**Introduction to Maven**

Table of Contents

[Why do we need Maven? 2](#_Toc79535746)

[Executing a Java Project 2](#_Toc79535747)

[So, how did we do it? 2](#_Toc79535748)

[For single java file applications 2](#_Toc79535749)

[For multiple java files applications 2](#_Toc79535750)

[For multiple types of files Java Project 2](#_Toc79535751)

[Build System 3](#_Toc79535752)

[Dependencies 3](#_Toc79535753)

[The problem without Maven 4](#_Toc79535754)

[Why Study Maven? 4](#_Toc79535755)

[What is Maven? 4](#_Toc79535756)

[Maven Features 4](#_Toc79535757)

[Installing Maven 5](#_Toc79535758)

[Maven Overview - Core Concepts 5](#_Toc79535759)

[Project Object Model (pom.xml) 5](#_Toc79535760)

[Project Identifiers 6](#_Toc79535761)

[Dependencies 7](#_Toc79535762)

[Repositories 7](#_Toc79535763)

# Why do we need Maven?

To understand Maven, let us first make ourselves familiar with some terminologies of a Java Project

## Executing a Java Project

We have developed many java applications until now such as:

* Simple Java Application containing a single .java file
* A bit more complex Java Program containing multiple .java files
* Java Project that contains not only .java files but also html, css, js, jsp, .jpeg, mp4, xml and jar files

Whatever be the size of the application/project, we have to execute the application.

## So, how did we do it?

### For single java file applications

When we were developing single .java files application, we did it by

* Compiling the source code : javac MyFirstProgram.java
* Executing the generated .class file: java MyFirstProgram 12 45

And then we get the result

### For multiple java files applications

* We have to compile all the .java file that the main program needed
* We can only then run the .class file that contained main method

### For multiple types of files Java Project

##### If we do it manually

This task will be exhausting and virtually impossible as it will include

* Compiling all the .java files that does not depend upon any other .java file
* After the above step we will have to compile .java files that depend upon other smaller classes
* Converting the other resources and attaching them to the java application
* Including the third party .jar files such as connector.jar, servlet-api.jar, jstl.jar, hibernate.jar in the project
* Attach xml files and resources with the executable code

##### If we used IDE (Eclipse)

* Click on the green arrow and the IDE will take care of compiling all the .java files into .class files, attaching the resources, parsing xml files, generate compressed .jar, .war files etc.
* In the same flow after compilation, the IDE will execute the code

# Build System

So, as you can see from the above discussion that writing code and executing code requires a lot of effort. This process of bringing all the resources together to a state where we can successfully execute our Java Application is known as Building the Project.

And the system that we follow to build the project is known as a Build System.

In other words we can say that:

**Software Build** is an activity to translate the human-readable source code into the efficient executable program.

There are a lot of build systems, softwares, plugins available in the market, famous amongst which are:

1. Jenkins
2. Apache Ant
3. Gradle
4. Maven
5. Some IDEs use their custom build process
6. Some IDEs use third party plugins to build their projects

So, we can say that building a software project typically includes one or more of these activities:

* Generating source code (if auto-generated code is used in the project).
* Generating documentation from the source code.
* Compiling source code.
* Packaging compiled code into JAR files or ZIP files.
* Installing the packaged code on a server, in a repository or somewhere else.

## Dependencies

Suppose we write a simple java application which contains two classes: Box and BoxDemo. The main method of BoxDemo class contains the code to instantiate Box class. Now in order to execute the code we will require .class files of both classes: Box.class and BoxDemo.class. The programs will not execute if Box.class is not available.

That is what we call a dependency. Our program depends upon Box.class to execute.

Now take this knowledge into our JEE projects.

* We use Tomcat server that provides servlet-api.jar package that contains .class files for implementing Servlet logic
* We use connector.jar files to connect our Java application to database by using interfaces such as Connection, PreparedStatement, ResultSet etc

All these .jar files that we use are known as dependencies that have to be included in our Project

## The problem without Maven

The good thing is that our IDE takes care of building the whole Java Project by taking proper care of all the dependencies in just a click. Still we need to search, paste and include the jars in our project manually. To automate this redundant task, Maven is used

## Why Study Maven?

In the modern world, we need faster, cost effective and efficient development cycles. So, to achieve this there had been continuous efforts in the field of software development to

* Ease the life of developers,
* Reduce the time of development,
* To reduce the cost of development
* Reduce the bugs

Maven is the result of one such effort

We know that a Java Application needs a lot of dependencies, so it becomes hard for the java programmer to maintain them at a certain point of time. Maven simplifies this process

# What is Maven?

Building a software project typically consists of such tasks as downloading dependencies, putting additional jars on a class-path, compiling source code into binary code, running tests, packaging compiled code into deployable artifacts such as JAR, WAR, and ZIP files, and deploying these artifacts to an application server or repository.

Apache Maven automates these tasks, minimizing the risk of humans making errors while building the software manually and separating the work of compiling and packaging our code from that of code construction.

## Maven Features

The key features of Maven are:

* **Simple project setup that follows best practices:** Maven tries to avoid as much configuration as possible, by supplying project templates (named archetypes)
* **Dependency management:** It includes automatic updating, downloading and validating the compatibility, as well as reporting the dependency closures (known also as transitive dependencies)
* **Isolation between project dependencies and plugins:** with Maven, project dependencies are retrieved from the dependency repositories while any plugin's dependencies are retrieved from the plugin repositories, resulting in fewer conflicts when plugins start to download additional dependencies
* **Central repository system:** project dependencies can be loaded from the local file system or public repositories, such as Maven Central

## Installing Maven

To install Maven on your own system (computer), go to the Maven download page and follow the instructions there. In summary, what you need to do is:

1. Download Maven from: <https://maven.apache.org/download.cgi>
2. Set the JAVA\_HOME environment variable to point to a valid Java SDK

Example: (JAVA\_HOME = C:\Program Files\Java\jdk-14.0.1\).

1. Download and unzip Maven.
2. Set the M2\_HOME environment variable to point to the directory you unzipped

Example: (M2\_HOME = C:\apache-maven-3.6.3)

1. Set the MAVEN\_HOME environment variable to point to the directory you unzipped

Example: (MAVEN\_HOME = C:\apache-maven-3.6.3)

1. Set the M2 environment variable to point to:

Example: (M2 = M2\_HOME/bin)

1. Add M2 to the PATH environment variable (%M2% on Windows)
2. Open a command prompt and type 'mvn -version' (without quotes) and press enter.
3. After typing in the mvn -version command you should be able to see Maven execute, and the version number of Maven written out to the command prompt

## Maven Overview - Core Concepts

### Project Object Model (pom.xml)

Maven is centered on the concept of POM files (Project Object Model). A POM file is an XML representation of project resources like source code, test code, dependencies (external JARs used) etc. The POM contains references to all of these resources. The POM file should be located in the root directory of the project it belongs to.

Let's look at the basic structure of a typical POM file:

|  |
| --- |
| <project>      <modelVersion>4.0.0</modelVersion>      <groupId>org.baeldung</groupId>      <artifactId>org.baeldung</artifactId>      <packaging>jar</packaging>      <version>1.0-SNAPSHOT</version>      <name>org.baeldung</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>              //...              </plugin>          </plugins>      </build>  </project> |

Here is a diagram illustrating how Maven uses the POM file, and what the POM file primarily contains:

|  |
| --- |
| Overview of Maven core concepts. |

### ****Project Identifiers****

Maven uses a set of identifiers, also called coordinates, to uniquely identify a project and specify how the project artifact should be packaged:

* groupId – a unique base name of the company or group that created the project
* artifactId – a unique name of the project
* version – a version of the project
* packaging – a packaging method (e.g. WAR/JAR/ZIP)

The first three of these (groupId:artifactId:version) combine to form the unique identifier and are the mechanism by which you specify which versions of external libraries (e.g. JARs) your project will use.

### ****Dependencies****

These external libraries that a project uses are called dependencies. The dependency management feature in Maven ensures automatic download of those libraries from a central repository, so you don't have to store them locally.

This is a key feature of Maven and provides the following benefits:

* uses less storage by significantly reducing the number of downloads off remote repositories
* makes checking out a project quicker

In order to declare a dependency on an external library, you need to provide the groupId, artifactId, and the version of the library. Let's take a look at an example:

|  |
| --- |
| <dependency>      <groupId>org.springframework</groupId>      <artifactId>spring-core</artifactId>      <version>4.3.5.RELEASE</version>  </dependency> |

### ****Repositories****

A repository in Maven is used to hold build artifacts and dependencies of varying types. The default local repository is located in the .m2/repository folder under the home directory of the user.

If an artifact or a plug-in is available in the local repository, Maven uses it. Otherwise, it is downloaded from a central repository and stored in the local repository. The default central repository is Maven Central.

Some libraries, such as JBoss server, are not available at the central repository but are available at an alternate repository. For those libraries, you need to provide the URL to the alternate repository inside pom.xml file:

|  |
| --- |
| <repositories>    <repository>      <id>JBoss repository</id>      <url><http://repository.jboss.org/nexus/content/groups/public/></url>      </repository>  </repositories> |

**Note:** Please note that you can use multiple repositories in your projects