Spring Is open source and light weight. It uses pojo class. So no need of ejb container,they can go with tomcat server. Spring is organized in a modular fashion. Even though the number of packages and classes, you have to worry only about the ones you need and ignore the rest. Spring does not reinvent the wheel, instead it truly makes use of some of the existing technologies

In Spring Objects are loosely coupled i.e., each class is independent of each other so that everything can be tested individually. But when using those classes, a class may be dependent on other classes which need to be instantiated first. So, we tell spring that class A is dependent on class B. So, when creating bean(like class) for class A, it instantiates class B prior to that of class A and injects that in class A using setter or constructor DI methods. I.e., we are telling spring the dependency at run-time. This is DI.

As, we are assigning the responsibility of creating objects(beans), maintaining them and their aggregations to Spring instead of hard-coding it, we call it Inversion Of Control(IOC).

The Spring Inversion of Control container is the heart of the Spring Framework. The IoC container receives metadata from either an **XML file, Java annotations, or Java code**. The container gets its instructions on what objects to instantiate, configure, and assemble from simple Plain Old Java Objects (POJO) by reading the configuration metadata provided.

Constructor-based DI is accomplished when the container invokes a class constructor with a number of arguments, each representing a dependency on the other class.

Setter-based DI is accomplished by the container calling setter methods on your beans after invoking a no-argument constructor or no-argument static factory method to instantiate your bean.

You can mix both, Constructor-based and Setter-based DI but it is a good rule of thumb to use constructor arguments for mandatory dependencies and setters for optional dependencies.

Reason for DI:

The code is cleaner with the DI principle. The object does not look up its dependencies and does not know the location or class of the dependencies, rather everything is taken care by the Spring Framework.

note the difference in Beans.xml file defined in the constructor-based injection and the setter-based injection. The only difference is inside the <bean> element where we have used <constructor-arg> tags for constructor-based injection and <property> tags for setter-based injection.

The second important point to note is that in case you are passing a reference to an object, you need to use **ref** attribute of <property> tag and if you are passing a **value** directly then you should use value attribute.

1Architecture layers:

Core container-

Core ioc and di

Bean beanfactory

Context applicationcontext

2Dataacces/integeration

JDBC

ORM

OXM

JMS

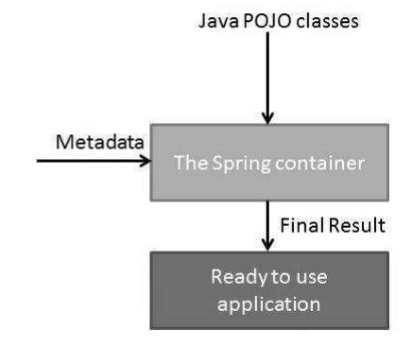
Transaction

3Web:

Web, Web-MVC, Web-Socket, and Web-Portlet

There are few other important modules like AOP, Aspects, Instrumentation, etc;

The container gets its instructions on what objects to instantiate, configure, and assemble by reading the configuration metadata provided. The configuration metadata can be represented either by XML, Java annotations, or Java code. The following diagram represents a high-level view of how Spring works. The Spring IoC container makes use of Java POJO classes and configuration metadata to produce a fully configured and executable system or application.



A lazy-initialized bean tells the IoC container to create a bean instance when it is first requested, rather than at the startup.

Bean scopes: 5

Singleton (default)

Prototype

request

Session

Global session

For example, to force Spring to produce a new bean instance each time one is needed, you should declare the bean's scope attribute to be **prototype**. Similarly, if you want Spring to return the same bean instance each time one is needed, you should declare the bean's scope attribute to be **singleton**.

Beanlifecycle:  
To define setup and teardown for a bean, we simply declare the <bean> with **initmethod** and/or **destroy-method** parameters. The init-method attribute specifies a method that is to be called on the bean immediately upon instantiation. Similarly, destroymethod specifies a method that is called just before a bean is removed from the container.

<bean id = "exampleBean" class = "examples.ExampleBean" init-method = "init"/>

Following is the class definition −

public class ExampleBean {

public void init() {

// do some initialization work

}

}

void destroy() throws Exception;

public class ExampleBean implements DisposableBean {

public void destroy() {

// do some destruction work

}

}

**MainApp.java** file. Here you need to register a shutdown hook **registerShutdownHook()** method that is declared on the AbstractApplicationContext class. This will ensures a graceful shutdown and calls the relevant destroy methods.

* The first step is to create factory object where we used framework API**FileSystemXmlApplicationContext** to create the factory bean after loading the bean configuration file from the given path. The**FileSystemXmlApplicationContext()** API takes care of creating and initializing all the objects ie. beans mentioned in the XML bean configuration file.
* The second step is used to get the required bean using **getBean()**method of the created context. This method uses bean ID to return a generic object, which finally can be casted to the actual object. Once you have an object, you can use this object to call any class method.

**BeanFactory** is also called basic IOC and **ApplicationContext** is called Advanced IOC. The**Application Context** is **Spring's** advanced container. Similar to BeanFactory, it can load bean definitions, wire beans together, and dispense beans upon request. ... BeanFactory can still be used for lightweight applications like mobile devices or applet-based applications.

The most commonly used ApplicationContext implementations are −

* **FileSystemXmlApplicationContext** − This container loads the definitions of the beans from an XML file. Here you need to provide the full path of the XML bean configuration file to the constructor.
* **ClassPathXmlApplicationContext** − This container loads the definitions of the beans from an XML file. Here you do not need to provide the full path of the XML file but you need to set CLASSPATH properly because this container will look like bean configuration XML file in CLASSPATH.
* **WebXmlApplicationContext** − This container loads the XML file with definitions of all beans from within a web application.