K DHANUSH VARDDHAN

21011102042

MAVEN QUESTION BANK – MIDSEM 7

Question 1: Customizing Maven Build Lifecycle

Your team requires that unit tests should only be executed during the integration-test phase and not during the standard test phase, which is part of the default Maven lifecycle.

How would you modify the Maven build lifecycle to achieve this requirement? What changes would you make in the pom.xml file?

ANS: To modify the Maven build lifecycle so that unit tests are only executed during the integration-test phase, update the pom.xml file as follows:

1. Use the <build> section to define the maven-surefire-plugin.
2. Bind the execution of tests to the integration-test phase.

```
<build>
  <plugins>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-surefire-plugin</artifactId>
      <version>3.0.0-M5</version>
      <executions>
        <execution>
          <id>default-test</id>
          <phase>integration-test</phase>
          <goals>
            <goal>test</goal>
          </goals>
        </execution>
      </executions>
```

```
    </plugin>
  </plugins>
</build>
```

Question 2: Adding External Dependencies and Managing Versions

Your project requires the latest stable version of the Spring Framework, but the repository's current version is outdated. You also need to ensure that the same version of the Spring Framework is used across all modules of your multi-module project.

How would you manage the external dependency in your pom.xml and ensure version consistency across multiple modules?

ANS: To manage the external Spring Framework dependency and ensure version consistency across modules, follow these steps:

1. Use <dependencyManagement> in the parent pom.xml to define the version.
2. In each child module, add a dependency without specifying the version, as it will inherit from the parent.

Parent pom.xml:

```
<dependencyManagement>
  <dependencies>
    <dependency>
      <groupId>org.springframework</groupId>
      <artifactId>spring-core</artifactId>
      <version>5.3.9</version>
    </dependency>
  </dependencies>
</dependencyManagement>
```

Child module pom.xml:

```
<dependencies>
  <dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-core</artifactId>
  </dependency>
```

```
</dependencies>
```

Question 3: Skipping Tests During Build

You are in a rush to deploy a hotfix to production. Running the full suite of unit and integration tests is time-consuming, and you want to skip them during the build process for this hotfix.

How would you configure Maven to skip tests during the build? Is there a specific command you can use to achieve this?

ANS:

To skip tests during a build, you can configure Maven to skip tests using one of the following methods:

1. Use the" -DskipTests" command when running Maven:

   mvn install -DskipTests

2. Configure <skipTests>true</skipTests> in the maven-surefire-plugin in the pom.xml.

```
<build>
  <plugins>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-surefire-plugin</artifactId>
      <configuration>
        <skipTests>true</skipTests>
      </configuration>
    </plugin>
  </plugins>
</build>
```

Question 4: Managing Plugins for Code Coverage

Your project team wants to ensure that the code coverage is above 80% and wants to integrate code coverage reports as part of the build process.

How would you integrate a code coverage tool like JaCoCo into the Maven build lifecycle, and ensure it breaks the build if the coverage falls below 80%?

ANS:

To integrate JaCoCo for code coverage and break the build if coverage falls below 80%, configure JaCoCo in the pom.xml as follows:

```
<build>
  <plugins>
    <plugin>
      <groupId>org.jacoco</groupId>
      <artifactId>jacoco-maven-plugin</artifactId>
      <version>0.8.5</version>
      <executions>
        <execution>
          <goals>
            <goal>prepare-agent</goal>
            <goal>report</goal>
          </goals>
        </execution>
      </executions>
      <configuration>
        <rules>
          <rule>
            <element>BUNDLE</element>
            <limits>
              <limit>
                <counter>LINE</counter>
                <value>COVEREDRATIO</value>
                <minimum>0.80</minimum>
              </limit>
            </limits>
          </rule>
        </rules>
      </configuration>
```

```
    </plugin>
  </plugins>
</build>
```

Question 5: Managing Multi-Module Project Dependencies

You are working on a multi-module project where Module A is dependent on Module B. You need to ensure that any changes in Module B are automatically reflected in Module A without requiring external builds or manual version management.

How would you structure the pom.xml files for both modules to handle this dependency? How would you ensure that changes in Module B are integrated into Module A during the build?

ANS:

To ensure that changes in Module B are reflected in Module A, and to manage dependencies between the modules, follow these steps:

1. In Module A's pom.xml, declare a dependency on Module B without specifying a version:

```
<dependencies>
  <dependency>
    <groupId>com.example</groupId>
    <artifactId>module-b</artifactId>
  </dependency>
</dependencies>
```

2. Ensure that Module B is a sibling or parent module of Module A in the multi-module project's structure, and declare the modules in the parent pom.xml.

Question 6: Overriding a Dependency Version in a Child Module

You are using a third-party library (e.g., Hibernate) in both parent and child modules of your project. However, the child module needs to use a different version of Hibernate from the one declared in the parent POM.

How would you override the dependency version in the child module's pom.xml without affecting the parent module?

ANS:

To override a dependency version in a child module without affecting the parent, redeclare the dependency in the child module's pom.xml with the desired version. Example:

```
<dependencies>
```

```
  <dependency>
    <groupId>org.hibernate</groupId>
    <artifactId>hibernate-core</artifactId>
    <version>5.4.10.Final</version>
  </dependency>
</dependencies>
```

Question 7: Excluding Transitive Dependencies

You have a project that includes a third-party library, but the library comes with unwanted transitive dependencies that are causing conflicts with your existing dependencies. You want to exclude these transitive dependencies.

How would you exclude a specific transitive dependency from a third-party library in your pom.xml?

ANS:

To exclude unwanted transitive dependencies, use the <exclusions> tag within the dependency declaration. Example:

```
<dependency>
  <groupId>com.example</groupId>
  <artifactId>example-library</artifactId>
  <version>1.0.0</version>
  <exclusions>
    <exclusion>
      <groupId>org.unwanted</groupId>
      <artifactId>unwanted-library</artifactId>
    </exclusion>
  </exclusions>
</dependency>
```

Question 8: Using Maven Properties to Manage Project Information

You want to reuse specific information, such as the project version or database URL, across multiple places in the pom.xml. Instead of hardcoding values, you want to manage them using Maven properties.

How would you define and use Maven properties in the pom.xml to avoid duplicating values across different configurations? (Use properties tag)

ANS:

To define and use Maven properties, add them under the <properties> tag in the pom.xml. You can reference these properties throughout the pom.xml. Example:

```
<properties>
  <project.version>1.0.0</project.version>
  <db.url>jdbc:mysql://localhost:3306/mydb</db.url>
</properties>

<version>${project.version}</version>
<url>${db.url}</url>
```

Question 9: Enforcing a Minimum Java Version for the Project

Your project requires a minimum version of Java (e.g., Java 11) to build successfully. You want Maven to fail the build if a lower version of Java is used.

How would you configure the pom.xml to ensure that the project only builds with Java 11 or higher?

ANS:

To enforce a minimum Java version (e.g., Java 11), configure the maven-compiler-plugin as follows:

```
<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>
```

Question 10: Customizing the Default Build Lifecycle

You are working on a Maven project where your team has decided that static code analysis using Checkstyle should be enforced before the unit tests are executed. Currently, Checkstyle is not part of the default Maven lifecycle.

How would you modify the build lifecycle to ensure that Checkstyle runs before the test phase? Describe the changes needed in the pom.xml and explain how the Maven build lifecycle would work after the changes.

ANS:

To ensure that Checkstyle runs before the test phase, modify the Maven lifecycle by binding Checkstyle to a phase before test, such as validate or compile:

```
<build>
  <plugins>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-checkstyle-plugin</artifactId>
      <version>3.1.0</version>
      <executions>
        <execution>
          <phase>validate</phase>
          <goals>
            <goal>check</goal>
          </goals>
        </execution>
      </executions>
    </plugin>
  </plugins>
</build>
```

Question 11: Skipping Specific Lifecycle Phases

You are preparing a quick build for a production hotfix. Your team has decided to skip running unit tests temporarily, but you still want to perform integration tests.

How would you skip the test phase and still execute the integration-test and verify phases in your Maven build? What commands and configurations would you use?

ANS:

To skip the test phase while still running the integration-test and verify phases, use the following command:

mvn verify -DskipTests

Alternatively, configure the pom.xml as follows:

```
<build>
  <plugins>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-surefire-plugin</artifactId>
      <configuration>
        <skipTests>true</skipTests>
      </configuration>
    </plugin>
  </plugins>
</build>
```

Question 12: Binding Goals to Custom Phases

In your project, you need to perform a database schema migration using Flyway right before the integration-test phase. The migration should only be performed in certain environments (e.g., staging or production).

How would you configure the Maven lifecycle and bind the Flyway plugin to run during a custom phase, ensuring it only runs for the correct environments?

ANS:

To run a Flyway database migration before the integration-test phase and only in certain environments, use Maven profiles to define environment-specific configurations. Bind Flyway to the pre-integration-test phase.

```
<profiles>
  <profile>
    <id>staging</id>
    <activation>
      <activeByDefault>false</activeByDefault>
    </activation>
    <build>
      <plugins>
        <plugin>
          <groupId>org.flywaydb</groupId>
          <artifactId>flyway-maven-plugin</artifactId>
          <executions>
            <execution>
              <phase>pre-integration-test</phase>
              <goals>
                <goal>migrate</goal>
              </goals>
            </execution>
          </executions>
        </plugin>
      </plugins>
    </build>
  </profile>
</profiles>
```

Question 13: Handling Failed Builds

During the build process, you notice that your Maven project fails during the package phase due to a compilation error in one of the modules. However, the other modules compile and package successfully.

How would you troubleshoot and isolate the cause of the build failure? Which Maven commands would you use to focus on the failing module without rerunning the entire build?

(Key : Usage of -pl (projects list) and -am (also make) options to run the build for the affected module and its dependencies only)

ANS:

To troubleshoot a failed module without rebuilding everything, use the -pl and -am options to focus on the specific module and its dependencies.

mvn install -pl <module> -am

This command builds the specified module and its dependencies, isolating the cause of the failure without triggering the entire project build.

Question 14: Using the Maven Site Lifecycle

Your team wants to generate a project site with documentation, code reports, and dependency analysis. You need to ensure that the site is generated automatically after the build is successful.

How would you configure Maven to automatically generate a project site as part of the build process? What goals or phases would you bind the site generation to?

ANS:

To generate a project site after a successful build, bind the Maven Site Plugin to the site lifecycle phase. Configure it in the pom.xml as follows:

```
<build>
  <plugins>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-site-plugin</artifactId>
      <version>3.9.1</version>
      <executions>
        <execution>
          <phase>site</phase>
          <goals>
            <goal>site</goal>
          </goals>
```

```
            </execution>
          </executions>
        </plugin>
      </plugins>
</build>
```

Question 15: Multi-Module Build with Independent Modules

You have a multi-module Maven project, but some modules are independent and do not rely on others. You want to be able to build certain modules independently without building the entire project.

How would you structure your pom.xml and use Maven commands to allow independent builds of specific modules?

ANS:

To build independent modules, structure the pom.xml using the <modules> tag in the parent POM. Use Maven commands to build specific modules:

```
<modules>
  <module>moduleA</module>
  <module>moduleB</module>
</modules>
```

Run the following command to build a specific module independently:

```
mvn install -pl moduleA
```

Question 16: Resolving Dependency Conflicts

Your project is facing issues due to conflicting transitive dependencies. Multiple versions of the same library are being pulled in by different dependencies, causing build failures or unexpected runtime behavior.

How would you resolve this dependency conflict in Maven? What are the possible strategies for managing different versions of the same library in the pom.xml?

ANS:

To resolve transitive dependency conflicts, use the <dependencyManagement> section to control versions and apply exclusions to conflicting dependencies.

```
<dependencyManagement>
  <dependencies>
    <dependency>
      <groupId>com.example</groupId>
      <artifactId>library</artifactId>
      <version>1.2.0</version>
      <exclusions>
        <exclusion>
          <groupId>com.unwanted</groupId>
          <artifactId>unwanted-lib</artifactId>
        </exclusion>
      </exclusions>
    </dependency>
  </dependencies>
</dependencyManagement>
```

Question 17: Ensuring a Clean Build for a Multi-Module Project

You are managing a multi-module Maven project, and you notice that sometimes old or generated files from previous builds interfere with the current build, causing failures. To prevent this, you need to ensure that a clean operation is always performed before any new build.

How would you ensure that the clean phase is executed every time before a new build in a multi-module project? Which command would you run to ensure a clean build across all modules?

ANS:

To ensure a clean operation is performed before every build in a multi-module project, run the following Maven command:

mvn clean install

his ensures that the clean phase is executed across all modules before the new build starts.

Question 18: Adding a Custom Clean Goal

In addition to the standard cleaning of the target directory, your project generates temporary files in a custom folder (temp) outside of the target directory. You want to ensure that these files are also deleted during the clean phase.

How would you customize the clean lifecycle to include the deletion of files from the temp directory? How would you configure the pom.xml to achieve this?

ANS:

To include additional directories, such as the temp folder, in the clean lifecycle, customize the maven-clean-plugin in the pom.xml:

```
<build>
  <plugins>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-clean-plugin</artifactId>
      <version>3.1.0</version>
      <executions>
        <execution>
          <goals>
            <goal>clean</goal>
          </goals>
        </execution>
      </executions>
      <configuration>
        <filesets>
          <fileset>
            <directory>${project.basedir}/temp</directory>
            <includes>
              <include>**/*</include>
            </includes>
          </fileset>
        </filesets>
      </configuration>
```

```
    </plugin>
  </plugins>
</build>
```

Question 19: Handling Dependencies that Generate Files

Your project has a dependency on a third-party library that generates temporary files in a specific directory (generated-files/). You need to make sure these files are cleaned up before the next build starts.

How would you ensure that the files generated by the third-party dependency are cleaned up when the clean lifecycle is triggered? What changes to the pom.xml would you make?

ANS:

To clean up files generated by third-party dependencies during the clean phase, add a custom goal in the maven-clean-plugin for the directory where these files are stored.

```
<build>
  <plugins>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-clean-plugin</artifactId>
      <configuration>
        <filesets>
          <fileset>
            <directory>${project.basedir}/generated-files</directory>
            <includes>
              <include>**/*</include>
            </includes>
          </fileset>
        </filesets>
      </configuration>
    </plugin>
  </plugins>
</build>
```

Question 20: Automatically Cleaning Before Testing

You are working on a project where stale files from previous builds sometimes cause unit tests to fail. You want to ensure that every time the test phase is executed, a clean operation is automatically performed before it.

How would you modify the Maven lifecycle to ensure that the clean phase runs automatically before the test phase? How can you ensure that this happens without manually typing mvn clean test each time?

ANS:

To ensure the clean phase runs automatically before tests, bind the clean goal to the test phase in the pom.xml:

```
<build>
  <plugins>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-clean-plugin</artifactId>
      <executions>
        <execution>
          <id>clean-before-test</id>
          <phase>test</phase>
          <goals>
            <goal>clean</goal>
          </goals>
        </execution>
      </executions>
    </plugin>
  </plugins>
</build>
```

Question 21: Customizing the Clean Phase with Additional Goals

In addition to cleaning the build directory, you need to perform additional cleanup actions such as clearing cache files and deleting logs that are stored outside the project structure (e.g., in the user's home directory).

How would you extend the clean lifecycle to include additional cleanup actions, such as clearing a cache directory and removing log files?

ANS:

To extend the clean lifecycle for additional actions like clearing cache or deleting logs outside the project structure, use the maven-clean-plugin and add custom <filesets> in the pom.xml. For example:

```
<build>
  <plugins>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-clean-plugin</artifactId>
      <configuration>
        <filesets>
          <fileset>
            <directory>${user.home}/logs</directory>
            <includes>
              <include>**/*.log</include>
            </includes>
          </fileset>
        </filesets>
      </configuration>
    </plugin>
  </plugins>
</build>
```

This configuration will delete all log files in the user's home directory during the clean phase.

Question 22: Cleaning Dependent Projects in a Multi-Module Build

In a multi-module project, certain modules depend on each other. You want to ensure that the clean phase is executed across all dependent modules if one module is cleaned.

How would you ensure that cleaning one module in a multi-module project triggers the clean phase for all dependent modules? How would you configure the build to handle this situation?

ANS:

o ensure that cleaning one module triggers the clean phase for all dependent modules, Maven automatically handles this through the reactor mechanism. You can run:

mvn clean install

Maven will clean all modules based on dependencies specified in the parent pom.xml. For more control, you can use -pl (project list) with -am (also make) to clean dependencies of a specific module:

mvn clean install -pl <module> -am

Question 23: Managing Build Artifacts During the Clean Phase

Your project is using third-party libraries that download files into a local repository or external folders during the build process. However, you don't want these downloaded files to be removed during the clean phase because they are needed for future builds.

How would you exclude certain directories or files (e.g., the repository directory) from being deleted during the clean phase while still cleaning the main target directory?

ANS:

To exclude certain directories (e.g., the local repository) from being deleted during the clean phase, configure the maven-clean-plugin to exclude specific paths:

```
<build>
  <plugins>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-clean-plugin</artifactId>
      <configuration>
        <excludeDefaultDirectories>true</excludeDefaultDirectories>
        <filesets>
          <fileset>
            <directory>${project.build.directory}</directory>
            <excludes>
              <exclude>repository/**</exclude>
            </excludes>
          </fileset>
```

```
        </filesets>
      </configuration>
    </plugin>
  </plugins>
</build>
```

Question 24: Reusing Common Build Configurations

You are managing a set of Maven projects, and they all share common configurations such as Java version, plugin configurations, and dependency versions. You want to centralize these configurations so that each child project does not need to repeat them.

How would you structure a parent POM file to manage shared configurations (such as Java version and commonly used plugins) that all child projects can inherit? What elements of the parent POM would you define for reuse?

ANS:

To centralize common configurations across Maven projects, use a parent POM to define shared settings like Java versions, plugins, and dependency management. Define shared configurations in the parent POM's <dependencyManagement>, <build>, and <properties> sections:

```
<properties>
  <maven.compiler.source>11</maven.compiler.source>
  <maven.compiler.target>11</maven.compiler.target>
</properties>

<dependencyManagement>
  <dependencies>
    <!-- Common dependencies -->
  </dependencies>
</dependencyManagement>

<build>
  <plugins>
    <!-- Common plugins -->
```

```
  </plugins>
</build>
```

Question 25: Overriding Parent Configurations in Child Projects

You have a parent POM that defines shared configurations, but one of your child projects requires a different Java version (Java 11 instead of Java 8) and an additional plugin. You want to override the Java version and add the new plugin in the child POM without affecting other child projects.

How would you override the Java version and add a new plugin in the child POM? What parts of the child POM would you need to modify?

ANS:

In the child pom.xml, you can override the Java version and add new plugins by redefining them in the <properties> and <build> sections of the child project:

```
<properties>
  <maven.compiler.source>11</maven.compiler.source>
  <maven.compiler.target>11</maven.compiler.target>
</properties>

<build>
  <plugins>
    <plugin>
      <groupId>com.example</groupId>
      <artifactId>new-plugin</artifactId>
      <version>1.0.0</version>
    </plugin>
  </plugins>
</build>
```

Question 26: Managing Dependencies in Parent and Child Projects

Your parent POM defines several common dependencies, but a child project requires a different version of one of these dependencies. You want to override the version of this dependency in the child project without modifying the parent POM.

How would you override the version of a dependency in a child project while inheriting other dependencies from the parent? What part of the child POM would you modify to achieve this?

ANS:

To override a dependency in a child project without modifying the parent POM, use <dependencyManagement> in the parent POM to control versions, and then redeclare the dependency with the desired version in the child project:

```
<dependencies>
  <dependency>
    <groupId>org.example</groupId>
    <artifactId>example-dependency</artifactId>
    <version>2.0.0</version> <!-- Overriding version -->
  </dependency>
</dependencies>
```

Question 27: Using Profiles in Parent POM for Environment-Specific Configurations

You have a parent POM that defines different configurations for various environments (development, staging, production). Each child project should automatically inherit these configurations, and you want to use Maven profiles to switch between environments.

How would you set up environment-specific configurations using Maven profiles in the parent POM? How can the child projects inherit and use these profiles?

ANS:

Set up environment-specific profiles in the parent POM using <profiles>. Each child project will automatically inherit these profiles, and you can activate them based on different environments:

```
<profiles>
  <profile>
    <id>development</id>
    <activation>
      <activeByDefault>true</activeByDefault>
    </activation>
    <properties>
      <!-- Development-specific properties -->
```

```
    </properties>
  </profile>
  <profile>
    <id>production</id>
    <properties>
      <!-- Production-specific properties -->
    </properties>
  </profile>
</profiles>
```

Question 28: Inheriting and Extending Build Plugins

The parent POM defines several build plugins that are used across all projects. However, one of the child projects requires additional configuration for one of these plugins (e.g., a different goal or execution phase). You need to extend the plugin configuration in the child POM without affecting other projects.

How would you extend the configuration of an inherited plugin in the child POM? What changes would you make to the child POM to achieve this?

ANS:

To extend an inherited plugin in a child POM, redefine the plugin with the additional configuration needed. For instance, if you need to add a new goal to the inherited plugin:

```
<build>
  <plugins>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-surefire-plugin</artifactId>
      <executions>
        <execution>
          <id>custom-test</id>
          <goals>
            <goal>test</goal>
          </goals>
        </execution>
```

```
        </executions>
      </plugin>
    </plugins>
</build>
```

Question 29: Managing Parent-Child Relationships in a Multi-Module Project

You are working on a multi-module Maven project where each child module inherits from a common parent POM. However, one of the child modules has specific requirements that need a new plugin or different configuration that should not affect the other child modules. You want to ensure that only this child module includes the new plugin without altering the configurations in other modules.

How would you handle the situation where one child module requires a new plugin or different configuration, but this should not impact the other child modules? What changes would you make in the child POM to meet this requirement?

ANS:

In a multi-module project, if a specific child module requires a new plugin or different configuration, define the plugin or configuration in that child's pom.xml without affecting other modules:

```
<build>
  <plugins>
    <plugin>
      <groupId>com.example</groupId>
      <artifactId>new-plugin</artifactId>
      <version>1.0.0</version>
    </plugin>
  </plugins>
</build>
```

Question 30: Inheriting and Customizing Dependency Versions in Child Projects

Situation: You have a parent POM that defines common dependencies using <dependencyManagement>. Some child projects need to use the same dependencies but with different versions. You need to ensure that child projects can override the versions without modifying the parent POM.

How would you allow child projects to inherit common dependencies from the parent POM but use different versions where needed? What changes would you make in the child POM to override the dependency versions?

ANS:

To allow child projects to inherit common dependencies but use different versions, declare the dependencies in <dependencyManagement> in the parent POM and override specific versions in the child project:

```
<dependencyManagement>
  <dependencies>
    <dependency>
      <groupId>org.example</groupId>
      <artifactId>example-dependency</artifactId>
      <version>1.0.0</version>
    </dependency>
  </dependencies>
</dependencyManagement>

<!-- Child POM -->
<dependencies>
  <dependency>
    <groupId>org.example</groupId>
    <artifactId>example-dependency</artifactId>
    <version>2.0.0</version> <!-- Overriding version -->
  </dependency>
</dependencies>
```

Question 31: Customizing the Clean Lifecycle

You have a multi-module project, and one of the child modules requires a custom action during the clean phase (such as deleting specific files that are not included in the standard Maven clean). The other child modules should retain the default behavior of the clean phase.

How would you customize the Maven clean phase for a specific module while retaining the default clean behavior for other modules? What changes would you make in the child POM to customize the clean phase?

ANS:

To customize the clean lifecycle for a specific child module, modify the child POM's <build> section with the maven-clean-plugin, while leaving other modules unchanged:

```
<build>
  <plugins>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-clean-plugin</artifactId>
      <executions>
        <execution>
          <id>custom-clean</id>
          <goals>
            <goal>clean</goal>
          </goals>
          <configuration>
            <filesets>
              <fileset>
                <directory>specific-directory</directory>
              </fileset>
            </filesets>
          </configuration>
        </execution>
      </executions>
    </plugin>
  </plugins>
</build>
```

Question 32: Using Plugin Inheritance with Custom Execution Goals

The parent POM defines several common plugins used by all child modules. One of the child modules, however, requires an additional goal for one of these plugins (e.g., running tests with a specific configuration).

How would you extend the plugin configuration in the child POM to add a custom execution goal for a plugin that is inherited from the parent POM? What changes would you make to the child POM to include the additional goal?

ANS:

To add custom execution goals to an inherited plugin in the child POM, extend the plugin definition by adding new executions:

```
<build>
  <plugins>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-surefire-plugin</artifactId>
      <executions>
        <execution>
          <id>custom-test</id>
          <goals>
            <goal>test</goal>
          </goals>
          <configuration>
            <!-- Custom test configuration -->
          </configuration>
        </execution>
      </executions>
    </plugin>
  </plugins>
</build>
```

This allows the child project to customize the inherited plugin without altering the parent.