Spring Framework

**CORE**

<https://spring.io/>

The Spring Framework is an application framework and inversion of the control container for the Java platform. The framework's core features can be used by any Java application, but there are extensions for building web applications on top of the Java EE platform.

Spring was the company created by the founders of the Spring Framework to support and develop Spring and related projects. Originally incorporated as Interface 21, it was renamed Spring Source in 2008 to better reflect its main business.

Parent company Pivotal (now owned by VMware for $2.7 billion)

[Developer(s)](https://www.google.com/search?client=firefox-b-d&q=spring+framework+developers&sa=X&ved=2ahUKEwj9pKSp5vv5AhXQ7GEKHShAD8IQ6BMoAHoECGQQAg): [VMware](https://www.google.com/search?client=firefox-b-d&q=VMware&stick=H4sIAAAAAAAAAONgVuLUz9U3MCyxzDZdxMoW5lueWJQKAHlN1McWAAAA&sa=X&ved=2ahUKEwj9pKSp5vv5AhXQ7GEKHShAD8IQmxMoAXoECGQQAw)

[Repository](https://www.google.com/search?client=firefox-b-d&q=spring+framework+repository&sa=X&ved=2ahUKEwj9pKSp5vv5AhXQ7GEKHShAD8IQ6BMoAHoECGYQAg): [github.com](https://www.google.com/search?client=firefox-b-d&q=github.com&stick=H4sIAAAAAAAAAONgVuLSz9U3KKosKMhNX8TKlZ5ZklGapJecnwsAdm3QDBsAAAA&sa=X&ved=2ahUKEwj9pKSp5vv5AhXQ7GEKHShAD8IQmxMoAXoECGYQAw)/spring-projects/spring-framework

[License](https://www.google.com/search?client=firefox-b-d&q=spring+framework+license&sa=X&ved=2ahUKEwj9pKSp5vv5AhXQ7GEKHShAD8IQ6BMoAHoECGIQAg): [Apache License](https://www.google.com/search?client=firefox-b-d&q=Apache+License&stick=H4sIAAAAAAAAAONgVuLUz9U3MDTKjrdcxMrnWJCYnJGq4JOZnJpXnAoAzGCpBx4AAAA&sa=X&ved=2ahUKEwj9pKSp5vv5AhXQ7GEKHShAD8IQmxMoAXoECGIQAw) 2.0

[Initial release](https://www.google.com/search?client=firefox-b-d&q=spring+framework+initial+release&sa=X&ved=2ahUKEwj9pKSp5vv5AhXQ7GEKHShAD8IQ6BMoAHoECFkQAg): 1 October 2002; 19 years ago

[Platform](https://www.google.com/search?client=firefox-b-d&q=spring+framework+platform&sa=X&ved=2ahUKEwj9pKSp5vv5AhXQ7GEKHShAD8IQ6BMoAHoECFgQAg): [Java EE](https://www.google.com/search?client=firefox-b-d&q=Java+EE&stick=H4sIAAAAAAAAAONgVuLQz9U3SCo2q1jEyu6VWJao4OoKAIutXnEWAAAA&sa=X&ved=2ahUKEwj9pKSp5vv5AhXQ7GEKHShAD8IQmxMoAXoECFgQAw)

* The Spring Framework is an application framework and inversion of the control container for the Java platform. It's also known as the Dependency Injection framework.
* The framework's core features can be used by any Java application, but there are extensions for building web applications on top of the Java EE platform
* Spring is known as the ***framework of the frameworks***.
* Spring CORE is one of the Spring Modules and it provides basic fundamental support for SPRING.
* Spring framework makes the easy development of JAVA-EE web applications.
* Spring is known as the framework of the frameworks.
* Spring **CORE** is one of the Spring Modules and it provides basic fundamental support for SPRING.
* Spring framework makes the easy development of JAVA-EE web applications.

# Dependency injection

Dependency injection is a design pattern in which an object or function receives other objects or functions that it depends on.

A form of **Inversion of Control**, dependency injection aims to separate the concerns of constructing objects and using them, leading to loosely coupled programs.

## MVC Layers

Normally, we use the three-layer architecture, for example, the MVC architecture pattern, to organize our project. Below is the background of this pattern.

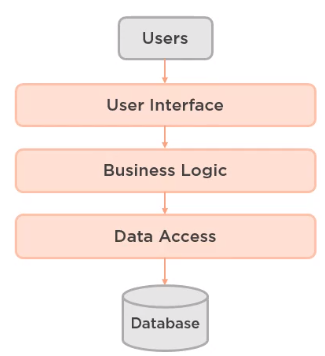


Figure MVC architecture

Below are the functionalities of each layer in the above image.

### User Interface / Web Service layer

This layer will manage all user interfaces that can be interacted with users. Or it can communicate with the other system by using SOAP API or Restful API.

After receiving requests from users, it will transfer requests to the below layers.

### Business Logic layer

This layer will process requests that satisfy all business rules. If requests violate some business rules, this layer will return the result for the WebService layer or response result to users.

If requests pass some rules, they will be pushed to the Data Access layer to persist in their states. Or this layer will call the Data Access layer to get data to check the information that it wants.

### Data Access layer

The Data Access layer will interact with the physical system such as the database or Redis, search engine. It will be used to save the requests’ states permanently. Or it will query SQL to get the result based on the need of the Business Logic layer.

We use some ORM frameworks such as Hibernate, and EclipseLink to map fields of the object to the table’s columns. This object is called entities.

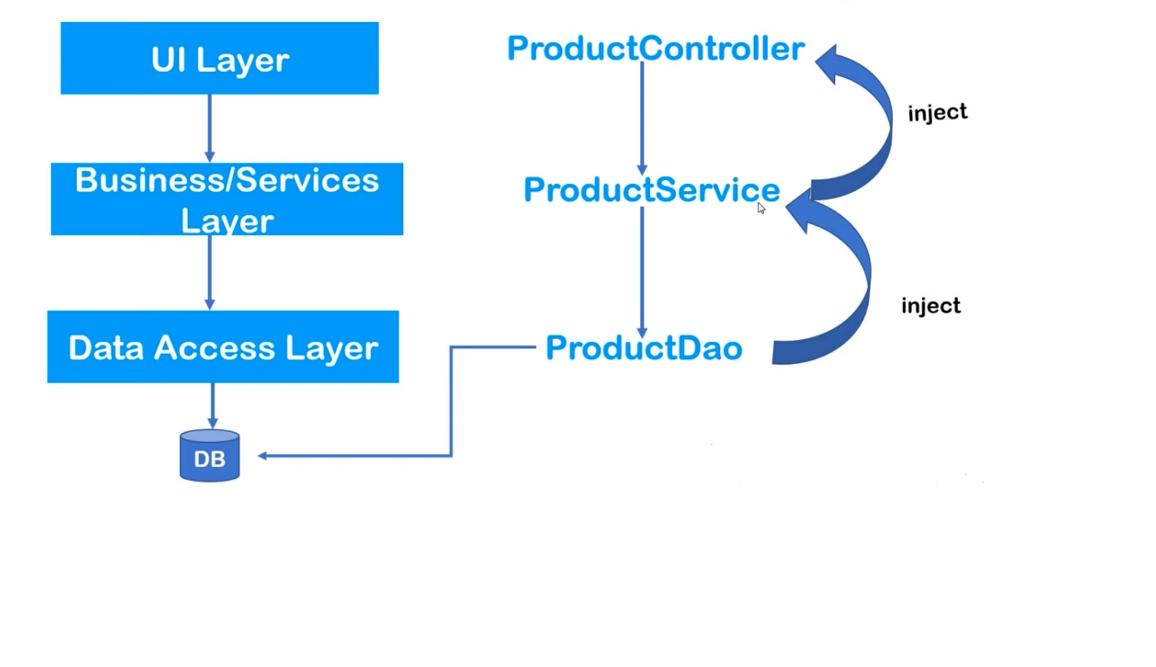


Figure Spring MVC using IOC

# Spring Modules

## Spring CORE

|  |  |  |  |
| --- | --- | --- | --- |
| **Core** | **Beans** | **Context** | **spEL** |
| * The fundamental part of Spring core * Provides IOC and dependency features. | | * inherits multiple features from beans module ie: factory design feature * internationalization * event propagation * transparent creation of context * resource loading * Provides JavaEE features like EJB, JMS, Basic Remoting | * A very powerful module that Queries El and manipulates object values at runtime. * Extension to EL is JSP |

* ***IOC can be taken as the Spring takes these responsibilities into their hands***
* ***creating the objects at runtime***
* ***managing the lifecycle***

|  |  |  |  |
| --- | --- | --- | --- |
| **AOP** | **Aspects** | **Instrumentation** | **Messaging** |
| * AOP:(Aspect-oriented programming) * Allows us to define method interceptors (Like some stuff we wanted to put before/after the method) * We can define pointcuts * Enables us to cleanly decouple the code. |  | * Provides class instrumentation support * class loader implementation | * Serves as a foundation for messaging-based applications. * We can map the message to our methods using appropriate annotation |

## Spring Data Integration

|  |  |  |  |
| --- | --- | --- | --- |
| **Spring JDBC** | **ORM** | **JMS** | **OXM** |
| * Provides an extra abstraction JDBC layer to ease out database connectivity and manage the error codes easily. * Cleaner API | * Spring ORM (Object Relational Mapping) enables us to integrate hibernate, JPA etc | * Java messaging service (JMS) * Produces and consumes messages | * Provides an object XML mapping * Castor, X-Stream, JAXB |

## Spring WEB

|  |  |  |  |
| --- | --- | --- | --- |
| **Web** | **Servlet** | **Portlet** | **WebSocket** |
| * Provides web-oriented integration feature * helpful for rest or web-based app | | * useful for portlet-based app | * contains HTTP client and web-related kinds of stuff |

## Spring TEST

|  |
| --- |
| **Provides J Unit/Mockito and testNG for unit/integration testing** |
| * provides mock objects for testing in isolation. |

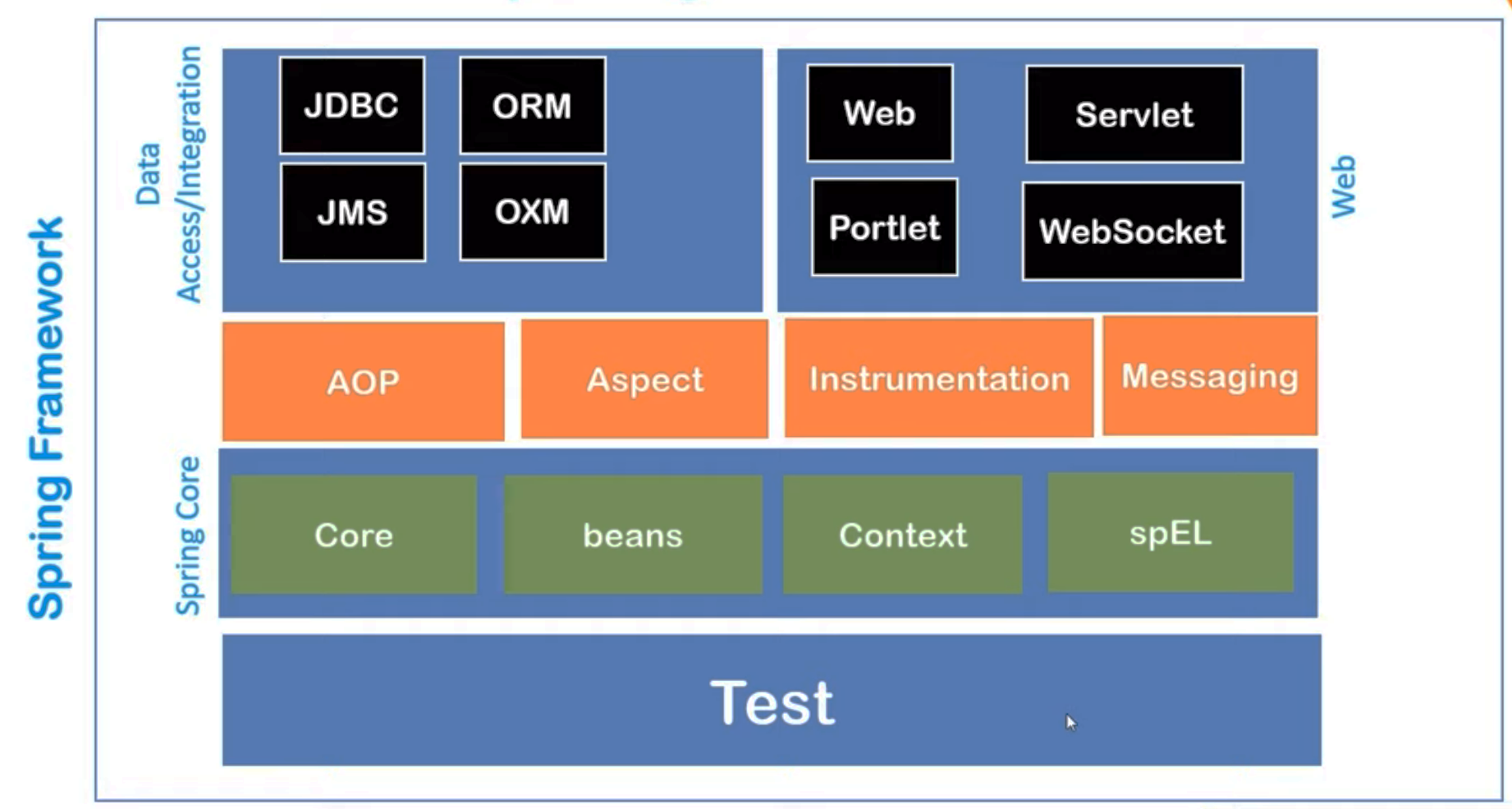


Figure : Spring Modules

# Spring IOC Container

## Spring IOC

|  |  |
| --- | --- |
| **About Spring IOC** | **Usage Instructions** |
| * Spring IOC is a predefined component available with SPRING * Helps in creating the object * Holds the created object in the memory and helps in DI * Holds account for Object life cycle * Provides ready-to-use object | * We have to instruct what beans or POJO classes it has to manage in the configuration XML file * Few objects have a dependency on other objects   which it will inject at suitable places accordingly.   * Need two things here:   + POJO classes   + XML speaking about dependency |

## Application Context

|  |  |
| --- | --- |
| **ApplicationContext** | **Usage Instructions** |
| * It represents our IOC container * We can the objects from IOC * It’s an interface that extends the bean factory * Since its an interface, we can’t create an object for it * So we will use and create an object of any of its derived sub-classes. | * AnnotationConfigAppicationContext   + Searches for annotation used on the bean * \*ClasspathXMLApplicatonContext   + Searches XML config in JAVA classpath * FileSystemXMLApplicatonContext   + Searches file system for the config file |

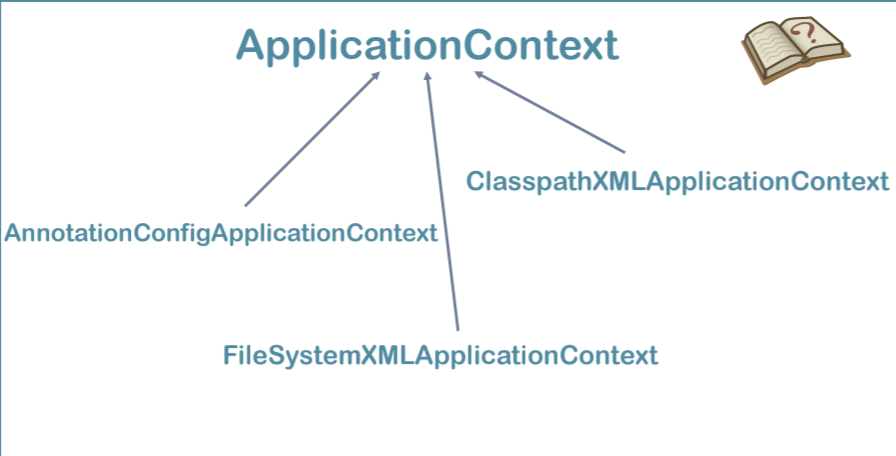


Figure - ApplicationContext Implementation

# DI with Spring

* IOC Container will create the object of Address and inject all its state
* It will also create an object of Student and inject its state and most important it will inject the Previous object of address at **RUNTIME**
* After this, we can request IOC for the beans of Student with all its dependencies full filled.
* Dependency can be injected in 2 ways:
  + Setter Injection (Property Injection)
  + Constructor Injection

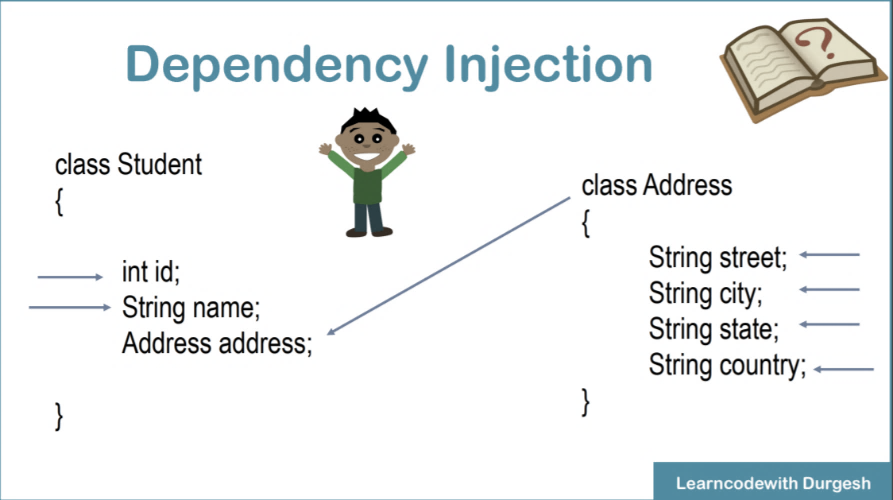


Figure Spring DI

*\*The boy is the IOC container here which will perform the injecting at Runtime.*

## Setter Injection

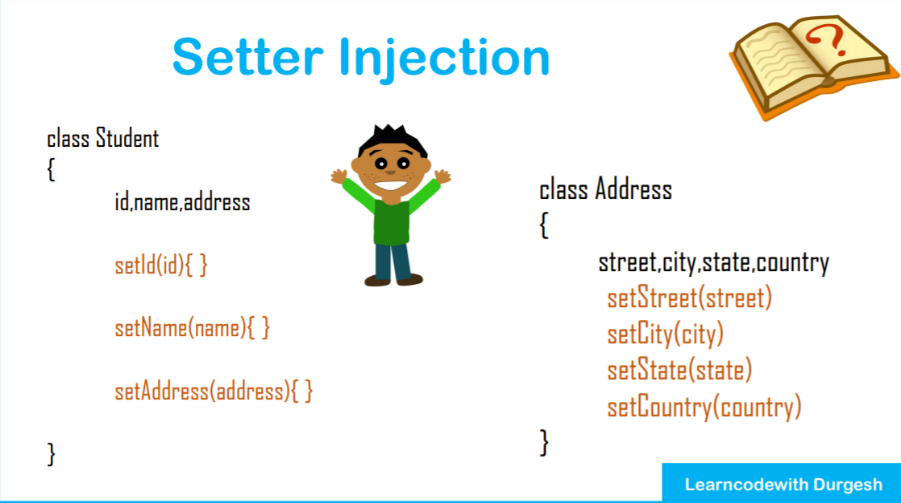


Figure Spring DI with SETTER injection

* While creating the object (Student) it will call the setters and inject all the values even the values of dependencies thus updating the state(instance variables).

## Constructor Injection

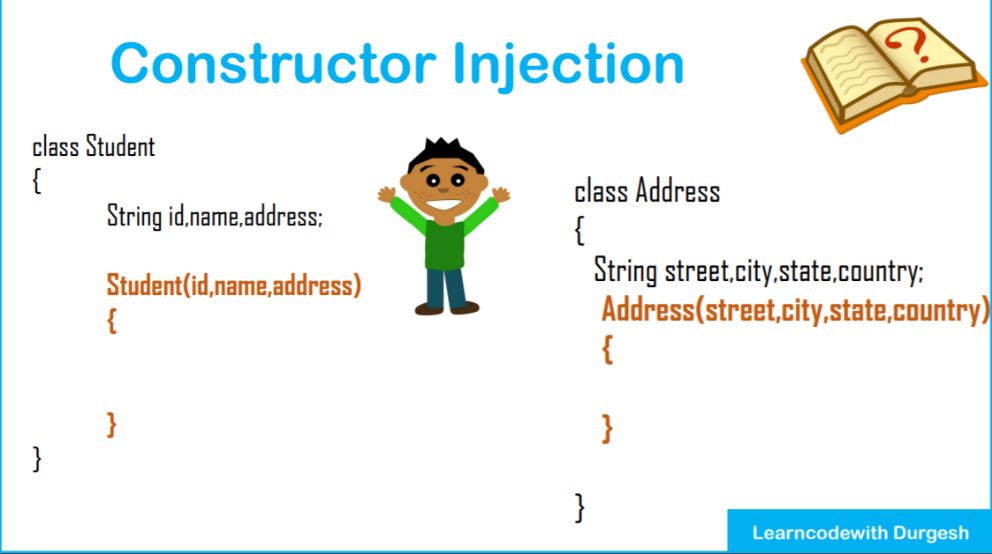
**

Figure Spring DI with CONSTRUCTOR injection

* In this case, while creating an object, it will use a constructor for updating the values and fulfilling the dependencies.

## Configuration File

### Java Bean

A JavaBean is just a [standard](http://www.oracle.com/technetwork/java/javase/documentation/spec-136004.html). It is a regular Java class, except it follows certain conventions:

1. All properties are private (use [getters/setters](http://en.wikipedia.org/wiki/Mutator_method))
2. A public [no-argument constructor](http://en.wikipedia.org/wiki/Nullary_constructor)
3. Implements [Serializable](http://docs.oracle.com/javase/8/docs/api/java/io/Serializable.html).

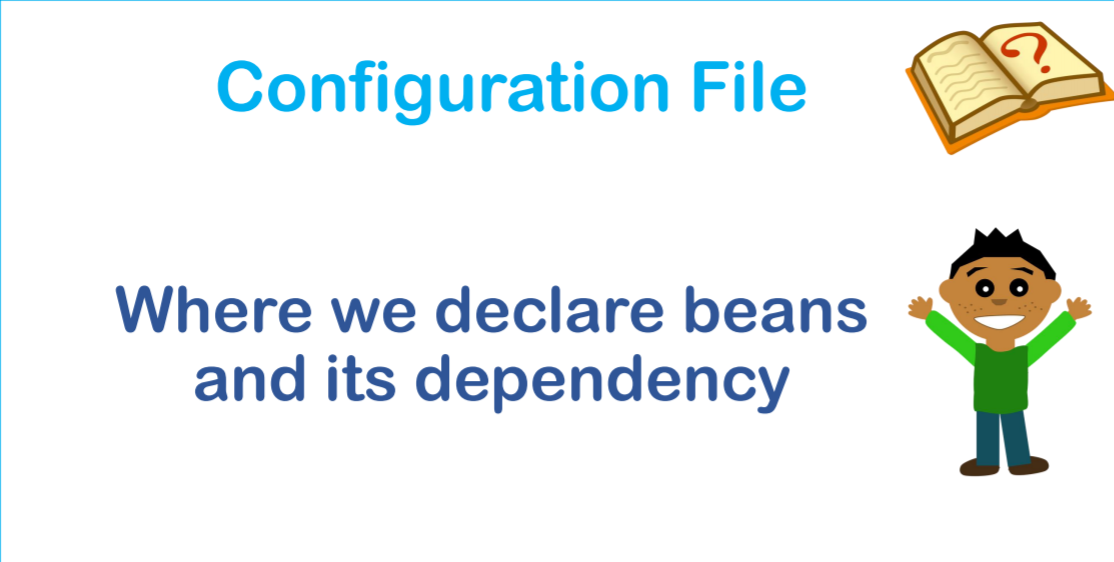
That's it. It's just a convention. Lots of libraries depend on it though.

For Serializable, from the [API documentation](http://docs.oracle.com/javase/8/docs/api/java/io/Serializable.html):

### Spring Beans

Spring beans run in a Spring IoC container. The programmer can configure via XML configuration files, annotations or a mix of both.

In Spring, if a bean constructor has simple-type or class/interface type parameters, values can be assigned as strings (as the <value> attribute of a constructor argument element in the former case and as an <idref> element of a constructor argument in the latter case) in a type-safe manner. Making references to other Spring beans (called collaborators; via the <ref> element in a constructor argument element) is dependency injection and is also typesafe. A dependency (collaborator bean) might have a constructor with injected parameters; those injected dependencies (ies) might have a constructor with parameters and so on. This scenario should ultimately terminate at injected dependency(ies) that are prototype beans that the container can instantiate by constructing.

**

* We will provide instructions for Spring (IOC) of the classes (beans and their dependencies)
* The classes which we provide to IOC are also known as Beans.
* After this IOC will be in a position to manage the beans, their state, and lifecycle.
* The configuration file will have *<bean> tags inside <beans> tag*.

1st tag

<?xml version="1.0" encoding="UTF-8"?>

<beans>

<bean> </bean>

</beans>

*\*While injecting it notices the data types*

## Some supported data types are:

|  |  |  |
| --- | --- | --- |
| **Primitive Data types** | **Collections** | **Reference Types** |
| * byte * short * char * int * long * float * double * boolean | * List * Set * Map * Properties | * Address * User Defined Class |

# Performing a Simple DI via Setter injection

## Getting Started

### Software

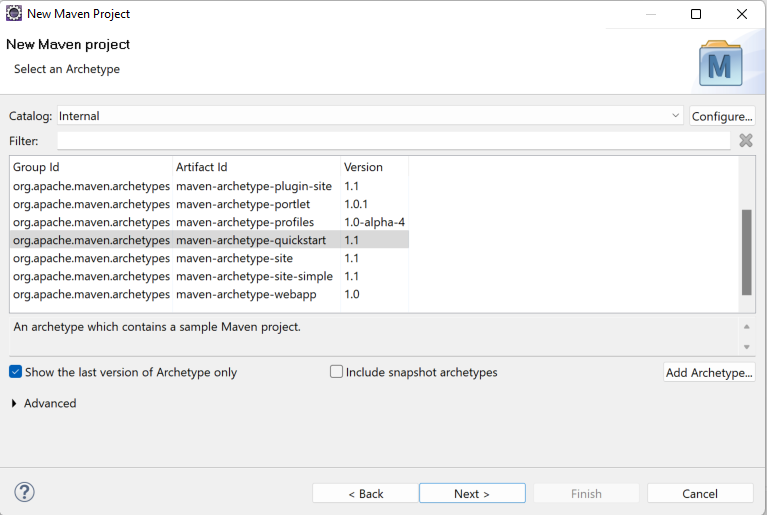
* Eclipse EE
* Tomcat
* My SQL
* SQL Client (D-Beaver, Data Gripper from JetBrains if you have a college id)

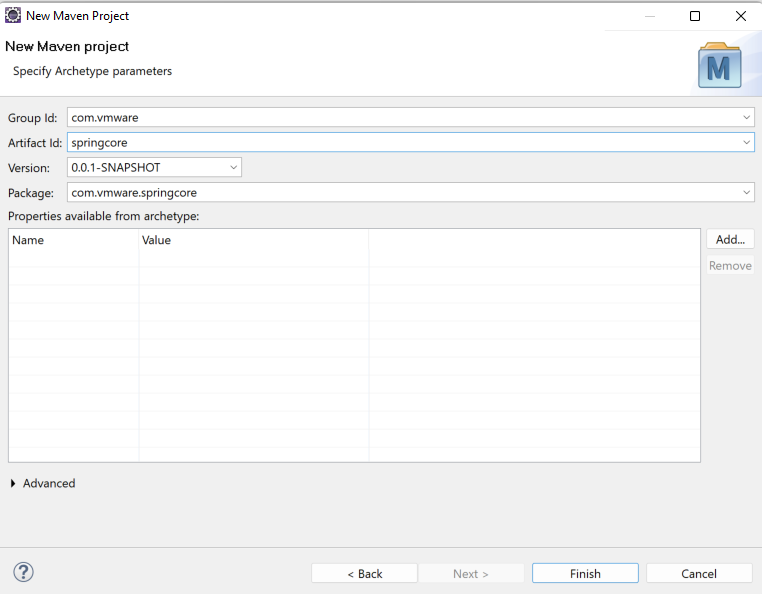
## Maven

Maven uses a set of identifiers, also called coordinates, to uniquely identify a project and specify how the project artefact should be packaged:

* ***groupId*** – a unique base name of the company or group that created the project
* ***artifactId*** – a unique name of the project
* ***version*** – a version of the project
* ***packaging*** – a packaging method (e.g. *WAR*/*JAR*/*ZIP*)

The first three of these (*groupId:artifactId:version*) combine to form the unique identifier and are the mechanism by which you specify which versions of external libraries (e.g. JARs) your project will use.





### Maven Archetype

#### What is an Archetype?

In short, Archetype is a Maven project templating toolkit. An archetype is defined as *an original pattern or model from which all other things of the same kind are made*. The names fit as we are trying to provide a system that provides a consistent means of generating Maven projects. The archetype will help authors create Maven project templates for users, and provides users with the means to generate parameterized versions of those project templates.

Using archetypes provides a great way to enable developers quickly in a way consistent with best practices employed by your project or organization. Within the Maven project, we use archetypes to try and get our users up and running as quickly as possible by providing a sample project that demonstrates many of the features of Maven while introducing new users to the best practices employed by Maven. In a matter of seconds, a new user can have a working Maven project to use as a jumping board for investigating more of the features in Maven. We have also tried to make the Archetype mechanism additive and by that, we mean allowing portions of a project to be captured in an archetype so that pieces or aspects of a project can be added to existing projects. A good example of this is the Maven site archetype. If, for example, you have used the quick start archetype to generate a working project you can then quickly create a site for that project by using the site archetype within that existing project. You can do anything like this with archetypes.

You may want to standardize J2EE development within your organization so you may want to provide archetypes for EJBs, WARs, or your web services. Once these archetypes are created and deployed in your organization's repository, they are available for use by all developers within your organization.

### Create a simple Maven Project

1. New **➜** Maven Project **➜** maven-archetype-quickstart
2. Group ID: com.springcore (uniquely identifies your project)
3. Artefact ID: springcore (project name)
4. version: 0.01-snapshot
5. package: com.springcore

**pom.xml**

<project **xmlns**="http://maven.apache.org/POM/4.0.0" **xmlns:xsi**="http://www.w3.org/2001/XMLSchema-instance" **xsi:schemaLocation**="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">

<modelVersion>4.0.0</modelVersion>

<groupId>com.vmware</groupId>

<artifactId>springcore</artifactId>

<version>0.0.1-SNAPSHOT</version>

<packaging>jar</packaging>

<name>springcore</name>

<url>http://maven.apache.org</url>

<properties>

<project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>

</properties>

<dependencies>

<dependency>

<groupId>junit</groupId>

<artifactId>junit</artifactId>

<version>3.8.1</version>

<scope>test</scope>

</dependency>

<!-- https://mvnrepository.com/artifact/org.springframework/spring-core -->

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-core</artifactId>

<version>5.3.22</version>

</dependency>

<!-- https://mvnrepository.com/artifact/org.springframework/spring-context -->

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-context</artifactId>

<version>5.3.22</version>

</dependency>

</dependencies>

</project>

## Spring Documentation

### Spring API javadoc 5.3.22

<https://docs.spring.io/spring-framework/docs/5.3.22/javadoc-api/>

### Spring HTML DOC 5.3.22

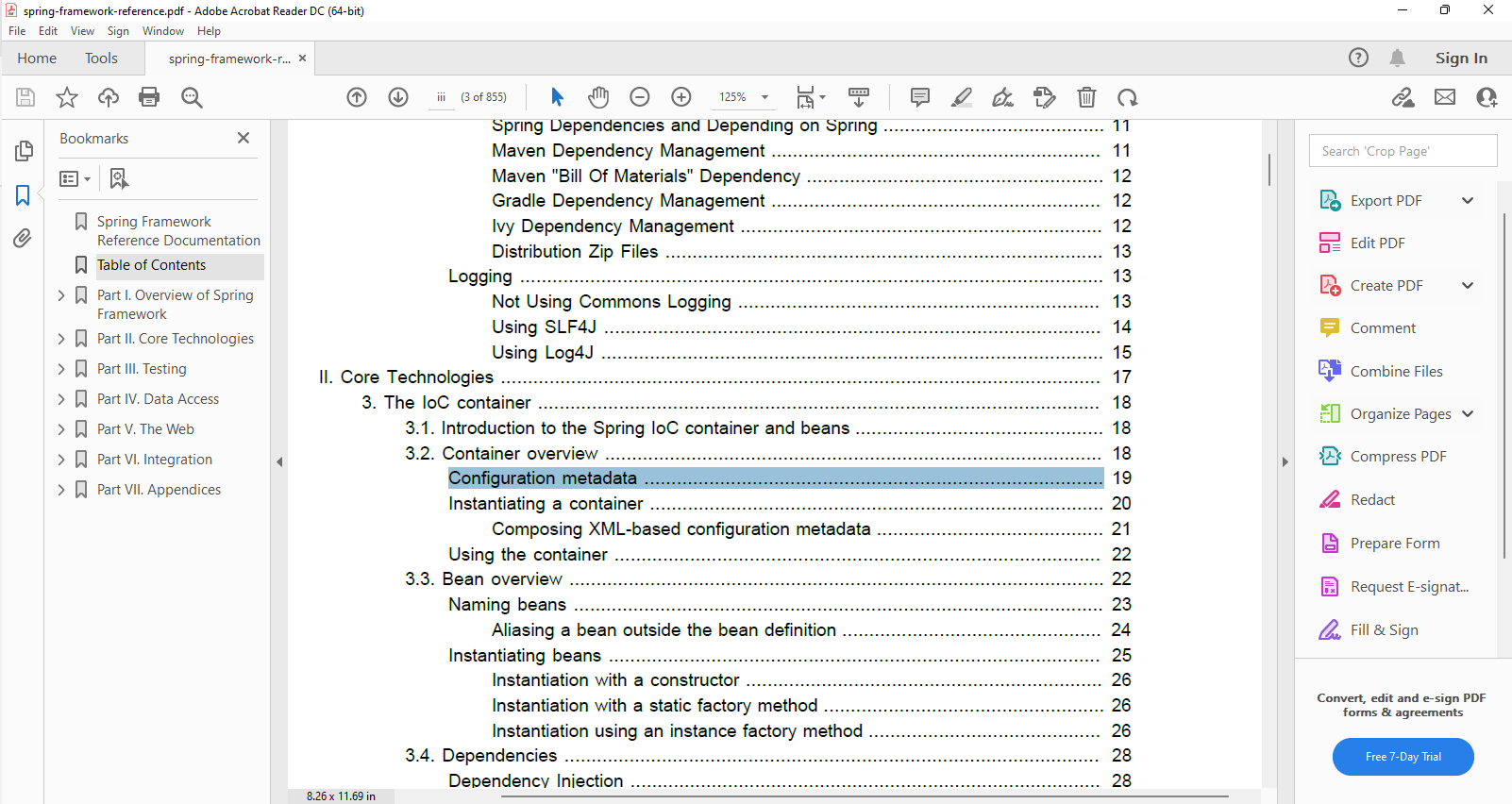
<https://docs.spring.io/spring-framework/docs/current/reference/html/>

### Spring pdf DOC 5.3.22

<https://docs.spring.io/spring-framework/docs/5.3.22/reference/pdf/index.pdf>

### Spring API kdoc 5.3.22

<https://docs.spring.io/spring-framework/docs/5.3.22/kdoc-api/>



## Basic XML config for Spring

Configuring Metadata

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"

**xsi:schemaLocation**="http://www.springframework.org/schema/beans

http://www.springframework.org/schema/beans/spring-beans.xsd">

<bean **id**="..." **class**="...">

<!-- collaborators and configuration for this bean go here -->

</bean>

<bean **id**="..." **class**="...">

<!-- collaborators and configuration for this bean go here -->

</bean>

<!-- more bean definitions go here -->

</beans>

The id attribute is a string that you use to identify the individual bean definition. The class attribute

defines the type of the bean and uses the fully qualified classname. The value of the id attribute refers

to collaborating objects. The XML for referring to collaborating objects is not shown in this example; see

Dependencies for more information.

Added a few more schemas

<?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/beans/spring-context.xsd">

### Instantiating a container

Instantiating a Spring IoC container is straightforward. The location path or paths supplied to an

ApplicationContext constructor are actually resource strings that allow the container to load

configuration metadata from a variety of external resources such as the local file system, from the Java

CLASSPATH, and so on.

ApplicationContext context =

new ClassPathXmlApplicationContext(new String[] {"services.xml", "daos.xml"});

## Instantiating our First Beans

**Student.java**

package com.spring.core.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;

}

@Override

public String toString() {

return "Student [studentId=" + studentId + ", studentName=" + studentName + ", studentAddress=" + studentAddress

+ "]";

}

public Student() {

super();

}

public Student(int studentId, String studentName, String studentAddress) {

super();

this.studentId = studentId;

this.studentName = studentName;

this.studentAddress = studentAddress;

}

public String getStudentName() {

return studentName;

}

public void setStudentName(String studentName) {

// System.out.println("====> setting name");

this.studentName = studentName;

}

public String getStudentAddress() {

return studentAddress;

}

public void setStudentAddress(String studentAddress) {

this.studentAddress = studentAddress;

}

}

**App.java**

package com.spring.core.springcore;

import java.net.URL;

import org.springframework.context.ApplicationContext;

import org.springframework.context.support.ClassPathXmlApplicationContext;

public class App {

public static void main(String[] args) {

System.out.println("START");

// We will not use new keyword to directly make objects if one .xml pass it as a

ApplicationContext context0 = new ClassPathXmlApplicationContext("config.xml");

// Every time a context is initialized it will create all the bean

// It will search the config.xml in the package

ApplicationContext context = new ClassPathXmlApplicationContext("com/spring/core/springcore/config.xml");

// It will search the config.xml in root directory of src/main/resources

// ApplicationContext context2 = new ClassPathXmlApplicationContext(new App().getPath("config.xml"));

/\*

\* if stored in package use full package path of .xml file; for multiconfig

\* files -> pass it as a String array: ApplicationContext context = new

\* ClassPathXmlApplicationContext(new String[] {"services.xml", "daos.xml"});

\*/

Student s1 = (Student) context.getBean("student1");// the bean will be of Object type , thus type casted

System.out.println(s1);

Student s2 = (Student) context.getBean("student2");

System.out.println(s2);

Student s3 = (Student) context.getBean("student3");

System.out.println(s3);

}

String getPath(String file) {

URL url = this.getClass().getResource(file);

String absoluteDiskPath = url.getPath();

System.out.println(absoluteDiskPath);

return absoluteDiskPath;

}

}

**config.xml**

<?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/beans/spring-context.xsd">

<!-- this is our 1st bean -->

<bean **class**="com.spring.core.springcore.Student" **name**="student1">

<!-- For setter injection we use property tag -->

<!-- <property name="studentId" value="6057" /> -->

<property **name**="studentId">

<value>6057</value>

</property>

<property **name**="studentName">

<value>Suman</value>

</property>

<property **name**="studentAddress">

<value>Bangalore</value>

</property>

</bean>

<!-- this is our 2nd bean using inline property tag -->

<bean **class**="com.spring.core.springcore.Student" **name**="student2">

<property **name**="studentId" **value**="6058" />

<property **name**="studentName">

<value>Suphlay</value>

</property>

<property **name**="studentAddress">

<value>Kolkata</value>

</property>

</bean>

<!-- this is our 3rd bean using P Schema(properties) and it has to be included

in the schema tag -->

<bean **class**="com.spring.core.springcore.Student" **name**="student3"

**p:studentId**="6059" **p:studentName**="Ujjwal" **p:studentAddress**="Hooghly" />

</beans>

# Injecting Various Data Types

## Injecting Collections

**Employees.java**

package com.spring.core.collections;

import java.util.List;

import java.util.Map;

import java.util.Properties;

import java.util.Set;

public class Employees {

private String name;

private List<String> phones;

private List<String> pin;

private List<Integer> tags;

private Set<String> address;

private Map<String, String> course;

private Properties general;

public Properties getGeneral() {

return general;

}

public void setGeneral(Properties general) {

this.general = general;

}

public List<Integer> getTags() {

return tags;

}

public void setTags(List<Integer> tags) {

this.tags = tags;

}

public List<String> getPin() {

return pin;

}

public void setPin(List<String> pin) {

this.pin = pin;

}

public String getName() {

return name;

}

public Employees() {

super();

// TODO Auto-generated constructor stub

}

// public Employees(String name, List<String> phones, Set<String> address, Map<String, String> course) {

// super();

// this.name = name;

// this.phones = phones;

// this.address = address;

// this.course = course;

// }

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> getAddress() {

return address;

}

public void setAddress(Set<String> address) {

this.address = address;

}

public Map<String, String> getCourse() {

return course;

}

public void setCourse(Map<String, String> course) {

this.course = course;

}

}

**collectionconfig.xml**

<?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/beans/spring-context.xsd">

<!-- this is our 1st bean and we are injecting collections -->

<bean **name**="emp1" **class**="com.spring.core.collections.Employees">

<property **name**="name" **value**="Radhe" />

<property **name**="phones">

<list>

<value>9563265148</value>

<value>9563265148</value>

<value>9563265148</value>

<!-- to insert null value use <null/> tag -->

<null />

</list>

</property>

<!-- for one element only , no need for list/set tag -->

<property **name**="pin">

<value>828130</value>

</property>

<property **name**="tags">

<!-- Creates an Empty List -->

<list>

</list>

</property>

<property **name**="address">

<set>

<value>Delhi</value>

<value>Lucknow</value>

<value>Kanpur</value>

<value>Delhi</value>

</set>

</property>

<property **name**="course">

<map>

<entry **key**="Java" **value**="3" />

<entry **key**="Python" **value**="4" />

<entry **key**="JS" **value**="6.5" />

</map>

</property>

<property **name**="general">

<props>

<prop **key**="gender">male</prop>

<prop **key**="age">24</prop>

<prop **key**="passportAvailbility">YES</prop>

</props>

</property>

</bean>

</beans>

**CollectionTest.java**

package com.spring.core.injecting\_collections;

import org.springframework.context.ApplicationContext;

import org.springframework.context.support.ClassPathXmlApplicationContext;

public class CollectionDriverCode {

@SuppressWarnings("resource")

public static void main(String[] args) {

System.out.println("START");

ApplicationContext context = new ClassPathXmlApplicationContext(

"com/spring/core/injecting\_collections/collection\_config.xml");

Employees emp1 = (Employees) context.getBean("emp1");

System.out.println(emp1.getName());

System.out.println(emp1.getPin());

System.out.println(emp1.getTags());

System.out.println(emp1.getPhones());

System.out.println(emp1.getPhones().getClass());// class java.util.ArrayList by default

System.out.println(emp1.getPhones().getClass().getName());// java.util.ArrayList

System.out.println(emp1.getAddress());

System.out.println(emp1.getCourse());

System.out.println(emp1.getGeneral());

System.out.println("END\n");

}

}

## Standalone Collections using **util** Schema

<list>

<value>9563265148</value>

<value>9563265148</value>

<value>9563265148</value>

<!-- to insert null value use <null/> tag -->

<null />

</list>

In the above code snippet default implementation class has been taken (unaware it is a linked list / array-list...) same goes for set *<set></set>* (unaware it is a linked hash set / hash set...)

Stand-alone Collections can be used for saving database credentials in a collection inside a file to avoid recompilation every time.

* We will use standalone collection to achieve global usage within the XML file
* We can also add specificity to the collection and define which type of it i.e., array list, LinkedList.

**alone-config.xml**

<?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"

**xmlns:util**="http://www.springframework.org/schema/util"

**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

http://www.springframework.org/schema/util

http://www.springframework.org/schema/util/spring-util.xsd

">

<!-- my list stand-alone independent custom list -->

<!-- list-class="java.util.Vector" -->

<util:list **list-class**="java.util.LinkedList"

**id**="mybestfriends">

<value>Suman</value>

<value>Satyam</value>

<value>Sukanya</value>

<value>Apurv</value>

<value>Ashutosh</value>

<value>Nitish</value>

<value>Soumadeep</value>

<value>null</value>

</util:list>

<!-- this is our 1st bean -->

<util:map **map-class**="java.util.TreeMap" **id**="fees">

<!-- TreeMap is used for key wise sorting -->

<!-- <util:map map-class="java.util.HashMap" id="fees"> -->

<entry **key**="Spring Core" **value**="8000" />

<entry **key**="Django" **value**="6000" />

<entry **key**="React" **value**="5000" />

</util:map>

<!-- Properties is a class extending Hashtable -->

<util:properties **id**="props">

<prop **key**="college">HIT</prop>

<prop **key**="branch">Chemical</prop>

<prop **key**="since">2015</prop>

</util:properties>

<!-- this is our 1st bean for Person -->

<bean **class**="com.spring.core.standalone\_collections.Person"

**name**="p1">

<property **name**="friends">

<ref **bean**="mybestfriends" />

</property>

<!-- using self closing property tag & reusing the Linkedlist -->

<property **name**="feeStructure" **ref**="fees" />

<property **name**="props" **ref**="props" />

</bean>

<bean **class**="com.spring.core.standalone\_collections.Person"

**name**="p2">

<property **name**="friends" **ref**="mybestfriends" />

<property **name**="props" **ref**="props" />

<property **name**="feeStructure" **ref**="fees" />

</bean>

</beans>

**Person.java**

package com.spring.core.standalone\_collections;

import java.util.List;

import java.util.Map;

import java.util.Properties;

public class Person {

private List<String> friends;

private Map<String, Integer> feeStructure;

private Properties props;

public Properties getProps() {

return props;

}

public void setProps(Properties props) {

this.props = props;

}

public Map<String, Integer> getFeeStructure() {

return feeStructure;

}

public void setFeeStructure(Map<String, Integer> feeStructure) {

this.feeStructure = feeStructure;

}

public List<String> getFriends() {

return friends;

}

@Override

public String toString() {

return "Person [friends=" + friends + ",\nfeeStructure=" + feeStructure + "]\n" + "Properties ->" + props

+ "\n";

}

// we will use setter injection

public void setFriends(List<String> friends) {

this.friends = friends;

}

}

**TestStanaloneCollections.java**

package com.spring.core.standalone\_collections;

import org.springframework.context.ApplicationContext;

import org.springframework.context.support.ClassPathXmlApplicationContext;

public class TestStanaloneCollections {

public static void main(String[] args) {

ApplicationContext context = new ClassPathXmlApplicationContext(

"com/spring/core/standalone\_collections/alone\_config.xml");

Person p1 = context.getBean("p1", Person.class);

System.out.println(p1);

System.out.println(p1.getFriends().getClass());

System.out.println(p1.getFriends().getClass().getName());

System.out.println(p1.getFeeStructure().getClass());

Person p2 = context.getBean("p2", Person.class);

System.out.println(p2);

System.out.println(p2.getFeeStructure().getClass());

// if ref isNull don't invoke getClass()

}

}

## Injecting Custom reference types

Here we will inject a class B inside A.

Instance variable names are quite crucial in injecting the dependencies.

**B.java**

package com.spring.core.injecting\_reference;

public class B {

private int y;

public int getY() {

return y;

}

@Override

public String toString() {

return "B [y=" + y + "]";

}

public void setY(int y) {

this.y = y;

}

public B() {

super();

// TODO Auto-generated constructor stub

}

}

**A.java**

package com.spring.core.injecting\_reference;

public class A {

private int x;

private B obj;

public int getX() {

return x;

}

public void setX(int x) {

this.x = x;

}

public B getObj() {

return obj;

}

public void setObj(B obj) {

this.obj = obj;

}

public A() {

super();

// TODO Auto-generated constructor stub

}

@Override

public String toString() {

return "A [x=" + x + ", obj=" + obj + "]";

}

}

**DriverReference.java**

package com.spring.core.injecting\_reference;

import org.springframework.context.ApplicationContext;

import org.springframework.context.ConfigurableApplicationContext;

import org.springframework.context.support.ClassPathXmlApplicationContext;

public class DriverReference {

public static void main(String[] args) {

ApplicationContext context = new ClassPathXmlApplicationContext(

"com/spring/core/injecting\_reference/ref\_config.xml");

A temp = (A) context.getBean("aref1");

System.out.println(temp.getX());

System.out.println(temp);

System.out.println(temp.getObj());

System.out.println(temp.getObj().getY());

A temp2 = (A) context.getBean("aref2");

System.out.println(temp2);

// https://stackoverflow.com/questions/14423980/how-to-close-a-spring-applicationcontext#:~:text=Type%20Cast%20the%20ApplicationContext%20Object,the%20close%20object%20on%20that.

((ConfigurableApplicationContext) context).close();// to close ==> downcast to Configurable Application context

}

}

**ref\_config.xml**

<?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/beans/spring-context.xsd">

<!-- this is our B bean -->

<bean **class**="com.spring.core.injecting\_reference.B" **name**="bref">

<property **name**="y" **value**="25" />

</bean>

<!-- this is for our A bean and we will inject B obj reference -->

<bean **class**="com.spring.core.injecting\_reference.A" **name**="aref1">

<property **name**="x" **value**="24" />

<!-- <property name="obj"> -->

<!-- <ref bean="bref"/> -->

<!-- </property> -->

<!-- new way #inline -->

<property **name**="obj" **ref**="bref" />

</bean>

<!-- using p Schema -->

<bean **class**="com.spring.core.injecting\_reference.A" **name**="aref2"

**p:x**="25" **p:obj-ref**="bref" />

<!-- p:obj-ref is telling obj is of ref type -->

</beans>

# DI via Constructor injection

**Person.java**

package com.spring.core.constructor\_injection;

import java.util.List;

public class Person {

private String name;

private int age;

private Certi certi;

private List<String> listOfCourses;

public Person(String name, int age,Certi certi,List<String> listOfCourses) {

super();

this.name = name;

this.age = age;

this.certi = certi;

this.listOfCourses = listOfCourses;

System.out.println("Person constructor called");

}

@Override

public String toString() {

return "Person [name=" + name + ", age=" + age + ", certi=" + certi + ", listOfCourses=" + listOfCourses + "]";

}

 }

**Certi.java**

package com.spring.core.constructor\_injection;

public class Certi {

private String name;

public Certi(String name) {

super();

this.name = name;

}

@Override

public String toString() {

return "Certi [name=" + name + "]";

}

}

**TestCI.java**

package com.spring.core.constructor\_injection;

import org.springframework.context.ApplicationContext;

import org.springframework.context.support.ClassPathXmlApplicationContext;

public class TestCI {

public static void main(String[] args) {

ApplicationContext context = new ClassPathXmlApplicationContext(

"com/spring/core/constructor\_injection/ci\_config.xml");

Person p1 = (Person) context.getBean("person1");

Person p2 = (Person) context.getBean("person2");

System.out.println(p1);

System.out.println(p2);

// Ambiguity Issue

Addition a1 = (Addition) context.getBean("add");

a1.doSum();

}

}

**ci\_config.xml**

<?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"

**xmlns:c**="http://www.springframework.org/schema/c"

**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/beans/spring-context.xsd">

<!-- this is our 1st bean for Certi using C-schema -->

<bean **class**="com.spring.core.constructor\_injection.Certi"

**name**="cert" **c:name**="JAVA-OCJB">

<!-- <constructor-arg value="Spring WEB"/> -->

</bean>

<!-- this is our 1st bean for Person -->

<bean **class**="com.spring.core.constructor\_injection.Person"

**name**="person1">

<constructor-arg>

<value>Suman</value>

</constructor-arg>

<!-- <constructor-arg value="18"/> -->

<!-- It will automatically convert compatible String to int. -->

<!-- but we can explicitly specify also the type -->

<!-- its a good habit to define the type explicitly -->

<!-- because it will avoid ambiguity in multiple constructors -->

<constructor-arg **value**="18" **type**="int" />

<constructor-arg **ref**="cert" />

<constructor-arg **type**="java.util.List">

<list>

<value>Python</value>

<value>JAVA</value>

<value>JS</value>

</list>

</constructor-arg>

</bean>

<!-- here we cannot use p-type, instead we use c-type schema -->

<!-- https://mkyong.com/spring/how-to-inject-null-value-in-spring/ -->

<bean **class**="com.spring.core.constructor\_injection.Person"

**name**="person2" **c:name**="Suman" **c:age**="20" **c:certi-ref**="cert">

<constructor-arg **name**="listOfCourses">

<null />

</constructor-arg>

</bean>

<bean **class**="com.spring.core.constructor\_injection.Addition"

**name**="add">

<constructor-arg **value**="40" **type**="int" **index**="1" />

<constructor-arg **value**="60" **type**="int" **index**="0" />

<!-- order can be interchanged via index -->

</bean>

</beans>

## Resolving Ambiguity

**Addition.java**

package com.spring.core.constructor\_injection;

public class Addition {

private int a;

private int b;

/\*

\* Here ambiguity arises if we have it define type in .xml due to ambiguity the

\* top order(the 1st) constructor matching the parameter is getting executed

\* First priority goes to String irrespective of order Then the order of

\* constructor in checked if it doesn't matches

\*

\*/

// it will be called if we are passing int int as this is the 1st constructor matching

public Addition(double a, double b) {

super();

this.a = (int) a;

this.b = (int) b;

System.out.println("constructor : double | double");

}

public Addition(int a, int b) {

super();

/\*

\* Spring is so intelligent that it can match the best constructor from the

\* first explicit type="int only"

\*/

this.a = a;

this.b = b;

System.out.println("constructor : int | int");

}

public Addition(String a, String b) {

// highest priority is given to String

super();

this.a = Integer.parseInt(a);

this.b = Integer.parseInt(b);

System.out.println("constructor : String | String");

}

public void doSum() {

System.out.println("a = " + a + " | b = " + b);

System.out.println("SUM =" + (this.a + this.b));

}

}

**TextCI.java**

package com.spring.core.constructor\_injection;

import org.springframework.context.ApplicationContext;

import org.springframework.context.support.ClassPathXmlApplicationContext;

public class TestCI {

public static void main(String[] args) {

ApplicationContext context = new ClassPathXmlApplicationContext(

"com/spring/core/constructor\_injection/ci\_config.xml");

// Ambiguity Issue

Addition a1 = (Addition) context.getBean("add");

a1.doSum();

}

}

**ci\_config.xml**

<?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"

**xmlns:c**="http://www.springframework.org/schema/c"

**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/beans/spring-context.xsd">

<!-- As we are defining the type and position, we are pin-pointing out constructor -->

<bean **class**="com.spring.core.constructor\_injection.Addition"

**name**="add">

<constructor-arg **value**="40" **type**="int" **index**="1" />

<constructor-arg **value**="60" **type**="int" **index**="0" />

<!-- order can be interchanged via index -->

</bean>

</beans>

## “Parent” Attribute while DI Injection

**package** com.spring.core.constructor\_injection.parent\_attribute;

**public** **class** Address {

**private** String addressLine1, city, state, country;

**public** Address(String addressLine1, String city, String state, String country) {

**super**();

**this**.addressLine1 = addressLine1;

**this**.city = city;

**this**.state = state;

**this**.country = country;

}

**public** String toString() {

**return** addressLine1 + " " + city + " " + state + " " + country;

}

}

# Life Cycle methods

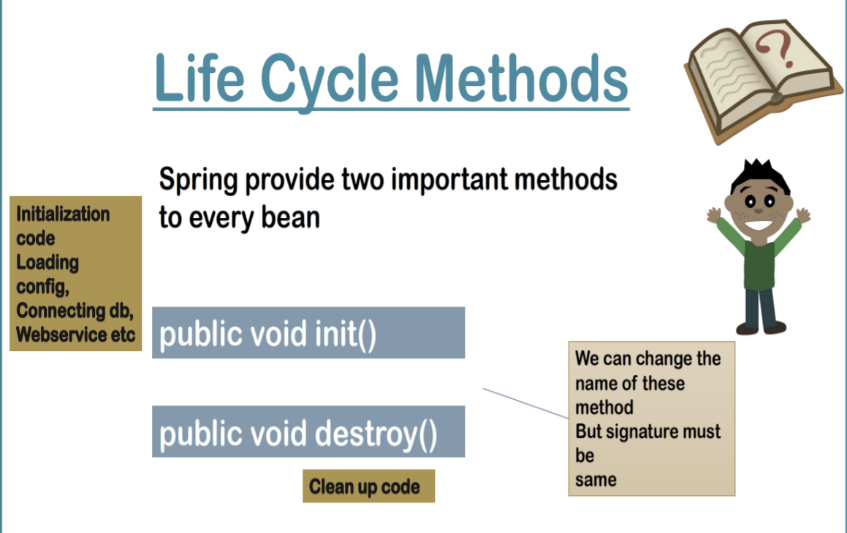


Figure Lifecycle methods

We can change the name of method but signature should be same.

* public void init()
  + Loading config, DB connection, pre config
* public void destroy()
  + Cleanup code

## Whole lifecycle of a Spring Bean

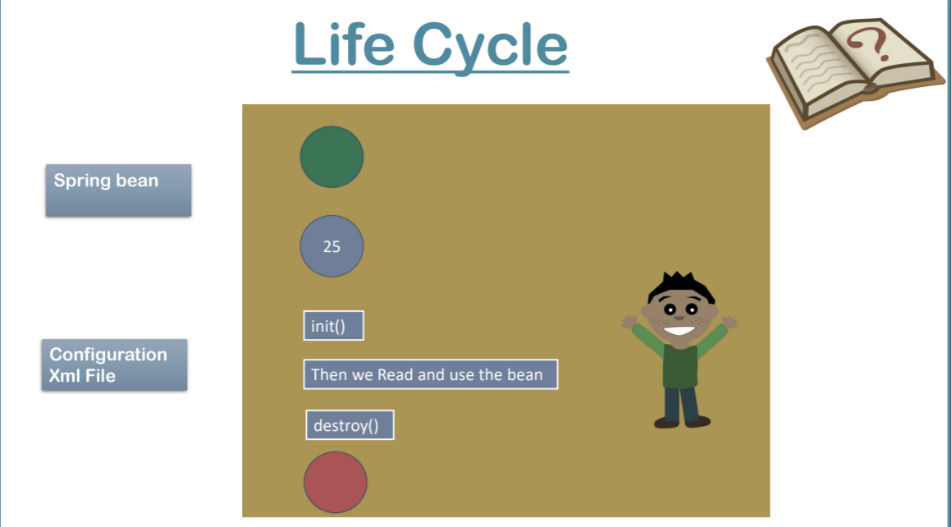


Figure Lifecycle Diagram

The key thing to notice here is init() is being called after property initialisation.

We can implement lifecycle methods in three ways.

1. Xml config
2. Spring Java interfaces
3. @Annotation

init() is quite easy to implement but for destroy() we have to enable the ***pre shutdown hook*** on the context.

//ApplicationContext

AbstractApplicationContext context = new ClassPathXmlApplicationContext("com/spring/core/lifecycle/config.xml");

// We used AbstractApplicationContext to invoke destroy() to enable pre-shutdown.

 context.registerShutdownHook();

The order of destroy for beans may differ but individual order of every beans is ***setting > init > destroy***

**Samosa.java** *XML*

package com.spring.core.lifecycle;

/\*lifecycle methods using xml\*/

public class Samosa {

private double price;

public double getPrice() {

return price;

}

public void setPrice(double price) {

System.out.println("Setting price of Samosa via setter");

this.price = price;

}

public Samosa() {

super();

// TODO Auto-generated constructor stub

}

@Override

public String toString() {

return "Samosa [price=" + price + "]";

}

public void hey() {

System.out.println("INIT -> samosa renamed hey() via xml");

}

public void destroy() {

System.out.println("DESTROY -> samosa destroy() via xml");

}

}

**CocaCoal.java** Spring Java Interfaces - *InitializingBean, DisposableBean*

package com.spring.core.lifecycle;

import org.springframework.beans.factory.DisposableBean;

import org.springframework.beans.factory.InitializingBean;

/\*lifecycle methods using interface\*/

public class CocaCola implements InitializingBean, DisposableBean {

private double price;

public double getPrice() {

return price;

}

public void setPrice(double price) {

System.out.println("Setting price of coke via setters");

this.price = price;

}

public CocaCola() {

super();

// TODO Auto-generated constructor stub

}

@Override

public String toString() {

return "CocaCola [price=" + price + "]";

}

// init code here

public void afterPropertiesSet() throws Exception {

System.out.println("INIT taking Coke: init via interface");

}

// destroy|cleanup code here

public void destroy() throws Exception {

System.out.println("DESTROY returning Coke bottle: destroy via interface");

}

}

**BreadPakoda.java** with @Annotation @PostConstruct @PreDestroy

package com.spring.core.lifecycle;

//lifecycle using @Annotation

import javax.annotation.PostConstruct;

import javax.annotation.PreDestroy;

public class BreadPakoda {

private double price;

public double getPrice() {

return price;

}

public void setPrice(double price) {

this.price = price;

System.out.println("Setting price of BreadPakoda via setter");

}

@Override

public String toString() {

return "BreadPakoda [price=" + price + "]";

}

public BreadPakoda() {

super();

}

// for 9+ java have to mention the dependencies

@PostConstruct

public void start() {

System.out.println("INIT Bread using @Annotations");

}

@PreDestroy

public void end() {

System.out.println("DESTROY Bread using @Annotations");

}}

config.xml

<?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

">

<!-- <context:annotation-config /> -->

<!-- teh above tag enables all the annotations in config.xml -->

<!-- enables only @PostConstruct,@PreDestroy -->

<bean **class**="org.springframework.context.annotation.CommonAnnotationBeanPostProcessor" />

<bean **class**="com.spring.core.lifecycle.Samosa" **name**="s1"

**init-method**="hey" **destroy-method**="destroy">

<property **name**="price" **value**="10" />

</bean>

<bean **class**="com.spring.core.lifecycle.CocaCola" **name**="c1"

**p:price**="15" />

<bean **class**="com.spring.core.lifecycle.BreadPakoda" **name**="b1"

**p:price**="20" />

</beans>

**TestLifeCycle.java**

package com.spring.core.lifecycle;

import org.springframework.context.support.AbstractApplicationContext;

import org.springframework.context.support.ClassPathXmlApplicationContext;

public class TestLifecycle {

public static void main(String[] args) {

// TODO Auto-generated method stub

// ApplicationContext

AbstractApplicationContext context = new ClassPathXmlApplicationContext("com/spring/core/lifecycle/config.xml");

// We used AbstractApplicationContext to invoke destroy() to enable pre-shutdown.

context.registerShutdownHook();

// Registering shutdown hook when you have to destroy object call the hooked-> destroy method()

Samosa s1 = (Samosa) context.getBean("s1");

System.out.println("\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_");

System.out.println(s1);

CocaCola c1 = (CocaCola) context.getBean("c1");

System.out.println(c1);

BreadPakoda b1 = (BreadPakoda) context.getBean("b1");

System.out.println(b1);

System.out.println("\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_");

  context.close();

// The order of destroy for beans may differ but individual order of every beans is setting->init->destroy

}

}

### Java 9+

Note that both the @PostConstruct and @PreDestroy annotations are part of Java EE. Since [Java EE was deprecated in Java 9](https://www.baeldung.com/java-enterprise-evolution), and removed in Java 11, we have to add an additional dependency to use these annotations:

<**dependency**>

<**groupId**>javax.annotation</**groupId**>

<**artifactId**>javax.annotation-api</**artifactId**>

<**version**>1.3.2</**version**>

</**dependency**>

<https://www.baeldung.com/spring-postconstruct-predestroy>

***Changes in config xml***

<!-- <context:annotation-config /> -->

<!-- teh above tag enables all the annotations in config.xml -->

<!-- enables only @PostConstruct,@PreDestroy -->

<bean **class**="org.springframework.context.annotation.CommonAnnotationBeanPostProcessor" />

# Auto Wiring

## Intro

1. It’s a feature of Spring in which Spring Container injects the dependencies automatically.
2. It’s used to inject reference (OBJECT) type only so cannot be used for primitive type. And it’s good because we mostly use reference as a medium of sending values.
3. B ---> A B is a dependency of A (Spring container will automatically link this and provide the dependency)

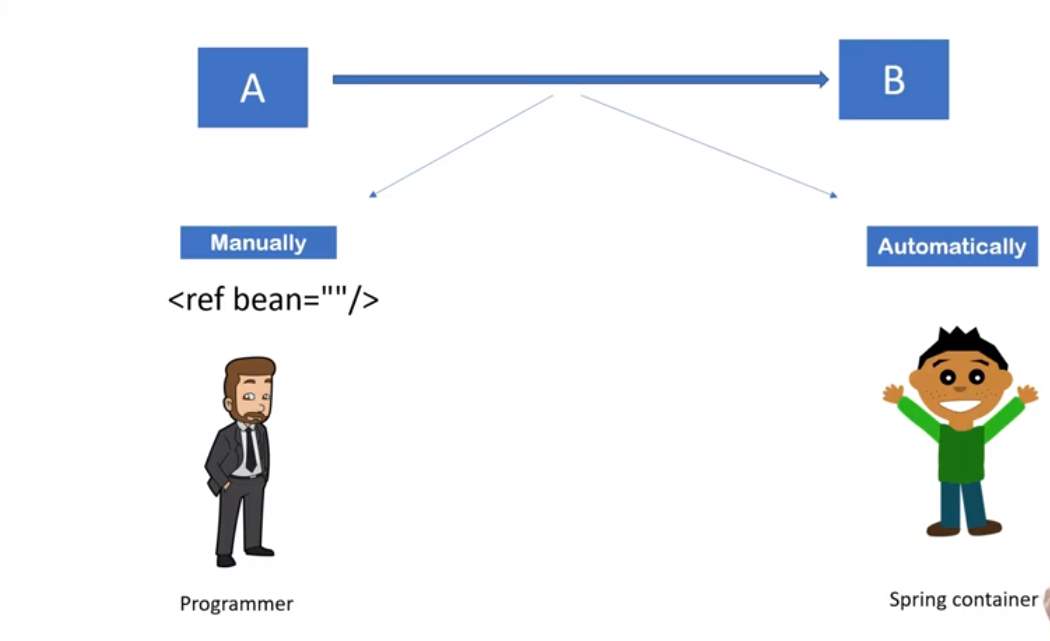


Figure Auto wiring Dependency

Auto wiring can be achieved by:

* XML Auto wiring modes
  + No mode (By default disabled)
  + byName
  + byType
  + constructor
  + autodetect (deprecated since Spring 3)
* Annotations
  + @Autowired

|  |  |
| --- | --- |
| **Auto-Wiring** | |
| **Advantages** | **Disadvantages** |
| * Automation on DI * Less code | * Almost no control of programmer * Can't be used for primitive and String values |

## By XML Declaration

**Address.java**

package com.spring.core.auto\_wire.xml;

public class Address {

private String street;

private int pin;

private String city;

@Override

public String toString() {

return "Address [street=" + street + ", pin=" + pin + ", city=" + city + "]";

}

public String getStreet() {

return street;

}

public void setStreet(String street) {

this.street = street;

}

public int getPin() {

return pin;

}

public void setPin(int pin) {

this.pin = pin;

}

public String getCity() {

return city;

}

public void setCity(String city) {

this.city = city;

}

}

**Worker.java**

package com.spring.core.auto\_wire.xml;

public class Worker {

private Address address;

public Address getAddress() {

return address;

}

public void setAddress(Address address) {

System.out.println("Setters are used for injection");

this.address = address;

}

@Override

public String toString() {

return "Worker [address=" + address + "]";

}

public Worker() {

super();

// TODO Auto-generated constructor stub

}

// public Worker(Address address3) {

public Worker(Address address) {

super();

System.out.println("Parameterized constructor is used for injection");

this.address = address;

}

}

**auto\_config.xml**

<?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

">

<bean **class**="com.spring.core.auto\_wire.xml.Address"

**name**="address1" **p:street**="Kalikapoor" **p:city**="Delhi" **p:pin**="700107" />

<bean **class**="com.spring.core.auto\_wire.xml.Address" **name**="address"

**p:street**="Doranda" **p:city**="Ranchi" **p:pin**="800107" />

<!-- we will use auto-wiring here -->

<!-- 1) it will search for name "address" in existing bean -->

<!-- so it need to be taken care while matching -->

<bean **class**="com.spring.core.auto\_wire.xml.Worker" **name**="w1"

**autowire**="constructor" />

<!-- autowire="byType" -->

<!-- when using byType ensure only 1 type matches otherwise Exception will

be thrown because of Ambiguity -->

<!-- autowire="byName" -->

<!-- autowire="constructor" -->

<!-- for all the technique other than constructor, setter injection will

be involved , also constructor injection also check name for injecting -->

</beans>

**TestAutoWiring.java**

package com.spring.core.auto\_wire.xml;

import org.springframework.context.support.AbstractApplicationContext;

import org.springframework.context.support.ClassPathXmlApplicationContext;

public class TestAutoWiring {

public static void main(String[] args) {

// TODO Auto-generated method stub

AbstractApplicationContext context = new ClassPathXmlApplicationContext(

"com/spring/core/auto\_wire/xml/auto\_config.xml");

// to skip type-casting

Worker w1 = context.getBean("w1", Worker.class);

System.out.println(w1);

}

}

## By @Autowired

**Address.java**

package com.spring.core.auto\_wire.annotations;

public class Address {

private String street;

private int pin;

private String city;

@Override

public String toString() {

return "Address [street=" + street + ", pin=" + pin + ", city=" + city + "]";

}

public String getStreet() {

return street;

}

public void setStreet(String street) {

this.street = street;

}

public int getPin() {

return pin;

}

public void setPin(int pin) {

this.pin = pin;

}

public String getCity() {

return city;

}

public void setCity(String city) {

this.city = city;

}

}

**Worker.java**

package com.spring.core.auto\_wire.annotations;

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

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

public class Worker {

// @Qualifier("current") // to pin point the dependency

@Autowired // uses "type" by default

private Address address;

// if there are multiple types available, if at least one should be matching the var name,

// it will not give error else it will be ambiguous for multiple beans

public Address getAddress() {

return address;

}

// @Autowired

// @Qualifier("permanent")//setter gets priority over properties

public void setAddress(Address address) {

System.out.println("Setters are used for injection");

this.address = address;

}

@Override

public String toString() {

return "Worker [address=" + address + "]";

}

public Worker() {

super();

}

// @Autowired

// @Qualifier("permanent")// not allowed in Constructo

public Worker(Address address) {

super();

System.out.println("Parameterized constructor is used for injection");

this.address = address;

}

}

* If we add @Autowired to all the 3 [Var, setter, constructor] First constructor will execute and then setter will execute
* @Qualifier is not allowed for constructors.
* @Qualifier("current") // may throw *NoSuchBeanFound* if the bean has been not declared with the given name

**auto\_config.xml**

<?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

">

<!-- Enable @Aurowire config -->

<context:annotation-config />

<!-- recommended to set same bean name as that name for variable -->

<bean **class**="com.spring.core.auto\_wire.annotations.Address"

**name**="current" **p:street**="Kalikapoor" **p:city**="Delhi 6" **p:pin**="700107" />

<bean **class**="com.spring.core.auto\_wire.annotations.Address"

**name**="permanent" **p:street**="White Field" **p:city**="Bangalore"

**p:pin**="701605" />

<bean **class**="com.spring.core.auto\_wire.annotations.Address"

**name**="address" **p:street**="Pink City" **p:city**="Jaipur"

**p:pin**="901675" />

<bean **class**="com.spring.core.auto\_wire.annotations.Worker"

**name**="w1" />

</beans>

**TestAutoWiring.java**

package com.spring.core.auto\_wire.annotations;

import org.springframework.context.support.AbstractApplicationContext;

import org.springframework.context.support.ClassPathXmlApplicationContext;

public class TestAutoWiring {

public static void main(String[] args) {

AbstractApplicationContext context = new ClassPathXmlApplicationContext(

"com/spring/core/auto\_wire/annotations/auto\_config.xml");

// to skip type-casting

Worker w1 = context.getBean("w1", Worker.class);

System.out.println(w1);

}

}

## Stereotype Annotations

Till now we were we were defining our bean in xml file.

**auto\_config.xml**

<?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

">

<!-- Enable @Aurowire config -->

<context:annotation-config />

<!-- recommended to set same bean name as that name for variable -->

<bean **class**="com.spring.core.auto\_wire.annotations.Address"

**name**="current" **p:street**="Kalikapoor" **p:city**="Delhi 6" **p:pin**="700107" />

<bean **class**="com.spring.core.auto\_wire.annotations.Address"

**name**="permanent" **p:street**="White Field" **p:city**="Bangalore"

**p:pin**="701605" />

<bean **class**="com.spring.core.auto\_wire.annotations.Address"

**name**="address" **p:street**="Pink City" **p:city**="Jaipur"

**p:pin**="901675" />

<bean **class**="com.spring.core.auto\_wire.annotations.Worker"

**name**="w1" />

</beans>

We can also use annotation to create our beans.

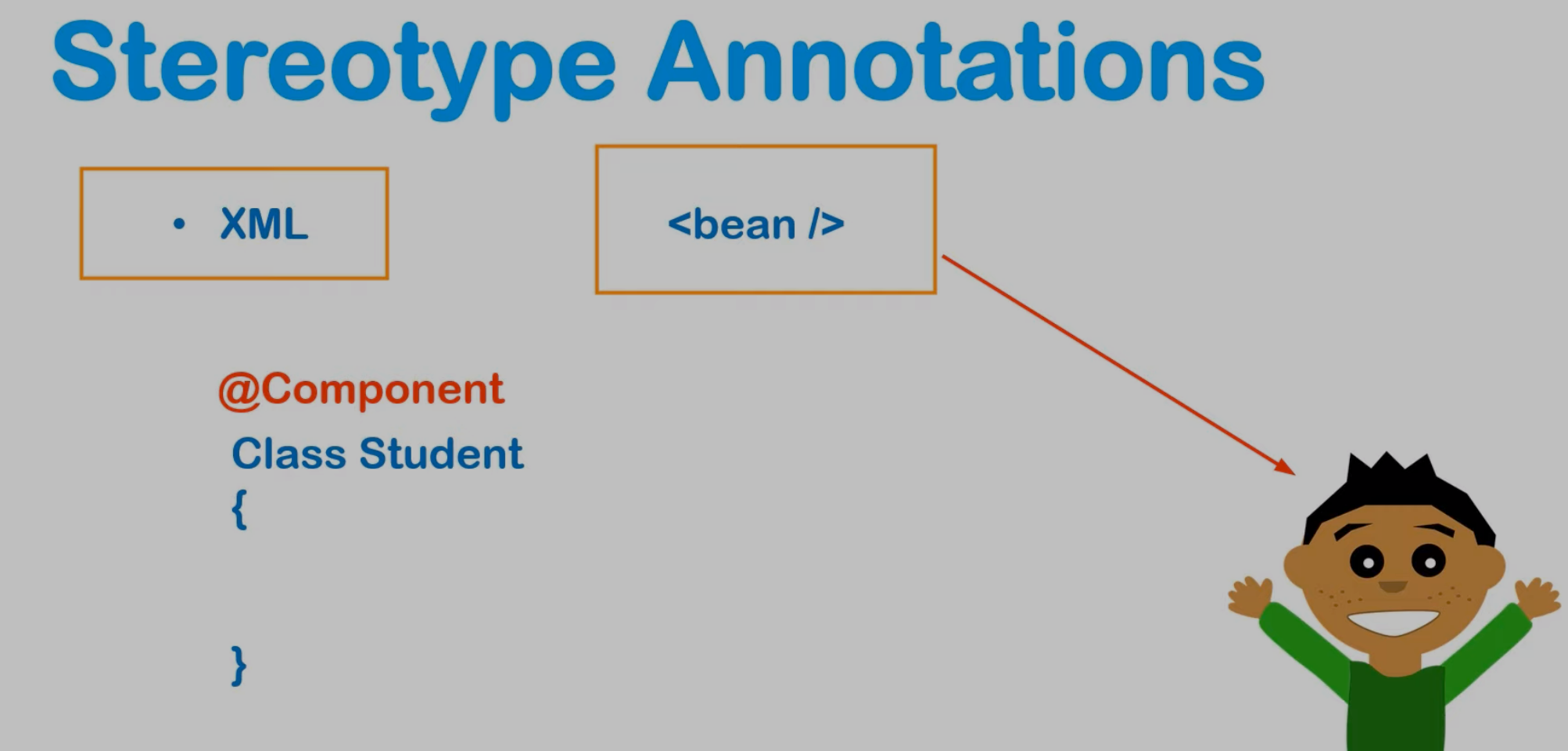
**Stereotype Annotations**



* To tell Spring container to make object we use to add the details inside <bean /> in XML
* But we can also use few annotations w/o using <bean /> in XML for the same.

**@Component** can be used to tell the Spring Container that this is the class which needs to instantiate.

**<bean />** is being replaced by **@Component**



**ie:**

package sprinhg.core.Student

@Component

Class Student {

}

Student student = new *Student*(); /*/ follows camelCase for automatic standard variable name.*

**Prerequisite:**

We have to tell in XML configuration file the package containing this class.

**<context:component-scan base-package=””/>**

Scans packages and all the sub packages for the target Class.

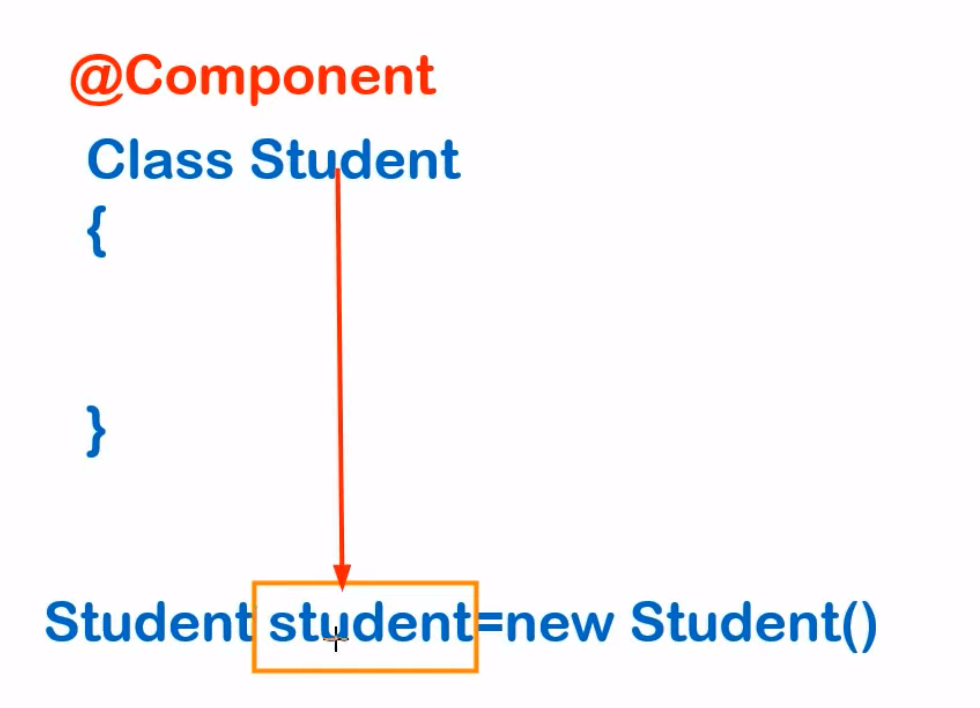


Figure : Internally it uses camel case

**Student.java**

package com.spring.core.stereotype;

import java.util.List;

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

import org.springframework.context.annotation.Scope;

import org.springframework.stereotype.Component;

@Component("s1") // Don't use student; use s1

@Scope("singleton")

//@Scope("prototype")

public class Student {

@Value("Suman Shekhar")

private String studentName;

@Value("Bangalore")

private String city;

@Value("#{course}") // spEL

private List<String> courses;

public String getStudentName() {

return studentName;

}

public void setStudentName(String studentName) {

this.studentName = studentName;

}

@Override

public String toString() {

return "Student [studentName=" + studentName + ", city=" + city + ", courses=" + courses + "]";

}

public String getCity() {

return city;

}

public List<String> getCourses() {

return courses;

}

public void setCourses(List<String> courses) {

this.courses = courses;

}

public void setCity(String city) {

this.city = city;

}

}

**Teacher.java**

package com.spring.core.stereotype;

public class Teacher {

}

**stereo\_config.xml**

<?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"

**xmlns:util**="http://www.springframework.org/schema/util"

**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

http://www.springframework.org/schema/util

http://www.springframework.org/schema/util/spring-util.xsd

">

<context:component-scan

**base-package**="com.spring.core.stereotype"></context:component-scan>

<util:list **list-class**="java.util.Vector" **id**="course">

<value>Spring</value>

<value>Git</value>

<value>Django</value>

<value>React</value>

</util:list>

<bean **class**="com.spring.core.stereotype.Teacher" **name**="teacher"

**scope**="prototype" />

</beans>

**TestStereo.java**

package com.spring.core.stereotype;

import org.springframework.context.ApplicationContext;

import org.springframework.context.support.ClassPathXmlApplicationContext;

import com.spring.core.standalone\_collections.Person;

public class TestStereo {

public static void main(String[] args) {

ApplicationContext context = new ClassPathXmlApplicationContext("com/spring/core/stereotype/stereo\_config.xml");

Student s1 = context.getBean("s1", Student.class);//

System.out.println(s1);

System.out.println(s1.getCourses().getClass());

System.out.println(s1.hashCode());

System.out.println();

Student s2 = context.getBean("s1", Student.class);

System.out.println(s2);

System.out.println(s2.hashCode());

// Changing scope via XML

System.out.println();

Teacher t1 = context.getBean("teacher", Teacher.class);

Teacher t2 = context.getBean("teacher", Teacher.class);

System.out.println("Teacher\n" + t1.hashCode());

System.out.println(t2.hashCode());

}

}