Spring Core Introduction / Overview

Spring Framework:

- A comprehensive framework for Java applications.
- Provides infrastructure support for developing Java applications.
- Initially designed to simplify enterprise Java development and to promote good programming practices by enabling POJO-based programming.

Core Concepts:

- **Inversion of Control (IoC):**
  - A design principle where the control flow of a program is inverted.
  - In Spring, the control of creating and managing objects is transferred from the application code to the Spring container.
- **Dependency Injection (DI):**
  - A design pattern to implement IoC.
  - Objects define their dependencies (e.g., via constructors or setters), and the container injects these dependencies at runtime.

Modules of Spring Framework:

- Core Container (Spring Core, Beans, Context, SpEL)
- Data Access/Integration (JDBC, ORM, OXM, JMS, Transactions)
- Web (Web, WebMVC, WebSocket, WebPortlet)
- AOP (Aspect-Oriented Programming)
- Instrumentation
- Messaging
- Test

Spring Container

Spring IoC Container:

- The core of the Spring Framework.
- Responsible for instantiating, configuring, and assembling the beans.
- Two main types of containers:
  - **BeanFactory:** The simplest container providing basic DI support.
  - **ApplicationContext:** A more advanced container that includes enterprise-specific functionality like event propagation, declarative mechanisms to create a bean, etc.

BeanFactory:

- Interface for the IoC container.
- Provides the configuration framework and basic functionality for managing beans.
- Supports lazy initialization, meaning beans are created only when they are requested.

ApplicationContext:

- Extends BeanFactory.
- Provides more advanced features.
- Supports:
    - **Message Resource Handling:** For internationalization.
    - **Event Propagation:** For implementing application event handling.
    - **AOP Integration:** Directly integrated with Spring's AOP.

Configuration Example:

```
java
Copy code
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

public class Main {
    public static void main(String[] args) {
        ApplicationContext context = new
ClassPathXmlApplicationContext("beans.xml");
        MyBean myBean = context.getBean(MyBean.class);
        myBean.doSomething();
    }
}
```

beans.xml:

```
xml
Copy code
<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>
```

Dependency Injection

Dependency Injection (DI):

- A fundamental concept in Spring that promotes loose coupling.
- Enables the creation of dependent objects outside of a class and provides those objects to a class in various ways.

Types of DI:

1. **Constructor Injection:**
    - Dependencies are provided through the constructor of the class.
    - Ensures that an object is created with all its dependencies.

```
java
Copy code
public class MyService {
    private final MyRepository repository;

    public MyService(MyRepository repository) {
```

```
        this.repository = repository;
    }

    public void performAction() {
        repository.action();
    }
}
```

Configuration:

```
xml
Copy code
<bean id="myService" class="com.example.MyService">
    <constructor-arg ref="myRepository" />
</bean>
<bean id="myRepository" class="com.example.MyRepository" />
```

2. **Setter Injection:**
    - Dependencies are provided through setter methods after the object is constructed.
    - Allows for optional dependencies and changing dependencies.

```
java
Copy code
public class MyService {
    private MyRepository repository;

    public void setRepository(MyRepository repository) {
        this.repository = repository;
    }

    public void performAction() {
        repository.action();
    }
}
```

Configuration:

```
xml
Copy code
<bean id="myService" class="com.example.MyService">
    <property name="repository" ref="myRepository" />
</bean>
<bean id="myRepository" class="com.example.MyRepository" />
```

3. **Field Injection (Not Recommended):**
    - Dependencies are injected directly into the fields.
    - Requires the use of reflection, making it less flexible and harder to test.

```
java
Copy code
public class MyService {
    @Autowired
    private MyRepository repository;

    public void performAction() {
        repository.action();
```

```
    }
}
```

Configuration:

```
xml
Copy code
<context:component-scan base-package="com.example" />
```

Metadata / Configuration

Configuration Metadata:

- Defines how beans are configured and managed in the Spring container.
- Can be provided in various formats:
    - **XML Configuration:**
        - Traditional way of configuring Spring beans.
        - Uses XML files to define beans and their dependencies.

```
xml
Copy code
<bean id="myBean" class="com.example.MyBean">
    <property name="property1" value="value1" />
</bean>
```

    - **Java-based Configuration:**
        - Uses `@Configuration` annotated classes to define beans.
        - Provides a type-safe, refactoring-friendly way of configuring Spring beans.

```
java
Copy code
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

@Configuration
public class AppConfig {

    @Bean
    public MyBean myBean() {
        return new MyBean();
    }
}
```

    - **Annotation-based Configuration:**
        - Uses annotations to define bean dependencies directly in the component classes.
        - Annotations like `@Component`, `@Service`, `@Repository`, `@Autowired`, and `@Qualifier` are commonly used.

```
java
Copy code
import org.springframework.stereotype.Component;
import org.springframework.beans.factory.annotation.Autowired;
```

```
@Component
public class MyService {

    private final MyRepository repository;

    @Autowired
    public MyService(MyRepository repository) {
        this.repository = repository;
    }

    public void performAction() {
        repository.action();
    }
}
```

Spring Profiles:

- Allows to segregate parts of the application configuration and make it available only in certain environments.
- Use `@Profile` to annotate configuration classes or bean definitions.
- Example:

```
java
Copy code
@Configuration
@Profile("development")
public class DevelopmentConfig {

    @Bean
    public MyBean myBean() {
        return new MyBean("Development Bean");
    }
}
```

Property Source:

- Externalizes configuration to properties files, YAML files, or environment variables.
- Use `@PropertySource` to specify the location of properties files.
- Example:

```
java
Copy code
@Configuration
@PropertySource("classpath:application.properties")
public class AppConfig {

    @Value("${my.property}")
    private String myProperty;

    @Bean
    public MyBean myBean() {
        return new MyBean(myProperty);
    }
}
```