Spring

Spring Framework's **Inversion of Control (IoC)** and its modules (spring-core.jar and spring-context.jar):

---

1. Containers

The **IoC Container** is the backbone of the Spring Framework, responsible for managing the lifecycle and configuration of beans (objects). It does this by creating, configuring, and assembling dependencies. There are two main types of containers:

- **BeanFactory**: A basic container that provides support for dependency injection and bean lifecycle management. It's lightweight and used in resource-constrained environments.
- **ApplicationContext**: A more advanced container that extends BeanFactory and provides additional features like event propagation, internationalization, and integration with other frameworks like AOP.

Key Role: The IoC container handles the entire object creation process and injects dependencies where needed.

---

2. Dependency Injection (DI)

Dependency Injection is a design pattern where the IoC container injects objects (dependencies) into other objects instead of creating them directly. Spring supports the following DI methods:

- **Constructor Injection**: Dependencies are provided through a class constructor.
- **Setter Injection**: Dependencies are set via public setter methods.
- **Field Injection**: Dependencies are directly injected into fields using annotations like @Autowired.

Key Advantage: It promotes loose coupling by allowing objects to depend on interfaces or abstractions rather than concrete implementations.

---

3. Auto Wiring

Autowiring simplifies dependency injection by letting the IoC container automatically wire (inject) relationships between beans. It reduces manual configuration. Spring provides several autowiring modes:

- **@Autowired**: Automatically resolves and injects dependencies based on type.

- **@Qualifier**: Specifies which bean to use when multiple candidates match the type.
- **@Primary**: Marks a bean as the default choice when multiple candidates exist.

Key Feature: Autowiring eliminates the need for explicit XML configuration, streamlining the setup process.

---

4. Lifecycle Management

Spring manages the lifecycle of beans, from their initialization to destruction. The container provides hooks for custom initialization and destruction methods:

- **Initialization Phase**: After the bean is instantiated, custom methods annotated with @PostConstruct (or declared in XML configuration) are called.
- **Destruction Phase**: When the container is shut down, methods annotated with @PreDestroy (or declared in XML) are executed.

Key Component: The IoC container ensures that beans are properly initialized, dependencies are injected, and resources are cleaned up efficiently.

---

5. Stereotypes

Stereotypes are annotations that help define the role of a class in the application. These annotations also act as metadata for Spring to register the class as a bean in the container:

- **@Component**: A general-purpose stereotype for any Spring-managed component.
- **@Service**: Specifically used for service-layer classes.
- **@Repository**: Indicates Data Access Object (DAO) classes.
- **@Controller**: Marks a class as a web controller in Spring MVC applications.
- **@RestController**: A combination of @Controller and @ResponseBody, used for RESTful APIs.

Key Benefit: Stereotypes make it easy to define and organize the different layers of an application by role.

Together, these elements make the **IoC Container** in the Spring Framework a powerful tool for managing the lifecycle, dependencies, and roles of objects in an application.

Spring

Q) Difference Between **BeanFactory** and **ApplicationContext**?

Ans: The **BeanFactory** and **ApplicationContext** are both types of IoC containers in Spring, but they differ in functionality and features.

Let me explain the Differences:

---

1. Basic Purpose

- **BeanFactory**:
    - It is the simpler and more basic container in Spring.
    - Its primary purpose is to manage the lifecycle of beans and handle dependency injection.
    - It lazily initializes beans, meaning beans are created only when they are requested.
- **ApplicationContext**:
    - It is an advanced container built on top of BeanFactory.
    - It provides additional enterprise-oriented features like event handling, internationalization, and application-layer-specific functionalities.
    - It eagerly initializes beans by default, creating them at startup unless explicitly configured otherwise.

---

2. Features

- **BeanFactory**:
    - Offers basic dependency injection.
    - Does not provide support for advanced features like events, AOP (Aspect-Oriented Programming), or declarative transaction management.
    - Requires manual loading of the XML configuration file.
- **ApplicationContext**:
    - Includes everything BeanFactory offers but with additional features like:
        - Event propagation and listener mechanisms.
        - Built-in support for resolving message sources (internationalization).

        - Integration with Spring AOP and declarative transactions.
        - Annotation-based configuration and scanning (@Component, @Service, etc.).
    - Automatically detects and registers beans from annotated classes.

---

3. Use Cases

- **BeanFactory**:
    - Suitable for lightweight applications with simple requirements.
    - Preferred in memory-constrained environments where performance is critical.
- **ApplicationContext**:
    - Ideal for enterprise applications with complex requirements.
    - Recommended for most Spring applications due to its rich set of features.

---

4. Examples of Usage

- **BeanFactory**:

```
Resource resource = new ClassPathResource("beans.xml");
BeanFactory factory = new XmlBeanFactory(resource);
MyBean myBean = factory.getBean(MyBean.class);
```

- **ApplicationContext**:

```
ApplicationContext context = new
ClassPathXmlApplicationContext("beans.xml");
MyBean myBean = context.getBean(MyBean.class);
```

Summary Table

Feature	BeanFactory	ApplicationContext
Initialization	Lazy (on request)	Eager (at startup)
Event Handling	Not supported	Supported
Internationalization	Not supported	Supported

Spring

AOP & Transactions	Limited	Fully supported
Annotation Support	Limited	Extensive
Use Case	Lightweight/simple apps	Enterprise/complex apps

In general, **ApplicationContext** is preferred for most real-world applications due to its extended functionality and developer convenience. However, understanding both is valuable depending on the context of your project.