### **MAVEN – What is the Build Process?**

The **build process** in a Java project (using Maven or any build tool) typically includes the following steps:

1. **Prepare Resources**  
   → Gather source files, configuration files, images, etc.
2. **Arrange in Folder Structure**  
   → Organize them in standard directories (like src/main/java, src/test/java, resources).
3. **Add Dependencies**  
   → Include required libraries/JAR files via Maven (declared in pom.xml).
4. **Compilation**  
   → Convert .java files to .class files using javac.
5. **Execution**  
   → Run the compiled code to check functionality or to start the app.
6. **Testing**  
   → Execute unit/integration tests (e.g., using JUnit, TestNG).
7. **Packaging**  
   → Bundle the application into a deployable format (like .jar or .war).
8. **Deployment/Release**  
   → Move the packaged app to staging or production environments.

### ✅ Core Java Project Build Process (Steps)

1. **Develop source code and other files**  
   → Write Java classes, configurations, and assets.
2. **Organize files into folders**  
   → Group them under standard folders like src, lib, etc.
3. **Add dependencies (JARs) to classpath**  
   → Include external libraries required for the application.
4. **Compile the code**  
   → Convert .java files to .class files.
5. **Execute & Test the application**  
   → Run the app and perform **unit testing** (testing individual pieces of code).
6. **Package the application for release**  
   → Create a distributable .jar or .war file.

### ❌ Limitations of Manual Build Process

* a. **Hard to remember repetitive tasks**  
  → Developers may forget steps like testing or packaging.
* b. **Risk of incorrect order**  
  → Compiling before adding JARs leads to errors.
* c. **Activities may be skipped**  
  → Testing or packaging may be missed.
* d. **Time-consuming**  
  → Doing each task manually slows down delivery.

..

To automate this process activities we can use .bat file

run.bat

======

cd e:

md xyz

cd xyz

copy

copy

set path =...

set classpath =

javac -d \*. java

java <pkg> .< M

batch file is given to combine all related commands into single command [by using single command we can automate the process]

limitations :

Imagine a .bat script like writing down every single chore you must do, in exact order, with no help if something goes wrong. As soon as the list grows, it falls apart:

* **You must list every step**—there’s no “only redo what changed.”
* **Errors don’t stop the rest**—if one step breaks, the next still runs.
* **Can’t grab outside tools**—it won’t go download the JARs or libraries you need.
* **Windows-only**—it won’t work on Mac or Linux, and even on Windows it’s picky about spaces and quotes.
* **Gets messy fast**—adding new files or folders means editing dozens of lines.

### To avoid this we use

### Ant

* **Like a labeled recipe**: you name each step (compile, test, package) and tell Ant which steps depend on others.
* **Pros**: handles “only do this after that,” runs on any OS with Java, and you can split your build into reusable XML chunks.
* **Cons**: you still have to manually list all your libraries and folder layouts.

Maven is suggested

### Maven

* **Like a self-stocking kitchen**:
  + **Conventions**: expects your code in standard folders (so you don’t have to say “look in src/main/java” every time).
  + **Pantry**: just declare “I need Spring Boot 2.7,” and Maven pulls down Spring and everything it needs.
  + **Workflow**: you call mvn install and it automatically runs compile → test → package → install for you.
* **Pros**: zero-touch dependency downloads, a clear build-to-release process, built-in reporting, and easy multi-module support.

Key features of maven:

### 1. Convention over Configuration

* **What it means**: Maven “knows” where you put your code (src/main/java), tests (src/test/java), resources, etc., so you don’t have to spell out every folder path.

### 2. Automatic Dependency Management

* **Pantry analogy**: Declare “I need Spring Boot” in your POM, and Maven goes out to its online pantry (Maven Central) and fetches Spring Boot + all its friends (transitive libraries) for you.

### 3. Central & Local Repositories

* **Local cache**: Once downloaded, jars sit in your local repository so you don’t redownload them each time.
* **Multiple remotes**: You can point Maven at your company’s private repo or public ones, and it’ll pull plugins and jars as needed.

### 4. Standard Project Layout

* **Readability**: Every Maven project “looks the same” on disk, so any teammate can jump in and know where code, tests, and resources live without asking.

### 5. Inheritance & Reuse

* **Share settings**: A “parent” POM can define common dependencies, plugin versions or corporate policies—child projects inherit those automatically.

### 6. Multi-Module Projects

* **One command, many services**: Group ten microservices under one parent POM. mvn clean install will build them all in the right order for you.

### 7. Built-In Packaging

* **One-stop build**: Want a JAR, WAR or EAR? Just pick the right packaging in your POM and Maven will compile, test and bundle it.

### 8. Plugin Ecosystem

* **Kitchen appliances**:
  + **Compiler plugin** ⇒ compiles your code
  + **Surefire plugin** ⇒ runs your tests
  + **Shade plugin** ⇒ makes a “fat” JAR
  + **Site plugin** ⇒ generates documentation and reports  
    …and dozens more you can drop in with zero hassle.

### 9. Standard Lifecycle

* **Phases you already know**:

mvn clean # wipe out old builds

mvn compile # turn .java → .class

mvn test # run unit tests

mvn package # bundle into JAR/WAR

mvn install # drop into your local repo

mvn deploy # send to remote repo

You just type mvn install and Maven runs all the steps in order.

### 10. Reporting & Documentation

* **One command for a project website**: mvn site will generate Javadocs, test reports, code coverage, styles, plugin docs—everything in neat HTML

## Maven Lifecycles

Maven groups its build work into **three** main lifecycles—each a sequence of phases you can run:

1. **clean**
   * **What it does:** deletes the target/ folder so you start fresh.
   * **Command:** mvn clean
2. **default (aka “build”)**
   * **What it does:** runs all the core build steps in order:

validate → compile → test → package → verify → install → deploy

* + **Command:** typically mvn install (which drives all earlier phases)

1. **site**
   * **What it does:** generates your project’s website, including Javadoc, test reports, coverage, code metrics, etc.
   * **Command:** mvn site

## Archetypes (Project Templates)

Archetypes are “starter kits” that give you a ready-made folder & POM layout:

* **maven-archetype-quickstart**
  + Good for a simple Java app or library.
  + Sets up src/main/java, src/test/java, a minimal POM, and a sample App.java.
* **maven-archetype-webapp**
  + For web applications (WARs).
  + Creates src/main/webapp/WEB-INF, web.xml, plus the usual Java/src folders.

You just run:

mvn archetype:generate \

-DarchetypeArtifactId=maven-archetype-quickstart \

-DgroupId=com.mycompany.app \

-DartifactId=my-app

…and you get a brand-new project scaffold.

## Running Maven Builds

You can invoke these lifecycles either:

1. **On the command line**
   * cd into your project folder and run mvn clean install, mvn site, etc.
2. **From an IDE**
   * Most Java IDEs (Eclipse, IntelliJ, VS Code) let you **import** the pom.xml and then click “Run” or use their Maven toolbar to invoke phases.

### Maven repository — plain-talk version

1. **What it is**  
   A repository is just a big folder (local or online) that keeps all the libraries and plugins your build might need.
2. **What’s inside**
   * **JARs** – normal Java libraries
   * **Plugins** – mini-tools Maven uses while building
   * (Sometimes) sample or parent projects you can inherit from
3. **Local vs. remote**
   * **Local repo**: a cache on your own PC (~/.m2/repository).
   * **Remote repo**: Maven Central or a company server; Maven downloads from here the first time and then stores a copy locally.
4. **How Maven finds the right file (G + A + V)**  
   Every item is labelled with three fields, often called **GAV**:

| **Field** | **Think of it as** | **Example (spring-aspects)** |
| --- | --- | --- |
| **groupId** | The maker’s name | org.springframework |
| **artifactId** | The product’s name | spring-aspects |
| **version** | Edition number | 5.3.17 (stable) or 5.4.0-SNAPSHOT (work-in-progress) |

1. **What happens during a build**
   * Maven reads your <dependency> entry.
   * Checks your local repo for that exact G + A + V.
   * If missing, pulls it once from the remote repo, saves it locally, and continues the build.

### . **Local Repository**

* **Where:** C:\Users\[YourUsername]\.m2\repository
* **What it is:** A personal cache on your machine.
* **Purpose:** Stores the libraries (JARs) downloaded from remote/central, so Maven doesn’t need to re-download them each time.
* **Auto-created:** When you run any Maven command like mvn install, Maven checks here first.

### 🔹 2. **Remote Repository**

* **Where:** On the internet or internal company servers.
* **What it is:** Maintained by third-party teams or organizations (e.g., internal Nexus/Artifactory).
* **Purpose:** Hosts custom or external dependencies not available in the central repo.
* **Example Use Case:** Your company might publish its private libraries to an internal remote repo.

### 🔹 3. **Central Repository**

* **URL:** https://repo.maven.apache.org/maven2
* **What it is:** The **default global repository** maintained by the Maven community.
* **Purpose:** Publicly available library store (e.g., Spring Boot, JUnit, etc.)
* **Access:** Automatically searched by Maven if not found in local or other configured remotes.

### ⚙️ Resolution Order

1. **Local → Remote → Central**
2. Maven looks in the local repo first.
3. If not found, it checks configured remote repos.
4. Then it checks the central repo.

Maven lifecycle:

## 1. Clean Lifecycle (3 phases)

This lifecycle prepares for a fresh build by removing previous outputs.

| **Phase** | **What it does** |
| --- | --- |
| **pre-clean** | (rarely used) do any setup before cleaning |
| **clean** | remove the target/ directory |
| **post-clean** | (rarely used) finalize after cleaning |

* **mvn clean**  
  Executes **all** phases in the Clean lifecycle, ending in post-clean.  
  ⇒ Deletes target/ so you start with a clean slate.

## 2. Default (a.k.a. Build) Lifecycle (23 phases)

This is where compilation, testing, packaging, and installation happen. When you run a command like mvn package, Maven walks **through** these phases in order, stopping at the one you requested.

| **Key Phase** | **Typical Plugin Goal Bound to It** | **What Happens** |
| --- | --- | --- |
| **validate** | — | Check if pom.xml is well-formed |
| **compile** | maven-compiler-plugin:compile | Compile your src/main/java code |
| **test** | maven-surefire-plugin:test | Compile & run your unit tests |
| **package** | maven-jar-plugin:jar (or WAR/EAR plugin) | Bundle compiled code into JAR/WAR |
| **verify** | integration-test verifications | Run any integration checks |
| **install** | maven-install-plugin:install | Install the artifact into local repo (~/.m2/repository) |
| **deploy** | maven-deploy-plugin:deploy | Copy the final artifact to a remote repo |

* **mvn package**  
  Runs **validate → compile → test → package**  
  ⇒ You end up with target/<artifactId>-<version>.jar
* **mvn install**  
  Runs **all the way through install**, so in addition to packaging, your JAR is placed in your local Maven repository for other projects to use.
* **mvn clean package**  
  First runs the **Clean** lifecycle (deleting target/), then runs **Default** up through package.

## 3. Site Lifecycle (4 phases)

This lifecycle generates your project’s documentation and reports.

| **Phase** | **What it does** |
| --- | --- |
| **pre-site** | any setup before reporting |
| **site** | generate the HTML reports/docs |
| **post-site** | finalize and copy resources |
| **site-deploy** | publish the generated site to a web server |

## Running Your JAR Manually

Once you’ve got your MathProj1-1.0.jar, launch it with:

java -cp target/MathProj1-1.0.jar in.ineuron.Arithmetic

* -cp (classpath) points to your JAR.
* in.ineuron.Arithmetic is the fully qualified name of the class containing public static void main(...).