Spring Core

1. **What are the advantages of using DI in application development?**

   Answer

   Advantages of using DI in application development

- Easier to manage dependencies
  Instead of creating objects manually inside a class, DI provides those objects from outside. This helps me avoid writing extra code for creating or configuring dependencies.
- Improves code readability and organization
  DI helps me write cleaner and more modular code. I can easily understand which class depends on what, just by looking at the constructor or fields.
- Better testing (especially unit testing)
  It's easier to write test cases because I can inject mock objects into the class. I don't have to rely on real database connections or services during testing.
- Encourages loose coupling
  DI makes classes depend on interfaces rather than concrete implementations. This makes my code flexible and easier to change or extend later.
- Easier to scale and maintain the project
  When the project grows, DI helps me manage complexity better. I can plug and play new components without breaking existing ones.
- Supports reuse of components
  I can easily reuse services or logic in different parts of my application because they're injected and not tightly bound.

2. What are the core modules of the Spring Framework? Can you briefly explain each?

Answer
The Spring Framework is made up of several modules that help us build powerful, flexible Java applications. Here are the main ones:

1. Core Container Module

- **Includes:** Core, Beans, Context, and Expression Language (SpEL)
- **Purpose:** Provides the basic functionality of the Spring Framework like **Dependency Injection (DI)** and **bean lifecycle management**.
- **Example:** When we annotate a class with `@Component` or use `@Autowired`, we're using this module.

2. Spring AOP (Aspect-Oriented Programming)

- **Purpose:** Allows us to separate **cross-cutting concerns** like logging, security, or transactions from business logic.
- **Example:** Logging method calls without writing the logging code in each method.

3. Spring Data Access / Integration

This includes several sub-modules:

- **JDBC Module:** Simplifies database access using **JDBC** by removing boilerplate code.
- **ORM Module:** Supports integration with **Hibernate**, **JPA**, and other ORM tools.
- **JMS Module:** Supports **Java Messaging Service** for asynchronous communication.
- **Transactions Module:** Manages declarative and programmatic **transactions** easily.

4. Spring Web Module

- **Purpose:** Supports web-based applications using **Servlets** and traditional **web MVC architecture**.
- **Includes:** Spring MVC — helps in building web applications using controllers, models, and views.
- **Example:** Creating RESTful web services with `@RestController` and `@RequestMapping`.

5. Spring WebFlux (Reactive Web)

- **Purpose:** Used to build **non-blocking**, **reactive** applications using the **Reactive Streams API**.
- **Example:** Handling a large number of requests efficiently using Mono and `Flux`.

6. Spring Test Module

- **Purpose:** Provides support for **unit testing** and **integration testing** of Spring components using **JUnit** and **Mockito**.
- **Example:** Using `@SpringBootTest` or `@MockBean` in test classes.

3. What is the role of the ApplicationContext in Spring? How is it different from BeanFactory?

Answer

In Spring, the `ApplicationContext` plays the role of the central interface that manages the complete lifecycle of beans and provides advanced features such as dependency injection, internationalization, event propagation, and application-layer context management. It is a more powerful and feature-rich container compared to `BeanFactory`, which is the basic container primarily used for simple dependency injection. While `BeanFactory` creates beans lazily (only when needed), `ApplicationContext` creates beans eagerly at startup, unless configured otherwise. Additionally, `ApplicationContext` supports automatic registration of BeanPostProcessors, message sources for i18n, and application event listeners, which are not supported by `BeanFactory`. Therefore, `ApplicationContext` is preferred for most modern Spring applications due to its

flexibility and rich capabilities.

4. What is Spring Bean? How is it different from a regular Java object?

Answer
A **Spring Bean** is simply a Java object that is **managed by the Spring container**. When we define a class and ask Spring to create and manage its objects (using annotations like `@Component`, `@Service`, or by declaring it in a configuration file), that object becomes a Spring Bean. The container takes care of its **lifecycle, dependencies, and configuration**.

The main difference between a **Spring Bean** and a **regular Java object** is **who manages it**. A regular Java object is created manually using the `new` keyword and managed by the developer. In contrast, a Spring Bean is created by the Spring container and managed automatically — meaning Spring handles things like dependency injection, lifecycle callbacks, and even scopes (like singleton or prototype). So while all Spring Beans are Java objects, not all Java objects are Spring Beans — only those that are registered and managed by the Spring framework.

5. How does Spring achieve loose coupling between components?

Answer
Spring achieves **loose coupling** between components primarily through **Dependency Injection (DI)** and **interface-based programming**.

Instead of creating objects directly inside a class (which leads to tight coupling), Spring allows us to **inject dependencies from the outside**. This means that a class doesn't need to know how its dependencies are created — it just uses them. For example, if a service depends on a repository, Spring will automatically inject the required repository object into the service using annotations like `@Autowired`, or through constructors/setters.

Additionally, Spring encourages the use of **interfaces** so that components depend on abstractions, not concrete implementations. This makes it easy to swap implementations without changing the dependent class. Because of these features, Spring applications are easier to maintain, test, and extend — which is the essence of loose coupling.

6. How is configuration handled in Spring? What are the different ways to define beans?

Answer
In Spring, **configuration** is the process of telling the Spring container **what beans to create and how to wire them together**. Spring provides multiple ways to handle configuration, giving developers flexibility based on their project needs.

Configuration in Spring can be done in three main ways:

1. XML-based Configuration

- Beans are defined in an XML file (like `beans.xml`).
- You specify the class, properties, and dependencies using XML tags.
- Useful for older projects or when configuration needs to be externalized.

```
<bean id="myService" class="com.example.MyService">
   <property name="myRepository" ref="myRepo" />
</bean>
```

2. Annotation-based Configuration

- Uses annotations like `@Component`, `@Service`, `@Repository`, and `@Autowired` directly in Java classes.
- Requires `@ComponentScan` and `@Configuration` in the config class.
- Reduces boilerplate and keeps config close to code.

```
@Component
public class MyService {
   @Autowired
   private MyRepository myRepository;
}
```

3. Java-based Configuration (using @Configuration and @Bean)

- Uses pure Java classes to define beans with `@Bean` methods.
- Gives full control with type safety and IDE support.

```
@Configuration
public class AppConfig {

   @Bean
   public MyService myService() {
      return new MyService(myRepository());
   }

   @Bean
   public MyRepository myRepository() {
      return new MyRepository();
   }
```

```
}
```

7. What is the difference between @Component, @Service, @Repository, and @Controller annotations?

Answer

These four annotations — `@Component`, `@Service`, `@Repository`, and `@Controller` — are all part of **Spring's stereotype annotations**, used for **automatic bean detection and registration** during component scanning. While they all behave similarly in terms of being registered as Spring Beans, each one has a **specific semantic role** in a typical layered application.

@Component

- It is the **generic stereotype annotation**.
- Used to define any Spring-managed component or bean.
- Best for utility classes or components that don't fall into service/repository/controller layers.

```
@Component
public class EmailValidator {
    // General-purpose component
}
```

@Service

- A specialization of `@Component`.
- Used to annotate **service layer classes**, which hold business logic.
- Helps with better code organization and clarity.

```
@Service
public class PaymentService {
    // Business logic here
}
```

@Repository

- A specialization of `@Component` for the **data access layer**.
- Spring also adds **exception translation** (converts DB-related exceptions into Spring's DataAccessException).

```
@Repository
public class UserRepository {
    // Database access logic
}
```

@Controller

- A specialization of `@Component` used for **web layer (MVC)**.

- Marks a class as a controller that handles **HTTP requests** (used with Spring MVC).
- Typically works with annotations like `@RequestMapping`.

```
@Controller
public class UserController {
    @GetMapping("/users")
    public String getUsers() {
        return "userList";
    }
}
```

8. What is a POJO and why is it significant in Spring?

Answer
A **POJO** (Plain Old Java Object) is a **simple Java class** that is **not bound by any special restrictions** — it doesn't extend specific classes, implement special interfaces, or require annotations. It just contains fields, constructors, getters, setters, and normal methods.

Example of a POJO:

```
public class Student {
    private String name;
    private int age;

    public Student() {}

    public Student(String name, int age) {
        this.name = name;
        this.age = age;
    }

    // Getters and setters
}
```

Why is it significant in Spring?

- **Lightweight and simple:** Spring promotes using POJOs to keep the application code **clean and easy to understand**.
- **No tight coupling to framework:** POJOs do not need to inherit from Spring classes or use special APIs, which makes the code **more reusable and easier to test**.
- **Supports dependency injection:** Spring can manage and inject POJOs as beans using DI without requiring any changes to the POJO.
- **Easy testing and maintenance:** Since POJOs are simple objects, they can be easily tested using unit tests, even outside the Spring context.

9. What is annotation-based configuration in Spring and how is it better than XML configuration?

Answer

Annotation-based configuration in Spring is a modern way of configuring your application by using Java annotations directly in the source code, instead of writing separate XML files. With annotations like @Component, @Autowired, @Configuration, @Bean, and @ComponentScan, you can define beans, inject dependencies, and manage configurations right in your Java classes. This makes the application more concise and easier to understand.

Compared to XML configuration, annotation-based configuration is better because it's less verbose, type-safe, and easier to maintain. It keeps configuration closer to the code, which improves readability and reduces context switching between XML and Java. It also offers better IDE support like auto-completion and error checking. While XML is still supported (mainly in legacy systems), annotation-based configuration is the preferred approach in modern Spring development, especially with Spring Boot.

10. What is the difference between @Configuration and @Component? Why is it important?

Answer

Both @Configuration and @Component are used to register classes as Spring beans, but they serve different purposes and are used in different contexts.

@Component

Marks a general-purpose class as a Spring-managed bean.

Used on regular classes like services, repositories, or utilities.

Spring detects it during component scanning.

```
@Component
public class MyService {
   // Business logic
}
```

@Configuration

A specialized version of @Component.

Indicates that the class contains bean definitions using @Bean methods.

Often used for Java-based configuration instead of XML.

Spring ensures that each @Bean method in a @Configuration class returns a singleton, even if called multiple times.

```
@Configuration
public class AppConfig {

   @Bean
   public MyService myService() {
      return new MyService();
   }
}
```

11. If I use `@Component` without enabling component scanning, will my class be available as a bean?

No, it won't. Using `@Component` by itself is not enough. For Spring to detect and register that class as a bean, component scanning must be explicitly enabled—either using Java configuration with `@ComponentScan` or XML configuration with `<context:component-scan>`. Without this, Spring won't even look into the package to discover annotated components, so the class won't be registered in the application context. The only exception would be if you define the same bean manually in XML, which can override the need for scanning, but that defeats the purpose of using `@Component`. It's also important to ensure that the base package provided in the component scan is correct, otherwise your component might still be missed.

12. You define a bean in XML and annotate the same class with `@Component`. Which one will Spring use?

When both XML configuration and annotation-based configuration define the same bean, Spring gives priority to the XML-defined bean. Essentially, the XML configuration overrides the annotated definition. This can lead to some confusion if both definitions are active, which is why it's generally recommended to stick to a single configuration strategy per class. Mixing both methods in the same application can make it harder to debug or maintain. If there's ever a need to switch between the two, conditional configuration or profiles can help manage that more cleanly.

13. Is it possible for two `ApplicationContext` instances to manage the same bean instance?

Technically, yes—it is possible, but it's not recommended. Each `ApplicationContext` in Spring is designed to be independent and manages its own set of beans. If you try to share a single bean instance across two contexts manually, you might run into synchronization issues, memory leaks, or unexpected behavior due to lifecycle mismatches. A better approach, if shared configuration is truly needed, is to use a parent-child context hierarchy, where the parent context can hold common beans that children can access. But in most real-world applications, it's best to stick with a single context unless you have a strong modularization requirement.

14. How does Spring IoC handle exceptions during bean instantiation?

If an exception occurs during bean instantiation, Spring throws a `BeanCreationException`. This typically happens at startup when the application context is being initialized. Common reasons include missing dependencies, incorrect constructor parameters, or errors in lifecycle methods like `@PostConstruct`. When such an exception occurs, Spring halts the context initialization, and the application won't start unless the issue is resolved. The stack trace generally points to the exact bean and line of code that failed, making debugging a bit easier. It's crucial to handle configuration and dependencies carefully to avoid such issues during startup.

15. What will happen if two beans depend on each other via constructor injection?

If two beans are mutually dependent via constructor injection, Spring will throw a circular dependency exception. This is because constructor injection requires each bean to be fully instantiated before it can be injected, creating a deadlock situation when two constructors depend on each other. To resolve this, you can switch to setter injection, which allows Spring to create the beans first and inject dependencies afterward, or you can use `@Lazy` to delay the instantiation. However, the best approach is usually to refactor the design to remove circular dependencies altogether, as they generally indicate a tight coupling between components.

16. Can a bean be destroyed even if no reference exists in user code?

Yes, it can. Spring manages the entire lifecycle of beans, so destruction doesn't depend on whether the user code holds a reference or not. When the application context is closed—either manually or automatically—Spring goes through all singleton beans and calls their destruction callbacks like `@PreDestroy` or custom destroy methods. However, this applies only to singleton and certain scoped beans. Prototype-scoped beans are not tracked by the container after creation, so Spring doesn't automatically call their destroy methods. Developers need to handle cleanup for prototype beans themselves if required.

17. What happens if you close the `ApplicationContext` but try to access a singleton bean afterward?

If you attempt to access a singleton bean after the `ApplicationContext` has been closed, Spring will throw an `IllegalStateException`. This is because when the context is closed, all its resources—including singleton beans—are destroyed and cleaned up. At that point, the container is essentially shut down, and any attempt to retrieve or interact with managed beans is considered unsafe and invalid. It's important to ensure that the context is active and open before accessing any beans, especially in standalone or manually managed applications. Developers working with `ApplicationContextAware` or similar interfaces need to be cautious to avoid holding stale references post-shutdown.

18. How does Spring detect which classes to load into the IoC container at runtime?

Spring uses component scanning to detect which classes should be loaded into the IoC container. This scanning is enabled via the `@ComponentScan` annotation (or equivalent XML configuration), which tells Spring to look for classes annotated with stereotype annotations such as `@Component`, `@Service`, `@Repository`, and `@Controller`. The scan only includes classes within the specified base package or its sub-packages, so it's important to configure the base package path correctly. If enabled, this scanning happens automatically at startup.

Developers can also use custom filters to include or exclude specific classes based on custom annotations or patterns, allowing for more fine-grained control.

19. Can the IoC container create a bean if its constructor requires a runtime parameter (not a bean)?

No, Spring cannot automatically inject runtime parameters into a constructor unless they are known to the container. If a constructor requires a parameter that is not a bean—such as user input or a value read at runtime—Spring will not be able to resolve, it on its own. To handle this, developers typically use `@Value` to inject values from configuration files or provide the parameter explicitly using a factory method. Alternatively, using setter injection can offer more flexibility in such cases. The key point is that Spring can only inject what it knows about or is configured to provide.

20. You've annotated a class with `@Component`, but it isn't being picked up by Spring. What could be the possible reasons?

If a class marked with `@Component` isn't being detected by Spring, the most likely reason is that component scanning hasn't been enabled—either `@ComponentScan` is missing or incorrectly configured. Another common issue is specifying the wrong base package in the scan, which causes Spring to skip the class during the scanning process. Sometimes, the class might not have been compiled correctly, or it might not even be on the classpath. Developers should also ensure that the `@Component` annotation itself is properly imported and not misspelled or from the wrong package. In rare cases, the context might be configured in such a way that it doesn't initialize or scan the package containing the class.

21. You changed a dependency version in pom.xml but Maven still uses the old version. Why?

This usually happens because of dependency mediation in Maven. Even though you've updated the version in your pom.xml, Maven may still be pulling in a different version as a transitive dependency from another artifact. Maven uses the nearest-wins strategy in the dependency tree—meaning the version that is closest to your project in the hierarchy is selected. Another possibility is that Maven is using a cached version from your local repository, in which case running mvn dependency:purge-local-repository or mvn clean install -U can help. Additionally, check for version overrides in a parent POM or in a dependency management section, which can silently force an older version.

22. What will happen if two dependencies bring in different versions of the same transitive dependency?

When two of your dependencies each rely on different versions of the same transitive library, Maven will resolve this conflict using its **dependency mediation mechanism**. It selects the version that is closest to your project in the dependency tree. If both are at the same level, Maven typically uses the first one it encounters. However, this automatic resolution might not always be what you want. To explicitly control which version should be used, you can declare the dependency directly in your `<dependencies>` section with the desired version, or use `<dependencyManagement>` in the parent POM to centralize version control.

23. Can you explain the difference between compile, provided, and runtime scopes using real scenarios?

Sure. `compile` is the default scope—these dependencies are available in all build phases and are packaged into your final artifact. You use it for core libraries like utility functions. `provided` scope means the dependency is needed to compile your project but is expected to be provided by the runtime environment, such as a servlet API in a web container like Tomcat. These won't be included in your WAR or JAR. `runtime` dependencies are not needed at compile time but required at runtime—JDBC drivers are a common example. So, choosing the correct scope ensures the right dependencies are available without bloating your packaged application.

24. You have multiple modules with the same groupId and artifactId but different versions. How does Maven handle it?

In Maven, a combination of `groupId`, `artifactId`, and `version` uniquely identifies an artifact. If you have multiple modules with the same groupId and artifactId but different versions, Maven considers them different artifacts. However, this can lead to **unexpected conflicts** or confusion during dependency resolution, especially in multi-module projects. Maven might inadvertently pick an older or incorrect version depending on how dependencies are defined and resolved. It's a best practice to keep artifactId unique within the same group or clearly control versions using `<dependencyManagement>` to avoid such conflicts.

25. What happens if you omit the <version> tag for a dependency in your pom.xml?

If you omit the `<version>` tag for a dependency, Maven will throw a build error—**unless** the version is specified in a parent POM or within a `<dependencyManagement>` section. The `<dependencyManagement>` block allows parent or aggregator POMs to define versions for dependencies that children can inherit without explicitly specifying them. This helps maintain consistency across modules and simplifies version control. However, if the dependency isn't managed anywhere in the hierarchy, Maven cannot resolve the version and will fail to build your project.

26 Define framework

A **framework** is a ready-made structure or set of tools that helps us build software faster and in an organized way. It gives us a basic setup, so we don’t have to start everything from scratch — we just fill in our own code where needed.

Think of a **framework like a cake mold**.
We pour our own ingredients (code), but the mold (framework) gives it a shape and holds everything in place.

27 What is a spring framework?

Spring Framework is a popular Java-based framework used to build web applications, desktop apps, and even microservices.
It helps developers write clean, flexible, and loosely-coupled code. Spring is like a **toolbox for Java developers**. It

gives us ready-made features like Connecting to databases, Managing security, Handling web requests, Managing application objects automatically (called **Dependency Injection**)

Spring reduces boilerplate code, makes our app easier to test and maintain and supports modern architectures like REST APIs and microservices.

Let's say you're building a hospital management system — Spring can help to handle appointments through web pages using Spring MVC. We can store patient records in a database easily using Spring Data JPA. We can secure login for doctors and admins using Spring Security.

28 What is the difference between a console based application and a web based application?

A console-based application is a program that runs in a command-line interface or terminal. It interacts with the user through text input and output, without any graphical interface. These are typically used for simple utilities, testing, or educational purposes. Example if we write a program in java for say collection and to execute the same we will provide input in the console and see the output in the same console.

On the other hand, a web-based application runs on a web server and is accessed through a web browser using a network, usually the internet. It uses technologies like HTML, CSS, JavaScript on the front-end, and a back-end framework like Spring, Django, or Node.js.

29 What is the advantage of using spring framework?

Spring manages object creation and dependencies automatically. This reduces tight coupling between classes and makes the code easier to test and maintain.

We can use only the parts of Spring we need (like Spring MVC, Spring Data, etc.), making the application lightweight and flexible.

Spring offers powerful tools like Spring MVC for building web applications, With Spring JDBC and Spring Data JPA, we can easily connect to databases, write less boilerplate code, and manage transactions efficiently.

Spring Security provides comprehensive authentication and authorization features for securing web applications.