**Frameworks: Hibernate, Spring,SpringBoot**

:Spring Framework:

**Spring framework is one the most popular JEE framework.**

**It is an open source and lightweight framework created by ROD JOHNSON in 2003.**

**Introduction to Spring Framework**

**Class Student{**

**Private Int id;**

**Private String name;**

**// public getter and setter method**

**}**

**Student s=new Student(); //dependency on new operator**

**s.setId(10); //we are dependent on . operator**

**s.getId()**

**Spring :**

**bean: pojo: plain old java object/ interface: POJI**

**Student id=”s”**

**Student s=new Student();**

**<bean id=”s” class=”student”/>====spring configuration file(xml type)**

**<property name=”id” value=”12”/>=== spring configuration file(xml type)**

**s.setId(12);**

**Student s=new Student(); //no need to perform manually**

Spring is a Java-based application framework that is designed and developed by the Pivotal Software Company.

Spring is an application framework that is used to create Enterprise Applications. We can create **web-based** applications easily due to its vast library and tools.

Spring provides an easy and friendly environment to create Java enterprise applications. It is [full of features](https://www.studytonight.com/spring-framework/spring-features) and provides various other sub-projects such as **Spring Security**, **Spring Boot**, **Spring MVC**, **Spring Cloud**, **Spring Data**, etc that help to build applications accordingly.

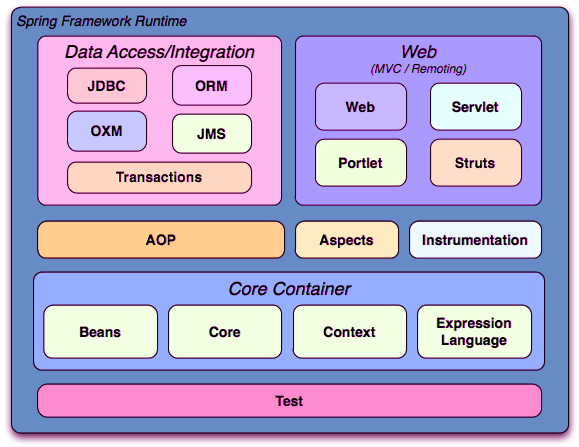
Spring 5 has been improved over time, in the early days of Java EE and Spring, we deploy applications to an application server but now with the help of Spring Boot we can create applications in a DevOps and cloud-friendly way.

The core and heart of Spring is an IOC container that manages bean objects and allows dependency injection.

Spring is built with several components(modules) to work with the **web**, **database**, **network**, etc. Below is the image of Spring Runtime that shows its internal architecture.

Spring Framework Architecture

In the diagram below, we have shown the Spring framework architecture:



1. Test:

Spring test module provides the supports for testing of spring components with JUnit or TestNG frameworks.

2. Core Container:

Spring core container contains the following:  
**a. Core:** Core module provides the fundamental features of spring framework like IoC and DI.  
**b. Bean:** Bean module provides the BeanFactory.  
**c. Context:** Context module provides a way to access any object. ApplicationContext interface is the main part of Context module.  
**d. Expression language:** Expression language module provides a way to manipulate objects at runtime.

3. Data Access/Integration contains the following:

**a. JDBC:** JDBC modules provides a JDBC-abstraction layer.  
**b. ORM:** ORM provides integration layers for object-relational mapping APIs like JPA, and Hibernate etc.  
**c. OXM:** OXM module provides an abstraction layer for Object/XML mapping APIs like JAXB, Castor and XMLBeans etc.  
**d. JMS:** JMS module provides feature of message processing.  
**e. Transaction:** Transaction module provides the facility of transaction management for classes like POJOs etc.

4. Web:

Web module consist of Web, Web-Servlet, Web-Struts, Web-Socket and Web-Portlet which provides the facility of creating web applications.

5. AOP:

AOP module provides aspect-oriented programming implementation which provides the facility to define method-interceptors.

6. Instrumentation:

Instrumentation module provides class instrumentation support and class loader implementations

Spring Modules

The Spring Framework is divided into several modules based on their services. These modules are:

* **Spring Core Container:** It is the core module of the Spring that provides containers like BeanFactory and ApplicationContext.
* **Inversion of Control(DI):** It is also known as dependency injection and used to configure application components and lifecycle management of Java objects.
* Aspect-Oriented Programming**:** This module enables implementing cross-cutting concerns inside the Spring framework such as transaction management, remote access, etc.
* Data Access**:** It helps with working with database systems by using Java Database Connectivity (JDBC) and ORM (Object-Relational Mapping) tools.
* **Model View Controller:** It is also known as the MVC model that helps to create web-based applications and RESTful Web services.
* **Authentication And Authorization:** It is used to configure security processes within the framework by using the Spring Security (a sub-project of Spring).
* **Messaging:** Spring uses a message listener object to convey the message by using JMS (Java Message Service) which is the improvement of JMS API.
* **Transaction Management:** It consists of several transaction management APIs and coordinates transactions for Java objects.
* **Remote Management:** It helps to configure Java objects for local or remote by using Java Management Extensions (JMX).
* **Testing:** Testing module that helps in writing unit tests and integration tests.

Spring 5 Updates

* It requires a minimum Java version is **Java 8** or higher.
* Deprecated some integration: Tiles, Gavava, Velocity, Portrait, etc.
* Spring MVC is upgraded to use **Servlet API 4.0**.
* Support for the new Reactive programming framework: **Spring WebFlux**.

History and Versions of Spring

The following table contains Spring Framework releases with the corresponding year. Its first version was released on **1 October 2002**.

|  |  |
| --- | --- |
| **Version** | **Date (Year)** |
| 0.9 | October 2002 |
| 1.0 | June 2003 |
| 2.0 | October 2006 |
| 3.0 | December 2009 |
| 4.0 | December 2013 |
| 5.0 | September 2017 |

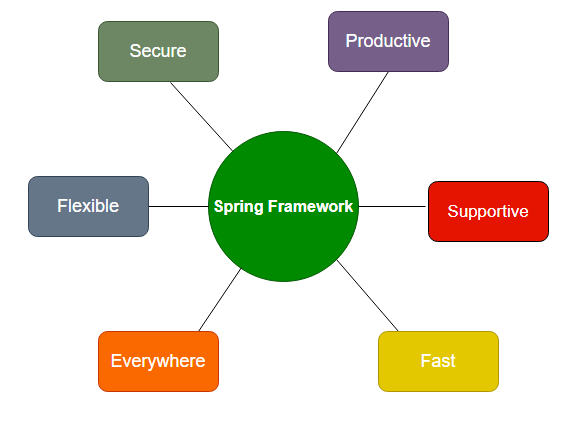
Advantages of Spring framework

1. Light weight: using pojo
2. Non invasive approach: no extend class and implements interface given by Spring API.
3. Loose coupling : DI
4. Modular fashion: use needed module and ignore rest
5. Easy testing
6. Transaction management interface
7. No need application server
8. MVC framework

# Spring Features

Java Spring Framework is full of features and provides and helps to create Java-based scalable applications. Here, we are discussing some features. Although these are not limited as spring provides dozens of variety of projects such as Spring Data, Spring Cloud, Spring Boot, etc. The following are the Features of the Spring Framework.

* Flexible
* Productive
* Fast
* Secure
* Supportive
* Everywhere



**Fig: Spring Features**

## Spring is Everywhere

Spring is one of the most popular frameworks worldwide. It is used for enterprise application development in Java. We can use it to create web services, web applications, cloud-based services, etc. It is used by world tech organizations like Google, Alibaba, Amazon, Oracle, etc. Java developers all over the world trust Spring’s libraries. It is used almost everywhere whether streaming TV, connection IoT, eCommerce applications, Banking, etc.

## Spring is Flexible

Spring provides flexible third-party libraries and extensions that help developers to build applications. The **Inversion of Control (IoC)** and **Dependency Injection (DI)** are the main features of Spring that made it flexible for creating enterprise applications. Spring provides several tools that help developers to build secure, reactive, and cloud-based microservices for the web, even you can use it for complex streaming data flows for enterprise applications.

## Spring is Productive

Spring is a productive framework no doubt and after adding the Spring Boot project Spring framework has transformed into a more productive framework. Spring Boot is combined with all the necessities and auto-configured settings. It has an embedded webserver to make microservices development faster. The most important is the integration of spring projects that help to create applications in a row. For example, to create an application it provides a framework, to connect databases it provides Spring Data, to work with Cloud, it provides Spring cloud and for security, it provides Spring security. So, what we need to do is just put them into our application based on the requirement and our application is ready.

## Spring is Fast

The performance of the Spring framework is **super** due to its design and architecture. Spring foundation is focused on the performance that gives the application a fast startup at the starting point, stable execution, and fast shutdown. For better performance and efficiency, Spring projects support the reactive programming model. Spring provides Spring Initializer tool to start with application quickly. With Spring Boot that already equipped with the tools like embedded web server, auto-configured helps developers to build applications with ease. Adding of **LiveReload in Spring DevTools** removes the issue and need for a server restart.

## Spring is Secure

Spring is secure by nature itself, along with the security provided by Java language too, but for more security purposes we can use **Spring Security**. Spring Security is one of the projects of Spring that is designed to handle the security of any Spring application. Since it is part of the Spring framework, hence it is easy to integrate with the application. Spring quickly handle and deal with security issues and handle them. It closely monitors third-party dependencies, and regular updates to keep our data and applications safe and secure.

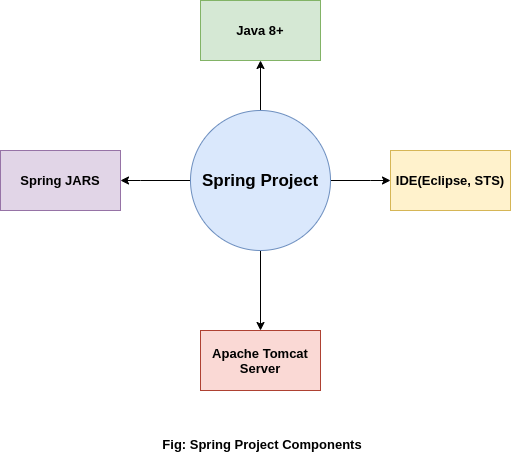
## Spring is Supportive

The Spring has a vast, global, enormous, and diverse community for all the developers worldwide. Spring provides supports for all no matter where you are residing and what level of knowledge you have. It helps for all ages and capabilities, from complete beginners to industry experts. Spring community provides a variety of resources like videos, guides, tutorials, meetups, support, or even formal training and certification.

# Spring Environment Setup

To set up a development environment for Spring Framework, we need to have the following tools:

* Install Java
* Install Eclipse
* Install Tomcat Server
* Download Spring JARs



So let's download and install these tools in our local system so that we can execute the Spring application successfully.

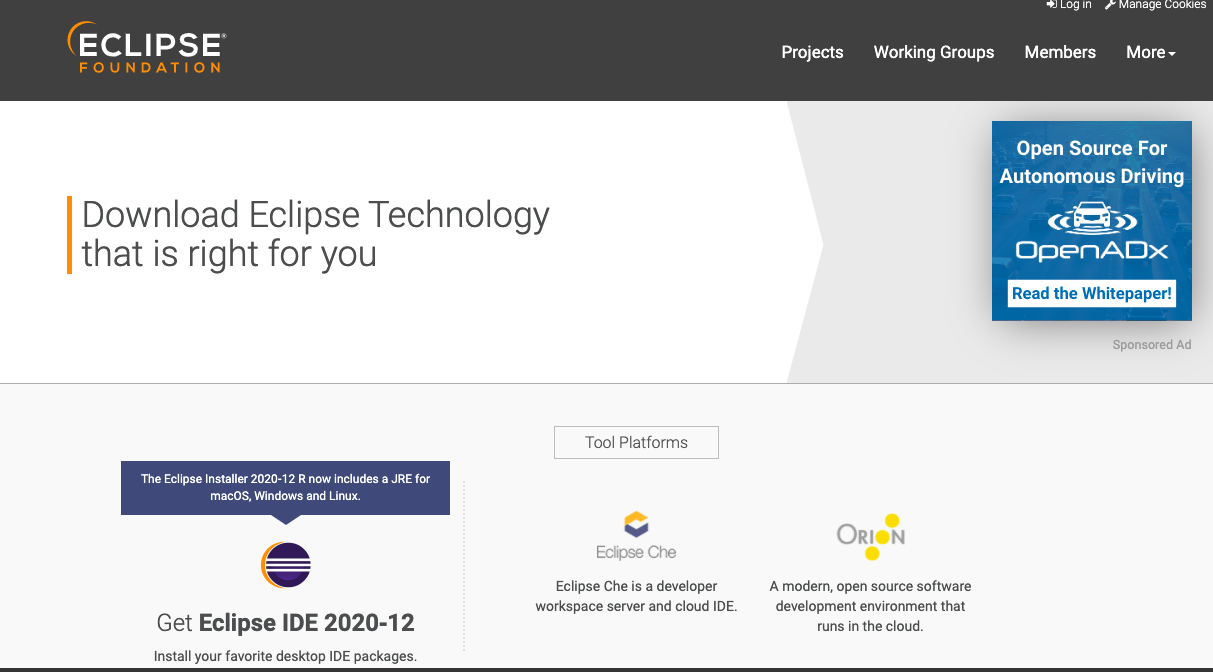
## Step 1: Install Java

Java is a programming language that is used by the Spring framework to create an application. So it is our first task to [Install Java](https://www.studytonight.com/post/step-by-step-guide-to-install-jdk-11-on-windows) and [set up its classpath](http://studytonight.com/java/setting-classpath-for-java.php) on our computer.

## Step 2: Download and Install Eclipse

Eclipse is an IDE(Integrated Development Environment) that is used for software development. It provides a platform to build applications a single place. We can handle multiple tools and by using its controls.

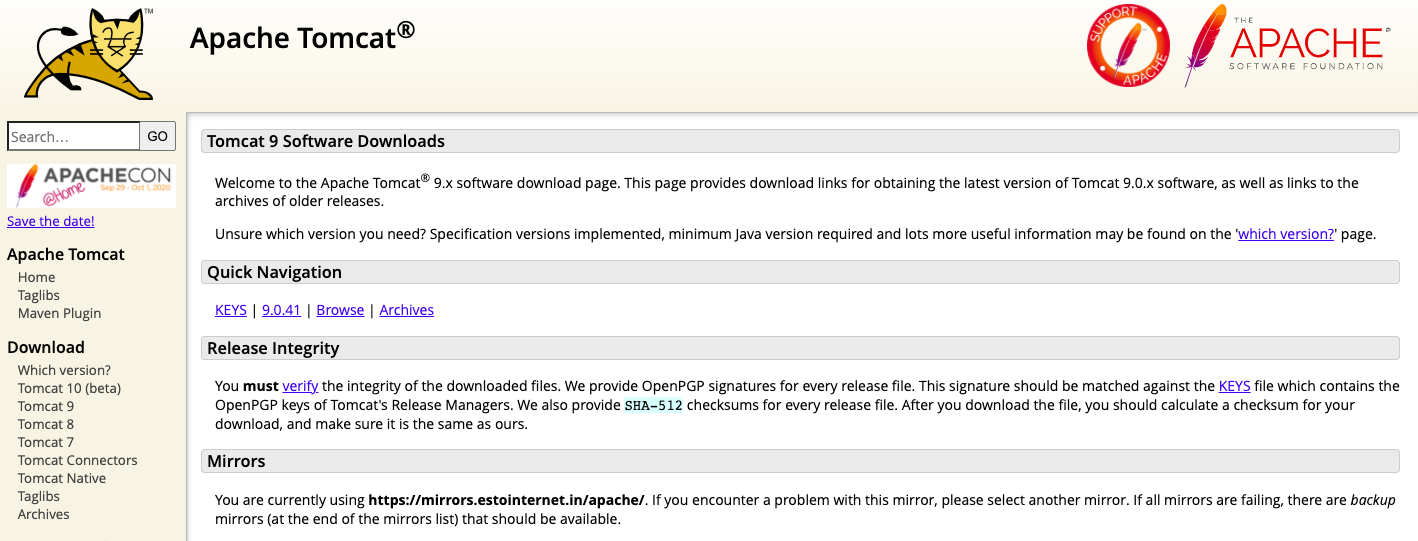
To download the Eclipse, visit the official site of the Eclipse and [download](https://www.eclipse.org/downloads/) it. After downloading the **Zip file**, extract and install it.



While installing Eclipse, make sure **Java**has **installed** along with its **classpath**because Eclipse picks the JVM path during installation**.**

## Step 3: Download Tomcat Server

Tomcat is a server that allows testing web applications on the local machine. Here, we will use it to test our Spring application. We can download it from the official site of [Apache Foundation](https://tomcat.apache.org/). After downloading, extract it, we will use it to run the application on the server.



We recommend you to download version 9 of the Tomcat server.

## Step 4: Download Spring JARs

To work with the Spring framework, we must have Spring JARs that are basically Java files that contain packages, classes, and interfaces to create the spring application.

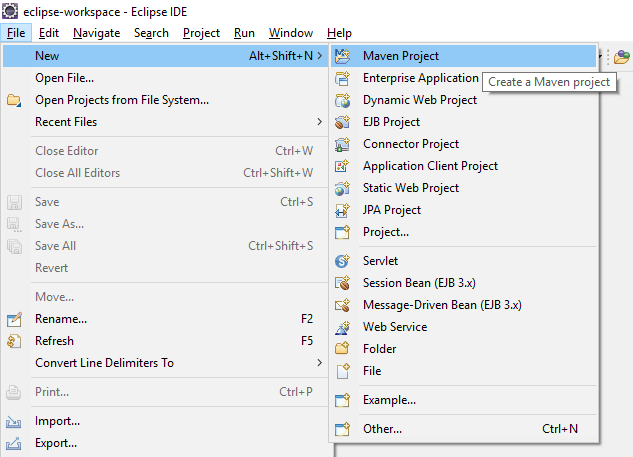
There are two ways to have these JARs. Either download them from the Spring official site and put them into the Spring project or create a [Maven project](https://www.studytonight.com/spring-framework/spring-maven-project) that will automatically download these JARs into your project.

If you want to download these JARs directly then visit the official repository of Spring by clicking here [Download Spring JARs](https://repo.spring.io/release/org/springframework/spring/). Pick the latest JARs from there, extract them, and put them into the Spring project's lib folder.

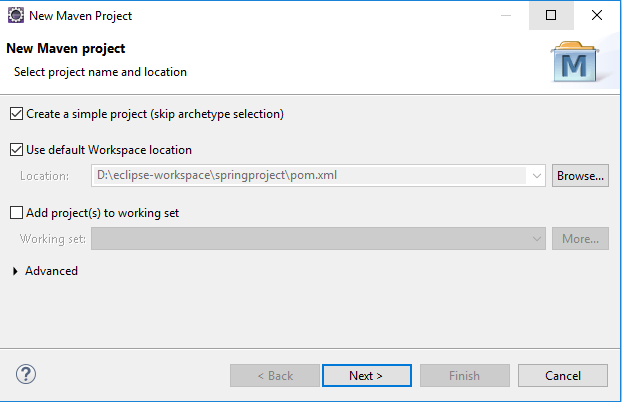
**After** doing all the above steps, now we can create a Spring application by using the Eclipse IDE. **In our next topic,** we will **create a spring application**.

# Spring Maven Project

Step1: Create Maven Project

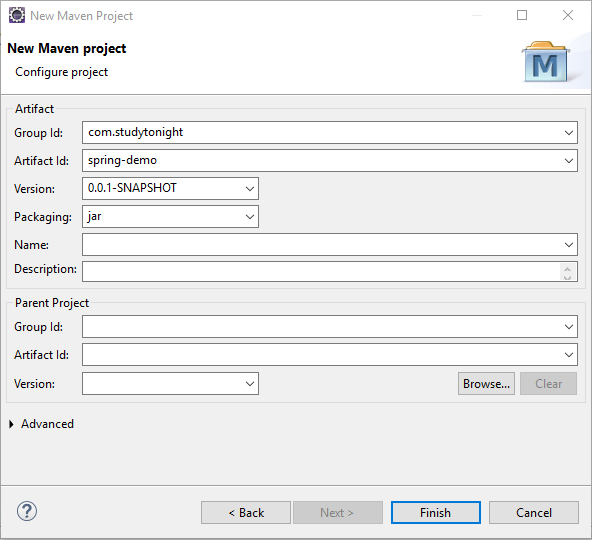
Open Eclipse and clike on **File** menu and then select **New --> Maven Project** as we did in the given screenshot.

After selecting maven, it will open a window like below and will ask to select archetype. Here, we clicked on Checkbox and then **press next** button.



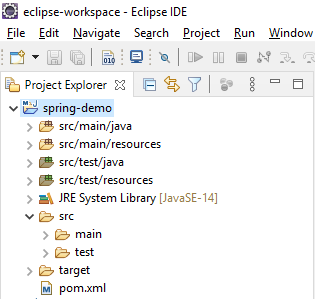
Step 2: Configure the Project

Set project name by providing group id and artifact id. Artifact id represents the project name, So give a meaning name for your project after that select packaging that specifies the plugin goals that are executed during each **Maven build** phase. After that **press Finish** button and your maven project is ready.



Maven Project Stucture

After above step, Eclipse creates a new project that look like the below screeshot. This project contains a **pom.xml** file that is used to configure maven project. We will discuss that later in our tutorial.



Fine, till here, we have created a maven based spring project successfuly. Now, **in next chapter, we will learn to create a simple hello world application** to understand the flow and structure of the application.

# Simple Hello World Application

first create a simple hello world application.

We have created a couple of files and updated the default **pom.xml** file with Spring 5 dependencies. So, first, add these dependencies into the **pom.xml** file.

### Spring 5 Dependencies For Maven Project

In the below-specified pom.xml, we will be adding all the Spring 5 dependencies:

<dependencies>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-core</artifactId>

<version>${spring.version}</version>

</dependency>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-context</artifactId>

<version>${spring.version}</version>

</dependency>

</dependencies>

<properties>

<spring.version>5.2.8.RELEASE</spring.version>

</properties>

After adding these dependencies into the file. Let's create some Java files and XML configuration files. These files are:

* Hello.java
* HelloWorldService.java
* applicationContext.xml
* pom.xml

Now, let's create all the above mentioned Java and XML files for our spring project.

### Hello.java

This file contains code for loading the application context file which is configured to load bean.

package com.example;

import org.springframework.context.support.ClassPathXmlApplicationContext;

public class Hello {

public static void main(String[] args) {

// loading the Bean and XML definitions from the given XML file

ClassPathXmlApplicationContext context = new ClassPathXmlApplicationContext("applicationContext.xml");

HelloWorldService obj = context.getBean(HelloWorldService.class);

obj.hello();

context.close();

}

}

### HelloWorldService.java

This file contains the code that will print the "Hello Spring" message to the console. Create this file with **"HelloWorldService.java"** name.

package com.example;

public class HelloWorldService {

public void hello() {

System.out.println("Hello Spring!");

}

}

### applicationContext.xml

This is the configuration file for the Spring project. We can name it anything, but now save it as **applicationContext.xml** in your maven 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:p="http://www.springframework.org/schema/p"

xmlns:aop="http://www.springframework.org/schema/aop" xmlns:context="http://www.springframework.org/schema/context"

xmlns:jee="http://www.springframework.org/schema/jee" xmlns:tx="http://www.springframework.org/schema/tx"

xmlns:task="http://www.springframework.org/schema/task"

xsi:schemaLocation="http://www.springframework.org/schema/aop http://www.springframework.org/schema/aop/spring-aop-3.2.xsd http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans-3.2.xsd http://www.springframework.org/schema/context http://www.springframework.org/schema/context/spring-context-3.2.xsd http://www.springframework.org/schema/jee http://www.springframework.org/schema/jee/spring-jee-3.2.xsd http://www.springframework.org/schema/tx http://www.springframework.org/schema/tx/spring-tx-3.2.xsd http://www.springframework.org/schema/task http://www.springframework.org/schema/task/spring-task-3.2.xsd">

<context:component-scan base-package="com.examples" />

<bean id="helloWorldService"

class="com.example.HelloWorldService">

</bean>

</beans>

### pom.xml

This file is a part of the maven project and used to add dependencies for our project. For our project, we added spring dependencies. This is the latest by the time of this project. Although we can get these latest dependencies from the maven repository. [Spring Dependencies](https://mvnrepository.com/artifact/org.springframework/spring-core).

<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 https://maven.apache.org/xsd/maven-4.0.0.xsd">

<modelVersion>4.0.0</modelVersion>

<groupId>springproject</groupId>

<artifactId>springproject</artifactId>

<version>0.0.1-SNAPSHOT</version>

<packaging>war</packaging>

<dependencies>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-core</artifactId>

<version>${spring.version}</version>

</dependency>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-context</artifactId>

<version>${spring.version}</version>

</dependency>

</dependencies>

<properties>

<spring.version>5.2.8.RELEASE</spring.version>

</properties>

<build>

<sourceDirectory>src</sourceDirectory>

<plugins>

<plugin>

<artifactId>maven-compiler-plugin</artifactId>

<version>3.8.1</version>

<configuration>

<source>1.8</source>

<target>1.8</target>

</configuration>

</plugin>

<plugin>

<artifactId>maven-war-plugin</artifactId>

<version>3.2.3</version>

<configuration>

<warSourceDirectory>WebContent</warSourceDirectory>

</configuration>

</plugin>

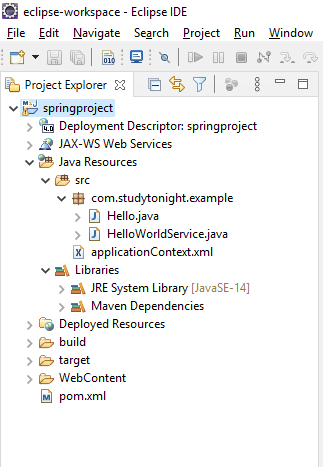
</plugins>

</build>

</project>

## Project Structure:

After creating all the above files, we end up with the following directory structure. Our Spring project should have the following directory structure.



## Run the Application:

# Spring IOC Container:create,wire,configure and manage object during their complete life cycle.

Spring IoC Container is a core part of the Spring framework which is used to manage the application bean. It injects dependencies when a bean is created and **manages the bean life cycle** during execution.

The fundamental tasks of Spring IoC are:

* Instantiating
* Configuring, and
* Assembling Bean

The IOC container gets configuration related information from the Spring configuration file. That can be either **XML** or **Java** files.

The container uses [**dependency injection (DI)**](https://www.studytonight.com/spring-framework/spring-constructorbased-dependency-injection) to manage the components that make up an application.

Spring provides two types of IOC containers:

* BeanFactory : light weight
* Application Context: heavy weight

## Spring BeanFactory Interface

It is an IoC container that is responsible for maintaining beans and their dependencies. It is basically an interface that provides basic functionalities.

### BeanFactory Interface Methods:

Following are the BeanFactory Interface methods:

|  |  |
| --- | --- |
| **Method Name** | **Description** |
| boolean containsBean(String name) | It checks whether this bean factory contains a bean definition or externally registered singleton instance with the given name. |
| String[] getAliases(String name) | It returns the aliases for the given bean name if any. |
| <T> T getBean(Class<T> requiredType) | It returns the bean instance that uniquely matches the given object type if any. |
| <T> T getBean(Class<T> requiredType, Object... args) | It returns an instance, which may be shared or independent, of the specified bean. |
| Object getBean(String name) | It returns an instance, which may be shared or independent, of the specified bean. |
| <T> T getBean(String name, Class<T> requiredType) | It returns an instance, which may be shared or independent, of the specified bean. |
| Object getBean(String name, Object... args) | It returns an instance, which may be shared or independent, of the specified bean. |
| <T> ObjectProvider<T> getBeanProvider(Class<T> requiredType) | It returns a provider for the specified bean, allowing for lazy on-demand retrieval of instances, including availability and uniqueness options. |
| <T> ObjectProvider<T> getBeanProvider(ResolvableType requiredType) | It returns a provider for the specified bean, allowing for lazy on-demand retrieval of instances, including availability and uniqueness options. |
| Class<?> getType(String name) | It determines the type of the bean with the given name. |
| Class<?> getType(String name, boolean allowFactoryBeanInit) | It determines the type of the bean with the given name. |
| boolean isPrototype(String name) | It checks whether this bean a prototype. |
| boolean isSingleton(String name) | It checks whether this bean a shared singleton. |
| boolean isTypeMatch(String name, Class<?> typeToMatch) | It checks whether the bean with the given name matches the specified type. |
| boolean isTypeMatch(String name, ResolvableType typeToMatch) | It checks whether the bean with the given name matches the specified type. |

## ==org.sprinframework.beans.factory.BeanFactory is the the interface and XmlBeanFactory is an implementation class of it..

## It is a simple container which provide support for DL.

## Resource res=new ClassPathResource(“spring.xml”);

## BeanFactory factory=new XmlBeanFactory(res);

## Or

## Resource res= new FileSystemResource("SpringStudent.xml"));

## BeanFactory factory=new XmlBeanFactory(res);

## It is light weight and support lazy intailization by default.

## Spring ApplicationContext Sub-Interface

The ApplicationContext is a sub-interface of BeanFactory and provides more enterprise like functionality. It adds Application-layer specific contexts such as the WebApplicationContext for web applications.

There are several implementations for this ApplicationContext interface such as:

* ClassPathXmlApplicationContext
* XmlWebApplicationContext
* FileSystemXmlApplicationContext

### ApplicationContext Methods:

The following are the methods in ApplicationContext Interface.

|  |  |
| --- | --- |
| **Method Name** | **Description** |
| String getApplicationName() | It returns a name for the deployed application that this context belongs to. |
| AutowireCapableBeanFactory getAutowireCapableBeanFactory() | It exposes AutowireCapableBeanFactory functionality for this context. |
| String getDisplayName() | It returns a friendly name for this context. |
| String getId() | It returns the unique id of this application context. |
| ApplicationContext getParent() | It returns the parent context, or null if there is no parent and this is the root of the context hierarchy. |
| long getStartupDate() | It returns the timestamp when this context was first loaded. |

## ApplicationContext org.springframework.context.ApplicationContext is the interface and ClassPathXmlApplicationContext is an implementation class of it. ApplicationContext container includes all functionality of the BeanFactory container with some extra functionality like internationalization, event listeners etc.

## ApplicationContext applicationContext =

## new ClassPathXmlApplicationContext("spring configuration file");

## Note: As ApplicationContext provides extra functionality including all given by BeanFactory it is better to use ApplicationContext container.

## It is heavy weight due to eager loading support by default.

## Difference Between BeanFactory and ApplicationContext

Both the interfaces(BeansFactory and ApplicationsContext) acts as the IoC container. The BeanFactory interface is a base interface and provides all the basic functionalities to create and run the IoC container while the ApplicationContext interface is a subinterface of the BeanFactory interface that adds some extra functionalities like simple integration with Spring's AOP, message resource handling (for I18N), application layer specific context, etc. So, we can use ApplicationContext for better features.

## How to Configure the IoC Container?

This is 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

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

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

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

</bean>

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

</beans>

# ClassPathXmlApplicationContext − This container loads the definitions of the beans from an XML file. Here you do not need to provide the full path of the XML file but you need to set CLASSPATH properly because

# this container will look like bean configuration XML file in CLASSPATH.

/\*FileSystemXmlApplicationContext to create the factory bean after

\* loading the bean configuration file from the given path.

\* takes care of creating and initializing all the objects

\* ie. beans mentioned in the XML bean configuration file.

\* \*/

/\* getBean() method of the created context.

\* This method uses bean ID to return a generic object,

\* which finally can be casted to the actual object.

\* Once you have an object, you can use this object to call any class method.

\*

\* \*/

# Spring Bean

A spring bean represents an object that is created, configured and managed by spring container.

A spring bean is created by configuration metadata passed to the spring container which tells the container about bean creation, bean lifecycle and bean dependencies.

## Spring bean properties:

|  |  |
| --- | --- |
| **Bean Properties** | **Description** |
| **1. class** | It is mandatory and specify the bean class which is used to create the bean. |
| **2. name** | It specifies the bean unique identifier. |
| **3. scope** | It specifies the scope of the objects created from a particular bean definition. |
| **4. constructor-arg** | It is used to inject the dependencies. |
| **5. properties** | It is used to inject the dependencies. |
| **6. autowiring mode** | It is used to inject the dependencies. |
| **7. lazy-initialization mode** | It tells the IoC container to create a bean instance when it is first requested, rather than at startup. |
| **8. initialization method** | It is a callback method to be called just after all necessary properties on the bean have been set by the container. |
| **9. destruction method** | It is a callback to be called when the container containing the bean is destroyed. |

<bean id="..." class="..." lazy-init="true" scope=””>

//bean configuration

</bean>

# Spring Bean

Bean is an object in Spring that is managed by the Spring IoC Container. Spring creates bean with the configuration metadata that we have supplied in the **<bean>** tag of the XML file. We provide metadata to the IoC container either by using the XML file or by Java annotations.

During metadata configuration, we provide bean definitions with some optional attributes such as:

* The fully qualified name of Bean class name. such as **com.studytonight.community.Reader**.
* Bean behavior such as Bean scope, lifecycle callback, etc.
* Bean dependencies (references to other beans) that are needed for the bean.

### The <bean> Tag Structure

<bean id="demoBean" class="com.examples.DemoBean" />

The **id** attribute sets a unique id for the class specified by the **class** attribute.

## Bean Definition Properties

The following table contains the <bean> tag properties that are used to configure the bean in the configuration file.

| **Bean Property** | **Description** |
| --- | --- |
| Class | This property is used to specify the class for which an object is created. |
| Name | This property is used to specify an identifier for a bean that is unique. |
| Scope | It specifies the scope of beans such as singleton or prototype. |
| Constructor arguments | It is used for constructor based dependency injection. |
| Properties | This property is used for property-based dependency injection. |
| Autowiring mode | It is used to set bean auto wiring. |
| collaborators and lazy initialization mode | It is used to set lazy bean initialization. |
| Initialization method | It is used to set the initialization method to execute at bean initialization. |
| Destruction method | It is used to set destructive methods that execute before destroying of bean object. |

## 1. Naming Bean

To set the name of a bean in XML-based configuration, we use the **id,** **name** attributes, or both. The **id** attribute lets us specify exactly one id.

While setting bean names we are required to follow the naming conventions it means bean names start with a lowercase letter and are camel-cased.

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

<!-- Configuration for this bean go here -->

</bean>

## 2.Lazy Initialization

By default, ApplicationContext implementations eagerly create and configure all singleton beans as part of the initialization process.(application context)

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

<bean id = "..." class = "..." lazy-init = "true">

<!-- Configuration for this bean go here -->

</bean>

### Init Method

This attribute is used to specify the method that executes at bean initialization time. Syntax of the bean tag and attribute is given below.

<bean id = "..." class = "..." init-method = "...">

<!-- Configuration for this bean go here -->

</bean>

### Destruction Method

This attribute is used to specify the method that executes at bean destroy time. Syntax of the bean tag and attribute is given below.

<bean id = "..." class = "..." destroy-method = "...">

<!-- Configuration for this bean go here -->

</bean>

# Spring Bean Scopes Examples

## Spring bean scopes:

As we discussed that spring container is responsible for creating and managing bean object. Spring provides the facility to return the same instance or a new instance each time when one is needed. It depends upon the bean scope.

## Spring framework bean scopes:

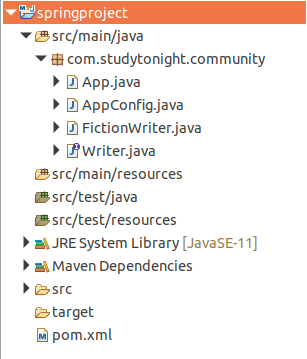
|  |  |
| --- | --- |
| **Bean Scope** | **Description** |
| **1. singleton** | It scopes the bean definition to a single instance per spring container. It is the default scope. Spring container keeps it into cache and returns the same instance each time a request for that particular bean is made. |
| **2. prototype** | It scopes a single bean definition to have new bean instance each time a request for that particular bean is made. |
| **3. request** | It scopes a bean definition to an HTTP request. |
| **4. session** | It scopes a bean definition to an HTTP session. |
| **5. global-session** | It scopes a bean definition to a global HTTP session. |

***Note: The request, session and global-session are only available in the context of a web-aware ApplicationContext.***

Example: Bean Singleton Scope (Default Scope) Using Annotation

Let's create an example to mark a bean scope as default and check whether it has a default scope or not.

Project Structure



Project Files Source Code:

**//App.java**

This file contains the code to create an IOC container for our application. AnnotationConfigApplicationContext class is used to create an object for application context. Here two bean objects are created and checked whether both are equal or not.

package com.bean;

import org.springframework.context.annotation.AnnotationConfigApplicationContext;

public class App {

public static void main(String[] args) {

AnnotationConfigApplicationContext context = new AnnotationConfigApplicationContext(AppConfig.class);

Writer writer1 = context.getBean("fictionWriter", Writer.class);

Writer writer2 = context.getBean("fictionWriter", Writer.class);

boolean isSame = writer1 == writer2;

System.out.println("Instance One :"+writer1);

System.out.println("Instance One :"+writer2);

System.out.println("Both bean instances are same: "+isSame);

//writer.write();

// Close the context

context.close();

}

}

**// AppConfig.java**

This is a configuration file in Java which is an alternate of the applicationContext.xml file that we created for the XML-based configuration example. The @Configuration annotation indicates that this is not a simple class but a configuration class and the @ComponentScan annotation is used to indicate the component location in our spring project.

package com.bean;

import org.springframework.context.annotation.ComponentScan;

import org.springframework.context.annotation.Configuration;

@Configuration

@ComponentScan("com.bean")

public class AppConfig {

}

**// FictionWriter.java**

package com.bean;

import org.springframework.context.annotation.Scope;

import org.springframework.stereotype.Component;

@Component

@Scope

public class FictionWriter implements Writer {

@Override

public void write() {

System.out.println("Write Fiction Novels...");

}

}

**// Writer.java**

package com.bean;

public interface Writer {

void write();

void getAward();

}

**// pom.xml**

This file contains all the dependencies of this project such as spring jars, servlet jars, etc. Put these dependencies into your project to run the application.

<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 https://maven.apache.org/xsd/maven-4.0.0.xsd">

<modelVersion>4.0.0</modelVersion>

<groupId>com.apolis</groupId>

<artifactId>springproject</artifactId>

<version>0.0.1-SNAPSHOT</version>

<dependencies>

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

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-core</artifactId>

<version>${spring.version}</version>

</dependency>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-context</artifactId>

<version>${spring.version}</version>

</dependency>

</dependencies>

<properties>

<spring.version>5.2.8.RELEASE</spring.version>

</properties>

<build>

<sourceDirectory>src</sourceDirectory>

<plugins>

<plugin>

<artifactId>maven-compiler-plugin</artifactId>

<version>3.8.1</version>

<configuration>

<source>1.8</source>

<target>1.8</target>

</configuration>

</plugin>

</plugins>

</build>

</project>

Example: Bean Scope Prototype

This is another type of bean scope and in this case, two bean objects of the same class are not equal.

**// FictionWriter.java**

package com.bean;

import org.springframework.context.annotation.Scope;

import org.springframework.stereotype.Component;

@Component

@Scope("prototype")

public class FictionWriter implements Writer {

@Override

public void write() {

System.out.println("Write Fiction Novels...");

}

}

##### **Prototype scope of a bean**

**Prototype** scope means container creates new instance every time we ask container to provide the bean using **getBean()** method.  
**Prototype** scope must be defined **explicitly**

In simple words, each **context.getBean()** method returns a new object.

//=================================================================

Injection:

Injection is a process of passing the dependency to a dependent object.

Dependency Injection (DI):

Dependency Injection (DI) is a design pattern that implements inversion of control principle for resolving dependencies. It allows a programmer to remove hard coded dependencies so that the application becomes loosely coupled and extendable.

**public** **class** Student {

**private** Address address;

**public** Student() {

address = **new** Address();

}

}

In above example Student class requires an Address object and it is responsible for initializing and using the Address object. If Address class constructor is changed in future then we have to make changes in Student class also. This approach makes tight coupling between Student and Address objects. We can resolve this problem using dependency injection design pattern. i.e. Address object will be implemented independently and will be provided to Student when Student is instantiated by using constructor-based or setter-based dependency injection.

Another example:

Consider a case when we want to implement sorting functionality. We have one implementation class InsertionSort which uses insertion sort algorithm for sorting. Now in our sorting test class, we will create the InsertionSort class object and call its sorting method.

public class SortingTest {

private SortAlgorithm sortAlgorithm;

public SortingTest() {

sortAlgorithm= new InersionSort();

}

}

Here SortAlgorithm is an interface which is implemented by InsertionSort class. Now let us think that we have one more sorting implementation using merge sort algorithm and we want to sort the numbers with merge sort algorithm. In this case also we have to change our implementation class i.e. we have to change SortingTest() constructor by replacing InersionSort() with MergeSort(). It again result into tight coupling. We can resolve this problem using dependency injection design pattern.

Types of dependency Injection:

1. Constructor-based Dependency Injection.

2. Setter-based Dependency Injection.

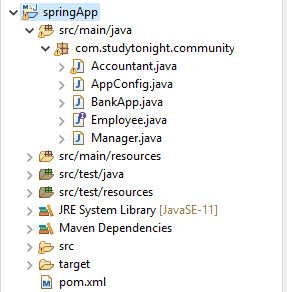
Dependency Injection is a technique by which an object defines its dependencies. The IOC container then injects these dependencies during bean creation. This process is fundamentally the inverse and known as Inversion of Control as well. Dependency Injection makes our code loosely coupled. It is classified into two major categories Constructor-based dependency injection and Setter-based dependency injection.

a [Maven-based Spring Project](https://www.studytonight.com/spring-framework/spring-maven-project) and that contains the following files.

* BankApp.java
* AppConfig.java
* Employee.java
* Manager.java
* Accountant.java
* pom.xml

And the following is a maven project structure created for the Spring application.

**Project Structure:**



The files created into the above project contains the following code. See the files below.

## Files Source Code:

**// BankApp.java**

This file contains the code to create an IOC container for our application. The AnnotationConfigApplicationContext class is used to create an object for application context.

package com.bean;

import org.springframework.context.annotation.AnnotationConfigApplicationContext;

public class BankApp {

public static void main(String[] args) {

AnnotationConfigApplicationContext context = new AnnotationConfigApplicationContext(AppConfig.class);

Manager manager = context.getBean(Manager.class);

manager.callMetting();

context.close();

}

}

**// AppConfig.java**

This is a configuration file in Java which is an alternate of the applicationContext.xml file that we created for the XML-based configuration example. The @Configuration annotation indicates that this is not a simple class but a configuration class and the @ComponentScan annotation is used to indicate the component location in our spring project.

package com.bean;

import org.springframework.context.annotation.ComponentScan;

import org.springframework.context.annotation.Configuration;

@Configuration

@ComponentScan("com.bean")

public class AppConfig {

}

**// Employee.java**

This is an interface Employee that contains a doWork() abstract method. Each class that implements this interface will have to override the doWork() method.

package com.bean;

public interface Employee {

void doWork();

}

**// Accountant.java**

This is a component class that is marked using @Component annotation. It implements the Employee interface and overrides its method doWork().

package com.bean;

import org.springframework.stereotype.Component;

@Component

public class Accountant implements Employee{

public Accountant() {

System.out.println("Inside Accountant Constructor");

}

public void doWork() {

System.out.println("Audit the accounts...");

}

}

**// Manager.java**

This is another component class that is marked using the @Component annotation and implements the Employee interface. In this class, we are implementing constructor-based dependency injection. See, the Manager class calls a method of Accountant class by using the Accountant class object which is instantiated inside the Manager class constructor. See the example below.

package com.bean;

import org.springframework.stereotype.Component;

@Component

public class Manager implements Employee{

Accountant accountant;

public Manager(Accountant accountant) {

System.out.println("manager constructor");

this.accountant = accountant;

}

public void doWork() {

System.out.println("Manage the branch office");

}

public void callMetting() {

accountant.doWork();

}

}

**// pom.xml**

This file contains all the dependencies of this project such as spring jars, servlet jars, etc. Put these dependencies into your project to run the application.

<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 https://maven.apache.org/xsd/maven-4.0.0.xsd">

<modelVersion>4.0.0</modelVersion>

<groupId>com.app</groupId>

<artifactId>springApp</artifactId>

<version>0.0.1-SNAPSHOT</version>

<dependencies>

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

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-core</artifactId>

<version>${spring.version}</version>

</dependency>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-context</artifactId>

<version>${spring.version}</version>

</dependency>

<dependency>

<groupId>javax.annotation</groupId>

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

<version>1.3.2</version>

</dependency>

</dependencies>

<properties>

<spring.version>5.2.8.RELEASE</spring.version>

</properties>

<build>

<sourceDirectory>src</sourceDirectory>

<plugins>

<plugin>

<artifactId>maven-compiler-plugin</artifactId>

<version>3.8.1</version>

<configuration>

<source>1.8</source>

<target>1.8</target>

</configuration>

</plugin>

</plugins>

</build>

</project>

### Run the Application

## Configuration using XML

The above project is configured using Java code only. No XML configuration did there but we can configure it with XML code as well. We just need to create a file **applicationContext.xml** and read it into the **BankApp** class. The **applicationContext.xml** file contains the following code.

**// applicationContext.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:p="http://www.springframework.org/schema/p"

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

http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">

<bean id="accountant"

class="com.bean.Accountant" />

<bean id="manager" class="com.bean.Manager">

<constructor-arg>

<ref bean="accountant" />

</constructor-arg>

</bean>

</beans>

## Injecting Primitive Values into Constructor

Apart from the reference variable, we can inject primitive values like int, float, etc into the constructor. For example, In the Manager class, we are using the int id and string name inside the constructor and injecting values from the **applicationContext.xml** file.

**// Manager.java**

package com.bean;

import org.springframework.stereotype.Component;

@Component

public class Manager implements Employee{

int id;

String name;

public Manager(int id, String name) {

this.id = id;

this.name = name;

}

public void doWork() {

System.out.println("Manage the branch office");

}

public void managerInfo() {

System.out.println("Name: "+name+" Id: "+id);

}

}

**// applicationContext.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:p="http://www.springframework.org/schema/p"

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

http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">

<bean id="manager" class="com.bean.Manager">

<constructor-arg type="int" value="10021" />

<constructor-arg type="java.lang.String" value="Ramesh" />

</bean>

</beans>

### Run the Application

Specify Constructor Argument Name

We can also use the constructor parameter name for value disambiguation, as we did in the below example.

<bean id="manager" class="com.bean.Manager">

<constructor-arg name="id" value="10021"/>

<constructor-arg name="name" value="Ramesh"/>

</bean>

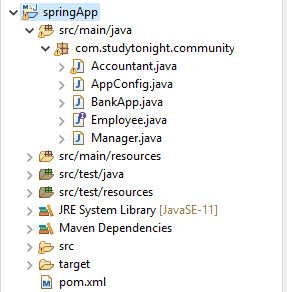
//==============

the setter dependency injection technique to inject values through the setter method. This project is a [Maven-Based Spring Project](https://www.studytonight.com/spring-framework/spring-maven-project) and contains the following files.

* BankApp.java
* AppConfig.java
* Manager.java
* Accontant.java
* Employee.java
* pom.ml

And the following is a maven project structure created for the Spring application.

**Project Structure:**



The files created into the above project contains the following code. See the files below.

Files Source Code:

**//BankApp.java**

This file contains the code to create an IOC container for our application. The AnnotationConfigApplicationContext class is used to create an object for application context.

package com.bean;

import org.springframework.context.annotation.AnnotationConfigApplicationContext;

public class BankApp {

public static void main(String[] args) {

AnnotationConfigApplicationContext context = new AnnotationConfigApplicationContext(AppConfig.class);

Manager manager = context.getBean(Manager.class);

manager.callMetting();

context.close();

}

}

**// AppConfig.java**

This is a configuration file in Java which is an alternate of applicationContext.xml file that we created for the XML-based configuration example. The @Configuration annotation indicates that this is not a simple class but a configuration class and the @ComponentScan annotation is used to indicate the component location in our spring project.

package com.bean;

import org.springframework.context.annotation.ComponentScan;

import org.springframework.context.annotation.Configuration;

@Configuration

@ComponentScan("com.studytonight.community")

public class AppConfig {

}

**// Manager.java**

This is another component class that is marked using the @Component annotation and implements the Employee interface. In this class, we are implementing a **setter dependency injection**. See, the Manager class calls a method of Accountant class by using the Accountant class object which is instantiated inside a setter method. See the example below.

package com.bean;

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

import org.springframework.stereotype.Component;

@Component

public class Manager implements Employee{

Accountant accountant;

@Autowired

public void setAccountant(Accountant accountant) {

this.accountant = accountant;

}

public void doWork() {

System.out.println("Manage the branch office");

}

public void callMetting() {

accountant.doWork();

}

}

**// Accountant.java**

This is a component class that is marked using @Component annotation. It implements the Employee interface and overrides its method doWork().

package com.bean;

import org.springframework.stereotype.Component;

@Component

public class Accountant implements Employee{

public Accountant() {

System.out.println("Inside Accountant Constructor");

}

public void doWork() {

System.out.println("Audit the accounts...");

}

}

**/ Employee.java**

This is an interface Employee that contains a doWork() abstract method. Each class that implements this interface will have to override the doWork() method.

package com.bean;

public interface Employee {

void doWork();

}

**// pom.xml**

This file contains all the dependencies of this project such as spring jars, servlet jars, etc. Put these dependencies into your project to run the application.

<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 https://maven.apache.org/xsd/maven-4.0.0.xsd">

<modelVersion>4.0.0</modelVersion>

<groupId>com.apo</groupId>

<artifactId>springApp</artifactId>

<version>0.0.1-SNAPSHOT</version>

<dependencies>

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

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-core</artifactId>

<version>${spring.version}</version>

</dependency>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-context</artifactId>

<version>${spring.version}</version>

</dependency>

<dependency>

<groupId>javax.annotation</groupId>

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

<version>1.3.2</version>

</dependency>

</dependencies>

<properties>

<spring.version>5.2.8.RELEASE</spring.version>

</properties>

<build>

<sourceDirectory>src</sourceDirectory>

<plugins>

<plugin>

<artifactId>maven-compiler-plugin</artifactId>

<version>3.8.1</version>

<configuration>

<source>1.8</source>

<target>1.8</target>

</configuration>

</plugin>

</plugins>

</build>

</project>

### Run the Application

After successfully completing the project and adding the dependencies run the application and you will get the output as below.

Inside Accountant Constructor

Audit the accounts...

## Configuration using XML

The above project is configured using Java code only. No XML configuration did there but we can configure it with XML code as well. We just need to create a file **applicationContext.xml** and read into the **BankApp** class. The **applicationContext.xml** file contains the following code.

**// applicationContext.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:p="http://www.springframework.org/schema/p"

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

http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">

<bean id="accountant"

class="com.bean.Accountant" />

<bean id="manager" class="com.bean.Manager">

<property name="accountant">

<ref bean="accountant" />

</property>

</bean>

</beans>

Spring Bean Life Cycle

**Let us understand Life Cycle of a bean in Spring**

The beans **life cycle** in spring is one of the most important features to understand.

Class Student{}

{Student s=new Student();

-

-

-

}

In many of the real time applications, it is necessary to perform some of the operations before **initializing** a bean and it is necessary to perform some cleanup operations before the bean is**destroyed**by the container.

In Java, The life cycle of an object begins with **new** keyword.

When we create an object using **new**, that time it calls the series of hierarchical class **constructors**(call goes from bottom to top and hence execution from top to bottom)  
And finally makes the object available.

And when this object will not have any reference, it will be **garbage collected**. This is the simple life cycle of an object in Java.

But in spring, Bean’s life cycle is having few more things to do.

When container starts – a Spring bean needs to be instantiated, based on Java or XML bean definition. It may also be required to perform some post-initialization steps to get it into a usable state.

After that, when the bean is no longer required, it will be removed from the IoC container.

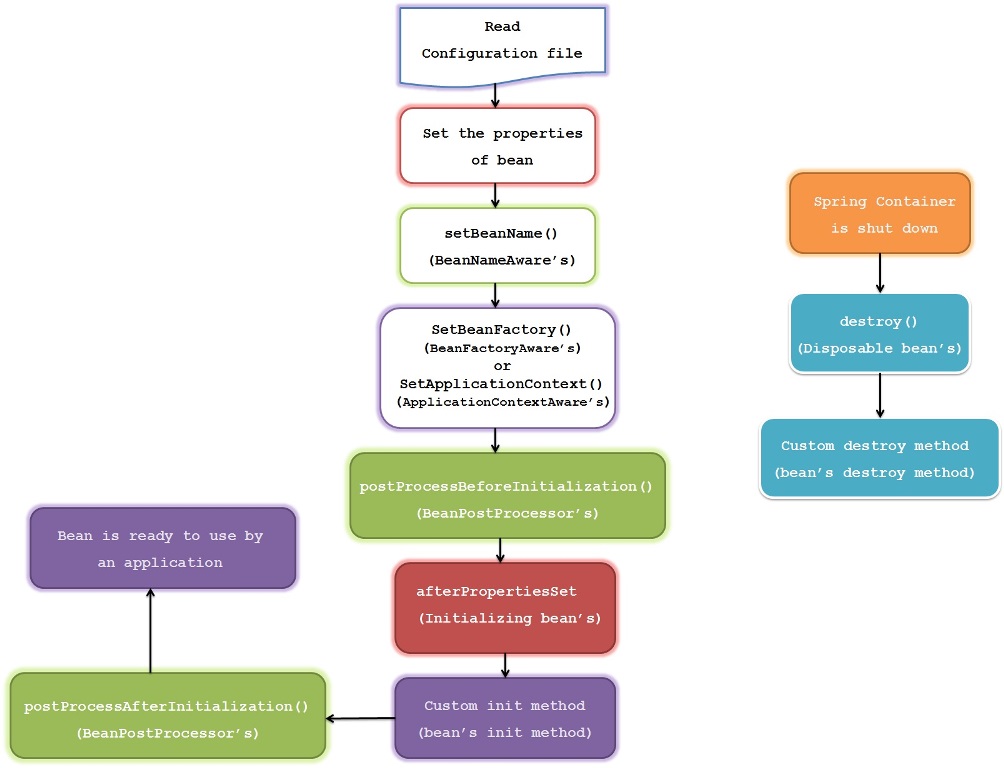
Spring bean factory is responsible for managing the life cycle of beans created through spring container.

#### Life cycle callbacks

Spring bean factory controls the creation and destruction of beans. To execute some custom code, it provides the call back methods which can be categorized broadly in two groups:

* **Post-initialization** call back methods
* **Pre-destruction** call back methods

The spring bean’s life cycle is as shown below



1) Spring container looks for the definition of the **bean** in the spring configuration xml file

2) Spring **instantiate** the bean by calling no argument default constructor of that class,  
If there is **only parameterized constructor** in the class , then bean must be defined in spring xml file with **constructor injection** using which container will instantiate the bean otherwise it will throw bean creation exception.

2) Spring injects the values and references if any into bean’s properties.

3) If the bean implements **BeanNameAware** interface, Spring passes the bean’s ID to the **setBeanName ()** method and executes this method.

4) If the bean implements **BeanFactoryAware** interface, Spring calls the **setBeanFactory ()** method, passing in the bean factory itself and executes this method.

5) If the bean implements **ApplicationContextAware** interface, Spring will call the **setApplicationContext ()** method, passing in a reference to the current application context and executes this method.

6) If the bean implements the **BeanPostProcessor** interface, Spring calls their **postProcessBeforeInitialization ()** method

## Life cycle callback methods

Spring framework provides following **4 ways for controlling life cycle events** of a bean:

1. InitializingBean and DisposableBean callback interfaces
2. \*Aware interfaces for specific behavior
3. Custom init() and destroy() methods in bean configuration file
4. @PostConstruct and @PreDestroy annotations

#### InitializingBean and DisposableBean

The [org.springframework.beans.factory.InitializingBean](https://docs.spring.io/spring-framework/docs/3.0.x/javadoc-api/org/springframework/beans/factory/InitializingBean.html" \o "InitializingBean) interface allows a bean to perform initialization work after all necessary properties on the bean have been set by the container.

The InitializingBean interface specifies a single method:

|  |
| --- |
| InitializingBean.java |
| void afterPropertiesSet() throws Exception; |

This is not a preferrable way to initialize the bean because it tightly couple your bean class with spring container. A better approach is to use “*init-method*” attribute in bean definition in applicationContext.xml file.

Similarly, implementing the [org.springframework.beans.factory.DisposableBean](https://docs.spring.io/spring/docs/1.2.9/api/org/springframework/beans/factory/DisposableBean.html" \o "DisposableBean) interface allows a bean to get a callback when the container containing it is destroyed.

The DisposableBean interface specifies a single method:

|  |
| --- |
| DisposableBean.java |
| void destroy() throws Exception;    A sample bean implementing above interfaces would look like this:      package com.task;    import org.springframework.beans.factory.DisposableBean;  import org.springframework.beans.factory.InitializingBean;    public class DemoBean implements InitializingBean, DisposableBean  {      //Other bean attributes and methods        @Override      public void afterPropertiesSet() throws Exception      {          //Bean initialization code      }        @Override      public void destroy() throws Exception      {          //Bean destruction code      }  } |

#### \*Aware interfaces for specific behavior

Spring offers a range of \*Aware interfaces that allow beans to indicate to the container that they require a certain infrastructure dependency. Each interface will require you to implement a method to inject the dependency in bean.

These interfaces can be summarized as :

|  |  |  |
| --- | --- | --- |
| **AWARE INTERFACE** | **METHOD TO OVERRIDE** | **PURPOSE** |
| ApplicationContextAware | void setApplicationContext (ApplicationContext applicationContext) throws BeansException; | Interface to be implemented by any object that wishes to be notified of the ApplicationContext that it runs in. |
| ApplicationEventPublisherAware | void setApplicationEventPublisher (ApplicationEventPublisher applicationEventPublisher); | Set the ApplicationEventPublisher that this object runs in. |
| BeanClassLoaderAware | void setBeanClassLoader (ClassLoader classLoader); | Callback that supplies the bean class loader to a bean instance. |
| BeanFactoryAware | void setBeanFactory (BeanFactory beanFactory) throws BeansException; | Callback that supplies the owning factory to a bean instance. |
| BeanNameAware | void setBeanName(String name); | Set the name of the bean in the bean factory that created this bean. |
| BootstrapContextAware | void setBootstrapContext (BootstrapContext bootstrapContext); | Set the BootstrapContext that this object runs in. |
| LoadTimeWeaverAware | void setLoadTimeWeaver (LoadTimeWeaver loadTimeWeaver); | Set the LoadTimeWeaver of this object’s containing ApplicationContext. |
| MessageSourceAware | void setMessageSource (MessageSource messageSource); | Set the MessageSource that this object runs in. |
| NotificationPublisherAware | void setNotificationPublisher (NotificationPublisher notificationPublisher); | Set the NotificationPublisher instance for the current managed resource instance. |
| PortletConfigAware | void setPortletConfig (PortletConfig portletConfig); | Set the PortletConfig this object runs in. |
| PortletContextAware | void setPortletContext (PortletContext portletContext); | Set the PortletContext that this object runs in. |
| ResourceLoaderAware | void setResourceLoader (ResourceLoader resourceLoader); | Set the ResourceLoader that this object runs in. |
| ServletConfigAware | void setServletConfig (ServletConfig servletConfig); | Set the ServletConfig that this object runs in. |
| ServletContextAware | void setServletContext (ServletContext servletContext); | Set the ServletContext that this object runs in. |

Java program to show usage of aware interfaces to control string bean life cycle.

|  |
| --- |
| DemoBean.java |
| package .task;    import org.springframework.beans.BeansException;  import org.springframework.beans.factory.BeanClassLoaderAware;  import org.springframework.beans.factory.BeanFactory;  import org.springframework.beans.factory.BeanFactoryAware;  import org.springframework.beans.factory.BeanNameAware;  import org.springframework.context.ApplicationContext;  import org.springframework.context.ApplicationContextAware;  import org.springframework.context.ApplicationEventPublisher;  import org.springframework.context.ApplicationEventPublisherAware;  import org.springframework.context.MessageSource;  import org.springframework.context.MessageSourceAware;  import org.springframework.context.ResourceLoaderAware;  import org.springframework.context.weaving.LoadTimeWeaverAware;  import org.springframework.core.io.ResourceLoader;  import org.springframework.instrument.classloading.LoadTimeWeaver;  import org.springframework.jmx.export.notification.NotificationPublisher;  import org.springframework.jmx.export.notification.NotificationPublisherAware;    public class DemoBean implements ApplicationContextAware,          ApplicationEventPublisherAware, BeanClassLoaderAware, BeanFactoryAware,          BeanNameAware, LoadTimeWeaverAware, MessageSourceAware,          NotificationPublisherAware, ResourceLoaderAware  {      @Override      public void setResourceLoader(ResourceLoader arg0) {          // TODO Auto-generated method stub      }        @Override      public void setNotificationPublisher(NotificationPublisher arg0) {          // TODO Auto-generated method stub        }        @Override      public void setMessageSource(MessageSource arg0) {          // TODO Auto-generated method stub      }        @Override      public void setLoadTimeWeaver(LoadTimeWeaver arg0) {          // TODO Auto-generated method stub      }        @Override      public void setBeanName(String arg0) {          // TODO Auto-generated method stub      }        @Override      public void setBeanFactory(BeanFactory arg0) throws BeansException {          // TODO Auto-generated method stub      }        @Override      public void setBeanClassLoader(ClassLoader arg0) {          // TODO Auto-generated method stub      }        @Override      public void setApplicationEventPublisher(ApplicationEventPublisher arg0) {          // TODO Auto-generated method stub      }        @Override      public void setApplicationContext(ApplicationContext arg0)              throws BeansException {          // TODO Auto-generated method stub      }  } |

#### Custom init() and destroy() methods

The default init and destroy methods in bean configuration file can be defined in two ways:

* **Bean local definition** applicable to a single bean
* **Global definition** applicable to all beans defined in beans context

##### Bean local definition

Local definition is given as below.

|  |
| --- |
| beans.xml |
| <beans>        <bean id="demoBean" class="com.howtodoinjava.task.DemoBean"                      init-method="customInit"                      destroy-method="customDestroy"></bean>    </beans> |

##### Global definition

Where as global definition is given as below. These methods will be invoked for all bean definitions given under <beans> tag. They are useful when you have a pattern of defining common method names such as init() and destroy() for all your beans consistently. This feature helps you in not mentioning the init and destroy method names for all beans independently.

|  |
| --- |
| <beans default-init-method="customInit" default-destroy-method="customDestroy">            <bean id="demoBean" class="com.task.DemoBean"></bean>    </beans> |

Java program to show methods configured in bean XML configuration file.

|  |
| --- |
| DemoBean.java |
| package com.task;    public class DemoBean  {      public void customInit()      {          System.out.println("Method customInit() invoked...");      }        public void customDestroy()      {          System.out.println("Method customDestroy() invoked...");      }  } |

#### @PostConstruct and @PreDestroy

Spring 2.5 onwards, you can use annotations also for specifying life cycle methods using @PostConstruct and @PreDestroy annotations.

* @PostConstruct annotated method will be invoked after the bean has been constructed using default constructor and just before it’s instance is returned to requesting object.
* @PreDestroy annotated method is called just before the bean is about be destroyed inside bean container.

Java program to show usage of **annotation configuration** to control using annotations.

|  |
| --- |
| package com.task;    import javax.annotation.PostConstruct;  import javax.annotation.PreDestroy;    public class DemoBean  {      @PostConstruct      public void customInit()      {          System.out.println("Method customInit() invoked...");      }        @PreDestroy      public void customDestroy()      {          System.out.println("Method customDestroy() invoked...");      }  } Spring bean life cycle methods – BeanPostProcessor **BeanPostProcessor** is used to perform some operations before and after creating a bean,this allows us to add some code before and after creating the bean.  **BeanPostProcessor** is applicable for all the beans, which means its methods will be executed for each bean we define in the xml.  We can use the **BeanPostProcessor** to execute some logic for all the beans in the application context before and after their initialization  **BeanPostProcessor** interface has 2 methods **postProcessBeforeInitialization()** and **postProcessAfterInitialization()** where former is called after the bean is created and before it is initialized And the latter is called after the bean initialization  import org.springframework.beans.BeansException;  import org.springframework.beans.factory.config.BeanPostProcessor;  public class MyBeanPostProcessor implements BeanPostProcessor {    public Object postProcessAfterInitialization(Object bean, String beanName) throws BeansException {  System.out.println("Post Process After Initialization for the bean "+beanName);  return bean;  }  public Object postProcessBeforeInitialization(Object bean, String beanName) throws BeansException {  System.out.println("Post Process Before Initialization for the bean "+beanName);  return bean;  }  } |

# Initializing collections in spring

We know how to configure primitive data type using **value** attribute and object references using **ref** attribute of the <property> tag in your Bean configuration file. Both the cases deal with passing singular value to a bean.

Now we want to pass plural values like Java Collection types such as List, Set, Map, and Properties. To handle the situation, Spring offers four types of collection configuration elements which are as follows −

|  |  |
| --- | --- |
| **Sr.No** | **Element & Description** |
| 1 | **<list>**  This helps in wiring ie injecting a list of values, allowing duplicates. |
| 2 | **<set>**  This helps in wiring a set of values but without any duplicates. |
| 3 | **<map>**  This can be used to inject a collection of name-value pairs where name and value can be of any type. |
| 4 | **<props>**  This can be used to inject a collection of name-value pairs where the name and value are both Strings. |

We can use either <list> or <set> to wire any implementation of java.util.Collection or an **array**.

We will come across two situations (a) Passing direct values of the collection and (b) Passing a reference of a bean as one of the collection elements.

: let see how to initialize any collections in spring.

Bean Inheritance

A bean definition can contain a lot of configuration information, including constructor arguments, property values, and container-specific information such as initialization method, static factory method name, and so on.

A child bean definition inherits configuration data from a parent definition. The child definition can override some values, or add others, as needed.

Spring Bean definition inheritance has nothing to do with Java class inheritance but the inheritance concept is same. You can define a parent bean definition as a template and other child beans can inherit the required configuration from the parent bean.

When you use XML-based configuration metadata, you indicate a child bean definition by using the **parent** attribute, specifying the parent bean as the value of this attribute.

# Spring Autowiring

Autowiring is a technique used in Spring to enable automatic dependency injection. By using it Spring container can autowire relationships between collaborating beans. It is known as Spring Autowiring.

Spring provides @Autowired annotation that enables you to inject the object dependency implicitly. It internally uses setter or constructor injection.

**Note:** We can not use @Autowired annotation to inject primitive and string values. It works with reference only.

Enable AutoWiring in Spring

The Spring framework enables automatic dependency injection. In other words, by declaring all the bean dependencies in a Spring configuration file, the Spring container can autowire relationships between collaborating beans. This is called Spring bean autowiring.

In a Java-based configuration, we can enable it by using the @componentScan annotation.

**Enabling *@Autowired* Annotations**

The Spring framework enables automatic dependency injection. In other words, by declaring all the bean dependencies in a Spring configuration file, Spring container can autowire relationships between collaborating beans. This is called Spring bean autowiring.

To use Java-based configuration in our application, let's enable annotation-driven injection to load our Spring configuration:

@Configuration

@ComponentScan("com.baeldung.autowire.sample")

public class AppConfig {}

Alternatively, the [*<context:annotation-config>* annotation](https://www.baeldung.com/spring-contextannotation-contextcomponentscan#:~:text=The%20%3Ccontext%3Aannotation%2Dconfig,annotation%2Dconfig%3E%20can%20resolve.) is mainly used to activate the dependency injection annotations in Spring XML files.

## Using @Autowired

After enabling annotation injection,**we can use autowiring on properties, setters, and constructors**.

### @Autowired****on Properties****

First, let's define a fooFormatter bean:

@Component("fooFormatter")

public class FooFormatter {

public String format() {

return "foo";

}

}

Then, we'll inject this bean into the FooService bean using @Autowired on the field definition:

@Component

public class FooService {

@Autowired

private FooFormatter fooFormatter;

}

### @Autowired****on Setters****

Now let's try adding @Autowired annotation on a setter method.

In the following example, the setter method is called with the instance of FooFormatter when FooService is created:

**public** **class** **FooService** {

**private** FooFormatter fooFormatter;

@Autowired

**public** **void** **setFooFormatter**(FooFormatter fooFormatter) {

**this**.fooFormatter = fooFormatter;

}

}

### @Autowired****on Constructors****

Finally, let's use @Autowired on a constructor.

We'll see that an instance of FooFormatter is injected by Spring as an argument to the FooService constructor:

**public** **class** **FooService** {

**private** FooFormatter fooFormatter;

@Autowired

**public** **FooService**(FooFormatter fooFormatter) {

**this**.fooFormatter = fooFormatter;

}

}

## Autowire Disambiguation

By default, Spring resolves @Autowired entries by type. **If more than one bean of the same type is available in the container, the framework will throw a fatal exception**.

To resolve this conflict, we need to tell Spring explicitly which bean we want to inject.

### ****Autowiring by****@Qualifier

[@Qualifier](https://www.baeldung.com/spring-qualifier-annotation) annotation to indicate the required bean.

First, we'll define 2 beans of type Formatter:

@Component("fooFormatter")

**public** **class** **FooFormatter** **implements** **Formatter** {

**public** String **format**() {

**return** "foo";

}

}

@Component("barFormatter")

**public** **class** **BarFormatter** **implements** **Formatter** {

**public** String **format**() {

**return** "bar";

}

}

Now let's try to inject a Formatter bean into the FooService class:

**public** **class** **FooService** {

@Autowired

**private** Formatter formatter;

}

In our example, there are two concrete implementations of Formatter available for the Spring container. As a result, **Spring will throw a NoUniqueBeanDefinitionException exception when constructing the FooService**:

Caused by: org.springframework.beans.factory.NoUniqueBeanDefinitionException:

No qualifying bean of type [com.autowire.sample.Formatter] is defined:

expected single matching bean but found 2: barFormatter,fooFormatter

**We can avoid this by narrowing the implementation using a @Qualifier annotation:**

**public** **class** **FooService** {

@Autowired

@Qualifier("fooFormatter")

**private** Formatter formatter;

}

When there are multiple beans of the same type, it's a good idea to**use @Qualifier to avoid ambiguity.**

Please note that the value of the @Qualifier annotation matches with the name declared in the @Component annotation of our FooFormatter implementation.

.

### ****Autowiring by Name****

**Spring uses the bean's name as a default qualifier value.**It will inspect the container and look for a bean with the exact name as the property to autowire it.

Hence, in our example, Spring matches the fooFormatter property name to the FooFormatter implementation. Therefore, it injects that specific implementation when constructing FooService:

**public** **class** **FooService** {

@Autowired

**private** Formatter fooFormatter;

}