Dependency Injection in Spring

In this section, we explored Dependency Injection (DI), a core concept in Spring that helps manage object dependencies efficiently. We learned how Spring creates and injects objects, eliminating the need for manual object creation and improving code maintainability and testability.

1. What is Dependency Injection?

- **Dependency Injection (DI)** is a design pattern that allows objects to receive their dependencies from an external source rather than creating them internally.
- DI makes applications more modular, testable, and maintainable by decoupling object creation from business logic.

2. Ways to Inject Dependencies

There are multiple ways to inject dependencies in Spring:

- **Constructor Injection (Recommended):** Dependencies are passed through the class constructor and assigned to fields.
- **Setter Injection:** Dependencies are injected via setter methods after the object is created. Useful when dependencies are optional or need to be changed at runtime.

3. The Spring IoC Container

- **The Inversion of Control (IoC) container** is responsible for managing the lifecycle of beans (Spring-managed objects).
- When an application starts, the IoC container creates, configures, and injects dependencies into beans automatically.
- Beans are stored in the **ApplicationContext**, allowing them to be retrieved and reused throughout the application.

4. Configuring Beans in Spring

There are two ways to configure beans in Spring:

- **Using Annotations:** We can define beans using `@Component`, `@Service`, `@Repository`, and `@Controller`.
- **Programmatically:** We can manually configure beans using Java-based configuration with `@Bean` inside a `@Configuration` class. This provides more fine-grained control over bean creation.

• When multiple beans of the same type exist, we can specify which one to use with `@Primary` or `@Qualifier`.

5. Lazy Initialization

- By default, Spring eagerly initializes beans when the application starts.
- We can enable lazy initialization using `@Lazy`, which defers bean creation until it is first needed, improving startup time in some cases.

6. Bean Scopes

- Spring beans can have different scopes, determining how they are managed:
  - **Singleton (default):** A single instance is created and shared across the application.
  - **Prototype:** A new instance is created each time it's requested.
  - **Request**, **Session**, and **Application** Scopes: Used in web applications to manage beans within HTTP request lifecycles.

7. Bean Lifecycle Methods

- Spring provides lifecycle hooks that allow us to execute code when a bean is created or destroyed.
- We can define these methods using:
  - `@PostConstruct` (executed after bean initialization).
  - `@PreDestroy` (executed before the bean is destroyed).