencies**.

Now Maven needs to **fetch JUnit**.

Here's the step-by-step:

1. Maven checks your **local cupboard** (.m2/repository/).
   - If JUnit jar is already there → use it.
   - If not → go to step 2.
2. Maven looks at settings.xml (if it exists).
   - If you told Maven "Buy from D-mart" (mirror repo), it will download from there.

- If you told Maven “Go through proxy”, it will use that.
- If you told Maven “Store groceries in D:/maven-repo”, it will save jars there instead of default .m2/repository/.

3. If no settings.xml is present → Maven goes to **Maven Central** on the internet and downloads the dependency.
4. The library (JUnit) is saved into .m2/repository/, so **next time it won’t download again**.

• pom.xml → Shopping list (“I need JUnit”).

• .m2/repository/ → Kitchen cupboard (where Maven stores all downloaded jars).

• settings.xml → Shopping instructions (where to shop, how to enter the shop, where to keep items).

d) Project Object Model (POM)

The POM (Project Object Model) is the core of a Maven project. Understanding the POM file is essential because it contains all the configuration details for a Maven project.

Structure of a pom.xml File

- The pom.xml file is an XML file that contains information about the project and configuration details used by Maven to build the 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.example</groupId>

<artifactId>my-app</artifactId>

<version>1.0-SNAPSHOT</version>

<packaging>jar</packaging>

<name>My App</name>

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

<dependencies>

<dependency>

<groupId>junit</groupId>

<artifactId>junit</artifactId>

<version>4.12</version>

<scope>test</scope>

</dependency>

</dependencies>

```
<build>
<plugins>
<plugin>
<groupId>org.apache.maven.plugins</groupId>
<artifactId>maven-compiler-plugin</artifactId>
<version>3.8.1</version>
<configuration>
<source>1.8</source>
<target>1.8</target>
</configuration>
</plugin>
</plugins>
</build>
</project>
```

e) Key Elements in POM

1. **modelVersion:**
    - This specifies the version of the POM model being used. Currently, 4.0.0 is the default and most widely used version.

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

2. **groupId:**
    - The groupId uniquely identifies your project across all projects. Typically, it follows the reverse domain name convention (e.g., com.example).
    - Like your project's company/organization ID.

```
<groupId>com.example</groupId>
```

3. **artifactId:**
    - The artifactId is the name of the jar without the version. It is the name of the project.

```
<artifactId>my-app</artifactId>
```

4. **version:**
    - The version is the project's current version. Maven uses this version to distinguish between different versions of the same project.
    - Versions can be specific like 1.0.0, or they can include SNAPSHOT, which indicates a version in development.
    - 1.0-SNAPSHOT means *development version*.

```
<version>1.0-SNAPSHOT</version>
```

5. **packaging:**
    - The type of artifact this project produces. Common packaging types are jar, war, pom, etc.
    - If omitted, jar is assumed by default.

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

6. **name and url:**
    - These are optional elements used to provide a human-readable name and project URL
7. • `<name>` → A friendly name for your project.
8. • `<url>` → Website link (optional, can be your project's site or GitHub).

```
<name>My App</name>
<url>http://maven.apache.org</url>
```

.

9. **dependencies:**
    - The dependencies section lists all the external libraries your project depends on. Maven will download these libraries automatically from the central repository or other specified repositories.

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

    -
10. **build:**

- The build section is used to specify how the project should be built. This includes specifying plugins, custom build directories, resources, etc.

```
<build>
    <plugins>
        <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-compiler-plugin</artifactId>
            <version>3.8.1</version>
            <configuration>
                <source>1.8</source>
                <target>1.8</target>
            </configuration>
        </plugin>
    </plugins>
</build>
```

✅ Summary of Your `pom.xml`

- **Group ID** = `com.example`
- **Artifact ID** = `my-app`
- **Version** = `1.0-SNAPSHOT`
- **Packaging** = `jar`
- **Dependency** = JUnit (for testing)
- **Build Plugin** = Compiler plugin (Java 8)

When you run commands:

1. `mvn compile` → compiles code into `target/classes`.
2. `mvn test` → runs JUnit tests.
3. `mvn package` → creates `target/my-app-1.0-SNAPSHOT.jar`.

---

4) Dependencies and Dependency Management

- **Dependencies:**
  - Dependencies are external libraries that your project needs to work. You specify these in the dependencies section of the POM.
  - Maven automatically downloads the specified versions of dependencies and any transitive dependencies they require.

Example:

<dependencies>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-web</artifactId>

<version>2.5.4</version>

</dependency>

</dependencies>

---

5) Transitive Dependencies:

- Maven resolves transitive dependencies automatically. For example, if your project depends on A, and A depends on B, Maven will include B as a dependency in your project.

Analogy (Shopping Example)

- You say in pom.xml:
  "I need **Cake Mix (A)**."
- But Cake Mix itself needs **Sugar (B)**.
- Maven sees this and says:
  "Since you need Cake Mix (A), I'll also bring Sugar (B) for you."

  So you only write A in your shopping list (pom.xml), but Maven gives you **A + B**.

---

6)Maven Dependency Scopes and Profiles

Scope = decides when a dependency is available and whether it goes inside the final JAR/WAR.

1. Dependency Scopes

Maven provides several scopes for dependencies, each determining at what phase of the build process the dependency is available and whether it is included in the final artifact (e.g., a JAR or WAR file). Understanding these scopes is crucial for managing project dependencies effectively.

Here's a detailed explanation of each scope, with examples:

a. compile Scope

- **Default Scope**: If no scope is specified, Maven uses compile by default.
- **Availability**: The dependency is available in all phases of the build lifecycle, including compilation, testing, and packaging.
- **Inclusion in Final Artifact**: The dependency is included in the final artifact.
- **Typical Use**: Used for dependencies required to build, test, and run the project.

Example:

Copy

```
<dependency>
  <groupId>org.apache.commons</groupId>
  <artifactId>commons-lang3</artifactId>
  <version>3.12.0</version>
  <scope>compile</scope> <!-- This is the default scope -->
</dependency>
```

- In this example, commons-lang3 is a utility library used during all phases of the build, so it's specified with compile scope.

b. provided Scope

- **Availability**: The dependency is available at compile-time but is not included in the final artifact.
- **Inclusion in Final Artifact**: Not included in the final artifact. The assumption is that the runtime environment (e.g., a web server) will provide it.
- **Typical Use**: Used for dependencies that are provided by the runtime environment, such as servlet APIs provided by an application server.

Example:

Copy

```
<dependency>
  <groupId>javax.servlet</groupId>
  <artifactId>javax.servlet-api</artifactId>
  <version>4.0.1</version>
  <scope>provided</scope>
</dependency>
```

- Here, the javax.servlet-api is only needed for compilation and testing. It's expected to be provided by the server (e.g., Tomcat) in production, so it's marked with provided scope.

c. runtime Scope

- **Availability**: The dependency is not required for compilation but is needed during runtime, including execution and tests.
- **Inclusion in Final Artifact**: The dependency is included in the final artifact.
- **Typical Use**: Used for dependencies that are needed only at runtime, like JDBC drivers.

Example:

Copy

```
<dependency>
  <groupId>mysql</groupId>
  <artifactId>mysql-connector-java</artifactId>
  <version>8.0.26</version>
  <scope>runtime</scope>
</dependency>
```

- In this example, the MySQL JDBC driver is not required for compiling the code but is needed at runtime when the application connects to a MySQL database.

d. test Scope

- **Availability**: The dependency is available only for the test compilation and execution phases.
- **Inclusion in Final Artifact**: Not included in the final artifact.
- **Typical Use**: Used for libraries that are required only for testing, such as JUnit or Mockito.

Example:

Copy

```
<dependency>
  <groupId>junit</groupId>
  <artifactId>junit</artifactId>
  <version>4.13.2</version>
  <scope>test</scope>
</dependency>
```

- Here, JUnit is only needed for running unit tests and isn't included in the final build output, so it's scoped as test.

e. system Scope

- **Availability**: The dependency is available for compilation and testing but must be explicitly provided by the developer on the system.
- **Inclusion in Final Artifact**: Not included in the final artifact.
- **Typical Use**: Used rarely when the dependency is not available in any remote repository and must be provided locally.
- **Important**: You must explicitly provide the path to the JAR file using the <systemPath> element.

Example:

Copy

```
<dependency>
    <groupId>com.example</groupId>
    <artifactId>custom-lib</artifactId>
    <version>1.0</version>
    <scope>system</scope>
    <systemPath>${project.basedir}/lib/custom-lib.jar</systemPath>
</dependency>
```

- In this example, custom-lib.jar is not available in any Maven repository and is instead provided manually in the lib directory of the project
-

Scope	Available At	Goes in Final Jar?	Example Use
compile	Compile, Test, Run	✅ Yes	Core libs like commons-lang3
provided	Compile, Test	❌ No	Servlet API (server provides it)
runtime	Test, Run	✅ Yes	JDBC Driver
test	Test only	❌ No	JUnit, Mockito
system	Compile, Test	❌ No	Local JARs not in repo

---

7) What is a Maven Profile?

A profile is like a switch in Maven.

- When you turn ON a profile, it changes how Maven builds your project.
- Each profile can have different dependencies, properties, or plugins.
- This is useful when the same project must run in different environments (like development vs production).

Real-Life Analogy

Imagine you're cooking :

- At home (development), you use a small pan and local ingredients.
- In a hotel (production), you use a big pan and premium ingredients.

Instead of keeping two separate recipes, you keep one recipe book with two sections:

- "Home cooking"
- "Hotel cooking"

That's what profiles do: different versions of the same project setup inside one pom.xml.

Example in Maven

Your pom.xml with Profiles

<project>

<modelVersion>4.0.0</modelVersion>

<groupId>com.example</groupId>

<artifactId>my-app</artifactId>

<version>1.0-SNAPSHOT</version>

```
<profiles>
<!-- Development Profile -->
<profile>
<id>development</id>
<properties>
<environment.name>dev</environment.name>
<db.url>jdbc:h2:mem:devdb</db.url>
</properties>
<dependencies>
<!-- Use lightweight H2 database in development -->
<dependency>
<groupId>com.h2database</groupId>
<artifactId>h2</artifactId>
<version>1.4.200</version>
</dependency>
</dependencies>
</profile>
```

```
<!-- Production Profile -->

<profile>

<id>production</id>

<properties>

<environment.name>prod</environment.name>

<db.url>jdbc:mysql://prod-server/db</db.url>

</properties>

<dependencies>

<!-- Use MySQL driver in production -->

<dependency>

<groupId>mysql</groupId>

<artifactId>mysql-connector-java</artifactId>

<version>8.0.26</version>

</dependency>

</dependencies>

</profile>

</profiles>
```

</project>

How to Use It

1. Activate Development Profile

Run:

mvn clean install -Pdevelopment

👉 What happens?

- Maven builds using the development profile.
- It adds H2 database dependency.
- The project connects to jdbc:h2:mem:devdb.

This is good for local testing because H2 is a lightweight, in-memory database.

2. Activate Production Profile

Run:

mvn clean install -Pproduction

👉What happens?

- Maven builds using the production profile.
- It adds MySQL driver dependency.
- The project connects to jdbc:mysql://prod-server/db.

This is good for real deployment where you use an actual MySQL database.

Key Point

- Same project, same pom.xml.
- Just switch profiles with -P.

- Maven will pick different dependencies and settings based on which profile you activate.

8) Maven Build Lifecycle

The build lifecycle is the step-by-step process Maven follows when building your project.
Think of it as a checklist that Maven runs automatically.

Maven has 3 built-in lifecycles:

1) Default Lifecycle (Build Process)

This is the main lifecycle – it takes your source code, compiles, tests, packages, and even deploys.
It has 23 phases, but the most important are:

- **validate →** Check project structure and info is correct.
- **compile** → Compile Java source code (src/main/java) into .class files.
- **test →** Run tests (e.g., JUnit from src/test/java).
- p**ackage** → Package compiled code into JAR/WAR file in target/.
- **verify →** Extra checks (like integration tests or code quality).
- **install →** Copy final JAR into your local repo (.m2/repository/) so other local projects can use it.
- **deploy** → Copy JAR/WAR to a remote repo so other developers/projects can use it.

Example Commands:

mvn compile # compiles code

mvn test # compiles + runs tests

mvn package # compiles + tests + packages into JAR

---

mvn install # compiles + tests + packages + installs in local .m2

mvn deploy # compiles + tests + packages + installs + deploys to remote repo

Rule: When you run one phase, Maven runs all earlier phases too.

- mvn package = validate → compile → test → package.
- mvn install = validate → compile → test → package → verify → install.

2) Clean Lifecycle

This lifecycle is about cleaning old build files.
Phases:

- **pre-clean →** Do something before cleaning.
- **clean →** Delete old build files (the target/ folder).
- **post-clean →** Do something after cleaning.

Example:

mvn clean

Deletes target/ folder so the next build is fresh.

3) Site Lifecycle

This lifecycle is about documentation for your project.
Phases:

- **pre-site →** Preparation before generating site.
- **site →** Generate site docs (from project info, reports, etc.).
- **post-site →** Final steps after generating site.
- **site-deploy →** Upload docs to a server.

Example:

mvn site

Creates project site documentation inside target/site/.

Maven Repositories

9)Types of Maven Repositories

Maven Repositories

A **repository** is simply a place where Maven stores and finds libraries (JARs), plugins, and artifacts.
Think of it like **shopping for ingredients**:

- First, you check your **kitchen cupboard** (local).
- If not found, you go to the **supermarket** (central).
- Or sometimes to a **private store** (remote).

a)Local Repository

- **Where?** Stored on your own computer.
- **What?** Holds all the dependencies Maven has already downloaded.
- **Why?** Saves time — Maven doesn't need to re-download the same library every time.

Default location:

- Windows → C:\Users\YourUsername\.m2\repository
- Linux/Mac → ~/.m2/repository

Example: If your project needs **JUnit**, Maven first checks inside .m2/repository/.

- If found → use it.
- If not → download from Central and store here.

Custom location (in settings.xml):

```
<settings>
  <localRepository>D:/maven-repo</localRepository>
</settings>
```

b)Central Repository

- **What?** Official **global public repo** provided by Maven.
- Contains **thousands of open-source libraries** (JUnit, Spring, Hibernate, etc.).
- **Default source** when local repo doesn't have the dependency.

Example: If your POM asks for Gson:

```
<dependency>
  <groupId>com.google.code.gson</groupId>
  <artifactId>gson</artifactId>
  <version>2.10</version>
</dependency>
```

Maven checks .m2/repository/ first.
If not found, downloads from **Maven Central** (https://repo.maven.apache.org/maven2).

c)Remote Repository

- **What?** A repo hosted on a server other than Central.
- Companies often use them for **private libraries** not available publicly.
- Can be configured in pom.xml or settings.xml.

Example (inside pom.xml):

```
<repositories>
  <repository>
    <id>my-company-repo</id>
    <url>http://repository.mycompany.com/maven2</url>
```

</repository>

</repositories>

Useful when:

- You have internal tools/libraries (not public).
- You want faster downloads (mirror repo inside company).

---

10)How Maven Resolves Dependencies

Dependency Resolution Order

1) Local Repository (.m2/repository/)

- First place Maven looks → already downloaded libraries.
- If found here → used directly, no need to download again.

2) Remote Repositories (if configured)

- If not found locally → Maven checks remote repos you added in pom.xml or settings.xml.
- Example: Company repo for private libraries.

3) Central Repository (Maven Central)

- If still not found → Maven checks the public Maven Central repo.
- URL: https://repo.maven.apache.org/maven2

4) Fail

- If not found in any repo, Maven gives an error →

Could not resolve dependencies for project ...

5) excluding transitive dependency

- When you add a dependency (like spring-boot-starter-web), Maven also pulls its transitive dependencies (like Tomcat).
- If you don't want one of them, you use <exclusions> to block it.
- Example: exclude Tomcat then add Jetty separately.

```xml
<dependencies>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-web</artifactId>
        <version>2.5.4</version>
        <exclusions>
            <exclusion>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-starter-tomcat</artifactId>
            </exclusion>
        </exclusions>
    </dependency>
</dependencies>
```

---

11)Using Repository Managers

Using Repository Managers in Maven

Big organizations don't just rely on Maven Central. Instead, they use tools like Nexus or Artifactory as a middleman between Maven and external repositories.

Why use Repository Managers?

1. Cache Artifacts
    - If one developer downloads a library, it's cached locally in Nexus/Artifactory.
    - Next time, others get it from the manager (faster + safer).
2. Store Internal Artifacts
    - Companies may have private libraries not available on Maven Central.
    - They upload those here so only team members can use them.
3. Enforce Policies
    - Control versions allowed.

- Block unsafe libraries.
- Manage licenses (open-source vs paid).

Example: Configuring Nexus in pom.xml

<repositories>

<repository>

<id>nexus-repo</id>

<url>http://nexus.mycompany.com/repository/maven-public/</url>

</repository>

</repositories>

This tells Maven:

- If a dependency is missing locally, check Nexus first.
- Nexus itself may fetch it from Maven Central if needed.

---

12) Using Mirrors

A mirror = redirecting Maven from one repo to another.

Example: redirect all traffic from Maven Central → Company's internal mirror.

Example in settings.xml:

<mirrors>

<mirror>

```
<id>central-mirror</id>

<mirrorOf>central</mirrorOf>

<url>http://mycompany.com/maven/central</url>

</mirror>

</mirrors>
```

- <mirrorOf>central</mirrorOf> → means “whenever you want Maven Central, use this mirror instead.”
- Useful if:
    - Internet is blocked by firewall
    - Company wants to control dependencies centrally

Super Easy Analogy

- Maven Central = Big supermarket
- Repository Manager (Nexus/Artifactory) = Your company’s warehouse
- Mirror = Redirect sign telling Maven: *“Don’t go to supermarket, take from our warehouse.”*

---

13) Maven Dependency Management

Maven's dependency management system is one of its most powerful features. It automatically handles the inclusion of libraries that your project needs, as well as their transitive dependencies (dependencies of your dependencies). Understanding how to manage these dependencies effectively is key to maintaining a clean and efficient build.

14) Managing Conflicts with Dependency Mediation

a)Dependency Mediation in Maven

When you add dependencies, sometimes **different versions of the same library** sneak in through transitive dependencies.
Maven must decide **which version to use** → this process is called **dependency mediation**.

b)Example of Conflict

- Library **A** depends on **B:1.0**
- Library **C** depends on **B:2.0**
- Your project uses **A + C** → now Maven sees **two versions of B**

Maven will not keep both (that would break the project).
It chooses **one version**: usually the **nearest one** in the dependency tree.

c)How to Fix (Explicit Version)

If you want to make sure a specific version of B is used, you declare it in your pom.xml:

<dependency>

<groupId>com.example</groupId>

<artifactId>B</artifactId>

<version>2.0</version>

</dependency>

This tells Maven: *"Ignore other versions, always use B:2.0."*

d)Best Practices for Dependency Management

1. **Minimize Dependencies**
    - Only include what's really needed → fewer conflicts.
2. **Use <dependencyManagement>**
    - In multi-module projects, keep all versions in one place so all modules use the same versions.
3. **Exclude Unnecessary Dependencies**
    - Use <exclusions> to avoid pulling unwanted extras.
4. **Resolve Conflicts Explicitly**
    - Declare the version you want in pom.xml if multiple versions appear.
5. **Update Regularly**
    - Keep dependencies up to date → fewer bugs + better security.

e)Analogy

Imagine you're making a cake :

- Recipe A says: use **sugar brand 1**.
- Recipe C says: use **sugar brand 2**.
- You can't use both → you must pick one.

By default, Maven picks the "nearest" recipe.
But you (the chef) can **explicitly declare** which sugar to use → to avoid confusion.