

**ECLIPSE BUILD**

**What is an Archetype?**

An **archetype** is a template that provides a standard structure and set of configurations for a Maven project. It helps you quickly generate a new project with a predefined set of files, dependencies, and plugins.

**Parameters to Add When Creating a Maven Project**

1. **Group ID (groupId)**:
   * Represents the unique identifier of the project’s group or organization. Typically structured as a reversed domain name (e.g., com.example).
   * Example: com.mycompany.myapp
2. **Artifact ID (artifactId)**:
   * The unique identifier of the project within the group.
   * Example: myapp
3. **Version (version)**:
   * The version of the project. Defaults to 1.0-SNAPSHOT.
   * Example: 1.0.0 or 1.0-SNAPSHOT
4. **Package (package)**:
   * The base Java package structure for the generated source code.
   * Defaults to the combination of the group ID and artifact ID.
   * Example: com.mycompany.myapp
5. **Archetype (archetypeArtifactId)**:
   * The ID of the archetype template you want to use. Maven comes with a set of predefined archetypes, like maven-archetype-quickstart or maven-archetype-webapp.
   * Example: maven-archetype-quickstart
6. **Archetype Group ID (archetypeGroupId)**:
   * The group ID of the archetype. For official Maven archetypes, this is often org.apache.maven.archetypes.
   * Example: org.apache.maven.archetypes
7. **Archetype Version (archetypeVersion)**:
   * The version of the archetype to use.
   * Example: 1.4

**Command to Create a Maven Project Using Archetype**

You can use the mvn archetype:generate command with the above parameters:

mvn archetype:generate -DgroupId=com.example -DartifactId=my-app -DarchetypeArtifactId=maven-archetype-quickstart -DinteractiveMode=false

**Common Maven Archetypes**

Here are some commonly used Maven archetypes:

1. **maven-archetype-quickstart**: A basic Java application template with a src/main/java directory.
2. **maven-archetype-webapp**: A template for a basic Java web application with src/main/webapp directory.
3. **maven-archetype-j2ee-simple**: A simple Java EE application template.
4. **maven-archetype-simple**: A simple, minimalistic project with just a pom.xml and an empty directory structure.
5. **spring-boot-sample-archetype**: An archetype for creating a Spring Boot application.

**Creating a Custom Archetype**: If you want to create your own archetype, follow these steps:

Add settings.xml file from C:\Users\SWAPNIL\Downloads\apache-maven-3.9.9\conf  
to C:\Users\SWAPNIL\.m2

1. Create a standard Maven project with the desired structure and configurations.
2. Package and install it to your local Maven repository:

mvn archetype:create-from-project

1. Navigate to target/generated-sources/archetype and run:

mvn install

This will install your custom archetype, which you can use with mvn archetype:generate in future projects.

[observation: you can delete repository folder in C:\Users\SWAPNIL\.m2, it will generated when you select archetype]

**For Eclipse :**

In Eclipse Build Automatically(enabled in Project tab) will triggered when we save any file, this will use its own incremental compiler(based on jre selected) other than javac, this will compile modified files only to speedup. but while using maven build pom.xml will considered for version if not mentioned then mvn default jdk will considered

This Build Automatically will not create snapshot as mvn, only compiles and save in target folder.

 Use **Preferences > Java > Installed JREs** to check and configure the globally used JDK.

 Use **Project Properties > Java Build Path** to see the JDK used by a specific project.

 Use **Project Properties > Java Compiler** to check the Java compliance level for compilation.

**src/main/resources:**

In a Maven project, the src/main/resources directory is a standard folder used to store non-source code files like configuration files, properties files, static resources, or any other files required by your application at runtime and is automatically recognized by the build system.

**Purpose of src/main/resources**

* **Resource Inclusion**: During the Maven build process, all files in the src/main/resources directory are included in the final JAR or WAR artifact under the classes directory. This means any file placed here will be available on the classpath.[.classpath in project]
* **Configurations**: Typically used for placing configuration files such as application.properties, log4j.xml, or other files that your application reads at runtime.
* **Static Assets**: Can also be used to include static assets such as images, XML files, or any other resources that the application needs.

**Artifact Consideration**

The folder itself (src/main/resources) is not defined in a specific Maven artifact, but it will be bundled into the target directory of your build and included in the final JAR or WAR file.

* For example, if you have src/main/resources/application.properties, it will be included in the built JAR/WAR at target/classes/application.properties.

**JDK vs. JRE Compatibility**:[Case when java 21 project is executed with jre 11]

* When you compile a project using JDK 21, the resulting bytecode may contain features and APIs introduced in Java 21.
* JRE 11 does not know how to interpret or run bytecode that relies on these newer language features or libraries, as they do not exist in Java 11.

**Class File Version Mismatch**:

* Each JDK version compiles .class files with a specific version number. JDK 21 produces class files with a higher version than JRE 11 can handle.
* If you try to run a JDK 21 compiled .class file on JRE 11, you’ll get an error like:

Unsupported major.minor version

* For example, JDK 21 has a class file version of 65.0, while JRE 11 only supports up to 55.0. This mismatch prevents JRE 11 from loading and executing the compiled code.

**Check the Bytecode Version of .class Files**

If you only have the compiled .class files and want to know the Java version:

1. Navigate to the directory containing the .class files.
2. Run the following command:

javap -verbose YourClassName.class | find "major version"

1. The major version number will indicate the Java version:
   * **Major version 52** = Java 8
   * **Major version 53** = Java 9
   * **Major version 54** = Java 10
   * **Major version 55** = Java 11, and so on.

 **Compile for a Lower Version** (If backward compatibility is needed):

* If you need the project to run on JRE 11, you must ensure that the code is compatible and compile it for that version.
* In your Maven or Gradle build file, set the target and source compatibility to a lower version, such as 11.

For Maven:

<properties>

<maven.compiler.source>11</maven.compiler.source>

<maven.compiler.target>11</maven.compiler.target>

</properties>

This way, the bytecode generated will be compatible with JRE 11. However, you’ll need to avoid using features introduced after Java 11.

 **Avoid Using Features Specific to JDK 21**:

* If using JDK 21 is required for development but you want to support running on JRE 11, avoid using APIs or language features that were introduced in versions after Java 11.

**Modified pom.xml File:** To ensure compatibility with JRE 11, modify your pom.xml as follows:

<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>

<!-- Setting Java version properties for Maven Compiler Plugin -->

<properties>

<maven.compiler.source>11</maven.compiler.source>

<maven.compiler.target>11</maven.compiler.target>

</properties>

<!-- Configure the Maven Compiler Plugin to use the specified version -->

<build>

<plugins>

<plugin>

<groupId>org.apache.maven.plugins</groupId>

<artifactId>maven-compiler-plugin</artifactId>

<version>3.8.1</version>

<configuration>

<source>11</source>

<target>11</target>

</configuration>

</plugin>

</plugins>

</build>

</project>

**3. Breakdown of the Configuration:**

* **<properties> Block:**
  + <maven.compiler.source>11</maven.compiler.source>: Tells Maven to treat the source code as Java 11. This means you cannot use language features introduced after Java 11.
  + <maven.compiler.target>11</maven.compiler.target>: Ensures that the compiled bytecode is compatible with Java 11 and can run on JRE 11.
* **Maven Compiler Plugin Configuration:**
  + <source>11</source> and <target>11</target> within the <configuration> block tell the Maven Compiler Plugin to use these settings when compiling the code.

**4. What Happens After This Configuration?[when installed jre is jdk 21]**

* When you run mvn clean install or mvn package with this configuration, Maven compiles the project using JDK 21 but ensures that the compiled bytecode is compatible with Java 11.
* As a result, you’ll get a .jar file or other artifact that you can run on JRE 11 without any issues.

**5. Trying to Use a Java 21 Feature:** Let’s say you try to use a Java 21 feature, like the new pattern matching for switch expressions. For example:

// A Java 21 feature: pattern matching with switch

public class Main {

public static void main(String[] args) {

Object obj = "Hello";

switch (obj) {

case String s -> System.out.println("String: " + s);

case Integer i -> System.out.println("Integer: " + i);

default -> System.out.println("Unknown");

}

}

}

When you compile this code with the above pom.xml configuration (targeting Java 11), you will get a **compilation error** because this feature is not supported in Java 11:

error: switch expressions are not supported in -source 11

(use -source 12 or higher to enable switch expressions)

C:\Users\SWAPNIL\eclipse-workspace\java21\target>java -jar java21-0.0.1-SNAPSHOT.jar no main manifest attribute, in java21-0.0.1-SNAPSHOT.jar

The error no main manifest attribute, in java21-0.0.1-SNAPSHOT.jar occurs because the MANIFEST.MF file in your JAR does not specify the main class to be executed. When you try to run a JAR file using the java -jar command, Java needs to know which class has the main() method to start the execution. This information is provided in the MANIFEST.MF file inside the JAR.

### Solution: Adding the Main-Class Attribute to the Manifest

If you are using Maven to build your project, you can specify the Main-Class attribute in the pom.xml file so that it gets included automatically in the MANIFEST.MF file.

1. Open your pom.xml file.
2. Add the following configuration to the <build> section:

<build>

<plugins>

<plugin>

<groupId>org.apache.maven.plugins</groupId>

<artifactId>maven-jar-plugin</artifactId>

<version>3.2.0</version>

<configuration>

<archive>

<manifest>

<!-- Specify the fully qualified name of the main class -->

<mainClass>com.example.Main</mainClass>

</manifest>

</archive>

</configuration>

</plugin>

</plugins>

</build>

Replace com.example.Main with the fully qualified class name of your main() class. For example, if your main class is in the src/main/java/com/example directory and named Main, it should be com.example.Main.

1. Save the pom.xml file.
2. Rebuild your project with the following Maven command:

mvn clean package or(mvn clean install)

This command will rebuild your JAR and include the Main-Class attribute in the MANIFEST.MF file.

1. Try running the JAR again:

java -jar target/java21-0.0.1-SNAPSHOT.jar

The MANIFEST.MF file is generated automatically when you build your Java project into a JAR file. Here’s a detailed breakdown of when and how the MANIFEST.MF file is created, especially in the context of using Maven:

### When is the MANIFEST.MF File Generated?

1. **During the JAR Creation Process**:
   * When you run a command to package your Java application into a JAR file (such as mvn package or mvn install in a Maven project), the build tool automatically creates the MANIFEST.MF file.
   * The MANIFEST.MF file is typically located in the META-INF directory of the resulting JAR.
2. **Maven Build Lifecycle**:
   * In a Maven project, the MANIFEST.MF file is generated during the package phase of the Maven build lifecycle, specifically by the maven-jar-plugin.
   * This file contains metadata about the JAR file, such as its version, the main class to execute, and other attributes.

Yes, when you build a Java project, it will typically go through a compilation phase. The exact steps involved in the build process depend on the build tool you are using, such as Maven, Gradle, or Ant. Here's an explanation of when and how the project is compiled during the build process:

### Compilation During the Build Process

When you build a Java project, it generally follows these steps:

1. **Clean (Optional)**: Deletes previously compiled files and build artifacts, typically in the target (Maven) or build (Gradle) directories.
2. **Compile**: Compiles the Java source code from src/main/java into bytecode (.class files), which are placed in the target/classes (Maven) or build/classes (Gradle) directory.
3. **Test**: Compiles and runs any unit tests defined in the src/test/java directory, using the compiled classes from the previous step.
4. **Package**: Packages the compiled code and any other resources (e.g., properties files) into a deployable format like a JAR or WAR file. During this step, the MANIFEST.MF file is also generated (for JAR files).
5. **Install/Deploy** (Optional): Installs the packaged artifact into the local repository (e.g., ~/.m2/repository for Maven) or deploys it to a remote repository.

### Example: Maven Compilation Process

If you are using Maven, the typical build commands are as follows:

1. **Compile**:

mvn compile

* + This command compiles your Java source files in src/main/java into .class files and places them in the target/classes directory.

1. **Package**:

mvn package

* + This command compiles the code (if not already compiled) and packages it into a JAR or WAR file in the target directory.

1. **Full Build (Including Tests and Installation)**:

mvn clean install

* + This command cleans the project (deletes the target directory), compiles the source code, runs the tests, packages the project, and installs it into your local Maven repository.

### Difference Between JDK and JRE

1. **JDK (Java Development Kit)**:
   * The JDK includes tools for compiling, debugging, and running Java applications. It contains:
     + The javac compiler for converting source code (.java files) into bytecode (.class files).
     + The java command-line tool for executing bytecode.
     + Other development tools like jar, javadoc, and more.
   * The JDK is required for **compiling** Java code, as the compiler (javac) is part of the JDK, not the JRE.
2. **JRE (Java Runtime Environment)**:
   * The JRE is a subset of the JDK and is used to **run** Java applications.
   * It does not include the javac compiler or other development tools—only the libraries and environment required to execute Java bytecode.
   * The JRE is suitable for running Java applications but **not for compiling** them.

<https://chatgpt.com/share/66fcfff4-e390-8000-b311-3a901a041131>

we have jre21 setup in in build path but in pom 11 we are getting error while building but not while running java app why so?

The error you are encountering is likely due to a mismatch between the JDK version specified in your pom.xml (which is set to Java 11) and the version being used by your IDE's build path (Java 21). Here’s a detailed explanation and how you can resolve it:

### 1. ****Mismatch between Java Build Path and Maven Configuration****

* **Build Path in IDE**: You have set Java 21 in your build path, which means your IDE (e.g., Eclipse or IntelliJ) is using JRE 21 for compiling and running the project.
* **pom.xml Configuration**: The pom.xml specifies Java 11. This configuration is used by Maven for building the project. Maven is unaware of the JRE configuration in your IDE and strictly follows what is defined in pom.xml.

### 2. ****Why It Builds Differently in IDE and Maven****

* **Running in IDE**: When running the application directly from the IDE, the IDE uses the build path settings (JRE 21), allowing the code to run successfully.
* **Building with Maven**: When you build with Maven, it uses the settings defined in the pom.xml. If Maven is configured to use JDK 11, but the code or dependencies require features only available in JDK 21, it will fail to build.

### 3. ****Typical Errors and Causes****

If you see errors like:

* javac: invalid target release: 11
* class not found or unsupported class file major version

These errors indicate a mismatch between the source or target version and the JDK being used.

### 4. ****Solutions****

If you want to use Java 21 consistently, update your pom.xml to specify Java 21:

xml

Copy code

<properties>

<maven.compiler.source>21</maven.compiler.source>

<maven.compiler.target>21</maven.compiler.target>

</properties>

This ensures Maven uses Java 21 for compiling and building the project.

If you want to stick with Java 11, change your IDE's build path to use JRE 11 to match the pom.xml. You can do this by:

* In Eclipse: Go to Project Properties -> Java Build Path -> Libraries, and change the JRE to JRE 11.
* In IntelliJ: Go to File -> Project Structure -> Project -> Set Project SDK and Language Level to Java 11.

In eclipse we set build path to 11 but using jdk 21 when we run it will compiled by which version

In Eclipse, the behavior for running and compiling Java code depends on two main configurations:

1. **Project's Build Path JRE**: Specifies which JRE/JDK library is used to resolve classes, compile, and run the code in the IDE.
2. **Installed JRE/JDK in Eclipse**: Specifies which JDK/JRE is used by the Eclipse IDE itself for various operations, including launching Maven builds.

### Scenario: Build Path Set to Java 11, but JDK 21 Installed

* **Build Path Configured to Java 11**: This means that when you compile or run your project directly from the Eclipse IDE, it will use JDK 11 for compilation and execution (regardless of other installed JDKs).
* **JDK 21 Installed in Eclipse**: If your Eclipse is configured to run with JDK 21 as the default installed JRE, it will not affect the compilation or execution of the project in Eclipse if your project's build path is set to JDK 11.

### What Happens When You Run the Application in Eclipse?

When you run the application in Eclipse:

1. **Compilation**: Eclipse will compile your project using the JDK specified in the project's build path, which in your case is set to JDK 11. This means that even if JDK 21 is installed and available, Eclipse will use JDK 11 for compiling the source code.
2. **Execution**: Similarly, when you execute (run) the application from Eclipse, it will use the JDK 11 runtime environment to execute it, not JDK 21. This is because the JRE System Library specified in the build path controls which runtime is used for executing the code.

### Important Points

* **Eclipse Compilation**: Eclipse has its own incremental compiler, which is separate from the javac provided by JDKs. This means that even if your Eclipse is running with JDK 21, the project's build path JRE (set to 11) dictates how the project is compiled and run.
* **Maven Builds**: If you use Maven to build the project inside Eclipse, the Maven build will follow the JDK specified in your pom.xml or maven.compiler properties, which might differ from the Eclipse build path settings.

### Conclusion

When your project's build path is set to JDK 11 in Eclipse, both the compilation and execution will be handled using JDK 11, even if Eclipse itself is running on JDK 21.

when compilation will happen before or after running

In Eclipse, the compilation process typically happens **before** you run your application. Here's a detailed breakdown of how it works:

### 1. ****Eclipse's Build and Run Process****

When you run or execute a Java application in Eclipse:

1. **Compilation**:
   * Eclipse automatically compiles your source code as soon as it detects a change in the code. This is known as **incremental compilation**.
   * By default, Eclipse compiles the project in the background every time you save a file (using its internal compiler).
   * If the project is not up-to-date or has errors, Eclipse will perform a compilation before executing the code.
2. **Execution (Running the Application)**:
   * Once the code is compiled successfully, Eclipse launches the program using the compiled .class files.
   * If there are any compilation errors (e.g., syntax errors), Eclipse will usually not proceed with execution and will show the errors in the Problems view.

### 2. ****Compilation Happens Before Running****

* **Pre-compilation Requirement**: For any Java code to be executed, it must be compiled into bytecode (.class files). So, when you choose to "Run" the application, if the source files are not yet compiled, Eclipse will compile them first.
* **Manual vs. Automatic Build**: By default, Eclipse uses automatic building (triggered on save). If this is turned off (e.g., using "Project → Build Automatically"), the code will still compile before running, but it will happen during the run execution phase.

### 3. ****How to Control Compilation in Eclipse?****

You can control when the compilation happens:

* **Automatic Build (Default Setting)**: Eclipse will compile the project each time you save a file. You can check this setting under Project → Build Automatically.
* **Manual Build**: If Build Automatically is disabled, you need to manually build the project by selecting Project → Build Project or Build All. When you run the application, Eclipse will compile it first before launching.

### 4. ****What Happens if There are Compilation Errors?****

* If there are compilation errors, Eclipse will usually not allow the project to run. You’ll need to resolve these errors before execution.
* If you run the project without fixing the errors, Eclipse might show a prompt indicating that there are errors, and depending on your settings, it may still try to run using the last successful build (if available).

### 5. ****Console Output During Run****

When you run the application in Eclipse, the console might show the following output sequence:

1. **Build Messages**: Indicating that compilation is happening (if required).
2. **Launch Configuration**: Showing the command to launch the application (e.g., java -classpath ...).
3. **Application Output**: Once the code is compiled and launched, you'll see the application output in the console.

### Summary

* Compilation happens **before** running the application.
* Eclipse uses incremental compilation to keep the code up-to-date.
* If Build Automatically is off, the project will be compiled right before the code is executede when you choose to run it.

If you want to observe this behavior, you can try turning off Build Automatically and making a small change in your code. When you click "Run", you will see Eclipse compile the code first, then launch the application.

what will happen when we build project

When you choose to **Build Project** in Eclipse, several key processes occur, depending on the state of your project and the build configuration settings. Here's what happens during the build:

### 1. ****Source Code Compilation****

* **Incremental Compilation**: Eclipse compiles the source code files (.java) into bytecode (.class files). It only compiles the files that have changed since the last build (incremental compilation), unless a full build is triggered.
* **Error Checking**: The build process identifies any compilation errors or warnings in the source code and updates the Problems view with the corresponding issues.
* **Output to bin or target Directory**: The compiled .class files are placed in the designated output folder. By default, this is the bin directory for standard Java projects, or the target directory for Maven projects.

### 2. ****Resource Copying****

* Non-Java resources (e.g., xml, properties, json files) in the src directory are copied to the output directory (e.g., bin or target). This ensures that all necessary resources are available for the compiled code to run correctly.

### 3. ****Dependency Management****

* Eclipse resolves dependencies (JAR files or other libraries) specified in the build path. For Maven projects, it will also download and link dependencies specified in the pom.xml to ensure the project is correctly configured.
* If using a pom.xml, the dependencies are managed by Maven and stored in the .m2 repository.

### 4. ****Project Validation****

* The build process includes a validation step to check for any inconsistencies in the project configuration, such as missing dependencies, incorrect project structure, or any other issues that could affect the project.

### 5. ****Updating Output Folders****

* Eclipse updates the output folders (bin or target) with the latest compiled classes and resources. If you have linked resources or external libraries, Eclipse will ensure they are correctly referenced.

### 6. ****Full Build vs. Incremental Build****

* **Incremental Build**: If Build Automatically is enabled, Eclipse performs incremental builds by compiling only the modified files. This is efficient and reduces build time.
* **Full Build**: A full build is triggered if there are changes that require the entire project to be rebuilt (e.g., changes in build path or project configuration). This compiles all source files and updates all output directories.

### 7. ****Cleaning Up (Optional)****

* If a "Clean" build is triggered (Project → Clean), Eclipse deletes all compiled .class files and other generated content in the output folders. Then, it rebuilds the entire project from scratch.

### 8. ****Other Possible Actions Depending on Project Type****

* **Maven Projects**: For Maven-based projects, the build process may involve generating additional files or artifacts (e.g., JAR or WAR files) if a specific build goal (e.g., mvn clean install) is executed.
* **Web Projects**: For dynamic web projects, additional steps may include generating deployment descriptors (web.xml), copying resources to the WEB-INF folder, and setting up context paths.

### 9. ****Updating Internal Build State****

* Eclipse updates its internal build state to reflect any changes made during the build. This allows Erclipse to maintain synchronization between the source files, output files, and the build configurations.

### What You See After Building the Project

* **Console Output**: Any output from the build process, such as compiler messages or errors, will be displayed in the console.
* **Problems View**: If there are compilation errors or warnings, they will appear in the Problems view.
* **Output Folder Update**: Check the bin or target folder to see the generated .class files and any copied resources.

### Summary

When you build a project in Eclipse, the IDE compiles the source code, resolves dependencies, copies resources, performs validation, and updates the output folder. The exact actions depend on the project settings, build configuration, and whether it's a standard Java project or uses a build tool like Maven.

what is clean install

The term **clean install** refers to a commonly used command in Maven, a build automation tool for Java projects. When you execute the mvn clean install command, Maven performs two sequential lifecycle phases: clean and install. Let's break down what happens during each of these phases:

### 1. clean ****Phase****

* **Purpose**: The clean phase removes all previously compiled or generated files from the project’s build directory.
* **Effect**: It deletes the target directory (or build directory in non-Maven projects) along with all .class files, compiled artifacts, and any other files that were generated during previous builds.
* **Use Case**: This is typically used to ensure that you start from a fresh state, removing any old or potentially corrupted files before building the project again.

### 2. install ****Phase****

* **Purpose**: The install phase compiles, tests, and packages the application, and then installs the packaged artifact (e.g., JAR, WAR file) into the local Maven repository.
* **Steps Involved**:
  + **Validate**: Ensures the project is correct and all necessary information is available.
  + **Compile**: Compiles the source code of the project.
  + **Test**: Runs the unit tests using a testing framework like JUnit.
  + **Package**: Packages the compiled code into a distributable format like a JAR or WAR file.
  + **Install**: Copies the packaged artifact into your local Maven repository (~/.m2/repository by default).
* **Local Repository**: After a successful install, the resulting artifact (e.g., your-artifact-1.0.jar) will be stored in the local repository (e.g., ~/.m2/repository/com/yourcompany/yourartifact/1.0). This allows other projects on your system to reference this artifact without having to rebuild it.

### 3. ****Combined Effect of**** mvn clean install

When you run mvn clean install, Maven executes the following:

1. **clean**:
   * Deletes the target directory and cleans up all compiled classes and previous build outputs.
2. **install**:
   * Compiles the source code.
   * Runs unit tests.
   * Packages the project into the specified format (e.g., JAR, WAR).
   * Installs the packaged artifact into the local Maven repository for future use by other projects or dependencies.

### 4. ****Use Cases for**** mvn clean install

* **Ensuring a Fresh Build**: Using clean ensures there are no leftover files or stale code from previous builds that could cause issues.
* **Creating a Local Copy**: The install phase helps create a local copy of the packaged project, making it available for other projects to reference as a dependency without needing to rebuild it.
* **Testing and Verifying**: This command is useful when you want to verify that the entire build process works correctly, including compiling, testing, and packaging the application.

### 5. ****Command Breakdown****

Here’s how you would typically run the command in your terminal:

mvn clean install

This command initiates the clean phase first, followed by the install phase. You might see the following output in the console:

s

[INFO] --- maven-clean-plugin:3.1.0:clean (default-clean) @ your-project ---

[INFO] Deleting /path/to/project/target

[INFO] --- maven-resources-plugin:3.1.0:resources (default-resources) @ your-project ---

[INFO] --- maven-compiler-plugin:3.8.1:compile (default-compile) @ your-project ---

[INFO] --- maven-resources-plugin:3.1.0:testResources (default-testResources) @ your-project ---

[INFO] --- maven-surefire-plugin:2.22.2:test (default-test) @ your-project ---

[INFO] --- maven-jar-plugin:3.1.0:jar (default-jar) @ your-project ---

[INFO] --- maven-install-plugin:2.5.2:install (default-install) @ your-project ---

[INFO] Installing /path/to/project/target/your-artifact-1.0.jar to ~/.m2/repository/com/yourcompany/yourartifact/1.0/your-artifact-1.0.jar

### 6. ****Additional Options and Variants****

You can add additional options or specify profiles when running clean install:

* **Skip Tests**: If you want to skip tests, use the -DskipTests flag:

mvn clean install -DskipTests

* **Specify Profiles**: Use profiles to activate specific build configurations:

mvn clean install -Pproduction

### 7. ****Comparison with Other Commands****

* **mvn clean**: Only deletes the target directory and does not compile, test, or install.
* **mvn install**: Compiles, tests, and installs the project without cleaning the target directory first.
* **mvn clean package**: Cleans, compiles, and packages the project, but does not install it in the local repository.

### Summary

mvn clean install is a powerful command that ensures a clean build environment and produces a final packaged artifact, which is then installed into your local repository. This command is useful for ensuring that your project builds successfully from scratch and that the resulting artifacts are ready for use or deployment.

In Java projects, the build process typically involves compiling source code, packaging it, and preparing it for deployment. Depending on the build tool you're using (e.g., Maven, Gradle, Ant), the build process and output folder structure may vary. Here’s a breakdown of how the build is created and where it stores its outputs:

### 1. ****Understanding the Java Build Process****

The build process generally follows these steps:

1. **Compile**: Java source files (.java) are compiled into bytecode (.class files).
2. **Process Resources**: Non-Java files like properties, XML, and configuration files are copied to the build output.
3. **Package**: The compiled classes and processed resources are packaged into a JAR or WAR file.
4. **Install**: The packaged file is added to the local repository (if using a tool like Maven or Gradle).
5. **Deploy**: The package is deployed to a remote repository or server (optional).

### 2. ****Build Tools and Their Output Directories****

#### a) **Maven**

* **Build Folder**: target/
* **Default Build Directory**: When Maven builds a project, the compiled classes are placed in the target/classes folder, and the final artifact (JAR/WAR) is generated in the target directory.
* **Steps to Build**:
  + Use the following commands to trigger a Maven build:

mvn clean install

* + The mvn clean command removes the previous target folder.
  + The mvn install command compiles, tests, and packages the project.

#### b) **Gradle**

* **Build Folder**: build/
* **Default Build Directory**: Gradle places the compiled classes in build/classes/java/main and the final artifact (JAR/WAR) is stored in the build/libs directory.
* **Steps to Build**:
  + Use the following command to build the project:

gradle build

* + The command compiles, tests, and packages the project into the build folder.

#### c) **Ant**

* **Build Folder**: Custom (depends on the build.xml configuration)
* **Default Build Directory**: Ant’s output directory can vary based on the settings in the build.xml file. Typically, a build or dist directory is used.
* **Steps to Build**:
  + Use the following command to build with Ant:

ant build

* + The command will execute the build target defined in the build.xml file.

### 3. ****Project Structure and Folders Considered****

For Java projects, certain folders are considered based on the build tool and structure:

* **Source Folder**: Contains the main application source code.
  + src/main/java → Java source code
  + src/main/resources → Application resources (properties, XML, etc.)
* **Test Source Folder**: Contains test code and resources.
  + src/test/java → Test Java code
  + src/test/resources → Test resources
* **Build Output Folder**: Contains compiled classes, generated files, and final artifacts.
  + target/ (Maven)
  + build/ (Gradle)
  + build/ or custom (Ant)

### 4. ****Location of the Final Build Artifacts****

The final build artifacts (e.g., JAR, WAR) are stored in different directories depending on the tool:

* **Maven**: target/
  + Example: target/my-app-1.0-SNAPSHOT.jar
* **Gradle**: build/libs/
  + Example: build/libs/my-app-1.0-SNAPSHOT.jar
* **Ant**: Custom directory defined in build.xml
  + Example: dist/my-app.jar (if defined as dist)

### 5. ****How Build Tools Decide the Output Folder****

* **Maven** and **Gradle** follow a convention-over-configuration approach, using predefined directories like target or build.
* **Ant** uses an XML-based configuration (build.xml) that explicitly defines source and destination directories.

You can change the default build directory for any of these tools by modifying their respective configuration files:

* **Maven**: pom.xml (<build><directory>custom-target</directory></build>)
* **Gradle**: build.gradle (buildDir = 'custom-build')
* **Ant**: build.xml (e.g., <property name="build.dir" value="custom-build"/>)

### Summary

* The default build folder for Maven is target/, for Gradle is build/, and for Ant it depends on the configuration in the build.xml.
* These folders store compiled classes, processed resources, and packaged artifacts like JARs and WARs.

### Understanding PATH and JAVA\_HOME

* **PATH Environment Variable**: The PATH variable tells your operating system where to look for executable files when you type a command in the terminal or command prompt. Adding the Java binary directory to PATH ensures you can execute Java commands from any directory.
* **JAVA\_HOME Environment Variable**: This variable points to the root directory of your Java installation (e.g., C:\Program Files\Java\jdk-17). While not necessary for running Java commands directly, JAVA\_HOME is useful for other tools (like Maven, Gradle, or IDEs) that need to know where Java is installed.

### Why Set the Java PATH Variable?

Here’s a detailed breakdown of why setting the PATH is important:

1. **Access Java Commands from Anywhere**:

By adding the Java bin directory (which contains executables like java and javac) to the PATH, you can run Java commands from any directory in your terminal without specifying the full path.

**Example**:

* + If the PATH is not set:

/usr/lib/jvm/java-17/bin/java -version

* + After adding the Java bin directory to PATH:

java -version

This makes using Java much more convenient, especially when working on multiple projects or running commands frequently.

1. **Avoid "Command Not Found" Errors**:

If the PATH is not set correctly, you’ll get errors like:

$ javac

bash: javac: command not found

This is because the operating system doesn’t know where to find the javac executable. Setting the PATH resolves this issue.

1. **Simplifies Execution and Compilation**:

When compiling and running Java programs, having the PATH set eliminates the need to include the full directory path to the java or javac commands, making execution easier.

1. **Integration with Other Tools**:

Many Java-based tools (like Apache Maven, Gradle, and IDEs like Eclipse or IntelliJ IDEA) rely on the PATH and JAVA\_HOME to locate the Java installation. Setting PATH ensures these tools can seamlessly use the Java commands.

Providing paths in environment variables is a common practice to make system resources accessible without specifying their exact location every time they are used. The most well-known environment variable related to paths is the PATH variable, which is used to define the directories that the operating system should search for executable files when running commands.

**Even if the variables have the same names, user variables take precedence over system variables.** If a user creates a variable with the same name as a system variable, the user-created local variable created by that user is used instead of the system variable when that user calls the variable. However, this only applies to the person who declared it.

If any other user accesses these variables, the system variable will be called by default. **The user PATH does not let us change the system PATH, which is a sharp contrast to other environment variables.**

**RUN JAVA PROGRAM(CMD):**

If java -version works without setting the path manually, the path is likely configured automatically by the installer or previously set up in the system.

Suppose we create file Test.java

public class Test1 {

public static void main(String[] args) {

System.out.println("Test1");

}

}

**cmd**

**javac Test.java**

**Test.java:1: error: class Test1 is public, should be declared in a file named Test1.java**

**public class Test1 {**

**^**

**1 error**

[If class is declared public then it should be in file with same name]

Suppose we create file Test1.java

public class Test1 {

public static void main(String[] args) {

System.out.println("Test1");

}

}

class Test2 {

public static void main(String[] args) {

System.out.println("Test2");

}

}

**cmd**

**javac Test1.java**

It will create two class files(Test1.class , Test2.class) in same directory. Then you can execute any one of them

**cmd**

**java Test1 // java Test2**

[here Test1 is public and in same file name]

Suppose we create file Test.java

class Test1 {

public static void main(String[] args) {

System.out.println("Test1");

}

}

class Test2 {

public static void main(String[] args) {

System.out.println("Test2");

}

}

**cmd**

**javac Test.java**

It will create two class files(Test1.class , Test2.class) in same directory. Then you can execute any one of them

**cmd**

**java Test1 // java Test2**

[here Test1 and Test2 are not public and so we are able to create file with diff name Test]

A **POJO** (Plain Old Java Object) is a simple Java object that does not follow any special restrictions other than those forced by the Java language specification. It’s used to create lightweight Java objects that primarily encapsulate data, without adding complex functionality or behavior. The term was coined as a way to emphasize that these objects are simple and not bound to any framework-specific conventions.

**Characteristics of a POJO**

1. **No Special Inheritance or Annotations**: A POJO does not implement any special interfaces (like Serializable or Remote) or extend any framework-specific base classes.
2. **Contains Only Fields and Methods**: A POJO usually contains private fields and provides public getter and setter methods to access or modify these fields.
3. **No Business Logic**: Typically, a POJO only holds data and does not contain any business logic.
4. **Easily Readable and Writable**: POJOs are easy to create, read, and maintain due to their simplicity.

**When to Use a POJO?**

POJOs are primarily used to represent data or as model objects in applications. They are often used in frameworks like Hibernate or Spring, where they serve as data carriers or entities that map to database tables or external data sources.

**POJO Example**

Let’s create a simple Employee POJO in Java:

// Employee.java - A simple POJO class

public class Employee {

// Private fields (attributes)

private int id;

private String name;

private String department;

// Default constructor (no-argument constructor)

public Employee() {}

// Parameterized constructor

public Employee(int id, String name, String department) {

this.id = id;

this.name = name;

this.department = department;

}

// Getter and Setter methods for each field

public int getId() {

return id;

}

public void setId(int id) {

this.id = id;

}

public String getName() {

return name;

}

public void setName(String name) {

this.name = name;

}

public String getDepartment() {

return department;

}

public void setDepartment(String department) {

this.department = department;

}

// Optional: Override the toString() method for better readability

@Override

public String toString() {

return "Employee [id=" + id + ", name=" + name + ", department=" + department + "]";

}

}

**Explanation of the POJO Example:**

1. **Private Fields**: The Employee class has three private fields: id, name, and department. This encapsulation ensures that the data is protected from direct modification.
2. **Constructors**: The class has two constructors: a default no-argument constructor and a parameterized constructor to initialize the fields.
3. **Getter and Setter Methods**: Public getter and setter methods are provided for each field, allowing controlled access and modification.
4. **toString() Method**: The toString() method is overridden to give a readable representation of the Employee object.

**When is it Not a POJO?**

An object is not considered a POJO if it:

1. Implements framework-specific interfaces (e.g., Serializable, Cloneable).
2. Extends classes or uses annotations that are required by a particular framework (e.g., @Entity or @Component annotations in Spring or Hibernate).
3. Contains additional behavior or business logic that is not just related to the data it holds.

**Use Cases for POJOs**

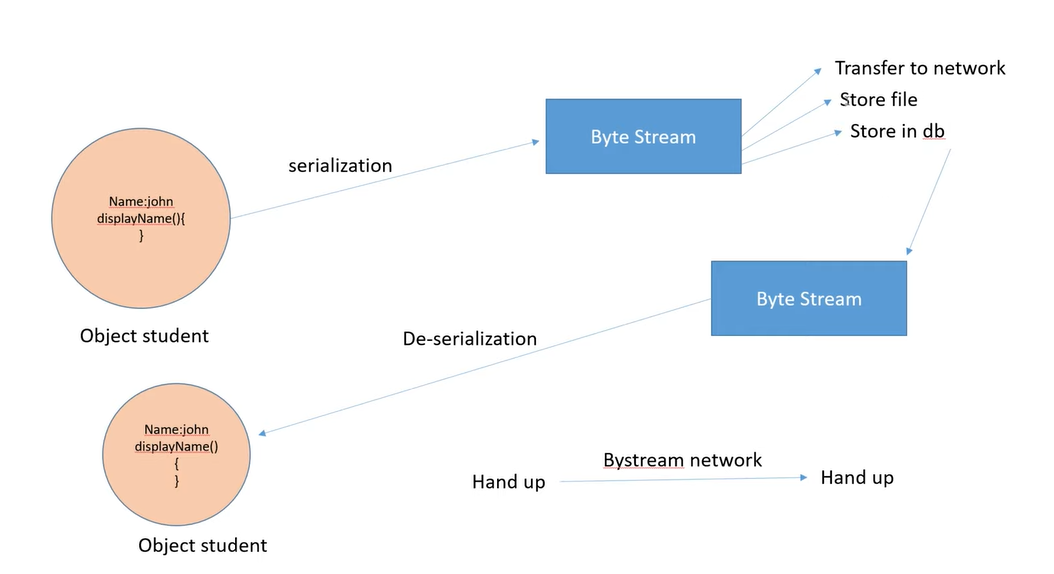
* **Data Transfer Objects (DTOs)**: POJOs are often used as DTOs to carry data between different parts of a program or across a network.
* **Model Objects**: Represent database entities when working with Object-Relational Mapping (ORM) frameworks like Hibernate.
* **Configuration or Setting Objects**: Store application configurations or settings.
* **Serialization/Deserialization**: Hold data to be serialized and deserialized (e.g., converting a POJO to JSON and vice versa).

**POJO vs. Bean vs. Java Object**

* **POJO**: A plain, simple Java object without any special requirements.
* **JavaBean**: A specific type of POJO with additional constraints: a no-argument constructor, properties that are accessible via getters and setters, and being Serializable.
* **Java Object**: Any instance of a class that extends the Object class in Java.

In summary, a POJO is a simple, straightforward Java class that serves as a data container, making it ideal for creating model objects or representing structured data without depending on external libraries or frameworks.

**Serialiation and Deserialization:**



In Java, the **Serializable** interface is used to mark a class so that its objects can be **converted into a byte stream**. This process is known as **serialization**. When a class implements the Serializable interface, it indicates that the objects of that class can be serialized and deserialized, which is helpful for saving objects to a file, sending them over a network, or caching them.

**Key Concepts of Serializable**

1. **Serialization**: The process of converting an object into a byte stream, which can then be stored in a file, transferred over a network, or saved in memory for future use.
2. **Deserialization**: The reverse process of converting a byte stream back into a copy of the original object, reconstructing the object's state.

**How Does Serializable Work?**

* The Serializable interface is a **marker interface**, meaning it doesn’t have any methods. It serves as a signal to the Java Virtual Machine (JVM) and ObjectOutputStream that the class can be serialized.
* When an object is serialized, its state (the values of its fields) is converted into a byte stream.
* When the object is deserialized, the byte stream is used to recreate the object in memory, retaining its original state.

**Why Use Serialization?**

Serialization is useful for a variety of scenarios, such as:

1. **Persistence**: Saving the state of an object to a file or database so that it can be restored later.
2. **Communication**: Transferring objects over a network, for example, in Remote Method Invocation (RMI) or socket programming.
3. **Caching**: Storing the state of objects in memory for performance reasons.
4. **Deep Cloning**: Creating deep copies of objects by serializing and deserializing them.

**How to Make a Class Serializable**

To make a class serializable, the class must implement the Serializable interface.

**Explanation of the Example:**

1. **Employee Class**: The Employee class implements the Serializable interface, making its objects eligible for serialization.
2. **Serialization**:
   * The ObjectOutputStream is used to write the employee object to a file named employee.txt.
   * This process converts the object into a byte stream and saves it to the file.
3. **Deserialization**:
   * The ObjectInputStream reads the employee.txt file and converts the byte stream back into an Employee object.
   * The deserialized object retains the same state as the original.

**Important Considerations for Serialization**

1. **serialVersionUID**:
   * It’s recommended to define a serialVersionUID field in a serializable class.
   * This is used to ensure version compatibility during the deserialization process. If the class definition changes and no matching serialVersionUID is found, a InvalidClassException is thrown.
   * Example:

private static final long serialVersionUID = 1L;

1. **Transient Fields**:
   * Fields marked with the transient keyword are **not serialized**.
   * Use transient for sensitive information (like passwords) or fields that can be recalculated.
   * Example:

private transient String password; // This field will not be saved during serialization

1. **Static Fields**:
   * Static fields belong to the class, not to an instance, and hence are **not serialized**.
   * Only the instance fields of an object are serialized.
2. **Inheritance**:
   * If a superclass implements Serializable, then all its subclasses are automatically serializable.
   * If a class doesn’t implement Serializable, any attempt to serialize its object will result in a NotSerializableException.
3. **Customization**:
   * You can customize the serialization and deserialization processes by defining special methods in your class:
     + private void writeObject(ObjectOutputStream oos) throws IOException
     + private void readObject(ObjectInputStream ois) throws IOException, ClassNotFoundException

**Use Cases of Serialization**

* **Storing User Sessions**: Save and retrieve the state of user sessions in web applications.
* **Sending Objects Over a Network**: Transfer objects in distributed applications (e.g., RMI, message queues).
* **Deep Cloning**: Create deep copies of objects by serializing and deserializing them.
* **Caching**: Store and retrieve cached objects in memory.

**Limitations of Serialization**

* **Performance Overhead**: Serialization can introduce overhead in terms of time and memory usage.
* **Security Risks**: Deserializing untrusted data can lead to security vulnerabilities (e.g., arbitrary code execution).
* **Versioning Issues**: Changes to class definitions can cause InvalidClassException if serialVersionUID is not managed properly.

**Summary**

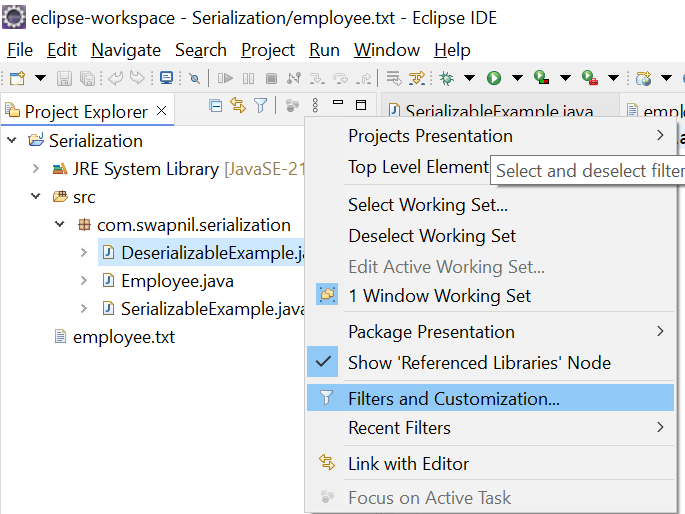
* The Serializable interface allows an object’s state to be converted into a byte stream and vice versa.
* Serialization is useful for saving objects to files, sending objects over a network, or deep cloning.
* Implementing Serializable is straightforward but should be used cautiously due to performance, security, and version compatibility concerns.
* If we change serialVersionUID after executing SerializableExample.java and then run DeserializableExample.java then it will give exception(runtime)

java.io.InvalidClassException: com.swapnil.serialization.Employee; local class incompatible: stream classdesc serialVersionUID = 2, local class serialVersionUID = 1

* If we remove Serializable from class then it will give exception (runtime)

java.io.NotSerializableException: com.swapnil.serialization.Employee

* If we remove ClassNotFoundException then it will give exception



Uncheck .\* resources so that files are visible in eclipse project explorer.