**Dependency Injection (DI) in Spring**

Dependency Injection is a form of Inversion of Control (IoC) where an external container (the Spring IoC Container) is responsible for providing (“injecting”) the required dependencies into your objects, rather than having the objects construct or look up those dependencies themselves. This promotes loose coupling, easier testing, and clearer separation of concerns.

**Why DI Came to Be**

* **Tight Coupling of Components:** Traditional code often used new to create dependencies, making classes hard to replace or mock.
* **Difficult Testing:** Hard-coded dependencies impeded unit testing.
* **Configuration Sprawl:** Applications grew large; managing dependencies manually (factory classes, service locators) became brittle.
* **Enterprise Needs:** Early Java EE (EJB) was heavyweight; Spring introduced a lightweight container with declarative DI support.

**Core Rules of Spring DI**

1. **Beans**: Any Spring-managed object is a “bean.”
2. **Configuration Metadata**: XML, Java-based @Configuration classes, or annotations (@Component, @Bean).
3. **Wiring**: Spring reads metadata, instantiates beans, resolves dependencies, and injects them.
4. **Lifecycle**: Container manages creation and destruction according to bean scope.

**1. Field Injection**

Injecting dependencies directly into fields using @Autowired (and optionally @Qualifier).

@Component

public class OrderService {

@Autowired

private PaymentGateway paymentGateway;

// ...

}

**Rules & Characteristics**

* **No setter or constructor required.**
* **Reflection-based:** Spring sets private fields via reflection.
* **Late binding:** Injection occurs after object construction.

**When & Where to Use**

1. **Quick prototypes** or small apps where brevity matters.
2. **Legacy code retrofit** where adding constructors/setters is cumbersome.

⚠️ **Drawback:** Harder to unit-test/mocking because you can’t pass mocks via constructor easily and the private fields are hidden.

**Real-World Examples**

* **Controller Layer:**

@RestController

public class UserController {

@Autowired

private UserService userService;

// ...

}

* **Event Listeners:**

@Component

public class AuditListener {

@Autowired   
 private AuditRepository auditRepo;

@EventListener

public void onUserLogin(UserLoginEvent e) {…}

}

* **Feature Toggles:**

@Component

public class FeatureToggleService {

@Autowired

private ToggleConfig toggleConfig;

// ...

}

**2. Constructor Injection**

Injecting dependencies through a class constructor.

@Service

public class InvoiceService {

private final TaxCalculator taxCalculator;

private final DiscountService discountService;

@Autowired // optional on single constructor

public InvoiceService(TaxCalculator taxCalculator, DiscountService discountService) {

this.taxCalculator = taxCalculator;

this.discountService = discountService;

}

// ...

}

**Rules & Characteristics**

1. **final dependencies:** Ideal for immutable references.
2. **Null-safety:** Guarantees all required dependencies are provided at instantiation.
3. **Best practice:** Recommended by Spring and many style guides.

**When & Where to Use**

* **Mandatory dependencies:** When a bean cannot function without certain collaborators.
* **Immutable design:** Encourage clear API and thread safety.
* **Unit testing:** You can easily pass mock implementations in tests.

**Real-World Examples**

1. **Payment Processing:**

@Component

public class PaymentService {  
@Autowired

public PaymentService(PaymentGateway gateway, FraudChecker checker) { … }

}

1. **Messaging:**

@Service

public class NotificationService {  
@Autowired

public NotificationService(MessageSender sender, TemplateEngine engine) { … }

}

1. **Data Access:**

@Repository

public class CustomerRepository {  
@Autowired

public CustomerRepository(JdbcTemplate jdbc) { … }

}

**3. Setter Injection**

Injecting dependencies via JavaBean setter methods.

@Component

public class ReportGenerator {

private DataSource dataSource;

@Autowired

public void setDataSource(DataSource dataSource) {

this.dataSource = dataSource;

}

// ...

}

**Rules & Characteristics**

* **Optional dependencies:** Setters can be omitted if not needed.
* **Mutable:** Dependencies can be changed after instantiation (though rarely desirable).
* **Circular references:** Spring can resolve simple circular dependencies using setter injection.

**When & Where to Use**

1. **Optional collaborators:** Features toggled on/off.
2. **Large number of dependencies:** To avoid long constructors.
3. **Circular dependency resolution:** If two beans depend on each other.

**Real-World Examples**

* **Optional Configuration:**

@Component

public class CacheManager {

private CacheProperties props;

@Autowired(required=false)

public void setProps(CacheProperties props) { this.props = props; }

}

* **Plugin Systems:**

@Component

public class ImageProcessor {

private List<FilterPlugin> plugins;

@Autowired

public void setPlugins(List<FilterPlugin> plugins) { this.plugins = plugins; }

}

* **Email Notifications:**

@Services

public class EmailService {

private SmtpConfig config;

@Autowired

public void setConfig(SmtpConfig config) { this.config = config; }

}

**Choosing Between Injection Types**

| **Aspect** | **Constructor** | **Setter** | **Field** |
| --- | --- | --- | --- |
| Immutability | ✔️ | ❌ | ❌ |
| Mandatory deps | ✔️ | ❌ | ❌ |
| Circular deps | ❌ | ✔️ | ✔️ (but hidden) |
| Ease of testing | ✔️ (pass mocks) | ✔️ | ❌ (requires reflection/more setup) |
| Boilerplate | ↑ more code | Moderate | ↓ less code |

## **Spring Bean Scopes**

Bean scope determines the lifecycle and visibility of a bean in the Spring IoC container.

| **Scope** | **Lifecycle** | **Use Case / When to Use** |
| --- | --- | --- |
| **singleton** | One shared instance per Spring container (default). | Stateless services, thread-safe utility beans. |
| **prototype** | A new instance each time requested/injected. | State-full beans, per-use tasks (e.g., document builders). |
| **request** | One instance per HTTP request (Web only). | Per-request web controllers, request-scoped form handlers. |
| **session** | One instance per HTTP session (Web only). | Shopping carts, user session data. |
| **application** | One instance per ServletContext (Web only). | App-wide shared context, e.g., application settings loader. |
| **websocket** | One instance per WebSocket (Spring 5+) | WebSocket handlers maintaining connection state. |

**Configuring Scope**

1. **Annotation:**

@Component

@Scope("prototype")

public class ReportBuilder { … }

1. **XML:**

<bean id="reportBuilder" class="…" scope="prototype"/>

**Real-World Examples**

* **singleton**
  + UserService, OrderService, CacheManager → thread-safe, stateless, high-throughput components.
* **prototype**
  + DocumentParser that holds parsing context, SpreadsheetRowMapper that accumulates row data, WorkflowContext for each process instance.
* **request**
  + FormBackingObject in a Spring MVC handler, LocaleResolver storing per-request locale preferences.
* **session**
  + ShoppingCart bean holding items for the current user’s session, UserPreferences bean loaded on login.
* **application**
  + AppConfigLoader reading global settings from a file only once per web application, VersionInfo loaded at startup.
* **websocket**
  + ChatSessionHandler maintaining state for each WebSocket connection, GameSession in a multiplayer game.

**Summary**

1. **Field Injection** is quick but less testable and hides dependencies.
2. **Constructor Injection** is the most robust, promoting immutability and clear contracts.
3. **Setter Injection** handles optional or circular dependencies but allows mutable beans.
4. **Bean Scopes** let you control bean lifecycles from application-wide singletons to per-request/session instances in web contexts.

**Autowiring in Spring**

Autowiring is Spring’s ability to automatically resolve and inject collaborating beans into your components, without explicit <bean> wiring in XML or manual @Bean methods. It builds on the IoC/DI foundation to further reduce boilerplate and let the container “wire by type” (or name) for you.

**Why Autowiring Came to Be**

1. **Reduce XML/Java Config Verbosity**: Declaring every dependency by hand in XML or in @Bean methods becomes tedious for large apps.
2. **Promote Convention over Configuration**: If there’s exactly one suitable bean, Spring can inject it automatically.
3. **Speed Up Development**: Faster prototyping by letting the container infer the matches.

**Core Rules & Modes**

* **@Autowired** on constructors, setters, or fields triggers autowiring by type.
* **Required by default**: If Spring cannot find a bean of the required type, it throws NoSuchBeanDefinitionException; use @Autowired(required=false) for optional dependencies.
* **Disambiguation**:
  + **@Primary**: marks a bean as the default when multiple candidates exist.
  + **@Qualifier("beanName")**: when you need a specific bean by name.
* **Mode** (XML-only): <bean autowire="byName|byType|constructor">—mostly superseded by annotations.

**When & Where to Use**

* **Constructor injection**: for mandatory dependencies—use @Autowired on the sole constructor (or omit it on Spring 4.3+).
* **Setter injection**: for optional or mutable dependencies—annotate the setter.
* **Field injection**: when brevity is more important than testability (e.g., quick prototypes).
* **Lists/Maps of beans**: inject all beans of a type via List<MyInterface> or Map<String, MyInterface>.

**Real-World Examples**

1. **Service → Repository**

@Service

public class OrderService {

private final OrderRepository repo;

@Autowired // on single constructor, this is optional

public OrderService(OrderRepository repo) {

this.repo = repo;

}

// …

}

*Spring injects the one OrderRepository bean by type.*

1. **Multiple DataSources (Disambiguation)**

@Configuration

public class DataConfig {

@Bean @Primary

public DataSource mainDs() { … }

@Bean

public DataSource reportingDs() { … }

}

@Component

public class ReportService {

@Autowired

@Qualifier("reportingDs")

private DataSource ds;

// …

}

*Here @Qualifier picks the non-primary reportingDs.*

1. **Plugin Collection Injection**

public interface PaymentPlugin { … }

@Component

public class PaypalPlugin implements PaymentPlugin { … }

@Component

public class StripePlugin implements PaymentPlugin { … }

@Service

public class CheckoutService {

private final List<PaymentPlugin> plugins;

@Autowired

public CheckoutService(List<PaymentPlugin> plugins) {

this.plugins = plugins;

}

// plugins contains both PaypalPlugin and StripePlugin

}

1. **Optional Feature Toggle**

@Component

public class MetricsService {

private MetricsCollector collector;

@Autowired(required=false)

public void setCollector(MetricsCollector collector) {

this.collector = collector;

}

// collector may be null if no bean available

}

**Spring Annotations**  
Spring provides a rich set of annotations to declare beans, configure behavior, and enable cross-cutting concerns—all in Java code and on your classes/methods.

**Why Annotation-Driven Config Came to Be**

1. **Type-Safety & Refactoring**: IDEs know about annotations; XML strings are brittle.
2. **Co-locate Metadata**: Annotating a class directly keeps configuration alongside implementation.
3. **Reduce Verbosity**: Annotated components and Java @Configuration classes replace large XML files.

**Core Categories & Rules**

| **Category** | **Key Annotations** | **Notes** |
| --- | --- | --- |
| **Stereotypes** | @Component / @Service / @Repository / @Controller / @RestController | Marks class for component-scanning. |
| **Java Config** | @Configuration, @Bean, @Import | Define configuration classes and factory methods. |
| **Injection** | @Autowired, @Qualifier, @Value | Inject beans, specific beans, and external properties. |
| **Lifecycle** | @PostConstruct, @PreDestroy | Hook into bean initialization and destruction. |
| **Transaction / AOP** | @Transactional, @Aspect, @Before, @Around | Declare transactional boundaries and aspects. |
| **Web / MVC** | @RequestMapping, @GetMapping, @PathVariable, @RequestBody | Map HTTP routes to methods and bind parameters. |
| **Scheduling / Events** | @Scheduled, @Async, @EventListener | Schedule tasks, run async methods, or listen to events. |

**When & Where to Use**

* **@Component-family**: on any class you want Spring to detect via @ComponentScan.
* **@Configuration / @Bean**: in dedicated config classes—ideal for third-party or programmatic bean definitions.
* **@Value**: to inject simple externalized properties (application.properties or env vars).
* **@Transactional**: on service methods or classes that require ACID semantics.
* **MVC Annotations**: exclusively in web controller classes.
* **AOP Annotations**: on cross-cutting concern classes (logging, security, etc.).

**Real-World Examples**

1. **REST API Controller**

@RestController

@RequestMapping("/api/users")

public class UserController {

@Autowired

private UserService service;

@GetMapping("/{id}")

public ResponseEntity<User> getUser(@PathVariable Long id) {

return ResponseEntity.of(service.findById(id));

}

}

1. **Java-Based Configuration**

@Configuration

@PropertySource("classpath:mail.properties")

public class MailConfig {

@Value("${mail.host}") private String host;

@Value("${mail.port}") private int port;

@Bean

public JavaMailSender mailSender() {

JavaMailSenderImpl sender = new JavaMailSenderImpl();

sender.setHost(host);

sender.setPort(port);

return sender;

}

}

1. **Transactional Service**

@Service

public class BankService {

@Autowired private AccountRepository repo;

@Transactional

public void transfer(Long fromId, Long toId, BigDecimal amt) {

repo.debit(fromId, amt);

repo.credit(toId, amt);

}

}

1. **Scheduled Task & Event Listener**

@Component

public class MetricsCollector {

@Scheduled(fixedRateString = "${metrics.rate:60000}")

public void collect() { … }

@EventListener

public void onAppEvent(ApplicationReadyEvent e) {

// initialize metrics on startup

}

}

**Summary**

* **Autowiring** streamlines dependency injection by letting Spring infer and inject beans by type (or name), reducing explicit wiring.
* **Spring Annotations** provide a concise, type-safe, and co-located way to declare components, configuration, injection points, lifecycle hooks, transactions, web endpoints, scheduling, and more.

## **Why Annotation-Driven Configuration?**

* **Type-Safe & Refactorable**: IDEs understand annotations; renames/update flows.
* **Co-locate Metadata**: Configuration lives with the class or method rather than dispersed XML.
* **Reduced Boilerplate**: Fewer XML tags, more declarative Java.
* **Convention over Configuration**: Default behaviors let you write less.

**1. Stereotype Annotations**

Mark classes for component-scanning so Spring auto-detects and registers them as beans.

| **Annotation** | **Role** | **Rule/Behavior** |
| --- | --- | --- |
| @Component | Generic component | Base annotation—detects any class as a bean. |
| @Service | Business service layer | Semantically marks a service—no functional difference. |
| @Repository | Data access layer | Translates persistence exceptions into Spring’s DataAccessException. |
| @Controller | MVC controller | Spring MVC dispatches HTTP requests here. |
| @RestController | RESTful controller | Combines @Controller + @ResponseBody on methods. |

**When & Where**

* **@Component**: for utility classes or cross-cutting concerns (e.g., a custom converter).
* **@Service**: on your core domain/business-logic classes.
* **@Repository**: on DAOs or Spring Data interfaces.
* **@Controller/@RestController**: in your web layer to expose endpoints.

**Real-World Examples**

1. **Email Sending**

@Service

public class EmailService { … }

1. **JPA Repository**

@Repository

public interface UserRepository extends JpaRepository<User, Long> { … }

1. **Generic Utility**

@Component

public class CsvParser { … }

1. **Web MVC**

@Controller

public class HomeController { … }

1. **REST API**

@RestController

@RequestMapping("/api/orders")

public class OrderRestController { … }

**2. Configuration & Bean Definitions**

Define beans or import configurations programmatically.

| **Annotation** | **Role** | **Rule/Behavior** |
| --- | --- | --- |
| @Configuration | Configuration class container | Implicit @Bean scanning + CGLIB proxy to enforce singleton. |
| @Bean | Factory method for a bean | Method return value is registered as a bean. |
| @Import | Import other config classes | Pull in additional @Configuration classes. |
| @PropertySource | Load external property files | Makes @Value injections possible from that source. |

**When & Where**

* Use **@Configuration** to group related factory methods and enable conditional loading.
* Use **@Bean** when you need fine-grained control (third-party classes, complex setup).
* Use **@PropertySource** to load application-\*.properties or custom files.

**Real-World Examples**

1. **DataSource Configuration**

@Configuration

@PropertySource("classpath:db.properties")

public class DataSourceConfig {

@Bean

public DataSource dataSource(

@Value("${db.url}") String url,

@Value("${db.user}") String user,

@Value("${db.pass}") String pass) { … }

}

1. **Third-Party Client**

@Configuration

public class S3ClientConfig {

@Bean

public AmazonS3 s3Client(AwsCredentials creds) { … }

}

1. **Task Executor**

@Configuration

public class AsyncConfig {

@Bean

public TaskExecutor taskExecutor() {

ThreadPoolTaskExecutor exec = new ThreadPoolTaskExecutor();

exec.setCorePoolSize(10);

return exec;

}

}

**3. Dependency Injection Annotations**

Control how Spring injects bean dependencies.

| **Annotation** | **Role** | **Rule/Behavior** |
| --- | --- | --- |
| @Autowired | By-type autowiring | Required by default; optional with required=false. |
| @Qualifier | Disambiguate among beans | Must match bean name or qualifier value. |
| @Value | Inject property or literal | Supports SpEL (#{…}) and ${…} placeholders. |

**When & Where**

* **Constructor injection**: preferred for mandatory deps (omit @Autowired on single ctor).
* **Setter injection**: for optional or many smaller deps.
* **Field injection**: for prototypes or quick prototypes—use sparingly.
* Combine @Autowired + @Qualifier when multiple beans of the same type exist.

**Real-World Examples**

1. **Primary vs. Qualified DataSource**

@Autowired

@Qualifier("reportingDs")

private DataSource ds;

1. **External Config**

@Value("${app.upload.dir}")

private String uploadDir;

1. **Collection Injection**

@Autowired

public ReportService(List<ReportGenerator> generators) { … }

**4. AOP & Lifecycle Annotations**

Handle cross-cutting concerns and bean lifecycle hooks.

| **Annotation** | **Role** | **Rule/Behavior** |
| --- | --- | --- |
| @Transactional | Declarative transaction boundaries | Applies PlatformTransactionManager advice. |
| @Aspect | Define an aspect class | Combined with @EnableAspectJAutoProxy. |
| @Before, @After, @Around | Advice methods | Pointcut expressions match join points. |
| @PostConstruct | After bean creation | Runs once after DI, before first use. |
| @PreDestroy | Before bean destruction | Runs on context close (singleton only). |

**When & Where**

* **@Transactional**: on service methods modifying data.
* **@Aspect**: for logging, security checks, metrics.
* **@PostConstruct/@PreDestroy**: for custom init or clean-up logic.

**Real-World Examples**

1. **Transaction on Transfer**

@Transactional

public void transfer(…) { … }

1. **Logging Aspect**

@Aspect

public class LoggingAspect {

@Before("execution(\* com.app.service.\*.\*(..))")

public void logMethod(JoinPoint jp) { … }

}

1. **Init Resource**

@PostConstruct

public void initCache() { … }

@PreDestroy

public void shutdown() { … }

**Sample Project Structure**

src/main/java

└── com

└── example

├── Application.java // @SpringBootApplication

├── config

│ ├── DataSourceConfig.java // @Configuration, @PropertySource

│ └── AsyncConfig.java // @Configuration, @Bean

├── controller

│ ├── HomeController.java // @Controller

│ └── UserRestController.java // @RestController

├── service

│ ├── EmailService.java // @Service, @Transactional

│ ├── OrderService.java // @Service, @Autowired

│ └── LoggingAspect.java // @Aspect

├── repository

│ └── UserRepository.java // @Repository

├── model

│ └── User.java // Entity/POJO

└── util

└── CsvParser.java // @Component

* **Application.java**

@SpringBootApplication

public class Application {

public static void main(String[] args) {

SpringApplication.run(Application.class, args);

}

}

* **config/**: your Java-based configs with @Configuration + @Bean.
* **controller/**: MVC and REST controllers (@Controller, @RestController).
* **service/**: business logic (@Service), transaction boundaries, AOP aspects.
* **repository/**: data access interfaces (@Repository).
* **util/**: generic components (@Component).

**Putting It All Together**

1. **Spring Boot**’s @SpringBootApplication implicitly enables component scanning and auto-configuration.
2. **Configs** load properties and expose beans via @Configuration + @Bean.
3. **Components** (@Service, @Repository, @Component) are auto-discovered and injected via @Autowired.
4. **Controllers** handle web requests.
5. **AOP** and **transactions** wrap service methods declaratively.
6. **Lifecycle hooks** (@PostConstruct, @PreDestroy) let you manage resources.

This combination of annotations, clear project structure, and DI/AOP features is what makes Spring both powerful and maintainable in real-world enterprise applications.

## **Spring Dependency Injection & Related Concepts – MCQ Quiz (with answers)**

**1. Which of the following is NOT a valid way of performing Dependency Injection in Spring?**

a) Field Injection  
b) Constructor Injection  
c) Setter Injection  
d) Method Parameter Injection at runtime without Spring context  
  
**Answer:** d

**2. In Constructor Injection, dependencies are injected:**

a) After object creation using setters  
b) During object creation via constructor parameters  
c) Through reflection at runtime  
d) Using XML <property> tags only  
  
**Answer:** b

**3. What is a common disadvantage of Field Injection compared to Constructor Injection?**

a) It is more verbose  
b) It makes unit testing harder  
c) It requires more XML configuration  
d) It cannot be used with annotations  
  
**Answer:** b

**4. If a Spring bean is declared with scope "singleton", how many instances are created per Spring container?**

a) One instance per bean definition  
b) One per HTTP request  
c) One per user session  
d) One per class loader  
  
**Answer:** a

**5. Which Spring bean scope is not available in a non-web Spring application?**

a) singleton  
b) prototype  
c) request  
d) prototype-singleton hybrid  
  
**Answer:** c

**6. In Autowiring by type, Spring matches dependencies:**

a) By variable name  
b) By fully qualified class name  
c) By bean type  
d) By bean scope  
  
**Answer:** c

**7. Which of the following annotations is used to inject a dependency by type?**

a) @Autowired  
b) @Qualifier  
c) @Inject  
d) @Resource  
  
**Answer:** a

**8. Which annotation is used in Spring to define a bean without using XML <bean> tag?**

a) @Service  
b) @Configuration  
c) @Component  
d) @Bean  
  
**Answer:** c

**9. In Spring, @Bean annotation is typically placed inside:**

a) A @Component class  
b) A @Configuration class  
c) A @Service class  
d) A @Repository class  
  
**Answer:** b

**10. Which of the following is true for Prototype scoped beans?**

a) A new instance is created every time it’s requested  
b) Only one instance per Spring container  
c) It is only available in web applications  
d) It is destroyed automatically after use  
  
**Answer:** a

**11. If you want to inject a specific bean when multiple beans of the same type exist, you should use:**

a) @Autowired alone  
b) @Primary  
c) @Qualifier  
d) @Scope  
  
**Answer:** c

**12. Which of the following is NOT a stereotype annotation in Spring?**

a) @Controller  
b) @Component  
c) @Service  
d) @Inject  
  
**Answer:** d

**13. The @Scope("request") bean scope means:**

a) One bean instance per HTTP session  
b) One bean instance per Spring container  
c) One bean instance per HTTP request  
d) One bean per method call  
  
**Answer:** c

**14. Which annotation enables component scanning in Spring?**

a) @ComponentScan  
b) @EnableScanning  
c) @Autowired  
d) @BeanScan  
  
**Answer:** a

**15. Which of the following injection methods is recommended for mandatory dependencies?**

a) Field Injection  
b) Constructor Injection  
c) Setter Injection  
d) Static Method Injection  
  
**Answer:** b

**16. Which annotation is used to mark a method that should run after dependency injection is complete?**

a) @PostConstruct  
b) @AfterInit  
c) @InitMethod  
d) @Setup  
  
**Answer:** a

**17. What will happen if Spring cannot find a bean for a required @Autowired dependency?**

a) It will ignore it silently  
b) It will throw NoSuchBeanDefinitionException  
c) It will create a new empty bean  
d) It will use null  
  
**Answer:** b

**18. Which scope creates a bean once per HTTP session?**

a) singleton  
b) prototype  
c) session  
d) request  
  
**Answer:** c

**19. The @Repository annotation is mainly used for:**

a) Identifying DAO components  
b) Marking a controller class  
c) Defining configuration beans  
d) Handling service logic  
  
**Answer:** a

**20. Which of these is true about Autowiring in Spring?**

a) @Autowired can work with both constructor and setter injection  
b) @Autowired only works on fields  
c) @Autowired requires @Bean always  
d) @Autowired can only be used in XML config  
  
**Answer:** a

**1.1. What are Maven & Gradle?**

* **Maven** and **Gradle** are **build automation tools**.
* Used for:
  + Compiling source code
  + Packaging applications
  + Managing dependencies (external libraries)
  + Running tests
  + Deploying artifacts

**Example Analogy:**  
Think of Maven/Gradle as a "project manager" for your code — it keeps track of all parts, dependencies, and how they fit together.

**2. Why Use a Build Tool?**

* Manual compilation and dependency download is error-prone.
* Automates repetitive tasks (compile, test, package).
* Ensures consistent builds across environments.
* Handles versioning of libraries automatically.

**Example without build tool:**

javac -cp "lib/\*" src/com/example/\*.java

jar cf myapp.jar com/example/\*.class

**With Maven/Gradle:**

mvn package

# or

gradle build

**3. Apache Maven**

**3.1. Key Features**

* XML-based configuration (pom.xml)
* Convention over configuration (follows a standard project structure)
* Centralized dependency management via Maven Central

**Maven Standard Directory Layout:**

myproject/

├── src/

│ ├── main/java

│ └── test/java

└── pom.xml

**3.2. Example pom.xml**

<project>

<modelVersion>4.0.0</modelVersion>

<groupId>com.example</groupId>

<artifactId>demo-app</artifactId>

<version>1.0-SNAPSHOT</version>

<dependencies>

<!-- Example dependency -->

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-core</artifactId>

<version>6.1.2</version>

</dependency>

</dependencies>

</project>

**3.3. Common Maven Commands**

| **Command** | **Purpose** |
| --- | --- |
| mvn compile | Compiles source code |
| mvn test | Runs unit tests |
| mvn package | Packages into JAR/WAR |
| mvn clean | Deletes build output |
| mvn install | Installs artifact into local repository |

**4. Gradle**

**4.1. Key Features**

* Groovy or Kotlin-based DSL (build.gradle or build.gradle.kts)
* Incremental builds (only rebuilds changed parts)
* Flexible and fast compared to Maven

**Gradle Project Structure** (same as Maven by convention):

myproject/

├── src/main/java

├── src/test/java

└── build.gradle

**4.2. Example build.gradle**

plugins {

id 'java'

}

group = 'com.example'

version = '1.0-SNAPSHOT'

repositories {

mavenCentral()

}

dependencies {

implementation 'org.springframework:spring-core:6.1.2'

testImplementation 'junit:junit:4.13.2'

}

**4.3. Common Gradle Commands**

| **Command** | **Purpose** |
| --- | --- |
| gradle build | Compiles, tests, and packages |
| gradle clean | Cleans build output |
| gradle test | Runs tests |
| gradle tasks | Lists available tasks |

**5. Maven vs Gradle Comparison**

| **Feature** | **Maven** | **Gradle** |
| --- | --- | --- |
| Config format | XML (pom.xml) | Groovy/Kotlin DSL |
| Speed | Slower | Faster (incremental build) |
| Flexibility | Less flexible | Highly customizable |
| Popularity | Very high | Growing rapidly |
| Learning curve | Easier to start | Slightly steeper |