Spring

**Q: What is Spring IoC Container and Dependency Injection?**

The **Spring IoC Container** is the **core component** of the Spring Framework that is responsible for **managing the life cycle and configuration of application objects**. "IoC" stands for **Inversion of Control**, which means the control of object creation and dependency management is transferred from the application code to the container.

In simpler terms, instead of manually creating objects using new, we allow the container to create and inject the required dependencies into our classes.

The **IoC Container** does this through a process known as **Dependency Injection (DI)**.

**What is Dependency Injection?**

**Dependency Injection** is a design pattern used to **remove the dependency between objects**. Instead of a class instantiating its own dependencies, the dependencies are **injected externally**—by the Spring container—usually through:

1. **Constructor Injection**
2. **Setter Injection**
3. **Field Injection** (less recommended, but possible)

For example, if class A depends on class B, we don’t create B inside A. Instead, we let the container inject an instance of B into A.

**Why is it useful?**

* It promotes **loose coupling**
* It improves **testability** (e.g., with mocks)
* It enhances **code maintainability and readability**
* It aligns with **SOLID principles**, especially the **Dependency Inversion Principle**

**Types of IoC Containers in Spring:**

1. **BeanFactory** – Lightweight and lazy-loading container.
2. **ApplicationContext** – More advanced, feature-rich container (used in most applications).

**Example:**Suppose we have a StudentService class that depends on a StudentRepository. Instead of :

@Autowired

private StudentRepository repo; Then, Spring injects the appropriate implementation automatically at runtime.

**Spring JDBC**

central class in the Spring JDBC core package

Key Features:

* **Simplifies JDBC:**

It handles core JDBC workflow, such as opening and closing connections, executing SQL queries/updates, and iterating over ResultSets.

* **Exception Handling:**

It catches JDBC exceptions and translates them into Spring's generic DataAccessException hierarchy, providing informative error messages.

* **Callback Interfaces:**

It uses callback interfaces like PreparedStatementCreator, ResultSetExtractor, RowMapper, and PreparedStatementSetter to customize SQL execution and result processing.

* **Thread-Safe:**

Once configured, JdbcTemplate instances are thread-safe, allowing for shared use in multiple DAOs or repositories.

* **Configuration:**

It requires a DataSource to be configured, either directly or via dependency injection in a Spring application context.

* **Logging:**

All SQL operations performed by this class are logged at debug level, using org.springframework.jdbc.core.JdbcTemplate as log category.

Common Use Cases:

* Executing SQL queries and updates (INSERT, UPDATE, DELETE).
* Retrieving data from the database using various query methods.
* Mapping query results to Java objects using RowMapper.
* Handling database exceptions.
* Setting query timeouts.

Alternatives:

* NamedParameterJdbcTemplate: Wraps JdbcTemplate to support named parameters instead of traditional ? placeholders.
* SimpleJdbcInsert and SimpleJdbcCall: Optimizes database metadata to simplify coding for INSERT and stored procedure calls.
* RDBMS Objects (e.g., MappingSqlQuery, SqlUpdate, StoredProcedure): Creates reusable and thread-safe objects for data access.
* JdbcClient (introduced in Spring 6.1): Provides a fluent API style for common JDBC queries/updates with flexible use of indexed or named parameters.

JdbcTemplate is a powerful tool for simplifying JDBC operations in Spring applications. It is recommended to use it as the central class for accessing databases through JDBC.