Spring

--> Spring is a framework that can build enterprise applications
-> spring was introduced by Rod Johnson in the year 2004
-> Spring is named after the Spring season
-> spring is a lightweight application framework
-> spring needs less configuration compared to EJB(Enterprise java bean)
->spring follows lose coupling
-> it help us to build the application with the help of POJO class

* **Lose coupling** =In System one object does not know the existince of other object
Employee e=Class.createEmployee();

***Tight coupling**-In System one object know the existince of other object
Employee e=new Employee();

Modules of spring

Spring Core - heart of spring framework,basic configuration
Spring MVC -framework servlet
Spring BOOT -framework of spring
Spring AOP-Repeated logic in spring
Spring Security-authorization and authentication for spring application
Spring cloud-

Spring Components

1.POJO class
2.Driver Class
3.Configuration file

Spring Follows IOC

IOC -inversion of control - it transfering control to some other entity

DI-Dependency Injection

Spring Container

spring container maintains the life of POJO class along with help of configuration file

Configuration File

-->The configuration metadata is represented in XML or Java code

--> it should create in java\src\main

there are types of configuration

1.XML based

2.Java based

Two Spring container

1. Core container
2. JEE container

Core Container

BeanFactory

```
Resource r=new ClassPathResource("config.xml");

        BeanFactory b=new ClassPathXmlApplicationContext("config.xml");
        Person p=(Person) b.getBean("myPerson");
        Person p1=b.getBean("beanID",Person.class);
        p.call();
```

XML-based application context

->The id attribute is a string that identifies the individual bean definition.

->The class attribute defines the type of the bean and uses the fully qualified class name.

The following example shows the basic structure of XML-based configuration metadata:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:context="http://www.springframework.org/schema/context"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/context
        http://www.springframework.org/schema/context/spring-context-3.0.xsd">
        <bean id="myPerson" class="com.pojo.Person" scope="prototype"></bean>
        <bean id="myPersons" class="com.pojo.Person" init-method="call1" destroy-method="m2"></bean>
</beans>
```

->The id attribute is a string that identifies the individual bean definition.
The class attribute defines the type of the bean and uses the fully qualified class name.

The convention is to use the standard Java convention for instance field names when naming beans.
accountManager, accountService, userDao, loginController

J2EE container

-> The ApplicationContext interface represents the Spring IoC container and is responsible for instantiating, configuring, and assembling the beans.

->The container gets its instructions on what objects to instantiate, configure, and assemble by reading configuration metadata

-> implementations of the ApplicationContext interface are ClassPathXmlApplicationContext

Dependency injection (DI)

Dependency injection (DI) is a process whereby objects define their dependencies only through constructor arguments, arguments to a factory method, or properties that are set on the object instance after it is constructed or returned from a factory method.

->The container then injects those dependencies when it creates the bean. This process is fundamentally the inverse (hence the name, Inversion of Control) of the bean itself controlling the instantiation or location of its dependencies on its own by using direct construction of classes or the Service Locator pattern.

Constructor-based DI is accomplished by the container invoking a constructor with a number of arguments, each representing a dependency

```
<bean id="myTeacher" class="com.constructor.Teacher">
<constructor-arg value="101"></constructor-arg>
<constructor-arg><value>mr</value> </constructor-arg>
</bean>
```

Constructor-based DI collection

```
<bean id="myStudent" class="com.constructor.Student">
    <constructor-arg>
    <list>
     <value>hello</value>
     <value>hell</value>
     <value>35</value>
    </list>

    </constructor-arg>
        </bean>
```

Constructor-based DI has-relation

```
<bean id="myStudent" class="com.constructor.Student">
                <constructor-arg>
                        <value>1</value>
                </constructor-arg>
                <constructor-arg>
                        <value>dinga</value>
                </constructor-arg>
                <constructor-arg value="1252525"></constructor-arg>
                <constructor-arg ref="myTeacher"></constructor-arg>
        </bean>
```

Setter-based DI is accomplished by the container calling setter methods on your beans after invoking a no-argument constructor

```
<bean id="myTeacher" class="com.constructor.Teacher">
 <property name="id" value="101"></property>
 <property name="name"><value>mr</value> </property>
  </bean>
```

Setter based DI Has-relationship

```
<bean id="myTeacher" class="com.constructor.Teacher">
 <property name="id" value="101"></property>
 <property name="name"><value>dinga</value> </property>
  </bean>
<bean id="myStudent" class="com.constructor.Student">
                <property name="id" value="1"></property>
                <property name="name" value="dinga"></property>
                <property name="phone" value="1255255"></property>
                <property name="teacher" ref="myTeacher"></property>
 </bean>
```

Setter-based DI collection

```
<bean id="myStudent" class="com.constructor.Student">
                <list>
                <value>english</value>
                <value>science</value>
                <value>physics</value>
                <value>english</value>
```

```
        </list>
        </property>
        <property name="map">
        <map>
        <entry key="10" value="a"></entry>
        <entry key="11" value="a"></entry>
        <entry key="12" value="a"></entry>
        <entry key="13" value="a"></entry>
        </map>
        </property>
</bean>
```

Post construct and pre destroy

It defines which method need to be call after the spring create instance of object and before destroying class

```
<bean id="myPersons" class="com.pojo.Person" init-method="call1"
destroy-method="m2"></bean>
```

Spring Scope

1.Singleton

2.prototype

Spring by default use Singleton principle for bean instantiation

Singleton means one instance for class

prototype Scope

```
<bean id="myPerson" class="com.pojo.Person" scope="prototype"></bean>
```

The `@Value` annotation in Spring Boot is used to inject values from properties files, environment variables, or other Spring beans into fields or method parameters.

The `@Autowired` annotation in Spring Boot is used for automatic dependency injection. It can be applied to fields, constructors, or setter methods.

The `@Component` annotation in Spring is a generic stereotype annotation used to indicate that a class is a Spring component.

The `@Qualifier` annotation in Spring is used to disambiguate beans when multiple beans of the same type are present in the application context. It can be applied to fields, methods, or constructor parameters along with the `@Autowired` annotation to specify the name of the desired bean.

The `@Configuration` annotation is used to declare a class as a configuration class in Spring.

The `@Bean` annotation is used in a method within a `@Configuration` class to declare a bean.

The `@Scope` annotation is used to specify the scope of a Spring bean. Common scope values include "singleton," "prototype.

The `@Primary` annotation is used to indicate the primary bean when multiple beans of the same type are present in the Spring container. It is commonly used in scenarios where there are multiple candidate beans for injection, and the `@Primary` bean is chosen as the default.

The `@ComponentScan` annotation is used to enable component scanning in a Spring application. It specifies the base packages to scan for Spring components