# SpEL

## Introduction to SpEL (Spring Expression Language)

**Definition**
SpEL is a **powerful expression language in Spring** used to **inject values dynamically, perform operations, and evaluate expressions at runtime.**

---

**Key Points**

- Supports **mathematical, relational, and logical operations**
- Can access **properties, methods, and collections**
- Used inside @Value, XML config, or annotations

---

**Purpose / Importance**

- Enables **dynamic configuration**
- Reduces hardcoding and boilerplate code
- Can perform **runtime calculations or evaluations**

---

**Syntax / Example**
**Injecting a simple calculation:**

```
@Value("#{2 + 3}")
private int sum; // 5
```

**Accessing a bean property:**

```
@Component
public class Engine {
    private int power = 120;
    public int getPower() { return power; }
}

@Component
public class Car {
    @Autowired
    private Engine engine;

    @Value("#{engine.power}")
    private int enginePower;
}
```

**Using collections:**

```
@Value("#{T(java.lang.Math).random() * 100}")
private double randomNumber;
```

---

**Real-life Example**
Like **using a calculator** to compute values dynamically instead of manually entering them everywhere.

---

**Interview Tips / Questions**
**Q:** Where can SpEL be used?
**A:** @Value, XML, annotations, or programmatically.
**Q:** Difference between @Value simple injection and SpEL?
**A:** SpEL can **compute values, call methods, or access beans**, not just simple property injection.
**Q:** Can SpEL access static methods?
**A:** Yes, using T(className).method() syntax.

---

**One-Line Summary**
SpEL allows **dynamic value injection and runtime evaluation** in Spring applications.

---

---

## SpEL: Literal Expressions, Variables, and References

**Definition**

- **Literal Expressions:** Fixed values directly written in SpEL (numbers, strings, booleans).
- **Variables:** Dynamic placeholders that store and reuse values in SpEL.
- **References:** Accessing **beans, properties, or other SpEL objects**.

---

**Key Points**

- **Literal:** Simple constants like 'Hello', 100, true.
- **Variables:** Defined using #variableName syntax.
- **References:** Access beans or other objects via @beanName or #variableName.

---

**Purpose / Importance**

- **Literal:** Quickly inject fixed values.
- **Variables:** Reuse values and reduce redundancy.
- **References:** Dynamically link to other beans or objects.
- Makes **dynamic configuration and runtime evaluation easy**.

---

**Syntax / Examples**
**Literal Expression:**

```
@Value("#{'Hello World'}")
private String greeting;

@Value("#{10 + 20}")
private int sum;
```

**Variables:**

```
ExpressionParser parser = new SpelExpressionParser();
StandardEvaluationContext context = new StandardEvaluationContext();
context.setVariable("val", 50);

int result = parser.parseExpression("#val * 2").getValue(context, Integer.class); //
100
```

**References to Beans:**

```
@Component
public class Engine {
    private int power = 120;
    public int getPower() { return power; }
}

@Component
public class Car {
    @Value("#{engine.power}")
    private int enginePower;
}
```

---

**Real-life Example**

- **Literal:** Writing “5 apples” on a paper.
- **Variable:** Storing “5” in a box for repeated use.

- **Reference:** Referring to someone else's box of apples when needed.

---

**Interview Tips / Questions**
**Q:** Difference between literal and variable in SpEL?
**A:** Literal = fixed value, Variable = reusable dynamic value.
**Q:** How do you refer to another bean in SpEL?
**A:** Using @beanName in the expression.
**Q:** Can variables be used in @Value annotation?
**A:** Only if the context is provided programmatically (not directly in @Value).

---

**One-Line Summary**
**Literal = fixed, Variable = reusable, Reference = access beans/objects dynamically.**

---

---

## SpEL (Spring Expression Language) in Different Contexts

**Definition**
SpEL allows **dynamic evaluation of expressions** in Spring. It can be used in:
- @Value annotations
- XML configuration
- Other Spring annotations

---

**Key Points**
- Supports **mathematical, logical, relational, and string operations**
- Can **access beans, properties, collections, methods, and static fields**
- Enhances **flexibility and dynamic configuration**

---

**Purpose / Importance**
- Avoids hardcoding values
- Enables **runtime evaluation** and **dynamic injection**
- Works consistently across different Spring configuration approaches

---

**Syntax / Examples**

**SpEL with @Value**
```
@Component
public class Car {

    @Value("#{2 * 50}")
    private int maxSpeed;

    @Value("#{engine.power}")
    private int enginePower;
}
```

**SpEL in XML Configuration**
```
<bean id="car" class="com.example.Car">
    <property name="maxSpeed" value="#{2 * 50}"/>
    <property name="enginePower" value="#{engine.power}"/>
</bean>
```

**SpEL in Other Annotations**

```
@Scheduled(cron = "#{@cronExpressionBean.getCronValue()}")
public void scheduledTask() {
    System.out.println("Task running...");
}
```

---

**Real-life Example**

Like a **calculator integrated into a blueprint**:

- In @Value → quickly calculate a value for one machine
- In XML → blueprint applies calculation to many machines
- In annotations → dynamic scheduling or conditional logic

---

**Interview Tips / Questions**

**Q:** Can SpEL be used in XML and annotations?
**A:** Yes, SpEL works in XML, @Value, and other annotations.
**Q:** How do you refer to a bean in SpEL?
**A:** Using @beanName syntax.
**Q:** Can SpEL call methods or access static fields?
**A:** Yes, using @beanName.method() or T(className).staticMethod().

---

**One-Line Summary**

SpEL can be used in **@Value, XML, and annotations** to inject dynamic, calculated, or bean-referenced values.

---

---

## JDBC Template (Spring)

**Definition**

JdbcTemplate is a **Spring helper class** that simplifies **database operations** like querying, updating, and calling stored procedures using JDBC.

---

**Key Points**

- Reduces boilerplate JDBC code (connection, statement, resultset handling)
- Handles **exceptions internally** and translates to Spring exceptions
- Supports **CRUD operations** and batch updates

---

**Purpose / Importance**

- Makes **database access easier and safer**
- Avoids repetitive try-catch-finally code
- Provides **template methods** for common operations

---

**Syntax / Example**

**Bean Configuration**

```
@Configuration
public class AppConfig {

    @Bean
    public DataSource dataSource() {
        DriverManagerDataSource ds = new DriverManagerDataSource();
        ds.setDriverClassName("oracle.jdbc.driver.OracleDriver");
        ds.setUrl("jdbc:oracle:thin:@localhost:1521:xe");
        ds.setUsername("system");
```

```
        ds.setPassword("password");
        return ds;
    }

    @Bean
    public JdbcTemplate jdbcTemplate(DataSource ds) {
        return new JdbcTemplate(ds);
    }
}
```

**Using JdbcTemplate**

```
@Autowired
private JdbcTemplate jdbcTemplate;

// Insert operation
public int addUser(User user) {
    String sql = "INSERT INTO users(id, name) VALUES(?, ?)";
    return jdbcTemplate.update(sql, user.getId(), user.getName());
}

// Query operation
public List<User> getAllUsers() {
    String sql = "SELECT * FROM users";
    return jdbcTemplate.query(sql, (rs, rowNum) ->
        new User(rs.getInt("id"), rs.getString("name"))
    );
}
```

---

**Real-life Example**

Like **a pre-filled form**: Instead of filling all fields manually every time (managing connections, statements, etc.), JdbcTemplate **automatically handles the repetitive tasks.**

---

**Interview Tips / Questions**

**Q:** What is the main advantage of JdbcTemplate?
**A:** Simplifies JDBC operations and handles resource management.
**Q:** Does JdbcTemplate handle exceptions?
**A:** Yes, it converts checked SQLExceptions to **DataAccessException.**
**Q:** Can JdbcTemplate perform batch operations?
**A:** Yes, using batchUpdate() method.

---

**One-Line Summary**

JdbcTemplate is a Spring utility that **simplifies JDBC database operations**, reducing boilerplate code and improving safety.

---

---

## NamedParameterJdbcTemplate (Spring)

**Definition**

NamedParameterJdbcTemplate is a Spring class that **enhances JdbcTemplate** by allowing **named parameters** in SQL queries instead of using only ? placeholders.

---

**Key Points**

- Uses **named parameters** (:paramName) for readability
- Reduces errors when using **many parameters**
- Supports **Map or BeanPropertySqlParameterSource** for passing values

---

**Purpose / Importance**

- Improves **code readability** and maintainability
- Avoids confusion with multiple ? in queries
- Simplifies **binding values** from beans or maps

---

**Syntax / Example**

**Bean Configuration**

```
@Bean
public NamedParameterJdbcTemplate namedParameterJdbcTemplate(DataSource ds) {
    return new NamedParameterJdbcTemplate(ds);
}
```

**Using Named Parameters**

```
@Autowired
private NamedParameterJdbcTemplate npJdbcTemplate;

// Insert operation
public int addUser(User user) {
    String sql = "INSERT INTO users(id, name) VALUES(:id, :name)";
    Map<String, Object> params = new HashMap<>();
    params.put("id", user.getId());
    params.put("name", user.getName());
    return npJdbcTemplate.update(sql, params);
}

// Query operation
public List<User> getUserByName(String name) {
    String sql = "SELECT * FROM users WHERE name = :name";
    Map<String, Object> params = Map.of("name", name);
    return npJdbcTemplate.query(sql, params, (rs, rowNum) ->
        new User(rs.getInt("id"), rs.getString("name"))
    );
}
```

**Using BeanPropertySqlParameterSource**

```
@BeanPropertySqlParameterSource param = new BeanPropertySqlParameterSource(user);
npJdbcTemplate.update(sql, param);
```

---

**Real-life Example**

Like **labeling boxes with names instead of numbers**:

- Easier to know which item goes where
- Avoids mixing things up

---

**Interview Tips / Questions**

**Q:** Difference between JdbcTemplate and NamedParameterJdbcTemplate?
**A:** NamedParameterJdbcTemplate supports **named parameters**, JdbcTemplate uses ? only.
**Q:** Why use NamedParameterJdbcTemplate?
**A:** Improves readability and reduces errors with multiple parameters.

**Q:** Can it work with beans directly?
**A:** Yes, via BeanPropertySqlParameterSource.

---

**One-Line Summary**
NamedParameterJdbcTemplate allows **SQL queries with named parameters**, making database code **cleaner and easier to maintain**.

---

---

## RowMapper (Spring JDBC)

**Definition**
RowMapper is an interface in Spring JDBC used to **map each row of a ResultSet to a Java object**.

---

**Key Points**

- Converts **database rows → Java objects**
- Implement mapRow(ResultSet rs, int rowNum) method
- Used with JdbcTemplate and NamedParameterJdbcTemplate

---

**Purpose / Importance**

- Separates **SQL logic** from **object mapping**
- Makes **query results easier to work with** in Java
- Supports **custom mapping** for complex objects

---

**Syntax / Example**

**Using Inline Lambda**

```
List<User> users = jdbcTemplate.query(
    "SELECT * FROM users",
    (rs, rowNum) -> new User(rs.getInt("id"), rs.getString("name"))
);
```

**Using Custom RowMapper Class**

```
public class UserRowMapper implements RowMapper<User> {
    @Override
    public User mapRow(ResultSet rs, int rowNum) throws SQLException {
        return new User(rs.getInt("id"), rs.getString("name"));
    }
}

// Usage
List<User> users = jdbcTemplate.query("SELECT * FROM users", new UserRowMapper());
```

---

**Real-life Example**
Like **translating a table row into a form object**:

- Each row = one form
- Converts raw data into usable structured object

---

**Interview Tips / Questions**
**Q:** What is RowMapper used for?
**A:** Maps each database row to a Java object.
**Q:** Can RowMapper be used with NamedParameterJdbcTemplate?
**A:** Yes, works with any Spring JDBC query methods.

**Q:** Difference between inline RowMapper and custom class?
**A:** Inline = quick, simple queries; custom class = reusable and clean for multiple queries.

---

**One-Line Summary**
RowMapper maps **each row of a ResultSet** to a **Java object**, making database results easier to use.

---

---

## @Transactional (Spring)

**Definition**
@Transactional is a Spring annotation used to **manage database transactions declaratively**, ensuring **commit or rollback** based on execution success.

---

**Key Points**

- Can be applied at **class or method level**
- Supports **propagation, isolation, timeout, read-only** options
- Automatically **rolls back on runtime exceptions**

---

**Purpose / Importance**

- Maintains **data integrity** in case of failures
- Reduces **manual transaction management** with JDBC
- Ensures **ACID properties** in database operations

---

**Syntax / Example**

**Basic Usage:**

```
@Service
public class UserService {

    @Autowired
    private JdbcTemplate jdbcTemplate;

    @Transactional
    public void createUserAndLog(User user) {
        jdbcTemplate.update("INSERT INTO users(id, name) VALUES(?, ?)", user.getId(),
user.getName());
        jdbcTemplate.update("INSERT INTO logs(id, action) VALUES(?, ?)", user.getId(),
"User created");
        // If exception occurs here, both inserts will rollback
    }
}
```

**With Propagation and Isolation:**

```
@Transactional(propagation = Propagation.REQUIRED, isolation =
Isolation.READ_COMMITTED, timeout = 5, readOnly = false)
public void updateUser(User user) { ... }
```

---

**Real-life Example**
Like **bank money transfer**:

- Debit from one account
- Credit to another account

- If anything fails, **both operations are rolled back** to avoid loss

---

**Interview Tips / Questions**

- **Q:** What is the default rollback behavior?
  **A:** Rolls back on unchecked exceptions (RuntimeException), commits on checked exceptions.
- **Q:** Can @Transactional be applied on private methods?
  **A:** No, Spring AOP proxy cannot intercept private methods.
- **Q:** Difference between REQUIRED and REQUIRES_NEW propagation?
  **A:** REQUIRED joins existing transaction; REQUIRES_NEW creates a new one.

---

**One-Line Summary**

@Transactional manages **database transactions declaratively**, ensuring **commit or rollback** automatically.

---

---

## Transaction Propagation (Spring)

**Definition**

Transaction Propagation defines **how Spring transactions behave when multiple transactional methods call each other**.

---

**Key Points**

- Determines **whether to join an existing transaction or start a new one**
- Configured using propagation attribute in @Transactional
- Helps maintain **consistency across nested service calls**

---

**Purpose / Importance**

- Controls **nested transactions** behavior
- Ensures **data integrity** in complex service chains
- Avoids **partial commits** or unexpected rollbacks

---

**Types of Propagation**

| Propagation Type | Behavior |
|---|---|
| **REQUIRED** | Join existing transaction; create new if none |
| **REQUIRES_NEW** | Suspend existing transaction, create a new one |
| **SUPPORTS** | Join existing transaction; execute non-transactionally if none |
| **NOT_SUPPORTED** | Suspend existing transaction; execute non-transactionally |
| **MANDATORY** | Must join existing transaction; throw exception if none |
| **NEVER** | Must run non-transactionally; throw exception if transaction exists |
| **NESTED** | Executes within a nested transaction (savepoint support) |

**Example:**

```
@Transactional(propagation = Propagation.REQUIRED)
public void parentMethod() {
    childService.childMethod(); // joins parent's transaction
}

@Transactional(propagation = Propagation.REQUIRES_NEW)
public void childMethod() {
    // executes in a separate transaction
}
```

**Real-life Example**

- **REQUIRED:** Two bank operations share the same transaction.
- **REQUIRES_NEW:** A logging operation always commits, even if main transaction fails.

---

**Interview Tips / Questions**

**Q:** Default propagation type in Spring?
**A:** REQUIRED
**Q:** Difference between REQUIRED and REQUIRES_NEW?
**A:** REQUIRED joins existing, REQUIRES_NEW suspends existing and creates a new one.
**Q:** When to use NESTED?
**A:** For partial rollbacks within a larger transaction (savepoints).

---

**One-Line Summary**

Transaction Propagation controls **how transactions behave when nested or multiple methods are involved**, ensuring data consistency.

---

---

## Maven Project (Java / Spring)

**Definition**

Maven is a **build automation and project management tool** for Java projects, which manages **dependencies, builds, and project structure.**

---

**Key Points**

- Uses a **Project Object Model (POM) XML** file to define project info, dependencies, and build settings
- Automatically **downloads required libraries** from Maven repositories
- Supports **standardized project structure and lifecycle**

---

**Purpose / Importance**

- Simplifies **dependency management**
- Standardizes **project structure** across teams
- Automates **build, test, and deployment** tasks
- Ensures **reproducible builds**

---

**Syntax / Example**

**pom.xml example:**

```
<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>spring-app</artifactId>
    <version>1.0.0</version>

    <dependencies>
        <!-- Spring Core -->
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-context</artifactId>
            <version>5.3.30</version>
```

```
        </dependency>
        <!-- JUnit for testing -->
        <dependency>
            <groupId>junit</groupId>
            <artifactId>junit</artifactId>
            <version>4.13.2</version>
            <scope>test</scope>
        </dependency>
    </dependencies>
</project>
```

**Common Maven commands:**

```
mvn clean      # Removes target folder
mvn compile    # Compiles source code
mvn test       # Runs unit tests
mvn package    # Builds jar/war file
mvn install    # Installs jar to local repo
```

---

**Real-life Example**

Like a **recipe book**:

- Lists all **ingredients (dependencies)**
- Provides **steps to cook (build lifecycle)**
- Ensures **same dish (project) every time**

---

**Interview Tips / Questions**

**Q:** What is the role of pom.xml?
**A:** Defines project info, dependencies, and build instructions.
**Q:** Difference between mvn package and mvn install?
**A:** package creates jar/war; install adds it to local Maven repo.
**Q:** What is Maven's default project structure?
**A:**

- src/main/java
- src/main/resources
- src/test/java
- src/test/resources

---

**One-Line Summary**

Maven automates **dependency management, building, and project lifecycle**, standardizing Java projects across teams.

---

---

## Maven Project Structure

**Definition**

Maven is a **build automation tool** for Java projects, and its standard **project structure** organizes code, resources, and dependencies consistently.

---

**Key Points**

- **Standardized layout** ensures consistency across projects
- Separates **source code, tests, and resources**
- pom.xml manages **dependencies, plugins, and build configuration**

---

**Purpose / Importance**

- Simplifies **building, compiling, and packaging projects**

- Makes **team collaboration easier**
- Supports **dependency management and plugin execution** automatically

---

**Standard Maven Directory Layout**

```
project-root/
│
├─ src/
│  ├─ main/
│  │  ├─ java/       → Application source code
│  │  ├─ resources/  → Config files, properties
│  │  └─ webapp/     → JSP, HTML, static files (for web apps)
│  │
│  └─ test/
│     ├─ java/       → Test source code
│     └─ resources/  → Test resources
│
├─ target/           → Compiled output and packaged files
├─ pom.xml           → Maven project configuration
└─ README.md
```

**Important Files:**

- pom.xml → manages dependencies, plugins, project info
- src/main/java → main code
- src/test/java → unit tests

---

**Real-life Example**

Like a **well-organized office**:

- main/java = employees doing work
- resources = office supplies/configs
- test/java = quality checks

---

**Interview Tips / Questions**

**Q:** Default source directory in Maven?
**A:** src/main/java
**Q:** Where do you put configuration files?
**A:** src/main/resources
**Q:** What is pom.xml used for?
**A:** Dependency management, build plugins, and project metadata

---

**One-Line Summary**

Maven standard project structure **organizes code, resources, and tests consistently**, simplifying build and collaboration.

---

---

## POM (Project Object Model)

**Definition**

POM is an **XML file (pom.xml)** in a Maven project that **defines project configuration, dependencies, build settings, and plugins.**

---

**Key Points**

- Core of **Maven project configuration**
- Manages **dependencies, build plugins, and project metadata**
- Supports **multi-module projects and inheritance**

**Purpose / Importance**

- Centralizes project configuration in one file
- Automatically **downloads and manages dependencies**
- Simplifies **build, testing, and packaging processes**

---

**Structure / Example**

```
<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>1.0.0</version>
    <packaging>jar</packaging>

    <name>My Application</name>
    <description>Example Maven Project</description>

    <!-- Dependencies -->
    <dependencies>
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-core</artifactId>
            <version>5.3.28</version>
        </dependency>
    </dependencies>

    <!-- Build Plugins -->
    <build>
        <plugins>
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-compiler-plugin</artifactId>
                <version>3.10.1</version>
                <configuration>
                    <source>1.8</source>
                    <target>1.8</target>
                </configuration>
            </plugin>
        </plugins>
    </build>
</project>
```

**Key Sections:**

- groupId, artifactId, version → uniquely identify project
- dependencies → external libraries
- build/plugins → compile, package, or custom tasks

---

**Real-life Example**

Like a **recipe card**:

- Lists ingredients (dependencies)
- Provides cooking steps (build plugins)
- Gives metadata (name, version, author)

---

**Interview Tips / Questions**
**Q:** What is the purpose of pom.xml?
**A:** Defines project metadata, dependencies, and build configuration.
**Q:** Difference between dependency and plugin in POM?
**A:** Dependencies are libraries used in code; plugins are tools for building/packaging.
**Q:** Can POM inherit another POM?
**A:** Yes, parent POM allows inheritance and shared configuration.

---

**One-Line Summary**
POM is the **central configuration file in Maven**, managing dependencies, build, and project metadata.

---

---

## Maven Build Lifecycle

**Definition**
The Maven **Build Lifecycle** is a **sequence of phases** that defines how a Maven project is **built, tested, and packaged**.

---

**Key Points**
- **Default lifecycle** handles project compilation, testing, and packaging
- Consists of **phases executed in order**
- Plugins are attached to phases to perform tasks automatically

---

**Purpose / Importance**
- Automates **build, test, and deployment** processes
- Ensures **consistent builds across environments**
- Simplifies **project management for developers and teams**

---

**Lifecycle Phases / Examples**
**Default Lifecycle (most common for Java projects)**

| Phase | Purpose |
|---|---|
| validate | Checks if project is correct and all info is available |
| compile | Compiles source code in src/main/java |
| test | Runs unit tests in src/test/java |
| package | Packages compiled code into JAR/WAR |
| verify | Runs checks on the package (integration tests) |
| install | Installs package into local repository (~/.m2/repository) |
| deploy | Copies package to remote repository for sharing |

**Example Commands:**
```
mvn compile       # Compile source code
mvn test          # Run unit tests
mvn package       # Create JAR/WAR
mvn install       # Install in local repository
```

**Clean Lifecycle (optional)**

| Phase | Purpose |
|---|---|
| pre-clean | Actions before cleaning |
| clean | Deletes target/ directory |
| post-clean | Actions after cleaning |

**Site Lifecycle**

| Phase | Purpose |
|---|---|
| pre-site | Prepare for site generation |
| site | Generates project documentation |
| post-site | Actions after site generation |
| site-deploy | Deploy site to web server |

---

**Real-life Example**
Like **a cooking process**:

- validate → check ingredients
- compile → chop & mix
- test → taste test
- package → serve dish
- install → put in fridge for reuse

---

**Interview Tips / Questions**
**Q:** What is the default Maven lifecycle?
**A:** default (handles build, test, package, install, deploy).
**Q:** Difference between compile and package?
**A:** compile = converts source to bytecode, package = creates JAR/WAR.
**Q:** What does mvn clean install do?
**A:** Cleans target/ directory and installs the package to local repository.

---

**One-Line Summary**
Maven Build Lifecycle is a **sequence of phases** that automates **building, testing, packaging, and deploying projects**.

---

---

## Maven Plugins

**Definition**
Maven Plugins are **tools that perform specific tasks during the build lifecycle**, like compiling code, running tests, or packaging the project.

---

**Key Points**

- Extend Maven's **build capabilities**
- Attached to **lifecycle phases** to perform automatic actions
- Can be **official, third-party, or custom plugins**

---

**Purpose / Importance**

- Automates repetitive build tasks
- Ensures **consistent build processes** across projects
- Supports **compilation, testing, packaging, deployment, and documentation**

---

**Syntax / Example**
**Adding a Plugin in pom.xml:**

```
<build>
    <plugins>
        <!-- Compiler Plugin -->
        <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-compiler-plugin</artifactId>
```

```
            <version>3.10.1</version>
            <configuration>
                <source>1.8</source>
                <target>1.8</target>
            </configuration>
        </plugin>

        <!-- Surefire Plugin (for unit tests) -->
        <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-surefire-plugin</artifactId>
            <version>3.0.0-M7</version>
        </plugin>
    </plugins>
</build>
```

**Common Maven Plugins:**

- maven-compiler-plugin → Compiles Java code
- maven-surefire-plugin → Runs unit tests
- maven-jar-plugin → Creates JAR
- maven-clean-plugin → Cleans target/ folder
- maven-deploy-plugin → Deploys artifact to remote repository

---

**Real-life Example**

Like **machines in a factory**:

- Each machine (plugin) performs a **specific task** automatically during production (build).

---

**Interview Tips / Questions**

**Q:** Difference between Maven Plugin and Dependency?
**A:** Plugin = executes tasks; Dependency = external library used in code.
**Q:** Can plugins have configuration?
**A:** Yes, via <configuration> in pom.xml.
**Q:** Name two common Maven plugins.
**A:** maven-compiler-plugin, maven-surefire-plugin.

---

**One-Line Summary**

Maven Plugins **extend the build process**, automating tasks like compilation, testing, packaging, and deployment.

---

---

**Running Maven Builds**

**Definition**

Running Maven Builds means **executing Maven commands** to perform tasks like **compiling, testing, packaging, and installing** projects according to the **build lifecycle**.

---

**Key Points**

- Uses **mvn commands** in terminal/IDE
- Executes **specific lifecycle phases or goals**
- Can combine multiple commands like clean install

---

**Purpose / Importance**

- Automates **building, testing, and deploying projects**
- Ensures **consistent project builds across environments**
- Helps in **CI/CD integration** for professional development

---

**Common Commands / Examples**

**Basic Commands**

```
mvn compile       # Compile source code
mvn test          # Run unit tests
mvn package       # Create JAR/WAR file
mvn install       # Install package to local repository
mvn deploy        # Deploy artifact to remote repository
```

**Combined Commands**

```
mvn clean install   # Clean previous build and install new build
mvn clean package   # Clean previous build and create package
```

**Running Specific Plugin Goals**

```
mvn compiler:compile   # Only execute compile goal
mvn surefire:test      # Only run tests
```

**Skipping Tests**

```
mvn install -DskipTests   # Build and install without running tests
```

---

**Real-life Example**

Like **a production line**:

- compile → assemble parts
- test → quality check
- package → pack product
- install → store in warehouse
- deploy → ship to customer

---

**Interview Tips / Questions**

**Q:** How do you compile a Maven project?
**A:** mvn compile
**Q:** How do you skip tests during Maven build?
**A:** mvn install -DskipTests
**Q:** Difference between mvn package and mvn install?
**A:** package = creates JAR/WAR; install = installs it to local Maven repository.

---

**One-Line Summary**

Running Maven Builds executes **lifecycle phases or plugin goals**, automating **compile, test, package, and deploy tasks**.

---