1. Introduction to Spring Framework

Definition:

The Spring Framework is a robust and widely-used Java framework that provides comprehensive infrastructure support for building enterprise-level applications. It addresses many common challenges in application development, such as managing dependencies, enhancing modularity, and simplifying various tasks through reusable components.

Key Concepts:

- **Inversion of Control (IoC):** IoC is the heart of Spring. It moves the responsibility of object creation and management from the application to the **Spring container**. This decouples components and promotes easier testing and maintenance.

**- Dependency Injection (DI):** DI is a crucial aspect of IoC. It's the process of injecting the required dependencies into a class rather than the class creating its own dependencies.

**- Aspect-Oriented Programming (AOP):** AOP is used to address cross-cutting concerns like logging, security, and transactions. It allows you to modularize these concerns separately from the main application logic.

**Advantages of Spring:**

- Modularity: Spring promotes a modular architecture, making it easier to develop and maintain large-scale applications.

- Flexibility: It supports various programming models and integrates seamlessly with other frameworks and technologies.

- Testing: IoC and DI make it simple to write unit tests for your components.

- Easier Configuration: Spring provides options for configuration using XML, Java annotations, or a combination of both.

- Loose Coupling: Dependencies are managed externally, leading to loosely coupled components.

Example:

Consider a web application. Instead of creating instances of service classes within controllers, Spring's IoC container manages these instances, ensuring better organization and separation of concerns.

---

2. Spring Core

Definition:

The Spring Core module provides the foundation for IoC and DI. It includes the Spring container, which manages the lifecycle of beans, their creation, and their dependencies.

Key Concepts:

- Bean: A bean is an object that's created, assembled, and managed by the Spring container.

- ApplicationContext: This interface extends BeanFactory and provides additional functionality like event handling and internationalization.

- Bean Lifecycle: Beans go through phases - instantiation, population of properties, and initialization.

- Bean Scopes: Scopes determine the lifecycle and visibility of beans (singleton, prototype, etc.).

ApplicationContext vs. BeanFactory:

- ApplicationContext provides more features, including support for internationalization, event propagation, and AOP.

- BeanFactory focuses on providing basic DI functionality with reduced overhead.

**About BeanFactory**

The first foremost thing when we talk about spring is dependency injection which is possible because spring is actually a container and behaves as a factory of Beans. Just like the  BeanFactory interface is the simplest container providing an advanced configuration mechanism to instantiate, configure and manage the life cycle of beans. Beans are java objects that are configured at run-time by Spring IoC Container. BeanFactory represents a basic**IoC container** which is a parent interface of **ApplicationContext.** **BeanFactory** uses Beans and their dependencies metadata to create and configure them at run-time. BeanFactory loads the bean definitions and dependency amongst the beans based on a configuration file(XML) or the beans can be directly returned when required using Java Configuration. There are other types of configuration files like LDAP, RDMS, properties file, etc. BeanFactory does not support Annotation-based configuration whereas ApplicationContext does.

**About ApplicationContext**

Spring IoC container is responsible for instantiating, wiring, configuring, and managing the entire life cycle of objects. BeanFactory and ApplicationContext represent the Spring IoC Containers. ApplicationContext is the sub-interface of BeanFactory. BeanFactory provides basic functionalities and is recommended to use for lightweight applications like mobile and applets. ApplicationContext provides basic features in addition to enterprise-specific functionalities which are as follows:

* Publishing events to registered listeners by resolving property files.
* Methods for accessing application components.
* Supports Internationalization.
* Loading File resources in a generic fashion.

Example:

```java

public class MyApp {

public static void main(String[] args) {

ApplicationContext context = new ClassPathXmlApplicationContext("applicationContext.xml");

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

service.doSomething();

}

}

```

---

3. Spring Containers

Definition:

Spring containers manage the instantiation, configuration, and lifecycle of beans. They're responsible for wiring beans together and maintaining their dependencies.

Types of Containers:

- BeanFactory: The simplest container that provides basic DI support.

- ApplicationContext: Offers advanced features like automatic bean post-processing, event propagation, and more.

- WebApplicationContext: Specialized for web applications, it inherits from ApplicationContext and adds support for web-specific contexts.

Example:

```xml

<!-- applicationContext.xml -->

<beans xmlns="http://www.springframework.org/schema/beans"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd">

<bean id="myBean" class="com.example.MyBean"/>

</beans>

```

---

4. Spring Configuration

Definition:

Spring configuration is how you provide instructions to the Spring container on how to create and wire beans. It can be achieved using XML, Java configuration classes, or annotations.

Types of Configuration:

- XML Configuration: Beans and their relationships are defined in XML files.

- Java Configuration: Beans are defined in Java classes using the `@Configuration` annotation.

- Annotation-based Configuration: Beans are defined using annotations like `@Component`, `@Service`, etc.

Example:

```java

@Configuration

public class AppConfig {

@Bean

public MyService myService() {

return new MyServiceImpl();

}

}

```

---

5. Dependency Injection and Inversion of Control

Definition:

Dependency Injection is a design pattern that follows the Inversion of Control principle. It allows you to inject dependencies into a class rather than having the class create its dependencies.

Types of DI:

- Constructor Injection: Dependencies are injected through the constructor.

- Setter Injection: Dependencies are injected through setter methods.

- Method Injection: Dependencies are injected into specific methods.

Example:

```java

public class OrderService {

private PaymentService paymentService;

// Constructor Injection

public OrderService(PaymentService paymentService) {

this.paymentService = paymentService;

}

// ...

}

```

---

6. Spring Annotations

Definition:

Spring annotations reduce the need for XML-based configurations by allowing you to configure beans and their relationships directly in source code.

Common Spring Annotations:

- @Component: Marks a class as a Spring component (bean).

- @Autowired: Injects dependencies automatically.

- @Configuration: Indicates a Java configuration class.

- @Scope: Specifies the scope of a bean (singleton, prototype, etc.).

- @Service: Indicates a service component.

- @Controller: Marks a class as a controller component.

Example:

```java

@Component

public class MyService {

// ...

}

```