

**NAME : ROHIT KUMAR PANDEY**  
**SUBJECT : SPRING FRAMEWORK**

**Spring is a dependency injection framework to make java application loosely coupled.**

**Spring framework makes the easy development of JavaEE application.**

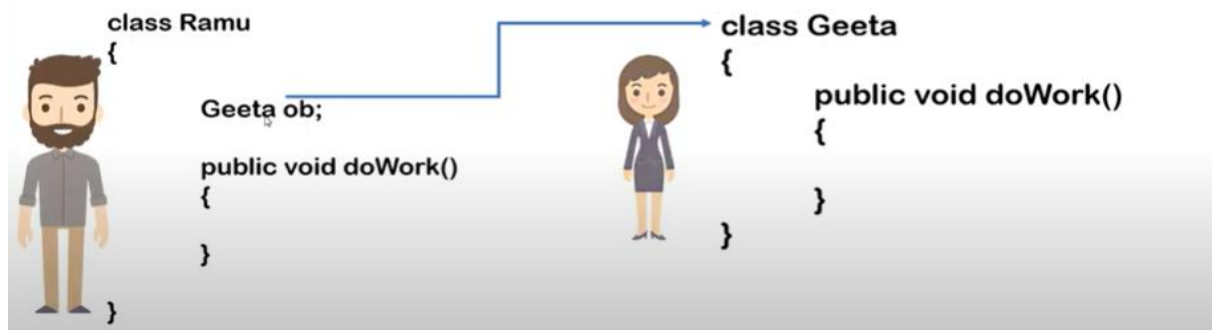
**It was developed by Rod Johnson in 2003.**

## **Dependency injection**

It is a design pattern that are used to remove dependency from the programming code.

Suppose we have two classes Ramu and Geeta. Ramu class needs the help of work for doing his task, that is Ramu class needs Geeta object. This is called as Ramu class is dependent on Geeta class.

In general what we do is we create an object of Geeta class in Ramu



class and use it. But it makes the Ramu class tightly coupled.

## **Problem with tight coupling**

Suppose in future we want to make changes in the code. Then if Geeta object is tightly coupled with Ram then we need to make changes with the code and need to rebuild the software and restart the server which is very unrealistic.

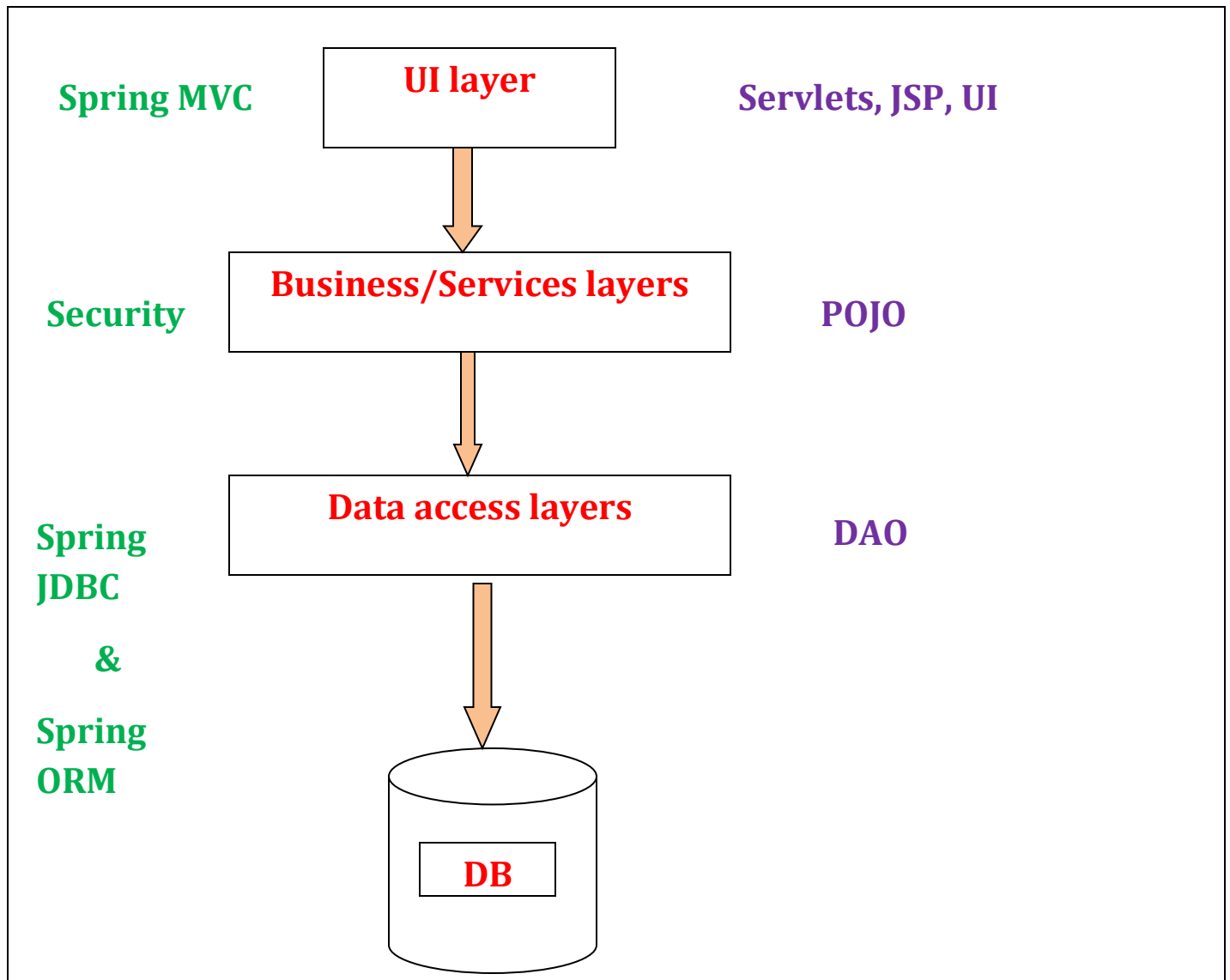
**Not easy for testing** : Also this approach creates a lot of problems while testing the application especially in black box testing.

## **How spring helps us**

Spring provides us feature of dependency injection. Spring will create the object of Geeta by default at runtime and will inject it in Ram . For doing this we just need to make some configuration in xml file(or annotation). In future if we are required to make some changes then we just need to reconfigure the application by making changes in the xml file. We don't need to restart the server which saves lot of manpower and money.

This process of giving the control for injecting the object from user to the Spring is called as ***Inversion of Control(IOC)***.

## **Java EE Layered Architecture**



- When some request came from client, then initially it is handled by UI Component (Servlets & JSP).
- This layer requires the object of POJO classes (Entity), so we create objects in the servlets.

- This it required the access of database which is taken care by the Dao classes, so we create Dao Objects also.

In general we create all classes by ourself, which creates tight coupling.

Now if we use spring then , Spring automatically will create will object of Dao classes and inject it in Business layer. Again will create the object of Business layer classes and inject it in UI Layer classes.

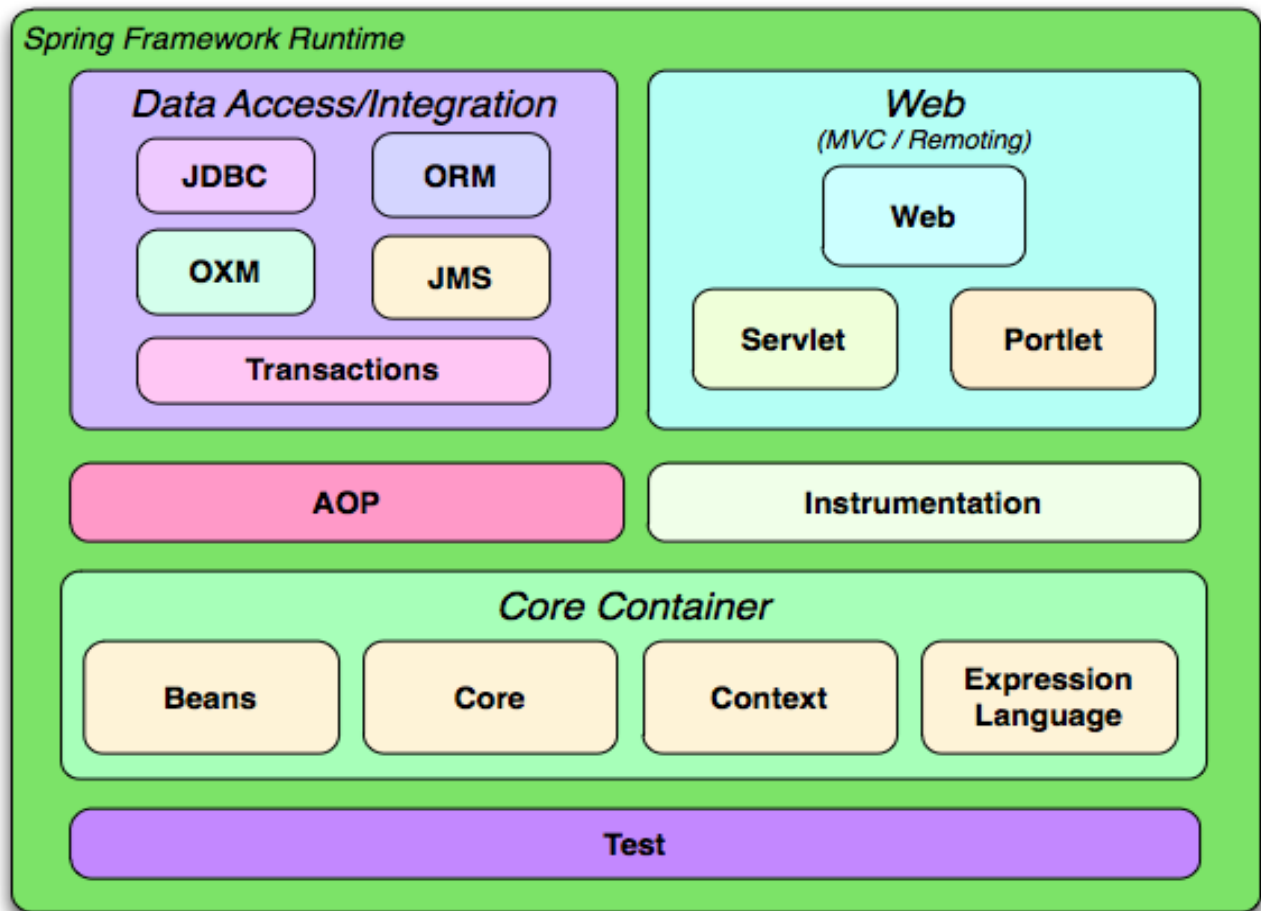
So, in this way spring removes tight coupling from the software.

## **Spring Modules**

The Spring Framework contains a lot of features, which are well-organized in about twenty modules.

These modules can be grouped together based on their primary features into –

- Core Container
- Data Access/Integration,
- Web,
- AOP (Aspect Oriented Programming),
- Instrumentation
- Test



## CORE CONTAINER

The Core Container consists of the Core, Beans, Context and Expression modules.

**Core and Beans :** These modules provide the most fundamental parts of the framework and provide the IoC and Dependency Injection features.

**Context module :** This module inherits its features from the Beans module and adds support for internationalization, event-propagation, resource-loading, and the transparent creation of contexts by, for example, a servlet container. The Context module

also contains support for some Java EE features like EJB, JMX and basic remoting support.

**Expression Language** This module provides a powerful expression language for querying and manipulating an object graph at runtime.

## **DATA ACCESS/INTEGRATION**

The *Data Access/Integration* layer consists of the JDBC, ORM, OXM, JMS and Transaction modules.

**JDBC** : This module provides a JDBC-abstraction layer that removes the need to do tedious JDBC coding and parsing of database-vendor specific error codes.

**ORM** : This module provides integration layers for popular object-relational mapping APIs, including [JPA](#), [JDO](#), [Hibernate](#), and [iBatis](#).

**OXM** : This module provides an abstraction layer for using a number of Object/XML mapping implementations. Supported technologies include JAXB, Castor, XMLBeans, JiBX and XStream.

**JMS** : This module provides Spring's support for the Java Messaging Service. It contains features for both producing and consuming messages.

**Transaction** : This module provides a way to do programmatic as well as declarative transaction management, not only for classes implementing special interfaces, but for *all your POJOs (plain old Java objects)*.

## **WEB**

The *Web* layer consists of the Web, Web-Servlet and Web-Portlet modules.

**Spring's Web :** This module provides basic web-oriented integration features, such as multipart file-upload functionality, the initialization of the IoC container using servlet listeners and a web-oriented application context.

**Web-Servlet :** This module provides Spring's Model-View-Controller ([MVC](#)) implementation for web-applications..

**Web-Portlet :** This module provides the MVC implementation to be used in a portlet environment and mirrors what is provided in the Web-Servlet module.

## **AOP AND INSTRUMENTATION**

**Spring's AOP :** This module provides an *AOP Alliance*-compliant aspect-oriented programming implementation allowing you to define,method-interceptors and pointcuts to cleanly decouple code implementing functionality that should logically speaking be separated.There is also a separate *Aspects* module that provides integration with AspectJ.

**The Instrumentation :**This module provides class instrumentation support and classloader implementations to be used in certain application servers.

# TEST

The *Test* module contains the Test Framework that supports testing Spring components using JUnit or TestNG. It also contains a number of Mock objects that are useful in many testing scenarios to test your code in isolation.

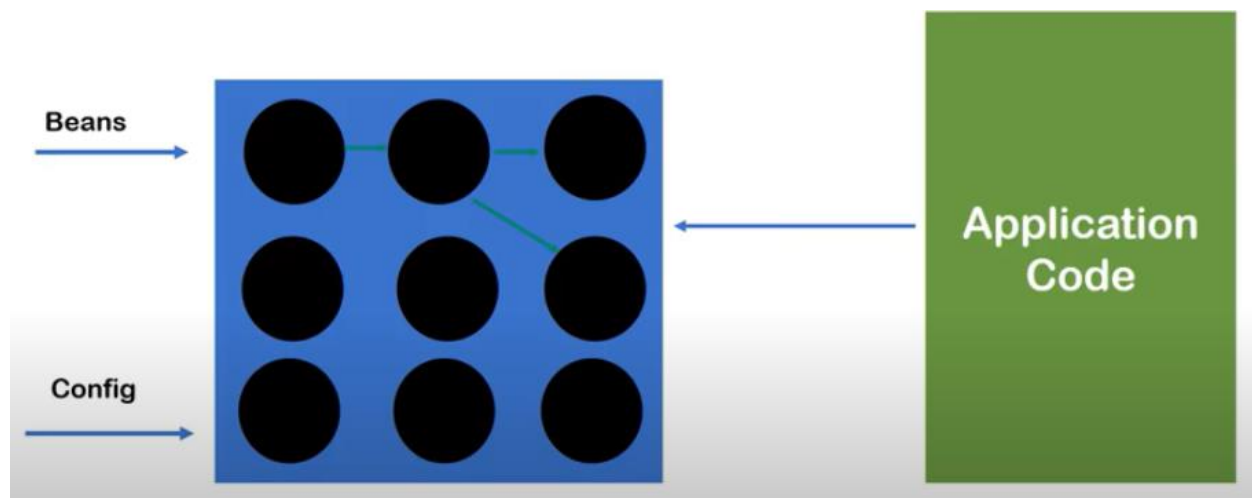
## Spring IOC Container

IOC container is a predefined program which we get by default with spring.

IOC Container is responsible for-

- Creating Objects
- Holding objects in the memory
- Injecting one object to another object as required.

Means, IOC Container manages the complete life cycle of the object.



We have to tell two things to the IOC Container-



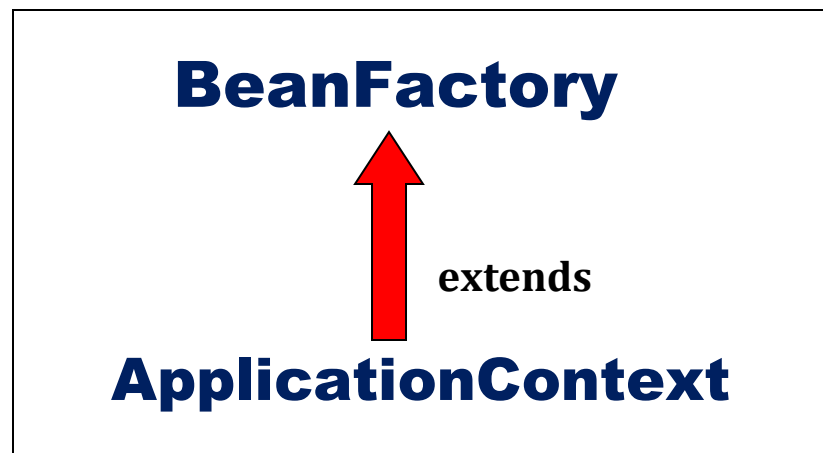
- **Beans** : That is ,the POJO classes which has to managed by IOC Container.
- **Configuration File(XML)** : Which bean is dependent of which other class.

Spring IOC Container will do all these things then , the user can directly use all those objects in the application code.

There are two types of IoC containers. They are:

1. **BeanFactory**
2. **ApplicationContext**

The **org.springframework.beans.factory.BeanFactory** and the **org.springframework.context.ApplicationContext** interfaces acts as the IoC container.



The **ApplicationContext** interface is built on top of the **BeanFactory** interface. It adds some extra functionality than **BeanFactory** .So it is better to use **ApplicationContext** than **BeanFactory**.

## Using ApplicationContext

ApplicationContext is an interface, so we cannot directly create an object of it. So we have to search for the classes that implement the ApplicationContext interface.

Following classes implement the ApplicationContext interface.

### **ApplicationContext**

- **AnnotationConfigApplicationContext**
- **ClasspathXMLApplicationContext**
- **FileSystemXMLApplicationContext**

**ClasspathXMLApplicationContext** : It searches for the xml configuration in the java class path.

**AnnotationConfigApplicationContext** : It is used for those beans for which we have used annotations.

**FileSystemXMLApplicationContext** : It searches for xml file from the file system.

In general purpose usage we have to use **ClasspathXMLApplicationContext** class to get IOC Container.

# Dependency Injection in Spring

Dependency Injection (DI) is a design pattern that removes the dependency from the programming code so that it can be easy to manage and test the application. Dependency Injection makes our programming code loosely coupled.

Class Student

```
{  
    int id;  
    String name;  
    Address address;  
}
```

Class Address

```
{  
    String Street;  
    String city;  
    String state;  
    String country;  
}
```

IOC container will first create Address object.

In the address object IOC container sets the value of all the fields at runtime .

After that IOC container will crete object of Student class.

First it will sets the values of id and name, then it will inject Address object in the student object.

## Address

Street : 91A

City : Kolkata

State : WB

Country : India



The diagram consists of two concentric ovals. The outer oval is labeled 'Student' and contains the text 'Id : 101' and 'Name:Rohit Pandey'. Inside the outer oval is a smaller inner oval labeled 'Address', which contains the text 'Street : 91A', 'City : Kolkata', 'State : WB', and 'Country : India'.

### Student

Id : 101

Name:Rohit Pandey

### Address

Street : 91A

City : Kolkata

State : WB

Country : India


The users don't need to do all those stuff manually. Now user can directly use the Student object wherever required.

**There are two ways to perform Dependency Injection in Spring framework-**

- **By Constructor :** Constructor is used by IOC Container to set the fields of object.

- **By Setter method :** Setter method is used by IOC Container to set the fields of the object.

## Setter Injection



```

class Student
{
    id,name,address


    setId(id){ }

    setName(name){ }

    setAddress(address){ }

}

```




```

class Address
{
    street,city,state,country
    setStreet(street)
    setCity(city)
    setState(state)
    setCountry(country)
}

```

## Constructor Injection



```


class Student
{
    String id,name,address;

    Student(id,name,address)
    {

    }

}

```



```

class Address
{
    String street,city,state,country;
    Address(street,city,state,country)
    {

    }

}

```

## Configuration File

Similar to hibernate, in spring also we will have a configuration file(xml file) . In that file we will provide all the informations about beans to the IOC Container.

**In simple , Configuration file is a file where we declare all the beans and its dependency.**

The basic structure of the xml file is like-

```
<beans>
    <bean></bean>
    <bean></bean>
    <bean></bean>
    ...
</beans>
```

## **Different data types** **(Dependencies)**

While injecting a dependency into some object,IOC container first checks which type of data to be injected.

Different type of data are handles in a different way by IOC Container.

The major data types are as follow :

## 1)Primitive Data Types

Byte, short, char, int, float, double, long, boolean

## 2)Collection Types

List, Set, Map and Properties

## 3)Reference Types/User Defined type

Student and Address

# Practical

(Injecting Primitive type data)

- Create maven project

We create maven project using archetype:

```
maven-archetype-quickstart
```

- Adding dependencies : spring core, spring context

```
<dependency>
  <groupId>org.springframework</groupId>
  <artifactId>spring-core</artifactId>
  <version>5.2.3.RELEASE</version>
</dependency>

<dependency>
  <groupId>org.springframework</groupId>
  <artifactId>spring-context</artifactId>
  <version>5.2.3.RELEASE</version>
```

</dependency>

- **Creating beans( Java Pojo)**

**Java beans(POJO)** are the simple classes( Like entity classes in hibernate) which contains variable fields, getter setters and constructors etc.

```
package com.rohit.SpringCore;
public class Student
{
    private int studentId;
    private String studentName;
    private String studentAddress;
    public int getStudentId() {
        return studentId;
    }
    public void setStudentId(int studentId) {
        this.studentId = studentId;
        System.out.println("Setting student Id");
    }
    public String getStudentName() {

        return studentName;
    }
    public void setStudentName(String studentName) {
        this.studentName = studentName;
        System.out.println("Setting student name");
    }
    public String getStudentAddress() {
        return studentAddress;
    }
    public void setStudentAddress(String studentAddress) {
        this.studentAddress = studentAddress;
        System.out.println("Setting student address");
    }

    @Override
    public String toString() {
        return "Student [studentId=" + studentId + ", studentName=" +
studentName + ", studentAddress=" + studentAddress
        + "]";
    }
}
```



- **Creating configuration file (config.xml)**

In the configuration file we need to tell about all the beans that we are using in the Project.

```
<?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"
       xmlns:p="http://www.springframework.org/schema/p"
       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.xsd">

<!--this is our beans -->
    <bean class="com.rohit.SpringCore.Student" name="student1">
        <property name="studentId">
            <value>22344</value>
        </property>
        <property name="studentName">
            <value>Rohit Pandey</value>
        </property>
        <property name="studentAddress">
            <value>Kolkata</value>
        </property>
    </bean>
</beans>
```

Here we are using property-value tag to tell the spring about the values of data fields,so this is called as Setter injection

By seeing this configuration file IOC Container will get to know which types of objects are to be created which which properties.

- **Creating Main class : which can pull the object and use**

```
package com.rohit.SpringCore;
import org.springframework.context.ApplicationContext;
import
org.springframework.context.support.ClassPathXmlApplicationContext;
public class App
{
    public static void main( String[] args )
    {
        System.out.println( "Hello World!" );
        ApplicationContext context=new
ClassPathXmlApplicationContext("config.xml");
        Student student1=(Student)context.getBean("student1");
        System.out.println(student1);
    }
}
```

In the above java file we have created object of Applicationcontext using **ClassPathXmlApplicationContext**.

By using **getBean("Bean name")** method we can fetch the values and methods of the java bean.

In the above example we have used injected **primitive type data** as the type of values given to the fields of bean class is int ,String etc(Primitive).

**Note :** In the configuration file we can inject the value of fields of bean class in three ways using setter injection: -

### 1)Using value element(tag)

```
<bean class="com.rohit.SpringCore.Student" name="student1">
    <property name="studentId">
        <value>22344</value>
    </property>
    <property name="studentName">
        <value>Rohit Pandey</value>
    </property>
</bean>
```

```
    </property>
    <property name="studentAddress">
      <value>Kolkata</value>
    </property>
  </bean>
```

## 2)Using value attribute

```
<bean class="com.rohit.SpringCore.Student" name="student1">
  <property name="studentId" value="22344"/>
  <property name="studentName" value="Rohit Pandey"/>
  <property name="studentAddress" value="Kolkata"/>
</bean>
```

## 3)Using p-schema

```
<bean class="com.rohit.SpringCore.Student" name="student1"
p:studentId="25632" p:studentName="Utkarsh Kumar"
p:studentAddress="Ranchi" />
```

This will only work if p-schema is included in the namespace of the configuration file.

## Injecting Collection type data

Using the spring configuration file we can also inject collection types if required.

### List:

```
<property name="">
  <list>
    <value>10</value>
    <value>288</value>
    <value>17.3</value>
    <value>Rohit pandey</value>
    <null/>
  </list>
</property>
```

## Set:

```
<property name="">
  <set>
    <value>10</value>
    <value>288</value>
    <value>17.3</value>
    <value>Rohit pandey</value>
    <null/>
  </set>
</property>
```

## Map:

```
<property name="">
  <map>
    <entry key="java" value="2 months"/>
    <entry key="python" value="1 months"/>
    <entry key="c++" value="2 months"/>
  </map>
</property>
```

## Properties :

```
<property name="">
  <props key="name">Rohit</props>
  <props key="RollNumber">120</props>
</property>
```

## Example :

We have a bean class Emp

```
package Collection;

import java.util.List;
import java.util.Map;
import java.util.Set;
```

```

public class Emp
{
    private String name;
    private List<String> phones;
    private Set<String> addresses;
    private Map<String, String> courses;
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    public List<String> getPhones() {
        return phones;
    }
    public void setPhones(List<String> phones) {
        this.phones = phones;
    }
    public Set<String> getAddresses() {
        return addresses;
    }
    public void setAddresses(Set<String> addresses) {
        this.addresses = addresses;
    }
    public Map<String, String> getCourses() {
        return courses;
    }
    public void setCourses(Map<String, String> courses) {
        this.courses = courses;
    }
}

```

In the configuration file we have injected values of the fields present in the bean class.

```

<bean class="Collection.Emp" name="emp1">
    <property name="name" value="Rohit"/>
    <property name="phones">
        <list>
            <value>9123114708</value>
            <value>8617479410</value>
            <value>7667187559</value>
            <value>7631070087</value>
            <value>8229819761</value>
        </list>
    
```

```

    </property>
    <property name="addresses">
        <set>
            <value>Kolkata</value>
            <value>Ranchi</value>
            <value>Delhi</value>
            <value>Hyderabad</value>
            <value>Punji</value>
        </set>
    </property>
    <property name="courses">
        <map>
            <entry key="java" value="2 months"/>
            <entry key="python" value="1 months"/>
            <entry key="c++" value="2 months"/>
        </map>
    </property>
</bean>

```

Test.java file contains the main method for testing the program.

```

package Collection;

import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

public class Test {
    public static void main(String[] args) {
        ApplicationContext context=new
ClassPathXmlApplicationContext("config.xml");
        Emp emp=(Emp)context.getBean("emp1");
        System.out.println(emp.getName());
        System.out.println(emp.getPhones());
        System.out.println(emp.getAddresses());
        System.out.println(emp.getCources());
    }
}

```

## Injecting Reference type

This type of injection is the most important and most widely used injection with large number of real life scope of usage.

Suppose we have a class B

```
package Reference;

public class B {
    private int y;

    public int getY() {
        return y;
    }

    public void setY(int y) {
        this.y = y;
    }

    @Override
    public String toString() {
        return "B [y=" + y + "]";
    }
}
```

We have another class A in which object of B is used as a field

```
package Reference;

public class A {
    private int x;
    private B ob;
    public int getX() {
        return x;
    }
    public void setX(int x) {
        this.x = x;
    }
    public B getOb() {
        return ob;
    }
    public void setOb(B ob) {
        this.ob = ob;
    }
    @Override
    public String toString() {
        return "A [x=" + x + ", ob=" + ob + "]";
    }
}
```

Now ,In configuration file ,Similar to primitive type we have three ways to inject any reference type object to any other object.

### Using ref tag:

```
<property name="ob">
    <ref bean="bref"/>
</property>
```

### Using ref attribute

```
<property name="ob" ref="bref"/>
```

### Using p schema

```
<bean class="Reference.A" name="aref" p:x="33" p:ob-ref="bref" />
```

### Our **config.xml** file

```
<bean class="Reference.B" name="bref">
    <property name="y" value="90"/>
</bean>

<bean class="Reference.A" name="aref" p:x="33" p:ob-ref="bref" />
```

### Test.java file

```
package Reference;

import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

public class Test {
    public static void main(String[] args) {
        ApplicationContext context=new
ClassPathXmlApplicationContext("config.xml");
        A a=(A)context.getBean("aref");
        System.out.println(a);
    }
}
```



# Dependency Injection by Constructor

We can inject the dependency by constructor. The **<constructor-arg>** subelement of **<bean>** is used for constructor injection.

Let we have a class Person

## Person.java

```
package ConstructionInjection;

public class Person
{
    private String name;
    private int personId;
    public Person(String name, int personId)
    {
        this.name = name;
        this.personId = personId;
    }
    public Person()
    {
        super();
    }
    @Override
    public String toString() {
        return "Person [name=" + name + ", personId=" + personId + "]";
    }
}
```

Configuration File

## Config.xml

```
<bean class="ConstructionInjection.Person" name="person">
    <constructor-arg value="Rohit"/>
    <constructor-arg value="25" />
</bean>
```

Again here, we have three options for setting the value : - **value tag**, **value attribute** and **c schema**.

Our Test file

## Test.java

```
package ConstructionInjection;

import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

public class Test {
    public static void main(String[] args) {
        ApplicationContext context=new
        ClassPathXmlApplicationContext("config.xml");
        Person person=(Person)context.getBean("person");
        System.out.println(person);
    }
}
```

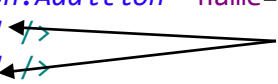
In the similar fashion we can also inject *reference type* values by using *ref* attribute.

```
<bean class="ConstructionInjection.Certificate" name="certi"
c:name="Java">
</bean>
<bean class="ConstructionInjection.Person" name="person">
    <constructor-arg value="Rohit" />
    <constructor-arg value="25" />
    <constructor-arg ref="certi" />
</bean>
```

## Ambiguity problem in Constructor injection

Suppose we have made a configuration for bean like this

```
<bean class="ConstructionInjection.Addition" name="add">
    <constructor-arg value="12" />
    <constructor-arg value="34" />
</bean>
```



Strings

The value injected using <constructor-arg> is by default considered as string value.

So while injecting the value in the Bean ,the IOC container will search for the constructor with string value.

If that type of constructor is not available then only the IOC container will look for other constructor.

If multiple constructor are there and none of them are of string type then the constructor which came first in the Bean class will be taken into consideration.

### **Example :**

```
package ConstructionInjection;

public class Addition {
    private int a;
    private int b;

    public Addition(double a,double b)
    {
        this.a=(int)a;
        this.b=(int)b;
        System.out.println("Constructor : double,double");
    }
    public Addition(int a, int b) {
        super();
        this.a = a;
        this.b = b;
        System.out.println("Constructor : int,int");
    }
    public void doSum()
    {
        System.out.println("a : "+a);
        System.out.println("b : "+b);
        System.out.println("Sum is "+ (this.a+this.b));
    }
}
```

Here this constructor will be considered as it comes first

So we can conclude that if multiple constructor are there then it creates an ambiguity problem.

To solve this ambiguity problem, we have to use an attribute “**type**” to tell the IOC container clearly about which type of value it is.

```
<constructor-arg value="12" type="int" />  
<constructor-arg value="34" type="int" />
```

Now the value 12 and 34 will always be considered as int value, so **public Addition(int a, int b)** constructor will be called ,irrespective of the order in which the they are written in the bean file.

In the above example, value 12 will be mapped with a and 34 with b.

a=12                      b=34

If we want reverse , i.e a=34 and b=12 then we can use another attribute “index”.

If we write **public Addition(int a, int b)** then

index 0                      index 1

```
<constructor-arg value="12" type="int" index="1"/>  
<constructor-arg value="34" type="int" index="0"/>
```

**This will result    a=34        b=12**

# Life Cycle Methods of Spring Beans

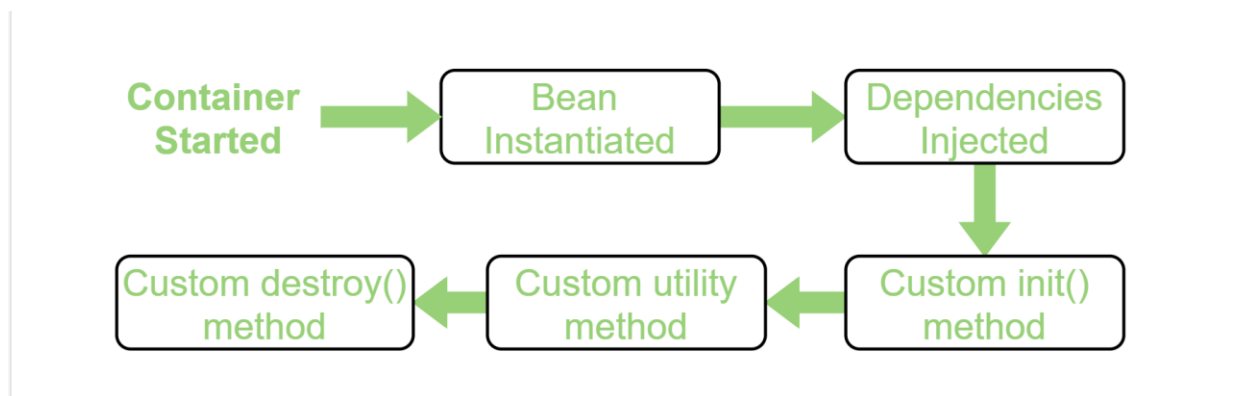
- Bean life cycle is managed by the spring container.
- When we run the program then, first of all, the spring container gets started.
- After that, the container creates the instance of a bean as per the request and then dependencies are injected.
- And finally, the bean is destroyed when the spring container is closed.

The spring provides two important methods to every bean

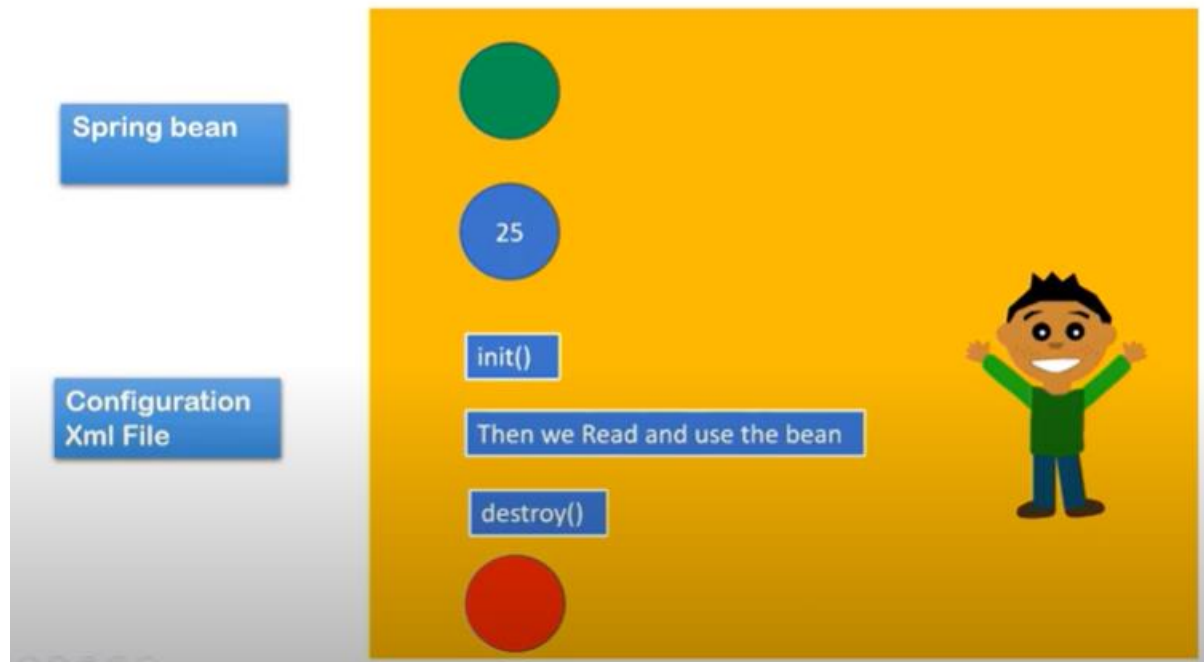
1. `public void init()`
2. `public void destroy()`

If we want to execute some code on the bean instantiation such as initialization code ,loading config,connecting DB,webservice etc. then we can write that code inside the custom **init()** method

And If we want to execute some code just after closing the spring container, such as cleanup code, then we can write that code inside the custom **destroy()** method



So we give two things as input to the spring container- Spring beans and Configuration XML file.



## Ways to implement the life cycle of a bean

Spring provides three ways to implement the life cycle of a bean-

- XML
- Spring Interface
- Annotation

## Implementing Life Cycle methods using XML

Suppose we have a class Samosa as shown below-

```
package lifeCycle;

public class Samosa
{
    private double price;
    public double getPrice() {
        return price;
    }
    public void setPrice(double price) {
        System.out.println("Setting price");
        this.price = price;
    }
}
```

```

    }
    @Override
    public String toString() {
        return "Samosa [price=" + price + "]";
    }

    public void init()
    {
        System.out.println("Inside init method..");
    }
    public void destroy()
    {
        System.out.println("Inside destroy method");
    }
}

```

Here we have created two custom methods-init() and destroy()

Now in the configuration file

```

<bean class="lifeCycle.Samosa" name="samosa" init-method="init"
destroy-method="destroy">
    <property name="price" value="7.5"/>
</bean>

```

Here we can see that we have used two attributes init() and destroy() which is used to map with the custom init() and destroy() method declared in Bean class.

## Test.java

```

package lifeCycle;
import javax.naming.Context;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.AbstractApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

public class Test {
    public static void main(String[] args) {
        AbstractApplicationContext context=new
ClassPathXmlApplicationContext("config.xml");
        Samosa samosa=(Samosa) context.getBean("samosa");
        System.out.println(samosa);
        context.registerShutdownHook();
    }
}

```

Here we are using a method **registerShutdownHook()** which tells the container that I have a method which has to be executed before the shut down of context.

This method is present in the **AbstractApplicationContext** and not in **ApplicationContext** so here we have used **AbstractApplicationContext**.

### **Output:**

```
Setting price  
Inside init method..  
Samosa [price=7.5]  
Inside destroy method
```

## **Implementing Life Cycle methods using Interfaces**

Instead of using xml file for configuring init() and destroy () method we can also use Interface technique.

We have two interfaces which provides us this facility-

**InitializingBean** : This Interface contains a method `afterPropertiesSet()` which acts as `init()` method. Whatever we want to write in `init()` method we can write it here.

**DisposableBean** : This Interface contains a method `destroy()` where we can write our custom cleanup code as required.

### **Example:**

```
package lifeCycle;  
  
import org.springframework.beans.factory.DisposableBean;  
import org.springframework.beans.factory.InitializingBean;
```



```

public class Pepsi implements InitializingBean ,DisposableBean
{
    private double price;
    public double getPrice() {
        return price;
    }
    public void setPrice(double price) {
        this.price = price;
    }
    @Override
    public String toString() {
        return "Pepsi [price=" + price + "]";
    }
    public void afterPropertiesSet() throws Exception {
        //works as init() method
        System.out.println("Taking pepsi");
    }
    public void destroy() throws Exception {
        //works as destroy method
        System.out.println("Going to put bottle back to shop");
    }
}

```

## Configuration file :

Here we just need to inject the values of the fiels and not need to configure init() or destroy() methods.

```

<bean class="LifeCycle.Pepsi" name="pepsi">
    <property name="price" value="45.5"/>
</bean>

```

## Test.java

```

package lifeCycle;

import javax.naming.Context;

import org.springframework.context.ApplicationContext;
import org.springframework.context.support.AbstractApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

public class Test {
    public static void main(String[] args) {

```

```

        AbstractApplicationContext context=new
        ClassPathXmlApplicationContext("config.xml");

        Pepsi pepsi=(Pepsi)context.getBean("pepsi");
        System.out.println(pepsi);

        context.registerShutdownHook();
    }
}

```

## **Implementing Life Cycle methods using Annotations**

We can also use annotations for implementing life cycle methods.

We have two annotations-

- `@PostConstruct` : Used on `init()`
- `@PreDestroy` : Used on `destroy()`

But both of these annotations are present in javaEE which is not by default available for latest java versions .So for using those annotations we have to first use a dependency in the pom.xml file.

```

<dependency>
    <groupId>javax.annotation</groupId>
    <artifactId>javax.annotation-api</artifactId>
    <version>1.3.2</version>
</dependency>

```

Now we can use those annotations in our bean class as follow.

### **Example.java**

```

package lifeCycle;

import javax.annotation.PostConstruct;
import javax.annotation.PreDestroy;

```

```

public class Example {
    private String subject;

    public String getSubject() {
        return subject;
    }

    public void setSubject(String subject) {
        this.subject = subject;
    }

    @Override
    public String toString() {
        return "Example [subject=" + subject + "]";
    }

    @PostConstruct
    public void init()
    {
        System.out.println("Init method");
    }

    @PreDestroy
    public void destroy()
    {
        System.out.println("Destroy method");
    }
}

```

## Test.java

```

package lifeCycle;

import javax.naming.Context;

import org.springframework.context.ApplicationContext;
import org.springframework.context.support.AbstractApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

public class Test {
    public static void main(String[] args) {
        AbstractApplicationContext context=new
ClassPathXmlApplicationContext("config.xml");

        Example example=(Example)context.getBean("example");
        System.out.println(example);

        context.registerShutdownHook();
    }
}

```

## Config.xml

```
<context:annotation-config/>
<!--this is our beans -->
<bean class="lifeCycle.Example" name="example">
  <property name="subject" value="Java"/>
</bean>
```

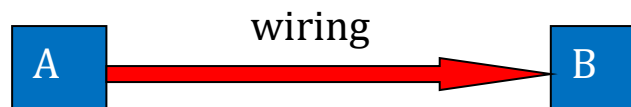
Used for  
activating the annotations.

## Autowiring in Spring

Autowiring is a feature of spring framework in which spring container inject the dependencies automatically.

Autowiring cannot be used to inject primitive and string values. It works with reference only.

When one object is dependent on another object then we have to link those two objects. This linking is called as **wiring**.



Till now we have done wiring manually using **ref** attribute in xml file like **<bean ref="" />**

Now instead of doing manually, spring provides us a feature of autowiring. Here programmer is not responsible to do linking in the xml file.

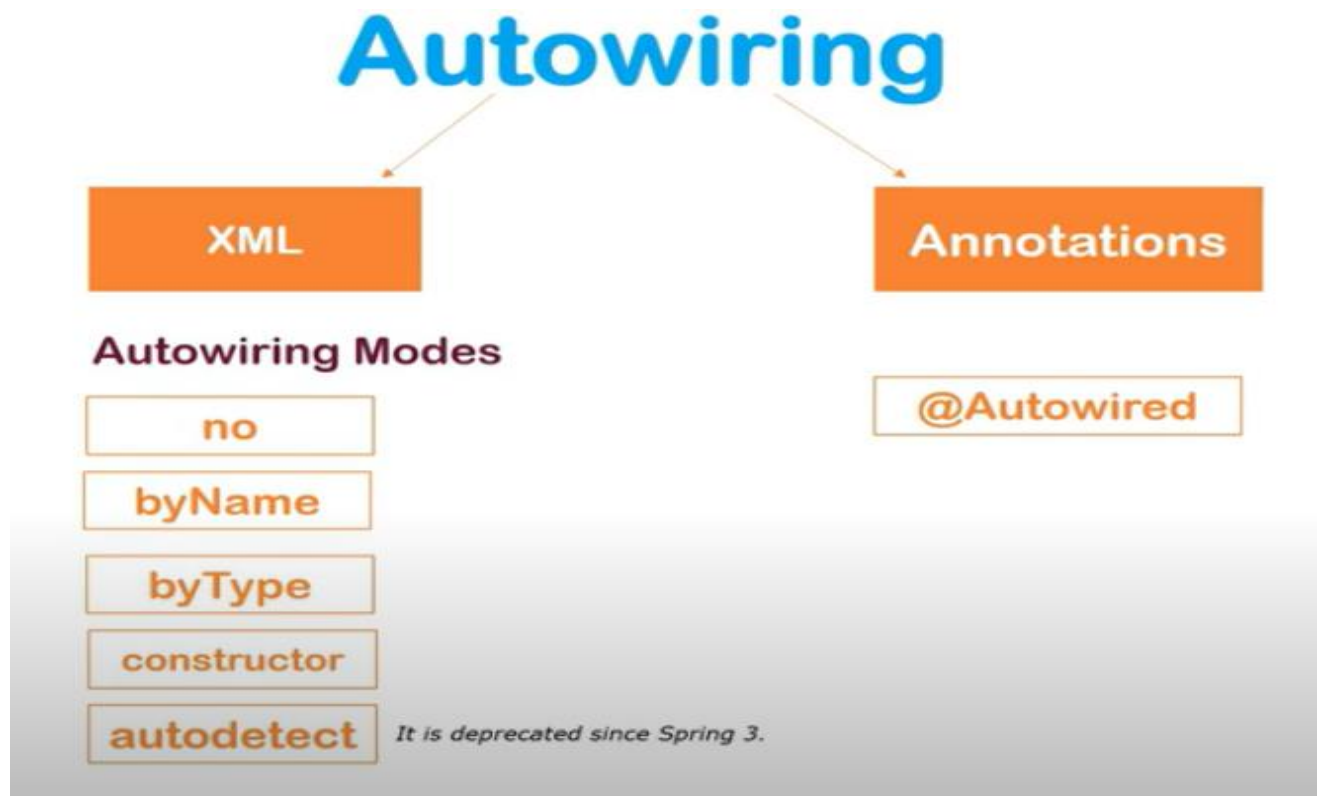
## Autowiring Advantages

- Automatic
- Less code

## Autowiring Disadvantages

- No control of programmer
- It cannot be used for primitive and string values.

Autowiring can be done in two ways-



No.	Mode	Description
1)	no	It is the default autowiring mode. It means no autowiring by default.
2)	byName	The byName mode injects the object dependency according to name of the bean. In such case, property name and bean name must be same. It internally calls setter method.
3)	byType	The byType mode injects the object dependency according to type. So property name and bean name can be different. It internally calls setter method.
4)	constructor	The constructor mode injects the dependency by calling the constructor of the class. It calls the constructor having large number of parameters.
5)	autodetect	It is deprecated since Spring 3.

## **Autowiring using XML**

Suppose we have a class Address

### **Address.java**

```
package Autowiring;

public class Address {
    private String street;
    private String city;
    public String getStreet() {
        return street;
    }
    public void setStreet(String street) {
```

```

        this.street = street;
    }
    public String getCity() {
        return city;
    }
    public void setCity(String city) {
        this.city = city;
    }
    @Override
    public String toString() {
        return "Address [street=" + street + ", city=" + city + "]";
    }
}

```

}  
 We have another class

## Employee.java

```

package Autowiring;

public class Employee {
    private Address address;

    public Address getAddress() {
        return address;
    }
    public void setAddress(Address address) {
        System.out.println("Setting value");
        this.address = address;
    }
    public Employee() {
        super();
    }
    public Employee(Address address) {
        System.out.println("Inside Constructor");

        this.address = address;
    }
    @Override
    public String toString() {
        return "Employee [address=" + address + "]";
    }
}

```

We can see that Employee class is dependent on Address class i.e. for creating an Employee object we need Address object injected in it.

Here we are using autowiring so this injection will be done automatically.

We have different modes in which we can use Autowiring

### 1) byName autowiring mode

In case of byName autowiring mode, bean id and reference name must be same. It internally uses setter injection.

```
<bean class="Autowiring.Address" name="address">
  <property name="street" value="GT Road"/>
  <property name="city" value="Delhi"/>
</bean>

<bean class="Autowiring.Employee" name="emp" autowire="byName" />
```

This name "address" must be same as the name of Field Addresss in Employee.java file.

**private Address address;**

But, if you change the name of bean, it will not inject the dependency.

### 2) byType autowiring mode

In case of byType autowiring mode, bean id and reference name may be different. But there must be only one bean of a type. It internally uses setter injection.



```
<bean class="Autowiring.Address" name="add1">
  <property name="street" value="GT Road"/>
  <property name="city" value="Delhi"/>
</bean>

<bean class="Autowiring.Employee" name="emp" autowire="byType" />
```

If you have multiple bean of one type, it will not work and throw exception.

### 3)constructor autowiring mode

In case of constructor autowiring mode, spring container injects the dependency by highest parameterized constructor.

If you have 3 constructors in a class, zero-arg, one-arg and two-arg then injection will be performed by calling the two-arg constructor.

```
<bean class="Autowiring.Address" name="addr">
  <property name="street" value="GT Road"/>
  <property name="city" value="Delhi"/>
</bean>

<bean class="Autowiring.Employee" name="emp" autowire="constructor" />
```

## Autowiring using Annotation

The **@Autowired** annotation provides a better way to do autowiring.

The **@Autowired** annotation can be used in three ways in the dependent bean class.

- With the property
- With the setter method
- With the constructor

## @Autowired on Properties

You can use **@Autowired** annotation on properties to get rid of the setter methods. When you will pass values of autowired properties using <property> Spring will automatically assign those properties with the passed values or references. So you don't need setter methods in this case.

### Address.java

```
package AutoAnnotation;

public class Address {
    private String street;
    private String city;
    public String getStreet() {
        return street;
    }
    public void setStreet(String street) {
        this.street = street;
    }
    public String getCity() {
        return city;
    }
    public void setCity(String city) {
        this.city = city;
    }
    @Override
    public String toString() {
        return "Address [street=" + street + ", city=" + city + "];"
    }
}
```

### Employee.java

```
package AutoAnnotation;
import org.springframework.beans.factory.annotation.Autowired;
public class Employee {
    @Autowired
    private Address address;

    @Override
    public String toString() {
        return "Employee [address=" + address + "];"
    }
}
```

## Configuration file

```
<bean class="AutoAnnotation.Address" name="address">
  <property name="street" value="GT Road"/>
  <property name="city" value="Delhi"/>
</bean>

<bean class="AutoAnnotation.Employee" name="emp" />
```

## Test.java

```
package AutoAnnotation;

import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

public class Test {
    public static void main(String[] args) {
        ApplicationContext context=new
ClassPathXmlApplicationContext("config.xml");
        Employee emp1=(Employee)context.getBean("emp");
        System.out.println(emp1);
    }
}
```

## @Autowired on Setter Methods

You can use @Autowired annotation on setter methods to get rid of the <property> element in XML configuration file. When Spring finds an @Autowired annotation used with setter methods, it tries to perform **byType** autowiring on the method.

In this case our Employee.java file will be

## Employee.java

```
package AutoAnnotation;
import org.springframework.beans.factory.annotation.Autowired;
public class Employee {
    private Address address;
    public Address getAddress() {
        return address;
    }
    @Autowired
    public void setAddress(Address address) {
        System.out.println("Setting value");
        this.address = address;
    }
}
```

```

    }
    @Override
    public String toString() {
        return "Employee [address=" + address + "]";
    }
}

```

## @Autowired on Constructors

You can apply @Autowired to constructors as well. A constructor @Autowired annotation indicates that the constructor should be autowired when creating the bean.

### Employee.java

```

package AutoAnnotation;
import org.springframework.beans.factory.annotation.Autowired;
public class Employee {
    private Address address;

    public Address getAddress() {
        return address;
    }
    @Autowired
    public Employee(Address address) {
        System.out.println("Inside Constructor");

        this.address = address;
    }
    @Override
    public String toString() {
        return "Employee [address=" + address + "]";
    }
}

```

## @Qualifier annotation

There may be a situation when you create more than one bean of the same type and want to wire only one of them with a property.

In such cases, you can use the **@Qualifier** annotation along with **@Autowired** to remove the confusion by specifying which exact bean will be wired.

```

<bean class="AutoAnnotation.Address" name="address2">
    <property name="street" value="GT Road"/>
    <property name="city" value="Delhi"/>

```

```

</bean>
<bean class="AutoAnnotation.Address" name="address3">
    <property name="street" value="GMG Road"/>
    <property name="city" value="Kolkata"/>
</bean>

<bean class="AutoAnnotation.Employee" name="emp" />

```

If we directly execute our code with this config file then it will give an error because of ambiguity as more than one bean of the same class is present there.

So we have to specify in the bean class itself that which bean to be considered. This is done using @Qualifier annotation

## Employee.java

```

package AutoAnnotation;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.beans.factory.annotation.Qualifier;

public class Employee {
    @Autowired
    @Qualifier("address10")
    private Address address;

    public Address getAddress() {
        return address;
    }

    public void setAddress(Address address) {
        System.out.println("Setting value");
        this.address = address;
    }

    public Employee() {
        super();
        // TODO Auto-generated constructor stub
    }

    public Employee(Address address) {
        super();
        this.address = address;
        System.out.println("Inside Constructor");
    }
}

```

```

    }

    @Override
    public String toString() {
        return "Employee [address=" + address + "]";
    }
}

```

## Standalone Collection(util-schema)

Suppose we have a class Person which has a list as field. Now while injecting value in the list ,In general we use

```

<property name="phones">
    <list>
        <value>9123114708</value>
        <value>8617479410</value>
        <value>7667187559</value>
        <value>7631070087</value>
        <value>8229819761</value>
    </list>
</property>

```

But there is one disadvantage with this approach, that this collection is not standalone but is a part of property of bean .So this is not reusable.

Instead of using this approach we can create standalone collection and then use it wherever required.

```

<!--creating standalone list -->
<util:list list-class="java.util.Vector" id="bestfriends">
    <value>Aman</value>
    <value>Rohit</value>
    <value>Deepak</value>
    <value>Arsh</value>
</util:list>

<!-- creating standalone map -->
<util:map map-class="java.util.HashMap" id="fee">
    <entry key="Java" value="2000"/>
    <entry key="Spring" value="8000"/>
    <entry key="Hibernate" value="2000"/>

```

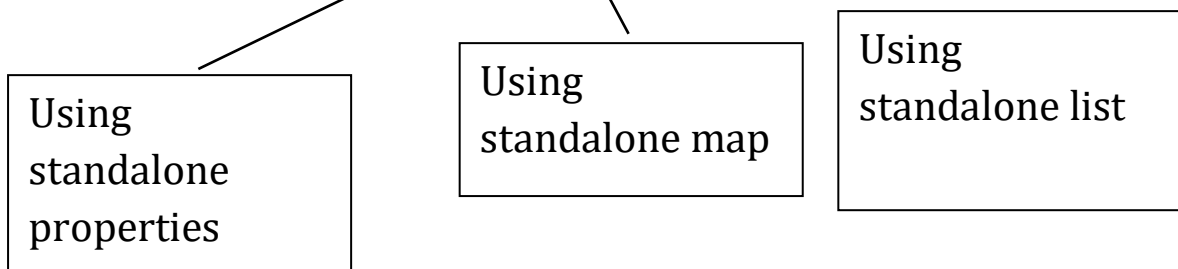
```

</util:map>

<!-- creating standalone properties -->
<util:properties id="dbconfig">
  <prop key="driver">com.mysql.cj.jdbc.Driver</prop>
  <prop key="username">root</prop>
  <prop key="password">76448</prop>
  <prop key="url">mysql:jdbc://localhost:3306/database</prop>
</util:properties>

<bean class="StandaloneCollection.Person" name="person">
  <property name="friends" ref="bestfriends"/>
  <property name="fee" ref="fee"/>
  <property name="property" ref="dbconfig"/>
</bean>

```



Now we can use the standalone collection (list,map,properties) created above to anywhere required.

## **Stereotype Annotations**

Till now, we have been doing the declaration of bean in the configuration (xml) file. But instead of it, we can also do it using annotations in the bean class.

For this we have to use

- @Component annotation which tells the IOC container that this class is a bean.
- @Value annotation with the properties which tells the value of the properties while injecting .

Suppose we have a class *Student*

## Student.java

```
package Strereo;

import org.springframework.beans.factory.annotation.Value;
import org.springframework.stereotype.Component;

@Component
public class Student {
    @Value("Rohit Pandey")
    private String name;
    @Value("Kolkata")
    private String city;
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    public String getCity() {
        return city;
    }
    public void setCity(String city) {
        this.city = city;
    }
    @Override
    public String toString() {
        return "Student [name=" + name + ", city=" + city + "];"
    }
}
```

**Note :** The bean created will have by default name same as the class name but starts with lowercase letter.

<u>Class Name</u>	<u>Bean name</u>
Student	student
Employee	employee
CollegeStudent	collegeStudents



In the configuration file we have to only declare the base package so that IOC container get to know where to search . We can do it as follow-

```
<context:component-scan base-package="Strereo"/>
```

## Test.java

```
package Strereo;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

public class Test {
    public static void main(String[] args) {
        ApplicationContext context=new
        ClassPathXmlApplicationContext("config.xml");
        Student student=context.getBean("student",Student.class);
        System.out.println(student);
    }
}
```

## Injecting collection using @Value

Suppose we have a collection field in our Bean class. Then we have to inject value to that collection using @Value annotation.

First of all we have to create a standalone collection in our configuration file

```
<util:list list-class="java.util.ArrayList" id="al">
    <value>Ranchi</value>
    <value>Kolkata</value>
    <value>Delhi</value>
    <value>Pune</value>
</util:list>
```

Now we can use the id of this collection in our Bean class as follow.

## Student.java

```
package Strereo;
import java.util.List;
```

```

import org.springframework.beans.factory.annotation.Value;
import org.springframework.stereotype.Component;

@Component("ob")
public class Student {
    @Value("Rohit Pandey")
    private String name;
    @Value("Kolkata")
    private String city;
    @Value("#{a1}")
    private List<String> address;

    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    public String getCity() {
        return city;
    }
    public void setCity(String city) {
        this.city = city;
    }
    @Override
    public String toString() {
        return "Student [name=" + name + ", city=" + city + ", address="
+ address + " ]";
    }
    public List<String> getAddress() {
        return address;
    }
    public void setAddress(List<String> address) {
        this.address = address;
    }
}

```

## Bean Scope

When we declare any bean (either by using xml or annotation) then there always a scope associate with it. Scope tells that how many number of objects will be created by IOC container.

There are five scope of bean provided in spring framework.

- Singleton
- Prototype
- Request
- Session
- global-session

We can specify the scope of bean in two ways-

**By using xml file** : Applicable when we declare bean in xml file

```
<bean class=" " name=" " scope=" " />
```

**By using annotation** : Applicable when we declare bean using annotation.

```
@Component
@Scope( " ")
Class Student
{

}
```

### **The singleton scope**

This is the by default scope of a bean. If a bean is declared as singleton then only one object will be created , and every time we ask for bean, IOC container will provides us the same object.

### **The prototype scope**

When we declare a bean as prototype then when we ask for any bean ,then every-time new object is created by IOC Container .

- Request, Session and global session are applicable in web applications only

## Student.java

```
package Strereo;
import java.util.List;

import org.springframework.beans.factory.annotation.Value;
import org.springframework.context.annotation.Scope;
import org.springframework.stereotype.Component;
```

```
@Component("ob")
```

```
@Scope("singleton") or @Scope("prototype")
```

```
public class Student {
    @Value("Rohit Pandey")
    private String name;
    @Value("Kolkata")
    private String city;
    @Value("#{a1}")
    private List<String> address;

    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    public String getCity() {
        return city;
    }
    public void setCity(String city) {
        this.city = city;
    }
    @Override
    public String toString() {
        return "Student [name=" + name + ", city=" + city + ", address=" +
address + " ]";
    }
    public List<String> getAddress() {
        return address;
    }
    public void setAddress(List<String> address) {
        this.address = address;
    }
}
```

# Spring Expression Language (SpEL)

Spring supports Parsing and executing expressions with the help of @Value annotations.

**Syntax :**        @Value("#{expression}")

The expression can be

- Classes, Variables, Methods, Constructors and Objects
- Symbols
- char, numerics, operators, keywords and special symbols which returns a value

**Example :**

- @Value("#{11+12}")
- @Value("#{8>6?88:55}")

We have a class Demo

## Demo.java

```
package SpEL;
```

```
import org.springframework.beans.factory.annotation.Value;
```

```
import org.springframework.stereotype.Component;
```

```
@Component
```

```
public class Demo
```

```
{
```

```
    @Value("#{10+5}")
```

```
    private int x;
```

```
    @Value("#{23+2}")
```

```
    private int y;
```

```
    @Override
```

```
    public String toString() {
```

```
        return "Demo [x=" + x + ", y=" + y + "];
```

```
    }
```

```
}
```

x will get value 15 and y will get 25



**Note :** We have another technique to solve expression language.

We have a class **SpELExpressionParser** which has a method **parseExpression()** which solves given expression to it .

### Test.java

```
package SpEL;

import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;
import org.springframework.expression.Expression;
import org.springframework.expression.spel.standard.SpelExpressionParser;

public class Test
{
    public static void main(String[] args)
    {
        SpelExpressionParser temp=new SpelExpressionParser();
        Expression exp= temp.parseExpression("33+69");
        System.out.println(exp.getValue());
    }
}
```

## More Example

**Static methods :  $T(\text{className}).\text{methodName}()$ Parameters**

**Static variables :  $T(\text{className}).\text{variableName}$**

```
package SpEL;
```

```
import org.springframework.beans.factory.annotation.Value;
```

```
import org.springframework.stereotype.Component;
```

```
@Component
```

```
public class Demo
```

```
{
```

```
    @Value("#{10+5}")
```

```
    private int x;
```

```
    @Value("#{23}")
```

```
    private int y;
```

```
    @Value("#{T(java.lang.Math).sqrt(25)}")
```

```
    private double z;
```

```
    @Value("#{T(java.lang.Math).E}")
```

```
    private double e;
```

```
    @Value("#{T(java.lang.Math).PI}")
```

```
    private double pi;
```

```
    @Value("#{new java.lang.String('Rohit Pandey')}")
```

```
    private String name;
```

```
    @Value("#{8>3}")
```

```
    private boolean isActive;
```

```
    @Override
```

```
    public String toString() {
```

```
        return "Demo [x=" + x + ", y=" + y + ", z=" + z + ", e=" + e + ", pi=" +  
pi + ", name=" + name + ", isActive=" + isActive + "];"
```

```
    }
```

```
}
```

resolving static method

resolving static variable

resolving object

resolving Boolean

# Removing Complete XML for Spring Configuration

Instead of using xml file for doing any type configuration, we can use only java for everything.

Suppose we have a Student class

## Student.java

```
package JavaConfig;

import org.springframework.stereotype.Component;

@Component
public class Student
{
    public void study()
    {
        System.out.println("Student is reading book");
    }
}
```

Here we are using @Component annotation for saying the IOC Container that this is a bean class.

Now instead of xml file,we can create a java file for configuration.

## Config.java

```
package JavaConfig;

import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;

@Configuration
@ComponentScan(basePackages = "JavaConfig")
public class Config {

}
```



**In the above java file we are using two annotations-**

**@Configuration** : For telling the spring container that this file is a configuration file.

**@ComponentScan(basePackages = "JavaConfig")** : To tell the spring container about the base package (Similar to component scan tag in xml file)

*Now in the Test. Java file we cannot use ClaaPathXMLApplicationContext class for getting the object of ApplicationContext as we are not using xml file here.*

So for getting the Object of ApplicationContext we have to use **AnnotationConfigApplicationContext** class here. And inside the constructor we have to pass the name of the configuration java class.

### **Test.java**

```
package JavaConfig;

import org.springframework.context.ApplicationContext;
import org.springframework.context.annotation.AnnotationConfigApplicationContext;

public class Test {
    public static void main(String[] args)
    {
        ApplicationContext context= new
AnnotationConfigApplicationContext(Config.class);
        Student student=context.getBean("student",Student.class);
        System.out.println(student);
        student.study();
    }
}
```

## @Bean Annotation

Till now we have been using @Component annotation for telling the IOC Container that this is a bean class. We can have another option.

In the configuration file ,we can create a method which will create object and return the object of bean. Like below-

```
package JavaConfig;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;

@Configuration
public class Config {
    @Bean
    public Student getStudent()
    {
        Student student =new Student();

        return student;
    }
}
```

The method will have annotation @Bean .This annotation will tell the spring container that, this method is returning the bean object.

Whenever we ask for the object of Student then this method will be called automatically and student object will be returned.

The @ComponentScan annotation is used to tell the IOC that which package contains the class with @Component annotation. Now here we are not using @Component so we also don't need the @ComponentScan annotation here.

**Student.java**

```

package JavaConfig;

import org.springframework.context.ApplicationContext;
import org.springframework.context.annotation.AnnotationConfigApplicationContext;

public class Test {
    public static void main(String[] args)
    {
        ApplicationContext context= new
AnnotationConfigApplicationContext(Config.class);
        Student student=context.getBean("getStudent",Student.class);
        System.out.println(student);
        student.study();
    }
}

```

## Config.java

```

package JavaConfig;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;

@Configuration
public class Config {
    @Bean
    public Student getStudent()
    {
        Student student =new Student();

        return student;
    }
}

```

## Test.java

```

package JavaConfig;

```

```
import org.springframework.context.ApplicationContext;
import
org.springframework.context.annotation.AnnotationConfigApplicationCont
ext;

public class Test {
    public static void main(String[] args)
    {
        ApplicationContext context= new
AnnotationConfigApplicationContext(Config.class);
        Student student=context.getBean("getStudent",Student.class);
        System.out.println(student);
        student.study();
    }
}
```

---