### ****JAR Files:****

JAR stands for **Java Archive**.  
It is a special file format that bundles Java class files, related data, and resources into a single compressed file.  
JAR files are used to package and distribute Java programs, libraries, or components.  
Since JAR files use the ZIP format, they take up less space and transfer faster over the internet.  
A JAR file can include multiple Java class files along with other resources like images, audio files, and configuration settings.  
You can add a JAR file to a Java program’s **classpath** to use its classes and resources while running the program.

### ****How to Create a JAR File in Eclipse:****

Right-click on your project.

Select **Export**.

Choose **Java → JAR File** and click **Next**.

Select the files you want to include in the JAR.

Check the box **Export Java source files and resources**.

Choose where to save the JAR file and click **Finish.**

### ****How to Add a JAR File to Your Project:****

Right-click on your project and select **Build Path → Configure Build Path**.

Go to the **Libraries** tab.

If needed, select **Classpath**, otherwise skip this step.

Click **Add External JARs** and choose the JAR file from its saved location.

Click **Apply and Close** to finish.

### ****JDBC (Java Database Connectivity) Basics****

#### ****1. Driver****

**What is it?**  
A **Driver** is an interface in **java.sql** package that helps Java programs connect to a database.

**How to use it?**

Java does not provide the actual implementation. Instead, each database company (like MySQL, Oracle) provides its own implementation.

If you're using **MySQL**, you need to create an object of **com.mysql.cj.jdbc.Driver** (a subclass that implements Driver).

**Why is it needed?**  
Before connecting to a database, you must **register** the Driver using DriverManager.registerDriver().

#### ****2. DriverManager****

**What is it?**  
DriverManager is a **helper class** in **java.sql** that manages database drivers and helps build a connection.

**What does it do?**

It provides **static methods** to manage drivers and connect to databases.

#### ****3. Connection****

**What is it?**  
A Connection is an **interface** that represents a link between a Java application and a database.

**How to get a connection?**

You get a Connection object using the DriverManager.getConnection() method.

**Ways to connect:**

getConnection(String URL, String username, String password);

getConnection(String URL, Properties p);

getConnection(String URL);

URL contains database details.

Username and password are required for authentication.

#### ****4. Statement****

**What is it?**  
A Statement is an **interface** that allows Java to send SQL queries to the database.

**What does it do?**

It helps in executing SQL commands.

**Types of execution methods:**

executeUpdate() – Used for **DML (INSERT, UPDATE, DELETE)** and **DDL (CREATE, ALTER, DROP)** queries.

Returns an **int** (number of rows affected).

executeQuery() – Used for **DQL (SELECT)** queries.

Returns a **ResultSet** (which holds query results).

execute() – Can run **any type of SQL query** (DDL, DML, DQL).

Returns **true** if it's a **SELECT (DQL)** query.

Returns **false** if it's a **DML or DDL** query.

### ****ResultSet in JDBC****

**ResultSet** is an interface from the java.sql package.

It holds the results after running an SQL query.

It lets us go through and read the data returned from the database.

The data is shown like a table — with rows and columns.

You usually get a ResultSet by running a query using a Statement or PreparedStatement.

#### ****Common Methods in ResultSet:****

**next()** – Moves to the next row. Returns true if there is another row, otherwise false. Commonly used in loops.

**previous()** – Moves to the previous row. Returns true if successful.

**first()** – Moves to the first row. Returns true if the ResultSet has data.

**last()** – Moves to the last row. Returns true if the ResultSet has data.

**absolute(int row)** – Moves to a specific row number. Returns true if that row exists.

**relative(int rows)** – Moves forward or backward by the given number of rows from the current position. Returns true if successful.

### ****Properties File in JDBC****

A **properties file** is a text file with key-value pairs like:

url=jdbc:mysql://localhost:3306/mydb

username=root

password=1234

It stores database connection info like URL, username, and password.

Java’s Properties class (from java.util package) is used to read it.

#### ****Why Use It?****

**Cleaner Code** – Keeps configuration separate from the main code.

**Easy Maintenance** – Change connection info without changing the Java code.

**More Secure** – You can protect the file and restrict access.

#### ****Steps to Read a Properties File****

Create a .properties file in your project.

Use FileReader to read it.

Create a Properties object.

Load the file using the load() method.

Get values using getProperty("key").

Use those values to connect to the database:

Connection conn = DriverManager.getConnection(dbURL, properties);

### ****XML (Extensible Markup Language)****

XML is a text file used to store or transfer data in a structured way.

It’s similar to HTML but meant for storing data, not displaying it.

#### ****Types of XML:****

**Self-defined XML**

No fixed rules. You define your own tags.

Used for simple data sharing between apps.

Read using file handling or libraries like JAXP.

**XML with XSD (XML Schema Definition)**

Follows strict rules defined by the XSD.

Used in applications where a fixed structure is needed (like configuration files).

### ****Apache Maven****

Maven is a tool that helps manage Java projects.

It automates building, testing, and managing dependencies.

It uses a file called pom.xml to declare project info, dependencies, plugins, etc.

### ****Maven Archetype****

An **archetype** is like a project template in Maven.

It creates a ready-to-use project with folders and sample files.

#### ****Steps to Create a Maven Project:****

In your IDE, go to:

File → New → Maven Project

OR: File → Others → Search "Maven" → Select Maven Project → Finish

Click **Next**.

Choose **Internal Catalog** and select maven-archetype-quickstart.

Set the **Group ID**, **Artifact ID**, and optionally a **base package name**.

Click **Finish**.

### ****About**** pom.xml

pom.xml stands for **Project Object Model**.

It’s where you define:

Project name, version

Dependencies (libraries you need)

Plugins and build settings

Maven reads this file and automatically downloads what's needed for your project.

JDBC Interview Questions:

# JDBC Interview Questions & Answers

Core Java & File Handling

1. What is the drawback of core Java?

- Core Java lacks built-in support for database connectivity, requiring external APIs like JDBC

- No direct support for distributed computing (needs RMI or other frameworks)

- Limited support for complex data structures compared to specialized libraries

2. What is the drawback of File Handling?

- File operations are slow compared to database operations

- No built-in querying capabilities (like SQL)

- Limited concurrent access support

- No transaction management

- No data relationships or constraints enforcement

JDBC Basics

3. What is JDBC?

- Java Database Connectivity (JDBC) is an API that enables Java programs to execute SQL statements

- It provides methods to connect to databases, send queries, and process results

- Acts as a bridge between Java applications and databases

4. What are the steps for JDBC?

- Load and register the driver

- Establish connection using DriverManager

- Create Statement/PreparedStatement/CallableStatement

- Execute queries

- Process ResultSet (for SELECT queries)

- Close connections (ResultSet, Statement, Connection)

5. What are the components used from java.sql package?

- Driver (interface)

- DriverManager (class)

- Connection (interface)

- Statement, PreparedStatement, CallableStatement (interfaces)

- ResultSet (interface)

- SQLException (class)

Driver Concepts

6. What is a Driver?

- A Driver is an interface in java.sql package that defines methods to connect to a database

- Database vendors provide implementations (e.g., com.mysql.cj.jdbc.Driver for MySQL)

7. What is the difference between Driver(I) and Driver(C)?

- Driver(I) is the interface in java.sql package

- Driver(C) is the concrete implementation provided by database vendors

- Example: MySQL's com.mysql.cj.jdbc.Driver implements java.sql.Driver

8. How many ways can we load & register a Driver? Explain.

- Using Class.forName(): `Class.forName("com.mysql.cj.jdbc.Driver");` (automatically registers via static block)

- Using DriverManager.registerDriver(): `DriverManager.registerDriver(new com.mysql.cj.jdbc.Driver());`

- Using Service Provider Mechanism (JDBC 4.0+): Just include the JAR, DriverManager auto-detects

Connection Management

9. What is DriverManager?

- A helper class that manages database drivers

- Provides static methods to establish connections

- Maintains a list of registered drivers and selects appropriate one for connection URL

10. What is a Connection? How to establish a Connection in JDBC?

- Connection represents a session with a database

- Established using: `Connection conn = DriverManager.getConnection(url, username, password);`

- Or with Properties: `Connection conn = DriverManager.getConnection(url, properties);`

11. How many types of getConnection() methods are present in JDBC?

- `getConnection(String url)`

- `getConnection(String url, Properties props)`

- `getConnection(String url, String user, String password)`

12. What is the returntype of getConnection() method?

- Returns a Connection interface object

13. What is a Database url? Explain.

- A string that identifies a database connection

- Format: `jdbc:<subprotocol>:<subname>`

- Example MySQL: `jdbc:mysql://localhost:3306/mydb`

- Example Oracle: `jdbc:oracle:thin:@localhost:1521:orcl`

Statements

14. What is a Statement?

- An interface used to execute static SQL queries

- Created via: `Statement stmt = connection.createStatement();`

- Used for queries without parameters

15. What are the types of Statement in JDBC?

- Statement: For simple queries

- PreparedStatement: For parameterized queries (precompiled)

- CallableStatement: For stored procedures

16. What is the returntype of createStatement() method?

- Returns a Statement interface object

17. What is the returntype of prepareStatement() method?

- Returns a PreparedStatement interface object

18. What are the methods used to execute a Statement? What are the returntypes?

- `executeQuery(String sql)`: Returns ResultSet (for SELECT)

- `executeUpdate(String sql)`: Returns int (row count for DML)

- `execute(String sql)`: Returns boolean (true if ResultSet available)

- `executeBatch()`: Returns int[] (row counts for batch operations)

Resource Management & Properties

19. Why is it necessary to close the Connection?

- To release database resources

- Prevent memory leaks

- Avoid exceeding maximum connection limits

- Best practice: Close in finally block or use try-with-resources

20. What are the steps to read the data from .properties file?

- Create Properties object: `Properties props = new Properties();`

- Load file using FileReader: `props.load(new FileReader("config.properties"));`

- Get values: `String url = props.getProperty("db.url");`

- Use values to establish connection

Framework, Hibernate & ORM - Simplified Technical Explanation

Framework

A framework is a reusable, structured platform that provides a foundation for developing software applications. It includes predefined classes, functions, and tools to streamline development by handling common tasks, reducing repetitive coding, and enforcing best practices.

- Purpose: Speeds up development by offering built-in functionalities (e.g., database interaction, security, UI components).

- Examples: Spring (Java), Django (Python), React (JavaScript).

---

ORM (Object-Relational Mapping)

ORM is a programming technique that maps Java objects to database tables, allowing developers to interact with databases using object-oriented concepts instead of writing raw SQL.

- How it works:

- A Java class = Database table

- Class attributes (fields) = Table columns

- Object instances = Table rows

- Advantages:

- Reduces manual SQL coding.

- Improves maintainability by working with objects instead of queries.

- Popular Java ORM Frameworks:

- Hibernate

- EclipseLink

- MyBatis (formerly iBatis)

---

Hibernate

Hibernate is a Java-based ORM framework that simplifies database interactions by replacing JDBC (Java Database Connectivity) with a higher-level, object-oriented approach.

- Key Features:

- Automatic table mapping (via annotations or XML).

- Simplifies CRUD operations (Create, Read, Update, Delete).

- Supports HQL (Hibernate Query Language), an object-oriented alternative to SQL.

- Caching for performance optimization.

- Database independence (works with MySQL, PostgreSQL, Oracle, etc.).

- How Hibernate Works:

1. Define Java classes (entities) mapped to database tables.

2. Configure the mapping (using @Entity, @Table, @Column annotations).

3. Use Hibernate Session to perform database operations.

- Example:

@Entity

@Table(name = "employee")

public class Employee {

@Id

@Column(name = "emp\_id")

private int id;

@Column(name = "emp\_name")

private String name;

}

Instead of writing SQL:

INSERT INTO employee (emp\_id, emp\_name) VALUES (1, "John");

Hibernate allows:

Employee emp = new Employee(1, "John");

session.save(emp);

### JPA:

- JPA represents Java Persistence API.

- JPA only provides specifications; it doesn't provide any implementation.

- The specifications of JPA are a set of rules and guidelines for implementing Object Relational Mapping (ORM).

- Hibernate is known for implementing JPA guidelines.

- All the specifications of Hibernate JPA are present in the javax.persistence package.

- Hibernate JPA internally uses JPQL (Java Persistence Query Language) to interact with the database.

- JPA uses the EntityManagerFactory interface to initiate the database connection process.

- All configurations related to database connectivity are stored in the persistence.xml file.

- The EntityManagerFactory interface helps create instances of the EntityManager interface, which in turn creates an EntityTransaction instance to perform database operations.

### Persistence:

- It is a class present in the javax.persistence package.

- It is a helper class used to create an object for EntityManagerFactory by invoking the static method createEntityManagerFactory().

### EntityManagerFactory (Interface):

- It is an interface present in the javax.persistence package.

- It reads configuration details from the persistence.xml file, builds a connection with the respective database based on the persistence-unit name provided, and creates tables for classes annotated with @Entity.

- An object for EntityManagerFactory can be created by invoking the createEntityManagerFactory() method using the Persistence class.

- It is responsible for creating EntityManager instances.

### EntityManager (Interface):

- It is an interface present in the javax.persistence package.

- An object for EntityManager can be created by invoking the createEntityManager() method using the EntityManagerFactory object.

- It is similar to the Session in core Hibernate.

- It has several built-in methods to perform database operations and is used to build different types of queries and Criteria.

- Methods to perform CRUD operations:

1. persist() - to insert data.

2. find() - to fetch data.

3. merge() - to update data.

4. remove() - to delete data.

### EntityTransaction (Interface):

- It is an interface present in the javax.persistence package.

- It acts as a gateway for managing transactions when interacting with a relational database.

- An object for EntityTransaction can be created by invoking the getTransaction() method using the EntityManager object.

### Relational Mapping:

- Relational mapping in Hibernate involves defining relationships between Java objects and corresponding tables in a relational database.

- There are four types of mappings in Hibernate:

1. @OneToOne

2. @OneToMany

3. @ManyToOne

4. @ManyToMany

#### 1. @OneToOne:

- Establishes a relationship where one instance of an entity is associated with one and only one instance of another entity.

- \*Uni-directional mapping\*: The entity class where the mapping information is provided is called the owner entity, and the other is the non-owning entity. An extra column is added to the owner entity table to represent the foreign key relationship.

- \*Bi-directional mapping\*: Both entities have mapping information, making both owner entities. An extra column is added to both tables. The mappedBy attribute can be used to reduce code and avoid extra columns.

- Example: Person and Passport.

#### 2. @OneToMany:

- Defines a relationship where one instance of an entity is related to multiple instances of another entity.

- \*Uni-directional mapping\*: An extra table is created to represent the foreign key relationship.

- \*Bi-directional mapping\*: An extra table is created for the @OneToMany side, and an extra column is added to the @ManyToOne side. The mappedBy attribute can be used on the @OneToMany side to reduce code and avoid extra tables.

- Example: Company and Employee, Person and Vehicle.

#### 3. @ManyToOne:

- Defines a relationship where multiple instances of an entity are associated with one instance of another entity (inverse of @OneToMany).

- \*Uni-directional mapping\*: An extra column is added to the owner entity table.

- \*Bi-directional mapping\*: Behaves the same as @OneToMany bi-directional. The mappedBy attribute cannot be used with @ManyToOne.

- Example: Employee and Company, Teachers and Department.

#### 4. @ManyToMany:

- Defines a relationship where multiple instances of one entity are associated with multiple instances of another entity.

- \*Uni-directional mapping\*: An extra table is created to represent the foreign key relationship.

- \*Bi-directional mapping\*: Two extra tables are created. The mappedBy attribute can be used in either entity to reduce code and avoid extra tables.

- Example: Student and Course, Customer and Product.

### mappedBy Attribute:

- Used to define the inverse side of a bidirectional relationship.

- Specifies the field that owns the relationship in the owning entity.

- Ensures Hibernate understands which side owns the foreign key, avoiding duplication and maintaining consistency in the database and entity relationships.

### 🔑 ****Primary Key Auto Generation Strategy****

#### ➤ GenerationType.IDENTITY

This tells the database to handle primary key generation automatically.

The database uses **auto-increment**, so every new row gets a unique number.

You don't need to set the ID manually.

**Example:**

@GeneratedValue(strategy = GenerationType.IDENTITY)

private int id;

#### ➤ GenerationType.AUTO

This lets **Hibernate** decide how to generate primary keys.

Hibernate may use a **single sequence** for the whole database, which can lead to shared numbering across tables.

**Disadvantage:**

All tables may share the same number sequence, which can be confusing.

**Example:**

@GeneratedValue(strategy = GenerationType.AUTO)

private int id;

### ****Cascading in Hibernate JPA****

Cascading means applying the **same operation** (like save, delete) on one object and its associated object(s) automatically.

**Example:**

@OneToOne(cascade = {CascadeType.PERSIST, CascadeType.REMOVE})

private Cart cart;

If you **save or delete** a User, the same will happen to the Cart.

Another example:

class Theater {

@OneToMany(cascade = {CascadeType.PERSIST, CascadeType.REMOVE})

private List<Seat> seat;

}

If you **save or delete** a Theater, all related Seats will also be saved or deleted automatically.

### ⚡ ****Fetch Types in Hibernate JPA****

Fetch type controls **when** associated data is loaded from the database.

There are **two types**:

#### 1️⃣ FetchType.LAZY (Lazy Loading)

Data from associated tables is **not loaded immediately**.

It's only loaded **when needed**.

**Example:** A Company has many Employees.  
When you load the company data, employee data is **not loaded unless you access it**.  
This saves memory and improves performance.

#### 2️⃣ FetchType.EAGER (Eager Loading)

Data from associated tables is **loaded immediately** with the main entity.

**Example:** A Cart has many Products.  
When you load the cart, all its products are also loaded at the same time — because you usually need them together.

Here’s the consolidated and structured text from the provided images:

---

### \*\*Spring Framework Overview\*\*

- \*\*Spring\*\* is a framework of frameworks, providing configuration and comprehensive programming support for Java applications.

- Key goals: Handle infrastructure/configuration so developers focus on business logic.

- Promotes modular code and integrates with testing tools via \*\*Dependency Injection (DI)\*\*.

- Objects managed by Spring containers are called \*\*beans\*\*.

- Core features: \*\*IOC (Inversion of Control)\*\* and \*\*DI (Dependency Injection)\*\*.

---

### \*\*IOC (Inversion of Control)\*\*

- Transfers control of object creation and lifecycle management from the application code to the Spring container.

- \*\*Spring IOC\*\* handles object creation, dependency resolution, and lifecycle management.

---

### \*\*Dependency Injection (DI)\*\*

- Injecting one object into another as a dependency without using `new` keyword.

- Both objects must be managed by the Spring container.

- \*\*Types of DI\*\*:

1. \*\*Setter Injection\*\*: Uses `<property>` tag in XML or setter methods.

2. \*\*Constructor Injection\*\*: Uses `<constructor-arg>` tag in XML or parameterized constructors.

---

### \*\*Bean Configuration\*\*

1. \*\*XML-Based Configuration\*\*:

- Define beans in `spring.xml` using `<bean>` tags.

- Specify dependencies via `<property>` or `<constructor-arg>`.

2. \*\*Annotation-Based Configuration\*\*:

- Use annotations like `@Component`, `@Autowired`, and `@Configuration`.

- `@ComponentScan` scans packages for annotated classes.

---

### \*\*Spring Containers\*\*

1. \*\*BeanFactory\*\*:

- Basic container (`org.springframework.beans.factory`).

- Lazy-loading: Creates beans only when requested via `getBean()`.

- Implemented via `XMLBeanFactory` and `FileSystemResource`.

2. \*\*ApplicationContext\*\*:

- Advanced container (`org.springframework.context`).

- Eager-loading: Creates all beans at startup.

- Subclasses: `ClassPathXmlApplicationContext` (XML) and `AnnotationConfigApplicationContext` (annotations).

---

### \*\*Bean Scopes\*\*

1. \*\*Singleton\*\* (default):

- One instance per container. All requests return the same object.

2. \*\*Prototype\*\*:

- New instance created each time `getBean()` is called.

---

### \*\*Annotations in Spring\*\*

- `@Configuration`: Marks a class as a source of bean definitions.

- `@ComponentScan`: Scans packages for `@Component`, `@Service`, `@Repository`, etc.

- `@Bean`: Declares a method as a bean producer.

- `@Component`: Marks a class as a Spring-managed bean.

- `@Autowired`: Injects dependencies automatically (fields, constructors, setters).

---

### \*\*Spring MVC Architecture\*\*

- \*\*MVC Components\*\*:

1. \*\*Model\*\*: Represents application data and business logic.

2. \*\*View\*\*: Renders UI (e.g., HTML).

3. \*\*Controller\*\*: Handles requests, processes data, and returns responses.

- \*\*Request Flow\*\*:

1. Client request hits `DispatcherServlet` (replaces `web.xml` via `AbstractAnnotationConfigDispatcherServletInitializer`).

2. `DispatcherServlet` uses `HandlerMapping` to find the appropriate controller.

3. `HandlerAdapter` bridges `DispatcherServlet` and controller.

4. Controller processes request, interacts with Model, and returns a view name.

5. `ViewResolver` maps the view name to a template (e.g., JSP).

6. Response rendered and sent back to the client.

---

This document combines all fragmented content into a structured guide covering Spring Framework fundamentals, configuration, containers, annotations, and MVC architecture.

Here’s a simplified and expanded version of your notes with additional explanations in \*\*plain English\*\*:

---

### \*\*Servlets and Web Applications\*\*

\*(For Beginners)\*

---

#### \*\*1. Types of Applications\*\*

- \*\*Standalone Application\*\*

- Runs on a single computer (e.g., MS Paint, Calculator).

- \*\*Web-Based Application\*\*

- \*\*Static Websites\*\*: Show fixed content (e.g., blogs, portfolios).

- \*Tools\*: HTML, CSS.

- \*\*Dynamic Websites\*\*: Content changes based on user actions (e.g., Facebook, YouTube).

- \*Tools\*: HTML, CSS, \*\*JSP\*\* (JavaServer Pages).

- \*\*Mobile Applications\*\*

- Apps for smartphones (e.g., WhatsApp, Instagram).

---

#### \*\*2. What is a Server?\*\*

A server is like a \*\*helper computer\*\* that stores data or runs programs for other devices (clients).

\*\*Types of Servers:\*\*

1. \*\*Web Server\*\*

- Sends static files (HTML, images) to your browser.

- \*Examples\*: Apache, Nginx.

2. \*\*Application Server\*\*

- Runs code for dynamic websites (e.g., login systems, shopping carts).

- \*Examples\*: Tomcat, IBM WebSphere.

3. \*\*Database Server\*\*

- Stores and manages data (e.g., user profiles, product details).

- \*Examples\*: MySQL, Oracle.

---

#### \*\*3. Servlets: Basics\*\*

- A \*\*servlet\*\* is a Java class that runs on a server to handle web requests.

- \*\*Package\*\*: `javax.servlet` (now `jakarta.servlet` in newer versions).

\*\*Servlet Lifecycle\*\* (Steps it follows):

1. \*\*Loading\*\*: The server loads the servlet class.

2. \*\*Initialization\*\*: Runs `init()` once (e.g., setup database connections).

3. \*\*Request Handling\*\*: Runs `service()` for every user request.

4. \*\*Destruction\*\*: Runs `destroy()` to clean up (e.g., close connections).

\*\*Key Methods in Servlet Interface\*\*

| Method | What It Does |

|-----------------------|-----------------------------------------------------------------------------|

| `init()` | One-time setup (e.g., load settings). |

| `service()` | Handles all requests (calls `doGet()` or `doPost()` for HTTP). |

| `destroy()` | Cleans up resources before the servlet is removed. |

---

#### \*\*4. GenericServlet vs. HttpServlet\*\*

| Feature | GenericServlet | HttpServlet |

|-----------------------|-----------------------------------------|------------------------------------------|

| \*\*Protocol\*\* | Works with any protocol (e.g., FTP). | Only for HTTP (websites). |

| \*\*Methods\*\* | Uses `service()` for all requests. | Has `doGet()`, `doPost()` for HTTP verbs.|

| \*\*Use Case\*\* | Rarely used. | Used in 99% of web apps. |

---

#### \*\*5. Handling Requests\*\*

- \*\*Static Requests\*\*

- The server sends pre-made files (e.g., `index.html`).

- \*\*Dynamic Requests\*\*

- The server runs code (servlets) to generate custom responses.

- \*How?\*: Uses `web.xml` (\*\*Deployment Descriptor\*\*) to map URLs to servlets.

- Example:

```xml

<!-- In web.xml -->

<servlet>

<servlet-name>LoginServlet</servlet-name>

<servlet-class>com.example.LoginServlet</servlet-class>

</servlet>

<servlet-mapping>

<servlet-name>LoginServlet</servlet-name>

<url-pattern>/login</url-pattern>

</servlet-mapping>

```

---

#### \*\*6. Redirects and Forwards\*\*

- \*\*`sendRedirect()`\*\*

- Sends the user to a new URL (e.g., after login).

- Example:

```java

response.sendRedirect("success.jsp"); // Browser shows new URL

```

- \*\*RequestDispatcher\*\*

- Forwards the request to another servlet/JSP \*without\* changing the URL.

- Example:

```java

RequestDispatcher rd = request.getRequestDispatcher("header.jsp");

rd.include(request, response); // Include header in the response

```

---

#### \*\*7. JSP (JavaServer Pages)\*\*

- Mixes HTML with Java code for dynamic web pages.

\*\*Common JSP Tags:\*\*

1. \*\*Directive Tag (`<%@ %>`)\*\*

- Imports files or sets rules.

- Example:

```jsp

<%@ page import="java.util.Date" %> <!-- Import Java Date class -->

```

2. \*\*Expression Tag (`<%= %>`)\*\*

- Displays values (e.g., variables).

- Example:

```jsp

<p>Time: <%= new Date() %></p> <!-- Shows current time -->

```

3. \*\*Scriptlet Tag (`<% %>`)\*\*

- Embeds Java logic (e.g., loops).

- Example:

```jsp

<% for(int i=0; i<5; i++) { %>

<p>Hello <%= i %></p> <!-- Prints "Hello 0" to "Hello 4" -->

<% } %>

```

---

#### \*\*8. Annotations (Simpler Than XML!)\*\*

- \*\*`@WebServlet`\*\*

- Replaces `web.xml` configuration. Just add it above your servlet class!

- Example:

```java

@WebServlet("/login") // Maps to http://yoursite.com/login

public class LoginServlet extends HttpServlet { ... }

```

---

#### \*\*9. Extra Tips\*\*

- \*\*Why Servlets?\*\*

- They handle user inputs (e.g., forms), connect to databases, and generate dynamic content.

- \*\*Session Management\*\*

- Use `HttpSession` to track users (e.g., shopping carts).

- \*\*Always Close Resources\*\*

- Close database connections in `destroy()` to avoid leaks.

# ****Spring Framework****

## ****Introduction****

Spring is a framework of frameworks.

"It provides a model of configuration and comprehensive programming for the development of any Java application."

This means Spring offers a structured way to set up and manage Java applications. Instead of writing everything from scratch, Spring provides ready-made tools and rules (like templates) to make development faster and easier.

"The major goal of Spring is to handle infrastructure or configuration so that the developer can focus on business logic."

Spring takes care of the "plumbing" work (like connecting components, managing databases, and handling security) so developers can concentrate on writing the actual features of the application (like payment processing, user login, or data analysis).

Spring framework encourages a modular approach, making it easier to develop and maintain code.

"Dependency Injection promotes writing testable code, and the framework integrates well with testing tools."

Dependency Injection (DI) means Spring automatically provides ("injects") the objects (dependencies) a class needs, instead of the class creating them itself.

**Why this helps testing:**

Since dependencies are injected (e.g., from outside), you can easily replace them with mock objects during testing.

Example: If your code needs a database, Spring lets you swap the real database with a fake one for testing, making tests faster and more reliable.

**Testing tools:** Spring works smoothly with tools like JUnit and Mockito, so writing and running tests is hassle-free.

"In Spring, the objects which are handled by the Spring container are known as beans."

**Spring Container:** Think of it as a "manager" that creates, configures, and manages objects in your application.

**Beans:** These are just Java objects that Spring manages. They’re called "beans" (instead of "objects") to highlight that Spring controls their lifecycle.

Example: If you have a UserService class, Spring turns it into a "bean," handles its dependencies, and makes it available where needed.

## ****Core Features****

IOC - Inversion of Control

DI - Dependency Injection

## ****What is IoC?****

**Inversion of Control (IoC)** is a design principle where the control of object creation and lifecycle management is transferred from the application code to a framework (Spring Container). Instead of developers manually instantiating and managing dependencies, the framework handles it automatically.

### ****Key Concepts of IoC****

#### ****Traditional Approach (No IoC)****

Developers explicitly create objects using new and manage dependencies.

Tight coupling makes code hard to test and maintain.

#### ****IoC Approach (Spring Container)****

Objects (beans) are created, configured, and managed by the Spring Container.

Dependencies are injected (DI) rather than hardcoded.

### ****Technical Example: Payment Service Without vs. With IoC****

#### ****1. Without IoC (Manual Control)****

class PaymentProcessor {

private PaymentGateway gateway;

public PaymentProcessor() {

this.gateway = new PayPalGateway(); // Hardcoded dependency

}

public void processPayment(double amount) {

gateway.charge(amount);

}

}

**Problems:**

Tight Coupling: PaymentProcessor is stuck with PayPalGateway. Switching to StripeGateway requires code changes.

Hard to Test: Cannot easily mock PaymentGateway for unit tests.

#### ****2. With IoC (Spring Container in Control)****

**Step 1: Define Interfaces (Loose Coupling)**

interface PaymentGateway {

void charge(double amount);

}

@Component

class PayPalGateway implements PaymentGateway {

public void charge(double amount) {

System.out.println("Charging via PayPal: " + amount);

}

}

**Step 2: Let Spring Inject the Dependency**

@Component

class PaymentProcessor {

private final PaymentGateway gateway;

public PaymentProcessor(PaymentGateway gateway) {

this.gateway = gateway;

}

public void processPayment(double amount) {

gateway.charge(amount);

}

}

**Step 3: Configuration (Annotation-Based)**

@Configuration

@ComponentScan("com.example")

public class AppConfig { }

### ****How Spring IoC Works Here:****

Spring Container scans for @Component classes (PayPalGateway, PaymentProcessor).

It detects that PaymentProcessor needs a PaymentGateway.

It automatically injects PayPalGateway into PaymentProcessor.

### ****Why IoC is Powerful****

**Decoupling**: PaymentProcessor doesn’t know which PaymentGateway is used.

**Testability**: Inject a MockPaymentGateway for tests.

**Lifecycle Management**: Handles scopes, lazy loading, destruction.

## ****What is Dependency Injection?****

**Dependency Injection (DI)** is a design pattern (and a core feature of Spring) where dependencies are "injected" into a class by an external system (Spring Container) instead of the class creating them itself. This promotes loose coupling and testability.

### ****Key Concepts of DI****

**Dependency**: An object that another object needs.

**Injection**: Providing the dependency externally.

**Spring Container**: Manages beans and their dependencies.

### ****Technical Example: Notification System****

#### ****1. Without DI (Tight Coupling)****

class EmailService {

public void sendEmail(String message) {

System.out.println("Email sent: " + message);

}

}

class NotificationService {

private EmailService emailService = new EmailService();

public void notify(String message) {

emailService.sendEmail(message);

}

}

**Problems:**

Hard to Test

Inflexible

#### ****2. With DI (Loose Coupling)****

**Step 1: Define Interfaces (Abstraction)**

interface MessageService {

void send(String message);

}

@Component

class EmailService implements MessageService {

public void send(String message) {

System.out.println("Email sent: " + message);

}

}

@Component

class SmsService implements MessageService {

public void send(String message) {

System.out.println("SMS sent: " + message);

}

}

**Step 2: Inject the Dependency**

@Component

class NotificationService {

private final MessageService messageService;

@Autowired

public NotificationService(MessageService messageService) {

this.messageService = messageService;

}

public void notify(String message) {

messageService.send(message);

}

}

**Step 3: Configuration (Annotation-Based)**

@Configuration

@ComponentScan

public class AppConfig { }

### ****How DI Works in Spring****

Detects @Component classes.

Resolves dependencies.

Injects implementations into dependent classes.

### ****Types of Dependency Injection****

#### ****Constructor Injection (Recommended)****

@Autowired

public NotificationService(MessageService messageService) {

this.messageService = messageService;

}

#### ****Setter Injection****

private MessageService messageService;

@Autowired

public void setMessageService(MessageService messageService) {

this.messageService = messageService;

}

#### ****Field Injection (Avoid)****

@Autowired

private MessageService messageService;

### ****Why DI is Powerful****

**Loose Coupling**

**Testability**

MessageService mockService = mock(MessageService.class);

NotificationService service = new NotificationService(mockService);

service.notify("Test");

verify(mockService).send("Test");

**Reusability**

## ****Spring Container: Bean Configuration Methods****

Two ways:

XML-Based Configuration

Annotation-Based Configuration

### ****1. XML-Based Configuration****

Uses XML file (e.g., spring.xml) to define beans and dependencies.

#### ****Key Components:****

Root Tag: <beans>

Bean Tag: <bean> with:

id

class

Property Injection: <property name="fieldName" value="defaultValue"/>

Constructor Injection: <constructor-arg name="paramName" value="defaultValue"/>

#### ****Example: spring.xml****

<beans>

<bean id="userService" class="com.example.UserService">

<property name="username" value="admin" />

</bean>

<bean id="paymentGateway" class="com.example.PayPalGateway">

<constructor-arg name="apiKey" value="12345" />

</bean>

</beans>

#### ****Key Points****

**Pros:**

Centralized configuration

Explicit declaration

**Cons:**

Verbose

No compile-time checks

### ****Comparison with Annotation-Based Configuration****

| **Feature** | **XML-Based** | **Annotation-Based (@Component, @Autowired)** |
| --- | --- | --- |
| Configuration | External (XML file) | Embedded in code (annotations) |
| Flexibility | High (runtime changes) | Lower (requires recompilation) |
| Verbosity | High | Low |
| Use Case | Legacy systems, large teams | Modern apps, rapid development |

**When to Use XML:**

Legacy migration

Runtime config change

Separation preference

## ****Spring Annotations****

### ****1. @Configuration - The Spring Blueprint****

**What it Does:**

Marks a class as configuration

Replacement for XML

**Example:**

@Configuration

public class AppConfig { }

**Key Points:**

Singleton behavior

Bean dependency management

### ****2. @ComponentScan - The Bean Discoverer****

**What it Does:**

Scans specified packages for components

**Example:**

@Configuration

@ComponentScan("com.example")

public class AppConfig { }

**Customization:**

@ComponentScan(

basePackages = "com.example",

excludeFilters = @ComponentScan.Filter(type = FilterType.REGEX, pattern = ".\*Test.\*")

)

### ****3. @Bean - The Explicit Bean Declarer****

**What it Does:**

Declares method as bean producer

**Example:**

@Configuration

public class DatabaseConfig {

@Bean

public DataSource dataSource() {

HikariDataSource ds = new HikariDataSource();

ds.setJdbcUrl("jdbc:mysql://localhost:3306/mydb");

return ds;

}

@Bean

public JdbcTemplate jdbcTemplate(DataSource dataSource) {

return new JdbcTemplate(dataSource);

}

}

**Key Points:**

Depends on other beans

Singleton by default

Use @Scope("prototype") to override

### ****How These Annotations Work Together****

**Startup:**

Finds @Configuration classes

**Component Scanning:**

Uses @ComponentScan

**Bean Registration:**

From @Component or @Bean

**Dependency Injection:**

Wires beans together

**Example Workflow:**

@Configuration

@ComponentScan("com.example")

public class AppConfig {

@Bean

public RestTemplate restTemplate() {

return new RestTemplate();

}

}

@Service

public class UserService {

@Autowired

private RestTemplate restTemplate;

}

### ****Key Differences: @Component vs. @Bean****

| **Feature** | **@Component** | **@Bean** |
| --- | --- | --- |
| Usage | On classes | On methods |
| Control | Limited | Full |
| Use Case | Your own classes | Third-party/library classes |

### ****Best Practices****

Use @Configuration + @ComponentScan for auto-wiring.

Use @Bean for external libraries.

Prefer constructor injection.

**Example:**

@Bean

public JdbcTemplate jdbcTemplate(DataSource dataSource) { ... }

### ****Final Tip****

@Configuration

public class AppConfig {

@Bean

public ObjectMapper objectMapper() {

return new ObjectMapper().registerModule(new JavaTimeModule());

}

@Bean

public RestTemplate restTemplate(ObjectMapper mapper) {

RestTemplate rt = new RestTemplate();

rt.getMessageConverters().add(new MappingJackson2HttpMessageConverter(mapper));

return rt;

}

}

### ****4. @Component (Stereotype Annotation)****

**Purpose:**

Marks a class as Spring bean

Base for @Service, @Repository, @Controller

**Example:**

@Component

public class PaymentGateway { }

### ****5. @Autowired****

**Purpose:**

Auto-inject dependencies by type

**Can be used on:**

Fields

Constructors (Recommended)

Setters

**Example:**

@Service

public class OrderService {

private final PaymentGateway gateway;

@Autowired

public OrderService(PaymentGateway gateway) {

this.gateway = gateway;

}

}