***What is Spring Framework?***

1. It is a Dependency Injection Framework to make java application light-weight and loosely coupled.
2. Spring framework is a java based framework which makes easy development of JavaEE application.
3. It was developed by Rod Johnson in 2003.
4. It is **non-invasive framework**, so we don’t need to extend/implement Spring related classes/interfaces for operations.
5. It is a **versatile framework** i.e We can use different java based frameworks with Spring.
6. By using Spring , we can develop
7. **Web Applications**
8. **Standalone applications** 🡪Non-sharable, desktop based apps, not hosted on server, Eg:-Notepad,Paint,etc
9. **Distributed Applications 🡪**If one application is communicating with another apps or completing certain operations suh apps are called as Distributed Applications.
10. **Microservices based Applications**.
11. Spring provides several modules, based on which we can develop applications 🡪21Modules out of which 7 are core Modules.

***Core Modules:***

1. **Spring Core Container:**
   * **Core Module**: The fundamental part of the Spring Framework, containing the foundational classes required by other modules.
   * **Beans Module**: Provides support for the Spring IoC (Inversion of Control) container and dependency injection (DI).
   * **Context Module**: Builds on the beans module, adding additional functionality like event propagation, resource loading, and internationalization.
   * **Expression Language (SpEL)**: A powerful expression language for querying and manipulating objects at runtime.
2. **Spring AOP (Aspect-Oriented Programming):**
   * Provides aspect-oriented programming functionality, allowing you to separate cross-cutting concerns like logging, transaction management, and security from business logic.
3. **Spring Data Access/Integration:**
   * **JDBC (Java Database Connectivity)**: Simplifies database interaction by reducing boilerplate code for database connections and queries.
   * **ORM (Object-Relational Mapping)**: Integrates with popular ORM frameworks like Hibernate, JPA, and JDO.
   * **JMS (Java Messaging Service)**: Provides support for working with JMS, enabling message-based communication in Spring applications.
   * **Transactions**: Provides abstractions for managing transactions across different data sources.
4. **Spring Web:**
   * **Web Module**: Provides functionality for building web applications, including support for HTTP and servlet-based applications.
   * **Spring MVC**: A model-view-controller framework for building web applications, offering flexible request mapping, form handling, and validation.
   * **WebSocket**: Support for building WebSocket-based applications for real-time communication.
   * **WebFlux**: A reactive programming model introduced in Spring 5, providing support for building asynchronous, non-blocking web applications.
5. **Spring Security:**
   * A powerful and customizable authentication and authorization framework to secure your applications.

**Additional Modules:**

1. **Spring Boot:**
   * A framework that simplifies the development of Spring-based applications. It uses conventions over configuration and provides embedded servers (like Tomcat or Jetty) to run applications with minimal setup.
2. **Spring Cloud:**
   * A set of tools for building cloud-native applications. It includes features for microservices architecture, service discovery, configuration management, and fault tolerance.
3. **Spring Batch:**
   * A framework for batch processing, designed for handling large volumes of data and tasks in a structured, efficient way.
4. **Spring Integration:**
   * A module designed for building enterprise integration solutions. It supports various messaging protocols, such as JMS, HTTP, and file-based communication.
5. **Spring HATEOAS:**
   * Provides support for creating RESTful APIs with HATEOAS (Hypermedia as the engine of application state) for better discoverability of REST services.
6. **Spring LDAP:**
   * Provides support for working with LDAP (Lightweight Directory Access Protocol) servers, making it easier to interact with LDAP-based systems.
7. **Spring Web Services:**
   * A framework for creating and consuming SOAP-based web services.
8. **Spring AMQP:**
   * Provides support for AMQP (Advanced Message Queuing Protocol), specifically for RabbitMQ, enabling message-driven architecture.
9. **Spring Cache:**
   * A caching abstraction to help you integrate caching into your Spring applications using various caching providers.
10. **Spring Test:**
    * Provides various testing utilities for unit and integration tests, allowing you to easily test Spring components, such as beans, services, and controllers.
11. **Spring Session:**
    * A module that provides session management, including support for distributed sessions and session persistence.
12. **Spring Mobile:**
    * Provides support for mobile web applications, helping you create mobile-friendly web pages and services.
13. **Spring WebFlow:**
    * A framework for building web-based user interfaces and workflows, making it easier to manage page navigation and flow in web applications.
14. **Spring JMS (Java Messaging Service):**
    * Offers a JMS template for integration with message brokers like ActiveMQ, allowing for easy interaction with JMS queues and topics.
15. **Spring Boot Actuator:**
    * Provides built-in features to monitor and manage Spring Boot applications, such as health checks, metrics, and application environment details.

**Specialized Modules:**

1. **Spring for Android**:
   * Provides a set of components and utilities for building Android applications using Spring technologies.
2. **Spring Integration Testing**:
   * Provides support for writing integration tests for Spring applications, including testing Spring beans and services.

--------------------------------------------------------------------------------------------------------------------------------------

***What are internals of Spring?***

1. **Inversion of Control (IoC) Container**

The IoC container is the backbone of the Spring Framework, Dependancy Injection is a way of impemeting Inversion Of Control .It’s responsible for creating beans and managing its lifecycle(objects), including instantiation, configuration, and wiring of dependencies.

**Types of IOC Containers:**

* ***Core Container (Deprecated)***

***🡪****It includes* ***BeanFactory******(Interface)*** *which is responsible for ,*

* 1. *Reading xml.config file*
  2. *Beans creation*
  3. *Maintaining Bean Lifecycle*
  4. *Destroying Beans*
* ***J2EE Container***

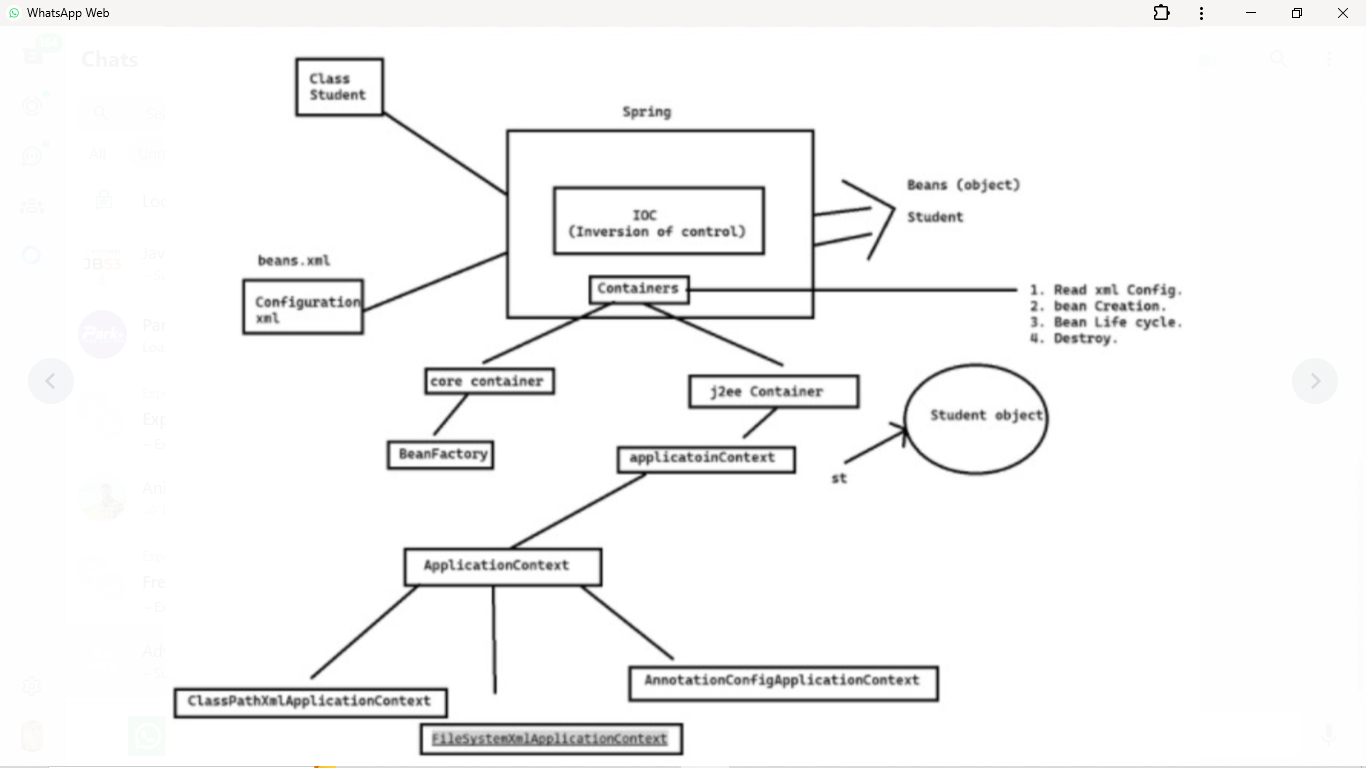
***🡪****It includes* ***ApplicationContext(Interface)***

***IQ🡪Difference between BeanFactory and ApplicationContext***

|  |  |
| --- | --- |
| ***BeanFactory (CoreContainer)*** | ***ApplicationContext (J2EE Container)*** |
| *It is core Container* | *It internally calls BeanFactory only* |
| *It doesn’t read .xml config files directly, it requires resource class* | *It directly reads config, xml config files, doesn’t need resource classes.* |
| *Can read only single xml file at a time.* | *Can read multiple xml files at a time.* |
| *Follows lazy-loading mechanism* | *Follows eager-loading mechanism.* |
| *It supports only xml config files* | *It supports java based, annotation based config files.* |
| *It is Interface of Core container* | *It is interface of J2EE Container* |
| *Lightweight container for managing beans, DI, and AOP* | *Full-featured container for managing enterprise components* |
| *Suitable for all types of Java applications (including microservices)* | *Suitable for large-scale enterprise applications* |
| *It is parent of ApplicationContext* | *It is child of BeanFactory* |

***IQ🡪What is container in Spring?***

*A container refers to the core component that* ***manages the lifecycle*** *and* ***configuration of application objects, often called beans****. The container is* ***responsible for creating objects****,* ***managing their lifecycle, injecting dependencies, and managing configuration settings****. It is at the heart of the Spring Framework and serves as the backbone of the Inversion of Control (IoC) pattern and Dependency Injection (DI).*



***IQ🡪What is ClassPathXmlApplicationContext?***

***🡪It scans and searches for the XML configuration in java classpath.***

***IQ🡪What is FileSystemXmlApplicationContext?***

***🡪It scans and searches for configuration from file systems.***

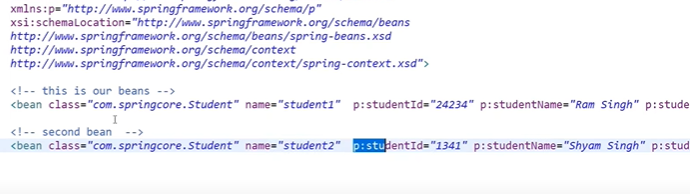
***IQ🡪What is AnnotationConfigApplicationContext?***

***🡪It scans and searches for the java based annotation which has confirguration.***

***IQ🡪What is beans.xml?***

***IQ🡪What are bean Tags?***

***IQ🡪 Why is BeanFactory lazy loader?***



***Beans Scopes-***

*Scope is use to define behaviour of beans (how many objects you want to create you can)*

*There are multiple types of scopes*

1. ***IOC-***

***Singleton****- whenever we call get bean method it will provide single object 🡪(it is default )*

***Prototype****- whenever we call it it will create diff objects 🡪(we have to use scope=”prototype” in bean).*

* *If we make prototype to only parent class bean then child class will use default singleton. so hashcodes of parents are different(creating new objects every time) but child hashcode will remain same (because of singleton)*
* *If parent class bean is singleton & child class bean is prototype then it will not checks childs scope. So hashcode of parent is same & childs hascode also remain same.*

*In that case we can use <lookup-method name="get --that child class--"/>*

1. ***WEBMVC-***

*Request- for every http request call it create new bean*

*Session- for every http session it create new bean.*

***Wiring:-***

*Injecting one bean Manually into another bean is called as wiring. Earlier we performed dependency injection manually but in real- time if we have more than one beans so we need to write injection property for every bean. Which will increase our boiler-plate code. To overcome this problem spring provides us Autowiring.*

***Autowiring:-***

*Autowiring provides us automatic injection based on multiple modes. Doing it automatically is called as Autowiring.*

* + 1. ***byType***🡪*It will check properties data type.*

*When we use byType and if we have multiple beans having same type then it will give us ambiguity. To overcome this problem we will use byName.*

* + 1. ***byName****🡪It will check properties reference name.*

*Based on that values it will search respected bean inside “ bean.xml” file.*

* + 1. ***Constructor****🡪 If we perform automatic injection using <constructor-arg> tag in that case we use constructor.*

*Constructor first checks using byType and then byName.*

* + 1. ***Autodetect****🡪It internally calls constructor.*
    2. ***No🡪No autowiring***

***Note:-***

*Autowiring supports only* ***Non-Primitive Data*** *Types.*

***IQ🡪What is bean Lifecycle?***

***(IOC) Annotation based Configuration:-***

1. ***@Configuration:-*** *It is class-level annotation*

*Which is used to represent java class as a Spring bean configuration file.*

1. ***@Bean:-****It is used to represent configuration method as a Spring bean*
2. ***@Scope:-*** *It Is used to define scope of the bean.*
3. ***@Autowired:-*** *It is used to perform automatic injection by using different modes.*

*We can use* ***@Autowired*** *annotation on field, constructor, method.*

*It is also called as field-level injection.*

1. ***@Qualifier:-*** *It is used to define qualified name to a bean.*
2. ***@Primary****:- If we have multiple beans having same type in that case if we want to represent a particular bean as a default bean, we can use* ***@Primary annotation.***

*(It is used to mark bean as default bean)*

***IQ🡪How to create custom init() and destroy method without calling in-built methods of spring?***

***Initialization & DisposableBean*** *are two interfaces who are responsible for initializing and destroying bean. They have methods afterPropertiesSet() and destroy() method respectively.*

1. ***@PostConstruct-:****It is used to define custom* ***init()*** *method*
2. ***@PreDestroy:-*** *It is used to define custom* ***destroy()*** *method.*

***IQ🡪 Difference between Spring and Servlet Bean Cycle***

***IQ🡪What was the need of Spring when we already had Java?***

*✅ 1) What is Spring Framework?*

*🟩 Definition:  
Spring is a powerful, open-source Java framework primarily used for building enterprise-level applications. It follows the Inversion of Control (IoC) and Dependency Injection (DI) principles, making applications more loosely coupled, testable, and maintainable.*

*🔍 What do we mean by lightweight and non-invasive in Spring?*

* *Lightweight:  
  Spring doesn’t force you to use heavy components like EJBs or containers like JBoss. You can run a Spring-based application with just a simple servlet container like Tomcat or even a main() method. There’s no need for full-blown servers or complex deployment.*

*➤ Real Use: You can write a REST API using Spring Boot and deploy it as a JAR—no WAR files, no application servers.*

* *Non-invasive:  
  Spring doesn’t make you extend any framework-specific class or implement interfaces. Your classes remain POJOs (Plain Old Java Objects). This keeps business logic clean and reusable even outside the Spring context.*

*➤ Example: Unlike Struts or older frameworks where you had to extend Action classes, Spring lets you write:*

*java*

*CopyEdit*

*@Service*

*public class OrderService {*

*public void processOrder() {*

*// Business logic here*

*}*

*}*

*✅ No need to implement any framework interface — it’s non-invasive.*

*🎯 Why Spring is preferred for enterprise development:*

* *Modular: You can use just what you need (e.g., Spring MVC without Spring AOP).*
* *Testable: Thanks to DI, it’s easy to inject mocks and test individual components.*
* *Scalable: Widely used in large-scale systems like banking and e-commerce.*
* *Annotation Support: Clean code using annotations like @Service, @Controller, @Autowired.*
* *Integration Ready: Works smoothly with Hibernate, JPA, MyBatis, JMS, Kafka, etc.*

*🏦 Real-Time Applications:*

* *Banking Systems – Transaction management using @Transactional.*
* *E-commerce Apps – REST APIs using Spring Boot + Spring MVC.*
* *Microservices – Built using Spring Cloud + Netflix OSS + Spring Boot.*
* *Healthcare – Secured applications using Spring Security.*

*🧠 Tricky / Indirect Interview Questions:*

* *If Spring is lightweight, how does it handle heavy enterprise apps?*
* *Can you write a Spring app without implementing any interfaces?*
* *Why is Spring called non-invasive when it still uses annotations?*

*❗ Common Exceptions:*

* *BeanCreationException: Issues during bean creation or injection.*
* *NoSuchBeanDefinitionException: Bean not found in the context.*
* *UnsatisfiedDependencyException: Spring can’t find a matching dependency to inject.*

***📌 Key Annotations to Remember:***

* *@Configuration: Defines Spring configuration class.*
* *@ComponentScan("com.x"): Automatically detects and registers beans.*
* *@EnableAutoConfiguration: Spring Boot-specific, enables auto bean configuration.*
* *@Component, @Service, @Repository, @Controller: For class-level role identification.*

*✅* ***2) Spring as an Alternative to?***

*🟩* ***Definition / Concept****:  
Spring is considered a powerful alternative to* ***EJB (Enterprise JavaBeans)*** *and the traditional* ***Servlet-JSP model****. While EJBs were part of heavyweight J2EE architecture, and Servlets/JSPs required boilerplate and tightly coupled code, Spring provides a lightweight, modular, and POJO-based framework to build enterprise-level applications with better control and simplicity.*

*🟩* ***Why Spring Replaced EJB and Servlets/JSP***

*🔹* ***EJB (Enterprise JavaBeans)****:*

* *EJB required complex deployment descriptors, heavyweight containers, and was difficult to test.*
* *Spring provided declarative transaction management using @Transactional, dependency injection, and AOP — without needing an application server.*

*🔹* ***Servlets & JSP****:*

* *Handled both business logic and request handling tightly coupled.*
* *Spring MVC separates concerns (Controller → Service → DAO), improving code organization, testability, and reusability.*

*🟩* ***Real-Time Application Use Case****:*

*💼* ***Before (EJB + JSP)****:  
An HRMS portal used stateful EJBs to handle sessions and JSPs to render UI. It required a full application server and complex JNDI lookups.*

*✅* ***After Migrating to Spring****:*

* *Stateless services created using @Service*
* *REST APIs using @RestController*
* *Simple annotations replaced EJB complexity*
* *Hosted on lightweight Tomcat instead of WebLogic or JBoss*

*🟩* ***Tricky/Indirect Interview Questions****:*

* *Why did the Java community shift from EJBs to Spring?*
* *Can you achieve declarative transactions without EJB?*
* *What’s the disadvantage of using Servlets with embedded business logic?*
* *If Spring is so powerful, why do legacy apps still use EJBs?*
* *How does Spring eliminate the need for application servers?*

*🟩* ***Common Migration Issues / Exceptions****:*

* *NoSuchBeanDefinitionException: Missing bean in context after replacing EJB*
* *UnsatisfiedDependencyException: Wrong DI configuration while refactoring*
* *Servlet mappings overlap when using Spring DispatcherServlet with old code*

*🟩* ***Annotations Used as Alternatives****:*

| ***Annotation*** | ***Purpose*** | ***Notes*** |
| --- | --- | --- |
| *@Service* | *Replaces EJB Stateless Bean* | *Mark service logic layer, Spring manages lifecycle* |
| *@Transactional* | *Replaces EJB Transaction Mgmt* | *Can be applied on class/method level, rollback behavior supported* |
| *@RestController* | *Replaces Servlet + JSP-based controllers* | *Combines @Controller + @ResponseBody* |
| *@RequestMapping* | *Maps HTTP requests to methods* | *Can use with class-level and method-level* |
| *@Autowired* | *Replaces manual object creation* | *Spring handles dependency injection internally* |

*🟩* ***Deep Dive Clarification (for Interview Follow-up)****:*

*🔹 “EJBs had container-managed services like pooling, transactions, etc. How does Spring handle that?”  
Spring uses AOP + @Transactional and manages beans using singleton or prototype scopes. Features like bean pooling, security, and logging are handled declaratively.*

*🔹 “If Servlets are still used under the hood, how is Spring better?”  
Spring’s DispatcherServlet acts as a* ***Front Controller****, managing all requests and delegating using annotations — reducing code and improving flexibility.*

*🟩* ***Conclusion****:  
Spring is not just an alternative but an evolution over the traditional Java EE stack. It offers modern architectural patterns (MVC, REST, AOP, DI) in a modular and testable way — with or without web containers — making it the go-to choice for enterprise development.*

***3) Explain the Different Versions of Spring Framework and Their Evolution***

*🟩* ***Definition / Overview****:  
Spring Framework has evolved from a basic lightweight container for managing beans (IoC) to a* ***comprehensive ecosystem*** *for building* ***full-stack enterprise and reactive applications****. Each version introduced enhancements in configuration style, annotations, performance, and support for modern Java features.*

*🟩* ***Spring Evolution Breakdown****:*

| ***Version*** | ***Major Highlights*** |
| --- | --- |
| ***Spring 1.x*** | *Introduced core IoC, basic AOP, XML-based config, BeanFactory, ApplicationContext* |
| ***Spring 2.x*** | *Enhanced AOP, simplified XML, @Transactional, @Required* |
| ***Spring 3.x*** | *Full Java-based config,* ***SpEL****, REST support, annotations like @Component, @Value* |
| ***Spring 4.x*** | *Support for Java 8 (lambdas), WebSocket, conditional annotations* |
| ***Spring 5.x*** | *Reactive Programming (WebFlux), JDK 8+ baseline, Kotlin support, @FunctionalInterface* |
| ***Spring Boot 1.x – 3.x*** | *Auto-configuration, Starter dependencies, embedded servers, microservices* |

*🟩* ***Real-Time Use Cases per Version****:*

* *🟢* ***Spring 2.x*** *used in traditional monolithic banking systems with XML config*
* *🟢* ***Spring 3.x*** *adopted by mid-size ERPs with REST APIs and annotation-based beans*
* *🟢* ***Spring 4.x*** *used by scalable systems needing WebSockets (e.g., stock price streaming apps)*
* *🟢* ***Spring 5.x*** *used in* ***reactive microservices*** *(like Netflix-style systems)*

*🟩* ***Tricky/Indirect Interview Questions****:*

* *What made Spring 3.x a major shift from its predecessors?*
* *When did annotation-based configuration become the standard in Spring?*
* *What’s the difference between Spring MVC and Spring WebFlux?*
* *How does Spring 5 handle reactive programming?*
* *Can you build a Spring Boot app using Spring 2.x?*

*🟩* ***Common Version Migration Issues****:*

| ***Error/Exception*** | ***Cause*** |
| --- | --- |
| *NoClassDefFoundError* | *Using old config or missing updated dependencies* |
| *BeanDefinitionParsingException* | *Schema mismatch in XML during upgrade* |
| *MethodNotFoundException* | *Using outdated APIs not supported in newer versions* |
| *ClassCastException or IllegalStateException* | *Framework core changes between major versions* |

*🟩* ***Important Annotations Introduced by Versions****:*

| ***Annotation*** | ***Introduced In*** | ***Purpose*** | ***Notes*** |
| --- | --- | --- | --- |
| *@Component, @Value, @Scope* | *3.x* | *For bean declaration, property injection* | *Eliminates XML in most cases* |
| *@RestController, @RequestBody* | *4.x* | *RESTful API support* | *Combines @Controller + @ResponseBody* |
| *@GetMapping, @PostMapping* | *4.3* | *Shortcut mapping annotations* | *Preferred over @RequestMapping(method = GET)* |
| *@ConditionalOnProperty, @Conditional* | *4.x & 5.x* | *Conditional bean loading* | *Common in Spring Boot apps* |
| *@FunctionalInterface* | *Java 8* | *Functional interface declaration* | *Used with lambda-style event handling* |
| *@EnableWebFlux, @ControllerAdvice* | *5.x* | *Reactive programming and global exception handling* | *For reactive apps and centralized error control* |

*🟩* ***Deep Interview Probes & Clarifications****:*

*🔸 “How is Spring 5 reactive when Java traditionally works on synchronous thread models?”  
Spring 5 uses* ***Reactor*** *(a reactive stream library) and* ***non-blocking I/O*** *under the hood with* ***Project Reactor*** *and* ***Netty****, unlike traditional servlet-based blocking threads.*

*🔸 “What changed after Spring 3 that made XML less relevant?”  
The introduction of* ***Java-based configuration*** *using @Configuration and @Bean, plus component scanning, reduced dependency on verbose XML setup.*

*🔸 “What was the role of Spring Boot and where does it fit in versioning?”  
Spring Boot started alongside Spring 4 and 5 to* ***simplify configuration****,* ***auto-wire dependencies****, and create* ***standalone microservices*** *without XML or external servlet containers.*

*🟩* ***Conclusion****:  
Spring's version evolution reflects the Java ecosystem's modernization — from tightly coupled, XML-heavy systems to lightweight, annotation-driven, and* ***reactive microservices****. Understanding version differences helps identify suitable tools and avoid compatibility issues in real-world projects.*

***4) What Are the Core Modules in Spring Framework?***

*🟩* ***Definition / Overview****:  
Spring Framework is modular. Instead of forcing you to use the whole framework, Spring allows you to use only the required modules. This modular architecture supports scalability, clean design, and performance optimization.*

*The* ***Core modules*** *act like different compartments inside the Spring train — each responsible for a specific function like dependency injection, AOP, web requests, data access, messaging, and testing.*

*🟩* ***List of Core Modules with Purpose and Use Case****:*

| ***Module Name*** | ***Purpose*** | ***Real-Time Use Case*** |
| --- | --- | --- |
| ***Core & Beans*** | *Provides IoC/DI mechanism, BeanFactory, ApplicationContext* | *Injecting dependencies like service, DAO beans into controllers* |
| ***Context*** | *Acts as a Spring-aware container; supports internationalization, events* | *Managing beans in banking apps with config from .properties or XML* |
| ***AOP (Aspect-Oriented Programming)*** | *Enables cross-cutting concerns like logging, security, transactions* | *Logging all service calls or securing access based on roles* |
| ***Web & Web MVC*** | *Handles HTTP requests, REST APIs, controllers, and view resolvers* | *REST endpoints in e-commerce app: /products, /cart, /checkout* |
| ***DAO & ORM*** | *JDBC abstraction, integrates with ORM frameworks (Hibernate, JPA)* | *Accessing MySQL via Spring Data JPA in inventory systems* |
| ***Messaging*** | *Supports asynchronous messaging via JMS, AMQP* | *Order confirmation messages using RabbitMQ or Kafka* |
| ***Testing*** | *Integration with JUnit, Mockito, and TestContext framework* | *Writing unit tests for service and repository layers in enterprise applications* |

*🟩* ***Real-Time Application Scenario (E-Commerce System)****:*

* ***Web MVC****: For exposing endpoints like /add-to-cart*
* ***DAO/ORM****: For connecting to MySQL via Hibernate*
* ***AOP****: To log user actions like login or checkout*
* ***Context****: To load configuration files and resources*
* ***Messaging****: To send confirmation emails after purchase*
* ***Testing****: To verify payment service logic using mocks*

*🟩* ***Tricky/Indirect Interview Questions****:*

* *Which Spring module would you use for logging all service method calls without changing the core logic?*
* *Is it possible to use Spring without Spring MVC?*
* *What is the difference between ApplicationContext and BeanFactory, and which module do they belong to?*
* *How would you decouple your data access logic from business logic in Spring?*

*🟩* ***Common Exceptions by Module****:*

| ***Exception*** | ***Cause*** |
| --- | --- |
| *BeanDefinitionStoreException* | *Invalid configuration or missing class in Java/XML config* |
| *UnsatisfiedDependencyException* | *Autowiring failed – dependency not available* |
| *NoSuchBeanDefinitionException* | *Trying to inject a bean not defined in the container* |
| *ConversionNotSupportedException* | *Type conversion failure in property injection or request mapping* |
| *ApplicationContextException* | *Failed to initialize Spring context* |

*🟩* ***Important Annotations per Module****:*

| ***Module*** | ***Annotation*** | ***Purpose / Notes*** |
| --- | --- | --- |
| ***Core*** | *@Component, @Autowired, @Value* | *Used for defining beans and injecting dependencies* |
| ***Context*** | *@Configuration, @Bean, @PropertySource* | *Java-based configuration, externalizing properties* |
| ***AOP*** | *@Aspect, @Before, @AfterReturning* | *Defining and applying cross-cutting concerns like logging* |
| ***Web MVC*** | *@Controller, @RestController, @RequestMapping* | *Handling web requests, RESTful services* |
| ***DAO / ORM*** | *@Repository, @Transactional* | *Defining DAO beans, managing transactions* |
| ***Messaging*** | *@JmsListener, @SendTo* | *Receiving/sending JMS messages* |
| ***Testing*** | *@RunWith(SpringRunner.class), @SpringBootTest* | *Writing integration tests using Spring Boot* |

*🟩* ***Deep Interview Insights****:*

*🔸 “Can I use Spring's AOP without using the Web module?”  
Yes. AOP is standalone. You can apply logging or security aspects in console-based or batch-processing Spring apps.*

*🔸 “What’s the difference between @Component, @Service, and @Repository?”  
They all mark beans, but:*

* *@Service is for business logic layer*
* *@Repository is for DAO layer and supports exception translation*
* *@Component is a generic stereotype*

*🔸 “What is the advantage of separating modules?”  
Better* ***separation of concerns****,* ***memory optimization****,* ***easy testing****, and* ***better modular design*** *for microservices.*

*🟩* ***Conclusion****:  
Spring modules offer a* ***plug-and-play architecture****. You only load what you need, reducing memory footprint while maintaining enterprise-grade capabilities like DI, AOP, transaction handling, and test support — all under one roof.*

*✅* ***5) What is IoC (Inversion of Control)?***

*🟩* ***Definition****:  
Inversion of Control (IoC) is a design principle where the control of creating and injecting dependencies into a class is given to the Spring container instead of the class managing its own dependencies. This leads to* ***loose coupling****, better* ***testability****, and* ***flexibility*** *in application design.*

*🟩* ***Detailed Explanation****:  
Traditionally, an object creates other objects it depends on. But in IoC, the object is not responsible for finding or creating its dependencies. Instead, the Spring container* ***manages object creation, configuration, and assembly****.*

*IoC is the foundation of the* ***Spring Framework****, implemented using* ***Dependency Injection (DI)****.*

*🟩* ***Real-Life Analogy****:  
Think of a car that doesn’t assemble its own engine. Instead, the* ***manufacturer (container)*** *installs the engine (dependency). The car just uses it.*

*🟩* ***Real-Time Application****:  
Suppose you're developing a banking app. AccountService needs a TransactionService. Instead of manually instantiating it:*

*java*

*CopyEdit*

*@Service*

*public class AccountService {*

*@Autowired*

*private TransactionService transactionService;*

*}*

*Now TransactionService is injected by Spring at runtime via IoC.*

*🟩* ***Tricky/Indirect Questions****:*

* *What does "Hollywood Principle" mean in Spring IoC? ("Don't call us, we’ll call you" — objects don’t find dependencies; container gives them.)*
* *Does IoC lead to tighter or looser coupling?*
* *Is IoC tied only to Spring? (No, but Spring popularized it in Java.)*

*🟩* ***Common Exceptions****:*

* *NoSuchBeanDefinitionException: If the bean isn’t defined in the context.*
* *BeanInstantiationException: If Spring cannot instantiate the bean (e.g., abstract class).*
* *UnsatisfiedDependencyException: Required dependency not found or ambiguous.*

*🟩* ***Important Annotations****:*

| ***Annotation*** | ***Purpose*** | ***Level*** | ***Notes*** |
| --- | --- | --- | --- |
| *@Component* | *Marks a class as a Spring-managed bean* | *Class* | *Enables component scanning* |
| *@Autowired* | *Injects dependency automatically* | *Field, Constructor, Method* | *Can be used with or without @Component* |
| *@Qualifier* | *Resolves ambiguity when multiple beans exist* | *Field* | *Used with @Autowired to specify exact bean* |
| *@Primary* | *Marks a bean as preferred when multiple exist* | *Class* | *Avoids use of @Qualifier in some cases* |

*🟩* ***How Interviewer May Go Deeper****:*

* *What’s the difference between IoC and DI?*
* *Can you implement IoC manually without Spring?*
* *What if one dependency depends on another? How does Spring resolve it?*

*✅* ***6) Submodules of IoC in Spring***

*🟩* ***Definition****:  
The IoC container in Spring is split into logical submodules to separate responsibilities and provide flexibility in configuration and bean handling.*

*🟩* ***List of IoC Submodules****:*

1. ***Core****: Contains the foundational features of the Spring framework like BeanFactory.*
2. ***Beans****: Deals with bean creation, lifecycle, and metadata management.*
3. ***Context****: Provides a way to access beans, internationalization, event propagation.*
4. ***Expression Language (SpEL)****: Allows querying and manipulating object graphs dynamically.*

*🟩* ***Real-Time Application****: For a dynamic dashboard application:*

* *Beans for defining UI elements.*
* *Context for loading properties and resource bundles.*
* *SpEL for evaluating dynamic pricing formulas.*

*🟩* ***Tricky Questions****:*

* *How is ApplicationContext different from BeanFactory?*
* *What happens if you use SpEL in @Value and the property is missing?*
* *Can we have multiple ApplicationContexts in one app?*

*🟩* ***Common Exceptions****:*

* *BeanDefinitionStoreException: Errors in loading bean definitions.*
* *SpelEvaluationException: Malformed or invalid expressions in SpEL.*
* *NoSuchMessageException: Internationalization message key not found.*

*🟩* ***Annotations (SpEL & Context)****:*

* *@Value("#{2 \* 5}"): Injects expression result.*
* *@PropertySource("classpath:app.properties"): Loads external config.*
* *@ImportResource("beans.xml"): Mix XML with annotation config.*

*🟩* ***How Interviewer May Go Deeper****:*

* *Can you combine annotation and XML configuration?*
* *How does Spring resolve expressions in SpEL internally?*
* *What are some performance issues with large ApplicationContexts?*

*✅* ***7) What is a Container in Spring?***

*🟩* ***Definition****:  
A* ***Spring container*** *is the heart of the framework that creates, configures, manages, and destroys beans. It uses metadata (XML, annotations, Java config) to perform dependency injection and manage bean lifecycles.*

*🟩* ***Responsibilities****:*

* *Read configuration*
* *Instantiate and inject dependencies (beans)*
* *Manage bean scopes and lifecycle*
* *Apply AOP proxies and event publishing*

*🟩* ***Types of Containers****:*

* ***BeanFactory*** *(basic container, lazy loading)*
* ***ApplicationContext*** *(advanced, eager loading, i18n, events)*

*🟩* ***Real-Time Scenario****: An inventory system might use:*

* *ApplicationContext for loading multiple config files and enabling internationalization.*
* *BeanFactory for lightweight memory-constrained environments.*

*🟩* ***Tricky/Indirect Questions****:*

* *How does Spring know when to destroy a bean?*
* *Can we use ApplicationContext in standalone Java applications?*
* *What happens when two containers are nested?*

*🟩* ***Exceptions****:*

* *ApplicationContextException: Failed to start the context.*
* *BeanDefinitionParsingException: Misconfiguration in XML or annotations.*
* *IllegalStateException: Accessing beans before context initialization.*

*🟩* ***Important Annotations for Context Loading****:*

* *@Configuration: Used in Java-based configuration classes.*
* *@ComponentScan: Auto-detects and registers components.*
* *@Import: Imports other configuration classes.*

*🟩* ***How Interviewer May Go Deeper****:*

* *What’s the lifecycle of a Spring container?*
* *Can we reload an ApplicationContext at runtime?*
* *How does Spring container relate to Servlet container?*

*✅* ***8) Types of Spring Containers (BeanFactory vs ApplicationContext)***

*🟩* ***BeanFactory****:*

* *Basic container*
* *Lazy loading (beans created only when needed)*
* *Minimal memory usage*
* *Used in lightweight applications*

*🟩* ***ApplicationContext****:*

* *Advanced container*
* *Eager loading (all beans created on startup)*
* *Supports internationalization, events, annotation scanning*
* *Widely used in enterprise and web apps*

*🟩* ***Real-Time Analogy****: BeanFactory is like a kiosk that gives items only when requested. ApplicationContext is like a buffet — everything is ready before you ask.*

*🟩* ***Tricky Interview Questions****:*

* *Why is ApplicationContext preferred over BeanFactory?*
* *Can you override ApplicationContext behavior?*
* *What’s the tradeoff between lazy and eager loading?*

*🟩* ***Common Exceptions****:*

* *BeanNotOfRequiredTypeException: Wrong type returned from context.*
* *BeanIsAbstractException: Trying to use an abstract class as a bean.*
* *NoSuchBeanDefinitionException: Bean not present in context.*

*🟩* ***Key Classes****:*

| ***Container*** | ***Class*** | ***Notes*** |
| --- | --- | --- |
| *BeanFactory* | *XmlBeanFactory (deprecated)* | *Now replaced by ApplicationContext* |
| *ApplicationContext* | *ClassPathXmlApplicationContext* | *Loads from classpath* |
|  | *AnnotationConfigApplicationContext* | *Loads Java config classes* |
|  | *FileSystemXmlApplicationContext* | *Loads XML from filesystem* |

*🟩* ***How Interviewer May Go Deeper****:*

* *Internals of ApplicationContext refresh mechanism*
* *How bean post-processors are applied in context*
* *Lazy loading in ApplicationContext — is it possible?*

*✅* ***9) What is the difference between Core (BeanFactory) and J2EE (ApplicationContext) containers in Spring?***

*🟩* ***Concept Overview****: Spring provides two major types of IoC containers:*

* ***BeanFactory*** *(from org.springframework.beans.factory)*
* ***ApplicationContext*** *(from org.springframework.context)*

*Both are responsible for managing Spring beans but differ significantly in features and usage.*

*🟩* ***Detailed Comparison Table****:*

| ***Feature*** | ***BeanFactory*** | ***ApplicationContext*** |
| --- | --- | --- |
| ***Package*** | *org.springframework.beans.factory* | *org.springframework.context* |
| ***XML Loading*** | *Requires manual resource loading* | *Directly reads XML via classpath or filesystem* |
| ***Default Loading*** | ***Lazy Loading*** *of beans* | ***Eager Loading*** *of beans* |
| ***Support for AOP, Events*** | *Limited or none* | *Fully supported* |
| ***Internationalization*** | *Not supported* | *Supported* |
| ***Bean Post Processors*** | *Not supported* | *Supported* |
| ***Use Case*** | *Lightweight, memory-sensitive apps* | *Full-featured enterprise apps* |
| ***Deprecated*** | *Yes, in favor of ApplicationContext* | *Preferred* |

*🟩* ***Real-Time Usage Example****:*

*java*

*CopyEdit*

*// BeanFactory (manual loading)*

*Resource resource = new ClassPathResource("beans.xml");*

*BeanFactory factory = new XmlBeanFactory(resource); // Deprecated*

*// ApplicationContext (preferred way)*

*ApplicationContext context = new ClassPathXmlApplicationContext("beans.xml");*

*MyService myService = context.getBean(MyService.class);*

*🟩* ***Tricky/Indirect Questions****:*

* *What’s the difference in bean instantiation between BeanFactory and ApplicationContext?*
* *Why is ApplicationContext preferred in enterprise apps?*
* *Is lazy loading always an advantage in microservices?*

*🟩* ***Common Exceptions****:*

* *BeanDefinitionStoreException: Issues while reading config files*
* *NoSuchBeanDefinitionException: Bean not found in config*
* *ClassCastException: Incorrect bean type casting*

*🟩* ***Relevant Annotations****: While these apply mostly in annotation-based configs, ApplicationContext is required to interpret:*

* *@ComponentScan*
* *@Configuration*
* *@Bean*

*🟩* ***Interview Tip****: Interviewers may ask you to justify why ApplicationContext is better — focus on support for* ***AOP, events, i18n, BeanPostProcessors****, and* ***multiple file support****.*

*✅* ***10) What is Injection in Spring?***

*🟩* ***Concept****: Injection in Spring refers to the process of providing values, objects, or configurations to a class (bean) from outside rather than the bean itself creating them. It helps achieve loose coupling and easier testability.*

*🟩* ***Types of Values Injected****:*

* *Primitive values (like int, String)*
* *Object references (dependencies)*
* *Collection types (List, Set, Map)*
* *Environment properties*

*🟩* ***Real-Time Analogy****: Think of injection as a power socket — instead of hardwiring an appliance, you "plug in" the power (dependency) when needed.*

*🟩* ***Code Example****:*

*java*

*CopyEdit*

*@Component*

*public class OrderService {*

*@Autowired*

*private PaymentService paymentService; // injected automatically*

*}*

*🟩* ***Tricky/Indirect Questions****:*

* *Why should an object not create its own dependencies?*
* *How does Spring promote dependency inversion?*
* *Can Spring inject a List or Map?*

*🟩* ***Common Exceptions****:*

* *UnsatisfiedDependencyException: Required dependency not available*
* *NoSuchBeanDefinitionException: Spring can't find the bean to inject*

*🟩* ***Annotations Used****:*

* *@Autowired: For injection*
* *@Value: For primitives or environment properties*
* *@Qualifier: When multiple candidates exist*

*🟩* ***Extra Note****: Injection is part of the larger concept* ***Dependency Injection****, a design pattern enabled by Spring's IoC.*

*✅* ***11) What is Dependency Injection (DI)?***

*🟩* ***Definition****: Dependency Injection is a design pattern where dependencies (i.e., collaborating objects) are provided to a class by an external entity (Spring container), instead of the class creating them itself.*

*🟩* ***Why DI is Important****:*

* *Promotes loose coupling*
* *Easier testing using mock dependencies*
* *Enhances reusability and separation of concerns*
* *Improves code maintainability*

*🟩* ***Real-Life Scenario****: In a banking app, LoanService may depend on InterestCalculator. Rather than creating it inside the service, Spring injects the right InterestCalculator implementation at runtime.*

*🟩* ***Example****:*

*java*

*CopyEdit*

*@Component*

*public class LoanService {*

*private InterestCalculator calculator;*

*@Autowired*

*public LoanService(InterestCalculator calculator) {*

*this.calculator = calculator;*

*}*

*}*

*🟩* ***Tricky/Indirect Questions****:*

* *How does Spring achieve loose coupling?*
* *What are the real-world benefits of DI?*
* *Is DI the same as IoC?*

*🟩* ***Exceptions****:*

* *BeanCreationException: Error during bean construction*
* *TypeMismatchException: Incorrect data type provided*
* *NoSuchBeanDefinitionException: Missing bean for injection*

*🟩* ***Annotations Used****:*

* *@Autowired, @Inject*
* *@Qualifier, @Primary*

*🟩* ***Extra Tip****: You can implement DI using:*

* ***XML configuration***
* ***Annotation-based configuration***
* ***Java-based configuration*** *(@Bean methods inside @Configuration class)*

*✅* ***12) What are the types of Dependency Injection in Spring?***

*🟩* ***Types****:*

1. ***Constructor-Based Injection***
2. ***Setter/Property-Based Injection***
3. ***Field Injection*** *(not recommended for testing or immutability)*

*🟩* ***Comparison Table****:*

| ***Feature*** | ***Constructor Injection*** | ***Setter/Property Injection*** |
| --- | --- | --- |
| ***How it works*** | *Via constructor parameters* | *Via public setter methods* |
| ***Order of injection*** | *Mandatory (all at once)* | *Flexible (can be partial)* |
| ***Immutability*** | *Supports immutability* | *Doesn’t enforce immutability* |
| ***Testability*** | *More testable* | *Less strict* |
| ***Overriding values*** | *Not possible* | *Possible* |

*🟩* ***Code Example (Constructor)****:*

*java*

*CopyEdit*

*@Autowired*

*public PaymentService(PaymentGateway gateway) {*

*this.gateway = gateway;*

*}*

*🟩* ***Code Example (Setter)****:*

*java*

*CopyEdit*

*@Autowired*

*public void setGateway(PaymentGateway gateway) {*

*this.gateway = gateway;*

*}*

*🟩* ***Tricky/Indirect Questions****:*

* *Which DI type is preferred for required dependencies?*
* *Can we use both types together?*
* *Is it possible to override constructor-injected values?*

*🟩* ***Exceptions****:*

* *UnsatisfiedDependencyException: Missing bean for parameter/setter*
* *BeanInstantiationException: Error while invoking constructor*

*🟩* ***Annotations****:*

* *@Autowired, @Qualifier, @Inject*

*🟩* ***Best Practices****:*

* *Use* ***constructor injection*** *for required dependencies*
* *Use* ***setter injection*** *for optional ones*
* *Avoid* ***field injection*** *in large-scale, test-heavy applications*

*✅* ***13) What is the difference between Constructor-based and Property (Setter)-based Injection?***

*🟩* ***Core Idea****:  
Both approaches are used to inject dependencies into Spring beans. The main difference lies in how the dependency is passed — via* ***constructor parameters*** *or* ***setter methods****.*

*🟩* ***Detailed Comparison****:*

| ***Feature*** | ***Constructor Injection*** | ***Setter (Property) Injection*** |
| --- | --- | --- |
| ***Injection Method*** | *Through class constructor* | *Through public setter method* |
| ***Required Dependencies*** | *Mandatory — injected at object creation* | *Optional — can be injected later* |
| ***Immutability*** | *Supports immutable objects* | *Does* ***not*** *support immutability* |
| ***Object State Safety*** | *Always initialized when object is created* | *May result in partially initialized objects* |
| ***Testing & Mocking*** | *Easy to test using constructor-based DI* | *Slightly harder due to mutable state* |
| ***When to Use*** | *For mandatory and final dependencies* | *For optional or configurable dependencies* |
| ***Best Practice*** | *Preferred in modern Spring (especially with Lombok)* | *Only if the dependency is optional or circular* |

*🟩* ***Real-Time Code Examples****:*

*✅ Constructor Injection:*

*java*

*CopyEdit*

*@Component*

*public class OrderService {*

*private final PaymentService paymentService;*

*@Autowired*

*public OrderService(PaymentService paymentService) {*

*this.paymentService = paymentService;*

*}*

*}*

*✅ Setter Injection:*

*java*

*CopyEdit*

*@Component*

*public class OrderService {*

*private PaymentService paymentService;*

*@Autowired*

*public void setPaymentService(PaymentService paymentService) {*

*this.paymentService = paymentService;*

*}*

*}*

*🟩* ***Tricky/Indirect Interview Questions****:*

* *Why is constructor injection considered more robust?*
* *What’s the danger of using setter injection excessively?*
* *Can I mix constructor and setter injection in the same class?*

*🟩* ***Common Exceptions****:*

* *UnsatisfiedDependencyException: If Spring can't resolve a constructor or setter dependency*
* *BeanCurrentlyInCreationException: Due to circular dependency (setter injection may be preferred here)*

*🟩* ***Best Practices****:*

* *Prefer constructor injection for mandatory dependencies.*
* *Use setter injection* ***only*** *for optional dependencies or to break circular dependencies.*
* *Use @Autowired(required = false) for optional setters.*

*🟩* ***Annotations Used****:*

* *@Autowired*
* *@Qualifier (to resolve multiple bean ambiguity)*
* *@Inject (JSR-330 alternative)*

*🟩* ***How Interviewer May Go Deeper****:*

* *"What would happen if a required constructor dependency isn't provided?"*
* *"What are the pros and cons of making your service class mutable?"*
* *"Why does Spring Boot prefer constructor injection by default?"*

*✅* ***14) How to handle Circular Dependency in Spring?***

*🟩* ***Concept****:  
A* ***circular dependency*** *occurs when two or more beans depend on each other for initialization. For example:*

* *A needs B*
* *B needs A*

*This creates a loop that the Spring container cannot resolve during* ***constructor injection****.*

*🟩* ***Example of Circular Dependency****:*

*java*

*CopyEdit*

*@Component*

*public class A {*

*@Autowired*

*private B b;*

*}*

*@Component*

*public class B {*

*@Autowired*

*private A a;*

*}*

*🟩* ***Spring’s Handling****:*

* ***Setter Injection****: Spring can handle circular dependencies via setter injection as it allows object creation first, and then dependency injection.*
* ***Constructor Injection****: Spring* ***cannot*** *resolve circular dependencies with constructor injection and throws an exception.*

*🟩* ***Tricky/Indirect Questions****:*

* *Why can't constructor injection resolve circular references?*
* *Can Spring resolve circular dependencies automatically?*
* *What design change would you make to avoid circular dependencies?*

*🟩* ***Solutions****:*

1. ***Use Setter Injection****: Spring can inject after object instantiation, resolving the cycle.*
2. ***Use @Lazy Annotation****:*

*java*

*CopyEdit*

*@Autowired*

*@Lazy*

*private B b;*

*Lazy initializes the dependency only when it’s accessed.*

1. ***Refactor the Design****: Extract common logic into a third bean (Mediator or Helper) to remove circular dependency.*

*🟩* ***Common Exceptions****:*

* *BeanCurrentlyInCreationException: Indicates circular reference during bean construction.*
* *StackOverflowError: Rare, if circular calls happen during runtime.*

*🟩* ***Annotations Used****:*

* *@Lazy: Helps delay the instantiation*
* *@Autowired: With setters to help avoid constructor cycles*

*🟩* ***Interview Tip****: Explain the difference in how* ***constructor*** *and* ***setter injection*** *handle bean instantiation order, and why constructor injection fails in a circular loop.*

*✅* ***15) What are different types of Spring Containers?***

*🟩* ***Overview****: Spring provides multiple containers for different scenarios. Each container is responsible for instantiating, configuring, and managing Spring beans.*

*🟩* ***Types of Containers****:*

| ***Container Type*** | ***Class Used*** | ***Features*** |
| --- | --- | --- |
| ***BeanFactory*** | *XmlBeanFactory (deprecated)* | *Basic IoC, Lazy Initialization* |
| ***ApplicationContext*** | *ClassPathXmlApplicationContext, AnnotationConfigApplicationContext* | *Advanced container with AOP, Events, i18n* |
| ***WebApplicationContext*** | *XmlWebApplicationContext* | *Web-aware ApplicationContext for Spring MVC* |
| ***GenericWebApplicationContext*** | *From Spring 5.x* | *For programmatic/bootstrapped MVC apps* |
| ***AnnotationConfigWebApplicationContext*** | *Annotation-based MVC config* | *Preferred in modern Spring Boot apps* |

*🟩* ***Tricky/Indirect Questions****:*

* *Why don’t we use BeanFactory anymore?*
* *Can you use ApplicationContext in non-web applications?*
* *What’s the role of WebApplicationContext in Spring MVC?*

*🟩* ***Use Case Examples****:*

* *ClassPathXmlApplicationContext: Used in standalone desktop or batch apps*
* *WebApplicationContext: Automatically loaded by DispatcherServlet in Spring MVC*
* *AnnotationConfigApplicationContext: Used in Spring Boot and annotation-driven apps*

*🟩* ***Common Errors****:*

* *BeanDefinitionStoreException: Misconfigured XML or annotations*
* *NoSuchBeanDefinitionException: When container can't find the specified bean*

*🟩* ***Important Notes****:*

* ***BeanFactory*** *is still internally used by ApplicationContext but shouldn't be used directly.*
* ***WebApplicationContext*** *inherits from* ***ApplicationContext*** *and adds web-specific features like ServletContext.*

*🟩* ***Interview Tip****: Mention that while ApplicationContext is used for most applications,* ***WebApplicationContext*** *is specific to* ***Spring MVC web apps****, and Spring Boot abstracts the container setup through annotations.*

*✅* ***16) What is the role of ApplicationContext in Spring?***

*🟩* ***Definition****:  
ApplicationContext is the* ***central interface*** *of the Spring IoC container. It’s responsible for:*

* ***Creating beans***
* ***Configuring beans***
* ***Managing their complete life cycle*** *It is an* ***advanced version of BeanFactory*** *with added features like event publishing, AOP, message resources, internationalization, and annotation support.*

*🟩* ***Key Features****:*

* *Supports eager initialization (beans created during container startup)*
* *Provides BeanPostProcessors and BeanFactoryPostProcessors*
* *Offers resource loading capabilities (getResource)*
* *Publishes events to registered listeners (ApplicationEventPublisher)*
* *Supports annotation-based configuration (@Component, @Configuration, etc.)*

*🟩* ***Common Implementations****:*

| ***Implementation*** | ***Use Case*** |
| --- | --- |
| *ClassPathXmlApplicationContext* | *Loads beans from XML in classpath* |
| *FileSystemXmlApplicationContext* | *Loads XML from filesystem path* |
| *AnnotationConfigApplicationContext* | *Loads beans using Java Config (@Configuration)* |

*🟩* ***Real-Time Code Example****:*

*java*

*CopyEdit*

*ApplicationContext context = new AnnotationConfigApplicationContext(AppConfig.class);*

*MyService service = context.getBean(MyService.class);*

*🟩* ***Tricky/Indirect Interview Questions****:*

* *Why is ApplicationContext preferred over BeanFactory?*
* *How does Spring resolve annotations internally?*
* *Can ApplicationContext manage beans declared in external modules?*

*🟩* ***Common Exceptions****:*

* *ApplicationContextException: General errors during context initialization*
* *BeanDefinitionStoreException: Configuration issues*
* *NoSuchBeanDefinitionException: Bean not found*
* *UnsatisfiedDependencyException: Dependency missing during bean creation*

*🟩* ***Annotations That Work with ApplicationContext****:*

* *@ComponentScan: Tells the context to scan for components*
* *@Bean: Declares bean in @Configuration class*
* *@Configuration: Declares a Java-based Spring configuration class*
* *@PropertySource: Loads external .properties files into the context*

*🟩* ***Real-Time Scenario****: In a Spring Boot e-commerce app, ApplicationContext is used to load all services like:*

* *CartService*
* *OrderService*
* *UserAuthService …and manage their full life cycle, dependency wiring, and configuration.*

*🟩* ***How Interviewer May Dig Deeper****:*

* *What’s the difference between ApplicationContext and WebApplicationContext?*
* *How does Spring’s context support AOP or Events internally?*
* *Can I have multiple ApplicationContexts in one app?*

*🟩* ***Bonus – Can We Create Child Contexts?*** *Yes! ApplicationContext supports hierarchical contexts using:*

*java*

*CopyEdit*

*GenericApplicationContext parent = new GenericApplicationContext();*

*AnnotationConfigApplicationContext child = new AnnotationConfigApplicationContext();*

*child.setParent(parent);*

*Useful for* ***modular*** *applications like different modules for Admin, Customer, and Vendor.*

*✅* ***17) BeanFactory vs ApplicationContext – In-Depth Comparison***

*🟩* ***Definition and Core Difference****:  
Both BeanFactory and ApplicationContext are interfaces representing* ***IoC containers*** *in Spring.  
However,* ***ApplicationContext is a superset*** *of BeanFactory, offering more enterprise-level features like internationalization, event handling, AOP, and annotation processing.*

*🟩* ***Comparison Table****:*

| ***Feature*** | ***BeanFactory*** | ***ApplicationContext*** |
| --- | --- | --- |
| *Bean Loading* | *Lazy (on-demand)* | *Eager (at startup)* |
| *Annotation Support (@Component)* | *Not supported by default* | *Fully supported* |
| *Event Handling (ApplicationEvent)* | *Not supported* | *Supported* |
| *MessageSource / i18n Support* | *Not supported* | *Supported* |
| *AOP Proxy Creation* | *Manual* | *Automatic* |
| *Environment Abstraction* | *No* | *Yes (Environment abstraction)* |
| *BeanPostProcessor / FactoryProcessor* | *Manual registration needed* | *Automatic discovery* |
| *Web Context Integration* | *Not applicable* | *WebApplicationContext extends it* |
| *Preferred For* | *Lightweight, non-enterprise apps* | *Full-scale, enterprise, web applications* |

*🟩* ***Real-Life Analogy****:*

* *BeanFactory is like a* ***bike****—light, minimal but not feature-rich.*
* *ApplicationContext is like a* ***car****—more powerful with additional built-in features (AC, GPS = AOP, i18n, Events).*

*🟩* ***Code Example – BeanFactory****:*

*java*

*CopyEdit*

*Resource res = new ClassPathResource("beans.xml");*

*BeanFactory factory = new XmlBeanFactory(res);*

*MyService service = (MyService) factory.getBean("myService");*

*🟩* ***Code Example – ApplicationContext****:*

*java*

*CopyEdit*

*ApplicationContext context = new ClassPathXmlApplicationContext("beans.xml");*

*MyService service = context.getBean(MyService.class);*

*🟩* ***Tricky/Indirect Interview Questions****:*

* *Why does Spring recommend ApplicationContext over BeanFactory?*
* *When should you use BeanFactory instead of ApplicationContext?*
* *Can you use annotation-based configuration in BeanFactory?*
* *If ApplicationContext loads all beans eagerly, can this cause memory issues?*

*🟩* ***Common Exceptions in Comparison****:*

| ***Exception*** | ***Common With*** | ***Reason*** |
| --- | --- | --- |
| *BeanCreationException* | *Both* | *Dependency injection failure* |
| *NoSuchBeanDefinitionException* | *Both* | *Bean not found* |
| *BeanDefinitionStoreException* | *Mostly ApplicationContext* | *Bean XML/Java config issues* |
| *UnsupportedOperationException* | *BeanFactory* | *When accessing unsupported features like events* |

*🟩* ***Annotations That Work Only With ApplicationContext****:*

* *@ComponentScan: Scans for components automatically*
* *@PropertySource: Loads external properties*
* *@EventListener: Handles events inside Spring context*

*🟩* ***Real-Time Use Case****: In* ***Spring MVC or Spring Boot*** *apps, ApplicationContext is used as it:*

* *Automatically injects beans*
* *Handles internal events (e.g., ContextRefreshedEvent)*
* *Supports annotations and REST API controllers (@RestController)*

*🟩* ***Interview Deep Dive Pointers****:*

* *Can both BeanFactory and ApplicationContext be used in the same application?  
  ✅ Yes. ApplicationContext extends BeanFactory. Internally, it still uses DefaultListableBeanFactory.*
* *Can you delay bean creation in ApplicationContext like BeanFactory?  
  ✅ Use @Lazy annotation to delay initialization.*
* *Which one is better for unit testing?  
  ✅ ApplicationContext, as it supports profiles, annotation configs, and mock beans more easily.*

*✅* ***18) What are Spring Bean Scopes with Real-Time Examples?***

*🟩* ***Definition****:  
Spring Bean Scopes define* ***how many instances*** *of a bean will be created and* ***how long they will live*** *in the Spring container. It controls the lifecycle and visibility of a bean.*

*🟩* ***Types of Bean Scopes in Spring Core****:*

| ***Scope*** | ***Description*** | ***Example Usage*** |
| --- | --- | --- |
| *singleton* | *Single shared instance per Spring container (default)* | *Services, DAOs, Business Logic Beans* |
| *prototype* | *A new instance created every time it's requested* | *Image processors, objects with mutable state* |
| *request* | *One instance per HTTP request (Spring Web only)* | *Web controllers, form data objects* |
| *session* | *One instance per HTTP session (Spring Web only)* | *User session-scoped beans* |
| *application* | *One instance per ServletContext (Spring Web only)* | *Application-wide configuration* |
| *websocket* | *One instance per WebSocket connection* | *Real-time messaging applications* |

*🟩* ***Default Scope in Spring****:  
singleton is the default bean scope if none is explicitly defined.*

*🟩* ***Real-Time Code Examples****:*

*✅ Singleton Scope:*

*java*

*CopyEdit*

*@Component*

*@Scope("singleton") // Optional, as it's default*

*public class NotificationService {*

*// Shared across entire application*

*}*

*✅ Prototype Scope:*

*java*

*CopyEdit*

*@Component*

*@Scope("prototype")*

*public class PdfReportGenerator {*

*// Each user gets a new report instance*

*}*

*✅ Request Scope (Web App Only):*

*java*

*CopyEdit*

*@Component*

*@Scope("request")*

*public class RequestScopedBean {*

*// New instance per HTTP request*

*}*

*🟩* ***Tricky/Indirect Interview Questions****:*

* *What scope would you use for a bean that should be created fresh for every request?*
* *Can you use @Autowired on a prototype bean inside a singleton?*
* *How can session scope beans be injected into singleton-scoped beans?*

*🟩* ***Interview Trap Concept – Scope Mismatch****: Injecting a prototype bean into a singleton does NOT give a new instance each time. Spring injects it once during singleton creation.*

*✅* ***Solution****: Use ObjectFactory<T> or Provider<T>:*

*java*

*CopyEdit*

*@Autowired*

*private ObjectFactory<ReportGenerator> generatorFactory;*

*public void generate() {*

*ReportGenerator generator = generatorFactory.getObject();*

*}*

*🟩* ***Scope Usage Examples by Context****:*

| ***App Type*** | ***Scope Usage*** |
| --- | --- |
| *REST API* | *Singleton for services, Request for DTOs* |
| *Desktop App* | *Prototype for GUI components* |
| *Chat App* | *WebSocket scope for session isolation* |
| *Banking System* | *Session scope for login state and user cache* |

*🟩* ***Common Scope-Related Exceptions****:*

| ***Exception*** | ***Cause*** |
| --- | --- |
| *IllegalStateException* | *Using request/session scope in non-web context* |
| *ScopeNotActiveException* | *Accessing session/request bean outside active lifecycle* |
| *BeanCreationException* | *Misconfigured scope or circular dependencies* |

*🟩* ***Annotations for Bean Scopes****:*

* *@Scope("singleton"): Default scope*
* *@Scope("prototype"): New instance on every request*
* *@Scope("request"): Tied to HTTP request (needs web context)*
* *@Scope("session"): Tied to HTTP session*
* *@Scope(value = "request", proxyMode = ScopedProxyMode.TARGET\_CLASS): Needed to inject request/session scoped beans into singleton beans*

*🟩* ***Deep Interview Probes****:*

* *How do you inject a prototype bean into a singleton safely?  
  → Use ObjectFactory, Provider, or @Lookup.*
* *How do scopes work in Spring Boot vs Spring Web MVC?  
  → Same concept, but scopes like request/session require Spring Web (Servlet-based context).*
* *Can you change bean scope at runtime?  
  → No. Bean scope is fixed once the context is initialized unless you're manually managing bean creation.*

*✅* ***19) What is Dependency Injection (DI) in Spring with Examples and Variants?***

*🟩* ***Definition****:****Dependency Injection*** *is a core principle of Inversion of Control (IoC). It means Spring will provide the required dependencies (objects) to a class, instead of the class creating them itself. This makes code loosely coupled, testable, and maintainable.*

*🟩* ***Why is it needed?*** *Tight coupling makes classes hard to test, reuse, and extend. DI promotes* ***loose coupling****, where components are not directly dependent on each other’s implementation.*

*🟩* ***Types of Dependency Injection in Spring****:*

| ***Type*** | ***Mechanism*** | ***Usage Example*** |
| --- | --- | --- |
| *Constructor-based* | *Inject through constructor* | *Immutable objects, required dependencies* |
| *Setter-based* | *Inject through setters* | *Optional or late-initialized dependencies* |
| *Field-based* | *Inject directly in fields* | *Less recommended due to testing issues* |
| *Interface-based* | *Rare in Spring* | *Used more in frameworks like Java EE* |

*🟩* ***Real-Time Code Examples****:*

*✅ Constructor Injection (Recommended):*

*java*

*CopyEdit*

*@Component*

*public class OrderService {*

*private final PaymentService paymentService;*

*@Autowired*

*public OrderService(PaymentService paymentService) {*

*this.paymentService = paymentService;*

*}*

*}*

*✅ Setter Injection:*

*java*

*CopyEdit*

*@Component*

*public class NotificationService {*

*private EmailSender emailSender;*

*@Autowired*

*public void setEmailSender(EmailSender emailSender) {*

*this.emailSender = emailSender;*

*}*

*}*

*✅ Field Injection (Not recommended for testing):*

*java*

*CopyEdit*

*@Component*

*public class UserService {*

*@Autowired*

*private UserRepository repo;*

*}*

*🟩* ***Tricky/Indirect Interview Questions****:*

* *What happens if Spring can't inject a required dependency?*
* *When would you prefer constructor injection over setter?*
* *Can you inject prototype-scoped beans into singleton ones?*

*🟩* ***Common Exceptions in DI****:*

| ***Exception*** | ***Cause*** |
| --- | --- |
| *UnsatisfiedDependencyException* | *Missing or ambiguous dependency injection* |
| *NoSuchBeanDefinitionException* | *Spring can’t find the required bean* |
| *BeanCurrentlyInCreationException* | *Circular dependency detected* |
| *BeanInstantiationException* | *Spring cannot instantiate the bean (e.g., abstract/interface)* |

*🟩* ***Annotations Used in Dependency Injection****:*

| ***Annotation*** | ***Purpose*** | ***Notes*** |
| --- | --- | --- |
| *@Autowired* | *Performs injection by type* | *Can be used on constructors, fields, or setters* |
| *@Qualifier("beanName")* | *Resolves ambiguity when multiple beans of the same type* | *Must be used with @Autowired* |
| *@Inject* | *JSR-330 equivalent of @Autowired* | *No required attribute available* |
| *@Resource(name="beanName")* | *Injects by name* | *Comes from JSR-250* |
| *@Primary* | *Marks a bean as preferred when multiple candidates exist* | *Alternative to @Qualifier* |
| *@Value("${prop}")* | *Injects property or value* | *Useful for injecting config or constant values* |

*🟩* ***Real-Time Use Case****:  
In a* ***banking app****, AccountService might depend on TransactionService and AuditLogger. You inject both instead of creating them inside AccountService, which makes testing with mocks easier and changing dependencies more flexible.*

*🟩* ***Deep Interview Probes****:*

* *If two beans match the same type, which one gets injected?  
  → Use @Primary or @Qualifier to specify.*
* *What are the issues with field injection?  
  → Makes unit testing difficult since private fields can’t be easily mocked.*
* *Why is constructor injection preferred in Spring Boot?  
  → Encourages immutability, makes testing easy, and avoids null-related bugs.*
* *Can you mix constructor and setter injection?  
  → Yes, but best practice is to use constructor injection for required dependencies and setter for optional ones.*

*✅* ***20) What is the difference between ApplicationContext and BeanFactory in Spring?***

*🟩* ***Definition****:  
Both BeanFactory and ApplicationContext are* ***Spring IoC containers*** *that are responsible for managing beans and their lifecycles. However, ApplicationContext is a* ***more advanced*** *container built on top of BeanFactory with additional enterprise-level features.*

*🟩* ***Comparison Table****:*

| ***Feature*** | ***BeanFactory*** | ***ApplicationContext*** |
| --- | --- | --- |
| ***Eager/Lazy Loading*** | *Lazy by default* | *Eager by default* |
| ***Internationalization (i18n)*** | *Not supported* | *Fully supported* |
| ***Event Publishing*** | *Not supported* | *Supported (ApplicationEventPublisher)* |
| ***AOP Integration*** | *Manual setup required* | *Integrated* |
| ***Annotation Support*** | *Partial (manual)* | *Full annotation-based configuration support* |
| ***BeanPostProcessor/BeanFactoryPostProcessor*** | *Manual registration* | *Auto-detected* |
| ***Context Hierarchy (Parent/Child)*** | *Not supported* | *Supported* |
| ***Environment Abstraction*** | *No support* | *Environment abstraction supported (@Profile)* |
| ***Preferred in*** | *Lightweight apps, testing scenarios* | *Enterprise, web, boot apps* |

*🟩* ***Real-Time Usage****:*

* *In* ***simple CLI tools or micro-libraries****, BeanFactory might be sufficient to manage lightweight dependency injection.*
* *In* ***web apps, Spring Boot, or enterprise systems****, ApplicationContext is always used for its rich capabilities like AOP, eventing, and profiles.*

*🟩* ***Code Examples****:*

*✅ BeanFactory (Legacy):*

*java*

*CopyEdit*

*BeanFactory factory = new XmlBeanFactory(new ClassPathResource("beans.xml"));*

*MyService service = factory.getBean("myService", MyService.class);*

*✅ ApplicationContext (Modern):*

*java*

*CopyEdit*

*ApplicationContext context = new ClassPathXmlApplicationContext("beans.xml");*

*MyService service = context.getBean("myService", MyService.class);*

*🟩* ***Tricky/Indirect Interview Questions****:*

* *Why is ApplicationContext preferred in modern Spring applications?*
* *What is the internal difference in how beans are loaded in ApplicationContext and BeanFactory?*
* *Can you replace ApplicationContext with BeanFactory in a Spring Boot app?*
* *If you want to handle custom Spring events, which container would you choose?*

*🟩* ***Common Exceptions****:*

| ***Exception*** | ***Reason*** |
| --- | --- |
| *NoSuchBeanDefinitionException* | *Bean not registered or wrongly configured* |
| *ClassCastException* | *Bean fetched and cast incorrectly using context* |
| *BeanInstantiationException* | *Bean class could not be instantiated (abstract/interface)* |

*🟩* ***Important Interfaces & Annotations****:*

| ***Element*** | ***Purpose*** |
| --- | --- |
| *ApplicationContext* | *Interface extending BeanFactory with full features* |
| *ConfigurableApplicationContext* | *Allows context refresh/close programmatically* |
| *@Profile("dev")* | *Environment-specific bean registration* |
| *@PropertySource("classpath:xyz")* | *Inject externalized config values* |
| *ApplicationListener<Event>* | *Handle Spring events* |
| *Environment* | *Access to profiles, properties, and runtime environment info* |

*🟩* ***Deep Interview Probes****:*

* *Why does Spring Boot internally use ApplicationContext instead of BeanFactory?  
  → Because it needs advanced features like @Profile, AOP, events, and auto-configuration, which are not available in BeanFactory.*
* *What happens if we register a BeanPostProcessor in BeanFactory?  
  → It won’t be automatically detected — we must register it manually, unlike ApplicationContext.*
* *In a microservice with limited dependencies, should we still prefer ApplicationContext?  
  → Yes. The extra features cost minimal overhead and offer great flexibility, especially in modern apps.*

*✅* ***21) Explain Spring Bean Lifecycle with Phases and Hook Methods (@PostConstruct, @PreDestroy)***

*🟩* ***Definition****:  
The* ***Spring Bean Lifecycle*** *represents the* ***sequence of steps*** *through which a Spring bean goes — from creation to destruction — inside the* ***Spring IoC container****. Spring provides several hooks and interfaces to customize this lifecycle.*

*🟩* ***Lifecycle Phases*** *(Step-by-step breakdown):*

| ***Phase*** | ***Description*** |
| --- | --- |
| ***1. Instantiation*** | *Spring creates the bean instance using reflection.* |
| ***2. Populate Properties*** | *Spring performs dependency injection (setter/constructor-based).* |
| ***3. BeanNameAware*** | *If bean implements this, Spring passes the bean name.* |
| ***4. BeanFactoryAware / ApplicationContextAware*** | *Provides access to container references.* |
| ***5. Pre-initialization (BeanPostProcessor#postProcessBeforeInitialization)*** | *Custom logic before initialization methods.* |
| ***6. @PostConstruct / afterPropertiesSet() (InitializingBean)*** | *Initialization logic by annotation or interface.* |
| ***7. Post-initialization (BeanPostProcessor#postProcessAfterInitialization)*** | *Custom logic after initialization.* |
| ***8. Ready to use*** | *Bean is available for use in the application.* |
| ***9. @PreDestroy / destroy() (DisposableBean)*** | *Cleanup logic when context is shutting down.* |

*🟩* ***Real-Time Example: Online Banking App***

*java*

*CopyEdit*

*@Component*

*public class TransactionService implements InitializingBean, DisposableBean {*

*@PostConstruct*

*public void postConstructInit() {*

*System.out.println(">> Connection pool initialized");*

*}*

*@Override*

*public void afterPropertiesSet() {*

*System.out.println(">> Properties injected, bean ready");*

*}*

*@PreDestroy*

*public void preDestroy() {*

*System.out.println(">> Closing connections before shutdown");*

*}*

*@Override*

*public void destroy() {*

*System.out.println(">> Bean destroyed");*

*}*

*}*

*🟩* ***Tricky/Indirect Interview Questions****:*

* *What are all the hooks Spring provides for bean lifecycle customization?*
* *How is @PostConstruct different from afterPropertiesSet()?*
* *What is the role of BeanPostProcessor in the lifecycle?*
* *Can I use both @PreDestroy and DisposableBean? What happens?*

*🟩* ***Important Annotations & Interfaces****:*

| ***Element*** | ***Purpose*** | ***Notes*** |
| --- | --- | --- |
| *@PostConstruct* | *Runs after DI is done* | *Comes from javax.annotation or jakarta.annotation* |
| *@PreDestroy* | *Called before bean destruction* | *Used for resource cleanup* |
| *InitializingBean* | *afterPropertiesSet() method is called* | *Java-based lifecycle hook* |
| *DisposableBean* | *destroy() method* | *Triggered during context close* |
| *BeanPostProcessor* | *Customize bean before/after initialization* | *Framework-level customizations* |
| *@DependsOn("beanName")* | *Ensures dependent bean initializes first* | *Avoids dependency initialization issues* |

*🟩* ***Common Exceptions in Lifecycle****:*

| ***Exception*** | ***Reason*** |
| --- | --- |
| *BeanCurrentlyInCreationException* | *Circular dependencies* |
| *BeanInstantiationException* | *Abstract/interface bean defined* |
| *IllegalStateException* | *@PostConstruct on static method or wrong signature* |

*🟩* ***Deep Interview Probes****:*

* *What if I use both @PostConstruct and afterPropertiesSet()? Which runs first?  
  → @PostConstruct runs* ***before*** *afterPropertiesSet() — both can be used together but @PostConstruct is more declarative and preferred.*
* *What happens if I forget to close a DB connection in a @PreDestroy method?  
  → Resource leaks can occur, especially in non-container-managed environments.*
* *Can lifecycle annotations be applied to prototype beans?  
  → No. Spring does* ***not*** *manage full lifecycle for prototype beans — especially destroy phase.*

*🟩* ***Best Practices****:*

* *Use @PostConstruct and @PreDestroy for simplicity and readability.*
* *Prefer annotations over implementing lifecycle interfaces.*
* *Avoid business logic in lifecycle methods — limit to resource setup/cleanup.*

*✅* ***22) What is the difference between @Component, @Controller, @Service, and @Repository? When to use each?***

*🟩* ***Definition****:  
All four annotations — @Component, @Controller, @Service, and @Repository — are* ***stereotype annotations*** *in Spring. They mark classes as Spring-managed* ***beans****, making them eligible for component scanning and dependency injection.*

*But they* ***serve different roles*** *in the layered architecture and help in* ***semantic clarity*** *and* ***exception handling*** *in specific layers.*

*🟩* ***Layer-wise Usage and Purpose****:*

| ***Annotation*** | ***Used In*** | ***Purpose*** |
| --- | --- | --- |
| *@Component* | *Common layer* | *Generic stereotype for any Spring-managed bean* |
| *@Controller* | *Web layer* | *Marks a class as a Spring MVC controller (handles HTTP requests)* |
| *@Service* | *Business layer* | *Used for service classes containing business logic* |
| *@Repository* | *Persistence layer* | *Used for DAO classes that interact with DB (adds exception translation)* |

*🟩* ***Real-Time Example: Banking App (Money Transfer Module)***

*java*

*CopyEdit*

*@Component*

*public class SmsNotifier { }*

*@Controller*

*public class TransactionController {*

*@Autowired private TransactionService service;*

*}*

*@Service*

*public class TransactionService {*

*@Autowired private TransactionRepository repo;*

*}*

*@Repository*

*public class TransactionRepository {*

*public void saveTransaction(...) {*

*// Insert into DB*

*}*

*}*

*🟩* ***Key Differences – Internals & Behaviors****:*

| ***Aspect*** | ***@Component*** | ***@Service*** | ***@Repository*** | ***@Controller*** |
| --- | --- | --- | --- | --- |
| ***Role*** | *Generic bean* | *Business logic* | *Data access* | *Web controller* |
| ***Extra Functionality*** | *None* | *None* | *Translates JDBC exceptions into Spring's DataAccessException* | *Automatically registers return values as views or JSON* |
| ***Best Practice*** | *Base for custom annotations* | *Domain service logic* | *DAO or JPA repo* | *REST/web endpoint mappings* |

*🟩* ***Tricky/Indirect Interview Questions****:*

* *How does Spring know which bean to inject if all are @Component?*
* *Why does Spring provide multiple stereotypes instead of just one?*
* *What happens if I use @Service instead of @Repository in DAO classes?*
* *Does @Repository have any special behavior compared to @Component?*

*🟩* ***Important Attributes****:*

| ***Annotation*** | ***Key Attributes*** | ***Notes*** |
| --- | --- | --- |
| *All 4* | *value* | *(Optional) Custom bean name for the component* |
| *@Controller* | *Can be used with @ResponseBody or @RestController* | *Useful in REST APIs* |
| *@Repository* | *Enables automatic exception translation* | *Converts SQL exceptions into unchecked Spring exceptions* |

*🟩* ***Common Mistakes & Exceptions****:*

| ***Problem*** | ***Cause*** |
| --- | --- |
| *NoSuchBeanDefinitionException* | *Forgot to annotate or use wrong stereotype* |
| *BeanNotOfRequiredTypeException* | *Autowired bean type mismatch* |
| *BeanInstantiationException* | *Class is abstract or has no constructor* |

*🟩* ***Deep Interview Probes****:*

* *What if I annotate a service class with @Component instead of @Service?  
  → Technically, it will still work. But @Service conveys clear intent and supports* ***AOP auto-detection*** *for transaction or logging advice.*
* *Can @Repository work with exception translation without using Spring JDBC or JPA?  
  → No. Exception translation only works if you're using Spring-supported persistence layers like JDBC or JPA.*
* *Is @Controller same as @RestController?  
  → No. @RestController = @Controller + @ResponseBody — it returns JSON directly without view resolution.*

*🟩* ***Best Practices****:*

* *Always use the correct stereotype to reflect the layer of your application — improves* ***readability****,* ***debuggability****, and* ***tooling support****.*
* *Use @Repository for database classes to get Spring's exception translation.*
* *Avoid overusing @Component when a more specific annotation is semantically better.*

*✅* ***23) What are the different types of Dependency Injection (DI) in Spring? When should you use Constructor vs Setter Injection?***

*🟩* ***Definition of Dependency Injection (DI):*** *Dependency Injection is a core Spring concept where* ***objects are provided with their dependencies instead of creating them directly****. Spring manages the life cycle and injection of dependencies, making your code more modular, testable, and loosely coupled.*

*🟩* ***Types of Dependency Injection in Spring****:*

| ***Type*** | ***Description*** | ***Example*** |
| --- | --- | --- |
| ***Constructor Injection*** | *Dependencies are injected through the class constructor* | *Preferred for* ***mandatory*** *dependencies* |
| ***Setter Injection*** | *Dependencies are injected via public setters* | *Suitable for* ***optional*** *dependencies* |
| ***Field Injection*** *(not recommended)* | *Dependencies are injected directly into fields using @Autowired* | *Easy but poor for testing & design* |

*🟩* ***Code Examples***

***Constructor Injection:***

*java*

*CopyEdit*

*@Component*

*public class OrderService {*

*private final PaymentService paymentService;*

*@Autowired*

*public OrderService(PaymentService paymentService) {*

*this.paymentService = paymentService;*

*}*

*}*

***Setter Injection:***

*java*

*CopyEdit*

*@Component*

*public class OrderService {*

*private PaymentService paymentService;*

*@Autowired*

*public void setPaymentService(PaymentService paymentService) {*

*this.paymentService = paymentService;*

*}*

*}*

***Field Injection (Not Preferred):***

*java*

*CopyEdit*

*@Component*

*public class OrderService {*

*@Autowired*

*private PaymentService paymentService;*

*}*

*🟩* ***Tricky/Indirect Interview Questions****:*

* *Why would a team choose constructor injection over setter injection?*
* *How can Spring enforce immutability using DI?*
* *Is there any risk of memory leaks with field injection?*
* *Can we mix constructor and setter injection?*

*🟩* ***When to Use What?***

| ***Scenario*** | ***Preferred DI Type*** | ***Reason*** |
| --- | --- | --- |
| *Immutable object* | *Constructor* | *Final fields, thread-safe* |
| *Optional dependency* | *Setter* | *Can skip if not available* |
| *Unit testing ease* | *Constructor* | *Easy to mock* |
| *Readability & flexibility* | *Setter* | *Allows modification post-instantiation* |
| *Code quality & safety* | *Constructor* | *Makes dependencies explicit* |

*🟩* ***Common Exceptions****:*

| ***Exception*** | ***When It Happens*** |
| --- | --- |
| *UnsatisfiedDependencyException* | *No bean found for the dependency* |
| *NoSuchBeanDefinitionException* | *Missing @Component/@Bean or package not scanned* |
| *BeanCurrentlyInCreationException* | *Circular dependencies in constructor injection* |

*🟩* ***Annotations Involved****:*

| ***Annotation*** | ***Usage*** | ***Notes*** |
| --- | --- | --- |
| *@Autowired* | *Marks the injection point* | *Can be used on constructor, setter, or field* |
| *@Qualifier("beanName")* | *Specifies which bean to inject when multiple exist* | *Resolves ambiguity* |
| *@Primary* | *Marks preferred bean* | *Used when multiple candidates exist* |
| *@Inject* | *JSR-330 equivalent of @Autowired* | *Less powerful in Spring context* |

*🟩* ***Real-Time Use Case (E-Commerce App)****:*

* ***CartService****: Injects ProductService using constructor because it's mandatory for all operations.*
* ***CouponService****: Injected via setter since some items may not use coupons, so it's optional.*

*🟩* ***Deep Interview Drill-Down****:*

*Can constructor injection cause circular dependency? → Yes. Constructor injection fails with circular references. Spring throws BeanCurrentlyInCreationException. Setter injection can handle it since it allows delayed setting.*

*Which DI is better for unit testing and why?  
→ Constructor injection — because dependencies are passed explicitly and final, making the class easier to mock and test.*

*Does Spring require @Autowired on constructor?  
→ No,* ***from Spring 4.3 onwards****, if there's only one constructor, @Autowired is optional.*

*🟩* ***Best Practices****:*

* *Prefer* ***constructor injection*** *for mandatory, final dependencies.*
* *Use* ***setter injection*** *only for optional values or legacy frameworks.*
* *Avoid field injection for better unit testing, readability, and maintainability.*

*✅* ***24) What are the different bean scopes in Spring? When and why would you use each scope?***

*🟩* ***Definition of Bean Scope****:  
A* ***bean scope*** *in Spring defines* ***how many instances*** *of a bean Spring should create and* ***how long they should live*** *within the container. Scopes control the lifecycle and visibility of Spring beans.*

*🟩* ***Available Bean Scopes in Spring (Core & Web)***

| ***Scope*** | ***Description*** | ***Where Used*** | ***Lifecycle*** |
| --- | --- | --- | --- |
| ***singleton*** | ***Default scope*** *– single instance per Spring container* | *Core & Web apps* | *Created at container startup, shared* |
| ***prototype*** | *New bean instance every time it's requested* | *Core apps* | *Created on-demand, not shared* |
| ***request*** | *One bean per HTTP request* | *Web apps only* | *Created for each HTTP request* |
| ***session*** | *One bean per HTTP session* | *Web apps only* | *Tied to the user session* |
| ***application*** | *One bean per ServletContext (web app)* | *Web apps only* | *Lives as long as app context* |
| ***websocket*** | *One bean per WebSocket session* | *WebSocket apps* | *Tied to a WebSocket connection* |

*🟩* ***Code Example*** *– Singleton vs Prototype*

*java*

*CopyEdit*

*@Component*

*@Scope("singleton")*

*public class EmailService {*

*// One shared instance*

*}*

*@Component*

*@Scope("prototype")*

*public class TaskRunner {*

*// New instance on each request*

*}*

*🟩* ***How to Declare Scope****: Use @Scope("scopeName") on a @Component or @Bean.*

*java*

*CopyEdit*

*@Bean*

*@Scope(ConfigurableBeanFactory.SCOPE\_PROTOTYPE)*

*public PaymentProcessor paymentProcessor() {*

*return new PaymentProcessor();*

*}*

*🟩* ***Real-Time Use Cases***

| ***Bean*** | ***Scope*** | ***Why?*** |
| --- | --- | --- |
| *UserService* | *singleton* | *Stateless service shared across requests* |
| *CartBean* | *session* | *Each user should have their own shopping cart* |
| *RequestLogger* | *request* | *Each HTTP request should be tracked separately* |
| *EmailNotificationJob* | *prototype* | *A new task each time it runs (stateful thread)* |

*🟩* ***Tricky/Indirect Interview Questions****:*

* *Why is singleton the default scope in Spring?*
* *When would prototype beans fail to behave as expected?*
* *Can I inject a prototype-scoped bean into a singleton bean?*
* *How does Spring handle session scope in REST APIs?*

*🟩* ***Common Exceptions & Issues***

| ***Issue*** | ***Cause*** |
| --- | --- |
| ***ScopeNotActiveException*** | *Using request/session scope outside web context (e.g., CLI or non-servlet)* |
| ***Prototype bean behaving like singleton*** | *Injected into singleton without proxy mode* |
| ***Memory leaks*** | *Improper use of session/request beans without proper invalidation* |

*🟩* ***Deep Interview Insights****:*

*What happens if you inject a prototype-scoped bean into a singleton bean?  
→ Only* ***one instance*** *of the prototype bean is created and injected during singleton’s creation.  
To solve this, use* ***ObjectFactory*** *or* ***@Lookup*** *method injection.*

*java*

*CopyEdit*

*@Component*

*public class SingletonBean {*

*@Autowired*

*private ObjectFactory<PrototypeBean> prototypeFactory;*

*public void usePrototype() {*

*PrototypeBean pb = prototypeFactory.getObject();*

*pb.execute();*

*}*

*}*

*Is scope relevant in multi-threaded apps?  
→ Yes. Stateless services should be singleton. Stateful or thread-sensitive beans should be prototype or custom scoped.*

*Can we define custom scopes?  
→ Yes. Spring allows creation of* ***custom scopes*** *by implementing Scope interface (e.g., for batch jobs, per-thread logic).*

*🟩* ***Annotations & Attributes Involved***

| ***Annotation*** | ***Purpose*** | ***Notes*** |
| --- | --- | --- |
| *@Scope("...")* | *Declares scope* | *Can be used on @Component or @Bean* |
| *@RequestScope, @SessionScope, @ApplicationScope* | *Shortcuts for web-specific scopes* | *Added in Spring 4.3+* |
| *@Lookup* | *Method injection to dynamically resolve scoped beans* | *Avoids singleton-prototype issues* |
| *@Component* | *Registers the class as Spring bean* | *Use with scope annotation* |

*🟩* ***Best Practices****:*

* *Use* ***singleton*** *for most services.*
* *Use* ***prototype*** *for stateful or short-lived tasks.*
* *Be cautious with request/session scope outside a web context.*
* *Avoid injecting prototype into singleton directly – use method injection or proxies.*
* ***Avoid unnecessary scope customizations*** *unless required for complex needs.*

*✅* ***25) What is Bean Lifecycle in Spring? How do you manage custom initialization and destruction logic?***

*🟩* ***Definition****:  
The* ***Bean Lifecycle*** *in Spring refers to the* ***series of steps*** *a bean goes through from* ***instantiation to destruction*** *within the Spring container. Spring provides several hooks where you can insert* ***custom logic*** *during these phases.*

*🟩* ***Bean Lifecycle Phases in Order****:*

1. ***Instantiation*** *– Bean is created using constructor or factory method.*
2. ***Populate Properties*** *– Dependencies injected via setter or constructor.*
3. ***BeanNameAware.setBeanName()***
4. ***BeanFactoryAware.setBeanFactory()***
5. ***ApplicationContextAware.setApplicationContext()***
6. ***Pre-initialization (BeanPostProcessor#postProcessBeforeInitialization)***
7. ***InitializingBean.afterPropertiesSet()*** *or method annotated with @PostConstruct*
8. ***Custom init method (configured via init-method)***
9. ***Post-initialization (BeanPostProcessor#postProcessAfterInitialization)***
10. ***Bean is ready to use***
11. ***Destruction Phase*** *(when container shuts down):*
    * *@PreDestroy*
    * *DisposableBean.destroy()*
    * *Custom destroy method via destroy-method*

*🟩* ***Code Example****:*

*java*

*CopyEdit*

*@Component*

*public class ReportService implements InitializingBean, DisposableBean {*

*@PostConstruct*

*public void init() {*

*System.out.println("Init with @PostConstruct");*

*}*

*@Override*

*public void afterPropertiesSet() {*

*System.out.println("Init via InitializingBean");*

*}*

*@PreDestroy*

*public void cleanup() {*

*System.out.println("Cleanup with @PreDestroy");*

*}*

*@Override*

*public void destroy() {*

*System.out.println("Cleanup via DisposableBean");*

*}*

*}*

*🟩* ***Real-Time Use Case****:*

| ***Scenario*** | ***Use*** |
| --- | --- |
| ***Logging setup*** | *Inside @PostConstruct* |
| ***Starting async schedulers*** | *In init method* |
| ***Releasing file/db connections*** | *In destroy method or @PreDestroy* |
| ***Stopping threads or timers*** | *In destroy() or @PreDestroy* |

*🟩* ***Tricky/Indirect Interview Questions****:*

* *What happens if both @PostConstruct and afterPropertiesSet() are used?*
* *Can bean lifecycle methods throw exceptions?*
* *How does Spring manage lifecycle of prototype beans?*
* *What’s the difference between @PreDestroy and DisposableBean.destroy()?*

*🟩* ***Common Exceptions & Gotchas****:*

| ***Issue*** | ***Cause*** |
| --- | --- |
| *BeanCurrentlyInCreationException* | *Circular dependency during instantiation* |
| *IllegalStateException* | *Method annotated with @PostConstruct is private or has args* |
| *Destroy method not called* | *Happens with prototype beans, as Spring doesn’t manage their full lifecycle* |

*🟩* ***Important Annotations & Interfaces***

| ***Name*** | ***Purpose*** | ***Notes*** |
| --- | --- | --- |
| *@PostConstruct* | *Marks a method to run after dependency injection* | *From javax.annotation.\** |
| *@PreDestroy* | *Marks method to run before bean destruction* | *Spring calls it during shutdown* |
| *InitializingBean* | *Provides afterPropertiesSet()* | *Alternative to @PostConstruct* |
| *DisposableBean* | *Provides destroy()* | *Alternative to @PreDestroy* |
| *BeanPostProcessor* | *Allows custom logic before/after initialization* | *Used for AOP, proxies* |
| *SmartInitializingSingleton* | *Runs logic after all singletons are created* | *Useful for auto-wiring all beans of a type* |

*🟩* ***How Spring Handles Lifecycle for Singleton vs Prototype Beans****:*

| ***Bean Scope*** | ***Who manages lifecycle?*** | ***Destruction called?*** |
| --- | --- | --- |
| ***Singleton*** | *Spring Container* | *✅ Yes* |
| ***Prototype*** | *User manages destruction* | *❌ No automatic cleanup* |

*🟩* ***Advanced Interview Hook****:*

*“How would you ensure cleanup logic runs for prototype beans?”*

*You must manage them manually. Example:*

*java*

*CopyEdit*

*AnnotationConfigApplicationContext context = new AnnotationConfigApplicationContext(AppConfig.class);*

*MyPrototypeBean bean = context.getBean(MyPrototypeBean.class);*

*// use the bean*

*bean.cleanup(); // manually call destroy logic*

*🟩* ***Best Practices****:*

* *Use @PostConstruct and @PreDestroy for modern Spring apps.*
* *Avoid using InitializingBean and DisposableBean unless needed for backward compatibility.*
* *Be aware that @PreDestroy is* ***not called*** *for prototype-scoped beans.*
* *Register* ***shutdown hooks*** *for graceful termination: context.registerShutdownHook();*
* *For large lifecycle control, use BeanPostProcessor or SmartLifecycle.*

*✅* ***26) What is Dependency Injection (DI) in Spring? Types of DI? Which one is preferred and why?***

*🟩* ***Definition****:****Dependency Injection (DI)*** *is a design pattern where objects (dependencies) are provided to a class by an external source instead of the class creating them itself. In Spring, the* ***container manages object creation and dependency wiring****, reducing tight coupling and making the application more modular and testable.*

*🟩* ***Why use DI in Spring?***

* *Promotes* ***loose coupling*** *between classes*
* *Makes code* ***easier to test****, maintain, and scale*
* *Supports the* ***Hollywood Principle****: "Don’t call us, we’ll call you"*

*🟩* ***Types of Dependency Injection in Spring****:*

| ***Type*** | ***Description*** | ***Example*** |
| --- | --- | --- |
| ***Constructor Injection*** | *Dependencies are injected via class constructor* | *✅ Immutable, preferred* |
| ***Setter Injection*** | *Dependencies are injected via public setter methods* | *Used when optional dependencies exist* |
| ***Field Injection*** | *Dependencies are injected directly into fields using reflection* | *❌ Discouraged for testability and immutability* |

***✅ 1) Constructor Injection (Preferred)***

*java*

*CopyEdit*

*@Component*

*public class OrderService {*

*private final PaymentService paymentService;*

*@Autowired*

*public OrderService(PaymentService paymentService) {*

*this.paymentService = paymentService;*

*}*

*}*

*🟩* ***Advantages****:*

* *Promotes* ***immutability***
* *Easier for* ***unit testing*** *(no need for setters)*
* ***Fails fast*** *– If dependency is missing, Spring throws error during bean creation*

***✅ 2) Setter Injection***

*java*

*CopyEdit*

*@Component*

*public class OrderService {*

*private PaymentService paymentService;*

*@Autowired*

*public void setPaymentService(PaymentService paymentService) {*

*this.paymentService = paymentService;*

*}*

*}*

*🟩* ***Use Case****:*

* *When a dependency is* ***optional***
* *When there is* ***circular dependency*** *(constructor injection may fail)*

***✅ 3) Field Injection (Discouraged)***

*java*

*CopyEdit*

*@Component*

*public class OrderService {*

*@Autowired*

*private PaymentService paymentService;*

*}*

*🟩* ***Why not recommended?***

* *Breaks* ***encapsulation***
* ***Hard to test*** *using mock frameworks*
* *Difficult to identify actual dependencies*

*🟩* ***Tricky/Indirect Interview Questions****:*

* *Why is constructor injection preferred in Spring Boot?*
* *Can you combine setter and constructor injection?*
* *What is the issue with field injection?*
* *How does DI differ from Service Locator pattern?*

*🟩* ***Real-Time Use Case Example****:*

*java*

*CopyEdit*

*@Service*

*public class NotificationService {*

*private final EmailService emailService;*

*@Autowired*

*public NotificationService(EmailService emailService) {*

*this.emailService = emailService;*

*}*

*public void send() {*

*emailService.sendEmail();*

*}*

*}*

*🟩* ***In a Banking Application****:*

* *AccountService depends on TransactionRepository*
* *Spring injects the repository dependency at runtime*
* *Makes services mockable for testing with JUnit or Mockito*

*🟩* ***Common Exceptions & Pitfalls****:*

| ***Exception*** | ***Cause*** |
| --- | --- |
| *UnsatisfiedDependencyException* | *Spring can’t find a matching bean for injection* |
| *NoSuchBeanDefinitionException* | *No bean found for autowiring* |
| *BeanCurrentlyInCreationException* | *Circular dependency with constructor injection* |
| *IllegalArgumentException* | *Null passed to constructor because no matching bean was found* |

*🟩* ***Relevant Annotations****:*

| ***Annotation*** | ***Use*** | ***Notes*** |
| --- | --- | --- |
| *@Autowired* | *Injects bean by type* | *Can be used on constructor, setter, or field* |
| *@Qualifier("beanName")* | *Disambiguates when multiple beans of same type exist* | *Must match exact bean name* |
| *@Inject* | *JSR-330 equivalent of @Autowired* | *No required attribute* |
| *@Value("${config.value}")* | *Injects values from properties file* | *For configuration-based DI* |
| *@Primary* | *Gives priority when multiple candidates exist* | *Alternative to using @Qualifier* |

*🟩* ***Best Practices****:*

* *Use* ***constructor injection*** *for required dependencies.*
* *Use* ***setter injection*** *for optional ones.*
* *Avoid field injection unless using* ***Lombok’s @RequiredArgsConstructor*** *with final fields.*
* *Use @Qualifier carefully when multiple beans exist.*
* *For large applications, prefer* ***Spring Profiles*** *for environment-specific injection.*

*🟩* ***Advanced Interview Hook****:*

*“How does Spring handle circular dependencies in constructor vs setter injection?”*

* ***Constructor injection*** *causes failure (BeanCurrentlyInCreationException)*
* ***Setter injection*** *works, as Spring creates bean references first and sets them later*

*✅* ***18) What is Wiring in Spring Framework?***

*🟩* ***Definition****:****Wiring*** *in Spring refers to the process of connecting beans together by injecting dependencies. When one bean depends on another, Spring "wires" them using* ***Dependency Injection (DI)*** *either through constructor, setter, or field.*

*🟩* ***Real-Time Analogy****:  
Imagine assembling a computer. The CPU (one bean) needs a RAM (another bean). Wiring is like plugging RAM into the motherboard — the connection (injection) is essential for functioning.*

*🟩* ***Example*** *(XML-Based Setter Wiring):*

*xml*

*CopyEdit*

*<bean id="employee" class="com.example.Employee">*

*<property name="address" ref="address"/>*

*</bean>*

*<bean id="address" class="com.example.Address"/>*

*Here, Spring wires the address bean into the employee bean using the setter.*

*🟩* ***Tricky/Indirect Questions****:*

* *How do beans in Spring interact with each other?*
* *What is the difference between bean reference and bean definition?*
* *Can a single bean be wired into multiple beans?*

*🟩* ***Common Exceptions****:*

* *NoSuchBeanDefinitionException: The reference bean doesn’t exist in the context.*
* *BeanNotOfRequiredTypeException: Bean defined is of a different type than expected.*

*🟩* ***Annotations for Wiring****:*

* *@Autowired: Automatically wires matching type bean.*
* *@Qualifier("beanName"): Helps disambiguate when multiple beans exist.*
* *@Inject: Java standard alternative to @Autowired.*

*🟩* ***Deep-Dive Interview Triggers****:*

* *What happens if two beans of same type exist?*
* *Can you wire a bean from another Spring config file?*
* *How is wiring different in Java config and annotations?*

*✅* ***19) What is Autowiring in Spring and Why Do We Use It?***

*🟩* ***Definition****:****Autowiring*** *is a feature that enables Spring to automatically resolve and inject collaborating beans by examining their type, name, or constructor signature without explicitly defining <property> or <constructor-arg>.*

*🟩* ***Why Autowiring?*** *Without autowiring, you need to manually wire all beans via XML or annotations. As the app grows, this becomes error-prone. Autowiring automates this process and reduces boilerplate code.*

*🟩* ***Example (Annotation-Based)****:*

*java*

*CopyEdit*

*@Component*

*public class Vehicle {*

*@Autowired*

*private Engine engine;*

*}*

*Spring automatically injects a matching Engine bean by type.*

*🟩* ***Autowiring Modes****:*

1. ***no (default)*** *– Manual wiring*
2. ***byName*** *– Injects by bean name*
3. ***byType*** *– Injects by type; must be unique*
4. ***constructor*** *– Uses constructor for DI (type-first, name-second)*
5. ***autodetect*** *– (Deprecated) Automatically chooses constructor/setter*

*🟩* ***Tricky/Indirect Questions****:*

* *What happens if multiple beans match the autowire criteria?*
* *How is autowiring resolved in Java vs XML config?*
* *Can we autowire collections?*

*🟩* ***Annotations in Autowiring****:*

* *@Autowired(required = true/false): Optional injection*
* *@Qualifier("beanName"): Resolves ambiguity*
* *@Primary: Default preferred bean when multiple matches*

*🟩* ***Common Exceptions****:*

* *NoUniqueBeanDefinitionException: More than one candidate found*
* *UnsatisfiedDependencyException: Dependency not found or failed injection*

*🟩* ***Real-Time Use Case****: In a banking app, @Autowired is used to wire NotificationService into TransactionService so that it automatically sends alerts after transactions.*

*🟩* ***How Interviewers May Dig Deeper****:*

* *What are the alternatives to @Autowired?*
* *Can autowiring be disabled?*
* *Difference between autowiring a field and a constructor?*

*✅* ***21) What Are the Limitations of Autowiring?***

*🟩* ***Limitations****:*

| ***Limitation*** | ***Explanation*** |
| --- | --- |
| *❌ Can’t autowire primitive types* | *You cannot autowire int, boolean, double directly.* |
| *✅ String is autowirable (special case)* | *Spring treats String as a valid bean if available or via SpEL.* |
| *❌ No clear behavior when multiple beans exist* | *If there are multiple candidates and no @Qualifier, exception occurs.* |
| *❌ Fails silently if @Autowired(required = false)* | *May lead to NullPointerException if not handled properly.* |
| *❌ Difficult to trace errors in large projects* | *Because wiring is implicit, tracking what gets injected becomes tough.* |

*🟩* ***Tricky Questions****:*

* *How would you inject a double using Spring?*
* *What happens if @Autowired bean is not available?*
* *Can you autowire values from properties file?*

*🟩* ***Real-Time Handling****: To inject a primitive:*

*xml*

*CopyEdit*

*<property name="taxRate" value="5.0"/>*

*Or use @Value("${tax.rate}") with property file.*

*✅* ***22) Explain Spring Bean Life Cycle***

*🟩* ***Definition****:  
The* ***Bean Life Cycle*** *refers to the steps a bean follows from creation to destruction inside the Spring container. Spring gives full control to hook into this lifecycle using interfaces, annotations, or XML.*

*🟩* ***Phases of Spring Bean Life Cycle****:*

1. ***Instantiation*** *– Bean is created.*
2. ***Populate Properties*** *– Dependencies are injected.*
3. ***Aware Interfaces*** *(optional) – Like BeanNameAware, ApplicationContextAware*
4. ***Pre-initialization*** *– Custom logic before init*
5. ***Initialization*** *– afterPropertiesSet() or custom init-method*
6. ***Post Processors*** *– BeanPostProcessors (like AOP)*
7. ***Ready to Use***
8. ***Destroy*** *– On container shutdown using destroy-method or DisposableBean*

*🟩* ***Real-Time Hook Usage****:*

*java*

*CopyEdit*

*@PostConstruct*

*public void init() {*

*// load resources, cache data*

*}*

*@PreDestroy*

*public void cleanup() {*

*// release resources*

*}*

*🟩* ***Tricky Questions****:*

* *When does @PostConstruct get called?*
* *What happens if destroy method is not defined?*
* *Can a bean skip init method but use post-processors?*

*🟩* ***Common Exceptions****:*

* *BeanCurrentlyInCreationException: Circular reference*
* *BeanInitializationException: Fails during init method*

*🟩* ***Annotations****:*

* *@PostConstruct – Called after DI*
* *@PreDestroy – Called before bean destruction*
* *@Scope("prototype") – To change bean lifecycle scope*

*🟩* ***Interview Follow-ups****:*

* *Can we override init() if lifecycle interface is used?*
* *How is bean lifecycle different in prototype scope?*
* *How does Spring handle lifecycle of @Component?*

*✅* ***23) Why and How to Create Custom Init and Destroy Methods in Spring?***

*🟩* ***Why Custom Init/Destroy?*** *Spring allows developers to hook* ***custom initialization and destruction logic*** *to manage resources such as file loading, connection pooling, or cleanup. This is useful when you want to perform actions after all dependencies are set or before the bean is removed.*

*🟩* ***How to Define Custom Init/Destroy?***

***A. Using XML Configuration:***

*xml*

*CopyEdit*

*<bean id="reportService" class="com.app.ReportService"*

*init-method="initReports" destroy-method="cleanupResources"/>*

***B. Using @PostConstruct and @PreDestroy:***

*java*

*CopyEdit*

*@Component*

*public class ReportService {*

*@PostConstruct*

*public void initReports() {*

*// load report templates, data*

*}*

*@PreDestroy*

*public void cleanupResources() {*

*// close connections, clean memory*

*}*

*}*

***C. Implementing Interfaces:***

*java*

*CopyEdit*

*public class MyBean implements InitializingBean, DisposableBean {*

*public void afterPropertiesSet() {*

*// init logic*

*}*

*public void destroy() {*

*// cleanup logic*

*}*

*}*

*🟩* ***Tricky/Indirect Interview Questions****:*

* *How is @PostConstruct different from afterPropertiesSet()?*
* *Can we define multiple init methods?*
* *What if the destroy method throws an exception?*

*🟩* ***Common Exceptions****:*

* *BeanInitializationException: If init-method fails.*
* *IllegalStateException: If destroy method accesses already-destroyed resources.*

*🟩* ***Real-Time Use Case****: In a stock trading application, you may use custom init to preload configuration files for stock symbols and use destroy to close Kafka or MQ connections before shutdown.*

*🟩* ***Deep Interview Probes****:*

* *Which approach is preferred — annotation, interface, or XML?*
* *Are these lifecycle methods called in prototype scope?*
* *Can you suppress destroy-method execution?*

*✅* ***24) Servlet vs Spring Bean Lifecycle***

*🟩* ***Key Differences****:*

| ***Aspect*** | ***Servlet Lifecycle*** | ***Spring Bean Lifecycle*** |
| --- | --- | --- |
| ***Managed By*** | *Web Container (Tomcat, Jetty)* | *Spring Container* |
| ***Creation*** | *init() called once when server starts* | *Bean instantiated when container starts or requested* |
| ***Execution*** | *service() method handles requests* | *Business methods are called as per app need* |
| ***Destruction*** | *destroy() called during server shutdown* | *destroy-method or @PreDestroy executed* |
| ***Scope*** | *Generally singleton per servlet* | *Supports Singleton, Prototype, Request, Session* |

*🟩* ***Servlet Lifecycle Methods****:*

*java*

*CopyEdit*

*public class MyServlet extends HttpServlet {*

*public void init() { } // Called once*

*public void service() { } // Called per request*

*public void destroy() { } // Called on shutdown*

*}*

*🟩* ***Spring Bean Lifecycle Summary****:*

* ***init phase****: After dependency injection*
* ***service methods****: Business logic*
* ***destroy phase****: Before bean is garbage collected or container shuts down*

*🟩* ***Tricky Interview Questions****:*

* *Can a Spring bean be injected into a Servlet?*
* *How would you explain bean lifecycle to a junior developer?*
* *Which one offers better control over object scope?*

*🟩* ***Real-Time Comparison****: In a banking web app:*

* *Servlet handles initial request mapping.*
* *Spring beans (@Service, @Repository) handle business logic and database operations. This separation enables better testing and decoupling.*

*🟩* ***Extra Notes****:*

* *You can wire Spring beans into Servlets using WebApplicationContextUtils.*
* *Servlets are limited to request-based lifecycle; Spring beans are much more flexible.*